FIFO Core User Guide

UG-CORE-FIFO2-v2.0
February 2024
www.efinixinc.com

Contents

Introduction

Note: The FIFO is available in the Efinity software v2021.1.165 with patch v2021.1.165.2.19 or higher. The FIFO (Legacy) is obsoleted and replaced with FIFO in Efinity software v2021.2. You cannot migrate automatically from the FIFO (Legacy) to the FIFO. Therefore, Efinix® recommends that you use the FIFO for all new designs. You can continue to use FIFO (Legacy) with the Efinity software v2021.1.165 or lower. However, the FIFO will not be supported in future Efinity releases.

The FIFO core is a customizable first-in first-out memory queue that uses block RAM in the FPGA for storage. The core has parameters you use to create a custom instance. For example, you can set the FIFO depth, the data bus width, whether the read and write domains are synchronous or asynchronous, etc.

Use the IP Manager to select IP, customize it, and generate files. The FIFO core has an interactive wizard to help you set parameters. The wizard also has options to create a testbench and/or example design targeting an Efinix® development board.

Features

- Depths up to 131,072 words
- Data widths from 1 to 256 bits
- Symmetric or non-symmetric aspect ratios (read-to-write port ratios ranging from 1:16 to 16:1)
- Synchronous or asynchronous clock domains supports standard or First–Word–Fall–Through (FWFT)
- Programmable full and empty status flags, set by user–defined parameters
- Almost full and almost empty flags indicate one word left
- Configurable handshake signals
- Asynchronous clock domain FWFT read mode
- FIFO datacount to indicate how many words available in FIFO
- Option to exclude optional flags
- Option to exclude overflow and underflow protection
- Includes example designs targeting the Trion® T20 BGA256 Development Board
- Verilog RTL and simulation testbench

Device Support

Table 1: FIFO Core Device Support

FPGA Family	Supported Device
Trion	All
Titanium	All

Resource Utilization and Performance

Note: The resources and performance values provided are based on some of the supported FPGAs. These values are just guidance and can change depending on the device resource utilization, design congestion, and user design.

The following timing data are based on default settings with overflow protection and underflow protection disabled.

Titanium Resource Utilization and Performance

Table 2: Synchronous Clock FIFO

FPGA	Mode	Asymmetric Width Ratio	Logic Elements (Logic, Adders, Flipflops, etc.)	Memory Block	DSP Block	f_{MAX} (MHz)[1]	Efinity Version[2]
Ti60 F225 C4	Standard	1:1	44/60800 (0.07%)	1/256 (0.4%)	0/160 (0%)	600	2023.2
		1:2	44/60800 (0.07%)	2/256 (0.8%)	0/160 (0%)	600	
	FWFT	1:1	70/60800 (0.1%)	1/256 (0.4%)	0/160 (0%)	600	
		1:2	65/60800 (0.1%)	2/256 (0.8%)	0/160 (0%)	600	

Table 3: Asynchronous Clock FIFO

FPGA	Mode	Asymmetric Width Ratio	Logic Elements (Logic, Adders, Flipflops, etc.)	Memory Block	DSP Block	f_{MAX} (MHz)[1]		Efinity Version[2]
						wr_clk_i	rd_clk_i	
Ti60 F225 C4	Standard	1:1	147/60800 (0.07%)	1/256 (0.4%)	0/160 (0%)	600	600	2023.2
		1:2	139/60800 (0.07%)	2/256 (0.8%)	0/160 (0%)	600	350	
	FWFT	1:1	189/60800 (0.1%)	1/256 (0.4%)	0/160 (0%)	600	600	
		1:2	174/60800 (0.1%)	2/256 (0.8%)	0/160 (0%)	600	350	

(1) Using default parameter settings.
(2) Using Verilog HDL.

Trion Resource Utilization and Performance

Table 4: Synchronous Clock FIFO

FPGA	Mode	Asymmetric Width Ratio	Logic Elements (Logic, Adders, Flipflops, etc.)	Memory Block	DSP Block	f_{MAX} (MHz)[3]	Efinity Version[4]
T20 F256 C4	Standard	1:1	44/19728 (0.2%)	2/204 (1%)	0/36 (0%)	200	2023.2
		1:2	41/19728(0.2%)	2/204(1%)	0/36 (0%)	200	
	FWFT	1:1	70/19728 (0.4%)	2/204 (1%)	0/36 (0%)	200	
		1:2	65/19728 (0.3%)	2/204 (1%)	0/36 (0%)	200	

Table 5: Asynchronous Clock FIFO

FPGA	Mode	Asymmetric Width Ratio	Logic Elements (Logic, Adders, Flipflops, etc.)	Memory Block	DSP Block	f_{MAX} (MHz)[3]		Efinity Version[4]
						wr_clk_i	rd_clk_i	
T20 F256 C4	Standard	1:1	147/19728(0.7%)	2/204 (1%)	0/36 (0%)	200	200	2023.2
		1:2	139/19728 (0.7%)	2/204 (1%)	0/36 (0%)	200	200	
	FWFT	1:1	189/19728 (1.0%)	2/204 (1%)	0/36 (0%)	200	200	
		1:2	174/19728 (0.8%)	2/204 (1%)	0/36 (0%)	200	200	

(3) Using default parameter settings.
(4) Using Verilog HDL.

Release Notes

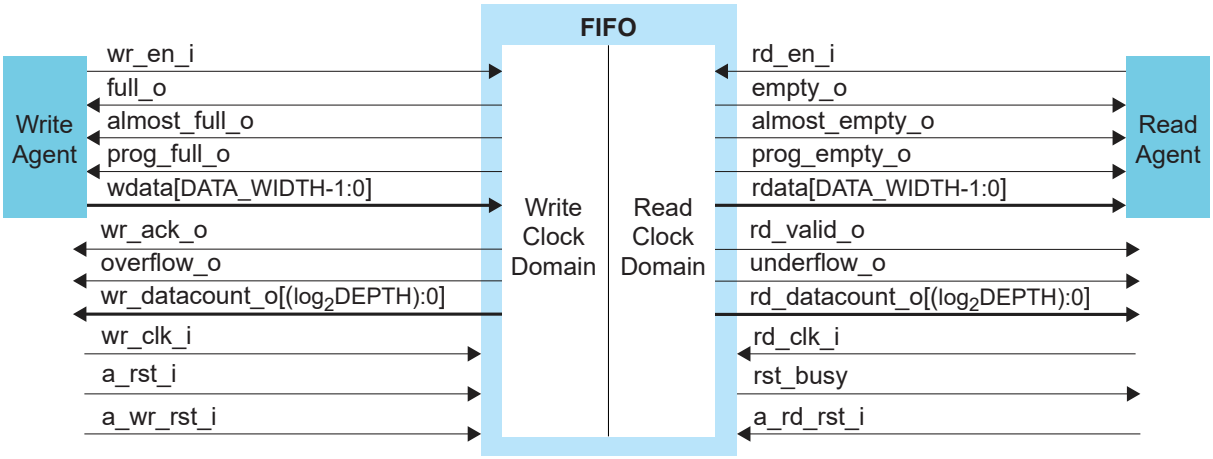
You can refer to the IP Core Release Notes for more information about the IP core changes. The IP Core Release Notes is available in the Efinity Downloads page under each Efinity software release version.

Note: You must be logged in to the Support Portal to view the IP Core Release Notes.

Functional Description

The FIFO core is a first-in first-out memory queue for any application requiring an ordered storage buffer and retrieval. The core provides an optimized solution using the block RAM in Trion® and Titanium FPGAs. The core supports synchronous (read and write use the same clock) and asynchronous (read and write use different clocks) clocking.

Figure 1: FIFO System Block Diagram



Ports

Table 6: FIFO Core Clock, Reset, and Datacount Ports

Port	Synchronous	Asynchronous	Direction	Description
a_rst_i	✓	✓	Input	Reset. Asynchronous reset signal that initializes all internal pointers and output flags.
a_wr_rst_i	✓		Input	The incoming reset signal should already synchronized to the write clock domain. You only use this port if you set the BYPASS_RESET_SYNC parameter to 1.
a_rd_rst_i	✓		Input	The incoming reset signal should already synchronized to the read clock domain. You only use this port if you set the BYPASS_RESET_SYNC parameter to 1.
rst_busy		✓	Output	When asserted, this signal indicate the core is being reset.
wr_clk_i		✓	Input	Write clock. All signals in the write domain are synchronous to this clock.
rd_clk_i		✓	Input	Read clock. All signals in the read domain are synchronous to this clock.
clk_i	✓		Input	Clock. All signals on the write and read domains are synchronous to this clock.
wr_datacount_o [*n*:0]	✓	✓	Output	FIFO write domain data count. Applicable to asymmetric width ratio. *n*=$\log_2$[DEPTH].
rd_datacount_o [*n*:0]	✓	✓	Output	FIFO read domain data count. Applicable to asymmetric width ratio. *n*=$\log_2$[DEPTH].
datacount_o [*n*:0]	✓		Output	FIFO data count. Applicable to symmetric width ratio. *n*=$\log_2$[DEPTH].

Table 7: FIFO Core Write Ports

For both synchronous and asynchronous clocks.

Port	Direction	Description
wdata [*m*-1:0]	Input	Write data. The input data bus used when writing to the FIFO buffer. *m*=DATA_WIDTH.
wr_en_i	Input	Write enable. If the FIFO buffer is not full, asserting this signal causes data (on wdata) to be written to the FIFO.
full_o	Output	Full flag. When asserted, this signal indicates that the FIFO buffer is full. Write requests are ignored when the FIFO is full if overflow protection option is enabled. In this case, initiating a write while full is not destructive to the FIFO.
almost_full_o	Output	Optional, almost full. When asserted, this signal indicates that only one more write can be performed before the FIFO is full.
prog_full_o	Output	Optional, programmable full. This signal is asserted when the number of words in the FIFO is greater than or equal to the assert threshold. It is deasserted when the number of words in the FIFO is less than the negate threshold.
wr_ack_o	Output	Optional, write acknowledge. This signal indicates that a write request (wr_en_i) during the prior clock cycle succeeded.
overflow_o	Output	Optional, overflow. This signal is exposed when overflow protection option is enabled to indicate that a write request (wr_en_i) during the prior clock cycle was rejected because the FIFO buffer is full. In this case, overflowing the FIFO is not destructive to the contents of the FIFO.

Table 8: FIFO Core Read Ports

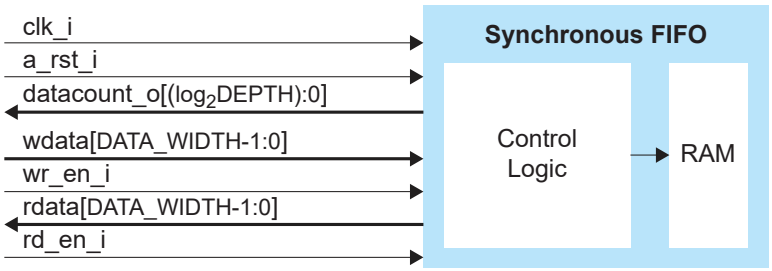
For both synchronous and asynchronous clocks.

Port	Direction	Description
rdata [*m*-1:0]	Output	Read data. The output data bus driven when reading the FIFO buffer. *m*=DATA_WIDTH.
rd_en_i	Input	Read enable. If the FIFO buffer is not empty, asserting this signal causes data to be read from the FIFO (output on rdata).
empty_o	Output	Empty flag. When asserted, this signal indicates that the FIFO buffer is empty. When empty, Read requests are ignored if underflow protection option is enabled. In this case, initiating a read while empty is not destructive to the FIFO.
almost_empty_o	Output	Optional, almost empty flag. When asserted, this signal indicates that only one word remains in the FIFO buffer before it is empty.
prog_empty_o	Output	Optional, programmable empty. This signal is asserted when the number of words in the FIFO buffer is less than or equal to the assert threshold. It is de-asserted when the number of words in the FIFO exceeds the negate threshold.
rd_valid_o	Output	Optional, read valid. This signal indicates that valid data is available on the output bus (rdata).
underflow_o	Output	Optional, underflow. This signal is exposed when underflow protection option is enabled to indicate that the read request (rd_en_i) during the previous clock cycle was rejected because the FIFO buffer is empty. In this case, underflowing the FIFO is not destructive to the FIFO.

Synchronous FIFO Operation

The FIFO core signals are synchronized on the rising edge clock of the respective clock domain. If you want to synchronize to the falling clock edge, use an inverter before sending the signal to the clock input.

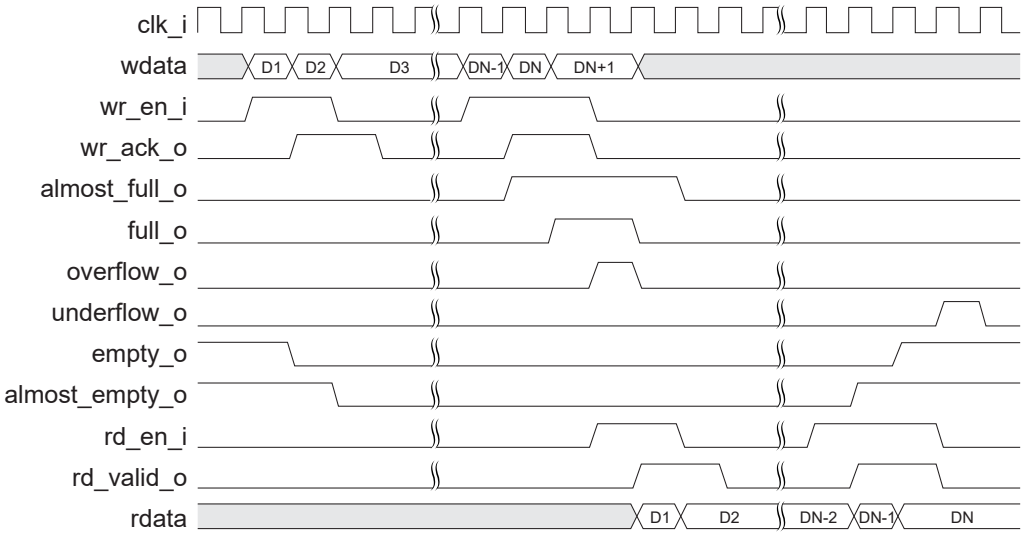
Figure 2: Synchronous FIFO Block Diagram



Standard Mode

The following waveform shows the FIFO behavior in standard mode when it is written until full and then read until empty. D1 and DN are the first and last data, respectively.

Figure 3: Synchronous FIFO Standard Mode Waveform



When overflow protection option is enabled, if the system tries to write data DN+1 when `full_o` is asserted, the core ignores DN+1 and asserts `overflow_o`. `full_o` deasserts during a read request, signaling that the FIFO is ready for more write requests. When the last data is read from the FIFO, the core asserts `empty_o`, indicating there is no more data. Further read requests when there is no more data triggers an assertion on `underflow_o` if the underflow protection option is enabled.

When overflow protection option is disabled, if the system tries to write data DN+1 when `full_o` is asserted, the core still writes DN+1. User logic needs to guarantee that it is not overflowing the FIFO by monitoring `almost_full` signal that indicates only 1 entry left before the FIFO is full. The same concept applies to underflow protection.

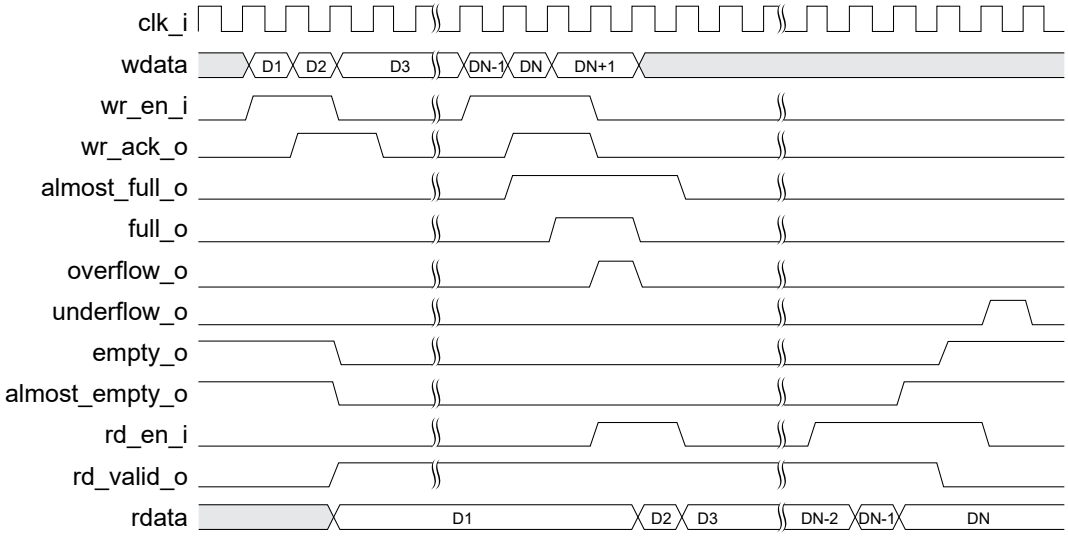
First-Word-Fall-Through Mode

First-Word-Fall-Through (FWFT), is a mode in which the first word written into the FIFO "falls through" and is available at the output without a read request. The following waveform shows the behavior of the FIFO in FWFT mode when it is written until full and then read until empty. D1 and DN are the first and last data, respectively.

The write behavior is the same as standard mode; the read behavior is different. When the first word is written into the FIFO buffer, the core deasserts `empty_o` and asserts `rd_valid_o`. There is one clock cycle of latency from `wr_en_i` to deassert `empty_o` and assert `rd_valid_o`. Consequently, the first word that falls through the FIFO onto the `rdata` also has the one additional clock cycle of latency.

D1 is available on the `rdata` output data bus without a read request (that is, `rd_en_i` is not asserted). When the second data is written into the FIFO buffer, the output data does not change until there is a read request. When it detects a read request, the FIFO core outputs the next available data onto the output bus. If the current data is the last data DN and the core detects a read request, it asserts `empty_o` and deasserts `rd_valid_o`. Additional reads underflow the FIFO.

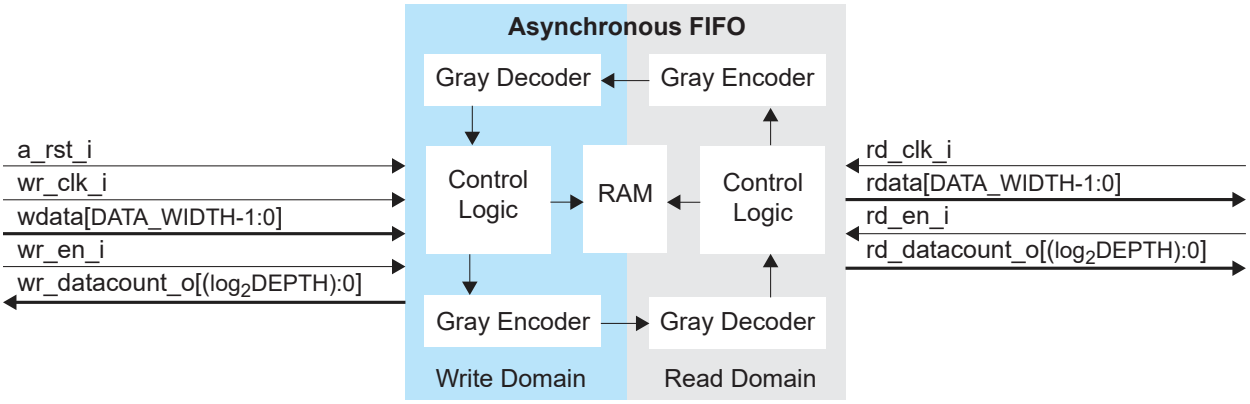
Figure 4: Synchronous FIFO FWFT Mode Waveform



Asynchronous FIFO Operation

With an asynchronous FIFO, the two protocols can work in their respective clock domains and still transfer reliable data to each other. When there is a write or read request affecting its own respective domain's flags, the asynchronous FIFO has 0 delays. Whereas when affecting the other domain's flags, it has a 1 clock cycle delay from its respective domain plus 2 clock cycles of the other domain. For example, a write request only reflects on the read domain after 1 write clock cycle plus 2 read clock cycles and vice versa. Enabling the `PIPELINE_REG`adds 1 more additional clock cycle of the other domain on top of it. Refer to the latency table for asynchronous FIFO in **Latency** for more info.

Figure 5: Asynchronous FIFO Block Diagram



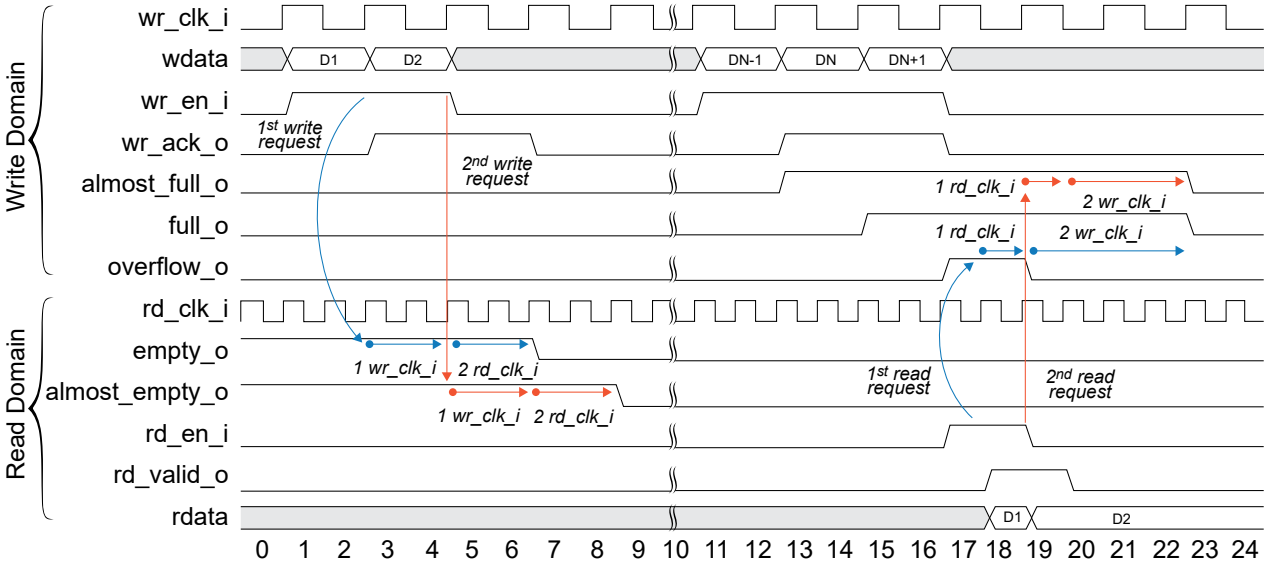
For asynchronous FIFO, a write operation affecting the write domain flags and a read operation affecting the read domain flags have the same behavior as the synchronous FIFO except when they are affecting crossed domain flags. The following examples emphasize the cross-clock domain flags update latency.

Standard Mode

The following figures show examples of asynchronous FIFO standard mode with a faster read clock and write clock, respectively. The waveforms show the FIFO written until full and a few read requests afterwards.

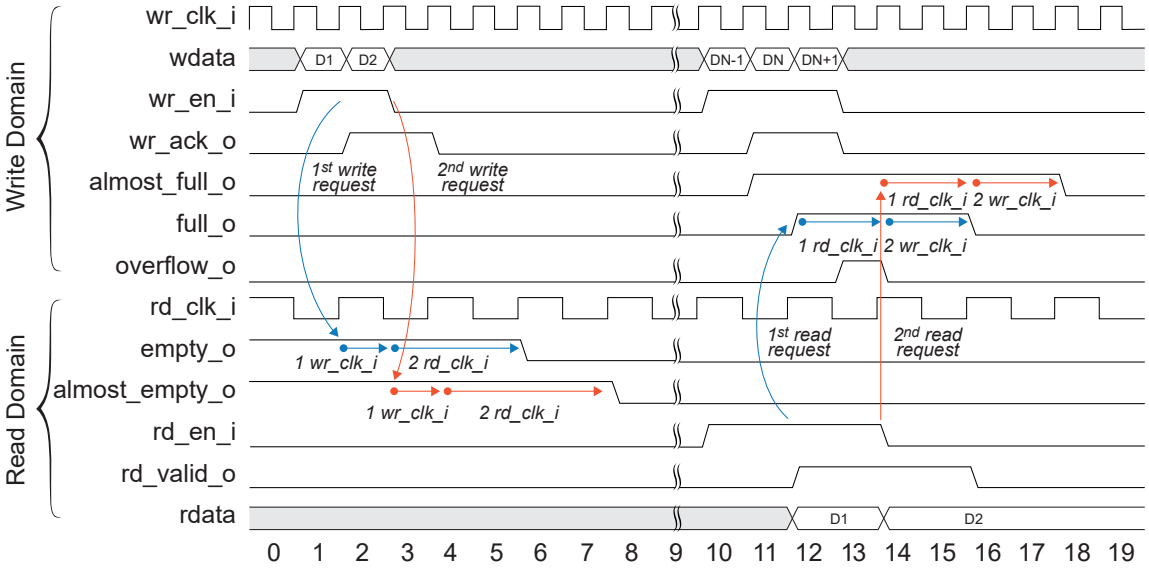
In the read example shown in **Figure 6: Asynchronous FIFO Standard Mode Faster Read Clock with PIPELINE_REG=0** on page 12, the read clock frequency is double that of the write clock with the same phase. When there is a write request at node 2, `empty_o` does not deassert immediately; instead, it deasserts 1 write clock plus 2 clock read clocks later at node 6. Similarly, `almost_empty_o` deasserts at node 8, which is 1 write clock plus 2 read clocks later after the second write request at node 4. `almost_full_o` and `full_o` deassert at the same time at node 22 because there are 2 read requests detected before the write domain is synchronized at node 20.

Figure 6: Asynchronous FIFO Standard Mode Faster Read Clock with PIPELINE_REG=0



In the write example shown in **Figure 7: Asynchronous FIFO Standard Mode Faster Write Clock with PIPELINE_REG=0** on page 12, the write clock frequency is double that of the read clock with the same phase. The `empty_o` deasserts at node 5 and `almost_empty_o` deasserts at node 7. Each of these signals are affected by write requests on node 1 and node 2 respectively. Read requests at node 11 and 13 reflect on the write domain at node 15 and 17, respectively.

Figure 7: Asynchronous FIFO Standard Mode Faster Write Clock with PIPELINE_REG=0

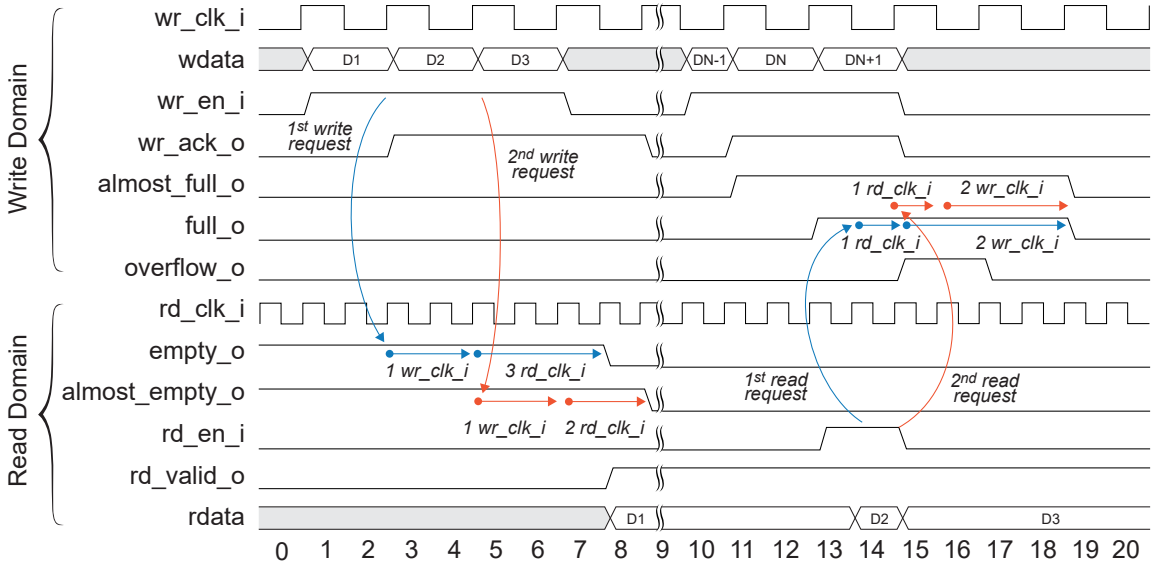


FWFT Mode

The following figures show example of asynchronous FIFO FWFT mode with faster read clock and faster write clock. Both examples have the similar read request to write flags update behavior as their standard mode counterpart. The write request to `empty_o` delay of synchronous FIFO FWFT applies here as well, just that the additional clock is of the read clock.

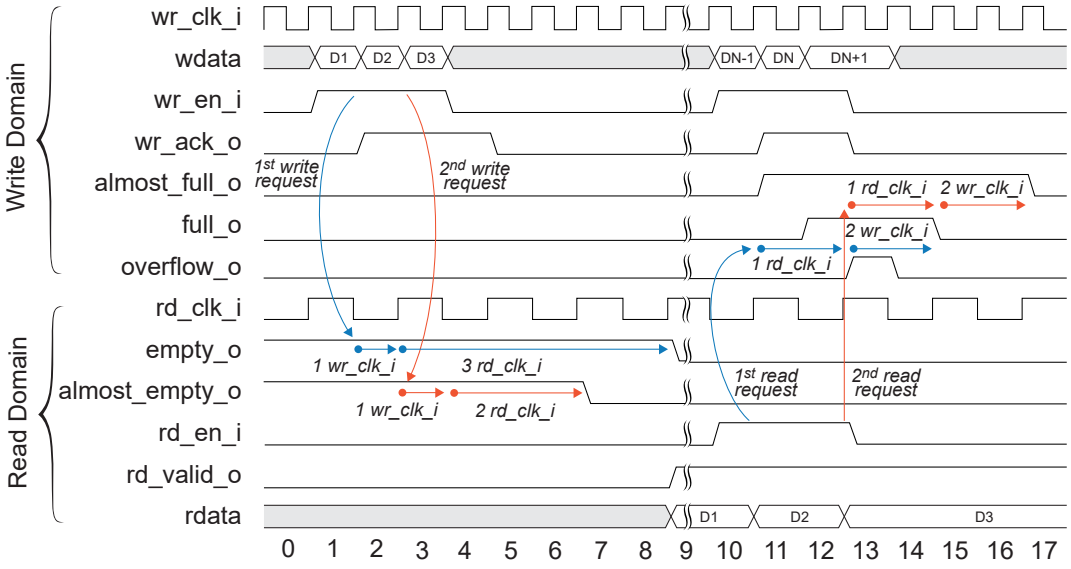
In the example shown in**Figure 8: Asynchronous FIFO FWFT Mode Faster Read Clock with PIPELINE_REG=0** on page 13, the read clock frequency is double that of the write clock with the same phase. When there is a write request at node 2, `empty_o` does not deassert immediately; instead, it deasserts 1 write clock plus 3 clocks later at node 7, which has one additional clock cycle latency compared to standard mode. Concurrently, the `empty_o` deasserts, the first data falls through the FIFO onto `rdata`, and the `rd_valid_o` is asserted. The `almost_empty_o` behaves the same as standard mode whereby it only needs 1 write clocks plus 2 clocks to deasserts at node 8, after the second write request at node 4. Subsequent read request outputs the next available word inside FIFO.

Figure 8: Asynchronous FIFO FWFT Mode Faster Read Clock with PIPELINE_REG=0



In the example shown in **Figure 9: Asynchronous FIFO FWFT Mode Faster Write Clock with PIPELINE_REG=0** on page 14, the write clock frequency is double that of the read clock with the same phase. Between positive edges of read clock at node 2 and node 4, 2 write requests are detected at the same time. The `empty_o` deasserts 3 clock cycles later at node 8, while `almost_empty_o` only requires 2 clock cycles to deassert at node 6. This means that the FIFO read domain detected 2 write words at node 6, however it is not ready for reading as the `empty_o` remains asserted. The first word only falls through at the same time as `empty_o` is deasserted and `rd_valid_o` is asserted. Always refer to `empty_o` instead of `datacount_o` value whenever you want to do a read request. Refer to the **Datacount** on page 18 for more information about the `datacount_o` signal.

Figure 9: Asynchronous FIFO FWFT Mode Faster Write Clock with PIPELINE_REG=0



Asymmetric Width Operation

Asymmetric aspect ratios allow the input and output of the FIFO width and depth to be configured differently. You only need to configure the write width and depth, while the read width and read depth are computed automatically by the FIFO based on your parameter settings. The following table lists the supported asymmetric width ratio.

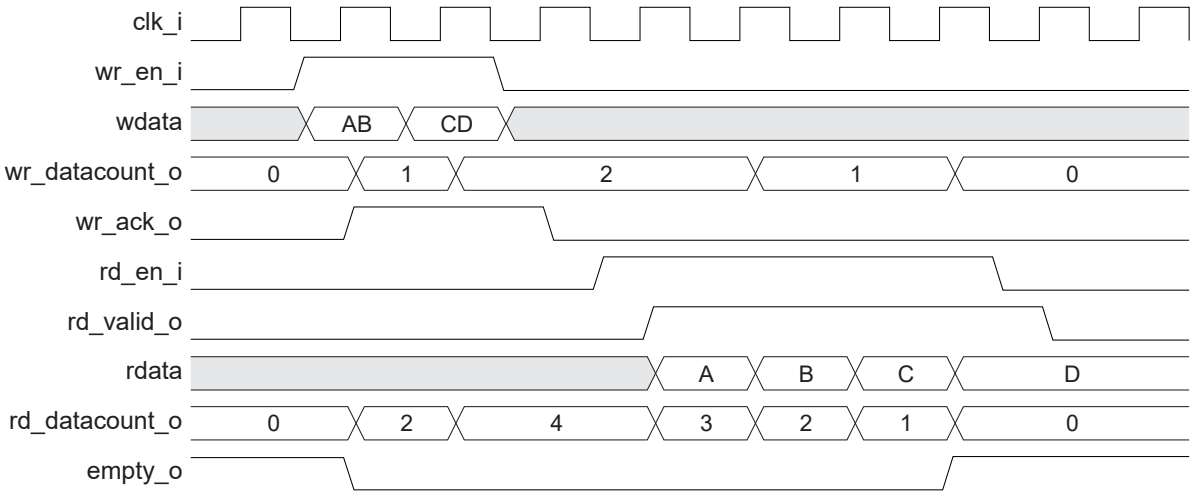
Note: The write width must be divisible by the selected ratio. For example, for 8:1 ratio, the write width can be 8, 16, 32, up to 256.

Table 9: Supported Asymmetric Width FIFO Ratio

Ratio	Write Width	Read Width	Write Depth	Read Depth
16:1	N	N/16	2^M	2^M x 16
8:1	N	N/8	2^M	2^M x 8
4:1	N	N/4	2^M	2^M x 4
2:1	N	N/2	2^M	2^M x 2
1:1	N	N	2^M	2^M
1:2	N	N*2	2^M	2^M / 2
1:4	N	N*4	2^M	2^M / 4
1:8	N	N*8	2^M	2^M / 8
1:16	N	N*16	2^M	2^M / 16

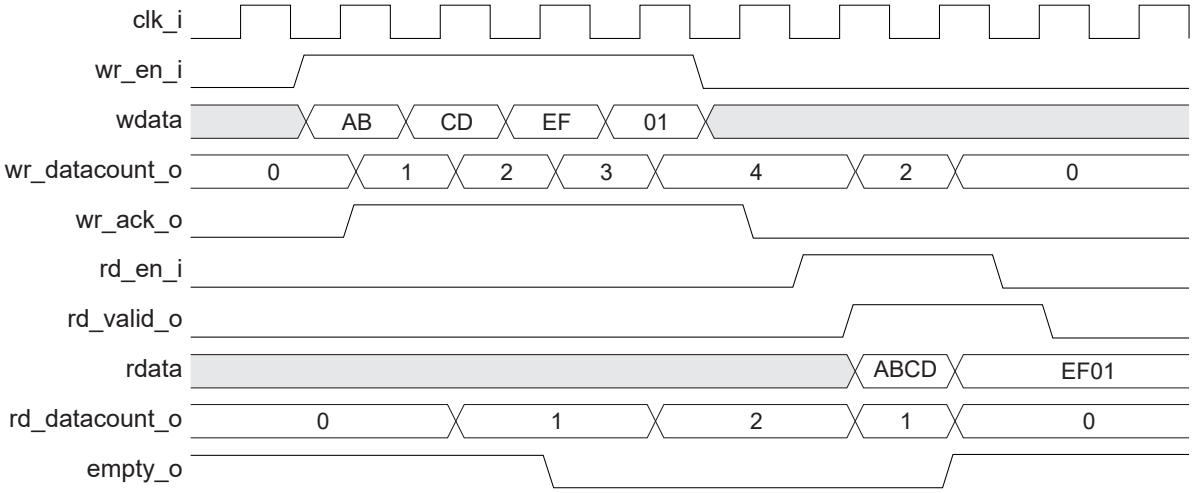
In operations with 2:1 aspect ratio, the write width is two times the read width. In the example below, each write request has 8-bit data which requires 2 read requests (4-bit width per clock cycle) to free-up the entry.

Figure 10: 2:1 Aspect Ratio Example Waveform



In operations with 1:2 aspect ratio, the read width is two times the write width. In the example below, each write request has 8-bit data where two write requests are required to contribute to a single read word (16-bit width).

Figure 11: 1:2 Aspect Ratio Example Waveform



For asymmetric width operation, the full and empty flags are active only when one complete word can be read or written. Therefore, accessing partial words is not allowed. For example, assuming a full FIFO, if the write width is 8 bits and read width is 4 bits, two valid read operations are needed before full de-asserts and a write operation is accepted.

Programmable Full and Empty Signals

The FIFO core supports user-defined full and empty signals with customized depths (`prog_full_o` and `prog_empty_o`). To enable these signals, set the `PROGRAMMABLE_FULL` or `PROGRAMMABLE_EMPTY` parameters as `STATIC_SINGLE` or `STATIC_DUAL`. Refer to **Parameters** for more info on the available values.

Important: For the asynchronous FIFO, these signals are synchronized to their respective clock domain's available words.

Table 10: prog_full_o Assert and Deassert Conditions

Value	Type	Condition
STATIC_SINGLE	Assert	*number of words in FIFO* ≥ PROG_FULL_ASSERT
	Deassert	*number of words in FIFO* < PROG_FULL_ASSERT
STATIC_DUAL	Assert	*number of words in FIFO* ≥ PROG_FULL_ASSERT
	Deassert	*number of words in FIFO* < PROG_FULL_NEGATE

Table 11: prog_empty_o Assert and Deassert Conditions

Value	Type	Condition
STATIC_SINGLE	Assert	*number of words in FIFO* ≤ PROG_EMPTY_ASSERT
	Deassert	*number of words in FIFO* > PROG_EMPTY_ASSERT
STATIC_DUAL	Assert	*number of words in FIFO* ≤ PROG_EMPTY_ASSERT
	Deassert	*number of words in FIFO* > PROG_EMPTY_NEGATE

To avoid erratic behavior, follow these rules for `STATIC_DUAL` modes:

- `PROG_FULL_ASSERT` ≥ `PROG_FULL_NEGATE`
- `PROG_EMPTY_ASSERT` ≤ `PROG_EMPTY_NEGATE`

Reset

The FIFO core uses active high asynchronous reset. By default, the reset signal (`a_rst_i`) is synchronized to the respective clock domains before it being used in the core logic. You must ensure that the `rst_busy` signal is low before the start any of the FIFO operation.

If the reset synchronization is already included in the user logic, you can bypass the reset synchronizer logic in FIFO core by setting the `SKIP_RESET_SYNC` parameter to value 1. In this scenario, you should directly connect `a_wr_rst_i` and `a_rd_rst_i` ports.

Datacount

The FIFO core includes datacount signal as output. Synchronous FIFO enables `datacount_o` while asynchronous FIFO enables both `wr_datacount_o` and `rd_datacount_o`.

The datacount is zero when the FIFO is in empty and full state. You must ensure that the width of datacount is $\log_2$(`DEPTH`) to get the correct value.

Note: Always refer to the `empty_o` and `full_o` signals when reading or writing FIFO.

Latency

This section defines the latency of the output signals in the FIFO core. The output signals latency are updated in response to the read or write requests. Latency is described in the following waveform. A 0 latency means the signal is asserted or deasserted at the same rising edge of the clock at which the write or read request is sampled. A latency of 1 means the signal is asserted or deasserted at the next rising edge of the clock.

Synchronous FIFO

Table 12: Synchronous FIFO Write Flags Update Latency (clk_i) Due to `wr_en_i` and `rd_en_i` Signals

Port	wr_en_i	rd_en_i
wr_ack_o	0	-
full_o	0	0
almost_full_o	0	0
prog_full_o	0	0
overflow_o	0	-

Table 13: Synchronous FIFO Read Flags Update Latency Due to `wr_en_i` and `rd_en_i` Signals

Port	wr_en_i	rd_en_i
rd_valid_o	-	0[5]
empty_o	0	0
almost_empty_o	0	0
prog_empty_o	0	0
underflow_o	-	0
datacount_o	0	0

[5] OUTPUT_REG adds one latency to these signal.

Asynchronous FIFO

Table 14: Asynchronous FIFO Write Flags Update Latency Due to wr_en_i

Port	Latency (PIPELINE_REG=0)	Latency (PIPELINE_REG=1)
wr_ack_o	0	0
full_o	0	0
almost_full_o	0	0
prog_full_o	0	0
overflow_o	0	0
wr_datacount_o	0	0

Table 15: Asynchronous FIFO Read Flags Update Latency Due to wr_en_i

Port	Latency (PIPELINE_REG=0)	Latency (PIPELINE_REG=1)
rd_valid_o	-	-
empty_o	1 wr_clk_i + 2 rd_clk_i	1 wr_clk_i + 3 rd_clk_i
almost_empty_o	1 wr_clk_i + 2 rd_clk_i	1 wr_clk_i + 3 rd_clk_i
prog_empty_o	1 wr_clk_i + 2 rd_clk_i	1 wr_clk_i + 3 rd_clk_i
underflow_o	-	-
rd_datacount_o	1 wr_clk_i + 2 rd_clk_i	1 wr_clk_i + 3 rd_clk_i

Table 16: Asynchronous FIFO Write Flags Update Latency Due to rd_en_i

Port	Latency (PIPELINE_REG=0)	Latency (PIPELINE_REG=1)
wr_ack_o	-	-
full_o	1 rd_clk_i + 2 wr_clk_i	1 rd_clk_i + 3 wr_clk_i
almost_full_o	1 rd_clk_i + 2 wr_clk_i	1 rd_clk_i + 3 wr_clk_i
prog_full_o	1 rd_clk_i + 2 wr_clk_i	1 rd_clk_i + 3 wr_clk_i
overflow_o	-	-
wr_datacount_o	1 rd_clk_i + 2 wr_clk_i	1 rd_clk_i + 3 wr_clk_i

Table 17: Asynchronous FIFO Read Flags Update Latency Due to rd_en_i

Port	Latency (PIPELINE_REG=0)	Latency (PIPELINE_REG=1)
rd_valid_o	0[6]	0[7]
empty_o	0	0
almost_empty_o	0	0
prog_empty_o	0	0
underflow_o	0	0
rd_datacount_o	0	0

[6] OUTPUT_REG adds one latency to these signal.
[7] OUTPUT_REG adds one latency to these signal.

IP Manager

The Efinity® IP Manager is an interactive wizard that helps you customize and generate Efinix® IP cores. The IP Manager performs validation checks on the parameters you set to ensure that your selections are valid. When you generate the IP core, you can optionally generate an example design targeting an Efinix development board and/or a testbench. This wizard is helpful in situations in which you use several IP cores, multiple instances of an IP core with different parameters, or the same IP core for different projects.

Note: Not all Efinix IP cores include an example design or a testbench.

Generating the FIFO Core with the IP Manager

The following steps explain how to customize an IP core with the IP Configuration wizard.

1. Open the IP Catalog.
2. Choose **Memory > FIFO** core and click **Next**. The **IP Configuration** wizard opens.
3. Enter the module name in the **Module Name** box.

   Note: You cannot generate the core without a module name.
4. Customize the IP core using the options shown in the wizard. For detailed information on the options, refer to the *Customizing the FIFO* section.
5. (Optional) In the **Deliverables** tab, specify whether to generate an IP core example design targeting an Efinix® development board and/or testbench. These options are turned on by default.
6. (Optional) In the **Summary** tab, review your selections.
7. Click **Generate** to generate the IP core and other selected deliverables.
8. In the **Review configuration generation** dialog box, click **Generate**. The Console in the **Summary** tab shows the generation status.

   Note: You can disable the **Review configuration generation** dialog box by turning off the **Show Confirmation Box** option in the wizard.
9. When generation finishes, the wizard displays the **Generation Success** dialog box. Click **OK** to close the wizard.

The wizard adds the IP to your project and displays it under **IP** in the Project pane.

Generated Files

The IP Manager generates these files and directories:

- **<module name>_define.vh**—Contains the customized parameters.
- **<module name>_tmpl.v**—Verilog HDL instantiation template.
- **<module name>_tmpl.vhd**—VHDL instantiation template.
- **<module name>.v**—IP source code.
- **settings.json**—Configuration file.
- **<kit name>_devkit**—Has generated RTL, example design, and Efinity® project targeting a specific development board.
- **Testbench**—Contains generated RTL and testbench files.

Customizing the FIFO

The core has parameters so you can customize its function. You set the parameters in the General tab of the core's IP Configuration window.

Table 18: FIFO Core Parameter

Parameter	Options	Description
Device Family	Trion , Titanium	Select the target device family. Default: Trion
Clock Mode	Asynchronous, Synchronous	Defines whether the FIFO read and write domain is synchronous or asynchronous. Default: Asynchronous
FIFO Depth	16 – 131072	Defines the FIFO depth, which determines the maximum number of words the FIFO can store before it is full. The depth is multiples of 2 from 16 – 2^{17}. Default: 512
Data Bus Width	1 – 256	Defines the FIFO's read and write data bus widths. Default: 16
FIFO Mode	STANDARD, FWFT	Defines the FIFO's read mode as standard or FWFT. Default: STANDARD
Output Register	Enable, Disable	Adds one pipeline stage to rdata and rd_valid_o to improve timing delay out from efx_ram. Default: 0 (Disable)
Programmable Full Assert Value	4 - (FIFO Depth - 2)	Threshold value when prog_full_o is enabled. When Enable Programmable Full Option is: STATIC_SINGLE: Single threshold value for assertion and deassertion of prog_full_o. STATIC_DUAL: Upper threshold value for assertion of prog_full_o. Default: 128
Enable Programmable Full Option	NONE, STATIC_SINGLE, STATIC_DUAL	Controls the prog_full_o signal: NONE: Disabled. STATIC_SINGLE: Enabled, asserts and deasserts at a single threshold value. (default) STATIC_DUAL: Enabled, asserts or deasserts at different threshold values.
Programmable Full Negate Value	3 - (FIFO Depth - 3)	Use when PROGRAMMABLE_FULL is set to STATIC_DUAL. Sets the lower threshold value for prog_full_o during deassertion. Default: 127
Programmable Empty Assert Value	2 - (FIFO Read Depth - 4)	Threshold value when prog_empty_o is enabled. When Enable Programmable Empty Option is: STATIC_SINGLE: Single threshold value for assertion and deassertion of prog_empty_o. STATIC_DUAL: Lower threshold value for assertion of prog_empty_o. Default: 2

Parameter	Options	Description
Programmable Empty Negate Value	3 - (FIFO Read Depth - 3)	Use when PROGRAMMABLE_EMPTY is set to STATIC_DUAL. Sets the upper threshold value for prog_empty_o during deassertion. Default: 3
Enable Programmable Empty Option	NONE, STATIC_SINGLE, STATIC_DUAL	Controls the prog_empty_o signal: NONE: Disabled (default). STATIC_SINGLE: Enabled, asserts and deasserts at a single threshold value. STATIC_DUAL: Enabled, asserts or deasserts at different threshold values.
Optional Signals	Enable, Disable	Enables the optional signals: wr_ack_o, almost_full_o, , rd_valid_o, and almost_empty_o. You can enable this feature with some trade-offs in timing performance. However, this feature must be enabled when generating the Example Designs or Testbench. Default: Disable
Pipeline Register	Enable, Disable	Applicable to asynchronous FIFO mode only. Adds one latency of the opposing clock domain to all applicable output signals when wr_en_i or rd_en_i signal is asserted. Enable this feature to improve the macro timing. Efinix recommends that you enable this parameter in asynchronous FIFO mode. Default: Enable
Synchronization Stages	2 – 8	Configures the number of synchronization stages for the cross clock domain signals in asynchronous mode. This increases the latency of opposing clock domain status flag signals. Default: 2
Asymmetric Width Ratio	16:1, 8:1, 4:1, 2:1, 1:1, 1:2, 1:4, 1:8, 1:16	Selects asymmetrical width ratios. 1:1 is symmetric width ratio. Default: 1:2
Reset Synchronizer	Enable, Disable	Disable if you do not want the reset signal to be synchronized to the respective clock domain during asynchronous mode. Ensure that the supplied reset signal is synchronized to the respective FIFO clock domain in design upper level order for the FIFO reset to operate correctly. Default: Enable
Endianness	BIG_ENDIAN, LITTLE_ENDIAN	Select the order in which the bytes are stored and read out first. Write width > Read width: • BIG_ENDIAN: The **most** significant portion of the *wdata* is stored into the FIFO first and being read out first. • LITTLE_ENDIAN: The **least** significant byte of the *wdata* is stored into the FIFO first and being read out first. Read width > Write width: • BIG_ENDIAN: **Least** significant portion of the*rdata* contains the newer data. • LITTLE_ENDIAN: **Most** significant portion of the *rdata* contains the newer data. Default: BIG_ENDIAN

Parameter	Options	Description
Overflow Protection	Enable, Disable	The overflow protection disables the `wr_en_i` port when the FIFO is full. If enabled, overflowing the FIFO is not destructive to the contents of the FIFO. The `overflow_o` port is exposed when this feature is enabled. You can choose to disable this feature to achieve better timing performance. Default: Enable
Underflow Protection	Enable, Disable	The underflow protection disables the `rd_en_i` port when the FIFO is empty. If enabled, underflowing the FIFO is not destructive to the contents of the FIFO. The `underflow_o` port is exposed when this feature is enabled. You can choose to disable this feature to achieve better timing performance. Default: Enable
FIFO Implementation	Block RAM, LUT Register	Defines the FIFO RAM implementation as Block RAM or LUT Register. This is only available for FIFO Depth <= 32 Default: Block RAM

FIFO Example Design

You can choose to generate the example design when generating the core in the IP Manager Configuration window. Compile the example design project and download the **.hex** or **.bit** file to your board. To generate example design, the **Optional Signals** option must be enabled.

Important: Efinix tested the example design generated with the default parameter options only.

The example design targets the Trion® T20 BGA256 Development Board. The design demonstrates the continuous read-write operation using both symmetric and asymmetric width FIFO as well as using FIFO status signal as part of the read write control operation.

The data generator produces continuous 16-bit incremental data once the system reset is release. The 16-bit data is directly written into the asymmetric FIFO (configured as 1:2 ratio including asynchronous clock settings). The same 16-bit data goes through the data accumulator block to assemble a 32-bit data before written into the symmetric FIFO. This process is to ensure that the write and read data has a 1:1 ratio.

Both FIFO read operations are triggered only after `prog_full_o` signal of asymmetric FIFO is asserted. The programmable full threshold is set to a quarter of the total write depth. The FIFO read-write operation can run continuously without hitting FIFO full / FIFO empty due to:

- The FIFO write clock is running two times faster than the read clock
- Both write and read clock is generated from the same PLL (0 PPM)

In order to observe asymmetric FIFO full or empty behavior, you can trigger a stop read or stop write to interrupt the FIFO read / write operation through the pushbuttons.

Figure 12: FIFO Example Design

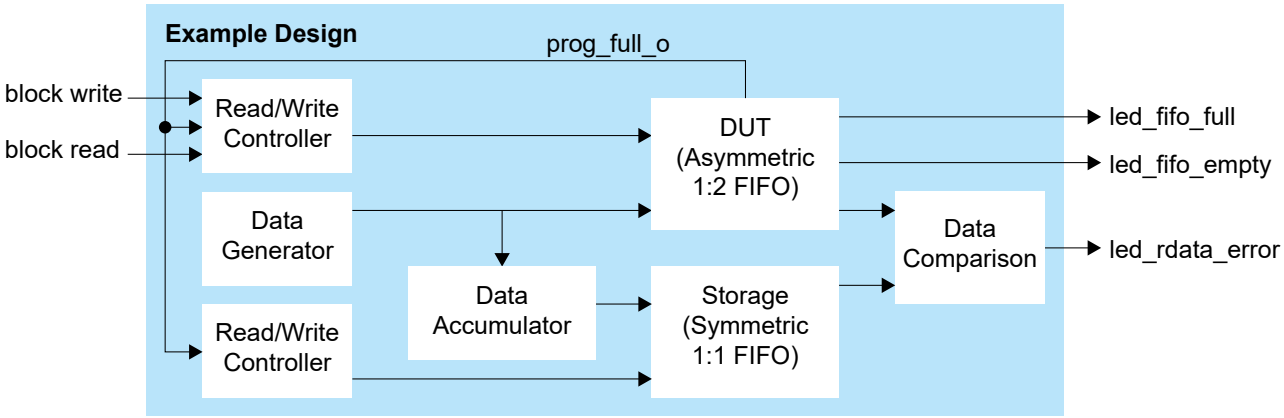


Table 19: Example Design Input and Output

Input / Output	Description
LED D3	Upon power-up, LED D3 blinks continuously to indicate that the design is running on the board.
LED D4	Turns on when there is read data error during comparison. Pressing SW5 / SW6 button can also cause read data comparison error.
LED D5	Turns on when asymmetric FIFO is full. Occurs when pressing SW6 pushbutton.
LED D6	Turns on when asymmetric FIFO is empty. Occurs when pressing SW5 pushbutton.
Pushbutton SW4	System reset. Use system reset to clear read comparison error.
Pushbutton SW5	Stop write. Triggers a stop write and causes the asymmetric FIFO to hit full status.
Pushbutton SW6	Stop read. Triggers a stop read and causes the asymmetric FIFO to hit empty status.

Table 20: Titanium Asynchronous Example Design Implementation

FPGA	Mode	Logic Elements (Logic, Adders, Flipflops, etc.)	Memory Block	DSP Block	f_{MAX} (MHz)[10]		Efinity Version[11]
					wr_clk_i	rd_clk_i	
Ti60 F225 C4	Standard	334/60800 (0.07%)	4/256 (2%)	0/160 (0%)	416.84	262.74	2023.2
	FWFT	351/60800 (0.1%)	4/256 (2%)	0/160 (0%)	431.22	259.67	

Table 21: Trion® Example Design Implementation

FPGA	Mode	Logic Elements (Logic, Adders, Flipflops, etc.)	Memory Block	DSP Block	f_{MAX} (MHz)[10]		Efinity Version[11]
					wr_clk_i	rd_clk_i	
T20 F256 C4	Standard	354/19728(2%)	2/204 (3%)	0/36 (0%)	155.06	134.48	2023.2
	FWFT	371/19728 (2%)	2/204 (3%)	0/36 (0%)	157.73	117.63	

(10) Using default parameter settings.
(11) Using Verilog HDL.

FIFO Testbench

You can choose to generate the testbench when generating the core in the IP Manager Configuration window. To generate testbench, the **Optional Signals** option must be enabled.

Note: You must include all **.v** files generated in the **/testbench** directory in your simulation.

Efinix provides a simulation script for you to run the testbench quickly using the Modelsim software. To run the Modelsim testbench script, run `vsim -do modelsim.do` in a terminal application. You must have Modelsim installed on your computer to use this script.

Revision History

Table 22: Revision History

Date	Version	Description
February 2024	2.0	Updated Features section. (DOC-1704) Updated Resource Utilization and Performance section. Updated Table FIFO Core Clock, Reset, and Data,FIFO Core Write Ports, FIFO Core Read Ports in Ports topic. Added additional information in Synchronous FIFO Operation and Asymmetric Width Operation section. Updated Table FIFO Core Parameter in Customizing the FIFO section. Updated Table Titanium Asynchronous Example Design Implementation and Trion Example Design Implementation in FIFO Example Design. Also, added a statement in the first paragraph. Updated figure FIFO System Block Diagram, Synchronous FIFO Block Diagram, and Asynchronous FIFO Block Diagram.
January 2024	1.9	Added in extra information in Assymetric Width Operation section. (DOC-1621) Updated Table: FIFO Core Parameter in Customizing the FIFO section. Corrected figure FIFO System Block Diagram in Functional Description section.
October 2023	1.8	Added description for wr_datacount_o, rd_datacount_o and datacount_o port. (DOC-1513)
February 2023	1.7	Added note about the resource and performance values in the resource and utilization table are for guidance only.
January 2023	1.6	Corrected reset signal name.
August 2022	1.5	Removed description about reset pulse width requirement. (DOC-903)
April 2022	1.4	Corrected supported data width in feature list.

Date	Version	Description
January 2022	1.3	Updated resource utilization table and Asymmetric Width Ratio parameter options. (DOC-700)
December 2021	1.2	Core included in main Efinity release.
October 2021	1.1	Added note to state that the f_{MAX} in Resource Utilization and Performance, and Example Design Implementation tables were based on default parameter settings. Corrected the Titanium FPGA used in Resource Utilization and Performance tables.
September 2021	1.0	Initial release.