

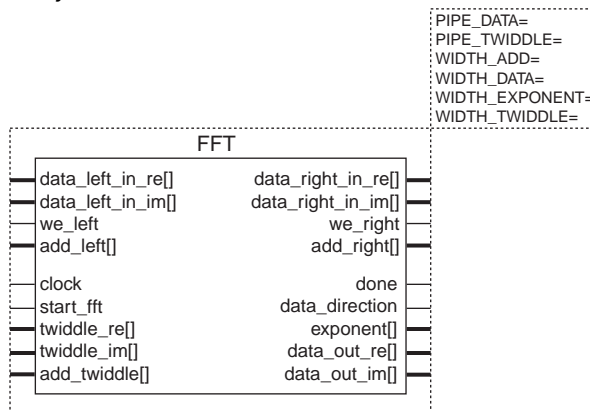
Features

- `fft` MegaCore function implementing a fast Fourier transform (FFT)
- Optimized for the Altera® FLEX® 10K architecture
- Block-floating-point to provide the best accuracy
- Parameterized data width, twiddle width, and number of points
- Dual memory architecture
- Complex data in and data out, decimation in frequency (DIF) FFT
- Flexible memory interface with data and twiddle memories that can be implemented in internal and/or external memory

General Description

The `fft` MegaCore function implements a fast Fourier transform (FFT), which is used to separate a signal into its constituent frequencies. This function is useful for a variety of digital signal processing (DSP) applications, including wireless communications, voice recognition, spectrum analysis, and noise analysis. [Figure 1](#) shows the symbol for the `fft` MegaCore function.

Figure 1. `fft` Symbol



To optimize throughput, the function uses a dual memory architecture in which data is read from one memory and written to the other memory. This dual-memory architecture consists of a right and left memory. The function also uses a third memory, known as the twiddle memory, that must be kept separate from the left and right memories to maximize throughput. All three memories can be implemented in internal memory, using FLEX 10K embedded array blocks (EABs), and/or external RAM.

AHDL Function Prototype

The Altera Hardware Description Language (AHDL) Function Prototype for the `fft` is shown below:

```
FUNCTION fft (clock, start_fft,
             data_left_in_re[WIDTH_DATA-1..0],
             data_left_in_im[WIDTH_DATA-1..0],
             data_right_in_re[WIDTH_DATA-1..0],
             data_right_in_im[WIDTH_DATA-1..0],
             twiddle_re[WIDTH_TWIDDLE-1..0],
             twiddle_im[WIDTH_TWIDDLE-1..0])
  WITH (WIDTH_DATA, WIDTH_TWIDDLE, PIPE_DATA,
        PIPE_TWIDDLE, WIDTH_EXPONENT, FFT_DIRECTION,
        WIDTH_ADD, EXPONENT_INITIAL_VALUE)
  RETURNS (done, data_direction, we_left,
           add_left[WIDTH_ADD-1..0], we_right,
           add_right[WIDTH_ADD-1..0],
           add_twiddle[WIDTH_ADD-2..0],
           data_out_re[WIDTH_DATA-1..0],
           data_out_im[WIDTH_DATA-1..0],
           exponent[WIDTH_EXPONENT-1..0]);
```

VHDL Component Declaration

The VHDL Component Declaration for the `fft` is shown below:

```
COMPONENT fft
  GENERIC(
    WIDTH_DATA : POSITIVE;
    WIDTH_TWIDDLE : POSITIVE;
    PIPE_DATA : INTEGER;
    PIPE_TWIDDLE : INTEGER;
    WIDTH_EXPONENT : POSITIVE;
    WIDTH_ADD : POSITIVE;
    EXPONENT_INITIAL_VALUE : INTEGER);
  PORT(
    clock : IN STD_LOGIC := '0';
    start_fft : IN STD_LOGIC;
    data_left_in_re, data_left_in_im,
    data_right_in_re, data_right_in_im : IN
    STD_LOGIC_VECTOR(WIDTH_DATA-1 DOWNT0 0);
    twiddle_re, twiddle_im : IN
    STD_LOGIC_VECTOR(WIDTH_TWIDDLE-1 DOWNT0 0);
    done, data_direction, we_left : OUT STD_LOGIC;
    add_left : OUT STD_LOGIC_VECTOR(WIDTH_ADD-1
```

```

DOWNT0 0);
we_right : OUT STD_LOGIC;
add_right : OUT STD_LOGIC_VECTOR(WIDTH_ADD-1
DOWNT0 0);
add_twiddle : OUT STD_LOGIC_VECTOR(WIDTH_ADD-2
DOWNT0 0);
data_out_re, data_out_im : OUT
STD_LOGIC_VECTOR(WIDTH_DATA-1 DOWNT0 0);
exponent : OUT STD_LOGIC_VECTOR(WIDTH_EXPONENT-1
DOWNT0 0));
END COMPONENT;
```

Ports

Table 1 describes the ports for the `fft`.

Table 1. <i>fft</i> Ports			
Name	Type	Required	Description
<code>clock</code>	Input	Yes	Clock signal.
<code>start_fft</code>	Input	Yes	Starts the <code>fft</code> after data is loaded.
<code>data_left_in_re[]</code>	Input	Yes	Real data input from the left memory.
<code>data_left_in_im[]</code>	Input	Yes	Imaginary data input from the left memory.
<code>data_right_in_re[]</code>	Input	Yes	Real data input from the right memory.
<code>data_right_in_im[]</code>	Input	Yes	Imaginary data input from the right memory.
<code>twiddle_re[]</code>	Input	Yes	Real twiddle input from the twiddle memory.
<code>twiddle_im[]</code>	Input	Yes	Imaginary data input from the twiddle memory.
<code>done</code>	Output	Yes	Goes high when the <code>fft</code> has completed the calculation.
<code>data_direction</code>	Output	Yes	When high, the <code>fft</code> reads data from the left memory and writes data to the right memory. When low, the inverse operation is performed.
<code>we_left</code>	Output	Yes	Write enable for the left memory.
<code>we_right</code>	Output	Yes	Write enable for the right memory.
<code>add_left[]</code>	Output	Yes	Address bus for the left memory.
<code>add_right[]</code>	Output	Yes	Address bus for the right memory.
<code>add_twiddle[]</code>	Output	Yes	Address bus for the twiddle memory.
<code>data_out_re[]</code>	Output	Yes	Real data output to both left and right memories.
<code>data_out_im[]</code>	Output	Yes	Imaginary data output to both left and right memories.
<code>exponent[]</code>	Output	Yes	Exponent of the resultant data. Valid after <code>done</code> goes high. Exponent of the block-floating-point notation. All data should be scaled by $2^{\text{exponent}[]}$.

Parameters

Table 2 describes the parameters for the fft.

Name	Value	Description
PIPE_DATA	Integer	Number of clock cycles after add_left[] or add_right[] becomes active that the data will be valid at data_left_in_re[], data_left_in_im[], data_right_in_re[], or data_right_in_im[].
PIPE_TWIDDLE	Integer	The number of clock cycles after add_twiddle[] becomes active that twiddle_re[] or twiddle_im[] becomes active.
WIDTH_ADD	Integer	Width of the data address buses. The number of points in the fft is $2^{\text{WIDTH_ADD}}$.
WIDTH_DATA	Integer	Data width.
WIDTH_EXPONENT	Integer	Number of bits in the exponent.
WIDTH_TWIDDLE	Integer	Twiddle width.

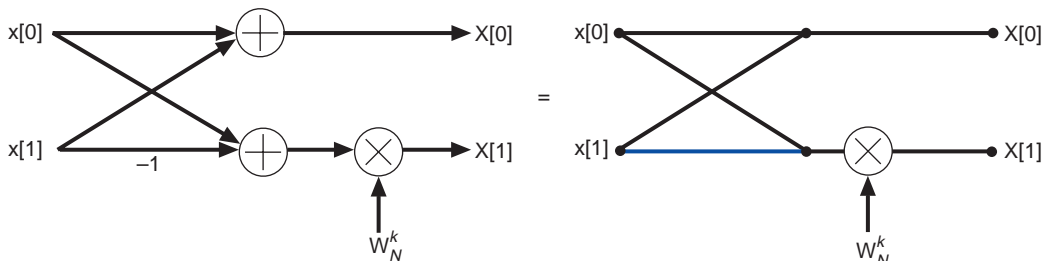
Functional Description

The fft function implements a decimation in frequency (DIF) algorithm and contains all the core logic functions necessary to compute an FFT. To maximize flexibility, the fft function does not include a memory or I/O interface. The memory and I/O interface is dependent upon the end application, and thus must be customized to fit each application.

Figure 2 shows the basic butterfly operation performed by the DIF algorithm.

Figure 2. Basic DIF FFT Butterfly Operation

Blue lines indicate negative numbers.



where:

$$X[0] = x[0] + x[1]$$

$$X[1] = W_N^k(x[0] - x[1])$$

$$W_N^k = \text{Twiddle} = e^{-j2\pi k/N} = \cos(2\pi k/N) - j\sin(2\pi k/N)$$

$$k = 0 \text{ to } (N/2 - 1)$$

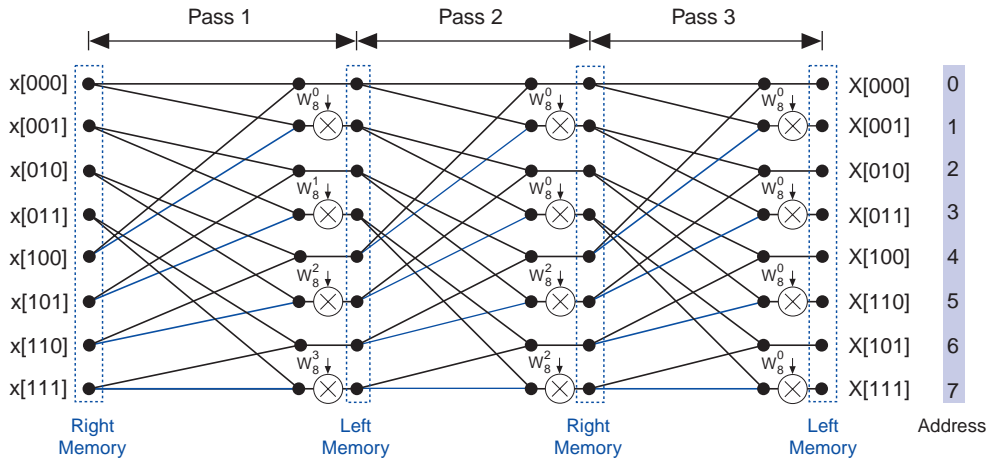
$$N = \text{Number of points in FFT}$$

$$j = \sqrt{-1}$$

Figure 3 shows a sample 8-point DIF FFT algorithm as implemented by the `fft` function. The input data addresses are in normal order, and the output data addresses are in bit-reversed order. However, dedicated hardware can be easily added to address the memories so that data appears in normal order for both inputs and outputs.

Figure 3. 8-Point DIF FFT Algorithm

Solid blue lines indicate negative numbers.



The `fft_on_chip` reference design includes a memory and I/O interface as well as dedicated hardware for addressing data in normal order for both inputs and outputs. Example 1 on page 8 and Example 2 on page 10 describe the implementation of the `fft_on_chip` reference design.

When processing data, the `fft` alternates which memory it reads from and writes to each pass. For example, on Pass 1 in Figure 3, the `fft` simultaneously reads data from the right memory, performs the butterfly operation, and writes the result to the left memory. On Pass 2, the `fft` simultaneously reads the result from the left memory, performs the next step of the DIF algorithm, and writes the result to the right memory. The `fft` continues this process until the calculation is complete.

The number of points in the `fft` determines which memory the `fft` reads first. When the number of points is an odd power of 2 (e.g., $2^3 = 8$, $2^5 = 32$), data is always read from the right memory on the first pass through the `fft`. When the number of points is an even power of 2 (e.g., $2^4 = 16$, $2^6 = 64$), the `fft` switches which memory it reads first after each complete calculation. The `data_direction` output indicates which memory the `fft` is reading from and writing to.

Twiddle Generation

The data is multiplied by twiddles (i.e., coefficients) each time it passes through the `fft`. Each twiddle (W) is calculated by the equation:

$$W_N^k = e^{(-j2\pi k)/N} = \cos(2\pi k/N) - j\sin(2\pi k/N)$$

where: N = Number of points in `fft` = $2^{\text{WIDTH_ADD}}$
 k = 0 to $(N/2 - 1)$
 j = $\sqrt{-1}$

The real part of the twiddle is $\cos(2\pi k/N)$ and the imaginary part is $-\sin(2\pi k/N)$. Both twiddle outputs must be stored in the twiddle memory.

The **twiddle** utility program distributed with the `fft` MegaCore function automatically generates an EAB Memory Initialization File (**.mif**) that contains all the twiddle data. This MIF can be easily translated to other formats for use in an external ROM.



For help on the syntax, type `twiddle` from the UNIX or DOS command prompt.

Number Representation

All buses in and out of the `fft` are in two's complement fractional notation, in which the numbers range from -1 to 1 . The input data is always in two's complement fractional notation, with one bit to the left of the binary point and `WIDTH_DATA` - 1 bits to the right. For example, the decimal number 0.5 is represented as 0.1000000_B , and -0.5 is 1.1000000_B . The smallest 8-bit wide number that can be represented in two's complement fractional notation is -1.0 and the largest number is:

$$\frac{2^{\text{WIDTHTWIDDLE} - 1} - 1}{2^{\text{WIDTHTWIDDLE} - 1}} = \frac{127}{128} = 0.9922$$

Twiddle data is in inverted two's complement notation (i.e., the decimal number 0.5 is represented as 1.1000000_B , and -0.5 is 0.1000000_B). The twiddle data uses this notation because the most used twiddle—twiddle zero (W_N^0)—has the value of $1.0 + j0$ and less error is accumulated when W_N^0 can be represented exactly. Although the accuracy of $W_{N/4}^0$ suffers slightly, the final result is more accurate because this twiddle is used less in the FFT calculation.

Memory Interface

The `fft` contains the core functions necessary to compute an FFT and does not include memory or an I/O interface. The function does not include a memory or I/O interface so the user has the most flexibility in every application. FLEX 10K devices contain a finite amount of memory, so not all memory configurations can be accommodated on-chip. [Table 3](#) shows the available memory configurations for EPF10K100 and EPF10K50 devices.

Table 3. Available Configurations in EPF10K100 & EPF10K50 Devices *Note (1)*

Device	EABs Available	Data Width (Bits)	Twiddle Width (Bits)	Memory Implementation					
				Twiddles & Data in EABs		Twiddles in EABs & Data in External RAM		Twiddles in External RAM & Data in EABs	
				Points	EABs	Points	EABs	Points	EABs
EPF10K100	12	≤ 8	≤ 8	512	10	2,048	8	512	8
		9 to 16	≤ 8	256	10	2,048	8	256	8
		9 to 16	9 to 16	256	12	1,024	8	256	8
		> 16	> 16	<i>Note (2)</i>	<i>Note (2)</i>	<i>Note (2)</i>	<i>Note (2)</i>	<i>Note (2)</i>	<i>Note (2)</i>
EPF10K50	10	≤ 8	≤ 8	512	10	2,048	8	512	8
		9 to 16	≤ 8	256	10	2,048	8	256	8
		9 to 16	9 to 16	<i>Note (2)</i>	<i>Note (2)</i>	1,024	8	256	8
		> 16	> 16	<i>Note (2)</i>	<i>Note (2)</i>	<i>Note (2)</i>	<i>Note (2)</i>	<i>Note (2)</i>	<i>Note (2)</i>

Notes:

- (1) If both the data and twiddles are in external RAM, the design can use an unlimited number of points.
- (2) The device cannot support this configuration because the RAM required exceeds the available memory in the device.

Pipelining

The designer can choose the number of pipeline stages in the memory path, allowing a trade-off between size and speed for a particular memory interface. For example, the `fft_on_chip` function described in “[Example 1: On-Chip RAM, Odd Number of Passes, with No Data Buffering](#)” uses a data pipeline delay of three, and a twiddle pipeline delay of two. Because the twiddle memory in the `fft_on_chip` function is a synchronous ROM with registered inputs and outputs, the `PIPE_TWIDDLE` parameter is set to two. The data memory is a synchronous RAM with registered inputs (address) and outputs (data), and with multiplexers on the data, address, and write enable inputs. Because these multiplexers are pipelined to maximize performance, the delay from the address/control valid to data valid on the data ports is three. Therefore, `PIPE_DATA` is set to three.

Example Implementations

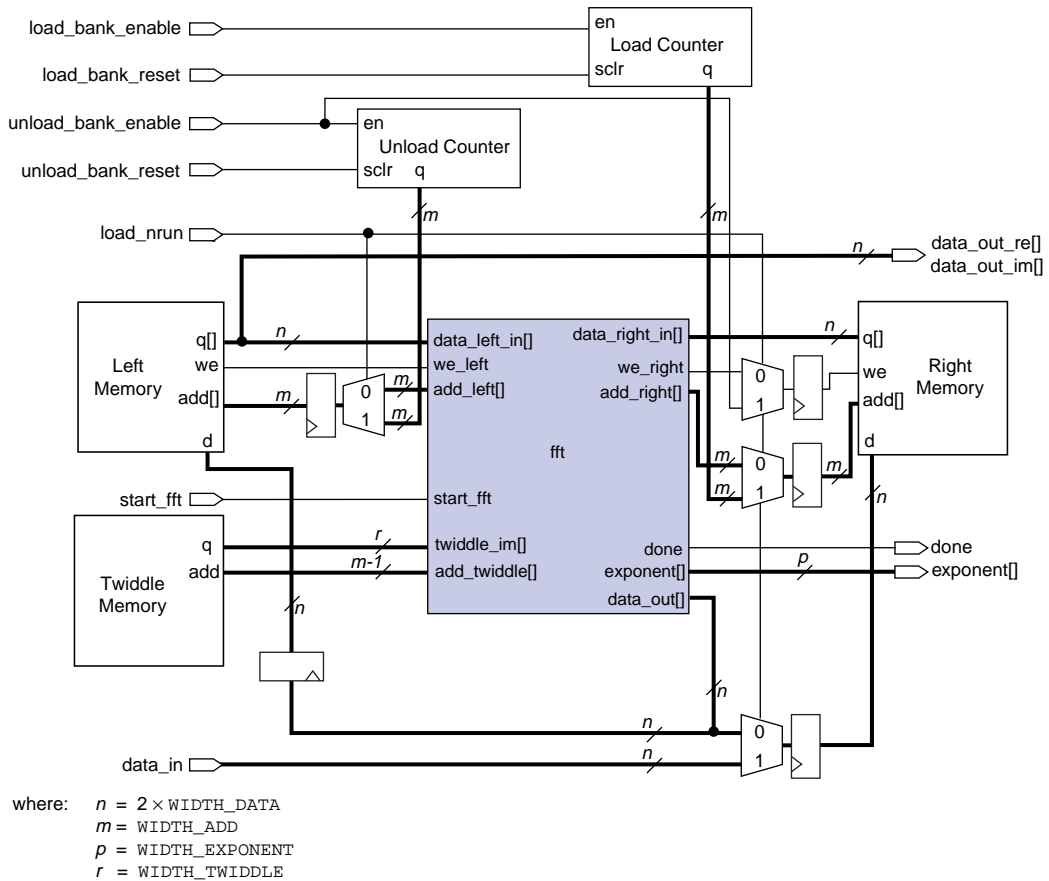
When using the `fft`, the majority of the design effort will be spent on creating the memory and I/O interface. The user can choose from a wide variety of memory and I/O interface schemes; this section describes four of these schemes:

- Example 1: On-chip RAM, odd number of passes (even number of address bits), with no data buffering
- Example 2: On-chip RAM, even number of passes, with no data buffering
- Example 3: Off-chip RAM, odd number of passes, with no data buffering
- Example 4: Off-chip RAM, odd number of passes, with data buffering

Example 1: On-Chip RAM, Odd Number of Passes, with No Data Buffering

This example describes the same memory architecture provided by the `fft_on_chip` reference design. The right, left, and twiddle memories are all implemented in FLEX 10K EABs. Data is loaded into the right memory and processed by the `fft`. When the `fft` is not processing data, data can be loaded and unloaded from both memories simultaneously. [Figure 4](#) shows a block diagram of this implementation.

Figure 4. Block Diagram of Example 1



Memory & I/O Interface

Because data passes through the `fft` an odd number of times, the `fft` always reads new data from the right memory and writes the result to the left memory. This scheme can cause a problem if both the `fft` and the I/O interface require access to the right and left memory control and data buses. However, this problem can be avoided by placing multiplexers on the data and address buses. Because these multiplexers can cause a performance bottleneck if they are not pipelined, a register is placed after the data and address multiplexers. Since the EAB is used with both registered inputs and outputs to maintain performance, the delay from address valid to data valid is 3. Therefore, `PIPE_DATA` is set to 3.

Load & Unload Addressing

The incoming data is in normal bit order, and the outgoing data (as placed in the left memory) is in bit-reversed order. Therefore, the designer can use either one or two counters to load and unload the memories. If one counter is used, the right memory receives the load counter output in normal bit order, and the left memory receives the load counter output in bit-reversed order. When the I/O interface has one address counter, loads and unloads must be completed simultaneously. If two counters are used (as shown in [Figure 4](#)), data can be individually loaded and unloaded once done is asserted.

Twiddle Memory

The twiddle memory interface is simple to design because the twiddle memory is always read-only. The twiddle memory does not require a write enable signal or an I/O interface, and thus the twiddle memory is connected directly to the `fft`. The twiddle memory has registered inputs and outputs, so the `PIPE_TWIDDLE` parameter is set to 2.

Usage

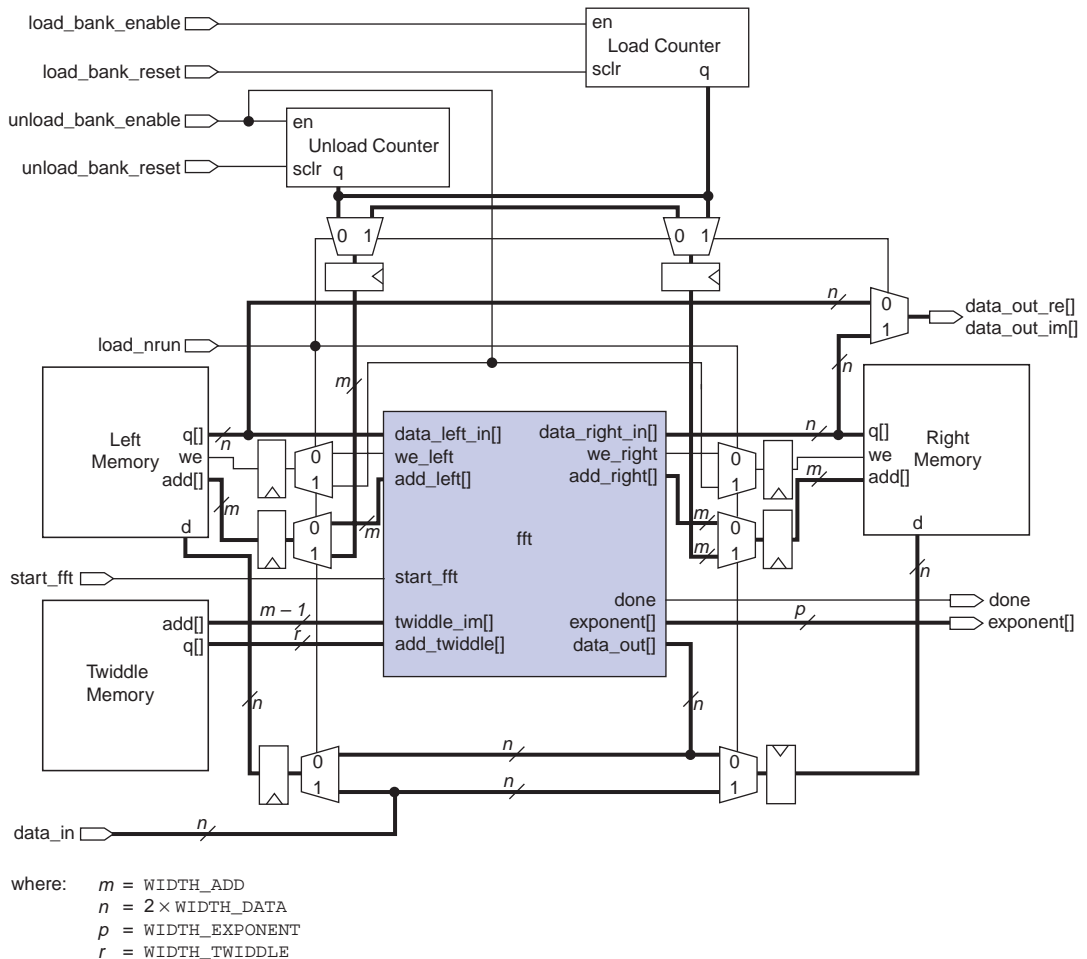
To load data into the right memory, assert `load_nrun` for $N + 1$ clock cycles, and assert `load_bank_enable` for N cycles while placing data on the `data_in` port. The `load_nrun` input is then deasserted to return control to the `fft`. On the next clock cycle, assert `start_fft` for one clock cycle to begin computation. After the `fft` has finished processing data and has written the result to the left memory, the `fft` asserts the `done` output.

Example 2: On-Chip RAM, Even Number of Passes, with No Data Buffering

Example 2 is similar to Example 1, except the data passes through the `fft` an even number of times. To allow maximum throughput in this scheme, the `fft` alternates the first memory it reads after each complete FFT calculation (i.e., for the first FFT calculation, `fft` reads new data from the right memory; for the second FFT calculation, the `fft` reads new data from the left memory.) When the data passes through the `fft` an even number of times, the `fft` reads new data from one memory, processes the data, and writes the result back to the same memory. Because each memory is storing either new data or the result of a calculation, the I/O interface should be able to load new data into one memory and unload the result from the other memory.

Figure 5 shows a block diagram of Example 2. This example uses a multiplexing scheme to allow both load and unload addresses to access the right and left memories. The multiplexing scheme also allows data to be loaded and unloaded to and from both memories. To maintain performance, the multiplexers are pipelined, thus adding a stage of delay between the address counters and the left and right memories. An extra stage of delay is added in the data_in path so that load_bank_enable, data_in, left_address, and right_address are all active during the same clock cycles.

Figure 5. Block Diagram of Example 2



Example 3: Off-Chip RAM, Odd Number of Passes, without Data Buffering

Using off-chip RAM allows the `fft` to be longer than 256 points. An even longer `fft` can be created by using external synchronous static RAM (SSRAM), while still maintaining a high clock rate. The SSRAM used in this example is a synchronous cache RAM intended for use with high-speed microprocessors. This type of RAM is available from a variety of vendors.

This architecture is similar to the architectures used in Examples 1 and 2, except for the use of off-chip RAM. The address, data multiplexing, and pipelining schemes remain the same as Examples 1 and 2. The only difference is that the RAM does not have separate `data_in` and `data_out` ports and instead uses bidirectional ports. Therefore, this design must use bidirectional pins on the FLEX 10K device that are used as output pins when writing and input pins at all other times.

Example 4: Off-Chip RAM, Odd Number of Passes, with Data Buffering

In examples 1, 2, and 3, new data cannot be captured when the `fft` is processing data because all of the RAM is in use. Through the use of four memories (i.e., right 1, right 2, left 1, and left 2 memories), this example allows new data to be captured while the `fft` is processing the current data. While the `fft` is processing data from the right 1 and left 1 memories, new data is being loaded into the left 2 memory. When both the left 2 memory is full and the `fft` has finished processing data, the left 1 and left 2 memories are swapped and the right 1 and right 2 memories are swapped. Then, the `start_fft` signal is immediately asserted to begin processing data from the left 2 and right 2 memories. While the `fft` is processing data from the left 2 and right 2 memories, the left 1 and right 1 memories are loaded and unloaded.



101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
<http://www.altera.com>
Applications Hotline:
(800) 800-EPLD
Customer Marketing:
(408) 544-7104
Literature Services:
(888) 3-ALTERA
lit_req@altera.com

Altera, MAX, MAX+PLUS, MAX+PLUS II, MegaCore, FLEX, FLEX 10K, EPF10K100, and EPF10K50 are trademarks and/or service marks of Altera Corporation in the United States and other countries. Altera acknowledges the trademarks of other organizations for their respective products or services mentioned in this document. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

Copyright © 1997 Altera Corporation. All rights reserved.



I.S. EN ISO 9001