DW_asymfifoctl_s2_sf

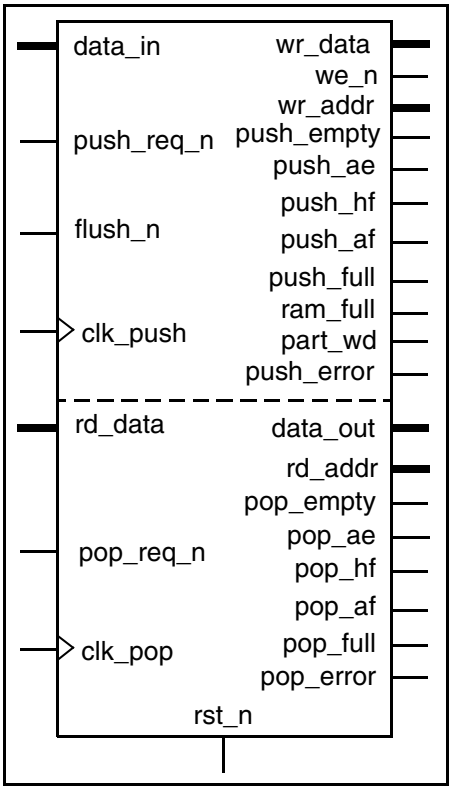
Asym. Synch. (Dual-Clock) FIFO Controller - Static Flags

Version, STAR and Download Information: IP Directory

Features and Benefits

- Parameterized asymmetric input and output bit widths (must be integer-multiple relationship)
- Parameterized word depth
- Fully registered synchronous flag output ports
- Separate status flags for each clock domain
- FIFO empty, half full, and full flags
- Parameterized almost full and almost empty flags
- FIFO push error (overflow) and pop error (underflow) flags
- Single clock cycle push and pop operations
- Parameterized byte order within a word
- Word integrity flag for *data_in_width* < *data_out_width*
- Partial word flush for *data_in_width* < *data_out_width*
- Interfaces to common hard macro or compiled ASIC dual-port synchronous RAMs
- Provides minPower benefits with the DesignWare-LP license (Get the minPower version of this datasheet.)

Revision History



Description

DW_asymfifoctl_s2_sf is an asymmetric I/O dual independent clock FIFO RAM controller. It is designed to interface with a dual-port synchronous RAM.

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

Table 1-1 Pin Description (Continued)

Pin Name	Width	Direction	Function
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
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
we_n	1 bit	Output	Write enable output for write port of RAM, active low
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
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's RAM full[a] output flag (including the input buffer of FIFO controller for *data_in_width* < *data_out_width*) synchronous to `clk_push`, active high
ram_full	1 bit	Output	FIFO's RAM (excluding the input buffer of FIFO controller for *data_in_width* < *data_out_width*) full output flag synchronous to `clk_push`, active high
part_wd	1 bit	Output	Partial word accumulated in the input buffer synchronous to `clk_push`, active high (for *data_in_width* < *data_out_width* only; otherwise, tied low)
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)

Table 1-1 Pin Description (Continued)

Pin Name	Width	Direction	Function
pop_full	1 bit	Output	FIFO's RAM full[b] output flag (excluding the input buffer of FIFO controller for case *data_in_width* < *data_out_width*) synchronous to `clk_pop`, active high
pop_error	1 bit	Output	FIFO pop error (under-run) output flag synchronous to `clk_pop`, active high
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

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

Table 1-2 Parameter Description

Parameter	Values	Description
data_in_width	1 to 4096 Default: 8	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 Default: 24	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	4 to 2^{24} 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: 28	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 or 1 Default: 1	Reset mode 0 = asynchronous reset, 1 = synchronous reset)
byte_order	0 or 1 Default: 0	Order of bytes or 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_S2_Sf_CFG_SIM[a]	Design unit name for VHDL simulation
dw/dw03/src/DW_asymfifoctl_s2_sf_sim.vhd	VHDL simulation model source code
dw/sim_ver/DW_asymfifoctl_s2_sf.v	Verilog simulation model source code

a. For reliable simulation in VHDL, always use a configuration in the design specifying the design unit name from DesignWare Building Blocks (for example, DW03.DW_ASYMFIFOCTL_S2_SF_CFG_SIM). Refer to the example titled “HDL Usage Through Component Instantiation - VHDL” on page 34.

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

Table 1-7 Flush Interface Function Table (for *data_in_width* < *data_out_width*)

flush_n	part_wd	ram_full	Action	push_err
0	0	X	No action	No
0	1	0	flush	No
0	1	1	No action other than error generation	Yes
1	X	X	No action	No

Refer to Figure 1-1 on page 7 for a block diagram of the DW_asymfifoctl_s2_sf. The RAM must have:

- A synchronous write port and an asynchronous read port (if use of synchronous read port RAM and/or RAM with input retiming register at the write port is desired, consider using the DW_asymfifoctl_2c_df).
- The bit width must be the maximum of *data_in_width* or *data_out_width*.

The input data bit width of DW_asymfifoctl_s2_sf 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). Refer to Figure 1-3 for an example of *data_in_width* ≠ *data_out_width*.

The asymmetric FIFO controller provides address generation, write-enable logic, flag logic, and operational error detection logic. Parameterizable features include FIFO *depth* (up to 24 address bits or 16,777,216 locations), almost empty level, almost full level, level of error detection, type of reset (either asynchronous or synchronous), and byte (or subword) order in a word. You specify these parameters when the controller is instantiated in the design.

Simulation Methodology

DW_asymfifoctl_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.

  In order 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)

Memory Depth

If the *depth* parameter is an integer power of two (i.e. 4, 8, 16, 32, etc.), then the FIFO controller reads from RAM addresses 0 through *depth* – 1 requiring a RAM depth of exactly *depth*.

If the *depth* parameter is an odd value (i.e. 5, 7, 9, 11, etc.), then the RAM depth must be (*depth* + 1) to allow addresses that range from 0 to *depth*. If *depth* is an even value but not an integer power of two, then the RAM depth must be (*depth* + 2) to allow addresses that range from 0 to *depth* + 1.

These restrictions are derived from the facts that,

- The memory depth must always be an even number to permit all transitions of the internal Gray coded pointers to be Gray.
- For non-power of two *depth*, the memory size must be at least one greater than *depth* to allow the pointer arithmetic to unambiguously differentiate between the empty and full states.

Figure 1-1 DW_asymfifoctl_s2_sf Block Diagram

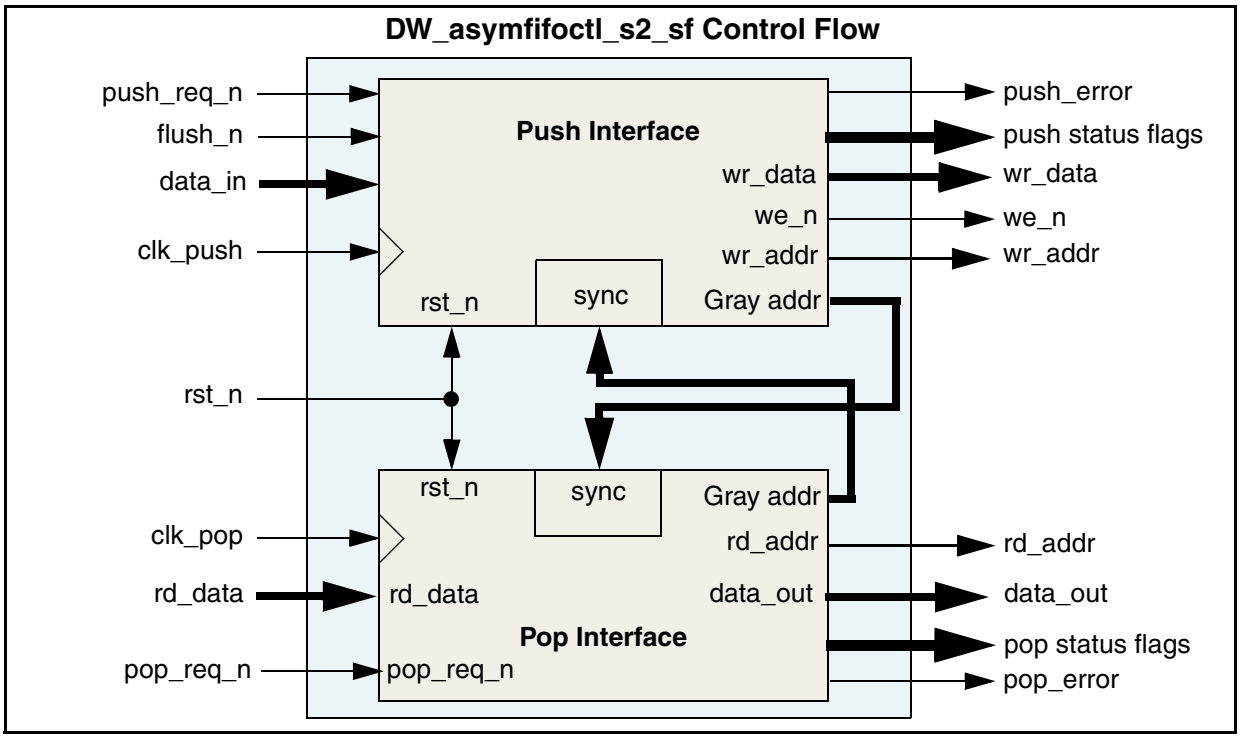
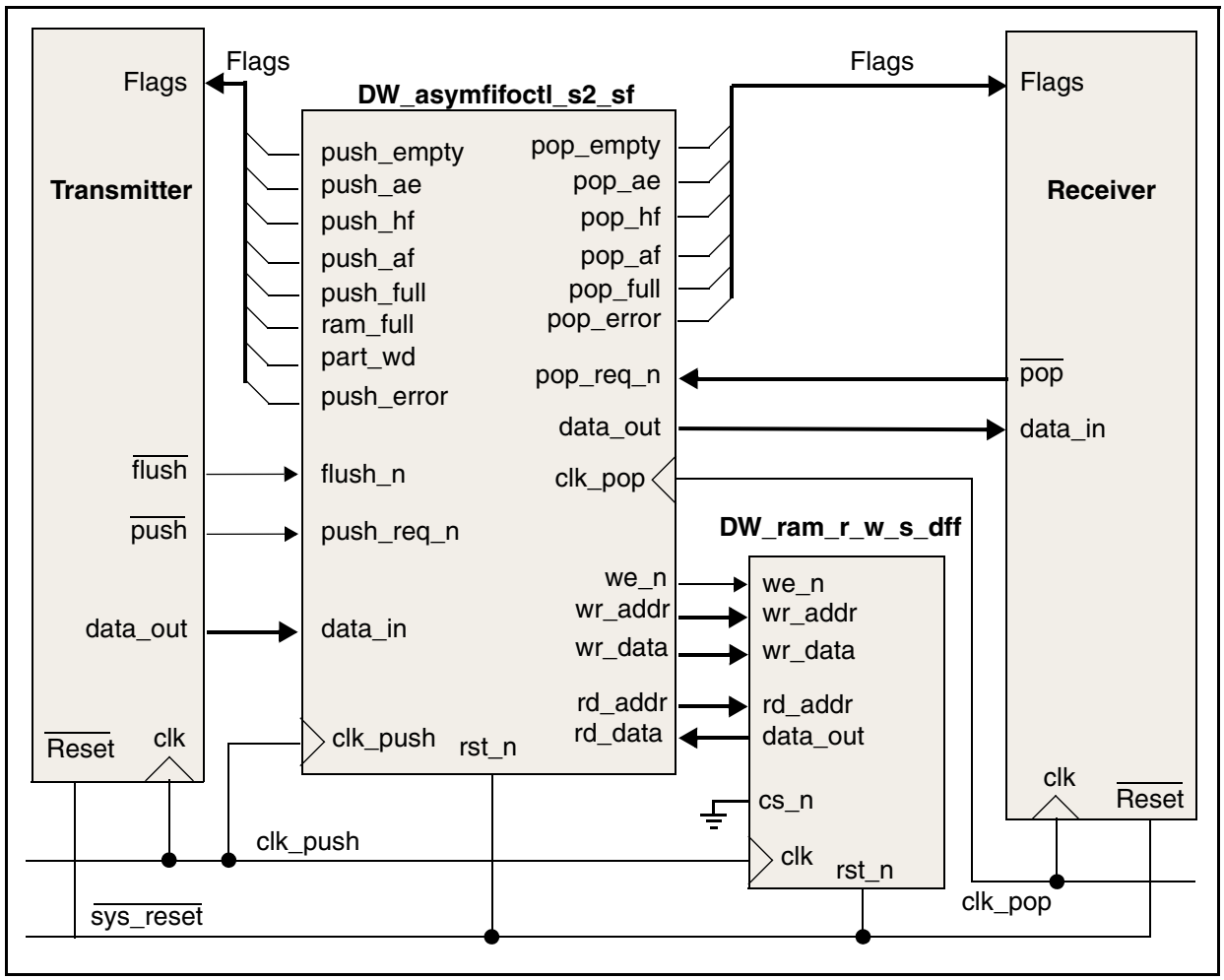


Figure 1-2 Example Usage of DW_asymfifoctl_s2_sf



Input Bus > Output Bus (data_in_width > data_out_width)

Writing to the FIFO (Push)

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):

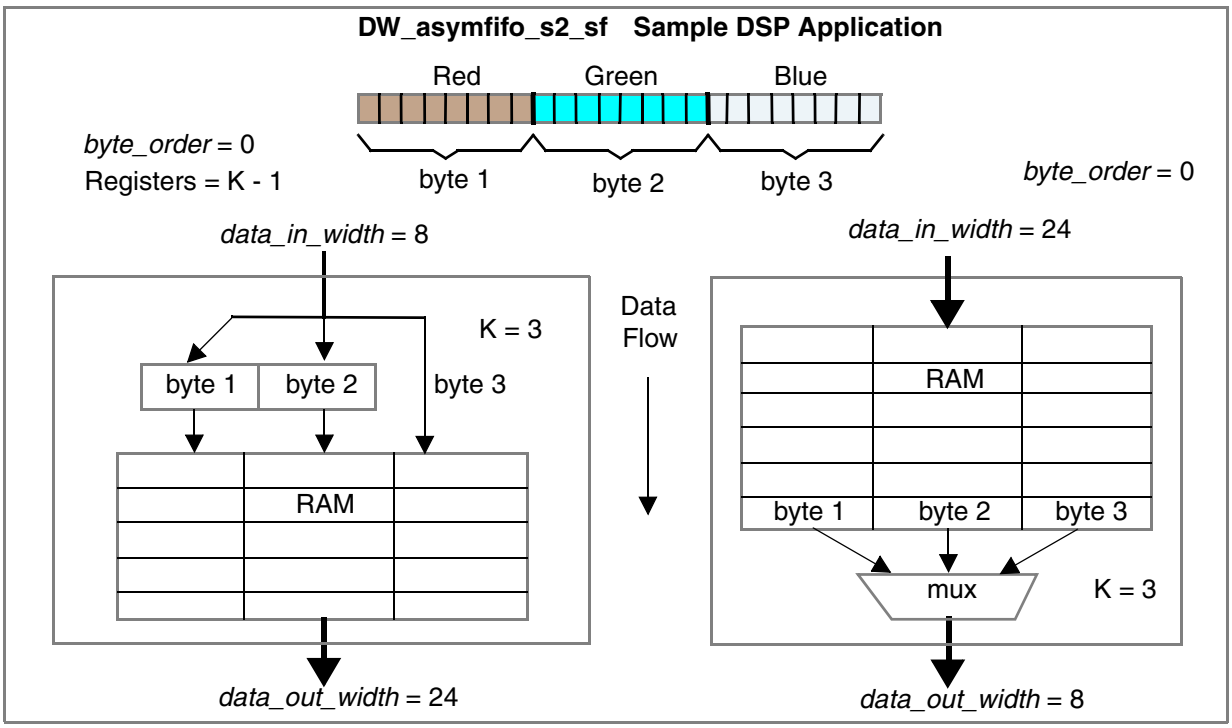
- The `flush_n` input pin is not used (at the system level, this pin should not be connected so that it is removed upon synthesis),
- The `part_wd` output pin is tied LOW, and
- The `data_in` bus is connected directly to the RAM `wr_data` output bus.

Refer to Figure 1-3 for an example of *data_in_width* > *data_out_width* 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_push`.

Figure 1-3 Example of Asymmetric FIFO Controller Operation



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

- The `we_n` line is asserted immediately, preparing for a write to the RAM on the next clock, and
- On the next rising edge of `clk_push`, `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_push` – 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 operation is attempted while the FIFO is full (as perceived by the push interface). That is, the `push_error` output goes active on the rising edge of `clk_push` if:

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

Reading from the FIFO (Pop)

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 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` should be set to active (LOW) at the positive edge of `clk_pop`. Each pop causes one byte (or subword) to be read. Popping *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 RAM must be asynchronous (since the synchronous write port requires the RAM's clock to be connected to `clk_push`, and therefore is asynchronous to `clk_pop`). The `rd_addr` output port of the DW_asymfifoctl_s2_sf provides the read address to the RAM. `rd_addr` 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) 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_pop`. Thus, memory read data must be captured on the `clk_pop` following the assertion of `pop_req_n`.

Refer to the "Timing Waveforms" on page 21 for details of the pop operation.

Read Errors

An error occurs if a pop operation is attempted while the FIFO is empty (as perceived by the pop interface). That is, the `pop_error` output goes active on the rising edge of `clk_pop` if:

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

Input Bus = Output Bus (*data_in_width* = *data_out_width*)

Writing to the FIFO (Push)

In this case, the FIFO controller is a symmetric I/O FIFO controller. Its function is the same as DW_fifoctl_s2_sf, except for the unused `part_wd`, `flush`, and `ram_full` pins.

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

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`.

Asserting `push_req_n` when `push_full` is inactive causes:

- `The we_n` to be immediately asserted in preparation for a write to the RAM on the next rising clock, and
- On the next rising edge of `clk_push`, `wr_addr` is incremented (modulus depth).

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_push – 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 operation is attempted while the FIFO is full (as perceived by the push interface). That is, the push_error output goes active if:

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

on the rising edge of clk_push.

Reading from the FIFO (Pop)

In this case, the FIFO controller is a symmetric I/O FIFO controller. Its function is the same as the DW_fifoctl_s2_sf, except for the part_wd, flush, and ram_full pins, which are unused.

The data_in bus is connected directly to the FIFO wr_data bus, and the FIFO data_out bus is connected directly to the FIFO controller's rd_data bus.

The read port of the RAM must be asynchronous (since the synchronous write port requires the RAM's clock to be connected to clk_push and therefore is asynchronous to clk_pop). The rd_addr output port of the DW_asymfifoctl_s2_sf provides the read address to the RAM. rd_addr always points to, thus prefetches, the next word of RAM read data to be popped.

A pop operation occurs when:

- The pop_req_n is asserted (LOW), and
- The pop_empty flag is not active (LOW) (the FIFO is not empty)

at the rising edge of clk_pop.

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 must be captured on the rising edge of clk_pop following the assertion of pop_req_n.

Read Errors

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

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

on the rising edge of clk_pop.

Input Bus < Output Bus (*data_in_width < data_out_width*)

Writing to the FIFO (Push)

For cases where *data_in_width* < *data_out_width* (assuming that *data_out_width* = *K* × *data_in_width*, where *K* is an integer larger than 1), every byte (or subword) written to the FIFO is first assembled into a full word with *data_out_width* bits. Refer to Figure 1-3 on page 9 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 a partial word is executed when `push_req_n` is asserted (LOW) and the `full` flag is inactive (LOW) at the rising edge of `clk_push`. Thus, a push can occur even if the FIFO is full, as long as a pop is executed in the same cycle.

The order of bytes within a word is determined by the *byte_order* parameter. For every byte (or subword) to be written, `push_req_n` must be active (LOW) at the positive edge of `clk_push`. 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 1-3 on page 9.) to be written to the next available location in the FIFO. This write occurs on the `clk_push` following the assertion of `push_req_n`.

The data at the `data_in` port must be stable for a setup time before the rising edge of `clk_push`, 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_push` – the first clock after `push_req_n` is asserted *K* times.

Partial Word

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. The 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 a Partial Word

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, and `part_wd` goes LOW.

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

A flush is allowed for data byte word alignment or when:

- *N* bytes have been read since the last complete word (where 0 < *N* < *K*), and
- The sender device has no more bytes (or subwords) to assemble the last full word,

while

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

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 the `ram_full` flag is inactive (LOW) at the rising edge of `clk_push`.

Asserting `flush_n` when flushing the FIFO is allowed 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_push` 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 Push and Flush

DW_asymfifoctl_s2_sf supports simultaneous push and flush under the following conditions:

- The `ram_full` is inactive (LOW),
- The `part_wd` is active (HIGH),
- The `push_req_n` is active (LOW), and
- The `flush_n` is active (LOW).

On the leading edge of `clk_push`, the partial word in the input buffer is flushed into the RAM, and the byte (or subword) at the `data_in` port is pushed into the first byte location of the input buffer.

If there is no partial word in the input buffer (`part_wd` inactive [LOW]), a simultaneous push and flush generates a normal push. Under this condition, flush is a null action. Table 1-8 details a simultaneous push and flush operation for *data_in_width* < *data_out_width*, when `flush_n` is LOW, `push_req_n` is LOW.

Table 1-8 Simultaneous Flush and Push Function Table (for data_in_width < data_out_width)

part_wd	ram_full	push_full	Action	push_err
0	X	0	No flush, push only	No
1	0	0	Flush and Push	No
1	1	0	No flush, push only. Potential misaligned word.	Yes
1	1	1	No action other than error generation. Data loss and potential misaligned word.	Yes

Write Errors

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

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

on the rising edge of `clk_push`.

For *data_in_width* < *data_out_width*, `push_error` also goes active if:

- The `ram_full` is active (HIGH),
- The `part_wd` is active (HIGH), and
- The `flush_n` input is asserted (LOW)

on the leading edge of `clk_push`.

Reading from the FIFO (Pop)

For cases where *data_in_width* < *data_out_width* (assuming that *data_out_width* = *K* × *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 FIFO `data_out` bus.

The read port of the RAM must be asynchronous (since the synchronous write port requires the RAM's clock to be connected to `clk_push` and therefore is asynchronous to `clk_pop`). The `rd_addr` output port of the DW_asymfifoctl_s2_sf provides the read address to the RAM. `rd_addr` 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 (as perceived by the pop interface) on the next rising edge of `clk_pop`. Thus, the RAM read data must be captured on the `clk_pop` following the assertion of `pop_req_n`.

Refer to the timing diagrams for details of the pop operation for RAMs with synchronous and asynchronous read ports.

Read Errors

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

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

on the rising edge of `clk_pop`.

Reset

rst_mode
This parameter selects whether the DW_asymfifoctl_s2_sf 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, and
- The flags and error outputs to be initialized.

- For cases where *data_in_width* < *data_out_width*, the input buffer is reset.
- For cases where *data_in_width* > *data_out_width*, the output buffer is reset.

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

- The write address pointer,
- Push flags, and
- The `push_error` output

are initialized. Also, for cases where *data_in_width* < *data_out_width*, the input buffer is reset.

After the assertion of `rst_n` and at the rising edge of `clk_pop`, the:

- Read address pointer,
- The pop flags, and
- The `pop_error` output

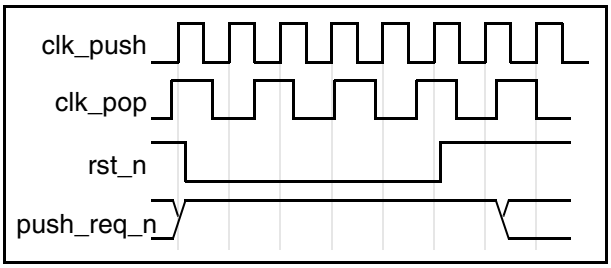
are initialized. Also, for cases where *data_in_width* < *data_out_width*, the output buffer is reset.

Metastability Issues

In order to avoid metastability upon reset, the assertion of `rst_n` (LOW) should be maintained 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). Refer to Figure 1-4.

Attention Since the one input that is critical to proper reset sequencing (i.e. `push_req_n`) is in the domain of `clk_push`, it is recommended that the reset input, `rst_n`, should be synchronous to `clk_push`

Figure 1-4 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).

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 in order for each interface to 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 (e.g. `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 will be captured by the sampling clock edge or whether the previous value will be 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 one (1) indicates single-stage synchronization; a value of two (2) indicates double-stage synchronization; a value of three (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 is 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 write pointer value has been synchronized to `clk_pop`. This could be as long as 2 or 3 cycles (depending on the value of the *pop_sync* parameter). Refer to the timing diagrams for more information.

You 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 read pointer value has been synchronized to `clk_push`. This could be as long as 2 or 3 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, `part_wd`, `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 read pointer value has been synchronized to `clk_push`. This could be as long as 2 or 3 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.

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 write pointer value has been synchronized to `clk_pop`. This could be as long as 2 or 3 cycles (depending on the value of the *pop_sync* parameter). Refer to the timing diagrams for more information.

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

ram_full

The `ram_full` output is used for the *data_in_width* < *data_out_width* case. This flag is synchronous to `clk_push`. The `ram_full` output indicates that the RAM (excluding the FIFO input buffer in the controller) is full, and there is no space available for flushing a partial word into the RAM. However, if `part_wd` is inactive (LOW), there are still some spaces in the input buffer for incoming subwords. 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 `push_full` output.

part_wd

This flag is only used for the *data_in_width* < *data_out_width* case. This flag is synchronous to `clk_push`. 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* and *data_in_width* = *data_out_width* cases, `part_wd` is tied LOW since the input data is always a full word.

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 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.

For *data_in_width* < *data_out_width* case, the `push_error` output signal may also indicate a flush request was seen while the `ram_full` output was active (HIGH). This indicates a potential overrun error and/or word misalignment error. Refer to Table 1-7 on page 24 for details.

pop_error
The `pop_error` output signal indicates 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, as 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 accepted by the receiving logic. This is easily avoided if the logic controlling the `pop_req_n` input 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 indicating 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 which 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) then 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 it is desirable 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) then 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 will 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) then 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 will 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 will not overrun the FIFO.

push_full
The `push_full` output, active HIGH, is synchronous to the `clk_push` input. `push_full` 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. `pop_empty` 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. `pop_ae` 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) then there are at least (*pop_ae_lvl* +1) 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 will 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) then 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 will not underrun the FIFO.

pop_af
The `pop_af` output, active HIGH, is synchronous to the `clk_pop` input. `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 it is desirable 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) then 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 will not underrun the FIFO.

pop_full
The `pop_full` output, active HIGH, is synchronous to the `clk_pop` input. `pop_full` 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 figures in this section show timing diagrams for various conditions of DW_asymfifoctl_s2_sf. Also refer to the DW_fifoctl_s2_sf datasheet or the following timing diagrams where Input = Output:

- push and pop timing waveforms
- single word push and pop timing waveforms
- FIFO *depth* $\neq 2^n$ push and pop timing waveforms
- FIFO *depth* $\neq 2^n$ single word timing waveforms

Figure 1-5 Push Timing Waveforms for Input > Output

FIFO *data_in_width*=16, *data_out_width*=8, *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

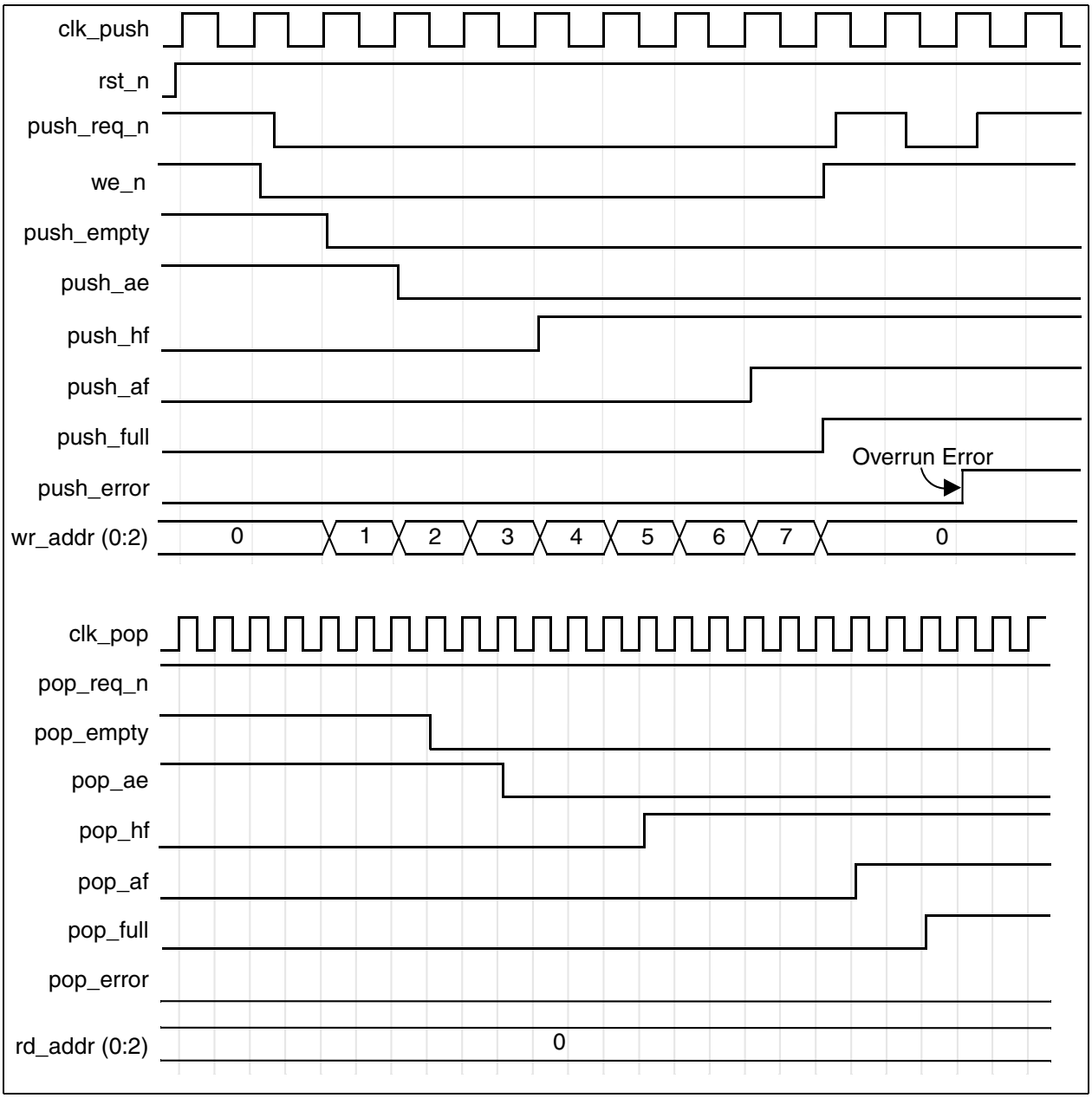


Figure 1-6 Push Timing Waveforms for Input < Output

FIFO *data_in_width*=8, *data_out_width*=16, *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

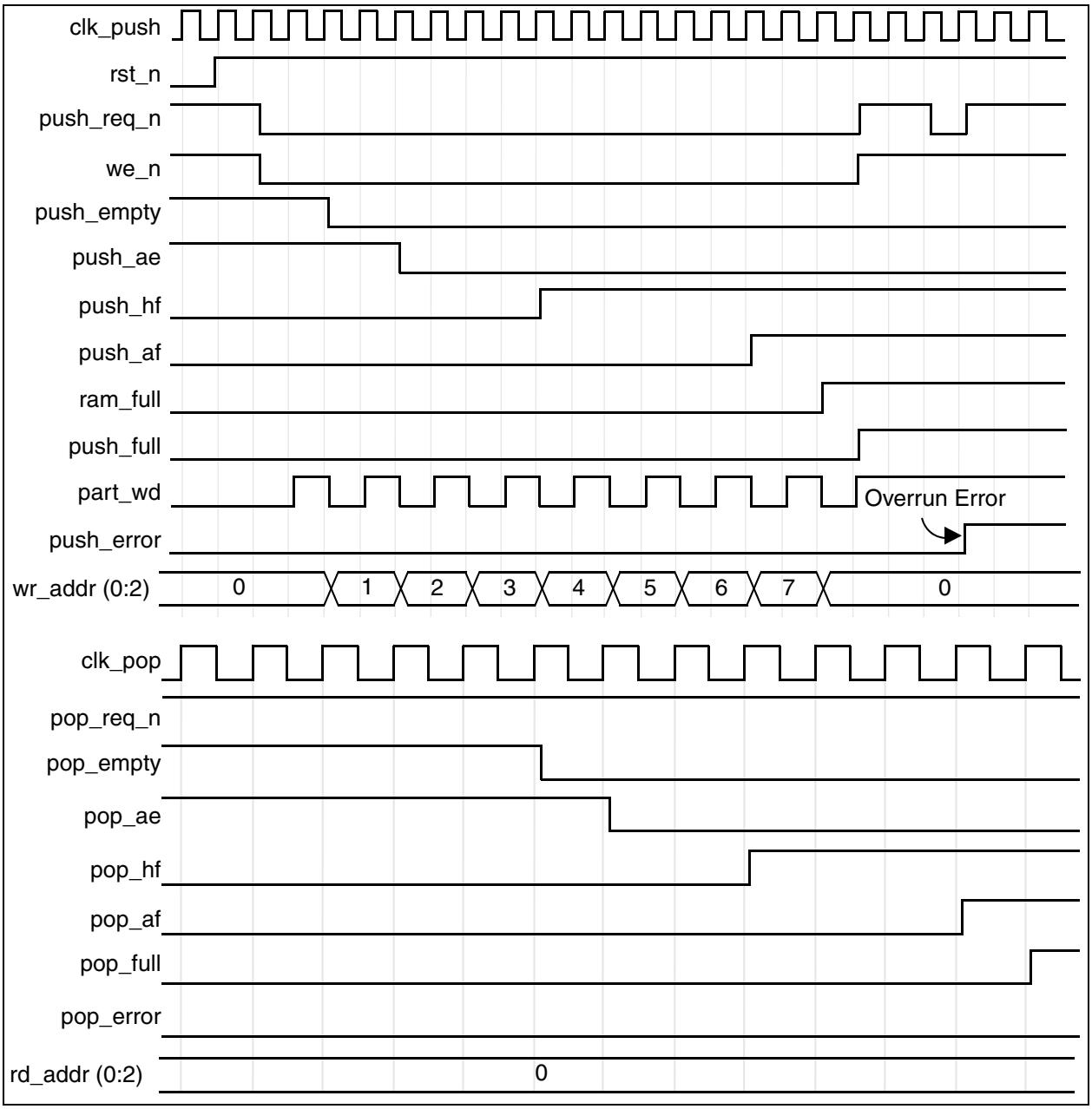


Figure 1-7 Pop Timing Waveforms for Input > Output

FIFO *data_in_width*=16, *data_out_width*=8, *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

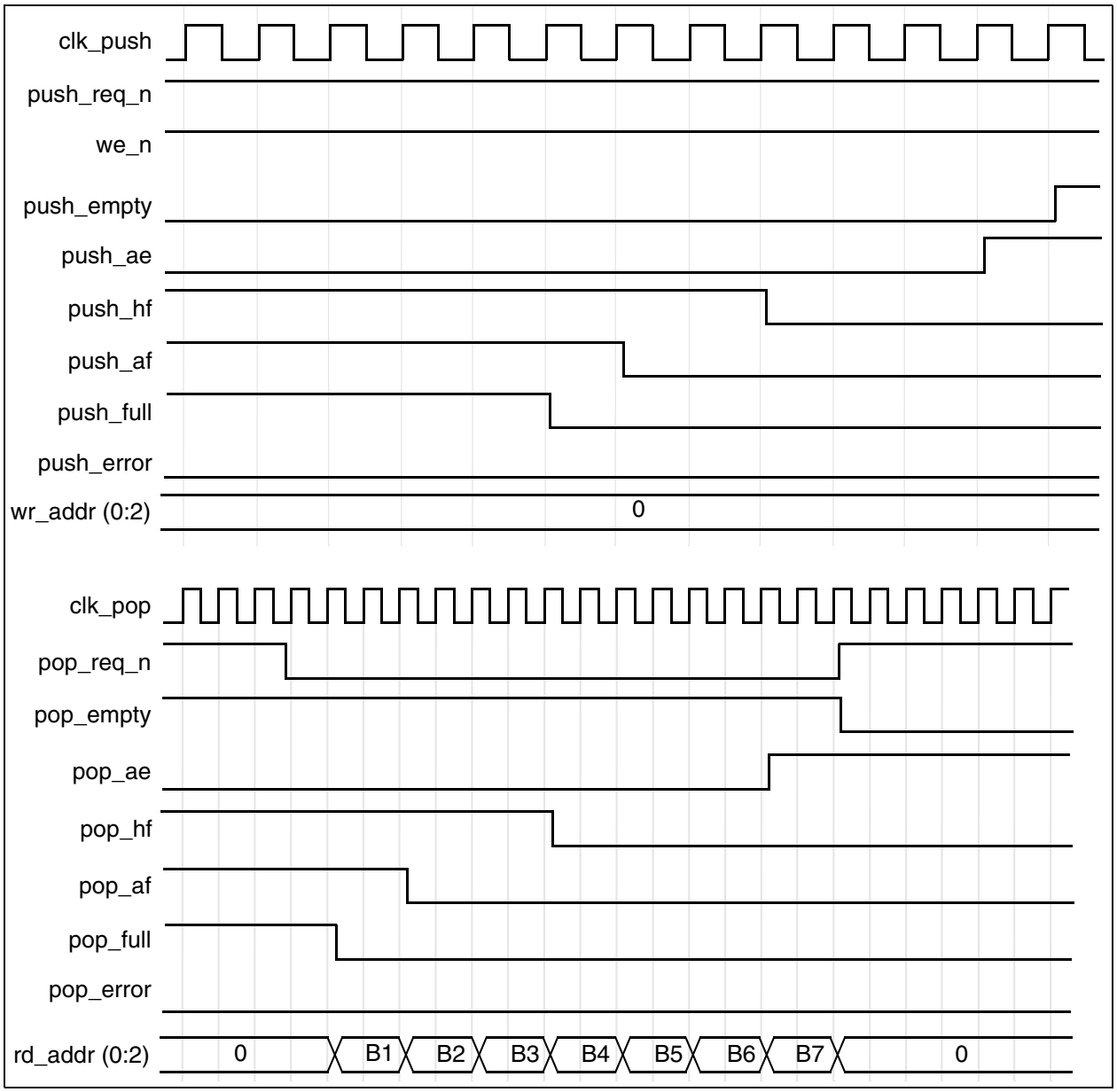


Figure 1-8 Pop Timing Waveforms for Input < Output

FIFO data_in_width=8, *data_out_width*=16, *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

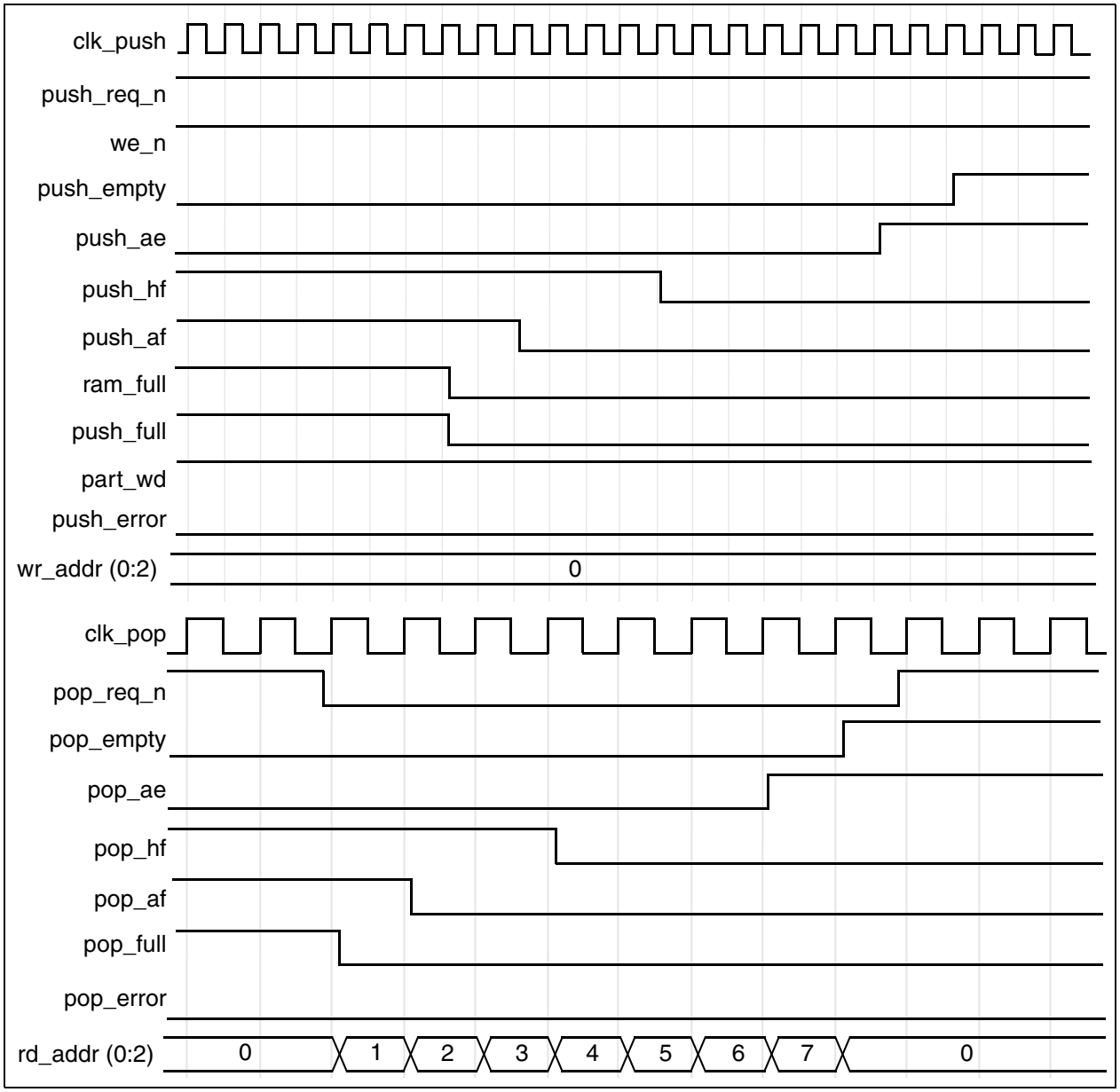


Figure 1-9 Single Word Push and Pop Timing Waveforms for Input > Output

FIFO *data_in_width*=16, *data_out_width*=8, *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

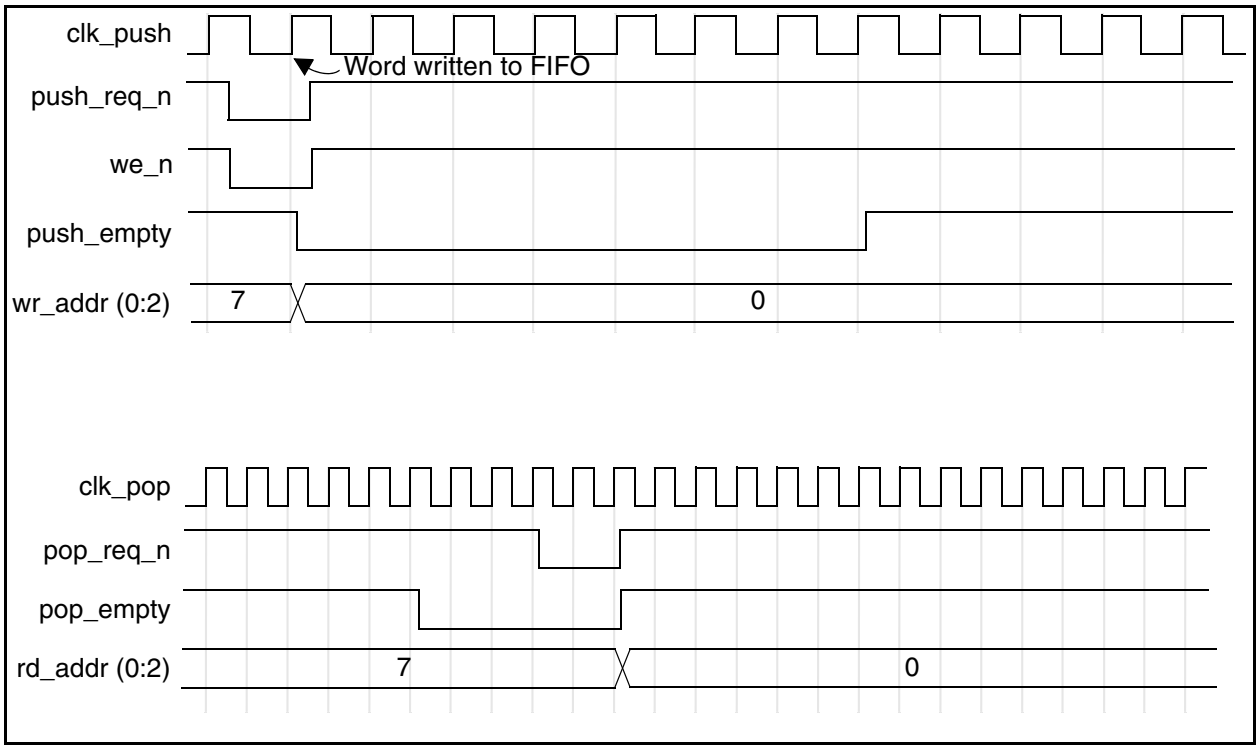


Figure 1-10 Single Word Push and Pop Timing Waveforms for Input < Output

FIFO *data_in_width*=8, *data_out_width*=16, *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

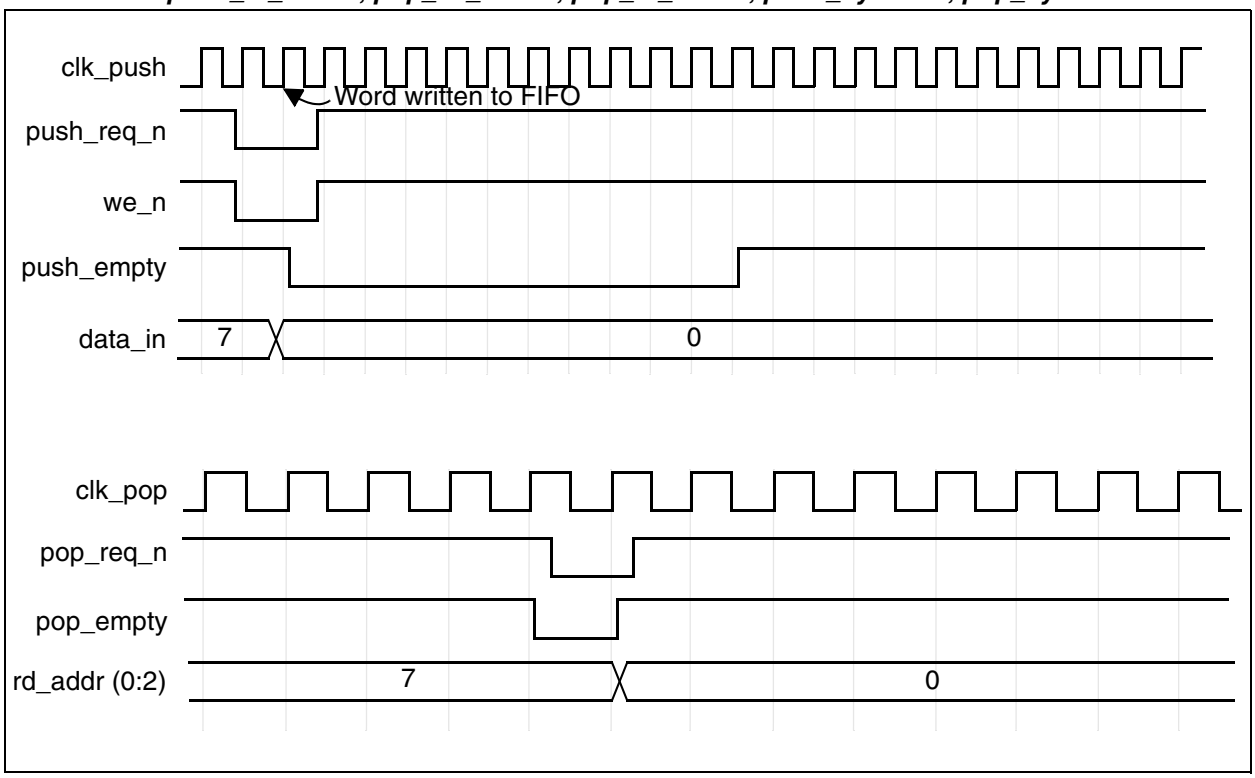


Figure 1-11 FIFO Depth ≠ 2^n Push Timing Waveforms for Input > Output

FIFO *data_in_width*=16, *data_out_width*=8, *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

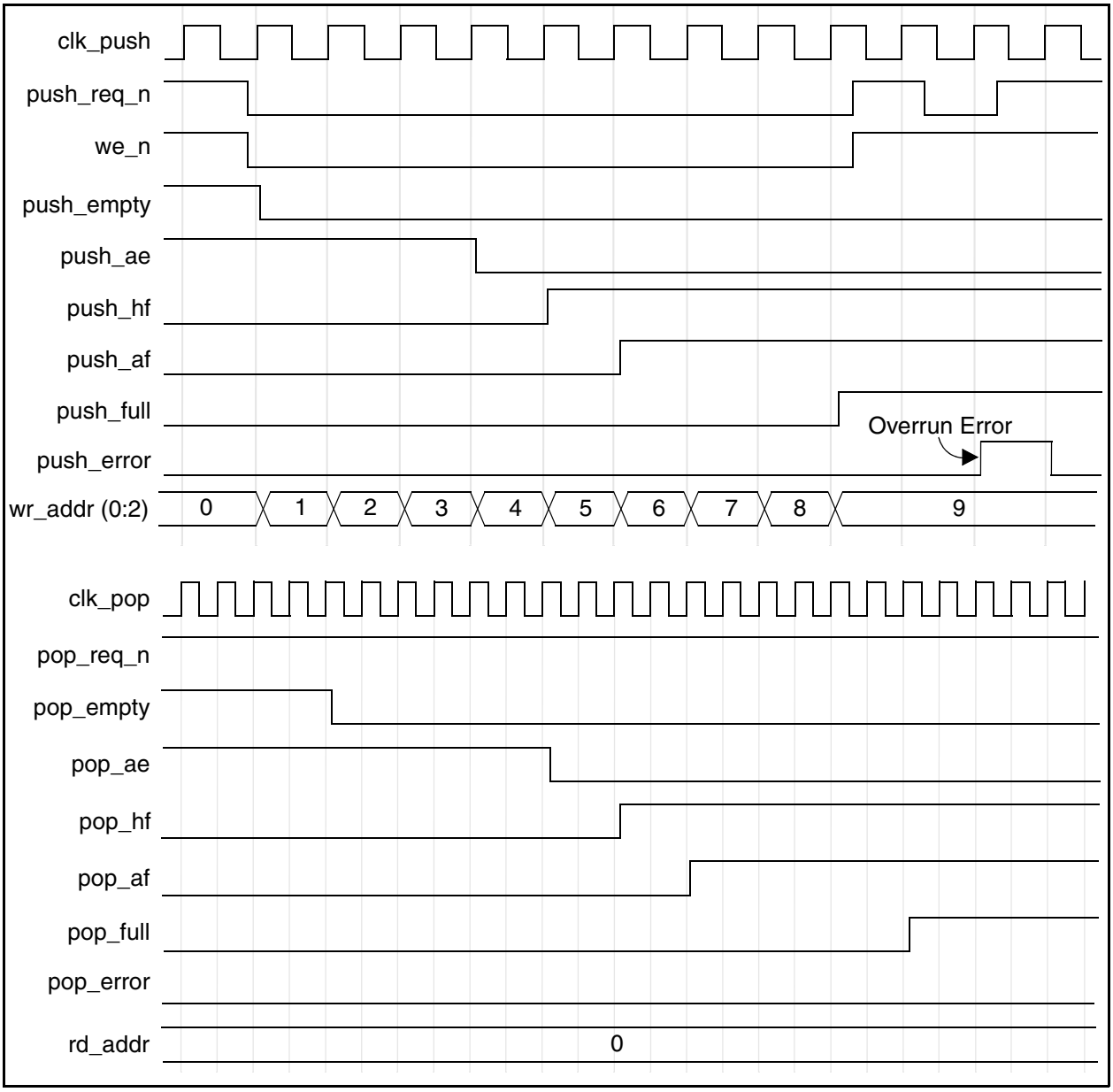


Figure 1-12 FIFO Depth ≠ 2^n Push Timing Waveforms for Input < Output

FIFO *data_in_width*=8, *data_out_width*=16, *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

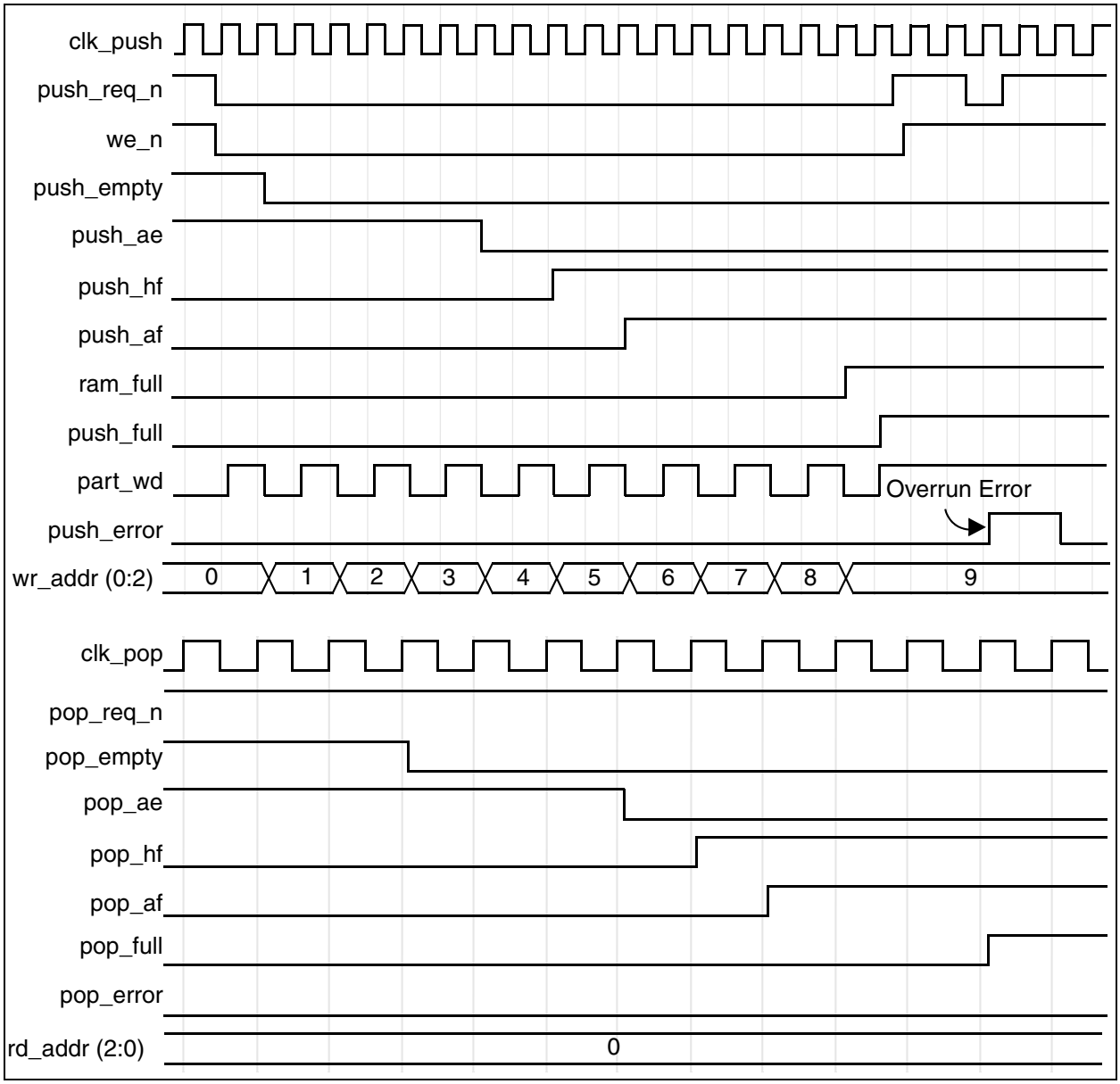


Figure 1-13 FIFO Depth ≠ 2^n Pop Timing Waveforms for Input > Output

FIFO *data_in_width*=16, *data_out_width*=8, *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

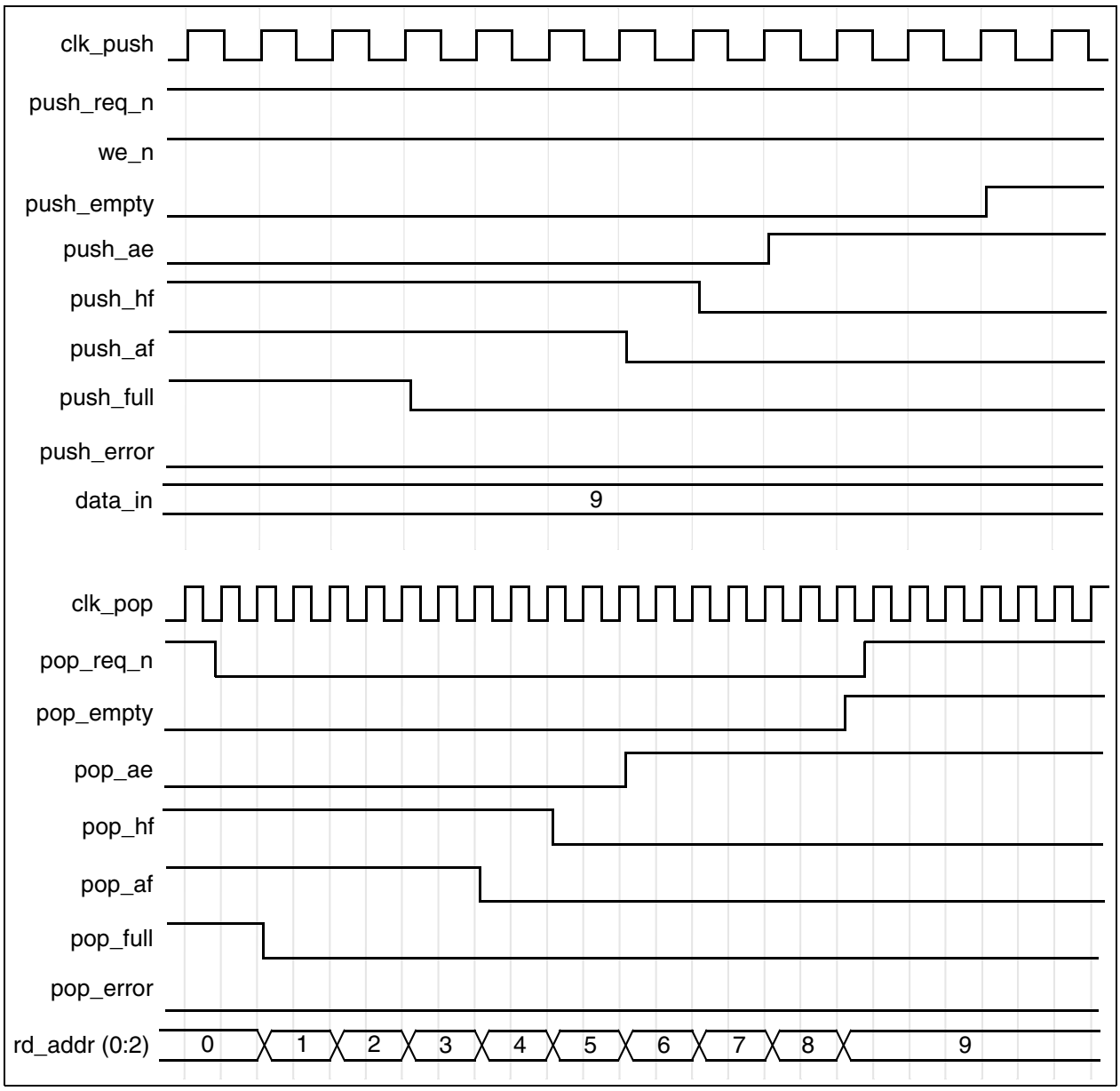


Figure 1-14 FIFO Depth ≠ 2^n Pop Timing Waveforms for Input < Output

FIFO *data_in_width*=8, *data_out_width*=16, *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

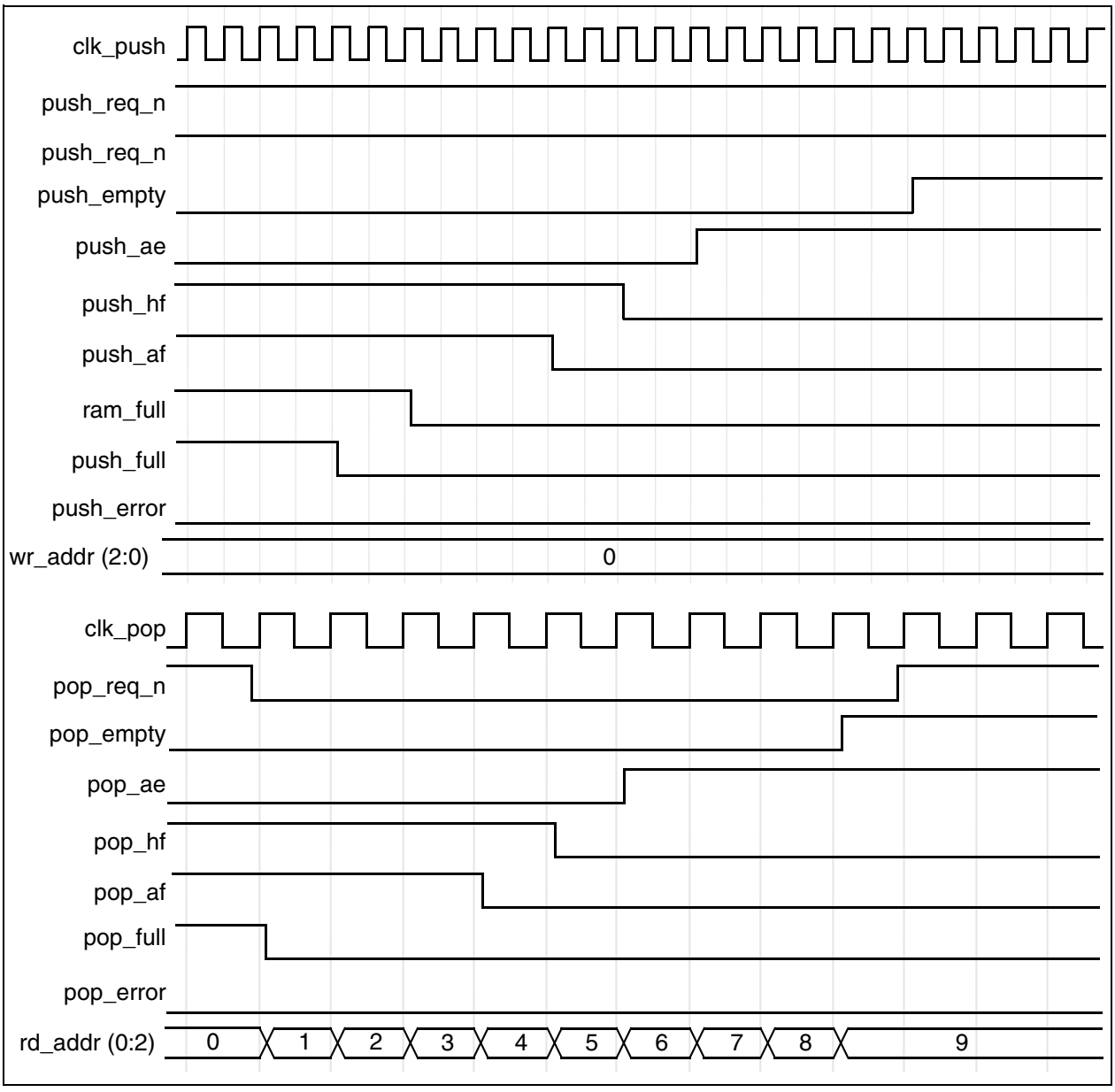


Figure 1-15 FIFO Depth ≠ 2^n Single Word Timing Waveform for Input > Output

FIFO *data_in_width*=16, *data_out_width*=8, *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

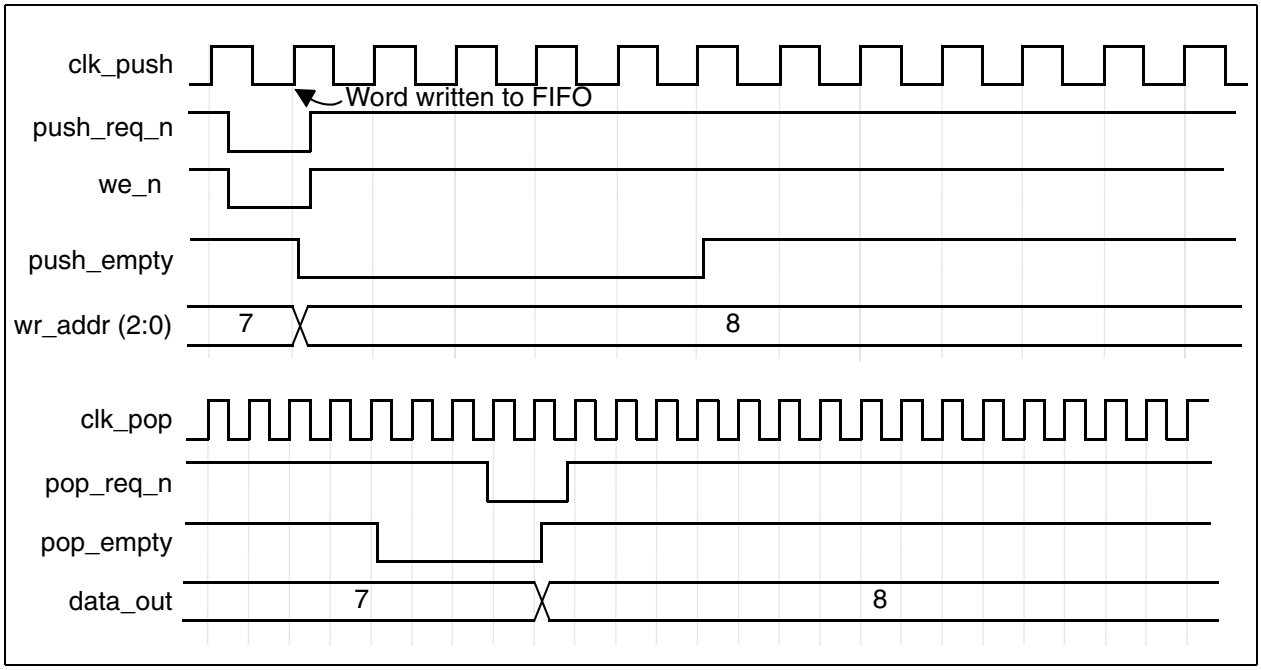


Figure 1-16 FIFO Depth ≠ 2^n Single Word Timing Waveform for Input < Output

FIFO *data_in_width*=8, *data_out_width*=16, *depth* = 9 (Even 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

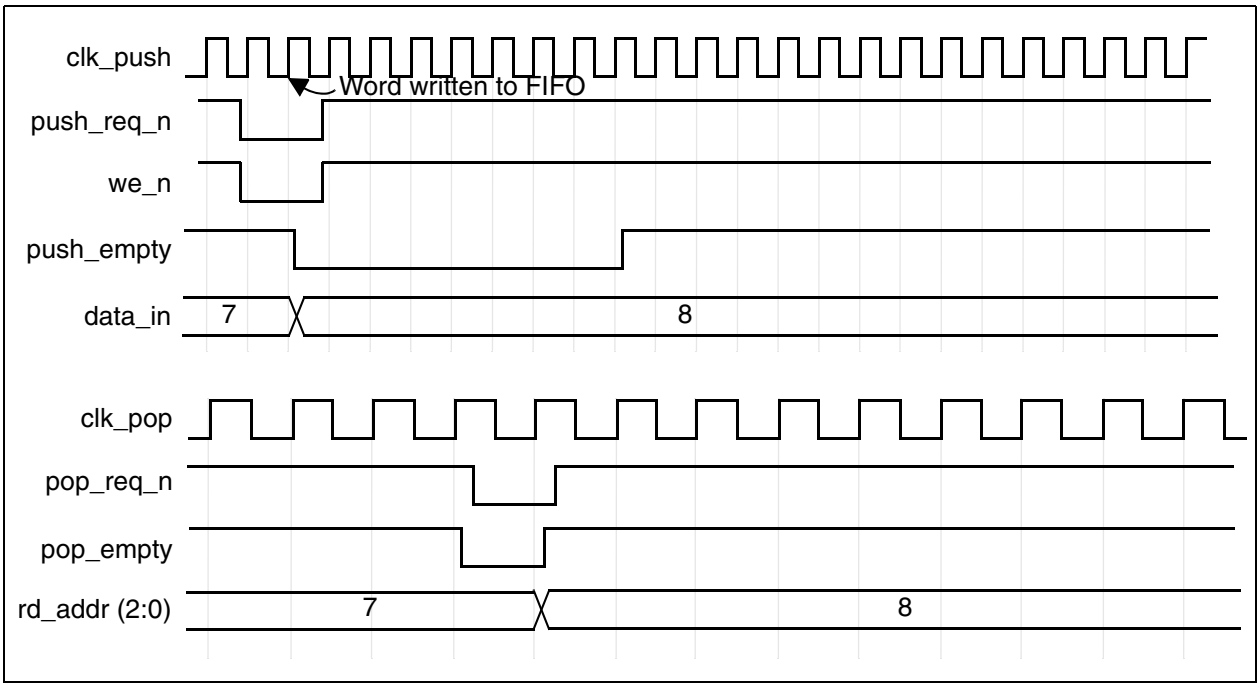
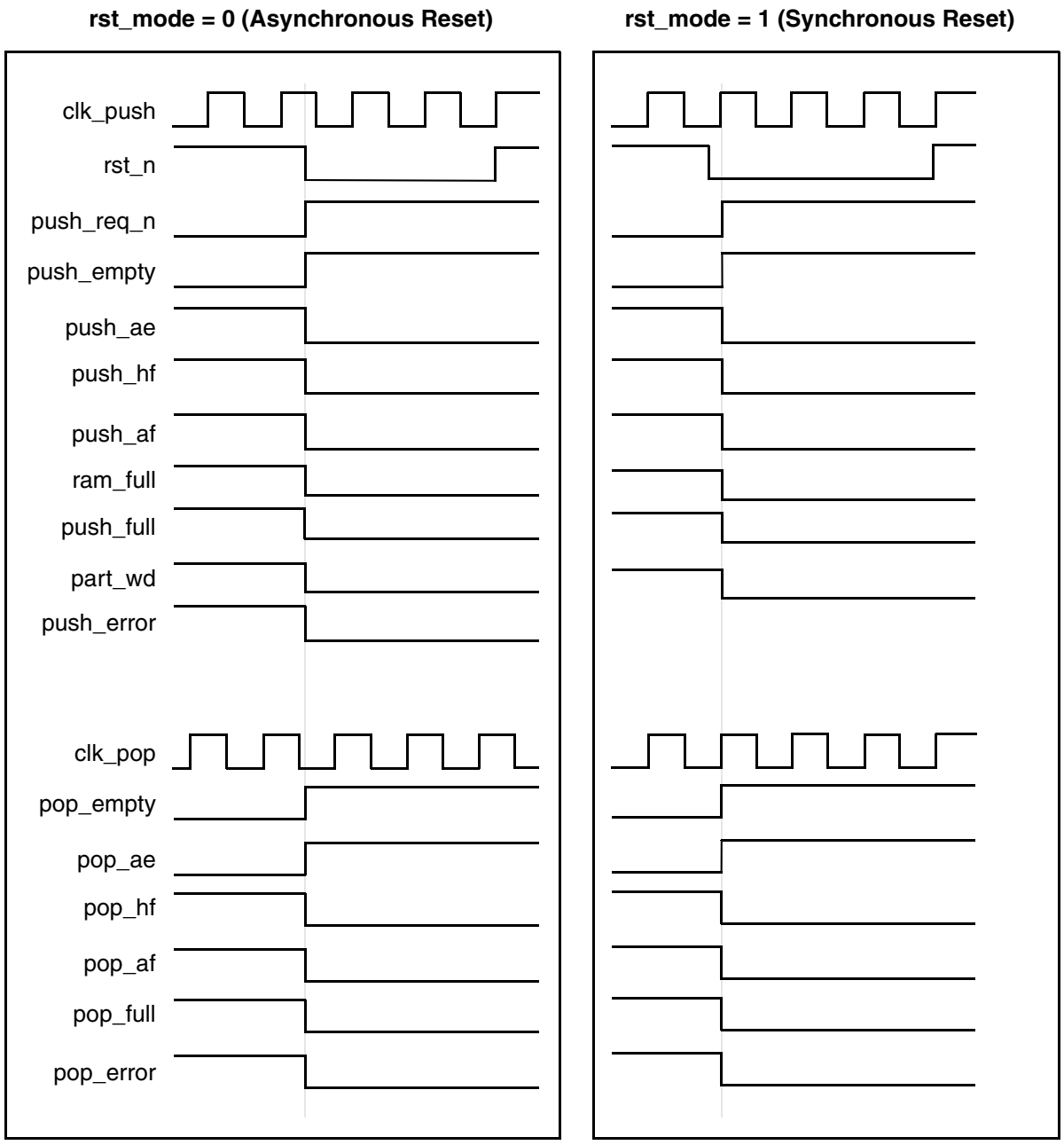


Figure 1-17 Reset Timing Waveforms



Related Topics

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

HDL Usage Through Component Instantiation - VHDL

```
library IEEE,DWARE,DW03;
use IEEE.std_logic_1164.all;
use DWARE.DW_Foundation_arith.all;
use DWARE.DW_Foundation_comp_arith.all;

entity DW_asymfifoctl_s2_sf_inst is
  generic (inst_data_in_width  : INTEGER := 8;
           inst_data_out_width : INTEGER := 24;
           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 := 1;
           inst_pop_sync       : INTEGER := 1;
           inst_rst_mode       : INTEGER := 1;
           inst_byte_order     : 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_flush_n    : in std_logic;   inst_pop_req_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);
        we_n_inst       : out std_logic;   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;
        ram_full_inst   : out std_logic;   part_wd_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;
        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_s2_sf_inst;

architecture inst of DW_asymfifoctl_s2_sf_inst is
begin

  -- Instance of DW_asymfifoctl_s2_sf
  U1 : DW_asymfifoctl_s2_sf
    generic map (data_in_width => inst_data_in_width,
                 data_out_width => inst_data_out_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,
                  byte_order => inst_byte_order )
    port map (clk_push => inst_clk_push,
              clk_pop => inst_clk_pop,
              rst_n => inst_rst_n,
              push_req_n => inst_push_req_n,
              flush_n => inst_flush_n,
              pop_req_n => inst_pop_req_n,
              data_in => inst_data_in,
              rd_data => inst_rd_data,
              we_n => we_n_inst,
              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,
              ram_full => ram_full_inst,
              part_wd => part_wd_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,
              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
library DW03;
configuration DW_asymfifoctl_s2_sf_inst_cfg_inst of
 DW_asymfifoctl_s2_sf_inst is
  for inst
  end for; -- inst
end DW_asymfifoctl_s2_sf_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_asymfifoctl_s2_sf_inst(inst_clk_push, inst_clk_pop, inst_rst_n,
   inst_push_req_n, inst_flush_n, inst_pop_req_n, inst_data_in, inst_rd_data,
   we_n_inst, push_empty_inst, push_ae_inst, push_hf_inst, push_af_inst,
   push_full_inst, ram_full_inst, part_wd_inst, push_error_inst,
   pop_empty_inst, pop_ae_inst, pop_hf_inst, pop_af_inst, pop_full_inst,
   pop_error_inst, wr_data_inst, wr_addr_inst, rd_addr_inst, data_out_inst
   );
  parameter data_in_width = 8;
  parameter data_out_width = 24;
  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 = 1;
  parameter byte_order = 0;
  `define bit_width_depth 3 // ceil(log2(depth))

  input inst_clk_push;
  input inst_clk_pop;
  input inst_rst_n;
  input inst_push_req_n;
  input inst_flush_n;
  input inst_pop_req_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;
  output we_n_inst;
  output push_empty_inst;
  output push_ae_inst;
  output push_hf_inst;
  output push_af_inst;
  output push_full_inst;
  output ram_full_inst;
  output part_wd_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 [((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_s2_sf
    DW_asymfifoctl_s2_sf #(data_in_width, data_out_width, depth, push_ae_lvl,
                           push_af_lvl, pop_ae_lvl, pop_af_lvl, err_mode,
                           push_sync, pop_sync, rst_mode, byte_order)
    U1 (.clk_push(inst_clk_push),   .clk_pop(inst_clk_pop),
        .rst_n(inst_rst_n),   .push_req_n(inst_push_req_n),
        .flush_n(inst_flush_n),   .pop_req_n(inst_pop_req_n),
        .data_in(inst_data_in),   .rd_data(inst_rd_data),
        .we_n(we_n_inst),   .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),
        .ram_full(ram_full_inst),   .part_wd(part_wd_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),   .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
December 2017	N-2017.09-SP2	■ Added "Simulation Methodology" on page 6 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 4 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