FIFO Core User Guide

UG-CORE-FIFO-v4.0
June 2021
www.efinixinc.com

Contents

Introduction

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 1024 bits
- Symmetric read–to–write port aspect ratio
- 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
- Verilog RTL and simulation testbench
- Includes example designs targeting the Trion® T20 BGA256 Development Board and Titanium Ti60 F225 Development Board
- Asynchronous clock domain FWFT read mode
- FIFO datacount to indicate how many words available in FIFO
- Recommended clock frequency for the FIFO core running C4 grade device with default setting is up to 125 MHz
- Option to exclude optional flags

New in Efinity® v2021.1

- Added Titanium FPGA support

FPGA Support

The FIFO core supports all Trion® and Titanium FPGAs.

Titanium Resource Utilization and Performance

To achieve the performance shown in the following tables, ensure that all inputs to the FIFO are registered and the outputs passed through a minimal number of logic levels.

Table 1: Synchronous Clock FIFO (DEPTH = 512, DATA_WIDTH = 32)

125 MHz clock constraint.

FPGA	Read Mode	Logic and Adders	Flipflops	Memory Blocks	DSP48 Blocks	f_{MAX} (MHz) - clk_i	Efinity® Version[1]
Ti60 F225 ES C3	Standard	100	66	2	0	630	2021.1
	FWFT	108	67	2	0	615	

Table 2: Asynchronous Clock FIFO (DEPTH = 512, DATA_WIDTH = 32)

125 MHz clock constraint.

FPGA	Read Mode	Logic and Adders	Flipflops	Memory Blocks	DSP48 Blocks	f_{MAX} (MHz) -		Efinity® Version[1]
						wr_clk_i	rd_clk_i	
Ti60 BGA225 ES C3	Standard	137	139	2	0	550	562	2021.1
	FWFT	166	143	2	0	591	604	

Trion Resource Utilization and Performance

To achieve the performance shown in the following tables, ensure that all inputs to the FIFO are registered and the outputs passed through a minimal number of logic levels.

Table 3: Synchronous Clock FIFO (DEPTH = 512, DATA_WIDTH = 32)

125 MHz clock constraint.

FPGA	Read Mode	Logic Utilization (LUTs)	Memory Blocks	f_{MAX} (MHz) - clk_i	Efinity® Version[1]
T8 BGA81 C2	Standard	98	4	62.	2020.1
	FWFT	110		65	
T20 BGA256 C3	Standard	98		168	
	FWFT	110		152	
T35 BGA324 C4	Standard	98		184	
	FWFT	110		173	
T120 BGA324 C4	Standard	98		194	
	FWFT	110		166	

[1] Using Verilog HDL.

Table 4: Asynchronous Clock FIFO (DEPTH = 512, DATA_WIDTH = 32)

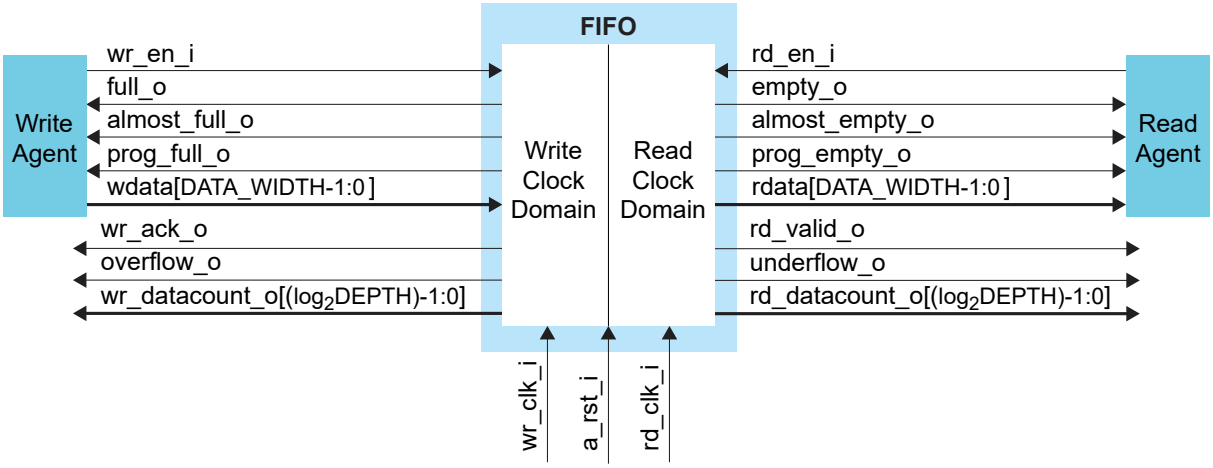
125 MHz clock constraint.

FPGA	Read Mode	LUTs	Memory Blocks	f_{MAX} (MHz) -		Efinity® Version[1]
				wr_clk_i	rd_clk_i	
T8 BGA81 C2	Standard	183	4	65	68	2020.1
	FWFT	214		69	61	
T20 BGA256 C3	Standard	183		151	166	
	FWFT	214		154	144	
T35 BGA324 C4	Standard	183		207	179	
	FWFT	214		174	183	
T120 BGA324 C4	Standard	183		196	188	
	FWFT	214		203	190	

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® 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 5: 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.
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*-1:0]		✓	Output	Asynchronous FIFO write domain data count. n=$\log_2$[DEPTH].
rd_datacount_o [*n*-1:0]		✓	Output	Asynchronous FIFO read domain data count. n=$\log_2$[DEPTH].
datacount_o [*n*-1:0]	✓		Output	Synchronous FIFO data count. n=$\log_2$[DEPTH].

Table 6: 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. 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 indicates that a write request (wr_en_i) during the prior clock cycle was rejected because the FIFO buffer is full. Overflowing the FIFO is not destructive to the contents of the FIFO.

Table 7: FIFO Core Read Ports

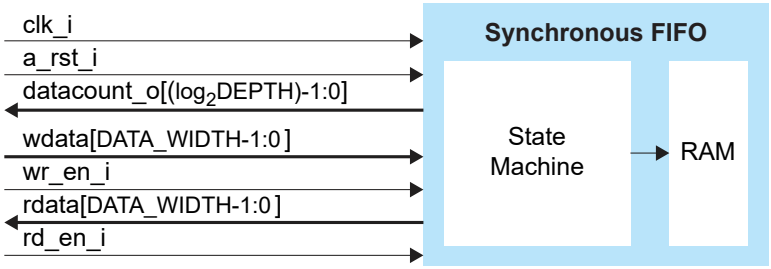
For both synchronous and asynchronous clocks.

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. 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. Indicates that the read request (rd_en_i) during the previous clock cycle was rejected because the FIFO buffer is empty. 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.

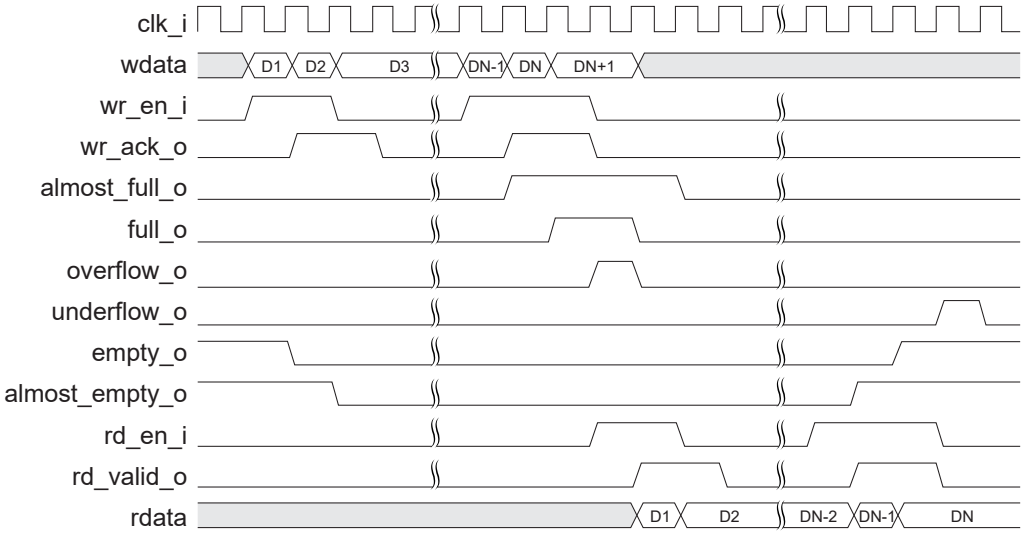
Figure 2: Synchronous FIFO Block Diagram



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



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`.

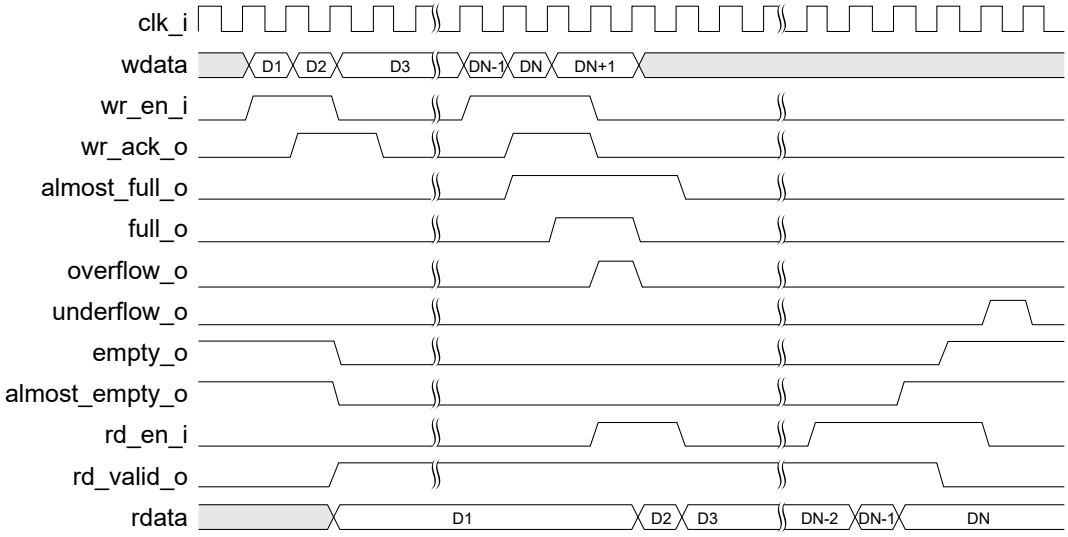
First-Word-Fall-Through Mode

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.

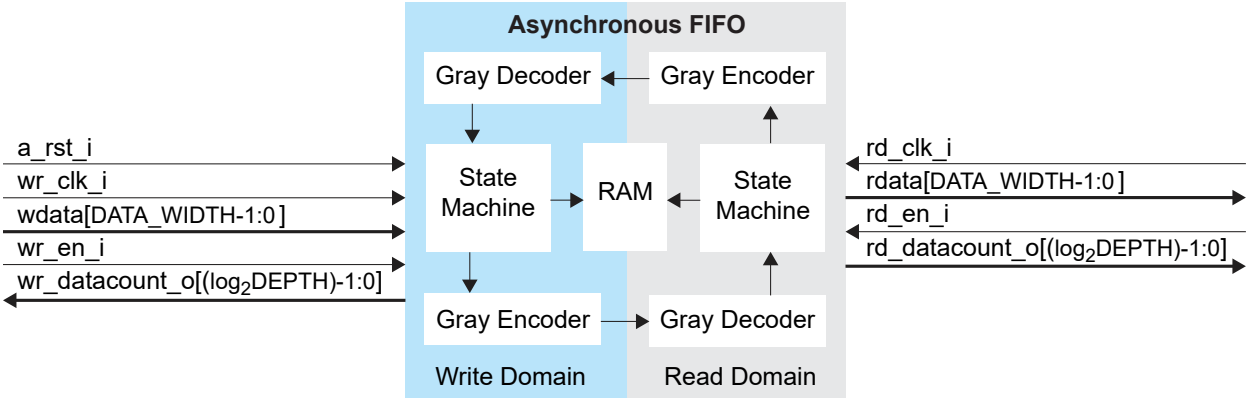
Figure 4: Synchronous FIFO FWFT Mode Waveform



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 on page 15 for more info.

Figure 5: Asynchronous FIFO Block Diagram



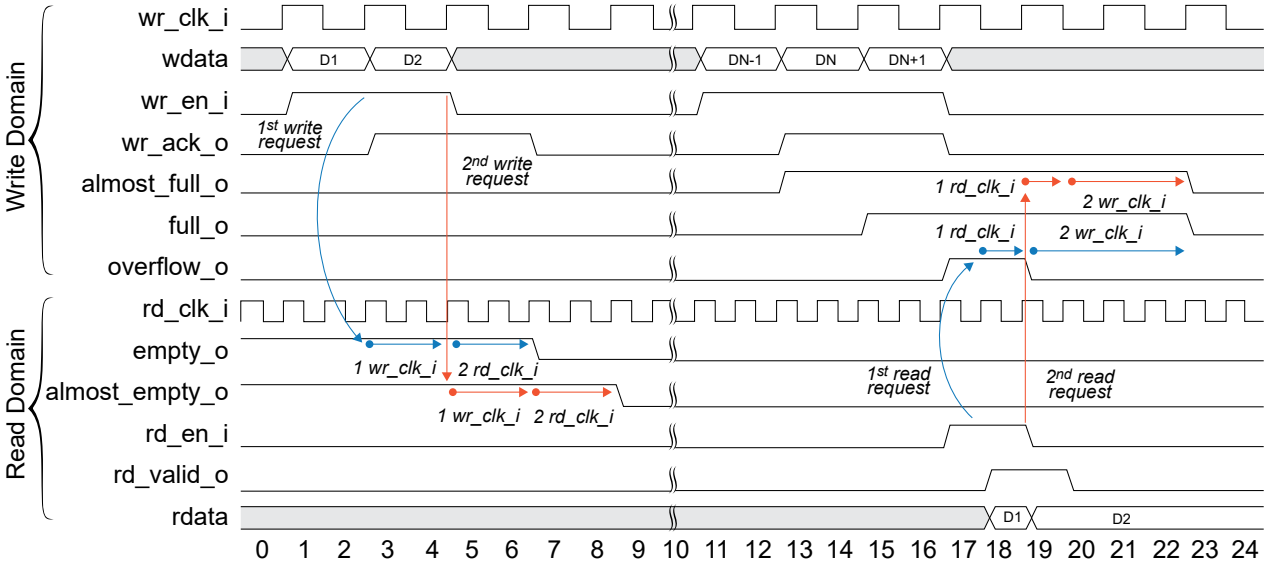
For asynchronous FIFO, a write operation affecting the write domain flags and a read operation affecting the read domain flags have the same behavior as the synchronous FIFO except when they are affecting crossed domain flags. The following examples emphasize the cross-clock domain flags update latency.

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.

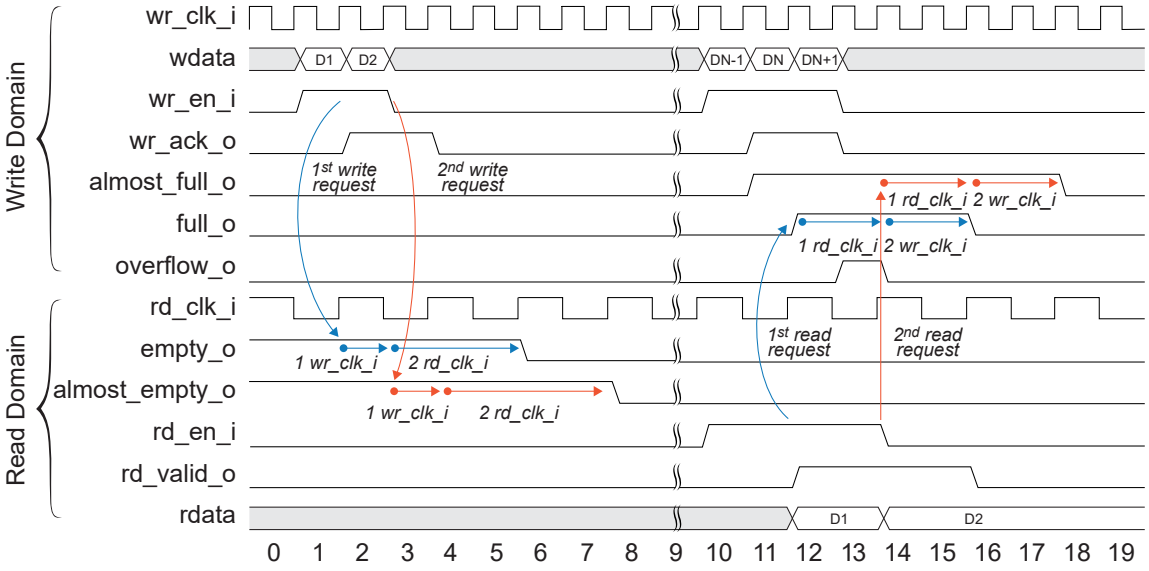
In the read example shown in **Figure 6: Asynchronous FIFO Standard Mode Faster Read Clock with PIPELINE_REG=0** on page 11, the read clock frequency is double that of the write clock with the same phase. When there is a write request at node 2, `empty_o` does not deassert immediately; instead, it deasserts 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 11, 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

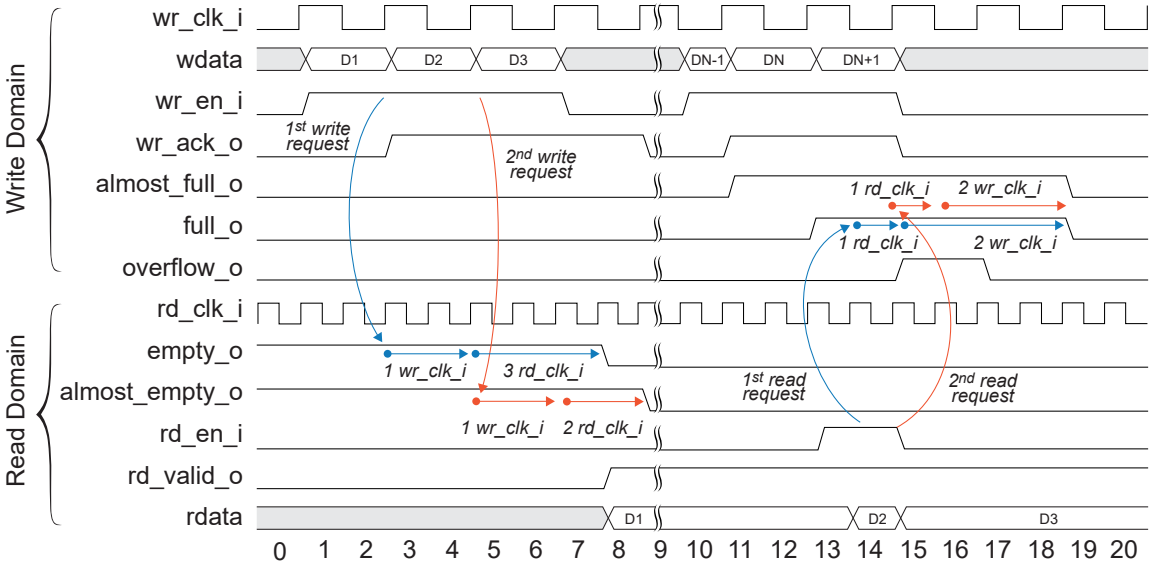


FWFT Mode

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.

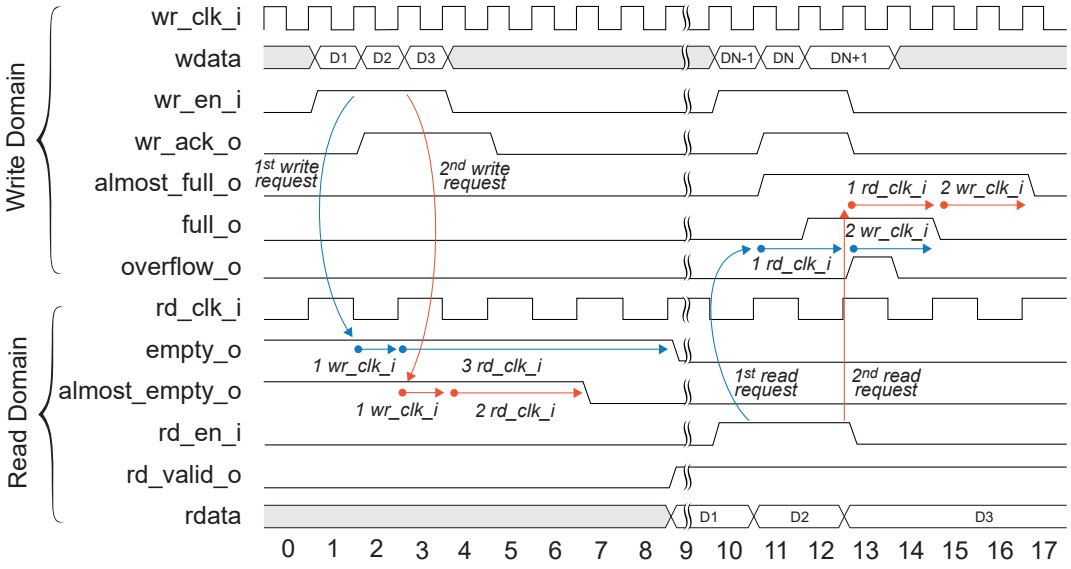
In the example shown in **Figure 8: Asynchronous FIFO FWFT Mode Faster Read Clock with PIPELINE_REG=0** on page 12, the read clock frequency is double that of the write clock with the same phase. When there is a write request at node 2, `empty_o` does not deassert immediately; instead, it deasserts 1 write clock plus 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 13, 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 14 for more information about the `datacount_o` signal.

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



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 8: 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 9: 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 includes an asynchronous reset signal, `a_rst_i`, which is active high. If your design requires an active low reset signal, invert the source before passing it to the input of `a_rst_i`. Efinix recommends you keep the reset signal high for at least 5 clock cycles of the slowest clock to ensure correct operation. During reset, you must deassert `wr_en_i` and `rd_en_i`.

The following figures show the reset behavior for the synchronous and asynchronous FIFO, respectively. Write requests during reset are ignored. After `a_rst_i` is deasserted, writes can occur. Reset does not initialize `rdata`. The `rdata` remains the same as before reset until the next valid read request for standard mode, or the next valid write request for FWFT mode.

Figure 10: Synchronous FIFO Standard Mode Reset

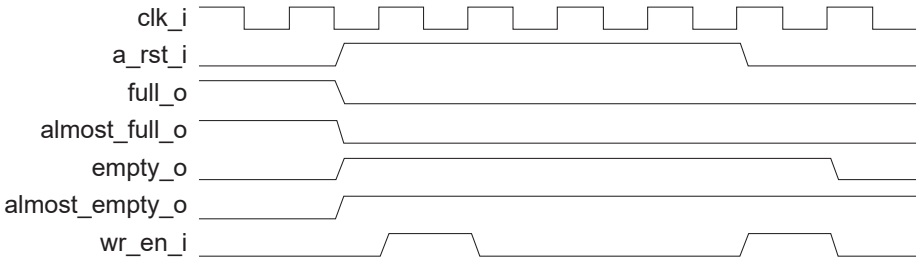
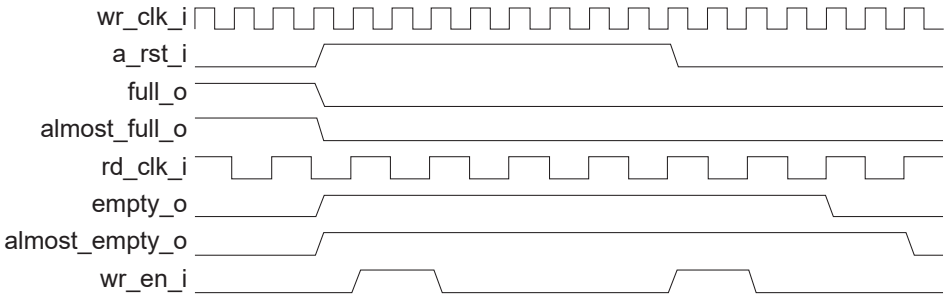


Figure 11: Asynchronous FIFO Standard Mode Reset



Note: These waveforms are examples to illustrate the reset functionality. For optimized behavior, do not assert `wr_en_i` during reset.

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`.

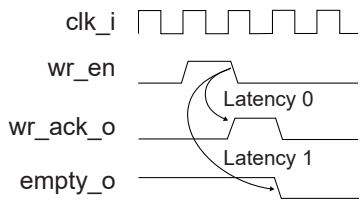
The datacount is zero when the FIFO is in empty and full state. You must ensure that the width of datacount is $\log_2$(`DEPTH`) to get the correct value.

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.

Figure 12: Latency Example Synchronous FIFO FWFT Mode



Synchronous FIFO

Table 10: Synchronous FIFO Write Flags Update Latency Due to `wr_en_i` Signal

Port	Latency (clk_i)	
	Standard Mode	FWFT Mode
wr_ack_o	0	0
full_o	0	0
almost_full_o	0	0
prog_full_o	0	0
overflow_o	0	0

Table 11: Synchronous FIFO Read Flags Update Latency Due to `wr_en_i` Signal

Port	Latency (clk_i)	
	Standard Mode	FWFT Mode
rd_valid_o	-	-
empty_o	0	1
almost_empty_o	0	0
prog_empty_o	0	0
underflow_o	-	-
datacount_o	0	0

Table 12: Synchronous FIFO Write Flags Update Latency Due to `rd_en_i` Signal

Port	Latency (clk_i)	
	Standard Mode	FWFT Mode
wr_ack_o	–	–
full_o	0	0
almost_full_o	0	0
prog_full_o	0	0
overflow_o	–	–

Table 13: Synchronous FIFO Read Flags Update Latency Due to `rd_en_i` Signal

Port	Latency (clk_i)	
	Standard Mode	FWFT Mode
rd_valid_o	0[2]	0[2]
empty_o	0	0
almost_empty_o	0	0
prog_empty_o	0	0
underflow_o	0	0
datacount_o	0	0

Asynchronous FIFO

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

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

[2] OUTPUT_REG adds one latency to these signal.

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

Port	Latency (PIPELINE_REG=0)		Latency (PIPELINE_REG=1)	
	Standard Mode	FWFT Mode	Standard Mode	FWFT Mode
rd_valid_o	–	–	–	–
empty_o	1 wr_clk_i + 2 rd_clk_i	1 wr_clk_i + 3 rd_clk_i	1 wr_clk_i + 3 rd_clk_i	1 wr_clk_i + 4 rd_clk_i
almost_empty_o	1 wr_clk_i + 2 rd_clk_i	1 wr_clk_i + 2 rd_clk_i	1 wr_clk_i + 3 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 + 2 rd_clk_i	1 wr_clk_i + 3 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 + 2 rd_clk_i	1 wr_clk_i + 3 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)	
	Standard Mode	FWFT Mode	Standard Mode	FWFT Mode
wr_ack_o	–	–	–	–
full_o	1 rd_clk_i + 2 wr_clk_i	1 rd_clk_i + 2 wr_clk_i	1 rd_clk_i + 3 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 + 2 wr_clk_i	1 rd_clk_i + 3 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 + 2 wr_clk_i	1 rd_clk_i + 3 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 + 2 wr_clk_i	1 rd_clk_i + 3 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)	
	Standard Mode	FWFT Mode	Standard Mode	FWFT Mode
rd_valid_o	$0^{(2)}$	$0^{(2)}$	$0^{(2)}$	$0^{(2)}$
empty_o	0	0	0	0
almost_empty_o	0	0	0	0
prog_empty_o	0	0	0	0
underflow_o	0	0	0	0
rd_datacount_o	0	0	0	0

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 a 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 an IP 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. In the **General** tab, customize the IP core. For detailed information on the options, refer to the IP core's user guide or on-line help.
5. (Optional) In the **Deliverables** tab, specify whether to generate an IP core example design targeting an Efinix® development board and/or testbench. For SoCs, you can also optionally generate embedded software example code. 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.

Note: Refer to the IP Manager chapter of the Efinity® Software User Guide for more information about the Efinity® IP Manager.

Customizing the FIFO

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

Table 18: FIFO Core Parameter

Parameter	Options	Description
System Clock	Asynchronous, Synchronous	Defines whether the FIFO read and write domain is synchronous or asynchronous. Default: Synchronous
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: 32
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: Enable
Programmable Full Assert Value	1 – DEPTH	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: 512
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	1 – Programmable Full Assert Value	Use when PROGRAMMABLE_FULL is set to STATIC_DUAL. Sets the lower threshold value for prog_full_o during deassertion. Default: 512
Programmable Empty Assert Value	0 – (FIFO Depth-1)	Threshold value when prog_empty_o is enabled. When Enable Programmable Full 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: 0
Programmable Empty Negate Value	Programmable Empty Assert Value – (DEPTH-1)	Use when PROGRAMMABLE_EMPTY is set to STATIC_DUAL. Sets the upper threshold value for prog_empty_o during deassertion. Default: 0

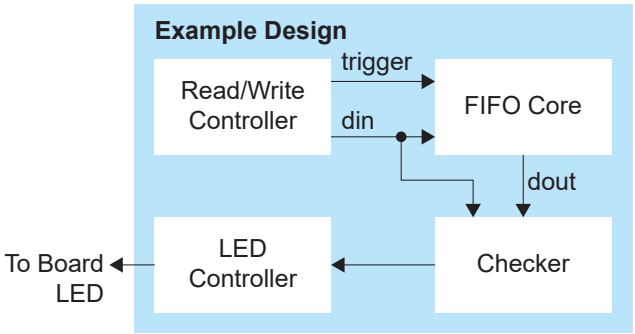
Parameter	Options	Description
Enable Programmable Empty Option	NONE, STATIC_SINGLE, STATIC_DUAL	Controls the prog_empty_o signal: NONE: Disabled. STATIC_SINGLE: Enabled, asserts and deasserts at a single threshold value. (default) STATIC_DUAL: Enabled, asserts or deasserts at different threshold values.
Optional Signals	Enable, Disable	Enables the optional signals: wr_ack_o, almost_full_o, overflow_o, rd_valid_o, almost_empty_o and underflow_o. You can disable this feature to improve macro timing. Default: Enable
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. You can disable this feature if a project does not require fast speed. Default: Enable

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.

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

Figure 13: FIFO Core Example Design



The example design targets the Trion® T20 BGA256 Development Board and Titanium Ti60 F225 Development Board. This design continuously generates data on `wdata` and writes it into the FIFO core.

The design has these blocks:

- *Read/Write Controller*–Sends `wdata` to the FIFO core and triggers the FIFO core for reads and writes.
- *Checker*–Compares `rdata` and `wdata` and sends the results to the LED controller.
- *LED Controller*–The MSB is the passed bit, and the reset is the number of errors detected.

Trion® T20 BGA256 Development Board

When the FIFO core has available words, the design reads the data and compares it with the expected data. If there are no errors, LED D6 is on; if errors are found, D6 blinks on and off while LEDs D3 through D5 count the number of errors found (up to 127 errors). You can force errors into the system by pressing pushbutton SW6. To reset, press pushbutton SW4.

Titanium Ti60 F225 Development Board

When the FIFO core has available words, the design reads the data and compares it with the expected data. If there are no errors, LED D19 is on; if errors are found, D19 blinks on and off while LEDs D16 through D18 count the number of errors found (up to 127 errors). You can force errors into the system by pressing pushbutton SW7. To reset, press pushbutton SW5.

Table 19: Trion® Example Design Implementation

125 MHz clock constraint.

FPGA	Clock and Read Mode	LUTs	Memory Blocks	I/Os	f_{MAX} (MHz)			Efinity® Version[3]
					clk_i	wr_clk_i	rd_clk_i	
T20 BGA256 C4	Synchronous Standard	250	4	15	143	–	–	2020.1
	Synchronous FWFT	257	4	15	145	–	–	
	Asynchronous Standard	336	4	16	–	184	163	
	Asynchronous FWFT	363	4	16	–	177	142	

Table 20: Titanium Example Design Implementation

FPGA	Clock and Read Mode	Logic and Adders	Flip-flops	Memory Blocks	DSP48 Blocks	f_{MAX} (MHz)			Efinity® Version[3]
						clk_i	wr_clk_i	rd_clk_i	
Ti60 F225 ES C3	Synchronous Standard	241	201	2	0	411	–	–	2021.1
	Synchronous FWFT	269	202	2	0	353	–	–	
	Asynchronous Standard	280	274	2	0	–	360	406	
	Asynchronous FWFT	309	278	2	0	–	409	338	

FIFO Testbench

You can choose to generate the testbench when generating the core in the IP Manager Configuration window.

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

The testbench reads data from the FIFO 100 times, checks each of them, and indicates a pass/fail for the last 16 sets of data. Additionally, it indicates an overall pass/fail for the entire test.

[3] Using Verilog HDL.

After running the simulation, the test prints the following message:

```
SYNC_CLK=1, MODE=STANDARD
 PIPELINE_REG=1, OUTPUT_REG=0
 OPTIONAL_FLAGS=1
#   85 PASSED: Read Data = 0x55
#   86 PASSED: Read Data = 0x56
#   87 PASSED: Read Data = 0x57
#   88 PASSED: Read Data = 0x58
#   89 PASSED: Read Data = 0x59
#   90 PASSED: Read Data = 0x5a
#   91 PASSED: Read Data = 0x5b
#   92 PASSED: Read Data = 0x5c
#   93 PASSED: Read Data = 0x5d
#   94 PASSED: Read Data = 0x5e
#   95 PASSED: Read Data = 0x5f
#   96 PASSED: Read Data = 0x60
#   97 PASSED: Read Data = 0x61
#   98 PASSED: Read Data = 0x62
#   99 PASSED: Read Data = 0x63
# 100 PASSED: Read Data = 0x64

#TEST PASSED
```

Note: If you use ModelSim to simulate the testbench, add the following line in the ModelSim command:

```
vlog -sv +define+SIM <example design directory>
```

Revision History

Table 21: Revision History

Date	Version	Description
June 2021	4.0	Added note about including all **.v** generated in testbench folder is required for simulation. Updated resource utilization and performance table. Updated example design output and implementation table. Added support for Titanium FPGAs and example design for Titanium Ti60 F225 Development Board. Updated for Efinity v2021.1.
December 2020	3.0	Updated user guide for Efinix® IP Manager which includes added IP Manager topics, updated parameters, and user guide structure.
August 2020	2.3	Updated the resource utilization and performance table, and example design implementation table.
July 2020	2.2	Added FIFO core new feature, option to exclude optional flags. Updated default value for DEPTH and DATA_WIDTH parameter.
July 2020	2.1	Updated simulation test print message. Corrected the download the example design to the board topic.

Date	Version	Description
July 2020	2.0	Updated for FIFO Core v2.0. Updated resource utilization and performance table. Added note to parameters to indicate if it is only available in FIFO core v2.0. Updated block diagrams to include datacount_o, rd_datacount_o, and wr_datacount_o signals. Updated FIFO core file name. Updated COMMON_CLOCK to SYNC_CLK. Updated DEPTH default value to 512. Updated DATA_WIDTH default value to 16. Removed statement stating that FWFT mode is undefined when COMMON_CLK is 0. Updated OUTPUT_REG possible values. Added description for FIFO v2.0. Added PIPELINE_REG, OPTIONAL_FLAGS, and FIFO_MODE parameters. Added FWFT mode waveform for FIFO core v2.0 and updated the description accordingly. Updated asynchronous FIFO operation section to include diagrams, descriptions, and examples for FWFT mode. Updated the reset topic to state that resetting does not initialize rdata and specifying reset waveform as standard mode. Added latency section to describe the latency of the output signal in detail. Updated example design implementation table.
May 2020	1.1	Updated OUTPUT_REG description. Updated MODE description. Updated figure titles for synchronous FIFO operation waveforms.
March 2020	1.0	Initial release.