DW_asymfifoctl_s1_df

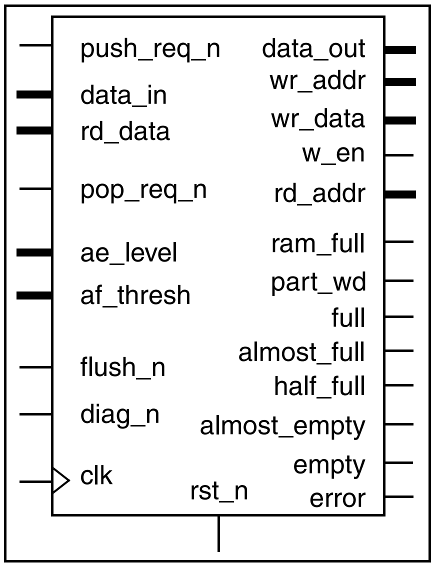
Asym. I/O Synch. (One Clock) FIFO Controller - Dynamic Flags

Version, STAR and Download Information: IP Directory

Features and Benefits

Revision History

- Fully registered synchronous address and flag output ports
- All operations execute in a single clock cycle
- FIFO empty, half full, and full flags
- Asymmetric input and output bit widths (must be integer-multiple relationship)
- Word integrity flag for *data_in_width* < *data_out_width*
- Flushing out partial word for *data_in_width* < *data_out_width*
- Parameterized byte order within a word
- FIFO error flag indicating underflow, overflow, and pointer corruption
- Parameterized word depth
- Dynamically programmable almost full and almost empty flags
- Parameterized reset mode (synchronous or asynchronous)
- Interfaces to common hard macro or compiled ASIC dual-port synchronous RAMs
- Provides minPower benefits with the DesignWare-LP license.



Description

DW_asymfifoctl_s1_df is a FIFO RAM controller designed to interface with a dual-port synchronous RAM.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
clk	1 bit	Input	Input clock
rst_n	1 bit	Input	Reset input, active low asynchronous if *rst_mode* = 0, synchronous if *rst_mode* = 1)
push_req_n	1 bit	Input	FIFO push request, active low
flush_n	1 bit	Input	Flushes the partial word into memory (fills in 0's) (for *data_in_width* < *data_out_width* only)
pop_req_n	1 bit	Input	FIFO pop request, active low

Table 1-1 Pin Description (Continued)

Pin Name	Width	Direction	Function
diag_n	1 bit	Input	Diagnostic control, active low (for *err_mode* = 0, NC for other *err_mode* values)
data_in	*data_in_width* bit(s)	Input	FIFO data to push
rd_data	max (*data_in_width*, *data_out_width*) bit(s)	Input	RAM data input to FIFO controller
ae_level	ceil($\log_2$[*depth*]) bit(s)	Input	Almost empty level (the number of words in the FIFO at or below which the `almost_empty` flag is active)
af_thresh	ceil($\log_2$[*depth*]) bit(s)	Input	Almost full threshold (the number of words stored in the FIFO at or above which the `almost_full` flag is active)
w_en	1 bit	Output	Write enable output for write port of RAM, active low
empty	1 bit	Output	FIFO empty output, active high
almost_empty	1 bit	Output	FIFO almost empty output, active high, asserted when FIFO level $\leq$ *ae_level*
half_full	1 bit	Output	FIFO half full output, active high
almost_full	1 bit	Output	FIFO almost full output, active high, asserted when FIFO level $\geq$ *af_thresh*
full	1 bit	Output	FIFO full output, active high
ram_full	1 bit	Output	RAM full output, active high
error	1 bit	Output	FIFO error output, active high
part_wd	1 bit	Output	Partial word, active high (for *data_in_width* < *data_out_width* only; otherwise, tied low)
wr_data	max (*data_in_width*, *data_out_width*) bit(s)	Output	FIFO controller output data to RAM
wr_addr	ceil($\log_2$[*depth*]) bit(s)	Output	Address output to write port of RAM
rd_addr	ceil($\log_2$[*depth*]) bit(s)	Output	Address output to read port of RAM
data_out	*data_out_width* bit(s)	Output	FIFO data to pop

Table 1-2 Parameter Description

Parameter	Values	Description
data_in_width	1 to 256	Width of the data_in bus. *data_in_width* must be in an integer-multiple relationship with *data_out_width*. That is, either *data_in_width* = *K* x *data_out_width*, or *data_out_width* = *K* x *data_in_width*, where *K* is an integer.
data_out_width	1 to 256	Width of the data_out bus. *data_out_width* must be in an integer-multiple relationship with *data_in_width*. That is, either *data_in_width* = K x *data_out_width*, or *data_out_width* = K x *data_in_width*, where K is an integer.
depth	2 to 2^{24}	Number of memory elements used in the FIFO (*addr_width* = ceil[$\log_2$(*depth*)])
err_mode	0 to 2 Default: 1	Error mode 0 = underflow/overflow with pointer latched checking 1 = underflow/overflow latched checking 2 = underflow/overflow unlatched checking
rst_mode	0 or 1 Default: 1	Reset mode 0 = asynchronous reset 1 = synchronous reset
byte_order	0 or 1 Default: 0	Order of bytes or subword [*subword* < 8 bits > *subword*] within a word 0 = first byte is in most significant bits position 1 = first byte is in the least significant bits position

Table 1-3 Synthesis Implementations

Implementation Name	Function	License Feature Required
str	Synthesis model	DesignWare

Table 1-4 Simulation Models

Model	Function
DW03.DW_ASYMFIFOCTL_S1_DF_CFG_SIM	Design unit name for VHDL simulation
dw/dw03/src/DW_asymfifoctl_s1_df_sim.vhd	VHDL simulation model source code
dw/sim_ver/DW_asymfifoctl_s1_df.v	Verilog simulation model source code

Table 1-5 Error Mode Description

error_mode	Error Types Detected	Error Output	diag_n
0	Underflow/Overflow and Pointer Corruption	Latched	Connected
1	Underflow/Overflow	Latched	N/C
2	Underflow/Overflow	Not Latched	N/C

The input data bit width of DW_asymfifoctl_s1_df can be different than its output data bit width, but must have an integer-multiple relationship (the input bit width being a multiple of the output bit width or vice versa). In other words, either of the following conditions must be true:

- *The data_in_width* $= K \times$ *data_out_width*, or
- *The data_out_width* $= K \times$ *data_in_width*

where K is a positive integer.

The RAM must have:

- A synchronous write port,
- Either asynchronous or synchronous read port, and
- The bit width must be the maximum of *data_in_width* or *data_out_width*.

The asymmetric FIFO controller provides address generation, write-enable logic, flag logic, and operational error detection logic.

The `almost_empty` and `almost_full` flags are dynamically set by the `ae_level` and `af_thresh` inputs.

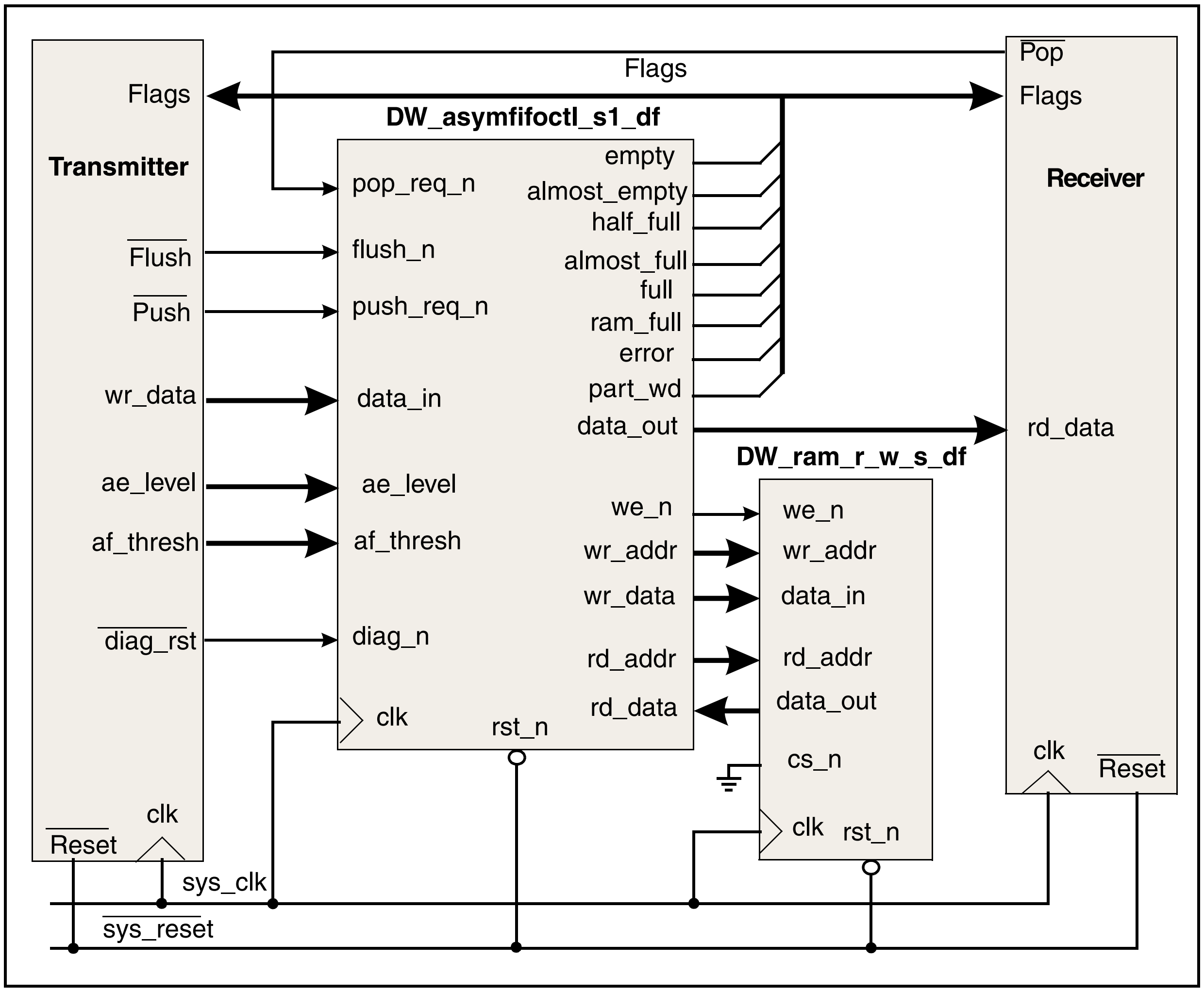
Parameterizable features include:

- FIFO depth (up to 24 address bits or 16,777,216 locations),
- Level of error detection, and
- Type of reset (asynchronous or synchronous).

You specify these parameters when the controller is instantiated in the design.

Figure 1-1 on page 5 shows a typical application of the asymmetric FIFO controller.

Figure 1-1 Example Usage of DW_asymfifoctl_s1_df



Writing to the FIFO (Push) for data_in_width > data_out_width Case

For cases where *data_in_width* > *data_out_width* (assuming that *data_in_width* = *K* × *data_out_width*, where *K* is an integer larger than 1):

- The `flush_n` input pin is not used (at the system level, this pin should not be connected so that it is removed upon synthesis),
- The `part_wd` output pin is tied LOW, and
- The `data_in` bus is connected directly to the `wr_data` output bus.

Refer to Figure 1-2 on page 6 for an example of this case.

The `wr_addr` and `we_n` output ports of the FIFO controller provide the write address and synchronous write enable to the FIFO.

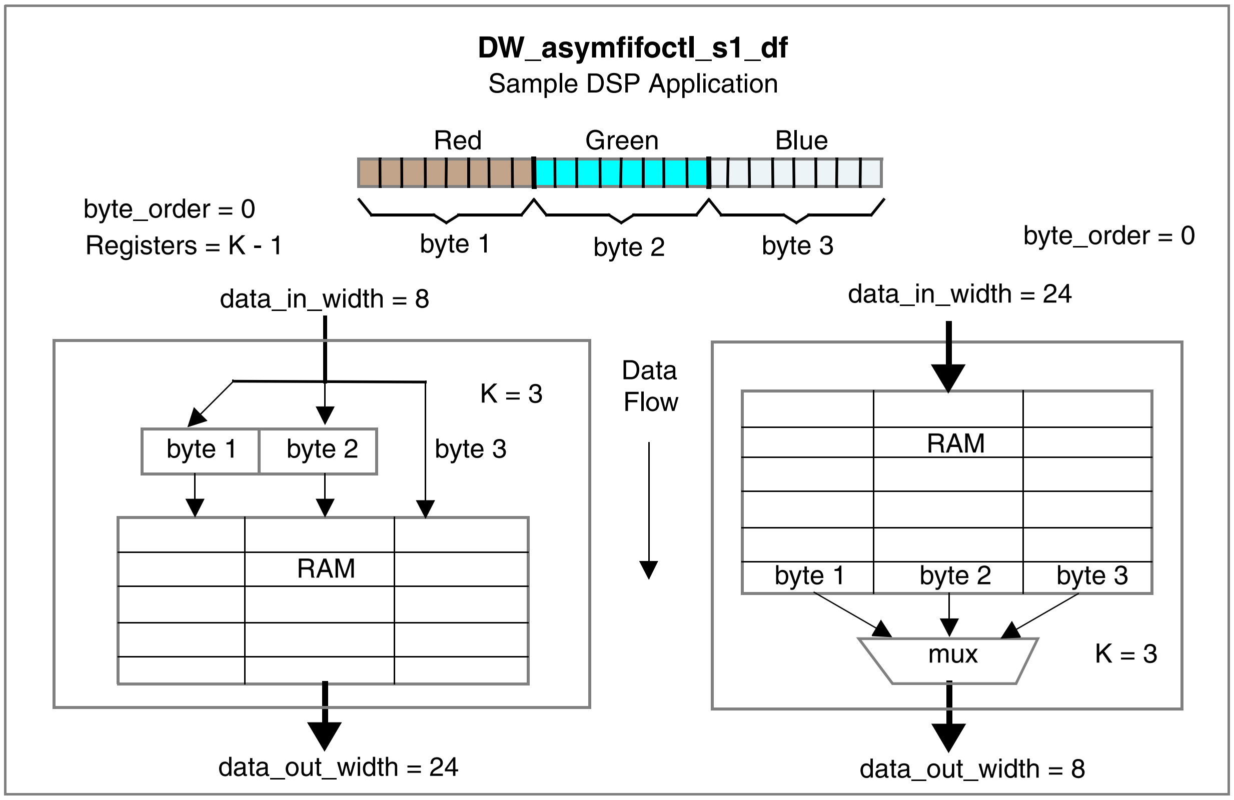
A push is executed when the `push_req_n` input is asserted (LOW) and the `full` flag is inactive (LOW) at the rising edge of `clk`.

Asserting `push_req_n` when the `full` flag is inactive causes the following events to occur:

- The `we_n` is asserted immediately, preparing for a write to the RAM on the next clock, and
- On the next rising edge of `clk`, the `wr_addr` signal is incremented.

Thus, the RAM is written, and `wr_addr` (which always points to the address of the next word to be pushed) is incremented on the same rising edge of `clk`—the first clock after `push_req_n` is asserted. This means that `push_req_n` must be asserted early enough to propagate through the FIFO controller to the RAM before the next clock.

Figure 1-2 Example of Asymmetric FIFO Controller Operation



Write Errors

An error occurs if a push is attempted while the FIFO is full. That is, if:

- The `push_req_n` input is asserted (LOW),
- The `full` flag is active (HIGH), and
- The `pop_req_n` input is inactive (HIGH), or there is more than one byte (or subword) left in the output buffer.

You should not use the DW_asymfifoctl_s1_df to perform a simultaneous push and pop when the RAM is full. For detailed information, refer to the topic titled "Simultaneous Push and Pop for data_in_width > data_out_width Case" on page 11.

Writing to the FIFO (Push) for data_in_width = data_out_width Case

In this case, the FIFO controller is a symmetric I/O FIFO controller. Its function is the same as DW_fifoctl_s1_df, except for the `part_wd`, `flush`, and `ram_full` pins, which are unused.

The `wr_addr` and `we_n` output ports of the FIFO controller provide the write address and synchronous write enable to the FIFO.

The `data_in` bus is connected directly to the `wr_data` bus and the `data_out` bus is connected directly to the `rd_data` bus.

A push is executed when the `push_req_n` input is asserted (LOW), and either:

- The `full` flag is inactive (LOW),

or:

- The `full` flag is active (HIGH), and
- The `pop_req_n` input is asserted (LOW).

Thus, a push can occur even if the FIFO is full, as long as a pop is executed in the same cycle.

Asserting `push_req_n` in either of the above cases causes the following events to occur:

- `The we_n` is asserted immediately, preparing for a write to the RAM on the next clock, and
- On the next rising edge of `clk`, `wr_addr` is incremented.

Thus, the RAM is written, and `wr_addr` (which always points to the address of the next word to be pushed) is incremented on the same rising edge of `clk`—the first clock after `push_req_n` is asserted. This means that `push_req_n` must be asserted early enough to propagate through the FIFO controller to the RAM before the next clock.

Write Errors

An error occurs if a push is attempted while the FIFO is full. That is, if:

- The `push_req_n` input is asserted (LOW),
- The `full` flag is active (HIGH), and
- The `pop_req_n` input is inactive (HIGH).

Writing to the FIFO (Push) for data_in_width < data_out_width Case

For cases where *data_in_width* < *data_out_width* (assuming that *data_out_width* = $K \times$ *data_in_width*, where *K* is an integer larger than 1), every byte (or subword) written to FIFO is first assembled into a full word with *data_out_width* bits. Refer to Figure 1-2 on page 6 for an example of this case.

The `wr_addr` and `we_n` output ports of the FIFO controller provide the write address and synchronous write enable to the FIFO.

A push of the partial word is executed at the rising edge of `clk` when the `push_req_n` input is asserted (LOW), and either:

- The `full` flag is inactive (LOW),

or,

- The `full` flag is active (HIGH), and
- The `pop_req_n` input is asserted (LOW).

Thus, a push can occur even if the FIFO is full, as long as a pop is executed in the same cycle.

For every byte (or subword) to be written, `push_req_n` toggles. Asserting `push_req_n` *K* times in either of the cases that enables a push causes the word accumulated in the input buffer (the first *K – 1* bytes are registered, the last byte is not. Refer to Figure 1-2) to be written to the next available location in the FIFO memory. This write occurs on the *clk* following the assertion of `push_req_n`.

The order of the input bytes within a word is determined by the `byte_order` parameter.

The data at the `data_in` port of the RAM must be stable for a setup time before the rising edge of `clk` and `push_req_n` must be asserted early enough to propagate through the FIFO controller to the RAM before the next clock.

In this way, the RAM is written, and `wr_addr` (which always points to the address of the next word to be pushed) is incremented on the same rising edge of `clk` – the first clock after `push_req_n` is asserted *K* times.

Partial Words

When a partial word is in the input buffer register, output flag `part_wd` is active (HIGH). After *K* times pushing, *K* bytes (or subwords) are assembled into a full word (*K* – 1 bytes in the input buffer register and the last byte on the `data_in` bus) by a combinational circuit. This achieves single clock cycle operation for the asymmetric FIFO controller.

This full word is then written into memory. When a full word is sent from the input buffer into memory, `part_wd` goes inactive (LOW).

The order of bytes within a word is determined by the `byte_order` parameter.

Flushing the RAM

A flush feature is provided for the *data_in_width* < *data_out_width* case. The flush feature pushes a partial word into memory when there are less than *K* bytes accumulated in the input buffer. The input buffer is cleared after a flush.

A flush is allowed:

- When *N* bytes have been read since the last complete word (where $0 < N < K$), and
- The sender device has no byte (or subword) to send at this moment,

while

- The higher level system requires that the receiver device be able to read these *N* bytes of data (from memory) without waiting, or,

- For data byte word alignment.

The sender device activates `flush_n` so that the *N* bytes data are pushed into memory without waiting for a complete word to be assembled.

When the receiver reads the partial word from the memory, the "leftover" bytes of the partial word *(K – N)* are filled with 0s.

A flush is executed when the `flush_n` input is asserted (LOW), and either:

- The `ram_full` flag is inactive (LOW),

or,

- The `ram_full` flag is active (HIGH), and
- The `pop_req_n` input is asserted (LOW)

at the rising edge of `clk`.

Asserting `flush_n` in either of the above cases causes the partial word accumulated in the input buffer to be written to the next available location in the FIFO memory. This write occurs on the `clk` following the assertion of `flush_n`.

Flushing the FIFO when the input buffer is empty (when the `part_wd` flag is inactive) is a "null" operation and does not cause an error.

Simultaneous Flush and Push, and Flush and Pop

Flush can occur at the same time as a push. When `flush_n` and `push_req_n` are active at the same time, the FIFO:

- Flushes the partial word in the input buffer, if any, into the memory, and
- Pushes the byte in the data_in bus into the input buffer

in the same clock cycle.

For a detailed description, refer to the topic titled "Reading from the FIFO (Pop) for data_in_width < data_out_width Case" on page 11.

Write Errors

An error occurs if a push is attempted while the FIFO is full. That is, if:

- The `push_req_n` input is active (LOW),
- The `empty` flag is active (HIGH), and
- The `pop_req_n` input is inactive (HIGH).

Reading from the FIFO (Pop) for *data_in_width > data_out_width* Case

For cases where *data_in_width* > *data_out_width* (assuming that *data_in_width* = $K \times$ *data_out_width*, where *K* is an integer larger than 1), the number of bits in a word stored in memory is *data_in_width*. The bit width for each out-going byte (or subword) is *data_out_width*.

For every byte (or subword) to be read, pop_req_n toggles. Each pop causes one byte (or subword) to be read. Toggling pop_req_n *K* times results in one full word (*data_in_width* bits) being read.

The order of the output bytes within a word is determined by the byte_order parameter.

The read port of the memory can be either synchronous or asynchronous. In either case, the data_out output port of the DW_asymfifoctl_s1_df provides prefetchable data (the next byte of memory read data to be read) to the output logic.

For RAMs with a synchronous read port, the output data is captured in the output stage of the memory. For RAMs with an asynchronous read port, the output data is captured by the next stage of logic after the FIFO.

A pop operation occurs when pop_req_n is asserted (LOW) when the FIFO is not empty. Asserting pop_req_n when the output buffer is not empty causes the data_out output port to be switched to the next byte (or subword) on the next rising edge of clk. Thus, memory read data must be captured on the clk following the assertion of pop_req_n.

Refer to the timing diagrams for details of the pop operation.

Read Errors

An error occurs if:

- The pop_req_n input is active (LOW), and
- The empty flag is active (HIGH).

Reading from the FIFO (Pop) for data_in_width = data_out_width Case

In this case, the FIFO controller is a symmetric FIFO controller. Its function is the same as the DW_fifoctl_s1_df, except for the part_wd, flush, and ram_full pins, which are unused.

The read port of the RAM can be either synchronous or asynchronous. In either case, the rd_addr output port of the DW_fifoctl_s1_sf provides the read address to the RAM. The rd_addr output bus always points to, thus prefetches, the next word of RAM read data to be popped.

A pop operation occurs when pop_req_n is asserted (LOW), as long as the FIFO is not empty. Asserting pop_req_n causes the rd_addr pointer to be incremented on the next rising edge of clk. Thus, the RAM read data must be captured on the clk following the assertion of pop_req_n. For RAMs with a synchronous read port, the output data is captured in the output stage of the RAM. For RAMs with an asynchronous read port, the output data is captured by the next stage of logic after the FIFO.

Read Errors

An error occurs if:

- The pop_req_n input is active (LOW), and
- The empty flag is active (HIGH).

Reading from the FIFO (Pop) for data_in_width < data_out_width Case

For cases where *data_in_width* < *data_out_width* (assuming that *data_out_width* = $K \times$ *data_in_width*, where *K* is an integer larger than 1), the number of bits in a word stored in memory is *data_out_width*. The `rd_data` bus is connected directly to the `data_out` bus.

The read port of the RAM can be either synchronous or asynchronous. In either case, the byte (or subword) to be read is available for prefetching at the FIFO `data_out` output port.

For RAMs with a synchronous read port, output data is captured in the output stage of the RAM. For RAMs with an asynchronous read port, output data is captured by the next stage of logic after the FIFO.

A pop operation occurs when `pop_req_n` is asserted (LOW) as long as the FIFO is not empty. The operation occurs on the next rising edge of `clk`. Thus, the RAM read data must be captured on the `clk` following the assertion of `pop_req_n`.

Refer to the timing diagrams for details of the pop operation.

Read Errors

An error occurs if:

- The `pop_req_n` input is active (LOW), and
- The `empty` flag is active (HIGH).

Simultaneous Push and Pop for data_in_width > data_out_width Case

You should not use the DW_asymfifoctl_s1_df to perform a simultaneous push and pop when the RAM is full.

For *data_in_width* > *data_out_width* (*data_in_width* = $K \times$ *data_out_width*) cases, push and pop can occur at the same time if:

- The FIFO is neither full nor empty, or
- The FIFO is full but there is only one byte (or subword) in the output buffer.

With the FIFO neither full nor empty (both `full` and `empty` signals inactive), the byte to be read is available for prefetching at the FIFO `data_out` output port.

When `pop_req_n` and `push_req_n` are both asserted, the following events occur on the next rising edge of `clk`:

- Pop data is captured by the next stage of logic after the FIFO, and
- Write data is pushed into the location pointed to by `wr_addr`.

When the FIFO is full, a simultaneous push and pop can occur only if $K - 1$ bytes of the word in the output buffer have been already read, and there is only one byte (or subword) left to be read in the output buffer; otherwise, simultaneous push and pop causes an overflow error. Refer to Figure 1-2 on page 6.

There are no flags that indicate a valid or invalid condition for a simultaneous push and pop when the FIFO is full. Designers who want an indication of this condition should create the necessary logic external to the FIFO controller.

When the FIFO is empty, simultaneous push and pop causes an error, since there is no pop data to prefetch.

Simultaneous Push and Pop for data_in_width = data_out_width Case

In this case, the FIFO controller is a symmetric FIFO controller. Its function is the same as DW_fifoctl_s1_df, except for the `part_wd`, `flush`, and `ram_full` pins, which are unused. The `data_in` bus is connected directly to `wr_data`, and `rd_data` is connected directly to the `data_out` bus.

Push and pop can occur at the same time if there is data in the FIFO, even when the FIFO is full. With the FIFO not empty, `rd_addr` is pointing to the next address to be popped, and the pop data is available to be prefetched at the RAM output.

When `pop_req_n` and `push_req_n` are both asserted, the following events occur on the next rising edge of `clk`:

- Pop data is captured by the next stage of logic after the FIFO, and
- The new data is pushed into the same location from which the data was popped.

Thus, there is no conflict in a simultaneous push and pop when the FIFO is full. A simultaneous push and pop cannot occur when the FIFO is empty, since there is no pop data to prefetch.

Simultaneous Push and Pop for data_in_width < data_out_width Case

For *data_in_width* < *data_out_width* (*data_out_width* = *K* × *data_in_width*) cases, a push (or flush) and pop can occur at the same time if the FIFO is not empty. With the FIFO not empty (`empty` active), pop data is available to be prefetched at the FIFO (and the RAM) output.

When `pop_req_n` and `push_req_n` are both asserted, the following events occur on the next rising edge of `clk`:

- Pop data is captured by the next stage of logic after the FIFO,
- Write data is pushed into the input buffer, which may in turn be pushed into the next available memory location after *K* pushes, and
- For a flush, the partial word in the input buffer is pushed into the next available memory location. The input buffer is cleared after the flush.

For *data_in_width* < *data_out_width* cases, there is no conflict in a simultaneous push and pop when the FIFO is full, because the bit width of the outgoing word is larger than that of the incoming byte (or subword), and the incoming data speed is slower than the outgoing data speed.

When the FIFO is empty, a simultaneous push and pop causes an error, since there is no pop data to prefetch.

Reset

rst_mode
This parameter selects whether reset is:

- Asynchronous (`rst_mode` = 0), or
- Synchronous (`rst_mode` = 1).

If the asynchronous mode is selected, asserting `rst_n` (setting it LOW) immediately causes the:

- Internal address pointers to be set to 0,

- Input or output buffer to be reset, and
- Flags and error output to be initialized.

If the synchronous mode is selected, the internal address pointers, flags, and error outputs are initialized at the rising edge of `clk` after `rst_n` is asserted.

The error output and flags are initialized as follows:

- `The empty` and `almost_empty` are initialized to 1, and
- All other flags and the `error` output are initialized to 0.

Errors

err_mode
The `err_mode` parameter determines which possible fault conditions are detected, and whether the `error` output remains active until reset or only for the clock cycle in which the error was detected.

When the `err_mode` parameter is set to 0 at design time, the `diag_n` input provides an unconditional synchronous reset to the value of the `rd_addr` output port. This can be used to intentionally cause the FIFO address pointers to become corrupted, forcing a pointer inconsistency-type error.

For normal operation when `err_mode` = 0, `diag_n` should be driven inactive (HIGH). When the `err_mode` parameter is set to 1 or 2, the `diag_n` input is ignored (unconnected).

error
The `error` output indicates a fault in the operation of the FIFO control logic. There are several possible causes for the `error` output to be activated:

1. Overflow (push with no pop while `full`; or, flush while `ram_full` for *data_in_width* < *data_out_width* case; or, push when `full` is active and the output buffer has more than one byte for *data_in_width* > *data_out_width* case).
2. Underflow (pop while empty).
3. Empty pointer mismatch (`rd_addr` ≠ `wr_addr` when empty).
4. Full pointer mismatch (`rd_addr` ≠ `wr_addr` when full).
5. In between pointer mismatch (`rd_addr` = `wr_addr` when neither empty nor full).

When `err_mode` = 0, all five causes are detected, and the `error` output (once activated) remains active until reset. When `err_mode` = 1, only causes 1 and 2 are detected, and the `error` output (once activated) remains active until reset. When `err_mode` = 2, only causes 1 and 2 are detected, and the `error` output only stays active for the clock cycle in which the `error` is detected. Refer to Table 1-5 on page 4 for error mode descriptions. The `error` output is set LOW when `rst_n` is applied.

Controller Status Flag Outputs

Refer to Figure 1-3 on page 15 for operation of the status flags.

empty
The `empty` output indicates that there are no words or bytes in the FIFO available to be popped. The `empty` output is set HIGH when `rst_n` is applied.

almost_empty
The `almost_empty` output is asserted when there are no more than `ae_level` words currently in the FIFO available to be popped. The value present on the `ae_level` port defines the almost empty threshold. The `almost_empty` output is updated only on the rising edge of `clk`. This signal is useful for preventing the FIFO from underflowing. The `almost_empty` output is set HIGH when `rst_n` is applied.

half_full
The `half_full` output is active (HIGH) when at least half the FIFO memory locations are occupied. The `half_full` output is set LOW when `rst_n` is applied.

almost_full
The `almost_full` output is asserted when there are no more than *depth* – `af_thresh` empty locations in the FIFO. The value present on the `af_thresh` port defines the almost full threshold. The `almost_full` output is updated only on the rising edge of `clk`. This signal is useful for preventing the FIFO from overflowing. The `almost_full` output is set LOW when `rst_n` is applied.

full
The `full` output indicates that the FIFO is full, and there is no space available for push data. The `full` output is set LOW when `rst_n` is applied.

ram_full
The `ram_full` output is used for the *data_in_width* < *data_out_width* case. The `ram_full` output indicates that the RAM is full, and there is no space available for flushing a partial word into the RAM. The `ram_full` output is set LOW when `rst_n` is applied.

For *data_in_width* ≤ *data_out_width*, `ram_full` is tied to the `full` output.

part_wd
This flag is only used for the *data_in_width* < *data_out_width* case. The `part_wd` output indicates that the FIFO has a partial word accumulated in the input buffer. The `part_wd` output is set LOW when `rst_n` is applied.

For *data_in_width* ≥ *data_out_width*, `part_wd` is tied LOW since the input data is always a full word.

Application Notes

The `ae_level` value is supplied by the application, and is chosen:

- To allow input flow control logic to interrupt the pushing of data into the FIFO, or
- to Give output flow control logic enough time to begin popping data.

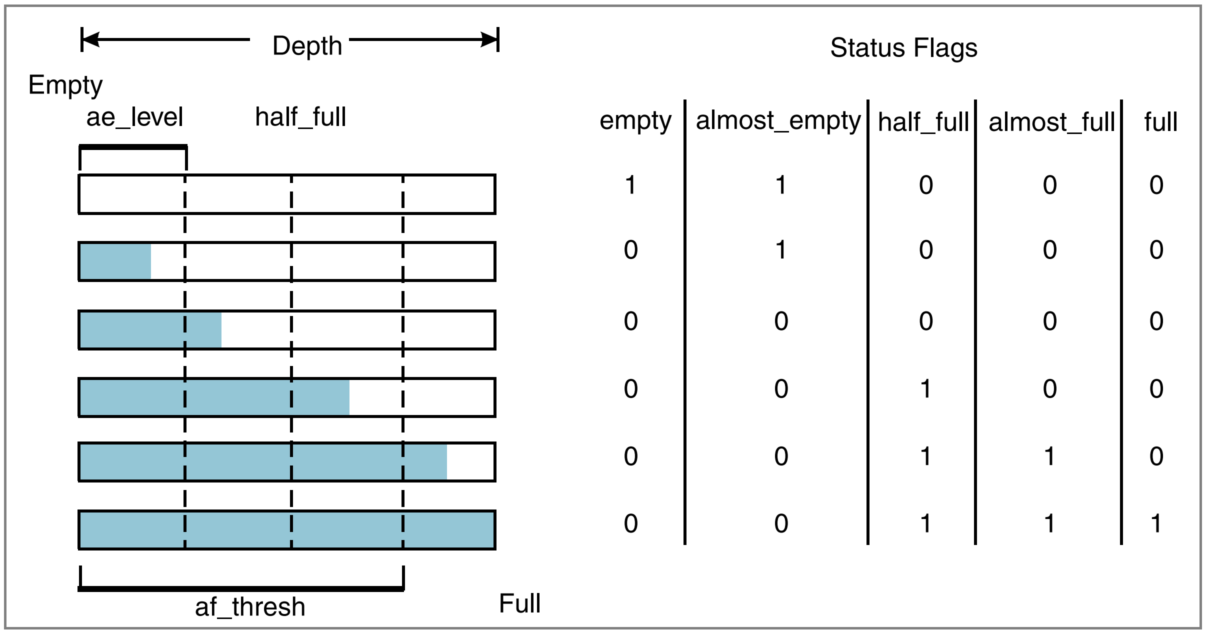
Systems can characterize their own response times dynamically against the data stream. This allows you to set the `ae_level` as tight as practical on the fly for optimal utilization of FIFO memory.

The `af_thresh` value is supplied by the application, and is chosen:

- To give output flow control logic enough time to begin popping data, or
- To allow input flow control logic to interrupt the pushing of data into the FIFO.

Systems can characterize their own response times dynamically against the data stream. This allows you to set the `almost_full` flag trip point on the fly for optimal utilization of FIFO memory.

Figure 1-3 DW_asymfifoctl_s1_df FIFO Status Flags



empty	almost_empty	half_full	almost_full	full
1	1	0	0	0
0	1	0	0	0
0	0	0	0	0
0	0	1	0	0
0	0	1	1	0
0	0	1	1	1

Timing Waveforms

The following figures show timing diagrams for various conditions of DW_asymfifoctl_s1_df.

Figure 1-4 Status Flag Timing Waveforms for data_in_width > data_out_width

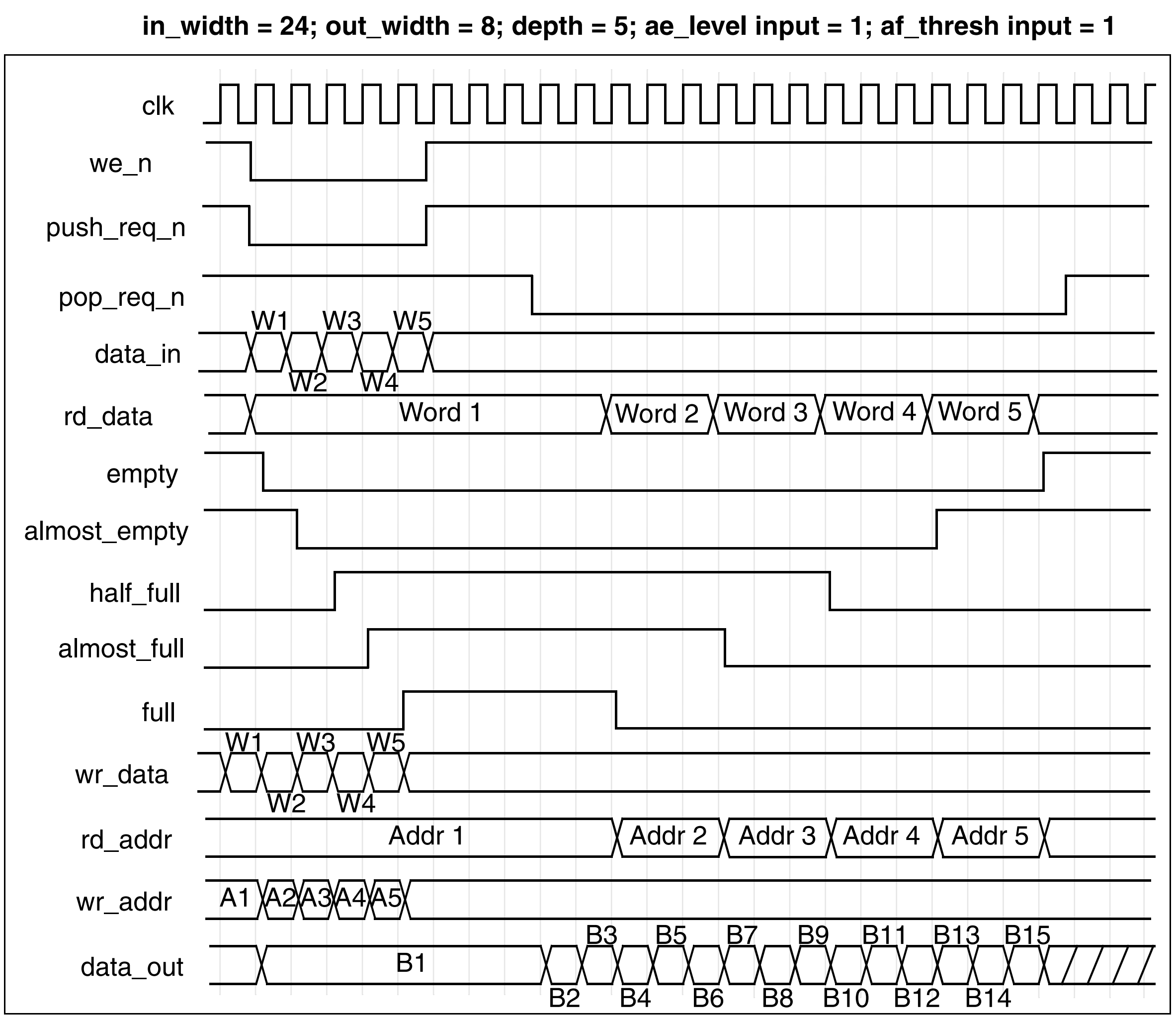
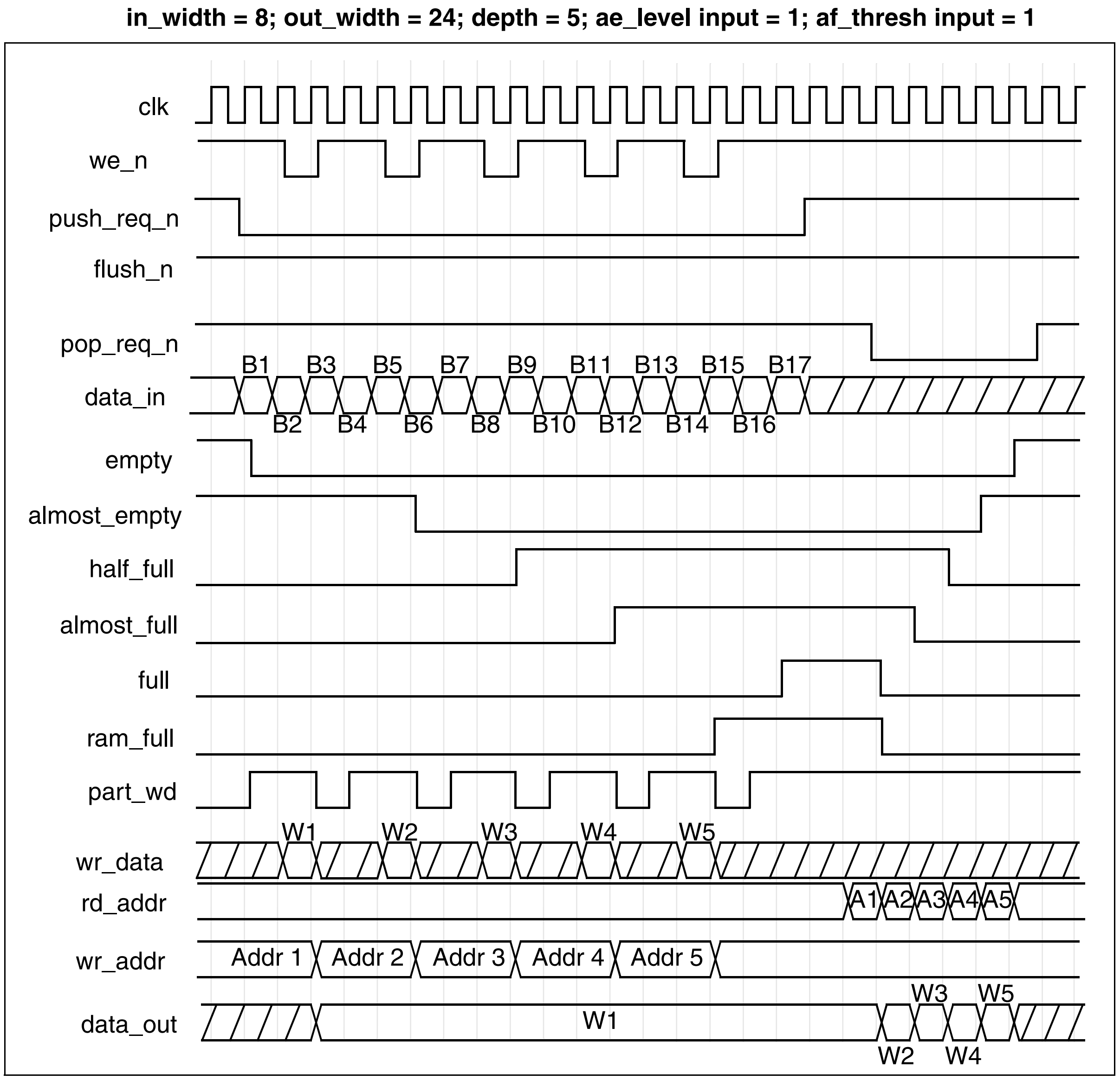


Figure 1-5 Status Flag Timing Waveforms for data_in_width < data_out_width


in_width = 8; out_width = 24; depth = 5; ae_level input = 1; af_thresh input = 1
clk
we_n
push_req_n
flush_n
pop_req_n
data_in
B1 B3 B5 B7 B9 B11 B13 B15 B17
B2 B4 B6 B8 B10 B12 B14 B16
empty
almost_empty
half_full
almost_full
full
ram_full
part_wd
wr_data
W1 W2 W3 W4 W5
rd_addr
A1 A2 A3 A4 A5
wr_addr
Addr 1 Addr 2 Addr 3 Addr 4 Addr 5
data_out
W1
W3 W5
W2 W4

* Note: B16 and B17 are only two of three slices needed for what would be W6 (not shown). B16 and B17 are waiting in a 2-stage assembly buffer in this case, as shown in Figure 1-2 on page 6, where B16 and B17 represent byte1 and byte2, respectively, for data_in_width=8.

Figure 1-6 Status Flag Timing Waveforms for Flush Operation

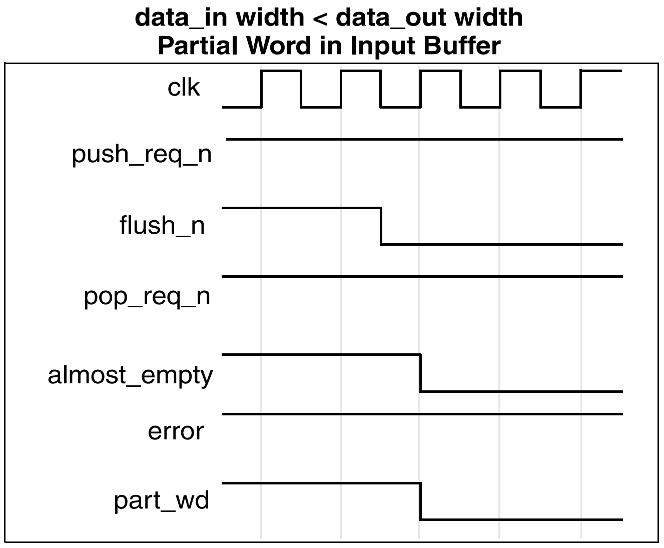
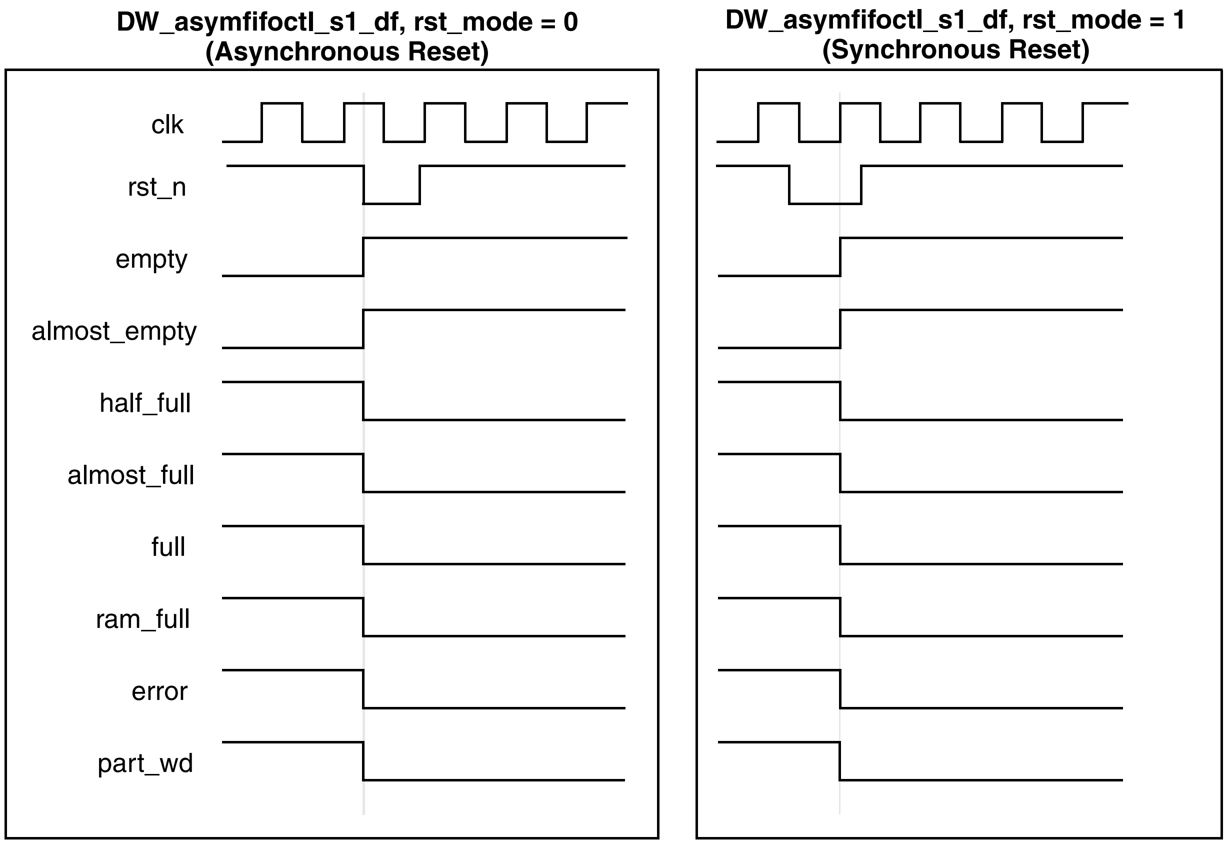


Figure 1-7 Reset Timing Waveforms



Related Topics

- Memory – FIFO Overview
- DesignWare Building Block IP Documentation Overview

HDL Usage Through Component Instantiation - VHDL

```
library IEEE,DWARE,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DWARE.DW_foundation_comp.all;

entity DW_asymfifoctl_s1_df_inst is
  generic (inst_data_in_width   : INTEGER := 8;
           inst_data_out_width  : INTEGER := 16;
           inst_depth           : INTEGER := 8;
           inst_err_mode        : INTEGER := 1;
           inst_rst_mode        : INTEGER := 1;
           inst_byte_order      : INTEGER := 0 );
  port (inst_clk           : in std_logic;
        inst_rst_n         : in std_logic;
        inst_push_req_n    : in std_logic;
        inst_flush_n       : in std_logic;
        inst_pop_req_n     : in std_logic;
        inst_diag_n        : in std_logic;
        inst_data_in    : in std_logic_vector(inst_data_in_width-1 downto 0);
        inst_rd_data    : in std_logic_vector(maximum(inst_data_in_width,
                                             inst_data_out_width)-1 downto 0);
      inst_ae_level   : in std_logic_vector(bit_width(inst_depth)-1 downto 0);
      inst_af_thresh  : in std_logic_vector(bit_width(inst_depth)-1 downto 0);
        we_n_inst          : out std_logic;
        empty_inst         : out std_logic;
        almost_empty_inst  : out std_logic;
        half_full_inst     : out std_logic;
        almost_full_inst   : out std_logic;
        full_inst          : out std_logic;
        ram_full_inst      : out std_logic;
        error_inst         : out std_logic;
        part_wd_inst       : out std_logic;
        wr_data_inst       : out std_logic_vector(maximum(inst_data_in_width,
                                             inst_data_out_width)-1 downto 0);
       wr_addr_inst : out std_logic_vector(bit_width(inst_depth)-1 downto 0);
       rd_addr_inst : out std_logic_vector(bit_width(inst_depth)-1 downto 0);
       data_out_inst : out std_logic_vector(inst_data_out_width-1 downto 0)       );
end DW_asymfifoctl_s1_df_inst;
```

```
architecture inst of DW_asymfifoctl_s1_df_inst is
begin

  -- Instance of DW_asymfifoctl_s1_df
  U1 : DW_asymfifoctl_s1_df
    generic map (data_in_width => inst_data_in_width,
                 data_out_width => inst_data_out_width,
                 depth => inst_depth,   err_mode => inst_err_mode,
                 rst_mode => inst_rst_mode,   byte_order => inst_byte_order )
    port map (clk => inst_clk,   rst_n => inst_rst_n,
              push_req_n => inst_push_req_n,   flush_n => inst_flush_n,
              pop_req_n => inst_pop_req_n,   diag_n => inst_diag_n,
              data_in => inst_data_in,   rd_data => inst_rd_data,
              ae_level => inst_ae_level,   af_thresh => inst_af_thresh,
              we_n => we_n_inst,   empty => empty_inst,
              almost_empty => almost_empty_inst,
              half_full => half_full_inst,   almost_full => almost_full_inst,
              full => full_inst,   ram_full => ram_full_inst,
              error => error_inst,   part_wd => part_wd_inst,
              wr_data => wr_data_inst,   wr_addr => wr_addr_inst,
              rd_addr => rd_addr_inst,   data_out => data_out_inst );
end inst;

-- pragma translate_off
configuration DW_asymfifoctl_s1_df_inst_cfg_inst
 of DW_asymfifoctl_s1_df_inst is
  for inst
  end for; -- inst
end DW_asymfifoctl_s1_df_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_asymfifoctl_s1_df_inst(inst_clk, inst_rst_n, inst_push_req_n,
    inst_flush_n, inst_pop_req_n, inst_diag_n, inst_data_in, inst_rd_data,
    inst_ae_level, inst_af_thresh, we_n_inst, empty_inst, almost_empty_inst,
    half_full_inst, almost_full_inst, full_inst, ram_full_inst, error_inst,
    part_wd_inst, wr_data_inst, wr_addr_inst, rd_addr_inst, data_out_inst );

  parameter data_in_width = 8;
  parameter data_out_width = 16;
  parameter depth = 8;
  parameter err_mode = 1;
  parameter rst_mode = 1;
  parameter byte_order = 0;
  `define bit_width_depth 3 // ceil(log2(depth))

  input inst_clk;
  input inst_rst_n;
  input inst_push_req_n;
  input inst_flush_n;
  input inst_pop_req_n;
  input inst_diag_n;
  input [data_in_width-1 : 0] inst_data_in;
  input [((data_in_width > data_out_width)?
         data_in_width : data_out_width)-1 : 0] inst_rd_data;
  input [`bit_width_depth-1 : 0] inst_ae_level;
  input [`bit_width_depth-1 : 0] inst_af_thresh;
  output we_n_inst;
  output empty_inst;
  output almost_empty_inst;
  output half_full_inst;
  output almost_full_inst;
  output full_inst;
  output ram_full_inst;
  output error_inst;
  output part_wd_inst;
  output [((data_in_width > data_out_width)?
          data_in_width : data_out_width)-1 : 0] wr_data_inst;
  output [`bit_width_depth-1 : 0] wr_addr_inst;
  output [`bit_width_depth-1 : 0] rd_addr_inst;
  output [data_out_width-1 : 0] data_out_inst;
```

```
    // Instance of DW_asymfifoctl_s1_df
    DW_asymfifoctl_s1_df #(data_in_width, data_out_width, depth, err_mode,
                           rst_mode, byte_order)
      U1 (.clk(inst_clk),   .rst_n(inst_rst_n),   .push_req_n(inst_push_req_n),
          .flush_n(inst_flush_n),   .pop_req_n(inst_pop_req_n),
          .diag_n(inst_diag_n),   .data_in(inst_data_in),
          .rd_data(inst_rd_data),   .ae_level(inst_ae_level),
          .af_thresh(inst_af_thresh),   .we_n(we_n_inst),   .empty(empty_inst),
           .almost_empty(almost_empty_inst),   .half_full(half_full_inst),
           .almost_full(almost_full_inst),   .full(full_inst),
           .ram_full(ram_full_inst),   .error(error_inst),
           .part_wd(part_wd_inst),   .wr_data(wr_data_inst),
           .wr_addr(wr_addr_inst), .rd_addr(rd_addr_inst),
           .data_out(data_out_inst) );
  endmodule
```

Revision History

For notes about this release, see the *DesignWare Building Block IP Release Notes*.

For lists of both known and fixed issues for this component, refer to the STAR report.

For a version of this datasheet with visible change bars, click here.

Date	Release	Updates
October 2017	N-2017.09-SP1	■ Replaced the synthesis implementations in Table 1-3 on page 3 with the str implementation ■ Added this Revision History table and the document links on this page