DW_fifo_s2_sf

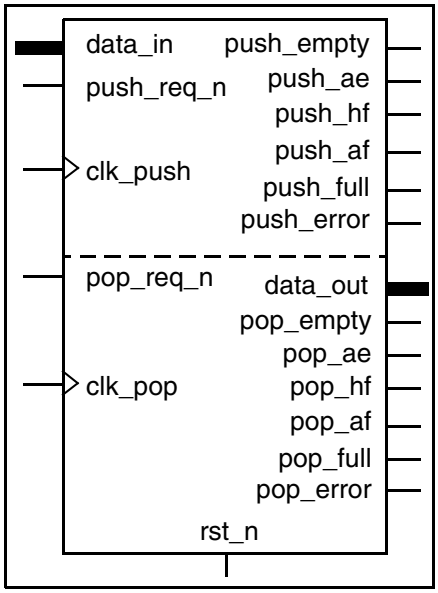
Synchronous (Dual-Clock) FIFO with Static Flags

Version, STAR and Download Information: IP Directory

Features and Benefits

Revision History

- Fully registered synchronous flag output ports
- Single clock cycle push and pop operations
- Parameterized word width
- Parameterized word depth
- Separate status flags for each clock system
- FIFO empty, half full, and full flags
- Parameterized almost full and almost empty flag thresholds
- FIFO push error (overflow) and pop error (underflow) flags



Description

DW_fifo_s2_sf is a dual independent clock FIFO. It combines the DW_fifoctl_s2_sf FIFO controller and the DW_ram_r_w_s_dff flip-flop-based RAM DesignWare components.

The FIFO provides parameterized width and depth, and a full complement of flags (full, almost full, half full, almost empty, empty, and error) for both of the clock domains.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
clk_push	1 bit	Input	Input clock for push interface
clk_pop	1 bit	Input	Input clock for pop interface
rst_n	1 bit	Input	Reset input, active low
push_req_n	1 bit	Input	FIFO push request, active low
pop_req_n	1 bit	Input	FIFO pop request, active low
data_in	*width* bit(s)	Input	FIFO data to push
push_empty	1 bit	Output	FIFO empty [a] output flag synchronous to `clk_push`, active high
push_ae	1 bit	Output	FIFO almost empty[a] output flag synchronous to `clk_push`, active high (determined by *push_ae_lvl* parameter)
push_hf	1 bit	Output	FIFO half full[a] output flag synchronous to `clk_push`, active high

Table 1-1 Pin Description (Continued)

Pin Name	Width	Direction	Function
push_af	1 bit	Output	FIFO almost full[a] output flag synchronous to `clk_push`, active high (determined by *push_af_lvl* parameter)
push_full	1 bit	Output	FIFO full[a] output flag synchronous to `clk_push`, active high
push_error	1 bit	Output	FIFO push error (overrun) output flag synchronous to `clk_push`, active high
pop_empty	1 bit	Output	FIFO empty [b] output flag synchronous to `clk_pop`, active high
pop_ae	1 bit	Output	FIFO almost empty[b] output flag synchronous to `clk_pop`, active high (determined by *pop_ae_lvl* parameter)
pop_hf	1 bit	Output	FIFO half full[b] output flag synchronous to `clk_pop`, active high
pop_af	1 bit	Output	FIFO almost full[b] output flag synchronous to `clk_pop`, active high (determined by *pop_af_lvl* parameter)
pop_full	1 bit	Output	FIFO full[b] output flag synchronous to `clk_pop`, active high
pop_error	1 bit	Output	FIFO pop error (under-run) output flag synchronous to `clk_pop`, active high
data_out	*width* bit(s)	Output	FIFO data to pop

a. As perceived by the push interface.
b. As perceived by the pop interface.

Table 1-2 Parameter Description

Parameter	Values	Description
width	1 to 2048 Default: 8	Width of the `data_in` and `data_out` buses
depth	4 to 1024 Default: 8	Number of words that can be stored in FIFO
push_ae_lvl	1 to *depth* - 1 Default: 2	Almost empty level for the `push_ae` output port (the number of words in the FIFO at or below which the `push_ae` flag is active)
push_af_lvl	1 to *depth* - 1 Default: 2	Almost full level for the `push_af` output port (the number of empty memory locations in the FIFO at which the `push_af` flag is active.)
pop_ae_lvl	1 to *depth* - 1 Default: 2	Almost empty level for the `pop_ae` output port (the number of words in the FIFO at or below which the `pop_ae` flag is active)
pop_af_lvl	1 to *depth* - 1 Default: 2	Almost full level for the `pop_af` output port (the number of empty memory locations in the FIFO at which the `pop_af` flag is active.)

Table 1-2 Parameter Description (Continued)

Parameter	Values	Description
err_mode	0 or 1 Default: 0	Error mode: ■ 0 = Stays active until reset [latched] ■ 1 = Active only as long as error condition exists [unlatched])
push_sync	1 to 3 Default: 2	Push flag synchronization mode: ■ 1 = Single register synchronization from pop pointer ■ 2 = Double register ■ 3 = Triple register
pop_sync	1 to 3 Default: 2	Pop flag synchronization mode: ■ 1 = Single register synchronization from push pointer ■ 2 = Double register ■ 3 = Triple register
rst_mode	0 to 3 Default: 0	Reset mode: ■ 0 = Asynchronous reset including memory ■ 1 = Synchronous reset including memory ■ 2 = Asynchronous reset excluding memory ■ 3 = Synchronous reset excluding memory

Table 1-3 Synthesis Implementations

Implementation Name	Function	License Feature Required
str	Synthesis model	DesignWare

Table 1-4 Simulation Models

Model	Function
DW06.DW_FIFO_S2_SF_CFG_SIM	Design unit name for VHDL simulation
dw/dw06/src/DW_fifo_s2_sf_sim.vhd	VHDL simulation model source code
dw/sim_ver/DW_fifo_s2_sf.v	Verilog simulation model source code

Table 1-5 Push Interface Function Table

push_req_n	push_full	Action	push_err
0	0	Push operation	No
0	1	Overrun; incoming data dropped (no action other than error generation)	Yes
1	X	No action	No

Table 1-6 Pop Interface Function Table

pop_req_n	pop_empty	Action	pop_err
0	0	Pop operation	No
0	1	Underrun (no action other than error generation)	Yes
1	X	No action	No

Simulation Methodology

DW_fifo_s2_sf contains synchronization of Gray coded pointers between clock domains for which there are two methods for simulation.

- The first method is to use the simulation models, which emulate the RTL model, with no modeling of metastable behavior. Using this method requires no extra action.
- The second method (only available for Verilog simulation models) is to enable modeling of random skew between bits of the Gray coded pointers that traverse to and from each domain.

  To use the second method, a Verilog preprocessing macro named DW_MODEL_MISSAMPLES must be defined in one of the following ways:

  - Specify the Verilog preprocessing macro in Verilog code:

    ```
    `define DW_MODEL_MISSAMPLES
    ```

  - Or, include a command line option to the simulator, such as +define+DW_MODEL_MISSAMPLES (which is used for the Synopsys VCS simulator)

Writing to the FIFO (Push)

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

With `push_req_n` input asserted and `push_full` inactive, the FIFO prepares to write the value at the `data_in` input port to the internal RAM. On the next `clk_push`, the data is written into the RAM and the internal write address pointer is advanced.

Write Errors

An error occurs if a push operation is attempted while the FIFO is full (as perceived by the push interface). That is, if:

- The `push_req_n` input is asserted (LOW), and
- The `push_full` flag is active (HIGH)

on the rising edge of `clk_push`, the `push_error` output goes active. When a push error (overrun) occurs, the data word that the application attempted to push onto the FIFO is lost. After the push error, other than the lost data, the FIFO remains in a valid full state and can continue to operate properly with respect to the data in the FIFO before the push error occurred.

For details of the push operation, refer to "Timing Waveforms" on page 13.

Reading from the FIFO (Pop)

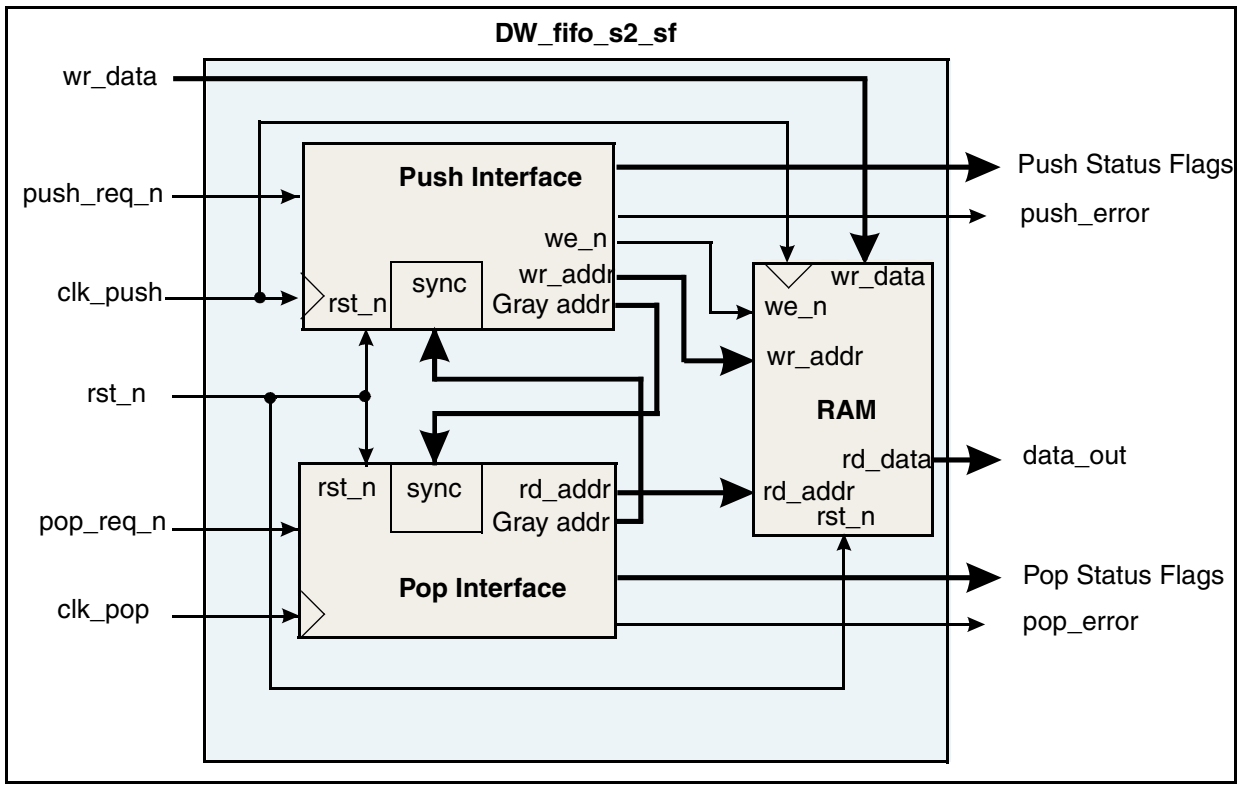
A pop operation occurs when `pop_req_n` is asserted (LOW) and the `pop_empty` flag is not active (LOW) (the FIFO is not empty).

Asserting `pop_req_n` while `pop_empty` is not active causes the internal read pointer to be incremented on the next rising edge of `clk_pop`. Thus, the RAM read data (available at the `data_out` output port) must be captured on the rising edge of `clk_pop` following the assertion of `pop_req_n`.

For details of the pop operation, see "Timing Waveforms" on page 13.

Figure 1-1 on page 6 shows an internal block diagram of the FIFO.

Figure 1-1 DW_fifo_s2_sf Block Diagram



Read Errors

An error occurs if a pop operation is attempted while the FIFO is empty (as perceived by the pop interface). That is, if:

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

on the rising edge of `clk_pop`, the `pop_error` output goes active. When a pop error (i.e. underrun) occurs, no action (other than the error flag) is taken. After the pop error, the FIFO remains in a valid empty state and can continue to operate properly.

Reset

rst_mode

This parameter selects whether reset is asynchronous (`rst_mode` = 0 or 2) or synchronous (`rst_mode` = 1 or 3).

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

- Internal address pointers are set to 0
- Flags and error outputs are initialized.

If a synchronous mode is selected, after the assertion of `rst_n`, the following are initialized at the rising edge of `clk_push`:

- Write address pointer
- Push flags
- `push_error` output

And at the rising edge of `clk_pop`, the following are initialized:

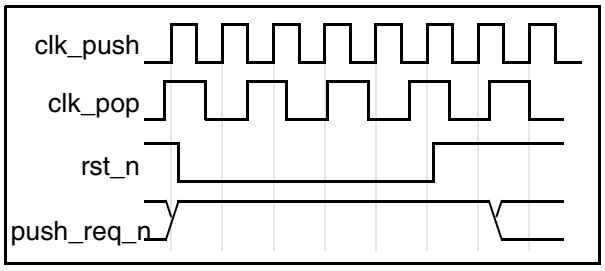
- Read address pointer
- Pop flags
- `pop_error` output

Metastability Issues Regarding Reset

To avoid metastability upon reset, assert `rst_n` (LOW) for at least three cycles of the slower of the two clock inputs, `clk_push` and `clk_pop`. During the assertion of `rst_n` and for at least one cycle of `clk_push` after `rst_n` goes high, `push_req_n` must be inactive (high). In addition, it is recommended that the `pop_req_n` signal also be held inactive for at least one clock cycle of `clk_pop` after the release of `rst_n`. For an example of this timing, see Figure 1-1.

Note Because the one input that is critical to proper reset sequencing (`push_req_n`) is in the domain of `clk_push`, it is recommended that the reset input, `rst_n`, be synchronous to `clk_push`.

Figure 1-2 Avoiding Metastability Upon Reset



Error Outputs and Flags Status

The error outputs and flags are initialized as follows:

- The `push_empty`, `push_ae`, `pop_empty`, and `pop_ae` are initialized to 1 (high), and
- All other flags and the error outputs are initialized to 0 (low).

If `rst_mode` = 0 or 1, the RAM array is also initialized when `rst_n` is asserted. If `rst_mode` = 2 or 3, only the address pointers and flag outputs are initialized; the RAM array is not initialized.

Synchronization Between Clock Systems

Each interface (push and pop) operates synchronous to its own clock: `clk_push` and `clk_pop`. Each interface is independent, containing its own state machine and flag logic. The pop interface also has the primary read address counter and a synchronized copy of the write address counter. The push interface also has the primary write address counter and a synchronized copy of the read address counter. The two clocks may be asynchronous with respect to each other. The FIFO controller performs inter-clock synchronization so each interface can monitor the actions of the other. This enables the number of words in the FIFO, at any given point in time, to be determined independently by the two interfaces.

The only information that is synchronized across clock domain boundaries is the read or write address generated by the opposite interface. If an address is transitioning while being sampled by the opposite interface (for example, `wr_addr` sampled by `clk_pop`), sampling uncertainty can occur. By Gray coding the address values that are synchronized across clock domains, this sampling uncertainty is limited to a single bit.

Single-bit sampling uncertainty results in only one of two possible Gray coded addresses being sampled: the previous address or the new address. The uncertainty in the bit that is changing near a sampling clock edge directly corresponds to an uncertainty in whether the new value is captured by the sampling clock edge or whether the previous value is captured (and the new value may be captured by a subsequent sampling clock edge). Thus, there are no errors in sampling Gray coded pointers, just a matter of whether a change of pointer value occurs in time to be captured by a given sampling clock edge or whether it must wait for the next sampling clock edge to be registered

push_sync and pop_sync
The `push_sync` and `pop_sync` parameters determine the number of register stages (1, 2 or 3) used to synchronize the internal Gray code read pointer to `clk_push` (for `push_sync`) and internal Gray code write pointer to `clk_pop` (for `pop_sync`). A value of 1 indicates single-stage synchronization, a value of 2 indicates double-stage synchronization, and a value of 3 indicates triple-stage synchronization.

Single-stage synchronization is only adequate when using very slow clock rates (with respect to the target technology). There must be enough timing slack to allow metastable synchronization events to stabilize and propagate to the pointer and flag registers.

Because timing slack and selection of register types are very difficult to control and metastability characteristics of registers are extremely difficult to ascertain, single-stage synchronization is not recommended.

Double-stage synchronization is desirable when using relatively high clock rates. It allows an entire clock period for metastable events to settle at the first stage before being cleanly clocked into the second stage of the synchronizer. Double-stage synchronization increases the latency between the two interfaces, resulting in flags that are less up to date with respect to the true state of the FIFO.

Triple-stage synchronization is desirable when using very high clock rates. It allows an entire clock period for metastable events to settle at the first stage before being clocked into the second stage of the synchronizer. Then, in the unlikely event that a metastable event propagates into the second stage, the output of the second stage is allowed to settle for another entire clock period before being clocked into the third stage. Triple-stage synchronization increases the latency between the two interfaces, resulting in flags that are less up to date with respect to the true state of the FIFO.

Empty to Not Empty Transitional Operation

When the FIFO is empty, both `push_empty` and `pop_empty` are active (high). During the first push (`push_req_n` active [low]), the rising edge of `clk_push` writes the first word into the FIFO. The `push_empty` flag is driven low.

The `pop_empty` flag does not go low until one cycle (of `clk_pop`) after the new internal Gray code write pointer has been synchronized to `clk_pop`. This can take as long as 2 to 4 cycles (depending on the value of the `pop_sync` parameter). For more information, see "Timing Waveforms" on page 13. The system design should allow for this latency in the depth budgeting of the FIFO design.

Not Empty to Empty Transitional Operation

When the FIFO is almost empty, both `push_empty` and `pop_empty` are inactive (low) and `pop_ae` is active (high). During the final pop (`pop_req_n` active [low]), the rising edge of `clk_pop` reads the last word out of the FIFO. The `pop_empty` flag is driven high.

The `push_empty` flag is not asserted (high) until one cycle (of `clk_push`) after the new internal Gray code read pointer is synchronized to `clk_push`. This can take as long as 2 to 4 cycles (depending on the value of the `push_sync` parameter). Refer to the timing diagrams for more information.

You should be aware of this latency when designing the system data flow protocol.

Full to Not Full Transitional Operation

When the FIFO is full, both `push_full` and `pop_full` are active (high) During the first pop (`pop_req_n` active [low]), the rising edge of `clk_pop` reads the first word out of the FIFO. The `pop_full` flag is driven low.

The `push_full` flag does not go low until one cycle (of `clk_push`) after the new internal Gray code read pointer has been synchronized to `clk_push`. This can take as long as 2 to 4 cycles (depending on the value of the `push_sync` parameter). For more information, see "Timing Waveforms" on page 13.

You should be aware of this latency when designing the system data flow protocol.

Not Full to Full Transitional Operation

When the FIFO is almost full, both `push_full` and `pop_full` are inactive [LOW], and `push_af` is active (high). During the final push (`push_req_n` active [low]), the rising edge of `clk_push` writes the last word into the FIFO. The `push_full` flag is driven high.

The `pop_full` flag is not asserted (high) until one cycle (of `clk_pop`) after the new internal Gray code write pointer has been synchronized to `clk_pop`. This can take as long as 2 to 4 cycles (depending on the value of the `pop_sync` parameter). For more information, see "Timing Waveforms" on page 13.

You should allow for this latency in the depth budgeting of the FIFO design.

Errors

err_mode
The `err_mode` parameter determines whether the `push_error` and `pop_error` outputs remain active until reset (persistent) or for only the clock cycle in which the error is detected (dynamic).

When the err_mode parameter is set to 0 at design time, persistent error flags are generated. When the err_mode parameter is set to 1 at design time, dynamic error flags are generated.

push_error
The push_error output signal indicates that a push request was seen while the push_full output was active (high) (an overrun error). When an overrun condition occurs, the write address pointer (wr_addr) cannot advance, and the RAM write enable (we_n) is not activated.

Therefore, a push request that would overrun the FIFO is, in effect, rejected, and an error is generated. This guarantees that no data already in the FIFO is destroyed (overwritten). Other than the loss of the data accompanying the rejected push request, FIFO operation can continue without reset.

pop_error
The pop_error output signal indicates that a pop request was seen while the pop_empty output signal was active (high) (an underrun error). When an underrun condition occurs, the read address pointer (rd_addr) cannot decrement because there is no data in the FIFO to retrieve.

The FIFO timing is such that the logic controlling the pop_req_n input would not see the error until "nonexistent" data had already been registered by the receiving logic. This is easily avoided if this logic can pay close attention to the pop_empty output and thus avoid an underrun completely.

Controller Status Flag Outputs

The two halves of the FIFO controller each have their own set of status flags to indicate their separate view of the state of the FIFO. It is important to note that both the push interface and the pop interface perceives the state of fullness of the FIFO independently, based on information from the opposing interface that is delayed up to three clock cycles for proper synchronization between clock domains.

The push interface status flags respond immediately to changes in state caused by push operations, but there is delay between pop operations and corresponding changes of state of the push status flags. This delay is due to the latency introduced by the registers used to synchronize the internal Gray coded read pointer to clk_push. The pop interface status flags respond immediately to changes in state caused by pop operations, but there is delay between push operations and corresponding changes of state of the pop status flags. This delay is due to the latency introduced by the registers used to synchronize the internal Gray coded write pointer to clk_pop.

Most status flags have a property that is potentially useful to the designed operation of the FIFO controller. These properties are described in the following explanations of the flag behaviors.

push_empty
The push_empty output, active high, is synchronous to the clk_push input. push_empty indicates to the push interface that the FIFO is empty. During the first push, the rising edge of clk_push causes the first word to be written into the FIFO, and push_empty is driven low.

The action of the last word being popped from a nearly empty FIFO is controlled by the pop interface. Thus, the push_empty output is asserted only after the new internal Gray code read pointer (from the pop interface) is synchronized to clk_push and processed by the status flag logic.

Property of push_empty
If push_empty is active (high), the FIFO is truly empty. This property does **not** apply to pop_empty.

push_ae
The `push_ae` output, active high, is synchronous to the `clk_push` input. The `push_ae` output indicates to the push interface that the FIFO is almost empty when there are no more than `push_ae_lvl` words currently in the FIFO to be popped, as perceived at the push interface.

The `push_ae_lvl` parameter defines the almost empty threshold of the push interface, independent of that of the pop interface. The `push_ae` output is useful when you want to push data into the FIFO in bursts (without allowing the FIFO to become empty).

Property of push_ae
If `push_ae` is active (high), the FIFO has at least (*depth* – *push_ae_lvl*) available locations. Thus, such status indicates that the push interface can safely and unconditionally push (*depth* – *push_ae_lvl*) words into the FIFO. This property guarantees that such a 'blind push' operation does not overrun the FIFO.

push_hf
The `push_hf` output, active high, is synchronous to the `clk_push` input, and indicates to the push interface that the FIFO has at least half of its memory locations occupied, as perceived by the push interface.

Property of push_hf
If `push_hf` is inactive (low), the FIFO has at least half of its locations available. Thus, such status indicates that the push interface can safely and unconditionally push (INT(*depth*/2) + 1) words into the FIFO. This property guarantees that such a 'blind push' operation does not overrun the FIFO.

push_af
The `push_af` output, active high, is synchronous to the `clk_push` input. The `push_af` output indicates to the push interface that the FIFO is almost full when there are no more than `push_af_lvl` empty locations in the FIFO as perceived by the push interface.

The `push_af_lvl` parameter defines the almost full threshold of the push interface independent of the pop interface. The `push_af` output is useful when more than one cycle of advance warning is needed to stop the flow of data into the FIFO before it becomes full (to avoid a FIFO overrun).

Property of push_af
If `push_af` is inactive (LOW) then the FIFO has at least (*push_af_lvl* + 1) available locations. Thus, such status indicates that the push interface can safely and unconditionally push (*push_af_lvl* + 1) words into the FIFO. This property guarantees that such a 'blind push' operation does not overrun the FIFO.

push_full
The `push_full` output, active high, is synchronous to the `clk_push` input. The `push_full` output indicates to the push interface that the FIFO is full. During the final push, the rising edge of `clk_push` causes the last word to be pushed, and `push_full` is asserted.

The action of the first word being popped from a full FIFO is controlled by the pop interface. Thus, the `push_full` output goes low only after the new internal Gray code read pointer from the pop interface is synchronized to `clk_push` and processed by the status flag state logic.

pop_empty
The `pop_empty` output, active high, is synchronous to the `clk_pop` input. The `pop_empty` output indicates to the pop interface that the FIFO is empty as perceived by the pop interface. The action of the last word being popped from a nearly empty FIFO is controlled by the pop interface. Thus, the `pop_empty` output is asserted at the rising edge of `clk_pop` that causes the last word to be popped from the FIFO.

The action of pushing the first word into an empty FIFO is controlled by the push interface. That means pop_empty goes low only after the new internal Gray code write pointer from the push interface is synchronized to clk_pop and processed by the status flag state logic.

pop_ae
The pop_ae output, active high, is synchronous to the clk_pop input. The pop_ae output indicates to the pop interface that the FIFO is almost empty when there are no more than *pop_ae_lvl* words currently in the FIFO to be popped, as perceived by the pop interface.

The pop_ae_lvl parameter defines the almost empty threshold of the pop interface, independent of the push interface. The pop_ae output is useful when more than one cycle of advance warning is needed to stop the popping of data from the FIFO before it becomes empty (to avoid a FIFO underrun).

> **Property of pop_ae**
> If pop_ae is inactive (low), there are at least (*pop_ae_lvl* +) words in the FIFO. Thus, such status indicates that the pop interface can safely and unconditionally pop (*pop_ae_lvl* + 1) words out of the FIFO. This property guarantees that such a 'blind pop' operation does not underrun the FIFO.

pop_hf
The pop_hf output, active high, is synchronous to the clk_pop input. pop_hf indicates to the pop interface that the FIFO has at least half of its memory locations occupied, as perceived by the pop interface.

> **Property of pop_hf**
> If pop_hf is active (high), at least half of the words in the FIFO are occupied. Thus, such status indicates that the pop interface can safely and unconditionally pop INT((*depth* + 1)/2) words out of the FIFO. This property guarantees that such a 'blind pop' operation does not underrun the FIFO.

pop_af
The pop_af output, active high, is synchronous to the clk_pop input. The pop_af indicates to the pop interface that the FIFO is almost full when there are no more than *pop_af_lvl* empty locations in the FIFO, as perceived by the pop interface.

The pop_af_lvl parameter defines the almost full threshold of the pop interface independent of that of the pop interface. The pop_af output is useful when you want to pop data out of the FIFO in bursts (without allowing the FIFO to become empty).

> **Property of pop_af**
> If pop_af is active (high), there are at least (*depth* – *pop_af_lvl*) words in the FIFO. Thus, such status indicates that the pop interface can safely and unconditionally pop (*depth* – *pop_af_lvl*) words out of the FIFO. This property guarantees that such a 'blind pop' operation does not underrun the FIFO.

pop_full
The pop_full output, active high, is synchronous to the clk_pop input. The pop_full output indicates to the pop interface that the FIFO is full, as perceived by the pop interface. The action of popping the first word out of a full FIFO is controlled by the pop interface. Thus, the pop_full output goes low at the rising edge of the clk_pop that causes the first word to be popped.

The action of the last word being pushed into a nearly full FIFO is controlled by the push interface. This means the pop_full output is asserted only after the new write pointer from the pop interface is synchronized to clk_pop and processed by the status flag state logic.

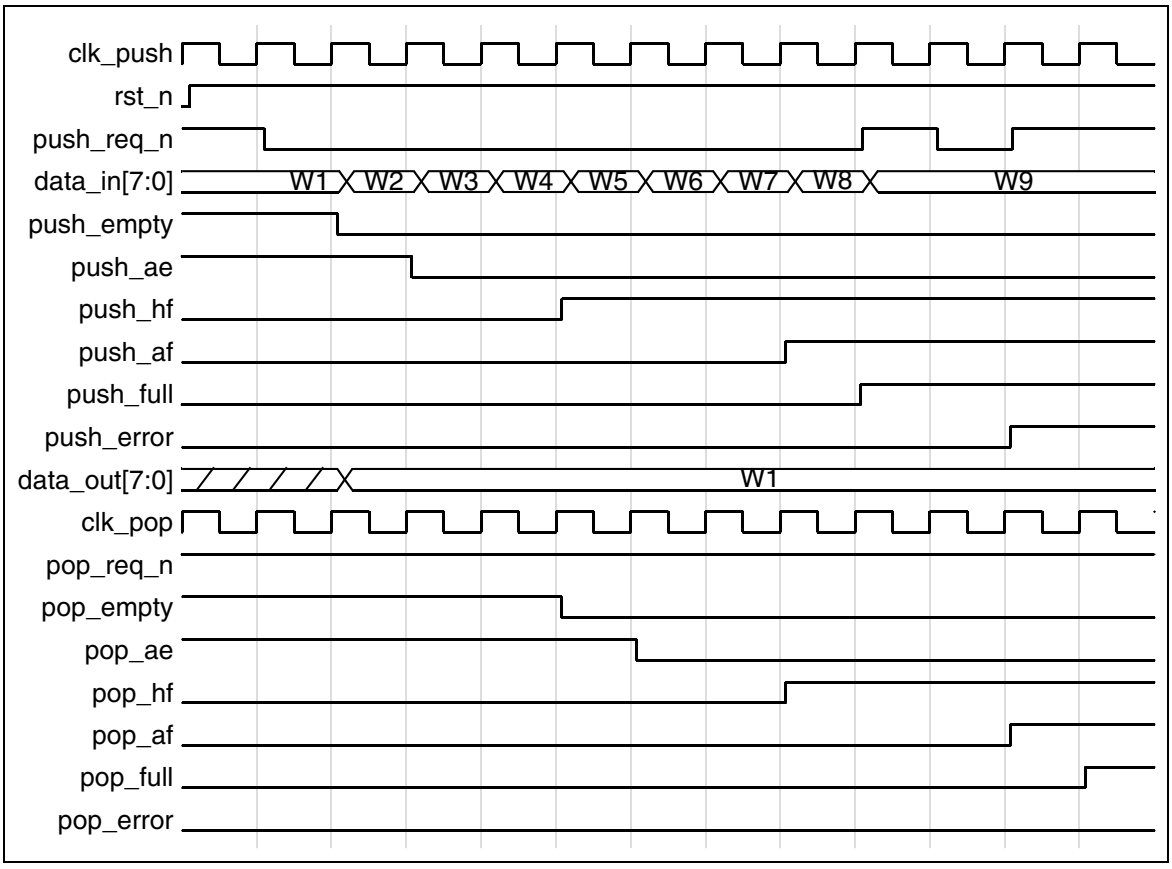
> **Property of pop_full**
> If pop_full is active (high) then the FIFO is truly full. This property does not apply to push_full.

Timing Waveforms

The following figures show timing diagrams for various conditions of DW_fifo_s2_sf.

Figure 1-3 Push Timing Waveforms

FIFO *depth* = 8 (Even 2^n Value); *push_ae_lvl* = 1; *push_af_lvl* = 1; *pop_ae_lvl* = 1; *pop_af_lvl* = 1; *push_sync* = 2; *pop_sync* = 2; *err_mode* = 0



# of Words Actually in FIFO

0	0	1	2	3	4	5	6	7	8	8	8	8

# of Words as Perceived by Push Interface

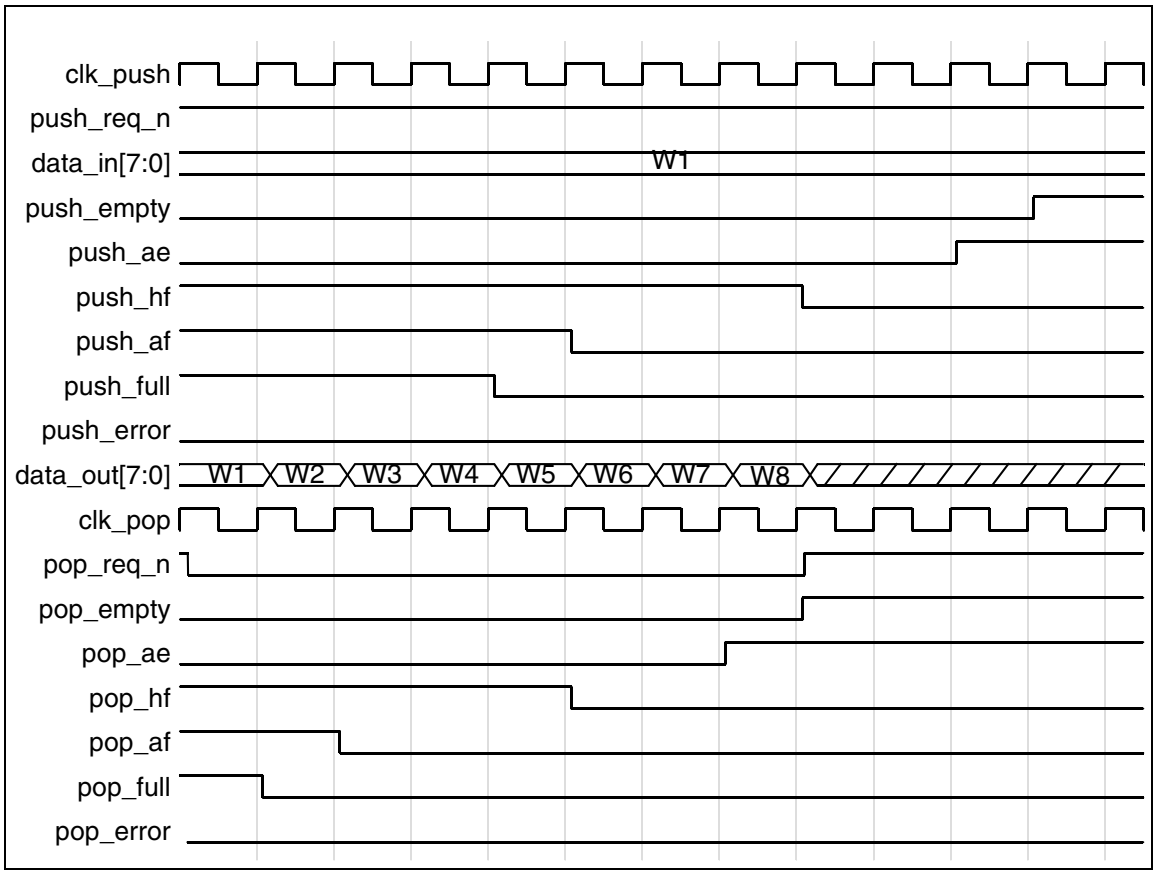
0	0	1	2	3	4	5	6	7	8	8	8	8

# of Words as Perceived by Pop Interface

0	0	0	0	0	1	2	3	4	5	6	7	8

Figure 1-4 Pop Timing Waveforms

FIFO *depth* = 8 (Even 2^n Value); *push_ae_lvl* = 1; *push_af_lvl* = 1; *pop_ae_lvl* = 1; *pop_af_lvl* = 1; *push_sync* = 2; *pop_sync* = 2



# of Words Actually in FIFO

8	7	6	5	4	3	2	1	0	0	0	0	0

# of Words as Perceived by Push Interface

8	8	8	8	7	6	5	4	3	2	1	0	0

# of Words as Perceived by Pop Interface

8	7	6	5	4	3	2	1	0	0	0	0	0

Figure 1-5 Single Word Push and Pop Timing Waveforms

FIFO *depth* = 8 (Even 2^n Value); *push_ae_lvl* = 1; *push_af_lvl* = 1; *pop_ae_lvl* = 1; *pop_af_lvl* = 1; *push_sync* = 2; *pop_sync* = 2

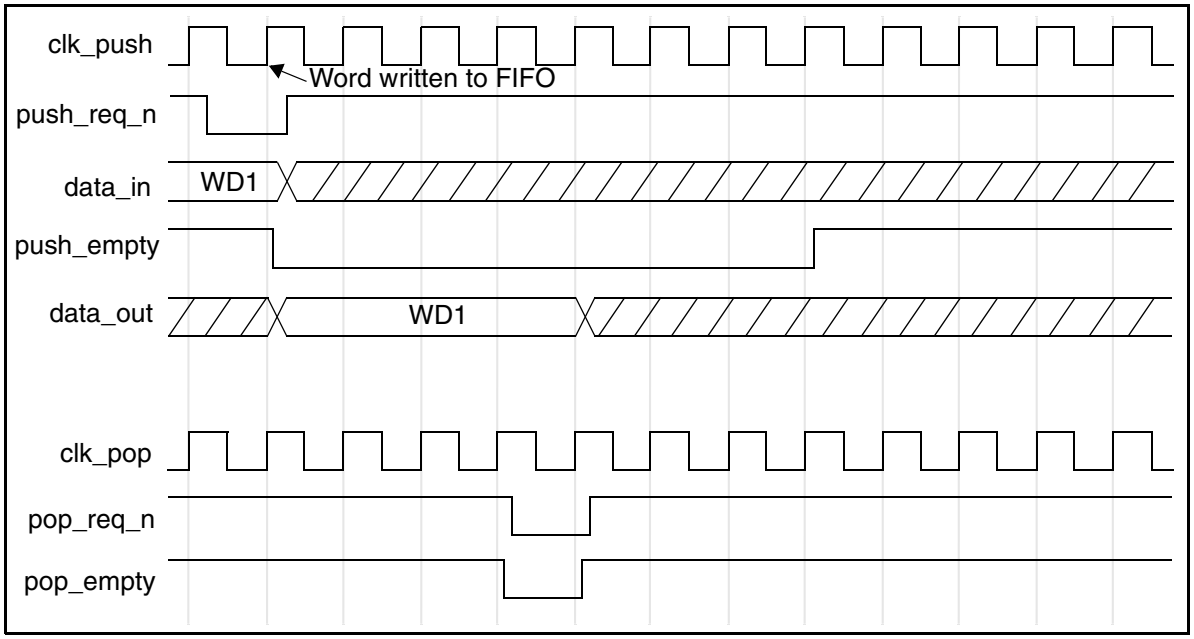


Figure 1-6 FIFO *depth* ≠ 2^n Push Timing Waveforms

FIFO *depth* = 9 (≠ 2^n Value); *push_ae_lvl* = 3; *push_af_lvl* = 3; *pop_ae_lvl* = 3; *pop_af_lvl* = 3; *push_sync* = 1; *pop_sync* = 1; *err_mode* = 1

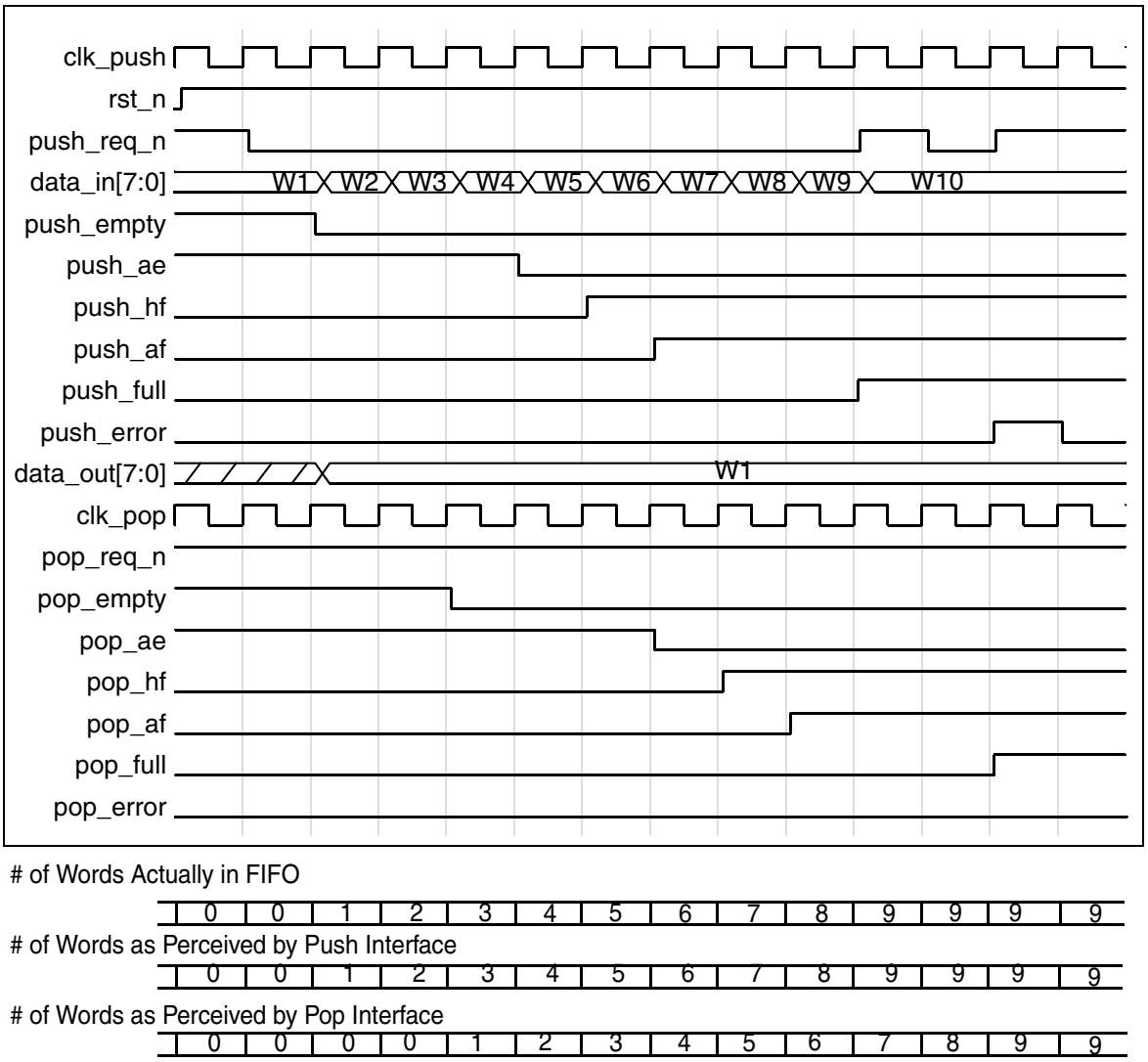


Figure 1-7 FIFO *depth* ≠ 2^n Pop Timing Waveforms

FIFO *depth* = 9 (≠ 2^n Value); *push_ae_lvl* = 3; *push_af_lvl* = 3; *pop_ae_lvl* = 3; *pop_af_lvl* = 3; *push_sync* = 1; *pop_sync* = 1

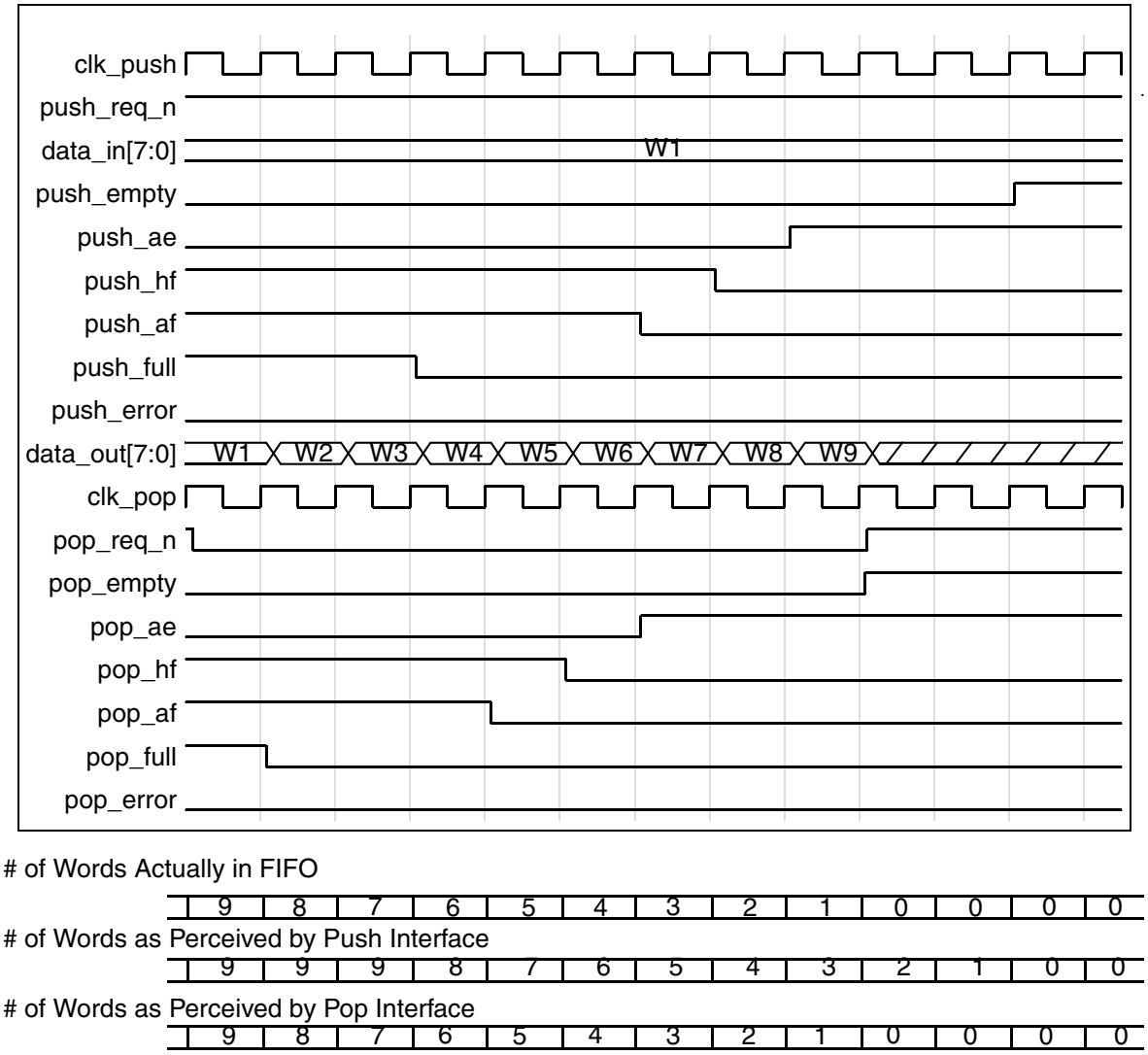


Figure 1-8 FIFO *depth* ≠ 2^n Single Word Timing Waveforms

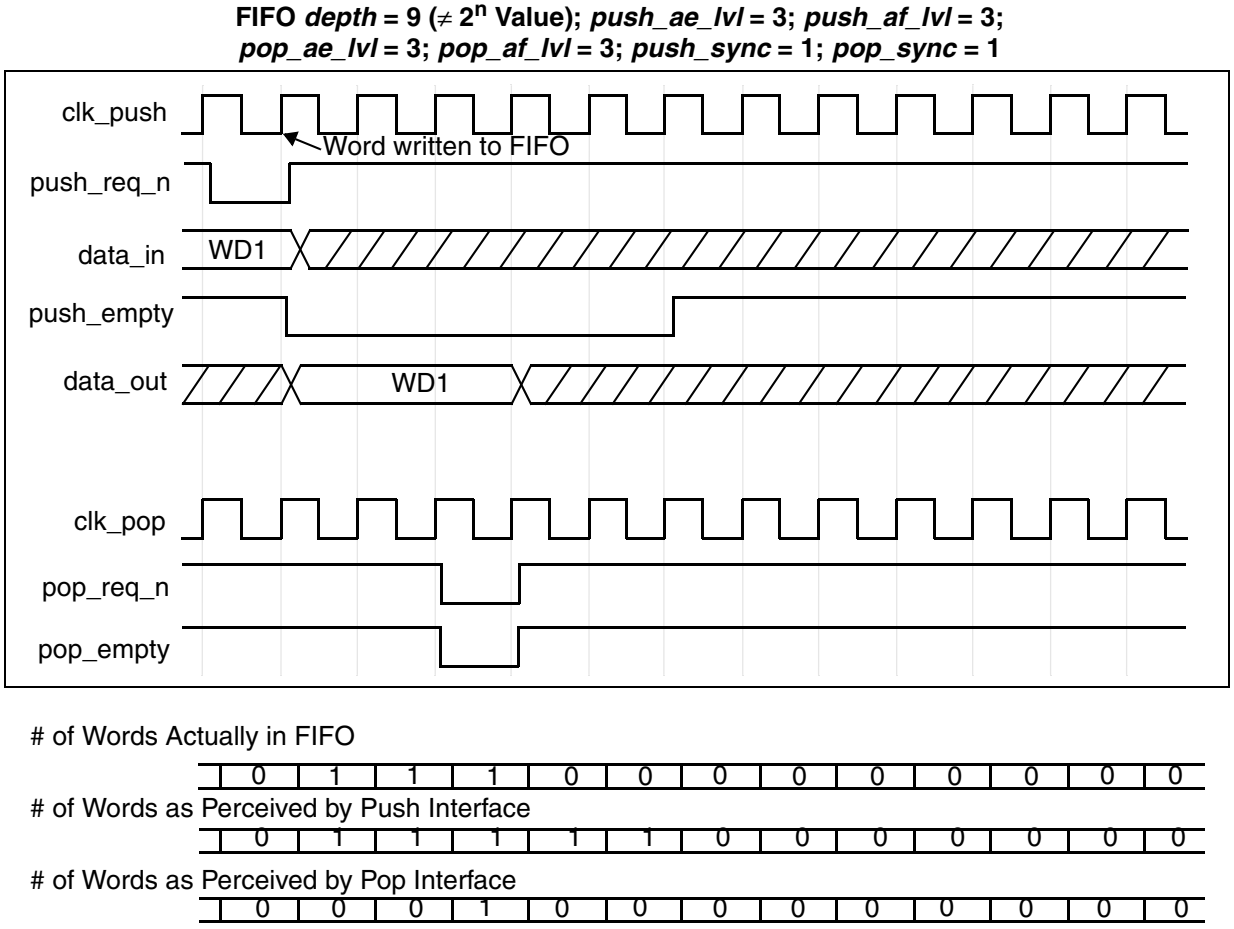
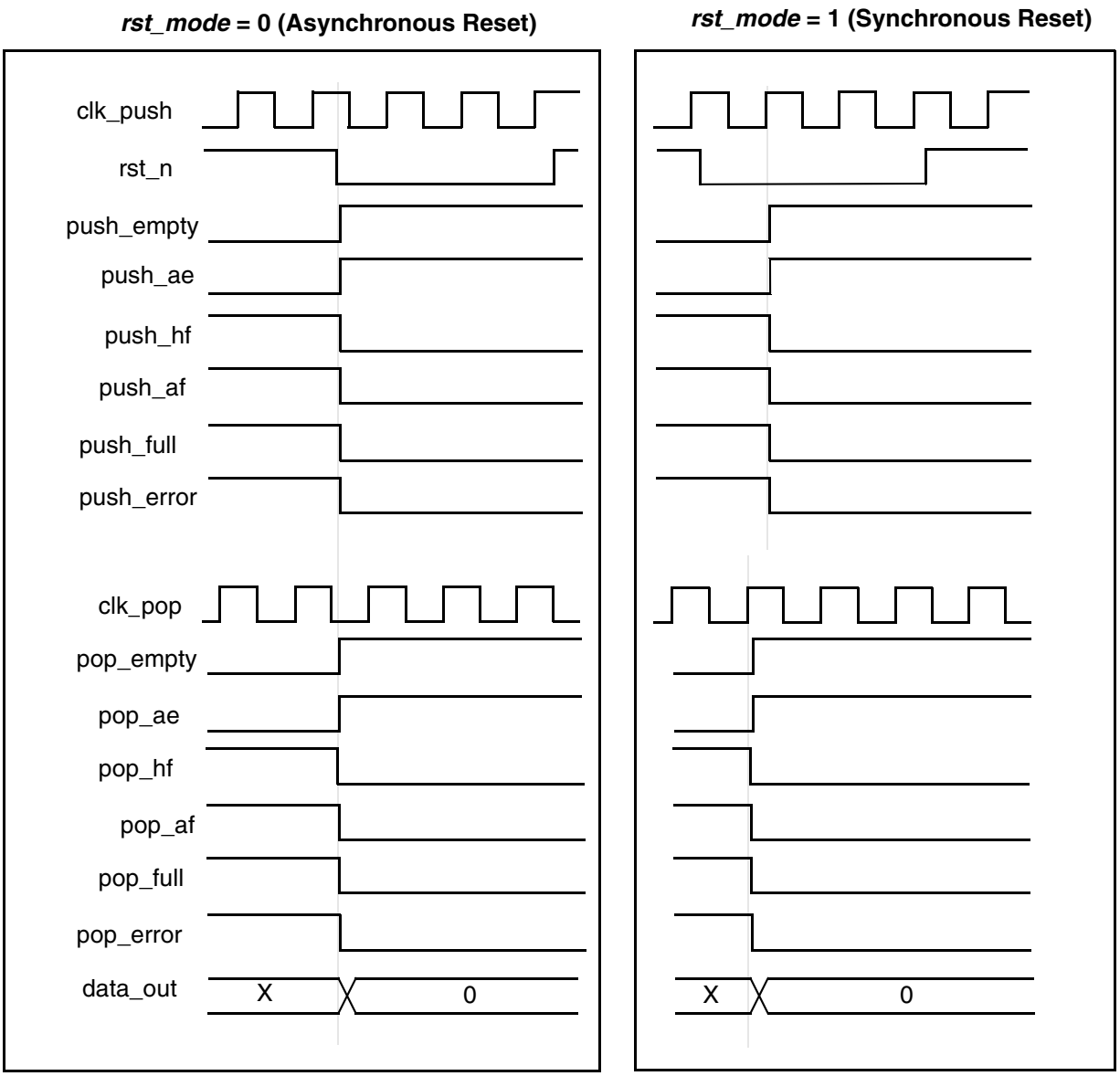


Figure 1-9 Reset Timing Waveforms



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_fifo_s2_sf_inst is
  generic (inst_width       : INTEGER := 8;
           inst_depth       : INTEGER := 8;
           inst_push_ae_lvl : INTEGER := 2;
           inst_push_af_lvl : INTEGER := 2;
           inst_pop_ae_lvl  : INTEGER := 2;
           inst_pop_af_lvl  : INTEGER := 2;
           inst_err_mode    : INTEGER := 0;
           inst_push_sync   : INTEGER := 2;
           inst_pop_sync    : INTEGER := 2;
           inst_rst_mode    : INTEGER := 0  );
  port (inst_clk_push   : in std_logic;
        inst_clk_pop    : in std_logic;
        inst_rst_n      : in std_logic;
        inst_push_req_n : in std_logic;
        inst_pop_req_n  : in std_logic;
        inst_data_in    : in std_logic_vector(inst_width-1 downto 0);
        push_empty_inst : out std_logic;
        push_ae_inst    : out std_logic;
        push_hf_inst    : out std_logic;
        push_af_inst    : out std_logic;
        push_full_inst  : out std_logic;
        push_error_inst : out std_logic;
        pop_empty_inst  : out std_logic;
        pop_ae_inst     : out std_logic;
        pop_hf_inst     : out std_logic;
        pop_af_inst     : out std_logic;
        pop_full_inst   : out std_logic;
        pop_error_inst  : out std_logic;
        data_out_inst   : out std_logic_vector(inst_width-1 downto 0) );
end DW_fifo_s2_sf_inst;

architecture inst of DW_fifo_s2_sf_inst is
begin
```

```
  -- Instance of DW_fifo_s2_sf
  U1 : DW_fifo_s2_sf
    generic map (width => inst_width,   depth => inst_depth,
                 push_ae_lvl => inst_push_ae_lvl,
                 push_af_lvl => inst_push_af_lvl,
                 pop_ae_lvl => inst_pop_ae_lvl,
                 pop_af_lvl => inst_pop_af_lvl,   err_mode => inst_err_mode,
                 push_sync => inst_push_sync,   pop_sync => inst_pop_sync,
                 rst_mode => inst_rst_mode )
    port map (clk_push => inst_clk_push,   clk_pop => inst_clk_pop,
              rst_n => inst_rst_n,   push_req_n => inst_push_req_n,
              pop_req_n => inst_pop_req_n,   data_in => inst_data_in,
              push_empty => push_empty_inst,   push_ae => push_ae_inst,
              push_hf => push_hf_inst,   push_af => push_af_inst,
              push_full => push_full_inst,   push_error => push_error_inst,
              pop_empty => pop_empty_inst,   pop_ae => pop_ae_inst,
              pop_hf => pop_hf_inst,   pop_af => pop_af_inst,
              pop_full => pop_full_inst,   pop_error => pop_error_inst,
              data_out => data_out_inst );
end inst;

-- pragma translate_off
configuration DW_fifo_s2_sf_inst_cfg_inst of DW_fifo_s2_sf_inst is
  for inst
  end for; -- inst
end DW_fifo_s2_sf_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_fifo_s2_sf_inst(inst_clk_push, inst_clk_pop, inst_rst_n,
                          inst_push_req_n, inst_pop_req_n, inst_data_in,
                          push_empty_inst, push_ae_inst, push_hf_inst,
                          push_af_inst, push_full_inst, push_error_inst,
                          pop_empty_inst, pop_ae_inst, pop_hf_inst,
                          pop_af_inst, pop_full_inst, pop_error_inst,
                          data_out_inst );

  parameter width = 8;
  parameter depth = 8;
  parameter push_ae_lvl = 2;
  parameter push_af_lvl = 2;
  parameter pop_ae_lvl = 2;
  parameter pop_af_lvl = 2;
  parameter err_mode = 0;
  parameter push_sync = 2;
  parameter pop_sync = 2;
  parameter rst_mode = 0;

  input inst_clk_push;
  input inst_clk_pop;
  input inst_rst_n;
  input inst_push_req_n;
  input inst_pop_req_n;
  input [width-1 : 0] inst_data_in;
  output push_empty_inst;
  output push_ae_inst;
  output push_hf_inst;
  output push_af_inst;
  output push_full_inst;
  output push_error_inst;
  output pop_empty_inst;
  output pop_ae_inst;
  output pop_hf_inst;
  output pop_af_inst;
  output pop_full_inst;
  output pop_error_inst;
  output [width-1 : 0] data_out_inst;
```

```
  // Instance of DW_fifo_s2_sf
  DW_fifo_s2_sf #(width,  depth,  push_ae_lvl,  push_af_lvl,  pop_ae_lvl,
                  pop_af_lvl,  err_mode,  push_sync,  pop_sync,  rst_mode)
    U1 (.clk_push(inst_clk_push),   .clk_pop(inst_clk_pop),
        .rst_n(inst_rst_n),   .push_req_n(inst_push_req_n),
         .pop_req_n(inst_pop_req_n),   .data_in(inst_data_in),
         .push_empty(push_empty_inst),   .push_ae(push_ae_inst),
         .push_hf(push_hf_inst),   .push_af(push_af_inst),
         .push_full(push_full_inst),   .push_error(push_error_inst),
         .pop_empty(pop_empty_inst),   .pop_ae(pop_ae_inst),
         .pop_hf(pop_hf_inst),   .pop_af(pop_af_inst),
         .pop_full(pop_full_inst),   .pop_error(pop_error_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
April 2018	N-2017.09-SP5	■ For STAR 9001317257, updated the maximum value for the *width* and *depth* parameters in Table 1-2 on page 2
December 2017	N-2017.09-SP2	■ Added "Simulation Methodology" on page 4 to explain how to simulate synchronization of Gray coded pointers between clock domains
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

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.

All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043

www.synopsys.com