FFT MegaCore Function

User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

UG-FFT-14.1

Feedback Subscribe

Contents

1. About This IP Core

Release Information

Table 1–1 lists information about this release of the Altera® FFT IP Core.

Table 1–1. FFT IP Core Release Information

Item	Description
Version	14.0 Arria 10 Edition
Release Date	August 2014
Ordering Code	IP-FFT
Product ID	0034
Vendor ID	6AF7

For more information about this release, refer to the *MegaCore IP Library Release Notes*.

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* report any exceptions to this verification. Altera does not verify compilation with MegaCore function versions older than one release.

Device Family Support

Altera offers the following device support levels for Altera IP cores:

- Preliminary support—Altera verifies the IP core with preliminary timing models for this device family. The IP core meets all functional requirements, but might still be undergoing timing analysis for the device family. You can use it in production designs with caution.
- Final support—Altera verifies the IP core with final timing models for this device family. The IP core meets all functional and timing requirements for the device family and you can use it in production designs.

Table 1–2 lists 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® II GX	Final
Arria II GZ	Final
Arria V	Final
Arria V GZ	Final
Arria 10	Preliminary
Cyclone IV	Final
Cyclone V	Final

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

Device Family	Support
MAX 10 FPGAs	Preliminary
Stratix IV GT	Final
Stratix IV GX/E	Final
Stratix V	Final
Stratix GX	Final

Features

- Bit-accurate MATLAB models
- Variable streaming FFT:
  - Single-precision floating-point or fixed-point representation
  - Radix-4, mixed radix-4/2 implementations (for floating-point FFT), and radix-2^2 single delay feedback implementation (for fixed-point FFT)
  - Input and output orders: natural order, bit-reversed or digit-reversed, and DC-centered (–N/2 to N/2)
  - Reduced memory requirements
  - Support for 8 to 32-bit data and twiddle width (foxed-point FFTs)
- Fixed transform size FFT that implements block floating-point FFTs—maintains the maximum dynamic range of data during processing (not for variable streaming FFTs)
  - Multiple I/O data flow options: streaming, buffered burst, and burst
  - Uses embedded memory
  - Maximum system clock frequency more than 300 MHz
  - Optimized to use Stratix series DSP blocks and TriMatrix™ memory
  - High throughput quad-output radix 4 FFT engine
  - Support for multiple single-output and quad-output engines in parallel
- User control over optimization in DSP blocks or in speed in Stratix V devices, for streaming, buffered burst, burst, and variable streaming fixed-point FFTs
- Avalon® Streaming (Avalon-ST) compliant input and output interfaces
- Parameterization-specific VHDL and Verilog HDL testbench generation
- Transform direction (FFT/IFFT) specifiable on a per-block basis

For more information about 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:

- Fixed transform size FFT
- Variable streaming FFT

Fixed Transform Size FFT

The fixed transform FFT implements a radix-2/4 decimation-in-frequency (DIF) FFT fixed-transform size algorithm for transform lengths of 2^m where $6 \leq m \leq 16$. This FFT 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 FFT accepts a two's complement format complex data vector of length N inputs, where N is the desired transform length in natural order. The function outputs the transform-domain complex vector in natural order. The FFT produces an accumulated block exponent to indicate any data scaling that has occurred during the transform to maintain precision and maximize the internal signal-to-noise ratio. You can specify the transform direction on a per-block basis using an input port.

Variable Streaming FFT

The variable streaming FFT implements two different types of FFT. The variable streaming FFTs implement either a radix-2^2 single delay feedback FFT, using a fixed-point representation, or a mixed radix-4/2 FFT, using a single precision floating point representation. After you select your FFT type, you can configure your FFT variation during runtime to perform the FFT algorithm for transform lengths of 2^m where $3 \leq m \leq 18$.

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 allows a large dynamic range of values to be represented while maintaining a high SNR at the output.

The order of the input data vector of size N can be natural, bit- or digit-reversed, or $–N/2$ to $N/2$ (DC-centered). The fixed-point representation supports a natural, bit-reversed, or DC-centered order and the floating point representation supports a natural, digit-reversed order. The FFT outputs the transform-domain complex vector in natural, bit-reversed, or digit-reversed order. You can specify the transform direction on a per-block basis using an input port.

MegaCore Verification

Before releasing a version of the FFT MegaCore function, Altera runs comprehensive regression tests to verify its quality and correctness.

Altera generates custom variations of the FFT MegaCore function to test its various parameter options. Altera simulates the resulting simulation models and verifies the results against master simulation models.

Performance and Resource Utilization

Table 1–3 shows typical expected performance for a FFT IP Core using the Quartus II software with the Arria V (5AGXFB3H4F40C4), Cyclone V (5CGXFC7D6F31C6), and Stratix V (5SGSMD4H2F35C2) devices:

Table 1–3. FFT IP Core Performance

Device	Parameters			ALM	DSP Blocks	Memory		Registers		f_{MAX} (MHz)
	Type	Length	Engines			M10K	M20K	Primary	Secondary	
Arria V	Buffered Burst	1,024	1	1,572	6	16	--	3,903	143	275
Arria V	Buffered Burst	1,024	2	2,512	12	30	--	6,027	272	274
Arria V	Buffered Burst	1,024	4	4,485	24	59	--	10,765	426	262
Arria V	Buffered Burst	256	1	1,532	6	16	--	3,713	136	275
Arria V	Buffered Burst	256	2	2,459	12	30	--	5,829	246	245
Arria V	Buffered Burst	256	4	4,405	24	59	--	10,539	389	260
Arria V	Buffered Burst	4,096	1	1,627	6	59	--	4,085	130	275
Arria V	Buffered Burst	4,096	2	2,555	12	59	--	6,244	252	275
Arria V	Buffered Burst	4,096	4	4,526	24	59	--	10,986	438	265
Arria V	Burst Quad Output	1,024	1	1,565	6	8	--	3,807	147	273
Arria V	Burst Quad Output	1,024	2	2,497	12	14	--	5,952	225	275
Arria V	Burst Quad Output	1,024	4	4,461	24	27	--	10,677	347	257
Arria V	Burst Quad Output	256	1	1,527	6	8	--	3,610	153	272
Arria V	Burst Quad Output	256	2	2,474	12	14	--	5,768	233	275
Arria V	Burst Quad Output	256	4	4,403	24	27	--	10,443	437	257
Arria V	Burst Quad Output	4,096	1	1,597	6	27	--	3,949	151	275

Table 1–3. FFT IP Core Performance

Device	Parameters			ALM	DSP Blocks	Memory		Registers		f_{MAX} (MHz)
	Type	Length	Engines			M10K	M20K	Primary	Secondary	
Arria V	Burst Quad Output	4,096	2	2,551	12	27	--	6,119	223	275
Arria V	Burst Quad Output	4,096	4	4,494	24	27	--	10,844	392	256
Arria V	Burst Single Output	1,024	1	672	2	6	--	1,488	101	275
Arria V	Burst Single Output	1,024	2	994	4	10	--	2,433	182	275
Arria V	Burst Single Output	256	1	636	2	3	--	1,442	95	275
Arria V	Burst Single Output	256	2	969	4	8	--	2,375	152	275
Arria V	Burst Single Output	4,096	1	702	2	19	--	1,522	126	270
Arria V	Burst Single Output	4,096	2	1,001	4	25	--	2,521	156	275
Arria V	Streaming	1,024	—	1,880	6	20	--	4,565	167	275
Arria V	Streaming	256	—	1,647	6	20	--	3,838	137	275
Arria V	Streaming	4,096	—	1,819	6	71	--	4,655	137	275
Arria V	Variable Streaming Floating Point	1,024	—	11,195	48	89	--	18,843	748	163
Arria V	Variable Streaming Floating Point	256	—	8,639	36	62	--	15,127	609	161
Arria V	Variable Streaming Floating Point	4,096	—	13,947	60	138	--	22,598	854	162
Arria V	Variable Streaming	1,024	—	2,535	11	14	--	6,269	179	223
Arria V	Variable Streaming	256	—	1,913	8	8	--	4,798	148	229
Arria V	Variable Streaming	4,096	—	3,232	15	31	--	7,762	285	210
Cyclone V	Buffered Burst	1,024	1	1,599	6	16	--	3,912	114	226

Table 1–3. FFT IP Core Performance

Device	Parameters			ALM	DSP Blocks	Memory		Registers		f_{MAX} (MHz)
	Type	Length	Engines			M10K	M20K	Primary	Secondary	
Cyclone V	Buffered Burst	1,024	2	2,506	12	30	--	6,078	199	219
Cyclone V	Buffered Burst	1,024	4	4,505	24	59	--	10,700	421	207
Cyclone V	Buffered Burst	256	1	1,528	6	16	--	3,713	115	227
Cyclone V	Buffered Burst	256	2	2,452	12	30	--	5,833	211	232
Cyclone V	Buffered Burst	256	4	4,487	24	59	--	10,483	424	221
Cyclone V	Buffered Burst	4,096	1	1,649	6	59	--	4,060	138	223
Cyclone V	Buffered Burst	4,096	2	2,555	12	59	--	6,254	199	227
Cyclone V	Buffered Burst	4,096	4	4,576	24	59	--	10,980	377	214
Cyclone V	Burst Quad Output	1,024	1	1,562	6	8	--	3,810	122	225
Cyclone V	Burst Quad Output	1,024	2	2,501	12	14	--	5,972	196	231
Cyclone V	Burst Quad Output	1,024	4	4,480	24	27	--	10,643	372	216
Cyclone V	Burst Quad Output	256	1	1,534	6	8	--	3,617	120	226
Cyclone V	Burst Quad Output	256	2	2,444	12	14	--	5,793	153	224
Cyclone V	Burst Quad Output	256	4	4,443	24	27	--	10,402	379	223
Cyclone V	Burst Quad Output	4,096	1	1,590	6	27	--	3,968	120	237
Cyclone V	Burst Quad Output	4,096	2	2,547	12	27	--	6,135	209	227
Cyclone V	Burst Quad Output	4,096	4	4,512	24	27	--	10,798	388	210
Cyclone V	Burst Single Output	1,024	1	673	2	6	--	1,508	83	222
Cyclone V	Burst Single Output	1,024	2	984	4	10	--	2,475	126	231
Cyclone V	Burst Single Output	256	1	639	2	3	--	1,382	159	229

Table 1–3. FFT IP Core Performance

Device	Parameters			ALM	DSP Blocks	Memory		Registers		f_{MAX} (MHz)
	Type	Length	Engines			M10K	M20K	Primary	Secondary	
Cyclone V	Burst Single Output	256	2	967	4	8	--	2,353	169	240
Cyclone V	Burst Single Output	4,096	1	695	2	19	--	1,540	105	237
Cyclone V	Burst Single Output	4,096	2	1,009	4	25	--	2,536	116	240
Cyclone V	Streaming	1,024	—	1,869	6	20	--	4,573	132	211
Cyclone V	Streaming	256	—	1,651	6	20	--	3,878	85	226
Cyclone V	Streaming	4,096	—	1,822	6	71	--	4,673	124	199
Cyclone V	Variable Streaming Floating Point	1,024	—	11,184	48	89	--	18,830	628	133
Cyclone V	Variable Streaming Floating Point	256	—	8,611	36	62	--	15,156	467	133
Cyclone V	Variable Streaming Floating Point	4,096	—	13,945	60	138	--	22,615	701	132
Cyclone V	Variable Streaming	1,024	—	2,533	11	14	--	6,254	240	179
Cyclone V	Variable Streaming	256	—	1,911	8	8	--	4,786	176	180
Cyclone V	Variable Streaming	4,096	—	3,226	15	31	--	7,761	320	176
Stratix V	Buffered Burst	1,024	1	1,610	6	--	16	4,141	107	424
Stratix V	Buffered Burst	1,024	2	2,545	12	--	30	6,517	170	427
Stratix V	Buffered Burst	1,024	4	4,554	24	--	59	11,687	250	366
Stratix V	Buffered Burst	256	1	1,546	6	--	16	3,959	110	493
Stratix V	Buffered Burst	256	2	2,475	12	--	30	6,314	134	440
Stratix V	Buffered Burst	256	4	4,480	24	--	59	11,477	281	383
Stratix V	Buffered Burst	4,096	1	1,668	6	--	30	4,312	122	432

Table 1–3. FFT IP Core Performance

Device	Parameters			ALM	DSP Blocks	Memory		Registers		f_{MAX} (MHz)
	Type	Length	Engines			M10K	M20K	Primary	Secondary	
Stratix V	Buffered Burst	4,096	2	2,602	12	--	30	6,718	176	416
Stratix V	Buffered Burst	4,096	4	4,623	24	--	59	11,876	249	392
Stratix V	Burst Quad Output	1,024	1	1,550	6	--	8	4,037	115	455
Stratix V	Burst Quad Output	1,024	2	2,444	12	--	14	6,417	164	433
Stratix V	Burst Quad Output	1,024	4	4,397	24	--	27	11,548	330	416
Stratix V	Burst Quad Output	256	1	1,487	6	--	8	3,868	83	477
Stratix V	Burst Quad Output	256	2	2,387	12	--	14	6,211	164	458
Stratix V	Burst Quad Output	256	4	4,338	24	--	27	11,360	307	409
Stratix V	Burst Quad Output	4,096	1	1,593	6	--	14	4,222	93	448
Stratix V	Burst Quad Output	4,096	2	2,512	12	--	14	6,588	154	470
Stratix V	Burst Quad Output	4,096	4	4,468	24	--	27	11,773	267	403
Stratix V	Burst Single Output	1,024	1	652	2	--	4	1,553	111	500
Stratix V	Burst Single Output	1,024	2	1,011	4	--	8	2,687	149	476
Stratix V	Burst Single Output	256	1	621	2	--	3	1,502	132	500
Stratix V	Burst Single Output	256	2	978	4	--	8	2,555	173	500
Stratix V	Burst Single Output	4,096	1	681	2	--	9	1,589	149	500
Stratix V	Burst Single Output	4,096	2	1,039	4	--	14	2,755	161	476
Stratix V	Streaming	1,024	—	1,896	6	--	20	4,814	144	490
Stratix V	Streaming	256	—	1,604	6	--	20	4,062	99	449
Stratix V	Streaming	4,096	—	1,866	6	--	38	4,889	118	461

Table 1–3. FFT IP Core Performance

Device	Parameters			ALM	DSP Blocks	Memory		Registers		f_{MAX} (MHz)
	Type	Length	Engines			M10K	M20K	Primary	Secondary	
Stratix V	Variable Streaming Floating Point	1,024	—	11,607	32	--	87	19,031	974	355
Stratix V	Variable Streaming Floating Point	256	—	8,850	24	--	59	15,297	820	374
Stratix V	Variable Streaming Floating Point	4,096	—	14,335	40	--	115	22,839	1,047	325
Stratix V	Variable Streaming	1,024	—	2,334	14	--	13	5,623	201	382
Stratix V	Variable Streaming	256	—	1,801	10	--	8	4,443	174	365
Stratix V	Variable Streaming	4,096	—	2,924	18	--	23	6,818	238	355

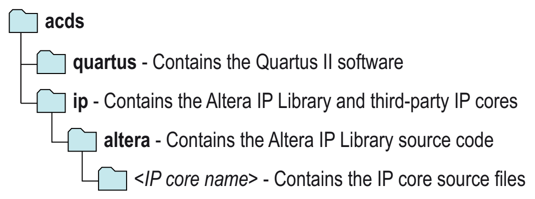
2. Getting Started

Installing and Licensing IP Cores

The Quartus II software includes the Altera IP Library. The library provides many useful IP core functions for production use without additional license. You can fully evaluate any licensed Altera IP core in simulation and in hardware until you are satisfied with its functionality and performance.

Some Altera IP cores, such as MegaCore® functions, require that you purchase a separate license for production use. After you purchase a license, visit the Self Service Licensing Center to obtain a license number for any Altera product. For additional information, refer to *Altera Software Installation and Licensing*.

Figure 2–1. IP core Installation Path



acds
quartus - Contains the Quartus II software
ip - Contains the Altera IP Library and third-party IP cores
altera - Contains the Altera IP Library source code
<IP core name> - Contains the IP core source files

The default installation directory on Windows is **<*drive*>:\altera\<*version number*>**; on Linux it is **<*home directory*>/altera/<*version number*>**.

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^{SM} 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 FFT MegaCore function when you are completely satisfied with its functionality and performance, and want to take your design to production. After you purchase a license, you can request a license file from the Altera website at **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.

For more information about OpenCore Plus hardware evaluation, refer to *AN 320: OpenCore Plus Evaluation of Megafunctions*.

OpenCore Plus Time-Out Behavior

OpenCore Plus hardware evaluation supports the following operation modes:

- *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 might be masked by the time-out behavior of the other megafunctions.

The untethered time-out for the FFT MegaCore function 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.

Customizing and Generating IP Cores

You can customize IP cores to support a wide variety of applications. The Quartus II IP Catalog displays IP cores available for the current target device. The parameter editor guides you to set parameter values for optional ports, features, and output files.

To customize and generate a custom IP core variation, follow these steps:

1. In the IP Catalog (**Tools > IP Catalog**), locate and double-click the name of the IP core to customize. The parameter editor appears.
2. Specify a top-level name for your custom IP variation. This name identifies the IP core variation files in your project. If prompted, also specify the target Altera device family and output file HDL preference. Click **OK**.
3. Specify the desired parameters, output, and options for your IP core variation:
   - Optionally select preset parameter values. Presets specify all initial parameter values for specific applications (where provided).
   - Specify parameters defining the IP core functionality, port configuration, and device-specific features.
   - Specify options for generation of a timing netlist, simulation model, testbench, or example design (where applicable).
   - Specify options for processing the IP core files in other EDA tools.
4. Click **Finish** or **Generate** to generate synthesis and other optional files matching your IP variation specifications. The parameter editor generates the top-level **.qip** or **.qsys** IP variation file and HDL files for synthesis and simulation. Some IP cores also simultaneously generate a testbench or example design for hardware testing.
5. To generate a simulation testbench, click **Generate > Generate Testbench System**. **Generate > Generate Testbench System** is not available for some IP cores.
6. To generate a top-level HDL design example for hardware verification, click **Generate > HDL Example**. **Generate > HDL Example** is not available for some IP cores.

When you generate the IP variation with a Quartus II project open, the parameter editor automatically adds the IP variation to the project. Alternatively, click **Project > Add/Remove Files in Project** to manually add a top-level **.qip** or **.qsys** IP variation file to a Quartus II project. To fully integrate the IP into the design, make appropriate pin assignments to connect ports. You can define a virtual pin to avoid making specific pin assignments to top-level signals.

Generated Files

Table 2–1 describes the files created in the project directory during generation of the MegaCore function. The design synthesis and simulation files are generated in the following two directories:

- <*variation name*> directory that ontains files for Quartus II synthesis
- <*variation name*>_sim directory that contains files for simulation purposes

The names and types of files specified in the report vary based on whether you selected the VHDL or Verilog HDL output format.

Table 2–1. Generated Files (Part 1 of 2) (1) & (2)

Filename	Description
<*variation name*>**_imag_input.txt**	The text file contains input imaginary component random data. This file is read by the generated VHDL or Verilog HDL MATLAB testbenches.
<*variation name*>_**real_input.txt**	Test file containing real component random data. This file is read by the generated VHDL or Verilog HDL and MATLAB testbenches.
<*variation name*>**.bsf**	Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<*variation name***>.cmp**	A VHDL component declaration file for the MegaCore function variation. Add the contents of this file to any VHDL architecture that instantiates the MegaCore function.
<*variation name***>.html**	A MegaCore function report file in hypertext markup language format.
<*variation name***>.qip**	A single Quartus II IP file is generated that contains all of the assignments and other information required to process your MegaCore function variation in the Quartus II compiler. You are prompted to add this file to the current Quartus II project when you exit from the MegaWizard.
<*variation name*>**.vo or .vho**	VHDL or Verilog HDL IP functional simulation model.
<*variation name*>**.vhd**, or **.v**	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.
<*variation name*>**_1n1024cos.hex**, <*variation name*>**_2n1024cos.hex**, <*variation name*>**_3n1024cos.hex**	Intel hex-format ROM initialization files (not generated for variable streaming FFT).
<*variation name*>**_1n1024sin.hex**, <*variation name*>**_2n1024sin.hex**, <*variation name*>**_3n1024sin.hex**	Intel hex-format ROM initialization files (not generated for variable streaming FFT).
<*variation name*>**_model.m**	MATLAB m-file describing a MATLAB bit-accurate model.

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

Filename	Description
<*variation name*>**_tb.m**	MATLAB testbench.
<*variation name*>**_syn.v** or <*variation name*>**_syn.vhd**	A timing and resource netlist for use in some third-party synthesis tools.
<*variation name*>**_tb.v** or <*variation name*>**_tb.vhd**	Verilog HDL or VHDL testbench file.
<*variation name*>**_nativelink.tcl**	Tcl Script that sets up NativeLink in the Quartus II software to natively simulate the design using selected EDA tools. Refer to "Simulating in Third-Party Simulation Tools Using NativeLink" on page 2–8.
<*variation name*>**_twr1_opt.hex**, <*variation name*>**_twi1_opt.hex**, <*variation name*>**_twr2_opt.hex**, <*variation name*>**_twi2_opt.hex**, <*variation name*>**_twr3_opt.hex**, <*variation name*>**_twi3_opt.hex**, <*variation name*>**_twr4_opt.hex**, <*variation name*>**_twi4_opt.hex**,	Intel hex-format ROM initialization files (variable streaming FFT only).

Notes to Table 2–1:

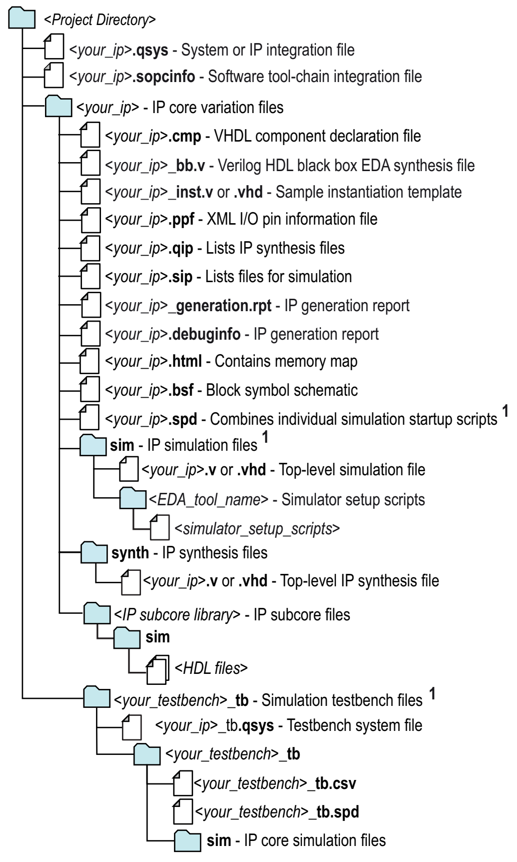
(1) These files are variation dependent, some may be absent or their names may change.

(2) <*variation name*> is a prefix variation name supplied automatically by IP Toolbench.

Files Generated for Altera IP Cores

The Quartus II software version 14.0 Arria 10 Edition and later generates the following output file structure for Altera IP cores:

Figure 2–2. IP Core Generated Files


<Project Directory>
<your_ip>.qsys - System or IP integration file
<your_ip>.sopcinfo - Software tool-chain integration file
<your_ip> - IP core variation files
<your_ip>.cmp - VHDL component declaration file
<your_ip>_bb.v - Verilog HDL black box EDA synthesis file
<your_ip>_inst.v or .vhd - Sample instantiation template
<your_ip>.ppf - XML I/O pin information file
<your_ip>.qip - Lists IP synthesis files
<your_ip>.sip - Lists files for simulation
<your_ip>_generation.rpt - IP generation report
<your_ip>.debuginfo - IP generation report
<your_ip>.html - Contains memory map
<your_ip>.bsf - Block symbol schematic
<your_ip>.spd - Combines individual simulation startup scripts 1
sim - IP simulation files 1
<your_ip>.v or .vhd - Top-level simulation file
<EDA_tool_name> - Simulator setup scripts
<simulator_setup_scripts>
synth - IP synthesis files
<your_ip>.v or .vhd - Top-level IP synthesis file
<IP subcore library> - IP subcore files
sim
<HDL files>
<your_testbench>_tb - Simulation testbench files 1
<your_ip>_tb.qsys - Testbench system file
<your_testbench>_tb
<your_testbench>_tb.csv
<your_testbench>_tb.spd
sim - IP core simulation files

1. If supported and enabled for your IP variation

Simulating IP Cores

The Quartus II software supports RTL- and gate-level design simulation of Altera IP cores in supported EDA simulators. Simulation involves setting up your simulator working environment, compiling simulation model libraries, and running your simulation.

You can use the functional simulation model and the testbench or example design generated with your IP core for simulation. The functional simulation model and testbench files are generated in a project subdirectory. This directory may also include scripts to compile and run the testbench. For a complete list of models or libraries required to simulate your IP core, refer to the scripts generated with the testbench. You can use the Quartus II NativeLink feature to automatically generate simulation files and scripts. NativeLink launches your preferred simulator from within the Quartus II software.

For more information about simulating Altera IP cores, refer to *Simulating Altera Designs* in volume 3 of the *Quartus II Handbook*.

Simulating the Design

- Simulating in the MATLAB Software
- Simulating with IP Functional Simulation Models
- Simulating in Third-Party Simulation Tools Using NativeLink

Simulating in the MATLAB Software

Fixed Transform FFT

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 about exponent values, refer to *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.

Modeling Fixed Transform FFT in MATLAB

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 about MATLAB and Simulink, refer to the MathWorks web site at www.mathworks.com.

Variable Streaming FFT

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.

Modeling Variable Streaming FFTs in MATLAB

To model your variable streaming FFT MegaCore function variation in the MATLAB software:

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 select 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)));
```

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

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

where `digit_reverse` is:

```
function y = digit_reverse(x, n_bits)
if mod(n_bits,2)
    z = dec2bin(x, n_bits);
    for i=1:2:n_bits-1
        p(:,i) = z(:,n_bits-i);
        p(:,i+1) = z(:,n_bits-i+1);
    end

    p(:,n_bits) = z(:,1);
    y=bin2dec(p);
else
    y=digitrevorder(x,4);
end
```

Simulating 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 "T** Setting 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 about IP functional simulation models, refer to the *Simulating Altera Designs* 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 about NativeLink, refer to the *Simulating Altera Designs* 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**.

Compiling the Design

Use the Quartus II software to compile your design. After a successful compilation, you can program the targeted Altera device and verify the design in hardware.

For more information about compiling your design, refer to the Quartus II Help.

Compiling Fixed Transform FFTs

To compile your fixed-transform FFT designs:

1. If you are using the Quartus II software to synthesize your design, skip to step 2. 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.

The **.qip** file supersedes the files you had to add to the project explicitly in previous versions of the Quartus II software. The **.qip** file contains the information about the MegaCore function that the Quartus II software requires.

2. On the Processing menu, click **Start Compilation**.

Compiling Variable Streaming FFT

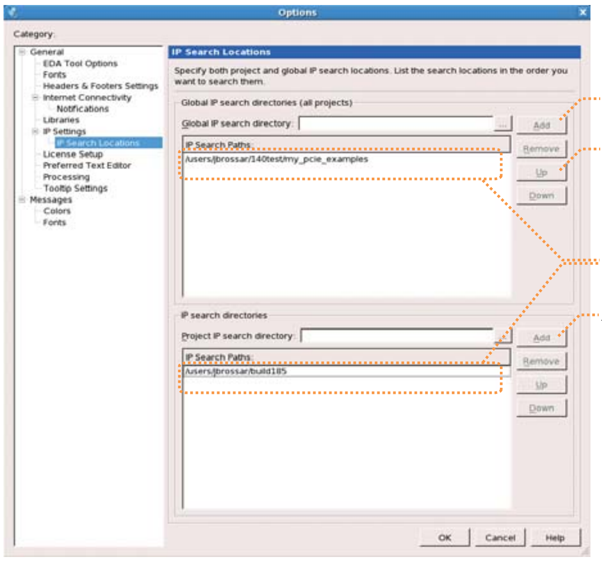
To compile your variable streaming FFT design:

1. If you are using the Quartus II software to synthesize your design, skip to step 2. 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. On the Project menu, click **Add/Remove Files in Project**.
3. You can see a list of files in the project. If no files are listed, browse to the **\lib** directory, then select and add all files with the prefix **auk_dspip_r22sdf**. Browse to the <*project*> directory and select all files with prefix **auk_dspip**.
4. On the Processing menu, click **Start Compilation**.

Including Other IP Libraries and Files

The Quartus II software searches for IP cores in the project directory, in the Altera installation directory, and in the defined IP search path. You can include IP libraries and files from other locations by modifying the IP search path. To use the GUI to modify the global or project-specific search path, click **Tools > Options > IP Search Locations** and specify the path to your IP.

Figure 2–3. Specifying IP Search Locations



Specify both project and global IP search locations. List the search locations in the order you want to search them.

/users/jbrossar/140test/my_pcie_examples

/users/jbrossar/build185

As an alternative to the GUI, use the following `SEARCH_PATH` assignment to include one or more project libraries. Specify only one source directory for each `SEARCH_PATH` assignment.

`set_global_assignment -name SEARCH_PATH` <*library or file path*>

If your project includes two IP core files of the same name, the following search path precedence rules determine the resolution of files:

1. Project directory files.
2. Project database directory files.
3. Project libraries specified in **IP Search Locations**, or with the `SEARCH_PATH` assignment in the Quartus II Settings File (**.qsf**).
4. Global libraries specified in **IP Search Locations**, or with the `SEARCH_PATH` assignment in the Quartus II Settings File (**.qsf**).
5. Quartus II software libraries directory, such as <*Quartus II Installation*>**\libraries**.

Upgrading Outdated IP Cores

IP cores generated with a previous version of the Quartus II software may require upgrade before use in the current version of the Quartus II software. Click **Project > Upgrade IP Components** to identify and upgrade outdated IP cores.

The **Upgrade IP Components** dialog box provides instructions when IP upgrade is required, optional, or unsupported for specific IP cores in your design. Most Altera IP cores support one-click, automatic simultaneous upgrade. You can individually migrate IP cores unsupported by auto-upgrade.

The **Upgrade IP Components** dialog box also reports legacy Altera IP cores that support compilation-only (without modification), as well as IP cores that do not support migration. Replace unsupported IP cores in your project with an equivalent Altera IP core or design logic.Upgrading IP cores changes your original design files.

Before you begin

- Migrate your Quartus II project containing outdated IP cores to the latest version of the Quartus II software. In a previous version of the Quartus II software, click **Project > Archive Project** to save the project. This archive preserves your original design source and project files after migration. le paths in the archive must be relative to the project directory. File paths in the archive must reference the IP variation **.v** or **.vhd** file or **.qsys** file, not the **.qip** file.
- Restore the project in the latest version of the Quartus II software. Click **Project > Restore Archived Project**. Click **Ok** if prompted to change to a supported device or overwrite the project database.

To upgrade outdated IP cores, follow these steps:

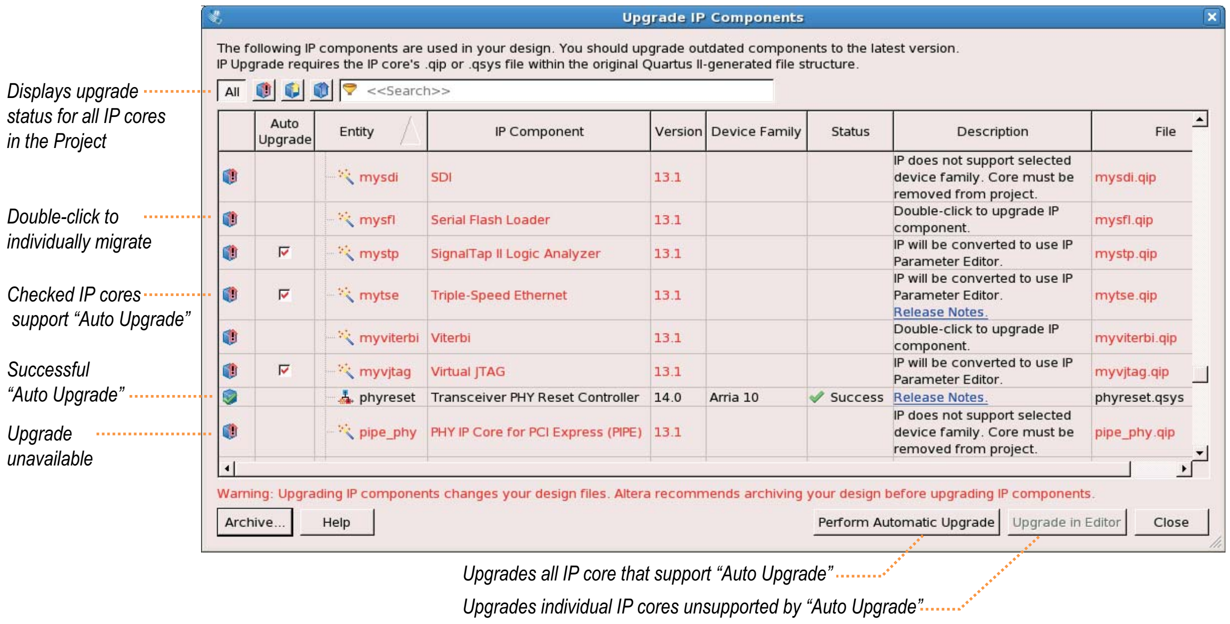
1. In the latest version of the Quartus II software, open the Quartus II project containing an outdated IP core variation.

   ☞ File paths in a restored project archive must be relative to the project directory and you must reference the IP variation **.v** or .**vhd** file or **.qsys** file, not the **.qip** file.

2. Click **Project > Upgrade IP Components**. The **Upgrade IP Components** dialog box displays all outdated IP cores in your project, along with basic instructions for upgrading each core.

3. To simultaneously upgrade all IP cores that support automatic upgrade, click **Perform Automatic Upgrade**. The IP cores upgrade to the latest version. The **Status** and **Version** columns reflect the update.

Figure 2–4. Upgrading IP Cores


Displays upgrade status for all IP cores in the Project
Double-click to individually migrate
Checked IP cores support "Auto Upgrade"
Successful "Auto Upgrade"
Upgrade unavailable
Upgrades all IP core that support "Auto Upgrade"
Upgrades individual IP cores unsupported by "Auto Upgrade"
Upgrade IP Components
All
<<Search>>
Archive...
Help
Perform Automatic Upgrade
Upgrade in Editor
Close

The following IP components are used in your design. You should upgrade outdated components to the latest version.
IP Upgrade requires the IP core's .qip or .qsys file within the original Quartus II-generated file structure.

	Auto Upgrade	Entity	IP Component	Version	Device Family	Status	Description	File
		mysdi	SDI	13.1			IP does not support selected device family. Core must be removed from project.	mysdi.qip
		mysfl	Serial Flash Loader	13.1			Double-click to upgrade IP component.	mysfl.qip
	✓	mystp	SignalTap II Logic Analyzer	13.1			IP will be converted to use IP Parameter Editor.	mystp.qip
	✓	mytse	Triple-Speed Ethernet	13.1			IP will be converted to use IP Parameter Editor. Release Notes.	mytse.qip
		myviterbi	Viterbi	13.1			Double-click to upgrade IP component.	myviterbi.qip
	✓	myvjtag	Virtual JTAG	13.1			IP will be converted to use IP Parameter Editor.	myvjtag.qip
		phyreset	Transceiver PHY Reset Controller	14.0	Arria 10	✓ Success	Release Notes.	phyreset.qsys
		pipe_phy	PHY IP Core for PCI Express (PIPE)	13.1			IP does not support selected device family. Core must be removed from project.	pipe_phy.qip

Warning: Upgrading IP components changes your design files. Altera recommends archiving your design before upgrading IP components.

Upgrading IP Cores at the Command Line

Alternatively, you can upgrade IP cores at the command line. To upgrade a single IP core, type the following command:

```
quartus_sh --ip_upgrade -variation_files <my_ip_path> <project>
```

To upgrade a list of IP cores, type the following command:

```
quartus_sh --ip_upgrade -variation_files
"<my_ip>.qsys;<my_ip>.<hdl>; <project>"
```

IP cores older than Quartus II software version 12.0 do not support upgrade. Altera verifies that the current version of the Quartus II software compiles the previous version of each IP core. The *MegaCore IP Library Release Notes* reports any verification exceptions for MegaCore IP. The *Quartus II Software and Device Support Release Notes* reports any verification exceptions for other IP cores. Altera does not verify compilation for IP cores older than the previous two releases.

DSP Builder Design Flow

DSP Builder shortens digital signal processing (DSP) design cycles by helping you create the hardware representation of a DSP design in an algorithm-friendly development environment.

This IP core supports DSP Builder. Use the DSP Builder flow if you want to create a DSP Builder model that includes an IP core variation; use IP Catalog if you want to create an IP core variation that you can instantiate manually in your design.

For more information about the DSP Builder flow, refer to the *Using MegaCore Functions* chapter in the *DSP Builder Handbook*.

3. Functional Description

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.

The following equation 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, ... $N - 1$

The following equation 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, ... $N - 1$

You can reduce the complexity of the DFT direct computation 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. You must perform each pass through the FFT processor $\log_r N$ 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.

Fixed Transform FFTs

The buffered, burst, and streaming FFTs use a radix-4 decomposition, which divides the input sequence recursively to form four-point sequences, requires only trivial multiplications in the four-point DFT. Radix-4 gives 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 fixed transform FFTs use a block-floating-point architecture, which is a trade-off point between fixed-point and full-floating-point architectures.

(Refer to "Block Floating Point Scaling" on page A–1.)

For more information about block floating-point FFTs, refer to "Block Floating Point Scaling" on page A–1.

Variable Streaming FFTs

The variable streaming FFTs use fixed-point data representation or the floating point representation. If you select the fixed-point data representation, the FFT variation uses a radix 2^2 single delay feedback, which is fully pipelined. If you select the floating point representation, the FFT variation uses a mixed radix-4/2. For a length N transform, $\log_4(N)$ stages are concatenated together. The radix 2^2 algorithm has the same multiplicative complexity of a fully pipelined radix-4 FFT, but the butterfly unit retains a radix-2 FFT. The radix-4/2 algorithm combines radix-4 and radix-2 FFTs to achieve the computational advantage of the radix-4 algorithm while supporting FFT computation with a wider range of transform lengths. The butterfly units use the DIF decomposition.

Fixed point representation allows for natural word growth through the pipeline. The maximum growth of each stage is 2 bits. After the complex multiplication the data is rounded down to the expanded data size using convergent rounding. The overall bit growth is less than or equal to $\log_2(N)+1$.

The floating point internal data representation is single-precision floating-point (32-bit, IEEE 754 representation). Floating-point operations provide more precise computation results but are costly in 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.

Fixed-Point Variable Streaming FFTs

Fixed point variable streaming FFTs implements a radix-2^2 single delay feedback. It is similar to radix-2 single delay feedback. However, the twiddle factors are rearranged such that the multiplicative complexity is equivalent to a radix-4 single delay feedback.

$\text{Log}_2(N)$ stages each 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 occur in the pipeline for an FFT of length $N = 16$.

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 feed 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.

Floating-Point Variable Streaming FFTs

floatin-point variable streaming FFTs implments a mixed radix-4/2, which combines the advantages of using radix-2 and radix-4 butterflies.

The FFT has `ceiling(log`$_4$`(N))` stages. If transform length is an integral power of four, a radix-4 FFT implements all of the `log`$_4$`(N)` stages. If transform length is not an integral power of four, the FFT implements `ceiling(log`$_4$`(N))` – `1` of the stages in a radix-4, and implements the remaining stage using a radix-2.

Each stage contains a single butterfly unit and a feedback delay unit. The feedback delay unit delays the incoming data by a specified number of cycles; in each stage the number of cycles of delay is one quarter of the number of cycles of delay in the previous stage. The delays align the butterfly input samples correctly for the butterfly calculations. The output of the pipeline is in index-reversed order.

Input and Output Orders

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-centered	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-centered	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.

FFT Processor Engines

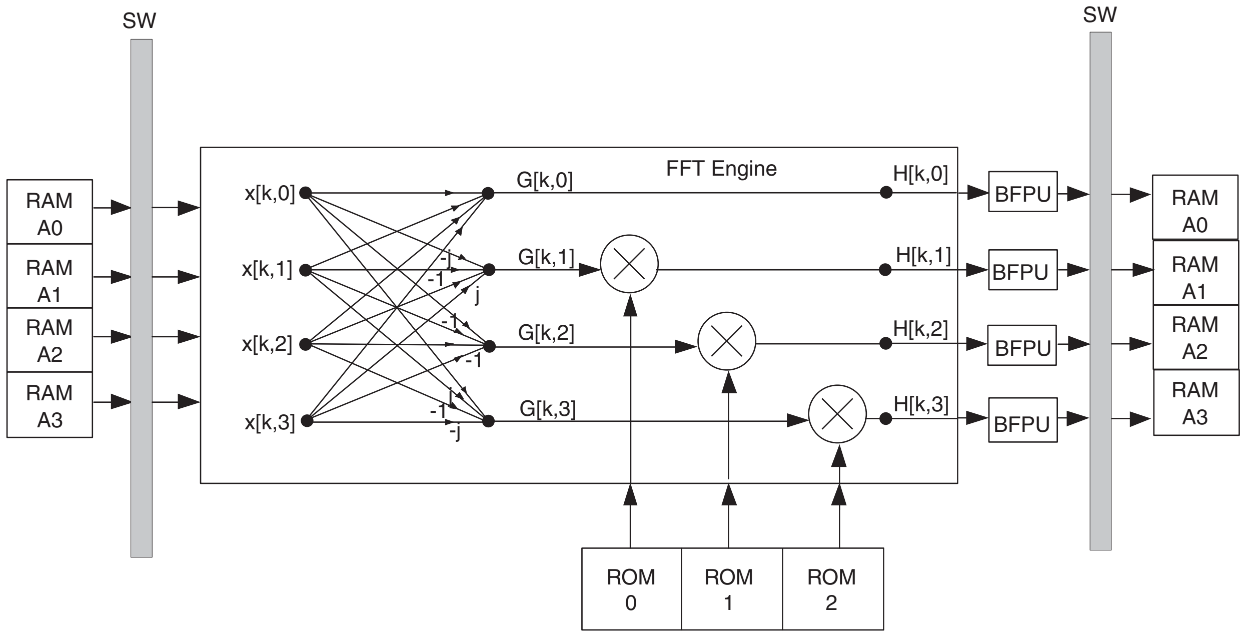
You can parameterize the FFT MegaCore function to use either quad-output or single-output engines. To increase the overall throughput of the FFT MegaCore function, you may also use multiple parallel engines of a variation.

Quad-Output FFT Engine

To minimize transform time, use a quad-output FFT engine. 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

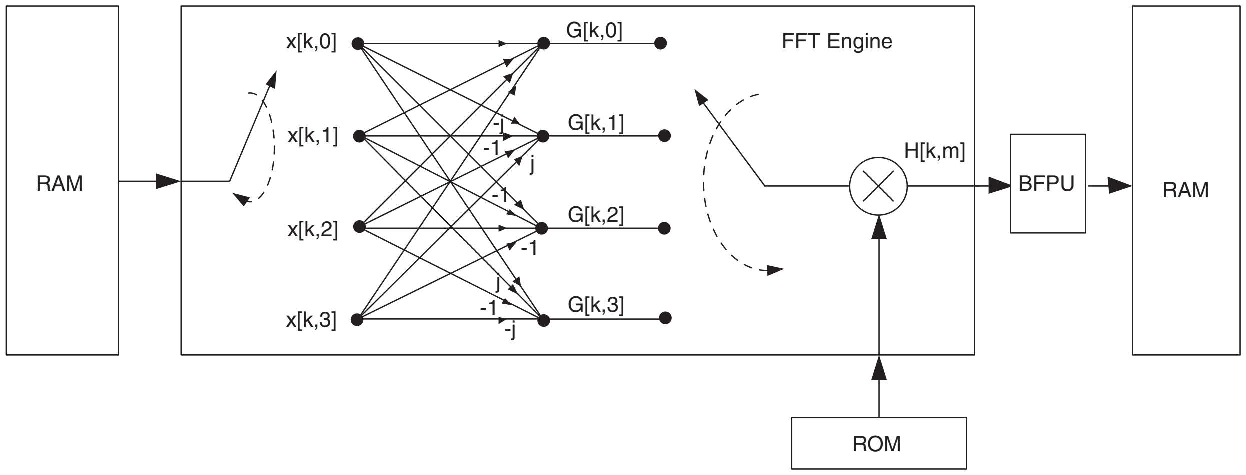


The FFT reads complex data samples x[*k*,*m*] from internal memory in parallel and reorders by switch (SW). Next, the radix-4 butterfly processor processes the ordered samples to form the complex outputs G[*k*,*m*]. Because of the inherent mathematics of the radix-4 DIF decomposition, only three complex multipliers 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 block-floating point units (BFPU) evaluate the four outputs in parallel. The FFT discards the appropriate LSBs and rounds and reorders the complex values before writing them back to internal memory.

Single-Output FFT Engine

For the minimum-size FFT function, use a single-output engine. The term single-output refers to the throughput of the internal FFT butterfly processor. In the engine, the FFT calculates a single butterfly output per clock cycle, requiring a single complex multiplier (Figure 3–2 on page 3–5).

Figure 3–2. Single-Output FFT Engine



I/O Data Flow

- Streaming FFT
- Variable Streaming
- Buffered Burst
- Burst

Streaming FFT

The streaming FFT 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.

The streaming FFT generates a design with a quad output FFT engine and the minimum number of parallel FFT engines for the required throughput.

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

Figure 3–3 on page 3–6 shows an example simulation waveform.

Using the Streaming FFT

1. Deassert the system reset, The data source asserts `sink_valid` to indicate to the FFT function that valid data is available for input.
2. Assert both the `sink_valid` and the `sink_ready` for a successful data transfer.

When the data transfer is complete, the FFT deasserts `sink_sop` and loads the data samples in natural order.

For more information about the signals, refer to Table 3–5 on page 3–17.

For more information about the Avalon-ST interface, refer to the *Avalon Interface Specifications*.

Figure 3–3. FFT Streaming Data Flow Simulation Waveform

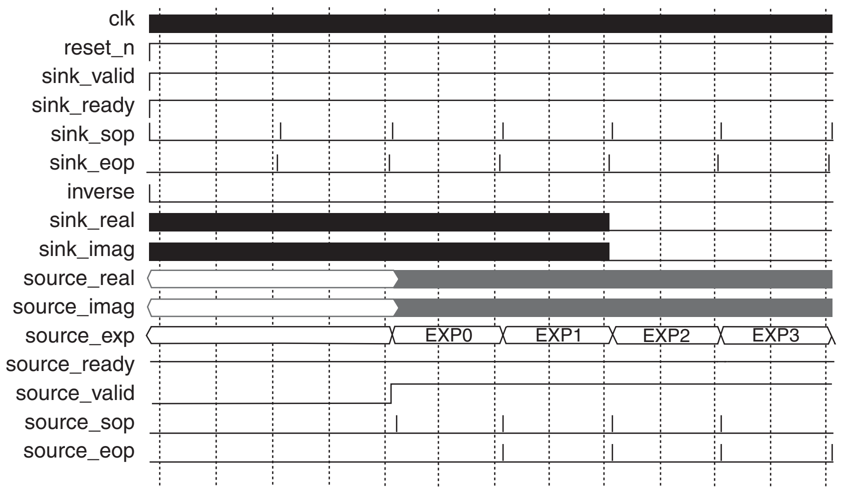
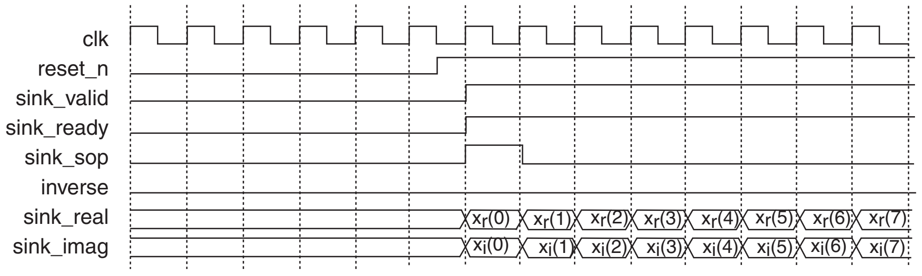


Figure 3–4 shows the input flow control. When the final sample loads, the source asserts `sink_eop` and `sink_valid` for the last data transfer.

Figure 3–4. FFT Streaming Data Flow Input Flow Control


clk
reset_n
sink_valid
sink_ready
sink_sop
inverse
sink_real
sink_imag
xr(0) xr(1) xr(2) xr(3) xr(4) xr(5) xr(6) xr(7)
xi(0) xi(1) xi(2) xi(3) xi(4) xi(5) xi(6) xi(7)

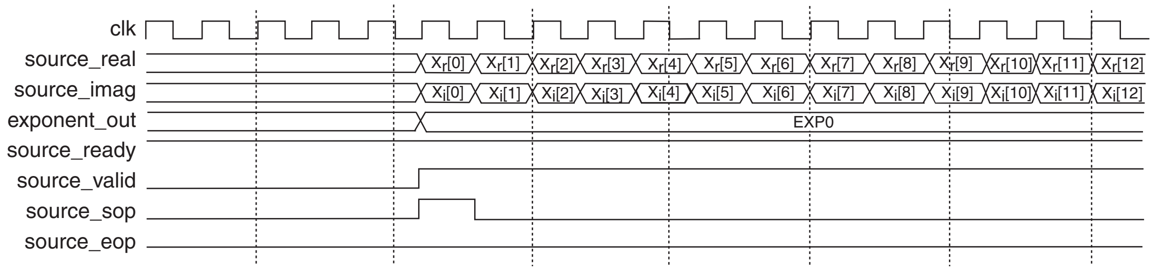
Changing the Direction

To change direction on a block-by-block basis:

1. 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 completes 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 Output Flow Control


clk
source_real
source_imag
exponent_out
source_ready
source_valid
source_sop
source_eop
Xr[0] Xr[1] Xr[2] Xr[3] Xr[4] Xr[5] Xr[6] Xr[7] Xr[8] Xr[9] Xr[10] Xr[11] Xr[12]
Xi[0] Xi[1] Xi[2] Xi[3] Xi[4] Xi[5] Xi[6] Xi[7] Xi[8] Xi[9] Xi[10] Xi[11] Xi[12]
EXP0

After *N* data transfers, the FFT asserts `source_eop` to indicate the end of the output data block (Figure 3–3 on page 3–6).

Enabling the Streaming FFT

1. You must assert the `sink_valid` signal for the FFT to assert `source_valid` (and a valid data output).
2. 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.

Variable Streaming

The variable streaming FFT allows continuous streaming of input data and produces a continuous stream of output data similar to the streaming FFT. With the variable streaming FFT, the transform length represents the maximum transform length. You can perform all transforms of length 2m where 6 < m < log2(transform length) at runtime.

Changing Block Size

To change the size of the FFT on a block-by-block basis, change 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 and Transform Size

fftpts	Transform Size
10000000000	1,024
01000000000	512
00100000000	256
00010000000	128
00001000000	64

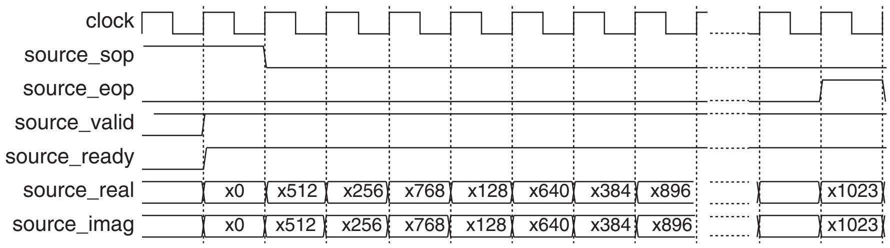
Changing Direction

To change direction on a block-by-block basis:

1. 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 completes 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 3–6 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 3–6. Output Flow Control—Bit Reversed Order



I/O Order

You can set the I/O order and data representation. The input order allows you to select the order in which you feed the samples to the FFT.

Table 3–3. Input Order

Order	Description
Natural order	The FFT requires the order of the input samples to be sequential (1, 2 …, n – 1, n) where n is the size of the current transform.
Bit reverse order	The FFT requires the input samples to be in bit-reversed order.
Digit Reverse Order	The FFT requires the input samples to be in digit-reversed order.
–N/2 to N/2	The FFT requires the input samples to be in the order –N/2 to (N/2) – 1 (also known as DC-centered order)

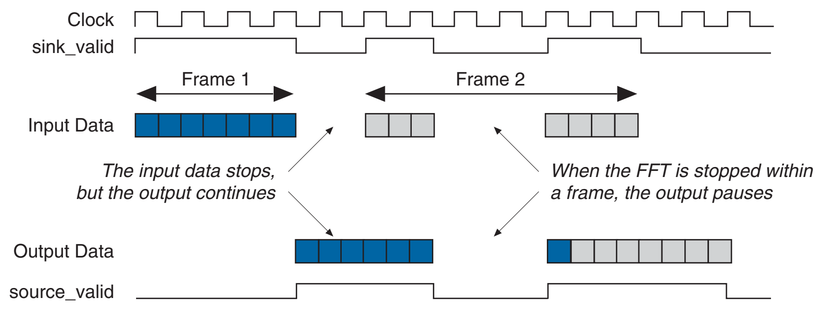
Similarly the output order specifies the order in which the FFT generates the output. Whether you can select **Bit Reverse Order** or **Digit Reverse Order** depends on your **Data Representation** (**Fixed Point** or **Floating Point**). If you select **Fixed Point**, the FFT variation implements the radix-22 algorithm and the reverse I/O order option is **Bit Reverse Order**. If you select **Floating Point**, the FFT variation implements the mixed radix-4/2 algorithm and the reverse I/O order option is **Digit Reverse Order**.

For sample digit-reversed order, if n is a power of four, the order is radix-4 digit-reversed order, in which two-bit digits in the sample number are units in the reverse ordering. For example, if n = 16, sample number 4 becomes the second sample in the sample stream (by reversal of the digits in 0001, the location in the sample stream, to 0100). However, in mixed radix-4/2 algorithm, n need not be a power of four. If n is not a power of four, the two-bit digits are grouped from the least significant bit, and the most significant bit becomes the least significant bit in the digit-reversed order. For example, if n = 32, the sample number 18 (10010) in the natural ordering becomes sample number 17 (10001) in the digit-reversed order.

Enabling the Variable Streaming FFT

1. Assert `sink_valid`.
2. Transfer valid data to the FFT. The FFT processes data. Figure 3–7 shows the FFT behavior when `sink_valid` is deasserted.

Figure 3–7. FFT Behavior When sink_valid is Deasserted


Clock
sink_valid
Frame 1
Frame 2
Input Data
The input data stops, but the output continues
When the FFT is stopped within a frame, the output pauses
Output Data
source_valid

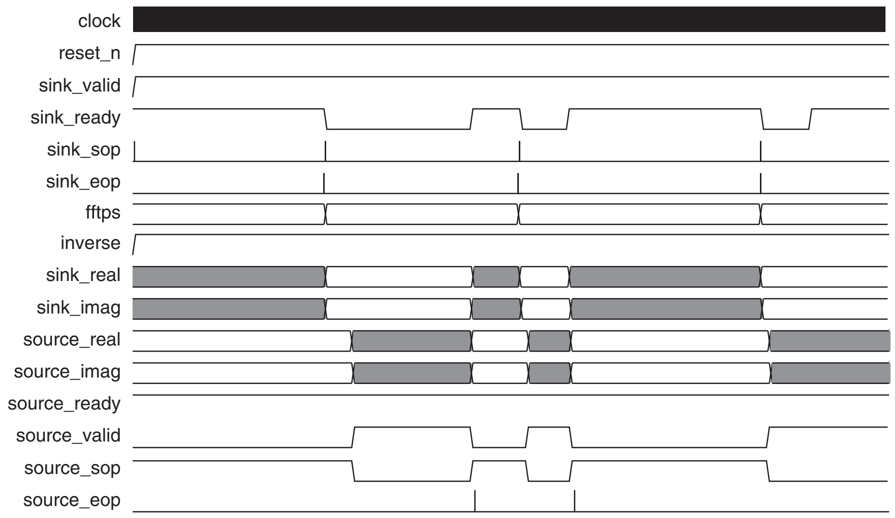
3. Deassert `sink_valid` during a frame to stall the FFT, which then processes no data until you assert `sink_valid`. Any previous frames that are still in the FFT also stall.
4. If you deassert `sink_valid` between frames, the FFT processes and transfers the data currently in the FFT to the output. Figure 3–7 shows the FFT behavior when you deassert `sink_valid` between frames and within a frame.
5. Disable the FFT by deasserting the `clk_en` signal.

Dynamically Changing the FFT Size

6. Change the size of the incoming FFT,

The FFT stalls the incoming data (deasserts the `sink_ready` signal) until all the FFT processes and transfers all of the previous FFT frames of the previous FFT size to the output. Figure 3–8 shows dynamically changing the FFT size for engine-only mode.

Figure 3–8. Dynamically Changing the FFT Size



I/O Order

The **I/O order** determines order of samples entering and leaving the FFT and also determines if the FFT is operating in engine-only mode or engine with bit-reversal or digit-reversal mode.

If the FFT operates in engine-only mode, the output data is available after approximately N + latency clocks cycles after the first sample was input to the FFT. Latency represents a small latency through the FFT core and depends on the transform size. For engine with bit-reversal mode, the output is available after approximately $2N$ + latency cycles.

Figure 3–9 shows the data flow output when the FFT is operating in engine-only mode.

Figure 3–9. Data Flow—Engine-Only Mode

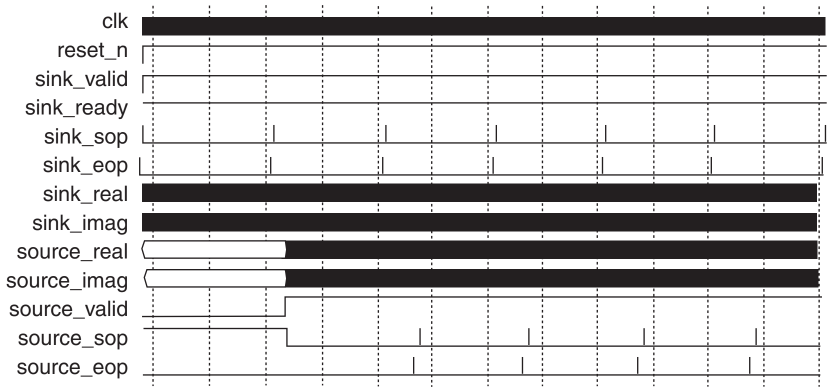
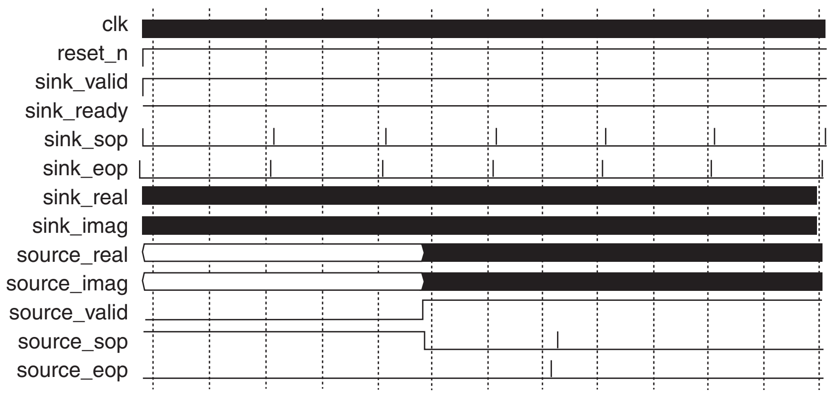


Figure 3–10 shows the data flow output when the FFT is operating in engine with bit-reversal or digit-reversal mode, respectively

Figure 3–10. Data Flow—Engine with Bit-Reversal or Digit-Reversal Mode



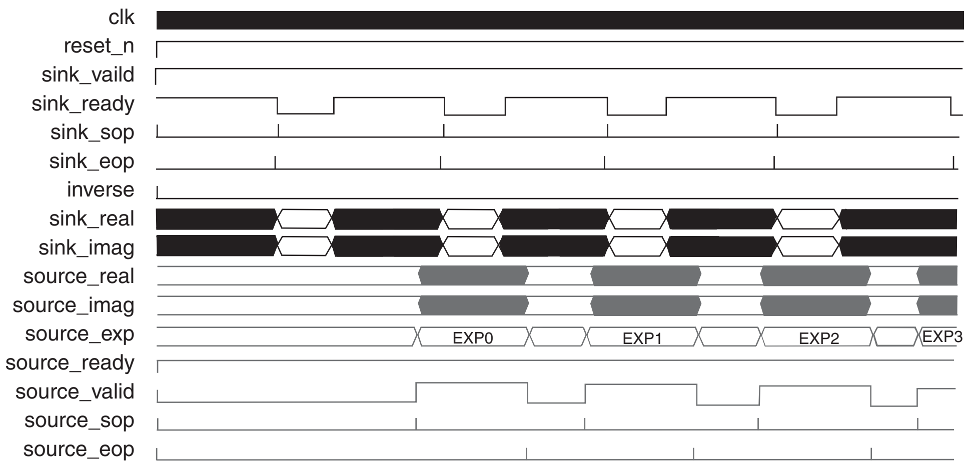
Buffered Burst

The buffered burst I/O data flow FFT requires fewer memory resources than the streaming I/O data flow FFT, but the tradeoff is an average block throughput reduction.

Enabling the Buffered Burst FFT

Figure 3–11 on page 3–11 shows an example simulation waveform.

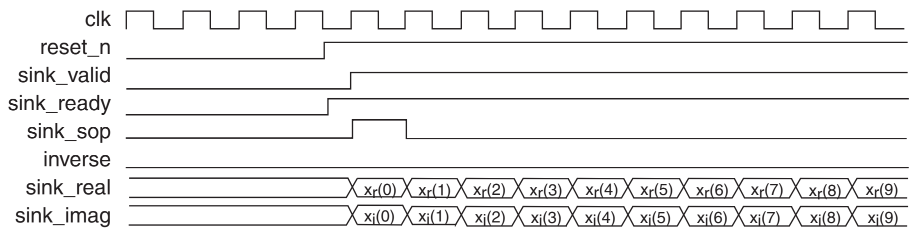
Figure 3–11. FFT Buffered Burst Data Flow Simulation Waveform



1. Deassert the system reset.
2. 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.
3. Load the first complex data sample into the FFT function and simultaneously asserts `sink_sop` to indicate the start of the input block.

4. On the next clock cycle, `sink_sop` is deasserted and you must load the following *N* – 1 complex input data samples in natural order.
5. On the last complex data sample, `assert sink_eop`.
6. When you load the input block, the FFT function begins computing the transform on the stored input block. Hold the `sink_ready` signal high as you can transfer the first few samples of the subsequent frame into the small FIFO at the input. If this FIFO buffer is filled, the FFT deasserts the `sink_ready` signal. It is not mandatory to transfer samples during `sink_ready` cycles. Figure 3–12 shows the input flow control.

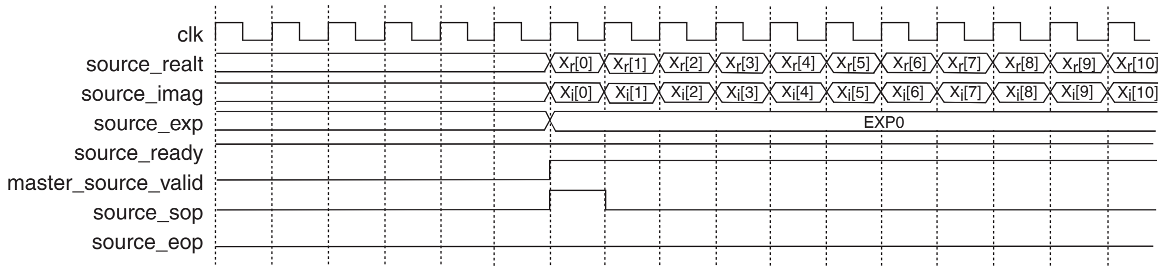
Figure 3–12. FFT Buffered Burst Data Flow Input Flow Control


clk
reset_n
sink_valid
sink_ready
sink_sop
inverse
sink_real
sink_imag
xr(0) xr(1) xr(2) xr(3) xr(4) xr(5) xr(6) xr(7) xr(8) xr(9)
xi(0) xi(1) xi(2) xi(3) xi(4) xi(5) xi(6) xi(7) xi(8) xi(9)

7. 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. Apply a pulse on `sink_sop` aligned in time with the first input sample of the next block to indicate the beginning of the subsequent input block.
8. As in all data flows, the logical level of inverse for a particular block is registered by the FFT at the time when you assert the start-of-packet signal, `sink_sop`.

When the FFT completes the transform of the input block, it asserts the `source_valid` and outputs the complex transform domain data block in natural order (Figure 3–13).

Figure 3–13. FFT Buffered Burst Data Flow Output Flow Control


clk
source_realt
source_imag
source_exp
source_ready
master_source_valid
source_sop
source_eop
Xr[0] Xr[1] Xr[2] Xr[3] Xr[4] Xr[5] Xr[6] Xr[7] Xr[8] Xr[9] Xr[10]
Xi[0] Xi[1] Xi[2] Xi[3] Xi[4] Xi[5] Xi[6] Xi[7] Xi[8] Xi[9] Xi[10]
EXP0

Signals `source_sop` and `source_eop` indicate the start-of-packet and end-of-packet for the output block data respectively (Figure 3–11).

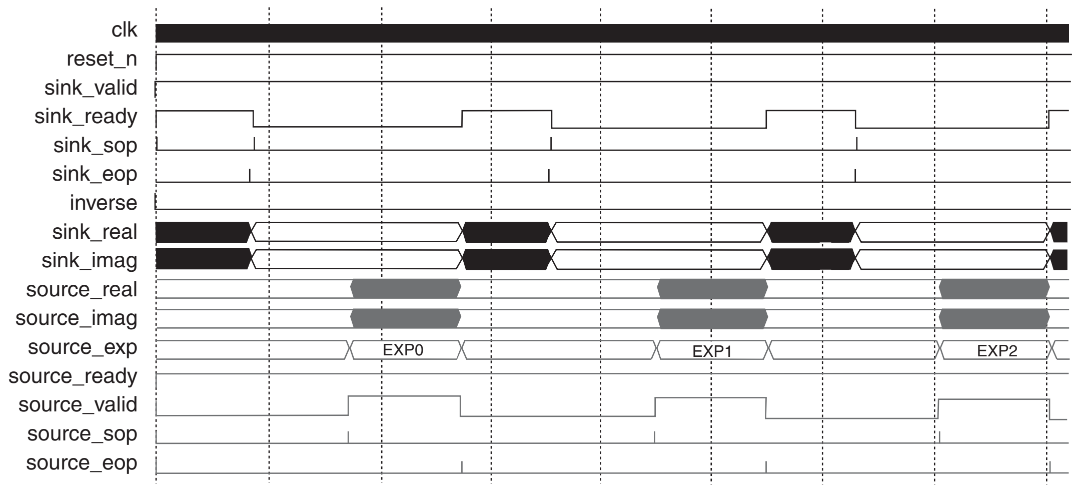
☞ You must assert the `sink_valid` signal for `source_valid` to be asserted (and a valid data output). You must leave `sink_valid` signal asserted at the end of data transfers to extract the final frames of data from the FFT.

For information about enabling the buffered burst FFT, refer to "Enabling the Streaming FFT" on page 3–7.

Burst

The burst I/O data flow FFT operates similarly to the buffered burst FFT, except that the burst FFT requires even lower memory resources for a given parameterization at the expense of reduced average throughput. Figure 3–14 shows the simulation results for the burst FFT. 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 3–14. FFT Burst Data Flow Simulation Waveform



In a burst I/O data flow FFT, the FFT can process a single input block only. A small FIFO buffer at the sink of the block and `sink_ready` is not deasserted until this FIFO buffer is full. You can provide a small number of additional input samples associated with the subsequent input block. You don't have to provide data to the FFT during `sink_ready` cycles. The burst FFT can load the rest of the subsequent FFT frame only when the previous transform is fully unloaded.

For information about enabling the buffered burst FFT, refer to "Enabling the Streaming FFT" on page 3–7.

Parameters

Table 3–4 lists the FFT MegaCore function's parameters.

Table 3–4. 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, 32768, or 65536. Variable streaming also allows 8, 16, 32, 131072, and 262144.	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.
Twiddle Precision	8, 10, 12, 14, 16, 18, 20, 24, 28, 32	The twiddle precision. The values 28 and 32 are available for variable streaming only. Twiddle factor precision must be less than or equal to data precision.
FFT Engine Architecture	Quad Output, Single Output	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. Not available for variable streaming or streaming FFTs.
Number of Parallel FFT Engines	1, 2, 4	
I/O Data Flow	Streaming Variable Streaming Buffered Burst Burst	If you select Variable Streaming and Floating Point, the precision is automatically set to 32, and the reverse I/O order options are Digit Reverse Order.
I/O Order	Bit Reverse Order, Digit Reverse Order, Natural Order, –*N*/2 to *N*/2	The input and output order for data entering and leaving the FFT (variable streaming FFT only). The **Digit Reverse Order** option replaces the **Bit Reverse Order** in variable streaming floating point variations.
Data Representation	Fixed Point or Floating Point	The internal data representation type (variable streaming FFT only), either fixed point with natural bit-growth or single precision floating point. Floating-point bidirectional IP cores expect input in natural order for forward transforms and digit reverse order for reverse transforms. The output order is digit reverse order for forward transforms and natural order for reverse transforms.

Table 3–4. Parameters (Part 2 of 3)

Parameter	Value	Description
Structure	3 Mults/5 Adders 4 Mults/2 Adders	You can implement the complex multiplier structure with four real multipliers and two adders/subtracters, 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 FFTs or in Arria V, Cyclone V, and Stratix V devices.
Implement Multipliers in	DSP Blocks/Logic Cells Logic Cells Only DSP Blocks Only	You can implement each real multiplication 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 × 18 multiplier resources with LEs as needed. Not available for variable streaming FFTs or Arria V, Cyclone V, and Stratix V devices.
DSP Resource Optimization	On or Off	Stratix V devices only. Turn on to implement the complex multiplier structure using Stratix V DSP block complex 18 × 25 multiplication mode or complex 27 × 27 multiplication mode for better DSP resource utilization, at the possible expense of speed. In the variable streaming FFTs, using the floating point representation, this option implements the complex multiplier structure using Stratix V DSP block complex 27 × 27 multiplication mode at the expense of accuracy. If you turn on DSP Resource Optimization, and your variation has data precision between 18 and 25 bits, inclusive, and twiddle precision less than or equal to 18 bits, the FFT MegaCore function configures the DSP blocks in complex 18 × 25 multiplication mode. If you turn on DSP Resource Optimization and your variation does not meet these criteria, the FFT MegaCore function configures the DSP blocks based on the criteria it uses when you do not turn on the option. The FFT MegaCore function configures the Stratix V device according to the following criteria when you turn off the option or it is not available: ■ If data precision and twiddle precision are both less than or equal to 27 bits, configures 3/4 of a DSP block in complex 27 × 27 multiplication mode. This configuration uses only three of the four DSP rows in a single DSP block. ■ If data precision is greater than 27 bits and twiddle precision is less than or equal to 18 bits, configures one DSP block in sum of two 18 × 36 multiplication mode. This configuration uses four DSP rows. ■ Otherwise, configures two DSP blocks in 36 × 36 multiplication mode. This configuration uses eight DSP rows in two DSP blocks. For more information about the Stratix V DSP block modes, refer to the Variable Precision DSP Blocks in Stratix V Devices chapter in the Stratix V Device Handbook.

Table 3–4. Parameters (Part 3 of 3)

Parameter	Value	Description
Global clock enable	On or Off	Turn on if you want to add a global clock enable to your design.
Twiddle ROM Distribution	100% M4K to 100% M512 or 100% M9K to 100% MLAB	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 IV and Stratix V devices), you can choose to distribute the ROM storage requirement between M4K (M9K in Stratix IV devices) RAM and M512 (MLAB) RAM blocks by adjusting the 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. In Stratix V devices, replace M4K (M9K) with M20K memory blocks. 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. You can set memory use balance with the Twiddle ROM Distribution, turn on Use M-RAM Blocks, and turn on Implement appropriate logic functions in RAM. If your FFT variation targets an appropriate device family, the Use M144K Blocks option replaces the Use M-RAM Blocks option. Not available for variable streaming FFTs or in the Cyclone series .
Use M-RAM or M144K blocks	On or Off	Implements suitable data RAM blocks within the FFT MegaCore function in M-RAM (M144K in Stratix IV devices) to reduce M4K (M9K) RAM block usage, in device families that support M-RAM blocks. Not available for variable streaming FFTs, or the Cyclone or Stratix series.
Implement appropriate logic functions in RAM	On or Off	Uses embedded RAM blocks to implement internal logic functions, for example, tapped delay lines in the FFT MegaCore function. This option reduces the overall logic element count. Not available for variable streaming FFTs.

Interfaces and Signals

The FFT IP core uses the Avalon-ST interface. 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.

Avalon-ST 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 in which 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 that includes an FFT MegaCore function, you may not need backpressure if you know the downstream components can always receive data.

For more information about the Avalon-ST interface, refer to the *Avalon Interface Specifications*.

Avalon-ST Signals

Table 3–5 lists the Avalon-ST interface signals.

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

Table 3–5. Avalon-ST Signals (Part 1 of 2)

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.This signal can be asserted asynchronously, but must remain asserted at least one `clk` clock cycle and must be deasserted synchronously with `clk`. Refer to the *Recommended Design Practices* chapter in volume 1 of the *Quartus II Handbook* for a sample circuit that ensures synchronous deassertion of an active-low reset signal.
`sink_eop`	Input	`endofpacket`	1	Indicates the end of the incoming FFT frame.
`sink_error`	Input	`error`	2	Indicates an error has occurred in an upstream module, because of an illegal usage of the Avalon-ST protocol. The following errors are defined (refer to Table 3–7): ■ 00 = no error ■ 01 = missing start of packet (SOP) ■ 10 = missing end of packet (EOP) ■ 11 = unexpected EOP If this signal is not used in upstream modules, set to zero.

Table 3–5. Avalon-ST Signals (Part 2 of 2)

Signal Name	Direction	Avalon-ST Type	Size	Description
`sink_imag`	Input	`data`	*data precision width*	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`	*data precision width*	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 `sink_valid` and `sink_ready` are asserted, a data transfer takes place. Refer to "Enabling the Variable Streaming FFT" on page 3–9.
`sink_data`	Input	`data`	Variable	In Qsys systems, this Avalon-ST-compliant data bus includes all the Avalon-ST input data signals.
`source_eop`	Output	`endofpacket`	1	Marks the end of the outgoing FFT frame. Only valid when `source_valid` is asserted.
`source_error`	Output	`error`	2	Indicates an error has occurred either in an upstream module or within the FFT module (logical `OR` of `sink_error` with errors generated in the FFT). Refer to Table 3–7 for error codes.
`source_exp`	Output	`data`	6	Streaming, burst, and buffered burst FFTs only. Signed block exponent: Accounts for scaling of internal signal values during FFT computation.
`source_imag`	Output	`data`	(*data precision width* + *growth*) (1)	Imaginary output data. For burst, buffered burst, streaming, and variable streaming floating point FFTs, the output data width is equal to the input data width. For variable streaming fixed point FFTs, the size of the output data is dependent on the number of stages defined for the FFT and is 2 bits per radix 2^2 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, streaming, and variable streaming floating point FFTs, the output data width is equal to the input data width. For variable streaming fixed point FFTs, the size of the output data is dependent on the number of stages defined for the FFT and is 2 bits per radix 2^2 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.
`source_data`	Output	`data`	Variable	In Qsys systems, this Avalon-ST-compliant data bus includes all the Avalon-ST output data signals.

Note to Table 3–5:

(1) Variable streaming fixed point FFT only. Growth is $\log_2(N) + 1$.

Component Specific Signals

Table 3–6 shows the component specific signals.

Table 3–6. Component Specific Signals

Signal Name	Direction	Size	Description
`fftpts_in`	Input	$\log_2$(*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	$\log_2$(*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.

Incorrect usage of the Avalon-ST interface protocol on the sink interface results in a error on `source_error`. Table 3–7 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 3–7. Error Handling Behavior

Error	source_error	Description
Missing SOP	01	Asserted when valid goes high, but there is no start of frame.
Missing EOP	10	Asserted if the FFT accepts *N* valid samples of an FFT frame, but there is no EOP signal.
Unexpected EOP	11	Asserted if EOP is asserted before *N* valid samples are accepted.

Signals in Qsys Systems

When you instantiate your design in a Qsys sytem and target Arria 10 devices, the signals appear as a single bus:

- In:
  - Real
  - Imaginary
- Out:
  - Real
  - Imaginary

A. Block Floating Point Scaling

Block-floating-point (BFP) scaling is a trade-off between fixed-point and full floating-point FFTs.

In fixed-point FFTs, 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.

Floating-point FFTs represents each number as a mantissa with an individual exponent. The improved precision is offset by demand for increased device resources.

In a block-floating point FFT, 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 FFT 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 scales according to a measure of the block dynamic range on the output of the previous pass. The FFT accumulates the number of shifts and then outputs them 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 is 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, you must scale the output by 2 $(^{-exponent\_out})$ to account for the discarded LSBs during the transform.

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:

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

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

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 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 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.

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 (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

Table A–1. Exponent Scaling Values for FFT / IFFT (1)

N	P	Single Output Engine		Quad Output Engine	
		Max (2)	Min (2)	Max (2)	Min (2)
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 maximum and minimum 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 with 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.

Example of Scaling

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:

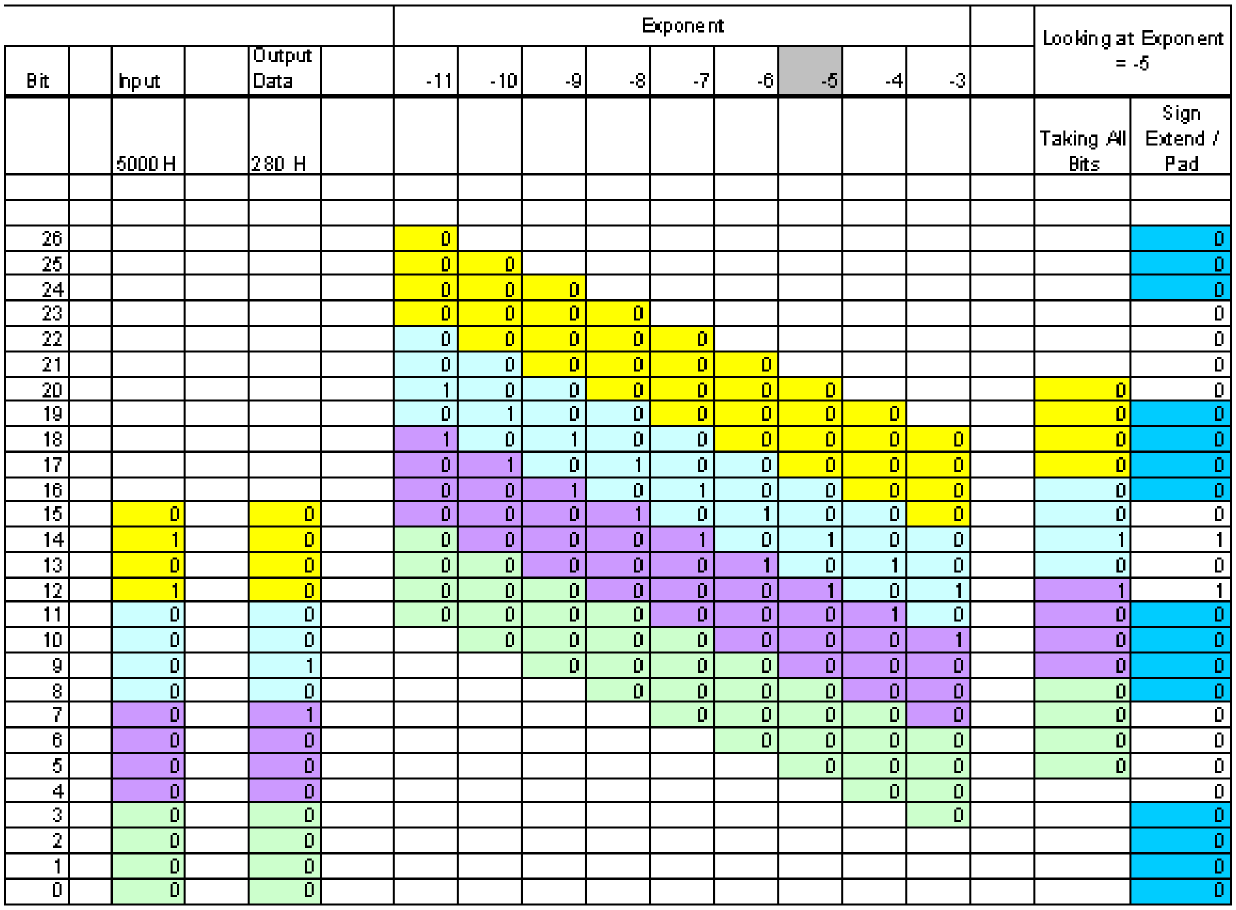
```
case (exp)
   6'b110101 : //-11 Set data equal to MSBs
      begin
         full_range_real_out[26:0] <= {real_in[15:0],11'b0};
         full_range_imag_out[26:0] <= {imag_in[15:0],11'b0};
      end
   6'b110110 : //-10 Equals left shift by 10 with sign extension
      begin
         full_range_real_out[26] <= {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 must choose how many and which bits must be carried forward in the processing chain. The choice of bits determines the absolute gain relative to the input sample level.

Figure A–1 on page A–5 demonstrates the effect of scaling for all possible values for the 256-point quad output FFT with an input signal level of 0x5000. The output of the FFT is 0x280 when the exponent = –5. The figure illustrates all cases of valid exponent values of scaling to the full scale storage register [26..0]. Because the exponent is –5, you must check 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 (0x0005000), 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 0x5000. However, if you choose a different bit range, such as the top 16 bits, the result is 0x000A. 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 must 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 = 0x5000



			Exponent									Looking at Exponent = -5	
Bit	Input	Output Data	-11	-10	-9	-8	-7	-6	-5	-4	-3		
	5000 H	280 H										Taking All Bits	Sign Extend / Pad
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
3	0	0									0		0
2	0	0											0
1	0	0											0
0	0	0											0

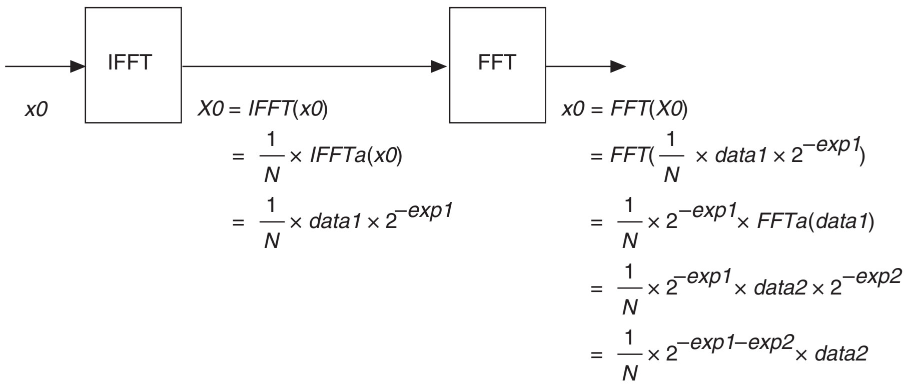
Unity Gain in an IFFT+FFT Pair

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.

BFP arithmetic does not provide an input for the exponent, so 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 on page A–6 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



$$X0 = IFFT(x0) = \frac{1}{N} \times IFFTa(x0) = \frac{1}{N} \times data1 \times 2^{-exp1}$$

$$x0 = FFT(X0) = FFT\left(\frac{1}{N} \times data1 \times 2^{-exp1}\right) = \frac{1}{N} \times 2^{-exp1} \times FFTa(data1) = \frac{1}{N} \times 2^{-exp1} \times data2 \times 2^{-exp2} = \frac{1}{N} \times 2^{-exp1-exp2} \times data2$$

where:

$x0$ = Input data to IFFT

$X0$ = Output data from IFFT

N = number of points

data1 = IFFT output data and FFT input data

data2 = FFT output data

exp1 = IFFT output exponent

exp2 = FFT output exponent

IFFTa = IFFT

FFTa = FFT

Any scaling operation on *X*0 followed by truncation loses the value of *exp1* and does not result in unity gain at *x0*. Any scaling operation must be done on *X*0 only when it is the final result. If the intermediate result *X*0 is first padded with *exp1* number of zeros and then truncated or if the data bits of *X*0 are truncated, the scaling information is lost.

One way to keep unity gain is by passing the *exp1* value to the output of the FFT block. The other way is to preserve the full precision of $data1 \times 2^{-exp1}$ 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 *Achieving Unity Gain in Block Floating Point IFFT+FFT Pair* design example under DSP Design Examples at **www.altera.com**.

Additional Information

This chapter provides additional information about the document and Altera.

Revision History

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

Date	Version	Changes Made
August 2014	14.0 Arria 10 Edition	■ Added support for Arria 10 devices. ■ Added new `source_data` bus description. ■ Added Arria 10 generated files description. ■ Removed table with generated file descriptions. ■ Removed `clk_ena`
June 2014	14.0	■ Removed Cyclone III and Stratix III device support ■ Added support for MAX 10 FPGAs. ■ Added instructions for using IP Catalog
November 2013	13.1	■ Added more information to variable streaming I/O dataflow. ■ Removed device support for following devices: ■ HardCopy II, HardCopy III, HardCopy IV E, HardCopy IV GX ■ Stratix, Stratix GX, Stratix II, Stratix II GX ■ Cyclone, Cyclone II ■ Arria GX
November 2012	12.1	Added support for Arria V GZ devices.
November 2011	11.1	■ Updated Table 1–1. ■ Added Arria V and Cyclone V device support in Table 1–2. ■ Added Stratix V in the "Performance and Resource Utilization" section. ■ Updated Table 3–4 to include 8-point FFT.
May 2011	11.0	■ Added user-controlled parameter for DSP resource optimization in Stratix V devices. ■ Changed device support level from Preliminary to Final for Arria II GX, Arria II GZ, Cyclone III LS, and Cyclone IV devices. ■ Changed device support level from HardCopy Companion to HardCopy Compilation for HardCopy III E, HardCopy IV E, and HardCopy IV GX devices.
December 2010	10.1	■ Added preliminary support for Arria II GZ devices. ■ Updated support level to final support for Stratix IV GT devices.
July 2010	10.0	■ Added preliminary support for Stratix V devices. ■ Added new Transform Length values.
November 2009	9.1	■ Maintenance update. ■ Added preliminary support for Cyclone III LS, Cyclone IV, and HardCopy IV GX devices.
March 2009	9.0	Added Arria`clk_ena` II GX device support.
November 2008	8.1	No changes.

Date	Version	Changes Made
May 2008	8.0	■ Added Stratix IV device support. ■ Changed descriptions of the behavior of `sink_valid` and `sink_ready`.
October 2007	7.2	■ Corrected timing diagrams. ■ Added single precision floating point data representation information.
May 2007	7.1	■ Added support for Arria GX devices. ■ Added new generated files.
December 2006	7.0	Added support for Cyclone III devices.
December 2006	6.1	■ Changed interface information. ■ Added variable streaming information.

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Nontechnical support (general)	Email	nacomp@altera.com
(software licensing)	Email	authorization@altera.com

Note to Table:

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

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	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	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.

Visual Cue	Meaning
"Subheading Title"	Quotation marks indicate references to sections in 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`. The suffix `n` denotes an active-low signal. For 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`).
↵	An angled arrow instructs you to press the Enter key.
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.
?	The question mark directs you to a software help system with related information.
👣	The feet direct you to another document or website with related information.
🎥	The multimedia icon directs you to a related multimedia presentation.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
⚠ WARNING	A warning calls attention to a condition or possible situation that can cause you injury.
✉	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.