FIFO User Guide

Version 1.0 March 2009

Agate Logic, Inc.

Contents

About This Guide

The Agate Logic FIFO IP v1.0 User Guide describes the function and operation of the FIFO, as well as information about designing, customizing, and implementing the IP.

About the FIFO IP Core

Section 1

1.1 Device Family Support

The synchronous or asynchronous FIFO supports the following target Agate Logic device families:

- Angelo

1.2 Introduction

As design complexities increase, the vendor-specific IP blocks has become a common design methodology. Agate Logic provides parameterized IPs that are optimized for Agate Logic device architecture. Using IP instead of coding your own logic saves your valuable design time.

The FIFO is built from EMB9K and can be configured with either asynchronous or synchronous for both write and read operations. The asynchronous configuration of the FIFO enables the user to implement unique clock domains on the write and read ports. A synchronous FIFO implementation optimizes the core for data buffering within a single clock domain.

1.3 Features

- Support data width up to 36 bits
- Support memory depths of up to 9x1024 locations
- Asynchronous and synchronous reset, high or low active is optional
- Fully synchronous and independent clock domains for the read and write ports
- Support full and empty status flags
- Optional almost_full and almost_empty status flags
- Optional half_full and half_empty status flags, they can be dynamically defined through half_full_thresh and half_empty_thresh, and you can set an range to control half_full/half_empty through half_full_assert/half_empty_assert and

half_full_negate/half_empty_negate.

- Invalid read or write requests are rejected without affecting the FIFO state
- Four optional handshake signals (wr_ack, rd_ack, overflow, underflow) provide feedback (acknowledgment or rejection) in response to write and read requests in the prior clock cycle
- Optional count vector(s) provide visibility into number of data words currently in the FIFO, synchronized to either clock domain
- Support different input and output data widths for asynchronous FIFO

Getting Started

Section 2

2.1 FIFO Interfaces

The following two sections provide definitions for the FIFO interface signals. Figure 1 illustrates these signals (both the standard and optional ports) for the asynchronous FIFO.

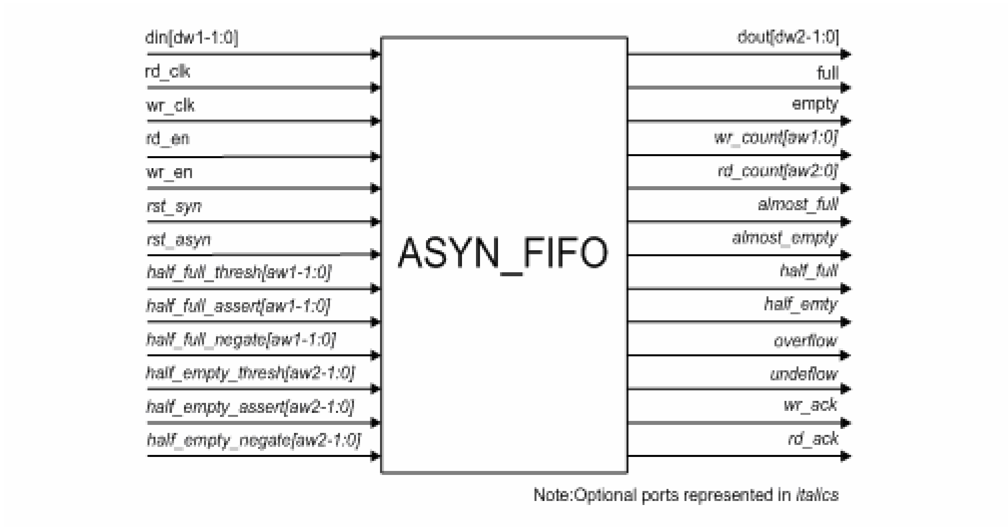
din[dw1-1:0]
rd_clk
wr_clk
rd_en
wr_en
rst_syn
rst_asyn
half_full_thresh[aw1-1:0]
half_full_assert[aw1-1:0]
half_full_negate[aw1-1:0]
half_empty_thresh[aw2-1:0]
half_empty_assert[aw2-1:0]
half_empty_negate[aw2-1:0]
ASYN_FIFO
dout[dw2-1:0]
full
empty
wr_count[aw1:0]
rd_count[aw2:0]
almost_full
almost_empty
half_full
half_emty
overflow
underflow
wr_ack
rd_ack
Note:Optional ports represented in italics

Figure 1 Asynchronous FIFO Interfaces

2.1.1.Interface Signals: Asynchronous FIFO

Table 2 defines the signals for the write interface of an asynchronous FIFO. The write interface signals are divided into required and optional signals and all signals are synchronous to the write clock (wr_clk).

Table 2 Write Interface Signals for Asynchronous FIFO

Name	Direction	Description
Required		
wr_clk	Input	write clock: All signals on the write domain are synchronous to this clock.
din[dw1-1:0]	Input	data input: The input data bus used when writing the FIFO.
wr_en	Input	write enable: If the FIFO is not full, asserting this signal causes data to be written to the FIFO. This signal is active high.
full	Input	full flag: When asserted, this signal indicated that the FIFO is full. Write requests are ignored when is full is

		non-destructive to the contents of the FIFO. This signal is active high.
Optional		
almost full	Output	almost full: When asserted, this signal indicates that only one more write can be performed before the FIFO is full. This signal is active high.
half_full	Output	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. This signal is active high.
wr_count[aw1:0]	Output	write data count: This bus indicates the number of words stored in the FIFO. The count is guaranteed to never under-report the number of words in the FIFO, to ensure the user never overflows the FIFO. The exception to this behavior is when a write operation occurs at the rising edge of wr_clk, that write operation will only be reflected on wr_count at the next rising clock edge.
wr_ack	Output	write acknowledge: This signal indicates that a write request (wr_en) during the prior clock cycle succeeded. This signal is active high.
overflow	Output	overflow: This signal indicates that a write request (wr_en) during the prior clock cycle is rejected, because the FIFO is full. Overflowing the FIFO is non-destructive to the contents of the FIFO. This signal is active high.
half_full_thresh[aw1-1:0]	Input	programmable full threshold: This signal is used to input the threshold value for the assertion and deassertion of the programmable full (half_full) flag. The threshold can be dynamically set in-circuit during reset. The user can either choose to set the assert and negate threshold to the same value (using half_full_thresh), or the user can control these values independently (using half_full_thresh_assert and half_full_thresh_negate).
half_full_thresh_assert[aw1-1:0]	Input	programmable full threshold assert: This signal is used to set the upper threshold value for the programmable full flag, which defines when the signal is asserted. The threshold can be dynamically set in-circuit during reset.
half_full_thresh_negate[aw1-1:0]	Input	programmable full threshold negate: This signal is used to set the lower threshold value for the programmable full flag, which defines when the signal is deasserted. The threshold can be dynamically set in-circuit during reset.
Reset signals		
rst_syn	Input	synchronous reset: A synchronous reset that initializes all

		internal pointers and output registers. Active low or high is optional.
rst_asyn	Input	asynchronous reset: An asynchronous reset that initializes all internal pointers and output registers. Active low or high is optional.

Table 3 defined the signals on the read interface of an asynchronous FIFO. The read interface signals are divided into required signals and optional signals, and all signals are synchronous to the read clock(rd_clk).

Table 3 Read Interface Signals for Asynchronous FIFO

Name	Direction	Description
Required		
rd_clk	Input	read clock: All signals on the read domain are synchronous to this clock.
dout[dw2-1:0]	Output	data output: The output data bus is driven when reading the FIFO.
rd_en	Input	read enable: If the FIFO is not empty, asserting this signal causes data to be read from the FIFO. This signal is active high.
empty	Output	empty flag: When asserted, this signal indicates that the FIFO is empty. Read requests are ignored when the FIFO is empty, initialing a read while empty is non-destructive to the FIFO. This signal is active high.
Optional		
almost_empty	Output	almost empty flag: When asserted, this signal indicates that the FIFO is almost empty and one word remains in the FIFO. This signal is active high.
half_empty	Output	programmable empty: This signal is asserted when the number of words in the FIFO is less than or equal to the programmable threshold. It is deasserted when the number of words in the FIFO exceeds the programmable threshold. This signal is active high.
rd_count[aw2:0]	Output	read data count: This bus indicates the number of words available for reading in the FIFO. The count is guaranteed to never over-report the number of words available for reading, to ensure that the user does not underflow the FIFO. The exception to this behavior is when read operation occurs at the rising edge of rd_clk, that read operation will only be reflected on rd_count at the next rising clock edge.
rd_ack	Output	rd_ack: This signal indicates that valid data is available on the output bus. This signal is active high.
underflow	Output	underflow: Indicates that the read request during the previous clock cycle was rejected because the FIFO is

		empty. Underflowing the FIFO is not destructive to the FIFO. This signal is active high.
half_empty_thresh[aw2-1:0]	Input	programmable empty threshold: This signal is used to input the threshold value for the assertion and deassertion of the programmable empty flag. The threshold can be dynamically set in-circuit during reset. The user can either choose to set the assert and negate threshold to the same value (using half_empty_thresh), or the user can control these values independent (using half_empty_thresh_assert and half_empty_thresh_negate).
half_empty_thresh_assert[aw2-1:0]	Input	programmable empty threshold assert: This signal is used to set the lower threshold value for the programmable empty flag, which defines when the signal is asserted. The threshold can be dynamically set in-circuit during reset.
half_empty_thresh_negate[aw2-1:0]	Input	programmable empty threshold negate: This signal is used to set the upper threshold value for the programmable empty flag, which defines when the signal is de-asserted. The threshold can be dynamically set in-circuit during reset.

2.1.2. Interface Signals: Synchronous FIFO

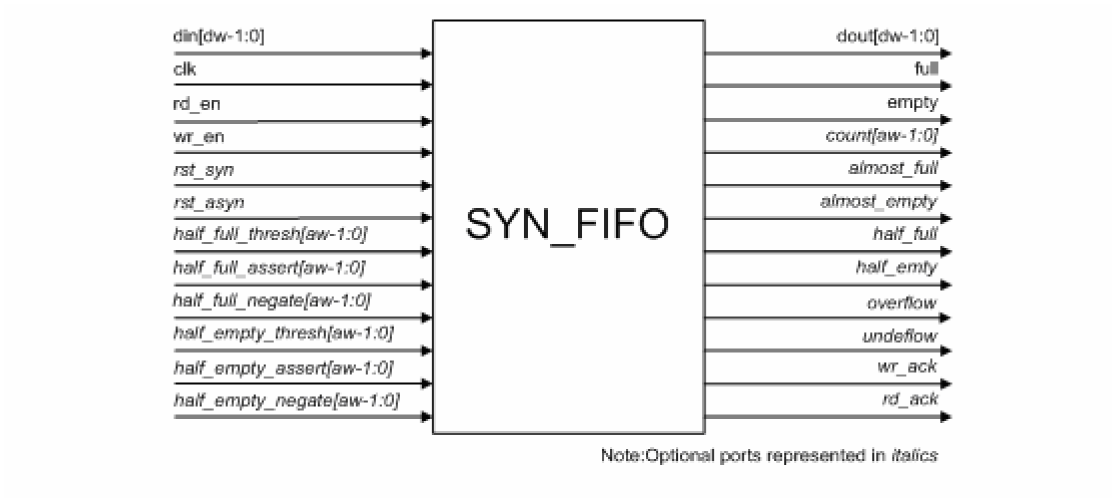

Figure 2 Synchronous FIFO Interface

Table 4 defines the interface signals of a synchronous FIFO with a common write and read clock. The table is divided into standard and optional interface signals, and all signals (except rst_asyn) are synchronous to the common clock. Users have the option to select synchronous or asynchronous reset for the FIFO .

Table 4 Write and Read Interface Signals for Synchronous FIFO

Name	Direcion	Description
Required		
rst_syn	Input	synchronous reset: An synchronous

		reset that initializes all internal pointers and output registers. Active low or high is optional.
rst_asyn	Input	asynchronous reset: An asynchronous reset that initializes all internal pointers and output registers. Active low or high is optional.
clk	Input	clock: All signals on the write and read domains are synchronous to this clock.
din[dw-1:0]	Input	data input: The input data bus used when writing the FIFO.
wr_en	Input	write enable: If the FIFO is not full, asserting this signal causes data to be written to the FIFO. This signal is active high.
full	Output	full flag: When asserted, this signal indicates that the FIFO is full. Write requests are ignored when the FIFO is full, initiating a write when the FIFO is full is non-destructive to the contents of the FIFO. This signal is active high.
dout[dw-1:0]	Output	data output: The output data bus is driven when reading the FIFO.
rd_en	Input	read enable: If the FIFO is not empty, asserting this signal causes data to be read from the FIFO. This signal is active high.
empty	Output	empty flag: When asserted, this signal indicates that the FIFO is empty. Read requests are ignored when the FIFO is empty, initiating a read while empty is non-destructive to the FIFO. This signal is active high.
Optional		
count[aw-1:0]	Output	data count: This bus indicates the number of words stored in the FIFO.
almost full	Output	almost full: When asserted, this signal indicates that only one more

		write can be performed before the FIFO is full. This signal is active high.
half_full	Output	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. This signal is active high.
wr_ack	Output	Write acknowledge: This signal indicates that a write request during the prior clock cycle succeeded. This signal is active high.
overflow	Output	overflow: This signal indicates that a write request during the prior clock cycle was rejected, because the FIFO is full. Overflowing the FIFO is non-destructive to the contents of the FIFO. This signal is active high.
half_full_thresh[aw-1:0]	Input	programmable full threshold: This signal is used to input the threshold value for the assertion and deassertion of the programmable full (half_full) flag. The threshold can be dynamically set in-circuit during reset. The user can either choose to set the assert and negate threshold to the same value (using half_full_thresh), or the user can control these values independently (using half_full_thresh_assert and half_full_thresh_negate).
half_full_thresh_assert[aw-1:0]	Input	programmable full threshold assert: This signal is used to set the upper threshold value for the programmable full flag, which defines when the signal is asserted. The threshold can be dynamically set in-circuit during reset.
half_full_thresh_negate[aw-1:0]	Input	programmable full threshold negate: This signal is used to set the lower

		threshold value for the programmable full flag, which defines when the signal is deasserted. The threshold can be dynamically set in-circuit during reset.
almost_empty	Output	almost empty flag: When asserted, this signal indicates that the FIFO is almost empty and one word remains in the FIFO. This signal is active high.
half_empty	Output	programmable empty: This signal is asserted when the number of words in the FIFO is less than or equal to the programmable threshold. It is deasserted when the number of words in the FIFO exceeds the programmable threshold. This signal is active high.
rd_ack	Output	rd_ack: This signal indicates that valid data is available on the output bus. This signal is active high.
underflow	Output	underflow: Indicates that the read request during the previous clock cycle was rejected because the FIFO is empty. Underflowing the FIFO is not destructive to the FIFO. This signal is active high.
half_empty_thresh[aw-1:0]	Input	programmable empty threshold: This signal is used to input the threshold value for the assertion and deassertion of the programmable empty flag. The threshold can be dynamically set in-circuit during reset. The user can either choose to set the assert and negate threshold to the same value (using half_empty_thresh), or the user can control these values independent (using half_empty_thresh_assert and half_empty_thresh_negate).
half_empty_thresh_assert[aw-1:0]	Input	programmable empty threshold assert: This signal is used to set the

		lower threshold value for the programmable empty flag, which defines when the signal is asserted. The threshold can be dynamically set in-circuit during reset.
half_empty_thresh_negate[aw-1:0]	Input	programmable empty threshold negate: This signal is used to set the upper threshold value for the programmable empty flag, which defines when the signal is deasserted. The threshold can be dynamically set in-circuit during reset.

2.2 FIFO Usage and Control

2.2.1 Asynchronous FIFO Write Operation

This section describes the behavior of a asynchronous FIFO write and the associated status flags. When write enable is asserted and the FIFO is not full, data is added to the FIFO from the input bus and write acknowledge is asserted. If the FIFO is continuously written to without being read, it fills with data. Write operations are only successful when the FIFO is not full. When the FIFO is full and a write is initiated, the request is ignored, the overflow flag is asserted and there is no change in the state of the FIFO.

Almost_full and Full flags

The almost_full flag indicates that only one more write can be performed before full is asserted. This flag is active high and synchronous to the write clock.
The full flag indicates that the FIFO is full and no more writes can be performed until data is read out. This flag is active high and synchronous to the write clock. If a writes is initiated when full is asserted, the write request is ignored and overflow is asserted.

Example Operation

Figure 3 shows a typical write operation. The user asserts wr_en, causing a write operation to occur on the next rising edge of the wr_clk. Since the FIFO is not full, wr_ack is asserted, acknowledging a successful write operation. When only one additional word can be written into the FIFO, the FIFO asserts the almost_full flag. When almost_full is asserted, one additional write causes the FIFO to assert full. When a write occurs after full is asserted, wr_ack is deasserted and overflow is asserted, indicating an overflow condition. Once the user performs one or more read operations, the FIFO deasserts full, and data can successfully be written to the FIFO, as is indicated by the assertion of wr_ack and deassertion of overflow.

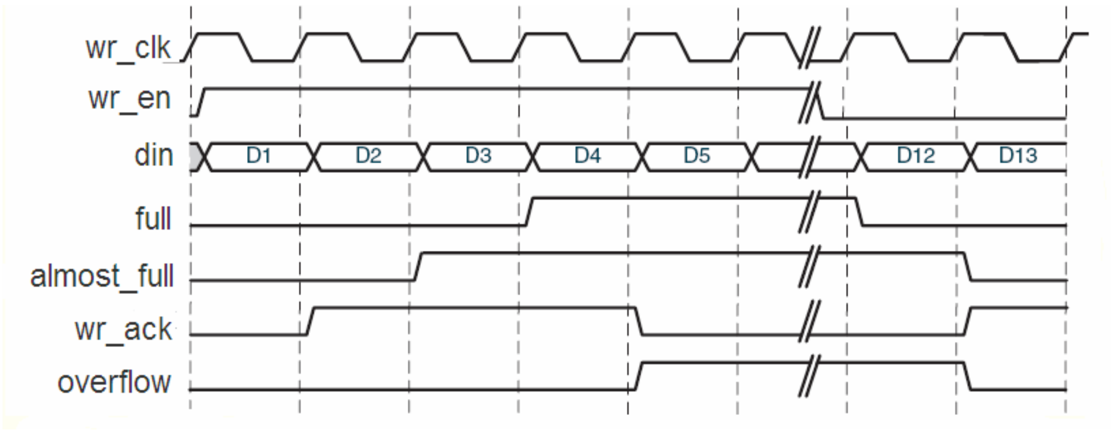

Figure 3 Write Operation for a Asynchronous FIFO

2.2.2 Asynchronous FIFO Read Operation

This section describes the behavior of an asynchronous FIFO read operation and the associated status flags. When read enable is asserted and the FIFO is not empty, data is read from the FIFO on the output bus, and the rd_ack flag is asserted. If the FIFO is continuously read without being written, the FIFO empties. Read operation are successful when the FIFO is not empty. When the FIFO is empty and a read is requested, the read operation is ignored, the underflow flag is asserted and there is no change in the state of the FIFO.

Almost_empty and Empty flags

The almost empty flag indicates that the FIFO will be empty after one more read operation. This flag is active high and synchronous to rd_clk. This flag is asserted when the FIFO has one remaining word that can be read.
The empty flag indicates that the FIFO is empty and no more reads can be performed until data is written into the FIFO. This flag is active high and synchronous to the rd_clk. If a read is initiated when empty is asserted, the request is ignored and underflow is asserted.

Example Operation

Figure 4 shows a typical read operation. The user asserts rd_en, causing a read operation to occur on the next rising edge of the rd_clk. Since the FIFO is not empty, rd_ack is asserted, acknowledging a successful read operation. When only one additional word can be read out from the FIFO, the FIFO asserts the almost_empty flag. When almost_empty is asserted, one additional read causes the FIFO to assert empty. When a read occurs after empty is asserted, rd_ack is deasserted and underflow is asserted, indicating an underflow condition. Once the user performs one or more write operations, the FIFO deasserts empty, and data can successfully be read from the FIFO, as is indicated by the assertion of rd_ack and deassertion of underflow.

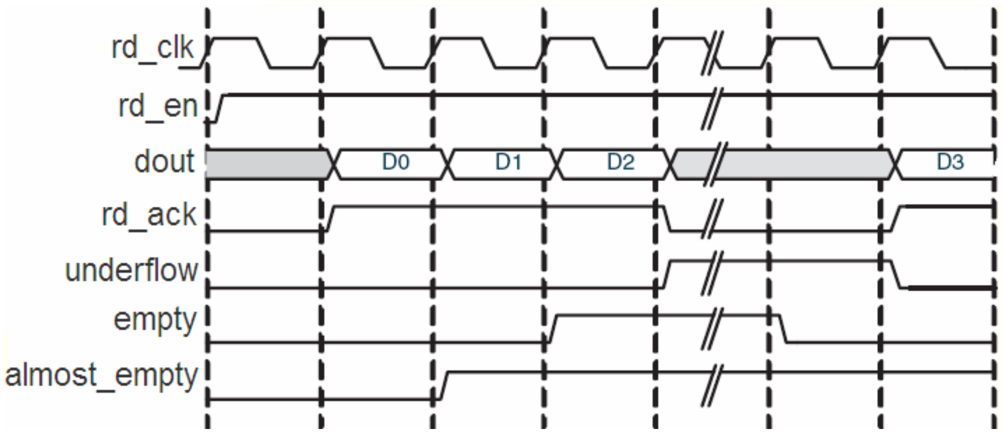

Figure 4 Read Operation for a Asynchronous FIFO

2.2.3 Synchronous FIFO Write and Read Operation

Figure 5 shows a typical write and read operation. A write is issued to the FIFO, resulting in the deassertion of the empty flag. A simultaneous write and read is then issued, resulting in no change in the status flags. Once two or more words are present in the FIFO, the almost_empty flag is deasserted. Write requests are then issued to the FIFO, resulting in the assertion of almost_full when the FIFO can only accept one more write. A simultaneous write and read is then issued, resulting in no change in the status flags. Finally one additional write without a read result in the FIFO asserting full, indicating no further data can be written until a read request is issued.

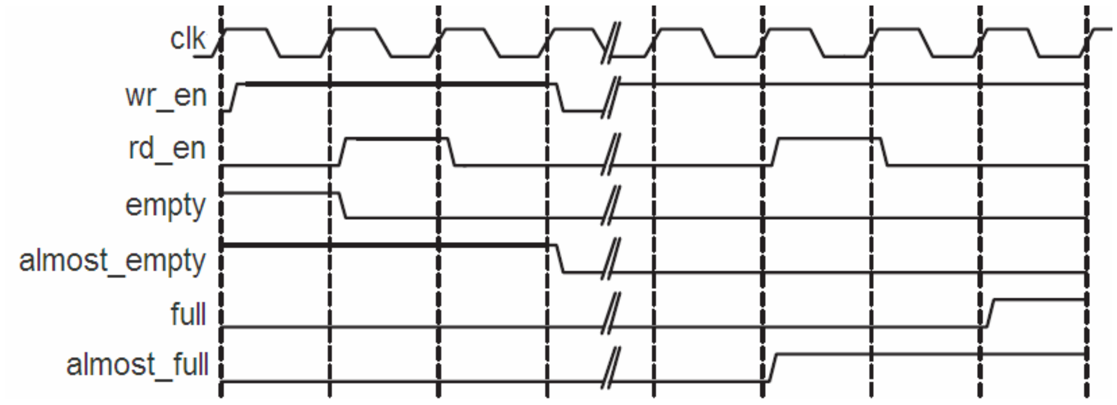

Figure 5 Write and Read Operation for a Synchronous FIFO

2.2.4 Asynchronous FIFO Handshaking flags

Handshaking flags (read acknowledge, underflow, write acknowledge and overflow) are supported to provide additional information regarding the status of the write and read operations. The handshaking flags are optional and active high. These flags are illustrated in Figure 6.

Write Acknowledge(wr_ack)

The write acknowledge flag is asserted at the completion of each successful write operation and indicates that the data on the din port has been stored in the FIFO. This flags is synchronous to the write clock.

Example Operation

Figure 6 illustrates the behavior of the FIFO flags. On the write interface, full is not asserted and writing to the FIFO is successful. When a write occurs after full is asserted, wr_ack is deasserted an overflow is asserted, indicating an overflow condition.

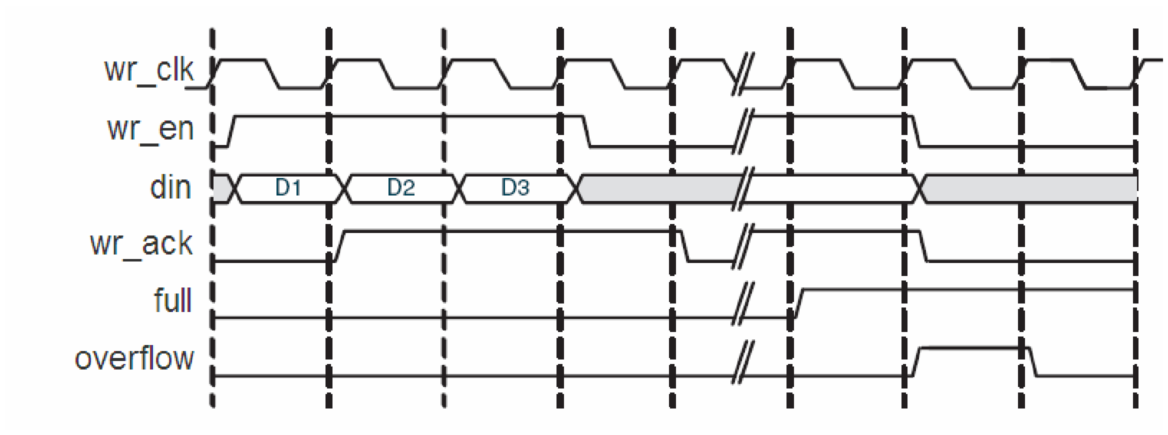

Figure 6 Write acknowledge signal for Asynchronous FIFO

Read acknowledge(rd_ack)

The read acknowledge is asserted at the rising edge of rd_clk for each successful read operation, and indicates that the data on the dout bus is valid. When a read request is unsuccessful, read acknowledge is not asserted. This flags is synchronous to the read clock.

Example Operation

Figure 7 illustrates the behavior of the FIFO flags. On the read interface, once the FIFO is not empty, the FIFO accepts read requests. In FIFO operation, rd_ack is asserted and dout is updated on the clock cycle following the read request.

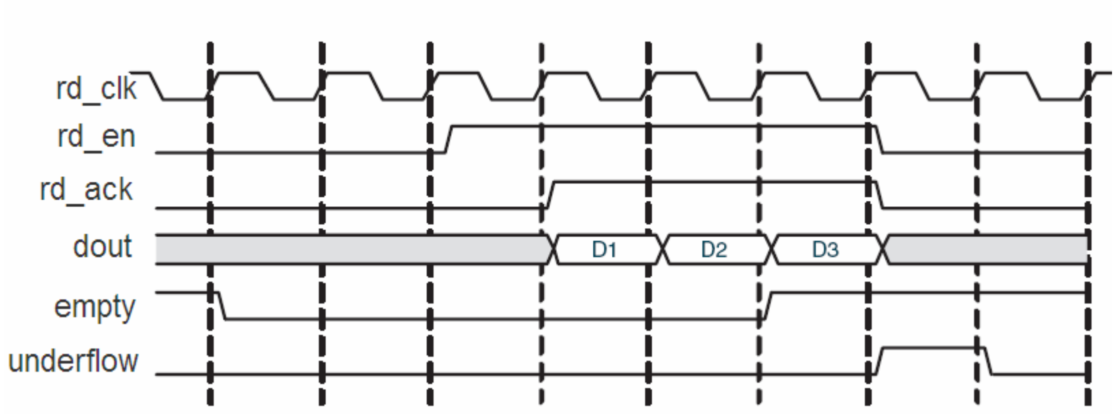

Figure 7 Read Acknowledge Aignal for Asynchronous FIFO

Underflow

The underflow flag is used to indicate that a read operation is unsuccessful. This occurs when a read is initiated and the FIFO is empty. This flag is synchrounous with the read clock. Underflowing the FIFO does change the state of the FIFO.

Example Operation

On the read interface, once the FIFO is not empty, the FIFO accepts read requests. Following a read request, rd_ack is asserted and dout is updated. When a read request is issued while empty is asserted, rd_ack is deasserted and underflow is asserted, indicating an underflow condition.

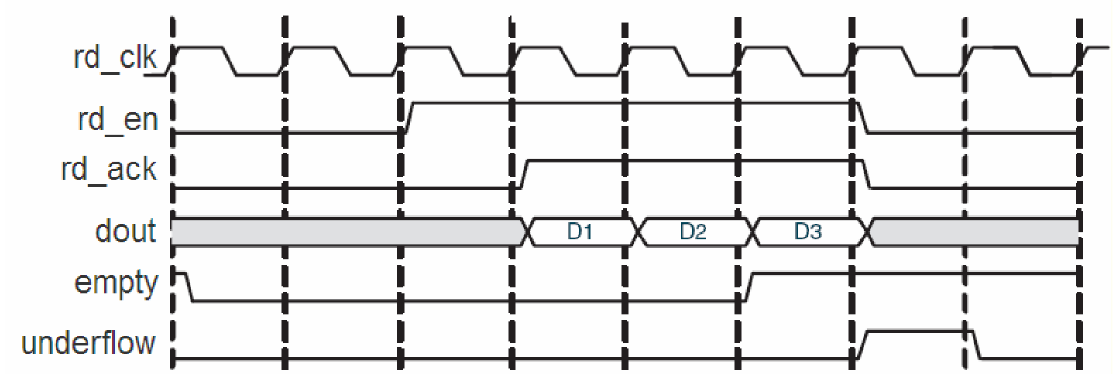

Figure 8 Underflow Signal for Synchronous FIFO

Overflow

The overflow flag is used to indicate that a write operation is unsuccessful. This flag is asserted when a write is initiated to the FIFO while full is asserted. The overflow flag is synchronous to the write clock. Overflowing the FIFO does not change the state of the FIFO.

Example Operation

On the write interface, full is deasserted and therefore writes to the FIFO are successful. When a write occurs after full is asserted, wr_ack is deasserted and overflow is asserted, indicating an overflow condition.

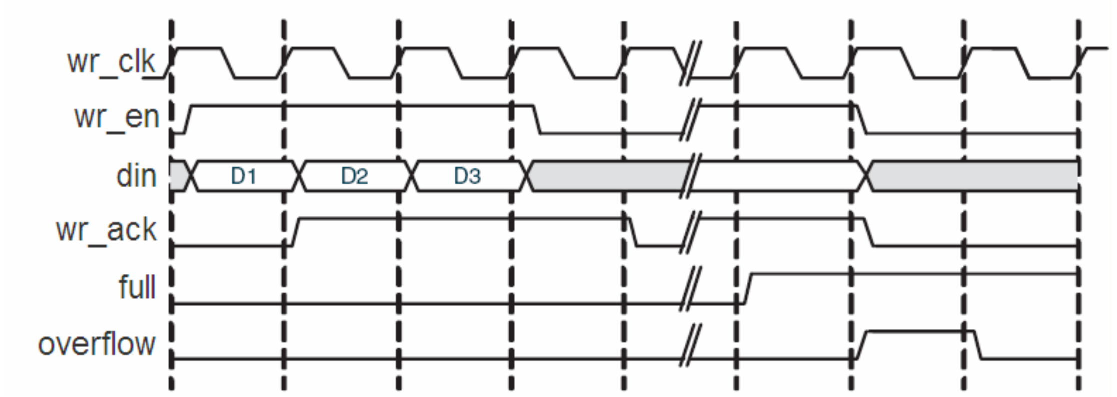

Figure 9 Overflow Signal for Ssynchronous FIFO

2.2.5 Synchronous FIFO Handshaking flags

Handshaking flags (read acknowledge, underflow, write acknowledge and overflow) are supported to provide additional information regarding the status of the write and read operations. The handshaking flags are optional and active high. These flags are illustrated in Figure 10.

Write Acknowledge(wr_ack)

The write acknowledge flag is asserted at the completion of each successful write operation and indicates that the data on the din port has been stored in the FIFO. This flags is synchronous to the clock.

Example Operation

Figure 10 illustrates the behavior of the FIFO flags. On the write interface, full is not asserted and writing to the FIFO is successful. When a write occurs after full is asserted, wr_ack is deasserted an overflow is asserted, indicating an overflow condition.

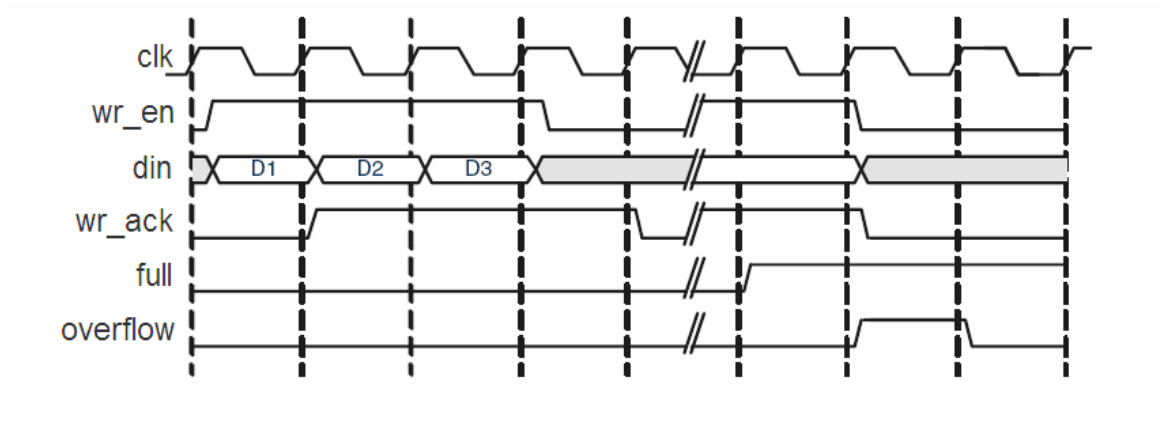

Figure 10 Write Acknowledge Signal for Synchronous FIFO

Read Acknowledge(rd_ack)

The read acknowledge is asserted at the rising edge of rd_clk for each successful read operation, and indicates that the data on the dout bus is valid. When a read request is unsuccessful, read acknowledge is not asserted. This flags is synchronous to the clock.

Example Operation

Figure 11 illustrates the behavior of the FIFO flags. On the read interface, once the FIFO is not empty, the FIFO accepts read requests. In FIFO operation, rd_ack is asserted and dout is updated on the clock cycle following the read request.

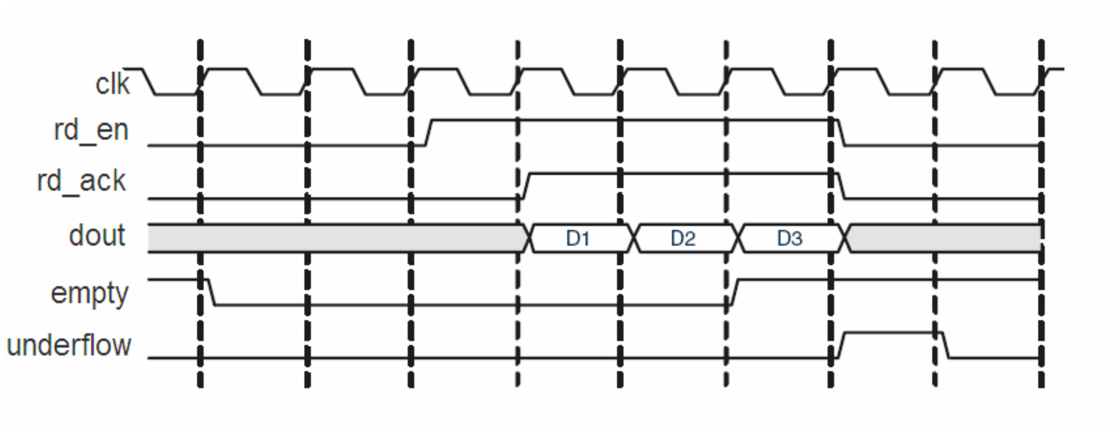

Figure 11 Read Acknowledge Signal for Synchronous FIFO

Underflow

The underflow flag is used to indicate that a read operation is unsuccessful. This occurs when a read is initiated and the FIFO is empty. This flag is synchrounous with the clock. Underflowing the FIFO does change the state of the FIFO.

Example Operation

Once the FIFO is not empty, the FIFO accepts read requests. Following a read request, rd_ack is asserted and dout is updated. When a read request is issued while empty is asserted, rd_ack is deasserted and underflow is asserted, indicating an underflow condition.

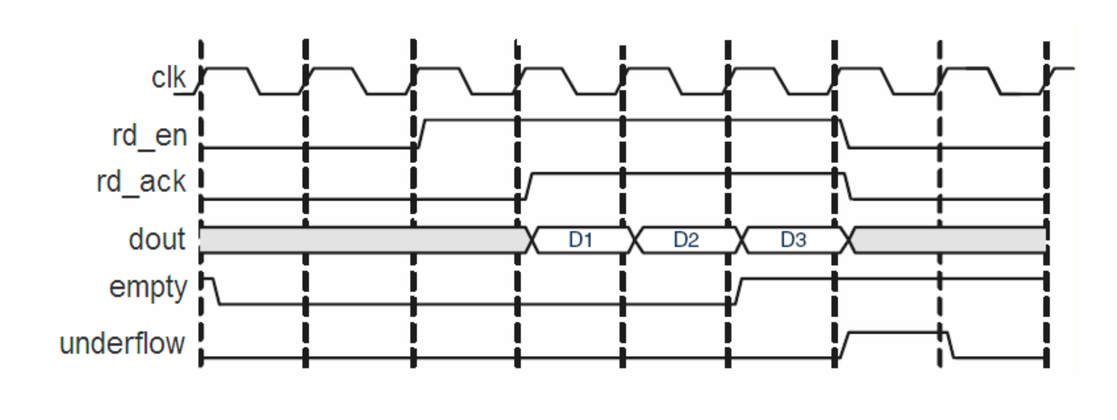

Figure 12 Underflow Signal for Synchronous FIFO

Overflow

The overflow flag is used to indicate that a write operation is unsuccessful. This flag is asserted when a write is initiated to the FIFO while full is asserted. The overflow flag is synchronous to the write clock. Overflowing the FIFO does not change the state of the FIFO.

Example Operation

Full signal is deasserted and therefore writes to the FIFO are successful. When a write occurs after full is asserted, wr_ack is deasserted and overflow is asserted, indicating an overflow condition.

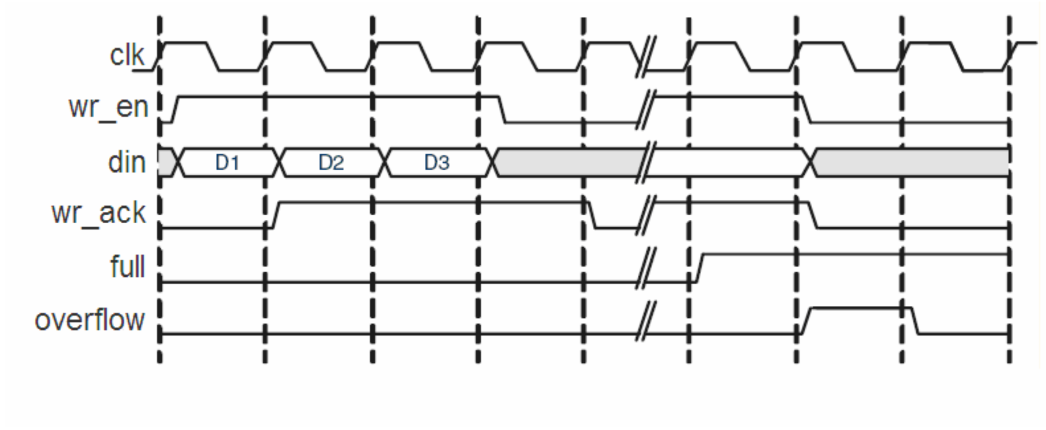

Figure 13 Overflow Signal for Synchronous FIFO

2.2.5 Asynchronous FIFO programmable flags

The FIFO supports programmable flags to indicate that the FIFO has reached a user-defined fill level.
Programmable full (half_full) indicates that the FIFO has reached a user-defined full threshold.
Programmable empty (half_empty) indicates that the FIFO has reached a user-defined empty threshold.
For these thresholds, the user can set a constant value or choose to have dedicated input ports, enabling the threshold to change dynamically in circuit. Detailed information about these options is provided below.

Programmable Full

The FIFO supports four ways to define the programmable full threshold:

- Single threshold constant
- Single threshold with dedicated input port
- Assert and negate thresholds constants
- Assert and negate thresholds with dedicated input ports

The programmable full flag (half_full) is asserted when the number of entries in the FIFO is greater than or equal to the user-defined assert threshold. When the programmable full flag is asserted, the FIFO can continue to be written to until full flag is asserted. If the number of words in the FIFO is less than the negate threshold, the flag is deasserted.

Programmable full: single threshold

This option enables the user to set a single threshold value for the assertion and deassertion of half_full,. When the number of entries in the FIFO is greater than or equal to the threshold value, half_full is asserted. When the number of entries in the FIFO is less than the threshold value, half_full is deasserted.

There are two options for implementing this threshold:

- **Single threshold constant.** User specifies the threshold value through the IP WIZARD GUI. Once the IP is generated, this value can only be changed by regenerating the IP CORE.
- **Single threshold with dedicated input port.** User specifies the threshold value through an input port on the core. This input can be changed while the FIFO is in reset, providing the user the flexibility to change the programmable full threshold in-circuit without re-generating the core.

Figure 14 shows the programmable full flag with a single threshold. The user writes to the FIFO until there are seven words in the FIFO. Since the programmable full threshold is set to seven, the FIFO asserts half_full once seven word are written into the FIFO. Note that both write data count and half_full have one clock cycle of delay. Once the FIFO has six or fewer words in the FIFO, half_full is deasserted.

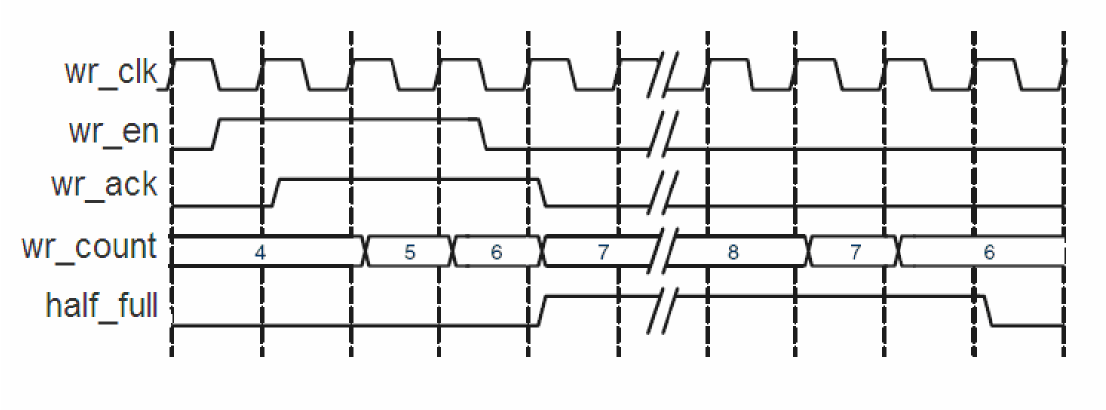

Figure 14 Half full signal for Threshold: Threshold Set to 7

Programmable Full: Assert and Negate Threshold

This option enables the user to set separate values for the assertion and deassertion of half_full. When the number of entries in the FIFO is greater than or equal to the asserted value, half_full is asserted. When the number of entries in the FIFO is less than the negate value, half_full is deasserted.

There are two options for implementing these thresholds:

- **Assert and negate threshold constants.** User specifies the threshold values through the IP WIZARD GUI. Once the core is generated, these values can only be changed by re-generating the core.
- **Assert and negate threshold with dedicated input ports.** User specifies the threshold values through input ports on the core. These input ports can be changed while the FIFO is in reset, providing the user the flexibility to change the values of the programmable full assert and negate thresholds in-circuit without re-generating the core.

Figure 15 shows the programmable full flag with assert and negate thresholds. The user writes to the FIFO until there are 10 words in the FIFO. Because the assert threshold is set to 10, the FIFO then asserts half_full. The negate threshold is set to seven, and the FIFO deasserts half_full once six words or fewer are in the FIFO. Both write data count and half_full have one clock cycle delay.

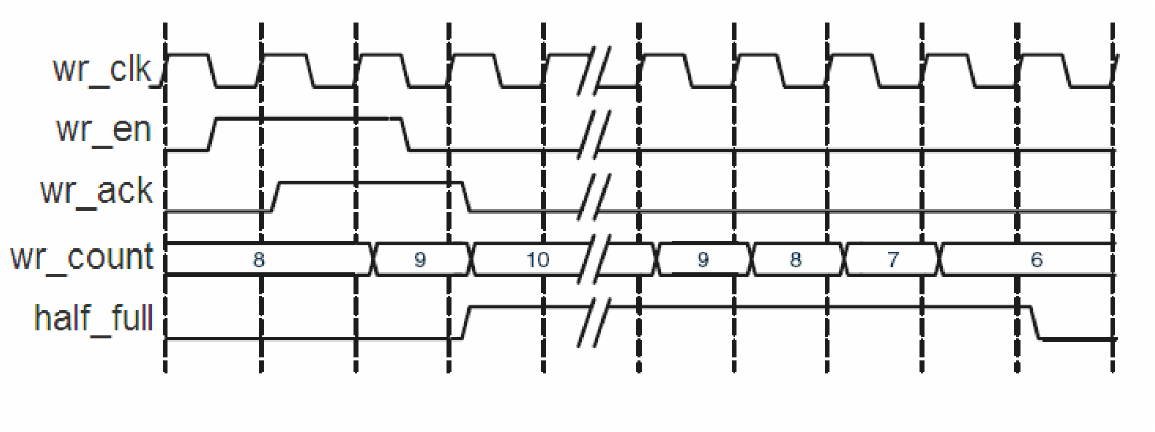

Figure 15 Half full signal with Assert and Negate Threshold: Assert Set to 10 and Negate Set to 7

Programmable Empty

The FIFO supports fours ways to define the programmable empty thresholds:

- Single threshold constant
- Single threshold with dedicated input port
- Assert and negate threshold constants
- Assert and negate threshold with dedicated input ports

The programmable empty flag is asserted when the number of entries in the FIFO is less than or equal to the user-defined assert threshold. If the number of words in the FIFO is greater than the negate threshold, the flag is deasserted.

Programmable empty: single threshold

This option enables the user to set a single threshold value for the assertion and deassertion of half_empty. When the number of entries in the FIFO is less than or equal to the threshold value, half_empty is asserted. When the number of entries in the FIFO is greater than the threshold value, half_empty is deasseerted.

There are two options for implementing this threshold.

Single threshold constant: User specifies the threshold value through the IP WIZARD GUI. Once the core is generated, this value can only be changed by re-generating the core. This option consumes fewer resources than the single threshold with dedicated input port.

Single threshold with dedicated input port: User specifies the threshold value through an input port on the core. This input can be changed while the FIFO is in reset, providing the user the flexibility to change the programmable empty threshold in-circuit without re-generating the core.

Figure 16 show the programmable empty flag with a single threshold. The user writes to the FIFO until there are five words in the FIFO. Since the programmable empty threshold is set to four, half_empty is asserted until more than four words are present in the FIFO. Once five words are present in the FIFO, half_empty is deasserted. Both read data count and half_empty have one clock cycle of delay.

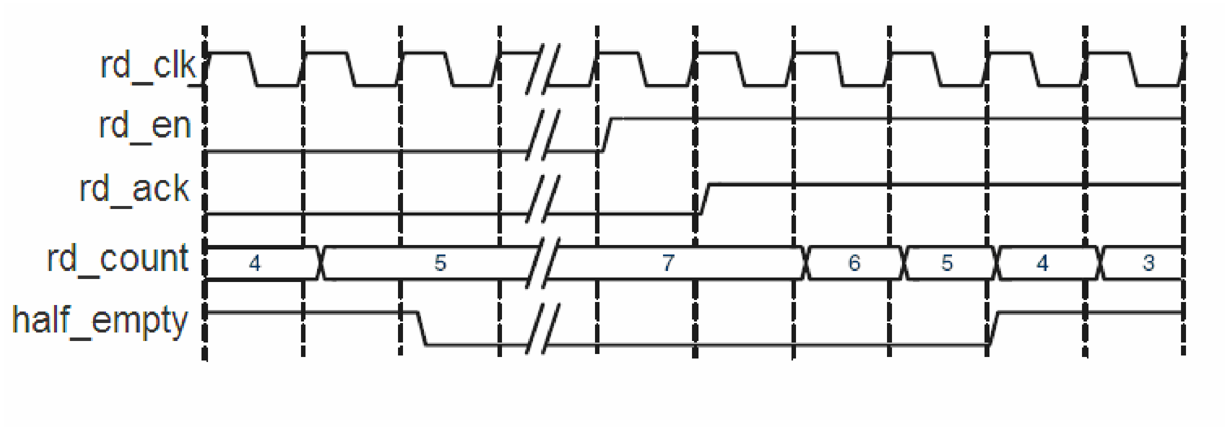

Figure 16 Half empty signal for Threshold: Threshold Set to 4

Programmable empty: Assert and negate threshold

This option enables the user to set separate values for the assertion and deassertion of half_empty. When the number of entries in the FIFO is less than or equal to the assert value, half_empty is asserted. When the number of entries in the FIFO is greater than the negate value, half_empty is deasserted.

There are two options for implementing the assert and negate threshold.

- Assert and negate threshold constants. The threshold values are specified through the IP WIZARD GUI. Once the core is generated, these values can only be changed by re-generating the core. This option consumes fewer resources than the assert and negate thresholds with dedicated input ports.
- Assert and negate thresholds with dedicated input ports. The threshold values are specified through input ports on the core. These input ports can be changed while the FIFO is in reset, providing the user the flexibility to change the values of the programmable empty assert and negate thresholds in-circuit without regenerating the core.

Figure 17 shows the programmable empty flag with assert and negate thresholds. The user writes to the FIFO until there are eleven words in the FIFO. Since the programmable empty deassert values is set to ten, half_empty is deasserted when there are more than ten words in the FIFO. Once the FIFO contains less than or equal to the programmable empty negate value, half_empty is asserted. Both read data count and half_empty have one clock cycle of delay.

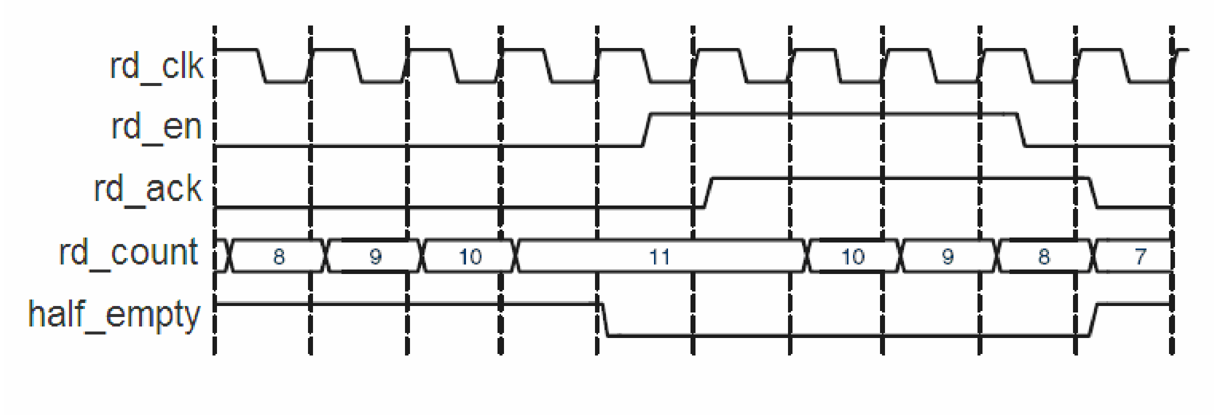

Figure 17 Half full signal with Assert and Negate Threshold: Assert Set to 7 and Negate Set to 10

2.2.6 Synchronous FIFO Programmable Flags

The FIFO supports programmable flags to indicate that the FIFO has reached a user-defined fill level.

Programmable full (half_full) indicates that the FIFO has reached a user-defined full threshold.

Programmable empty (half_empty) indicates that the FIFO has reached a user-defined empty threshold.

For these thresholds, the user can set a constant value or choose to have dedicated input ports, enabling the threshold to change dynamically in circuit. Detailed information about these options is provided below.

Programmable Full

The FIFO supports four ways to define the programmable full threshold:

- Single threshold constant
- Single threshold with dedicated input port
- Assert and negate thresholds constants
- Assert and negate thresholds with dedicated input ports

The programmable full flag (half_full) is asserted when the number of entries in the FIFO is greater than or equal to the user-defined assert threshold. When the programmable full flag is asserted, the FIFO can continue to be written to until full flag is asserted. If the number of words in the FIFO is less than the negate threshold, the flag is deasserted.

Programmable full: single threshold

This option enables the user to set a single threshold value for the assertion and deassertion of half_full,. When the number of entries in the FIFO is greater than or equal to the threshold value, half_full is asserted. When the number of entries in the FIFO is less than the threshold value, half_full is deasserted.

There are two options for implementing this threshold:

- **Single threshold constant.** User specifies the threshold value through the IP WIZARD GUI. Once the IP is generated, this value can only be changed by regenerating the IP CORE.
- **Single threshold with dedicated input port.** User specifies the threshold value through an input port on the core. This input can be changed while the FIFO is in reset, providing the user the flexibility to change the programmable full threshold in-circuit without re-generating the core.

Figure 18 shows the programmable full flag with a single threshold. The user writes to the FIFO until there are seven words in the FIFO. Since the programmable full threshold is set to seven, the FIFO asserts half_full once seven word are written into the FIFO. Note that both write data count and half_full have one clock cycle of delay. Once the FIFO has six or fewer words in the FIFO, half_full is deasserted.

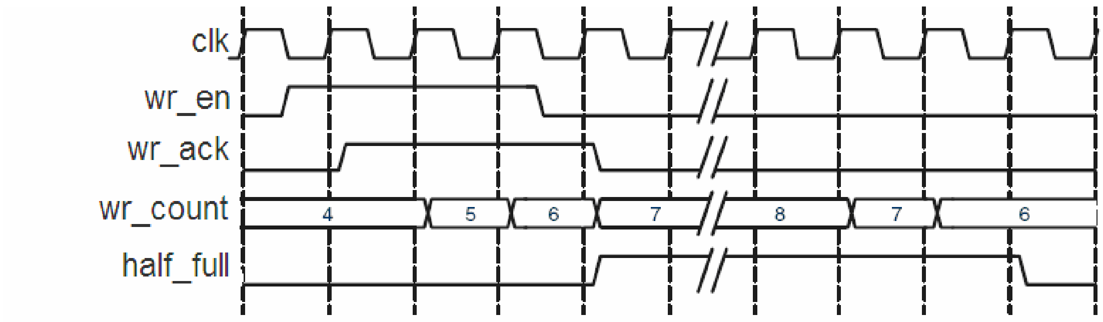

Figure 18 Half full signal for Threshold: Threshold Set to 7

Programmable Full: Assert and Negate Threshold

This option enables the user to set separate values for the assertion and deassertion of half_full. When the number of entries in the FIFO is greater than or equal to the asserted value, half_full is asserted. When the number of entries in the FIFO is less than the negate value, half_full is deasserted.

There are two options for implementing these thresholds:

- **Assert and negate threshold constants.** User specifies the threshold values through the IP WIZARD GUI. Once the core is generated, these values can only be changed by re-generating the core.
- **Assert and negate threshold with dedicated input ports.** User specifies the threshold values through input ports on the core. These input ports can be changed while the FIFO is in reset, providing the user the flexibility to change the values of the programmable full assert and negate thresholds in-circuit without re-generating the core.

Figure 19 shows the programmable full flag with assert and negate thresholds. The user writes to the FIFO until there are 10 words in the FIFO. Because the assert threshold is set to 10, the FIFO then asserts half_full. The negate threshold is set to seven, and the FIFO deasserts half_full once six words or fewer are in the FIFO. Both write data count and half_full have one clock cycle delay.

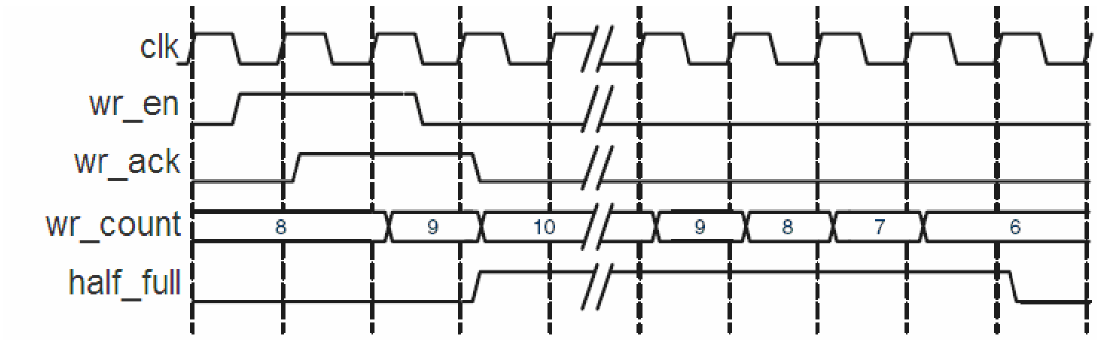

Figure 19 Half full signal with Assert and Negate Threshold: Assert Set to 10 and Negate Set to 7

Programmable Empty

The FIFO supports fours ways to define the programmable empty thresholds:

- Single threshold constant
- Single threshold with dedicated input port

- Assert and negate threshold constants
- Assert and negate threshold with dedicated input ports

The programmable empty flag is asserted when the number of entries in the FIFO is less than or equal to the user-defined assert threshold. If the number of words in the FIFO is greater than the negate threshold, the flag is deasserted.

Programmable empty: single threshold

This option enables the user to set a single threshold value for the assertion and deassertion of half_empty. When the number of entries in the FIFO is less than or equal to the threshold value, half_empty is asserted. When the number of entries in the FIFO is greater than the threshold value, half_empty is deasseerted.

There are two options for implementing this threshold.

Single threshold constant: User specifies the threshold value through the IP WIZARD GUI. Once the core is generated, this value can only be changed by re-generating the core. This option consumes fewer resources than the single threshold with dedicated input port.

Single threshold with dedicated input port: User specifies the threshold value through an input port on the core. This input can be changed while the FIFO is in reset, providing the user the flexibility to change the programmable empty threshold in-circuit without re-generating the core.

Figure 20 show the programmable empty flag with a single threshold. The user writes to the FIFO until there are five words in the FIFO. Since the programmable empty threshold is set to four, half_empty is asserted until more than four words are present in the FIFO. Once five words are present in the FIFO, half_empty is deasserted. Both read data count and half_empty have one clock cycle of delay.

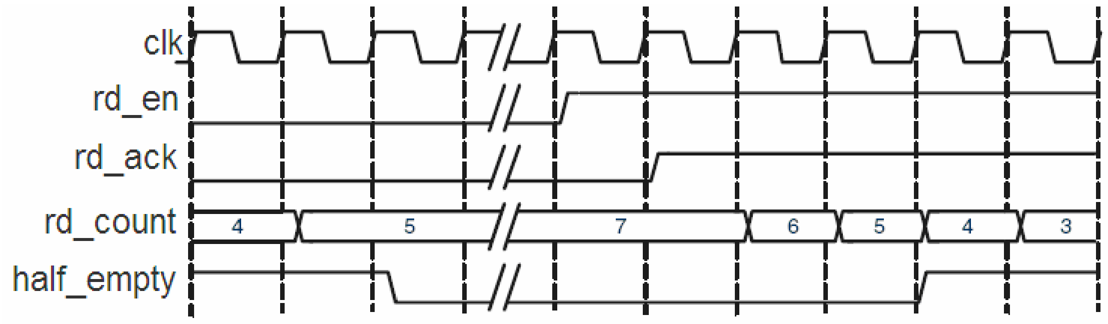

Figure 20 Half empty signal for Threshold: Threshold Set to 4

Programmable empty: Assert and negate threshold

This option enables the user to set separate values for the assertion and deassertion of half_empty. When the number of entries in the FIFO is less than or equal to the assert value, half_empty is asserted. When the number of entries in the FIFO is greater than the negate value, half_empty is deasserted.

There are two options for implementing the assert and negate threshold.

- Assert and negate threshold constants. The threshold values are specified through the IP WIZARD GUI. Once the core is generated, these values can only be changed by re-generating the core. This option consumes fewer resources than the assert and negate thresholds with dedicated input ports.
- Assert and negate thresholds with dedicated input ports. The threshold values are specified through input ports on the core. These input ports can be changed while the FIFO is in reset,

providing the user the flexibility to change the values of the programmable empty assert and negate thresholds in-circuit without regenerating the core.

Figure 21 shows the programmable empty flag with assert and negate thresholds. The user writes to the FIFO until there are eleven words in the FIFO. Since the programmable empty deassert values is set to ten, half_empty is deasserted when there are more than ten words in the FIFO. Once the FIFO contains less than or equal to the programmable empty negate value, half_empty is asserted. Both read data count and half_empty have one clock cycle of delay.

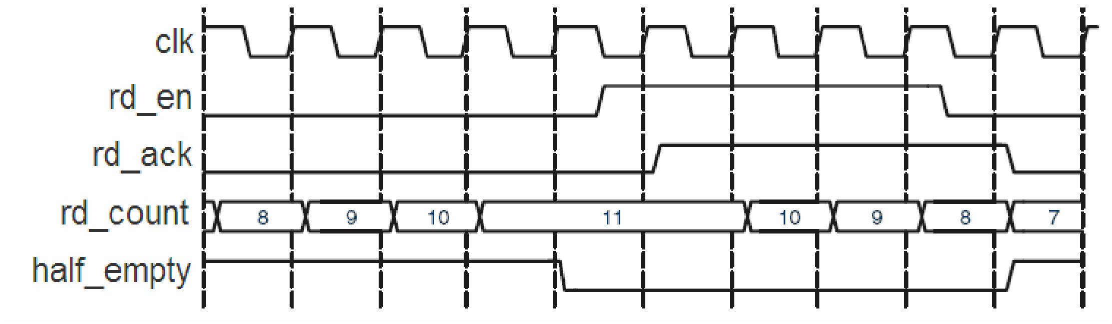

Figure 21 Half full signal with Assert and Negate Threshold: Assert Set to 7 and Negate Set to 10

2.2.7 Asynchronous FIFO Data Counts

data_count tracks the number of words in the FIFO. You can specify the width of the data count bus with a maximum width of $\log_2^{(\text{FIFO depth})}+1$. If the width specified is smaller than the maximum allowable width, the bus is truncated by removing the lower bits.

Read data count

Read data count pessimistically reports the number of words available for reading. The count is guaranteed to never over-report the number of words available in the FIFO to ensure that the user never underflows the FIFO. The user can specify the width of the read data count bus width a maximum width of $\log_2^{(\text{read depth})}+1$. If the width specified is smaller than the maximum allowable width, the bus is truncated width the lower bits removed.

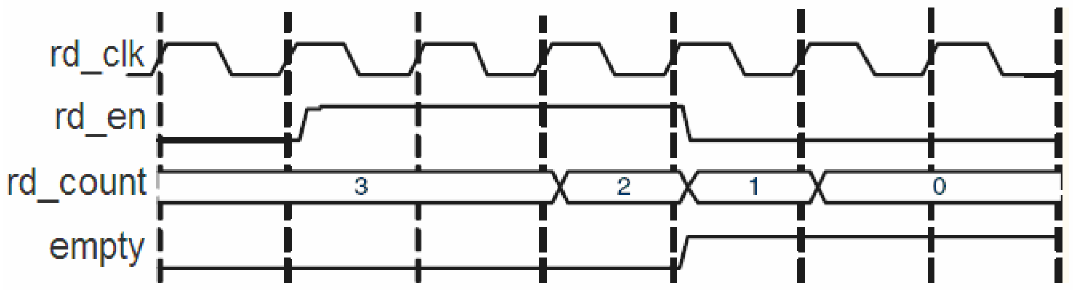

Figure 22 Read Data Count for Asynchronous FIFO

Write data count

Write data count pessimistically reports the number of words written into the FIFO. The count is guaranteed to never under-report the number of words in the FIFO to ensure that the user never overflows the FIFO. The user can specify the width of the write data count bus width a maximum width of $\log_2^{(\text{write depth})}+1$. If the width specified is smaller than the maximum allowable width, the bus is truncated with the lower bits removed.

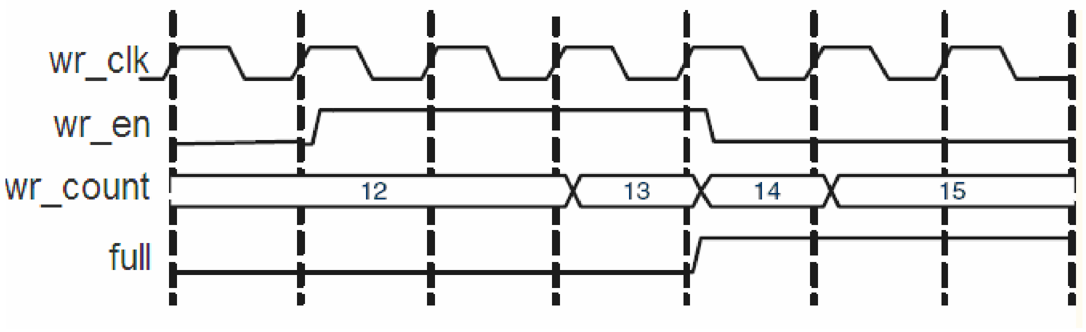

Figure 23 Write Data Count for Asynchronous FIFO

2.2.8 Synchronous FIFO Count

Count

Count output accurately reports the number of words available in a synchronous FIFO. You can specify the width of the count bus with a maximum width of $\log_2^{(\text{depth})}$. If the width specified is smaller than the maximum allowable width, the bus it truncated with the lower bits removed.
For example, you can specify to use two bits out of a maximum allowable three bits (provided a FIFO depth is eigth). These two bits indicate the number of words in the FIFO with a quarter resolution, providing the status of the contents of the FIFO for read and write operations.

2.2.9 Reset Behavior

The FIFO IP provides reset input that reset all counters, output registers, and memories when asserted. There are two reset options: asynchronous and synchronous. The asynchronous reset port is rst_asyn, the synchronous reset port is rst_syn and they can be active high or low.
Table 5 defines the FIFO reset values.

Table 5 FIFO reset values

signal	Asynchronous reset	Synchronous reset
dout	0	0
full	0	0
empty	1	1

wr_ack	0	0
rd_ack	0	0
half_full	0	0
half_empty	1	1
underflow	0	0
overflow	0	0
almost_full	0	0
almost_empty	1	1
wr_count	0	-
rd_count	0	-
count	-	0

FIFO Parameters

Section 3

Customers can set FIFO read and write data width, depth and optional handshaking flags by some parameters. Table 6 describes the parameters.

Table 6 FIFO Parameters

Asynchronous FIFO	
Parameters name	**comment**
dw1	Write data width
aw1	Write FIFO depth
dw2	Read data width
aw2	Read FIFO depth
rst_low	Set reset low or high: 1:low active 0:high active. There are asynchronous and synchronous ports. If rst_syn or rst_asyn port does not be used, they must be instantiated to 0.
Half_full_type	generate the half full signal type: 0: don't generate half_full signal 1: set the single constant threshold 2: set the assert and negate constant threshold 3: set the single input threshold 4: set the input assert and negate threshold
Half_empty_type	generate the half empty signal type: 0: don't generate half_empty signal 1: set the single constant threshold 2: set the assert and negate constant threshold 3: set the single input threshold 4: set the input assert and negate threshold
param_half_full_assert	Set the assert constant threshold
param_half_full_negate	Set the negate constant threshold If the half_full_type is 1, the param_half_full_negate equal to the param_half_full_assert decrease 1.
param_half_empty_assert	Set the assert constant threshold
param_half_empty_negate	Set the negate constant threshold If the half_empty_type is 1, the param_half_empty_negate equal to the param_half_empty_assert add 1.
Asynchronous FIFO	
dw	Data width
aw	FIFO depth
rst_low	Set reset low or high:

	1:low active 0:high active. There are asynchronous and synchronous ports. If rst_syn or rst_asyn port does not be used, they must be instantiated to 0.
Half_full_type	generate the half full signal type: 0: don’t generate half_full signal 1: set the single constant threshold 2: set the assert and negate constant threshold 3: set the single input threshold 4: set the input assert and negate threshold
Half_empty_type	generate the half empty signal type: 0: don’t generate half_empty signal 1: set the single constant threshold 2: set the assert and negate constant threshold 3: set the single input threshold 4: set the input assert and negate threshold
param_half_full_assert	Set the assert constant threshold
param_half_full_negate	Set the negate constant threshold If the half_full_type is 1, the param_half_full_negate equal to the param_half_full_assert decrease 1.
param_half_empty_assert	Set the assert constant threshold
param_half_empty_negate	Set the negate constant threshold If the half_empty_type is 1, the param_half_empty_negate equal to the param_half_empty_assert add 1.

FIFO Instantiation

Section 4

Asynchronous FIFO instantiation:

```
module ip_module_name(
        din,
       half_full_thresh,
        half_empty_thresh,
        half_empty_assert,
        half_empty_negate,
        half_full_assert,
        half_full_negate,
        rd_clk,
        rd_en,
        rst_syn,
       rst_asyn,
        wr_clk,
        wr_en,
        almost_empty,
        almost_full,
        dout,
        empty,
        full,
        overflow,
        half_empty,
        half_full,
        rd_ack,
        rd_count,
        underflow,
        wr_ack,
        wr_count);

parameter dw1=8;                //write data width
parameter aw1=10;               //write FIFO depth
parameter dw2=4;                //read data width
parameter aw2=11;               //read FIFO depth
 parameter rst_low = 0;           //reset optional    1:low active 0:high active
parameter half_full_type = 0;                 //set half full type
parameter half_empty_type = 0;                //set half empty type
parameter param_half_full_assert = 0;       //programmable full signal assert threshold
parameter param_half_full_negate = 0;      //programmable full signal negate threshold
parameter param_half_empty_assert = 0;    //programmable empty signal assert threshold
parameter param_half_empty_negate = 0;    //programmable empty signal negate threshold

input [dw1-1:0] din;                      //data write into FIFO
input [aw2-1:0] half_empty_assert; //programmable empty signal assert threshold, it is valid port only the
half_empty_type is 4
input [aw2-1:0] half_empty_negate; //programmable empty signal negate threshold, it is valid port only the
half_empty_type is 4
input [aw1-1:0] half_full_assert;     //programmable full signal assert threshold, it is valid only the
half_full_type is 4
input [aw1-1:0] half_full_negate; //programmable full signal negate threshold, it is valid port only the
half_full_type is 4
input [aw1-1 0] half_full_thresh; //programmable full signal threshold, it is valid port only the half_full_type
is 3
input [aw2-1:0] half_empty_thresh; //programmable empty signal threshold, it is valid port only the
half_empty_type is 3
input rd_clk; //read clock
input rd_en; //read enable
input rst_syn; //synchronous reset
input rst_asyn; //asynchronous reset
input wr_clk; //write clock
input wr_en; //write enable
output almost_empty; //almost empty signal
output almost_full; //almost full signal
output [dw2-1 : 0] dout; //data out from FIFO
output empty; //empty signal
output full; //full signal
output overflow; //overflow signal
output half_empty; //programmable empty signal
output half_full; //programmable full signal
output rd_ack; //read valid signal
output [aw2 : 0] rd_count; //read availably count
output underflow; //underflow signal
output wr_ack; //write acknowledge
output [aw1 : 0] wr_count; //write data count

    async_fifo    inst (
        .din(din),
        . half_full_thresh (half_full_thresh),
        . half_empty_thresh (half_empty_thresh),
        .half_empty_assert (half_empty_assert),
        .half_empty_negate (half_empty_negate),
        .half_full_assert (half_full_assert),
        .half_full_negate (half_full_negate),
                    .rd_clk (rd_clk),
                    .rd_en (rd_en),
                    .rst_syn (rst_syn),
                    .rst_asyn (rst_asyn),
                    .wr_clk (wr_clk),
                    .wr_en (wr_en),
                    .almost_empty (almost_empty),
                    .almost_full (almost_full),
                    .dout (dout),
                    .empty (empty),
                    .full (full),
                    .overflow (overflow),
                    .half_empty (half_empty),
                    .half_full (half_full),
                    .rd_ack (rd_ack),
                    .rd_count (rd_count),
                    .underflow (underflow),
                    .wr_ack (wr_ack),
                    .wr_count (wr_count));

            defparam inst.dw1 = dw1; //write data width
            defparam inst.dw2 = dw2; //read data width
            defparam inst.param_half_full_assert = param_half_full_assert; //programmable full signal assert
threshold constant
            defparam inst.param_half_full_negate = param_half_full_negate; //programmable full signal assert
threshold constant
            defparam inst.half_full_type = half_full_type;
            defparam inst.param_half_empty_assert = param_half_empty_assert; //programmable empty signal
assert threshold constant
            defparam inst.param_half_empty_negate=param_half_empty_negate; //programmable empty signal
assert threshold constant
            defparam inst.half_empty_type = half_empty_type;
            defparam inst.aw1 = aw1;//write data depth
            defparam inst.aw2 = aw2;//read FIFO depth
             defparam inst.rst_low = rst_low;//1 low reset,0 high reset

      endmodule
```

Synchronous FIFO instantiation:

```
module ip_module_name(
        din,
       half_full_thresh,
        half_empty_thresh,
        half_empty_assert,
        half_empty_negate,
        half_full_assert,
        half_full_negate,
        clk,
        rd_en,
        rst_syn,
         rst_asyn,
        wr_en,
        almost_empty,
        almost_full,
        dout,
        empty,
        full,
        overflow,
        half_empty,
        half_full,
        rd_ack,
        underflow,
        wr_ack,
        count);

parameter dw=8;                    //data width
parameter aw=10;                   //FIFO depth
 parameter rst_low = 0;      //reset optional    1:low active 0:high active
parameter half_full_type = 0;                    //set half full type
parameter half_empty_type = 0;                   //set half empty type
parameter param_half_full_assert = 0;        //programmable full signal assert threshold
parameter param_half_full_negate = 0;      //programmable full signal negate threshold
parameter param_half_empty_assert = 0;    //programmable empty signal assert threshold
parameter param_half_empty_negate = 0;    //programmable empty signal negate threshold

input [dw-1 : 0] din; //data write into FIFO
input [aw-1 : 0] half_empty_assert; //programmable empty signal assert threshold, it is valid port only the
half_empty_type is 4
input [aw-1 : 0] half_empty_negate; //programmable empty signal negate threshold, it is valid port only the
half_empty_type is 4
input [aw-1:0] half_full_assert; // programmable full signal assert threshold, it is valid port only the half_ full
```

_type is 4

input [aw-1:0] half_full_negate; // programmable full signal negate threshold, it is valid port only the half_ full _type is 4

input [aw-1:0] half_full_thresh; //programmable full signal threshold, it is valid port only the half_full_type is 3

input [aw-1:0] half_empty_thresh; //programmable empty signal threshold, it is valid port only the half_empty_type is 3

```
input clk; //clock
input rd_en; //read enable
input rst_syn; //synchronous reset
input rst_asyn; //asynchronous reset
input wr_en; //write enable
output almost_empty; //almost empty signal
output almost_full; //almost full signal
output [dw-1 : 0] dout; //data out from FIFO
output empty; //empty signal
output full; //full signal
output overflow; //overflow signal
output half_empty; //programmable empty signal
output half_full; //programmable full signal
output rd_ack; //read valid signal
output [aw-1 : 0] count; //data amount of in FIFO
output underflow; //underflow signal
output wr_ack; //write acknowledge

        sync_fifo    inst (
            .din(din),
            . half_full_thresh (half_full_thresh),
            . half_empty_thresh (half_empty_thresh),
            .half_empty_assert (half_empty_assert),
            .half_empty_negate (half_empty_negate),
            .half_full_assert (half_full_assert),
            .half_full_negate (half_full_negate),
            .clk (clk),
            .rd_en (rd_en),
            .rst_syn (rst_syn),
            .rst_asyn (rst_asyn),
            .wr_en (wr_en),
            .almost_empty (almost_empty),
            .almost_full (almost_full),
            .dout (dout),
            .empty (empty),
            .full (full),
            .overflow (overflow),
            .half_empty (half_empty),
            .half_full (half_full),
            .rd_ack (rd_ack),
            .underflow (underflow),
            .wr_ack (wr_ack),
            .count (count));

            defparam inst.dw = dw; //write data width
            defparam inst.param_half_full_assert = param_half_full_assert; //programmable full signal assert threshold constant
            defparam inst.param_half_full_negate = param_half_full_negate; //programmable full signal assert threshold constant
            defparam inst.half_full_type = half_full_type;
            defparam inst.param_half_empty_assert = param_half_empty_assert; //programmable empty signal assert threshold constant
            defparam inst.param_half_empty_negate=param_half_empty_negate; //programmable empty signal assert threshold constant
            defparam inst.half_empty_type = half_empty_type;
            defparam inst.aw = aw;//write data depth
            defparam inst.rst_low = rst_low;//1 low reset,0 high reset

endmodule
```

Simulating Design

Section 5

The Agate Logic provide the behavioral model to customers for simulation.

The behavioral models are considered to be zero-delay models, as the modeled write-to-read latency is nearly zero. The behavioral models are functionally correct, and will represent the behavioral of the configured FIFO, although the write-to-read latency and the behavioral of the status flags will differ from the actual implementation of the FIFO design.

Performance Information

Section 6

6.1 Resource Utilization and Performance

Performance and resource utilization for a FIFO varies depending on the configuration and features selected when customizing the core. The tables below provide example FIFO configurations and the maximum performance and resources required.

Table 7 FIFO Performance and Resources

FIFO Type	WidthxDepth	Family	Performance(MHz)	Resources		
				LUTs	FFs	EMB9K
Synchronous	1x8k	Angelo	53	99	47	1
	2x4k		47	93	44	1
	4x2k		52	85	41	1
	8x1k		48	80	38	1
	16x512		56	73	35	1
	32x256		52	66	32	1
	9x1k		44	80	38	1
	18x512		57	73	35	1
	36x256		48	66	32	1
Asynchronous	1x8k	Angelo	30	158	94	1
	2x4k		34	148	88	1
	4x2k		43	138	82	1
	8x1k		50	125	76	1
	16x512		41	118	70	1
	32x256		58	109	64	1
	9x1k		43	125	76	1
	18x512		40	118	70	1
	36x256		51	109	64	1

About Agate Logic

Agate Logic is the global pioneer and leader of the innovative Adaptable Programmable Gate Array (APGA) technologies. The company offers a full spectrum of programmable logic devices, software design tools, intellectual property (IP) and design services. Focusing on multiple applications such as telecommunication equipments, industrial control systems and consumer products, we use the Chinese leading foundry partner, SMIC, to manufacture our chips to offer solutions tailored for the market in China.

Technical Support Assistance

Tel:	+86 10 82150100
E-mail:	support@agatelogic.com
Website:	www.agatelogic.com.cn

Copyright © 2005-2009 Agate Logic, Inc. All rights reserved. No part of this document may be copied, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the written permission of Agate Logic, Inc. All trademarks are the property of their respective companies.