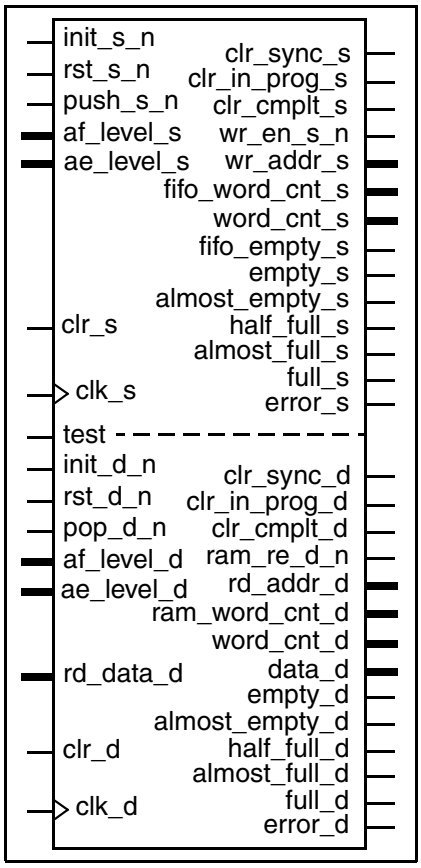
DW_asymfifoctl_2c_df

Asymmetric Dual Clock FIFO Controller with Dynamic Flags

Version, STAR and Download Information: IP Directory

Features and Benefits



- Parameterized asymmetric source and destination data widths
- Parameterized flush value
- Word integrity flag
- Parameterized byte (sub-word) ordering within a word
- Parameterized error status mode
- Registered push error (overflow)
- Pop interface caching (pre-fetching)
- Alternative pop cache implementations provided for optimum power savings
- Configurable pipelining of push and pop control/data to accommodate synchronous RAMs
- Single clock cycle push and pop operations
- Fully registered synchronous status flag outputs
- Status flags provided from each clock domain
- Parameterized input and output widths
- Parameterized RAM depth
- Parameterized full-related and empty-related flag thresholds per clock domain
- Push error (overflow) and pop error (underflow) flags per clock domain
- Provides minPower benefits when enabled with the minPower (DesignWare-LP) license (Get the minPower version of this datasheet)

Description

DW_asymfifoctl_2c_df is a dual independent clock asymmetric data width FIFO controller intended to interface with dual-port synchronous RAM. Word caching (or pre-fetching) is performed in the pop interface to minimize latencies and allow for bursting of contiguous words. The caching depth is configurable.

This component, DW_asymfifoctl_2c_df, consists of integrating current DesignWare Library components DW_asymdata_inbuf, DW_fifoctl_2c_df, and DW_asymdata_outbuf.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
clk_s	1 bit	Input	Source domain clock
rst_s_n	1 bit	Input	Source domain asynchronous reset (active low)
init_s_n	1 bit	Input	Source domain synchronous reset (active low)
clr_s	1 bit	Input	Source domain clear RAM contents
ae_level_s	ceil(log2[*ram_depth* + 1])	Input	Source domain almost empty level for the `almost_empty_s` output (the number of words in the FIFO at or below which the `almost_empty_s` flag is active) (see *eff_depth* note below table).
af_level_s	ceil(log2[*ram_depth* + 1])	Input	Source domain almost full level for the `almost_full_s` output (the number of empty memory locations in the FIFO at which the `almost_full_s` flag is active).
push_s_n	1 bit	Input	Source domain push request (active low)
flush_s_n	1 bit	Input	Source domain flush the partial word (active low) Note: only useful when *data_s_width* < *data_d_width*.
data_s	*data_s_width*	Input	Source domain input data. Note: directly routed to RAM when *data_s_width* ≥ *data_d_width*
clr_sync_s	1 bit	Output	Source domain coordinated clear synchronized (reset pulse that goes source sequential logic)
clr_in_prog_s	1 bit	Output	Source domain clear in progress
clr_cmplt_s	1 bit	Output	Source domain clear complete (single `clk_s` cycle pulse)
wr_en_s_n	1 bit	Output	Source domain write enable to RAM (active low and unregistered)
wr_addr_s	ceil(log2[*ram_depth* + 1])	Output	Source domain write address to RAM (registered)
wr_data_s	*data_s_width* or *data_d_width*	Output	Source domain write data to RAM Note: When *data_s_width* < *data_d_width*, width is '*data_d_width*'. When '*data_s_width*' ≥ *data_d_width*, width is *data_s_width*.
inbuf_part_wd_s	1	Output	Source domain partial word pushed flag Note: Meaningful only when *data_s_width* < *data_d_width*.
inbuf_full_s	1	Output	Source domain input buffer full flag
fifo_word_cnt_s	ceil(log2[*eff_depth*+1])	Output	Source domain total word count in the RAM and cache (see *eff_depth* note below table)

Table 1-1 Pin Description (Continued)

Pin Name	Width	Direction	Function
word_cnt_s	ceil(log2[*ram_depth*+1])	Output	Source domain RAM word count (see note on *ram_depth* below)
fifo_empty_s	1 bit	Output	Source domain FIFO empty flag
empty_s	1 bit	Output	Source domain RAM empty flag
almost_empty_s	1 bit	Output	Source domain almost empty flag (determined by `ae_level_s` port)
half_full_s	1 bit	Output	Source domain half full flag
almost_full_s	1 bit	Output	Source domain almost full flag (determined by `af_level_s` port)
rma_full_s	1 bit	Output	Source domain RAM full flag
push_error_s	1 bit	Output	Source domain push error flag (overrun)
clk_d	1 bit	Input	Destination domain clock
rst_d_n	1 bit	Input	Destination domain asynchronous reset (active low)
init_d_n	1 bit	Input	Destination domain synchronous reset (active low)
clr_d	1 bit	Input	Destination domain clear RAM contents
ae_level_d	ceil(log2[*ram_depth*+1])	Input	Destination domain almost empty level for the `almost_empty_d` output (the number of words in the FIFO at or below which the `almost_empty_d` flag is active) (see *eff_depth* note below table).
af_level_d	ceil(log2[*ram_depth*+1])	Input	Destination domain almost full level for the `almost_full_d` output (the number of empty memory locations in the FIFO at which the `almost_full_d` flag is active).
pop_d_n	1 bit	Input	Destination domain pop request (active low)
rd_data_d	*data_s_width* or *data_d_width*	Input	Destination domain read data. Note: When *data_s_width* < *data_d_width*, width is *data_d_width*. When *data_s_width* ≥ *data_d_width*, width is *data_s_width*.
clr_sync_d	1 bit	Output	Destination domain coordinated clear synchronized (reset pulse that goes to source sequential logic)
clr_in_prog_d	1 bit	Output	Destination domain clear in progress
clr_cmplt_d	1 bit	Output	Destination domain clear complete (single `clk_d` cycle pulse)
ram_re_d_n	1 bit	Output	Destination domain read enable to RAM (active low)

Table 1-1 Pin Description (Continued)

Pin Name	Width	Direction	Function
rd_addr_d	ceil(log2(*ram_depth*))	Output	Destination domain read address to RAM (registered)
data_d	*data_d_width*	Output	Destination domain data to pop
outbuf_part_wd_d	1	Output	Destination domain output buffer partial word popped flag. Note: only meaningful when *data_s_width* > *data_d_width*
word_cnt_d	ceil(log2[*eff_depth*+1])	Output	Destination domain RAM word count (see *eff_depth* note below table).
ram_word_cnt_d	ceil(log2[*ram_depth*+1])	Output	Destination domain RAM word count (see note on ram_depth parameter below)
empty_d	1 bit	Output	Destination domain RAM empty flag
almost_empty_d	1 bit	Output	Destination domain almost empty flag (determined by `ae_level_s` parameter)
half_full_d	1 bit	Output	Destination domain half full flag
almost_full_d	1 bit	Output	Destination domain almost full flag (determined by `af_level_s` port)
full_d	1 bit	Output	Destination domain almost full flag
pop_error_d	1 bit	Output	Destination domain push error flag (under-run)
test	1 bit	Input	Scan test mode select

Note: *eff_depth* (effective depth) is not a user parameter but is used here as a placeholder which is derived from the parameters *ram_depth* and *mem_mode* as defined in the following:

Table 1-2 Parameter Description

Parameter	Values	Description
data_s_width	1 to 1024 Default: 16	Vector width of input `data_s` Note: *data_s_width* must be in an integer-multiple relationship with *data_d_width*. That is, either: *data_s_width* = K *data_d_width*, or *data_d_width* = K *data_s_width*, where K is an integer
data_d_width	1 to 1024 Default: 8	Vector width of output `data_d` Note: *'data_d_width*' must be in an integer-multiple relationship with '*data_s_width*'. That is, either: *data_s_width* = K *data_d_width*, or *data_d_width* = K *data_s_width*, where K is an integer.

Table 1-2 Parameter Description (Continued)

Parameter	Values	Description
ram_depth[a]	4 to 1024 Default: 8	Depth of RAM
mem_mode	0 to 7 Default: 3	Memory Control/Datapath Pipelining. Defines where and how many re-timing stages in RAM: 0 = No pre or post retiming 1 = RAM data out (post) re-timing 2 = RAM read address (pre) re-timing 3 = RAM data out and read address re-timing 4 = RAM write interface (pre) re-timing 5 = RAM write interface and RAM data out re-timing 6 = RAM write interface and read address re-timing 7 = RAM data out, write interface and read address re-timing
arch_type[b]	0 or 1 Default: 0	Pre-fetch cache architecture type. 0 = Pipeline style 1 = Register File style
f_sync_type	0 to 4 Default: 2	Forward Synchronization Stages (direction from source to destination domains) 0 = No synchronizing stages 1 = 2-stage synchronization with 1st stage negative edge and 2nd stage positive edge capturing 2 = 2-stage synchronization with both stages positive edge capturing 3 = 3-stage synchronization with all stages positive edge capturing 3 = 3-stage synchronization with all stages positive edge capturing 4 = 4-stage synchronization with all stages positive edge capturing
r_sync_type	0 to 4 Default: 2	Return Synchronization Stages (direction from destination to source domains). 0 = No synchronizing stages 1 = 2-stage synchronization w/ 1st stage negative edge and 2nd stage positive edge capture 2 = 2-stage synchronization w/ both stages positive edge capture 3 = 3-stage synchronization w/ all stages positive edge capture 4 = 4-stage synchronization w/ all stages positive edge capture
byte_order	0 to 1 Default: 0	Sub-word ordering into Word 0 = The first byte (or sub-word) is in MSB of word 1 = The first byte (or sub-word) is in LSB of word
flush_value	0 to 1 Default: 0	Fixed flushing value 0 = Fill empty bits of partial word with 0's upon flush 1 = Fill empty bits of partial word with 1's upon flush Note: only used when *data_s_width* < *data_d_width*.

Table 1-2 Parameter Description (Continued)

Parameter	Values	Description
clk_ratio	-7 to -1, 0, or 1 to 7 Default: 1	Rounded quotient between `clk_s` and `clk_d` frequencies. NOTE: This parameter is ignored when *mem_mode* is 0 or 1, and should be set to the default value. See "When is it necessary to Determine clk_ratio" on page 7. 1 to 7 = when `clk_d` rate faster than `clk_s` rate: round(`clk_d` rate / `clk_s` rate) -7 to -1 = when `clk_d` rate slower than `clk_s` rate: 0 - round(clk_s rate / clk_d rate) 0 = No restriction on clock `clk_s` and `clk_d` relationship (will incur a small performance degradation to retime synchronized pointers into each clock domain)
ram_re_ext	0 or 1 Default: 0	Extend ram_re_d_n during active read through RAM 0 = Single-pulse of `ram_re_d_n` at read event of RAM 1 = Extend assertion of `ram_re_d_n` while active read event traverses through RAM
err_mode	0 or 1 Default: 0	Error Reporting 0 = Sticky error flag 1 = Dynamic error flag
tst_mode		Test Mode 0 = No 'latch' is inserted for scan testing 1 = Insert negative-edge capturing flip-flip on `data_s` input vector when test input is asserted 2 = Insert hold latch using active low latch
verif_en	0 to 4 Default: 2	Verification Enable Control 0 = No sampling errors inserted 1 = Sampling errors are randomly inserted with 0 or up to 1 destination clock cycle delays 2 = Sampling errors are randomly inserted with 0, 0.5, 1, or 1.5 destination clock cycle delays 3 = Sampling errors are randomly inserted with 0, 1, 2, or 3 destination clock cycle delays 4 = Sampling errors are randomly inserted with 0 or up to 0.5 destination clock cycle delays Note: For more about *verif_en*, see "Simulation Methodology" on page 9.

a. Parameter *ram_depth* is not necessarily the number of RAM locations needed to operate the FIFO; for more, see "Memory Depth Considerations and Setting ram_depth" on page 14.
b. Note: For *arch_type* equal to 1, the RF cache is used (lpwr synthesis implementation) when a "DW_minpower" license is available and mem_mode is not 0 or 4. If a "minpower" license is not available or *mem_mode* is 0 or 4, the PL cache (rtl synthesis implementation) is always used no matter the setting of *arch_type*.

Table 1-3 Synthesis Implementations

Implementation Name	Function	License Feature Required
rtl	Ripple carry synthesis model	DesignWare

Table 1-4 Simulation Models

Model	Function
DW03.DW_ASYMFIFOCTL_2C_DF_CFG_SIM	Design unit name for VHDL simulation
DW03.DW_ASYMFIFOCTL_2C_DF_CFG_SIM_MS	Design unit name for VHDL simulation with mis-sampling enabled
dw/dw03/src/DW_asymfifoctl_2c_df_sim.vhd	VHDL simulation model source code (modeling RTL) - no mis-sampling
dw/sim_ver/DW_asymfifoctl_2c_df.v	Verilog simulation model[a] source code

a. To use this simulation model, the '+v2k' options needs to be used on the VCS command line.

eff_depth (effective depth) is not a user parameter but is used here as a placeholder which is derived from the parameters *ram_depth* and *mem_mode* as defined in the following:

Table 1-5 Effective Depth of FIFO

Effective depth value based on *ram_depth* and *mem_mode*
eff_depth = *ram_depth*+1 when *mem_mode* = 0 or 4
eff_depth = *ram_depth*+2 when *mem_mode* = 1, 2, 5, or 6
eff_depth = *ram_depth*+3 when *mem_mode* = 3 or 7

When is it necessary to Determine *clk_ratio*

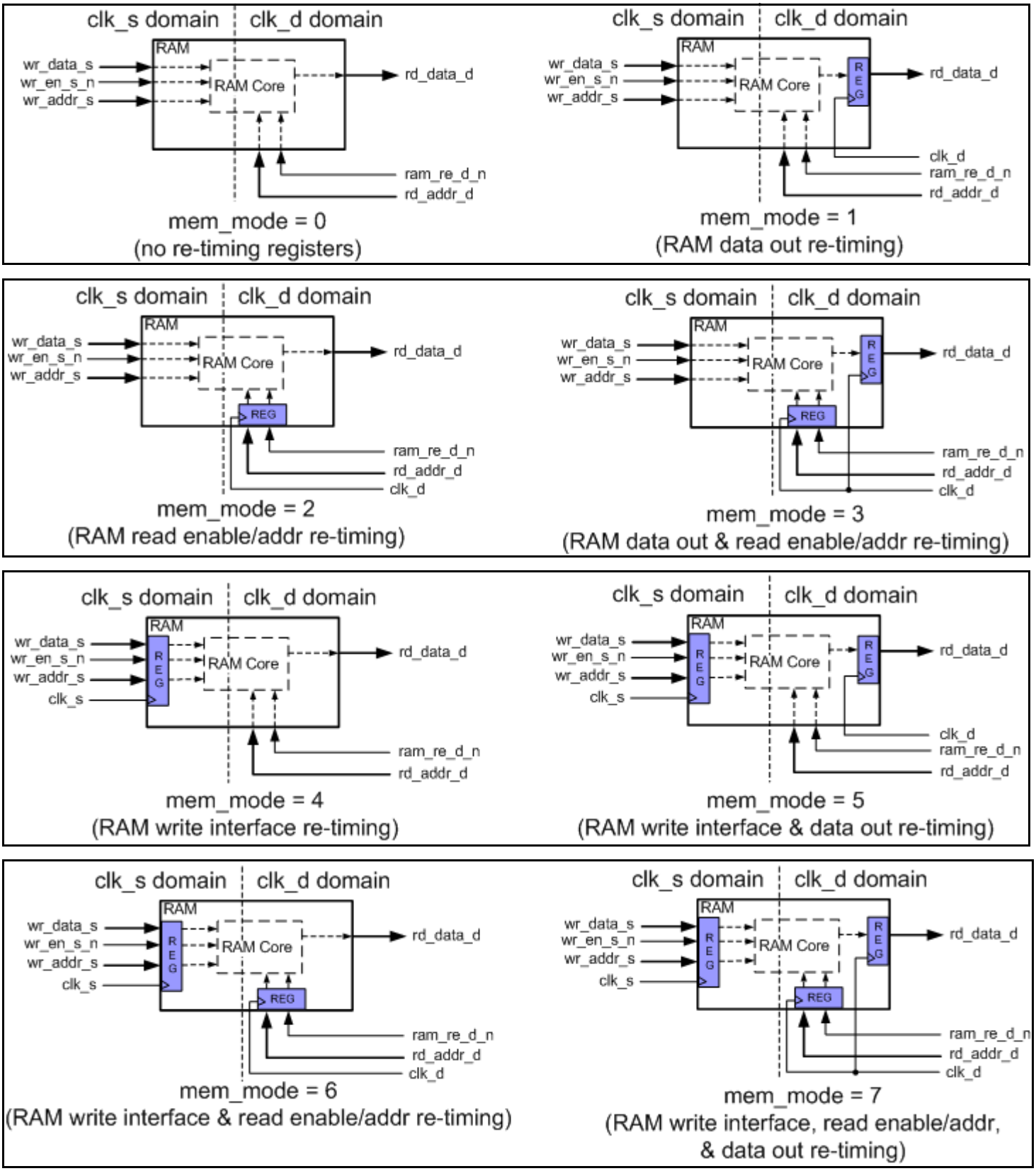
The parameter *clk_ratio* is only relevant when the parameter *mem_mode* indicates there are retiming registers on the input (write port or read port or both) of the RAM being used for the FIFO. When *mem_mode* is set to either 0 (no retiming registers in the RAM), or 1 (retiming registers only at the RAM data output port), the value of the parameter *clk_ratio* doesn't matter and should use the default value. For designs without a fixed clock ratio, it is least restrictive to use a configuration with the parameter *mem_mode* set to either 0 or 1, since the design will operate properly with any clock ratio -- with source faster than destination or destination faster than source at any ratio.

The special case of setting *clk_ratio* to 0 allows the design to operate properly regardless of the frequency relationship between `clk_s` and `clk_d` as well as for any RAM configuration (meaning for any value of the *mem_mode* parameter). This is useful when a design needs to interface to a data stream with characteristics that are not known until a connection is made. However, if a design will always operate with a specific clock ratio, setting *clk_ratio* to 0 could result in a design with more registers than necessary and lead to more latency than necessary.

Detailed Description of *mem_mode* Setting

To set the *mem_mode* parameter properly, knowledge of the RAM used with the DW_asymfifoctl_2c_df is needed. The following diagrams show the 8 possible RAM architectures that can interface with DW_asymfifoctl_2c_df and the required *mem_mode* setting for each."Simulation Methodology" on page 9"Memory Depth Considerations and Setting ram_depth" on page 14

Figure 1-1 *mem_mode* Settings based on RAM Architecture



Simulation Methodology

Since the DW_fifoctl_2c_df contains the DW_gray_sync and DW_sync (synchronizing devices) there are two methods available for simulation. One method is to utilize the simulation models as they emulate the RTL model. The other method is to enable modeling of random skew between bits of signals traversing to and from each domain (denoted as “missampling” here on out). When using the simulation models purely to behave as the RTL model, not special configuration is required. When using the simulation models to enable missampling, unique considerations must be made between Verilog and VHDL environments.

For Verilog simulation enabling missampling a preprocessing variable named DW_MODEL_MISSAMPLES must be defined as follows:

```
`define DW_MODEL_MISSAMPLES
```

Once `DW_MODEL_MISSAMPLES is defined, the value of the *verif_en* parameter comes into play and configures the simulation model as described by Table 1-6 on page 9. Note: If `DW_MODEL_MISSAMPLES is not defined, the Verilog simulation model behaves as if *verif_en* was set to 0.

For VHDL simulation enabling missampling, an alternative simulation architecture is provided. This architecture is named sim_ms. The parameter *verif_en* only has meaning when utilizing the sim_ms. That is, when binding the “sim” simulation architecture the *verif_en* value is ignored and the model effectively behaves as though *verif_en* is set to 0. See “HDL Usage Through Component Instantiation - VHDL” on page 51 for an example utilizing each architecture.

Simulation Assertions (SystemVerilog only)

The Verilog simulation model incorporates SystemVerilog assertions covering functionality in both clock domains. By default, all the assertions have a reporting severity of 'error' but they can be changed to report as warning, fatal, or not at all by defining the preprocessing variable named DW_SVA_MODE. Not defining DW_SVA_MODE behaves as if it were defined as 2 (report as error). So, for example, if the desire is to have all the assertions in this component report with severity of 'warning' do the following:

```
`define DW_SVA_MODE  1
```

See Table 1-6 for the assertion reporting severity based on the DW_SVA_MODE value. It is important to note the value of DW_SVA_MODE determines the same reporting severity of all the assertions within this component.

Table 1-6 Assertion Report Severity

DV_SVA_MODE	Assertion Report Severity
not defined (default)	error
0	disable reporting
1	warning
2	error
3	fatal

The following is a list of some (but not all) of the assertions included in this component:

- Word counts (`fifo_word_cnt_s`, `word_cnt_s`, `ram_word_cnt_d`, `word_cnt_d`) are legal values

- Status flags (full and empty in both clock domains) are in the proper state under system reset and clearing conditions.
- Full and empty status are in the proper states when not in system reset or clearing conditions
- Addresses (and internal pointers) do not change under push and/or pop error cases.

Note: For assertions related to word counts, initiation of system reset in one domain could result, temporarily, in the pointer math in the other domain to calculate values outside of the acceptable depth of the RAM or FIFO. However, after reset conditions, namely after both domains have been initialized and held idle as described in "System Resets (synchronous and asynchronous)" on page 37, the word counts will settle into their allowed ranges.

The following diagram is a basic block diagrams of the DW_asymifoctl_2c_df for both cases when *data_s_width* and *data_d_width* are not equal. For reference purposes, a RAM is included in the diagram but does not exist in the DW_asymfifoctl_2c_df component.

Functional Description

The DW_asymfifoctl_2c_df, depending on the relationship between the push (source clock domain) and the pop (destination clock domain) data widths, buffers up the data either written to or read from the RAM. The main purpose is of this component is to deal with data widths between the two domains that are not equal. When the pop domain data width is a multiple integer larger than the push domain data width, buffering of the data is made before writing to the RAM. Conversely, when the push domain data width is a multiple integer larger than the pop domain data width, reads from RAM are buffered up before being streamed out.

Additionally, this component performs word caching (or pre-fetching) in the pop interface to minimize latencies via a RAM by-pass feature which allows for bursting of contiguous words and provides registered data to the external logic.

Two implementations are available; one each to cover the standard DesignWare library via the "rtl" implementation and the "minPower" library with the "lpwr" implementation. The only difference between the two implementations is that "rtl" will use the DW_asymfifoctl_2c_df "rtl" implementation and "lpwr" will incorporate the DW_asymfifoctl_2c_df "lpwr" implementation. Details of the difference between the two DW_asymfifoctl_2c_df implementations will not be covered in this document.

Synchronous RAM that is supported can have one of the following architectures:

- Non re-timed write port and asynchronous read port
- Re-timed write port and asynchronous read port
- Non re-timed write port and synchronous read port with buffered read address and non-buffered read data
- Non re-timed write port and synchronous read port with non-buffered read address and buffered read data
- Non re-timed write port and synchronous read port with buffered read address and buffered read data
- Re-timed write port and synchronous read port with buffered read address and non-buffered read data
- Re-timed write port and synchronous read port with non-buffered read address and buffered read data
- Re-timed write port and synchronous read port with buffered read address and buffered read data

The FIFO controller generates RAM addressing, write enable logic (source domain), read enable and address logic (destination domain), and a comprehensive set of status flags (empty, almost empty, half full, almost full, and full) and operation error detection logic for both clock domains.

The FIFO controller provides parameterized data widths, RAM depth, pop data pipelined stages (pre-fetching cache), almost empty and almost full levels that are all configurable upon module instantiation.

To accommodate the dual clock environment, parameters are provided to adjust the number of synchronization stages needed in both directions between the two clock domains.

To provide a clean reset environment, the FIFO controller contains reset logic that coordinates clearing of both clock domains in a controlled and orchestrated algorithm for localized resets operations (see Clearing FIFO Controller section).

Unless otherwise stated here on out, the term FIFO means the grouping of the RAM module and pre-fetching cache.

Detailed Description of Parameter *arch_type*

The *arch_type* parameter is available for selection of the pre-fetch cache structure contained in the underlying DW_fifoctl_2c_df that resides in the data path after the RAM. This provides flexibility in choosing the best cache structure that yields the lowest power consumption based on system characteristics. See the “Pre-fetch Cache Architectures” section for more details regarding power aspects.

If arch_type is 0, the DW_fifoctl_2c_df uses the “pipeline” (PL) cache structure which is the rtl implementation. When arch_type is 1 and *mem_mode* setting is such that the pre-fetch cache depth is 2 or 3 (See Table 12: Pop Interface Cache Sizes), then a “register file” (RF) style of caching (lpwr implementation) is used provided a “minpower” license is available.

When the RF Cache structure is desired and this component is configured accordingly, the lpwr implementation is automatically selected when a “minpower” license is available. Only a “set implementation” to the rtl implementation will override the selection of the lpwr implementation. If no “minpower” license is available, but the component is configured to attempt to use the RF Cache (for example, the lpwr implementation), the rtl implementation will be used instead which pertains the PL cache structure. In general, whenever the PL Cache is desired and the component is configured as such then a “minpower” license is not consumed.

The following table shows how the arch_type setting along with the mem_mode setting and license availability determines the implementation and, hence, the style of pre-fetch cache that is utilized.

Table 1-7 Implementation Availability Based on *arch_type* and *mem_mode*

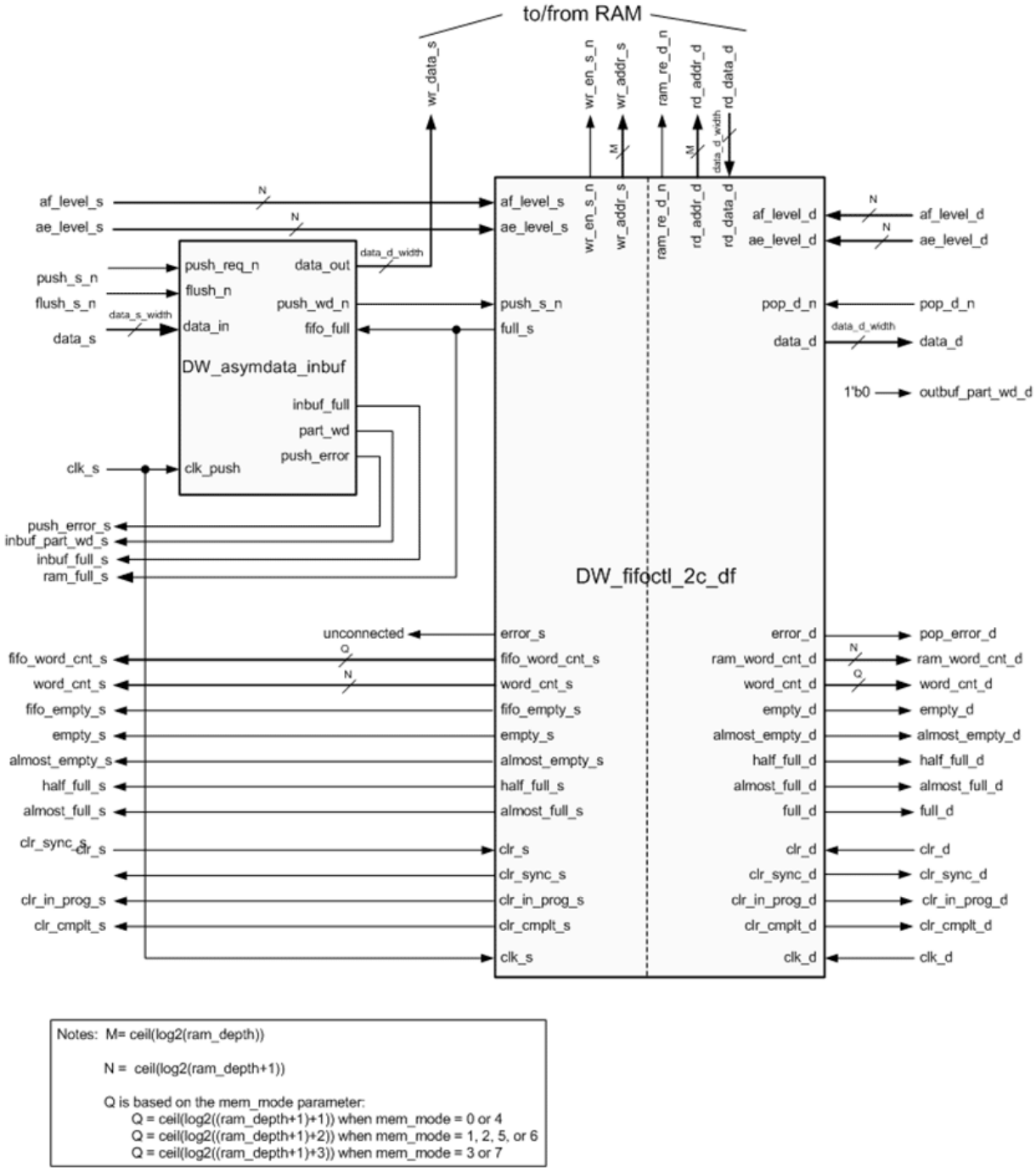
arch_type	mem_mode	Implementation Available
0	x	rtl
1	4 or 5	rtl
1	1-3, 6-7	rtl, lpwr[a]

a. NOTE: The lpwr implementation is only available with a “minpower” license. If a “minpower” license is available, for these *arch_type* and *mem_mode* settings the lpwr implementation is automatically selected over the rtl unless overridden by set implementation. When the rtl implementation is used, a “minpower” license is not consumed.

Block Diagrams

The following are the block diagrams for the DW_asymfifoctl_2c_df component for *data_s_width* < *data_d_width*.

Figure 1-2 DW_asymfifoctl_2c_df for *data_s_width* < *data_d_width*



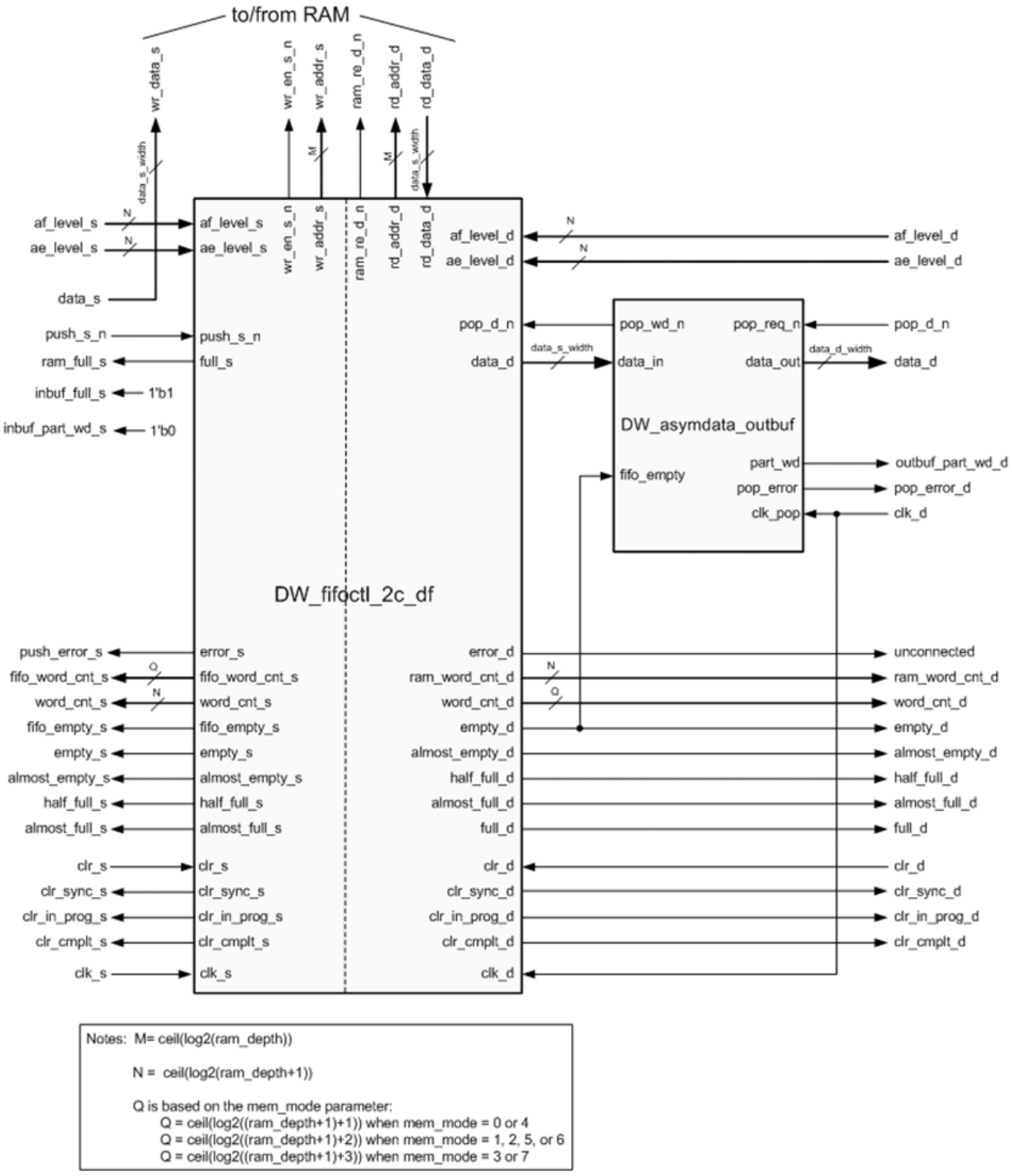
Notes: M= ceil(log2(ram_depth))

N = ceil(log2(ram_depth+1))

Q is based on the mem_mode parameter:
Q = ceil(log2((ram_depth+1)+1)) when mem_mode = 0 or 4
Q = ceil(log2((ram_depth+1)+2)) when mem_mode = 1, 2, 5, or 6
Q = ceil(log2((ram_depth+1)+3)) when mem_mode = 3 or 7

The following are the block diagrams for the DW_asymfifoctl_2c_df component for *data_s_width* >= *data_d_width*.

Figure 1-3 DW_asymfifoctl_2c_df for *data_s_width* >= *data_d_width*



Memory Depth Considerations and Setting *ram_depth*

Depending on the desired FIFO depth of the design, the *ram_depth* must be set accordingly.

Ultimately, the RAM must contain an even number of locations. Based on the desired number of FIFO locations, consider the following three cases in choosing the RAM size and *ram_depth* setting.

Case 1: If an odd number of FIFO locations are required by the system (from the push interface), call that 'x', then the parameter *ram_depth* should be set to 'x'. But, the RAM size should be 'x' + 1. The FIFO controller RAM addresses range from 0 to *ram_depth*.

Table 1-8 Desired Number of FIFO Locations is Odd

FIFO Locations Desired	RAM Size	RAM Address Range	*ram_depth* value
11	12	0 to 11	11
31	32	0 to 31	31

Case 2: If an even number of FIFO locations is required by the system (from the push interface) but that number is not an integer power of two, call it 'y' locations, then the parameter *ram_depth* should be set to 'y'. But, the size of the RAM should be 'y' + 2. The FIFO controller RAM addresses range from 0 to *ram_depth*+1.

Table 1-9 Desired Number of FIFO Locations is Even but not Power of 2

FIFO Locations Desired	RAM Size	RAM Address Range	*ram_depth* value
12	14	0 to 13	12
34	36	0 to 35	34

Case 3: If an even number of FIFO locations is needed in the system (from the push interface) and it is an integer power of two, for example, 4, 8, 16, 32,..., then set the *ram_depth* to exactly the desired FIFO depth and the number of RAM locations accessed will also be the value of *ram_depth*. The FIFO controller RAM addresses range from 0 to *ram_depth*-1.

Table 1-10 Desired Number of FIFO Locations is Even and Power of 2

FIFO Locations Desired	RAM Size	RAM Address Range	*ram_depth* value
12	14	0 to 13	12
34	36	0 to 35	34

These restrictions are derived from the following facts:

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

Writing to the Memory when *data_s_width* < *data_d_width* (push)

Figure 1-2 shows the top-level block diagram for configurations when *data_s_width* < *data_d_width*. The DW_asymdata_inbuf comes into play and the RAM width is determined by the largest value setting of the parameters *data_s_width* and *data_d_width*. So, in this case, the width for RAM will be *data_d_width*.

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

The RAM is written and `wr_addr_s` (which always points to the address of the next word to be pushed) is incremented on the same rising edge of `clk_s` (the first clock after `push_s_n` is asserted). This means that `push_s_n` must be asserted early enough to propagate through the FIFO controller to the RAM before the ensuing clock.

Pushing and Flushing

The following table shows the action that takes place for each state of `flush_s_n` and `push_s_n` based on the state of the input buffer and RAM.

Table 1-11 Flush and Push scenarios and results

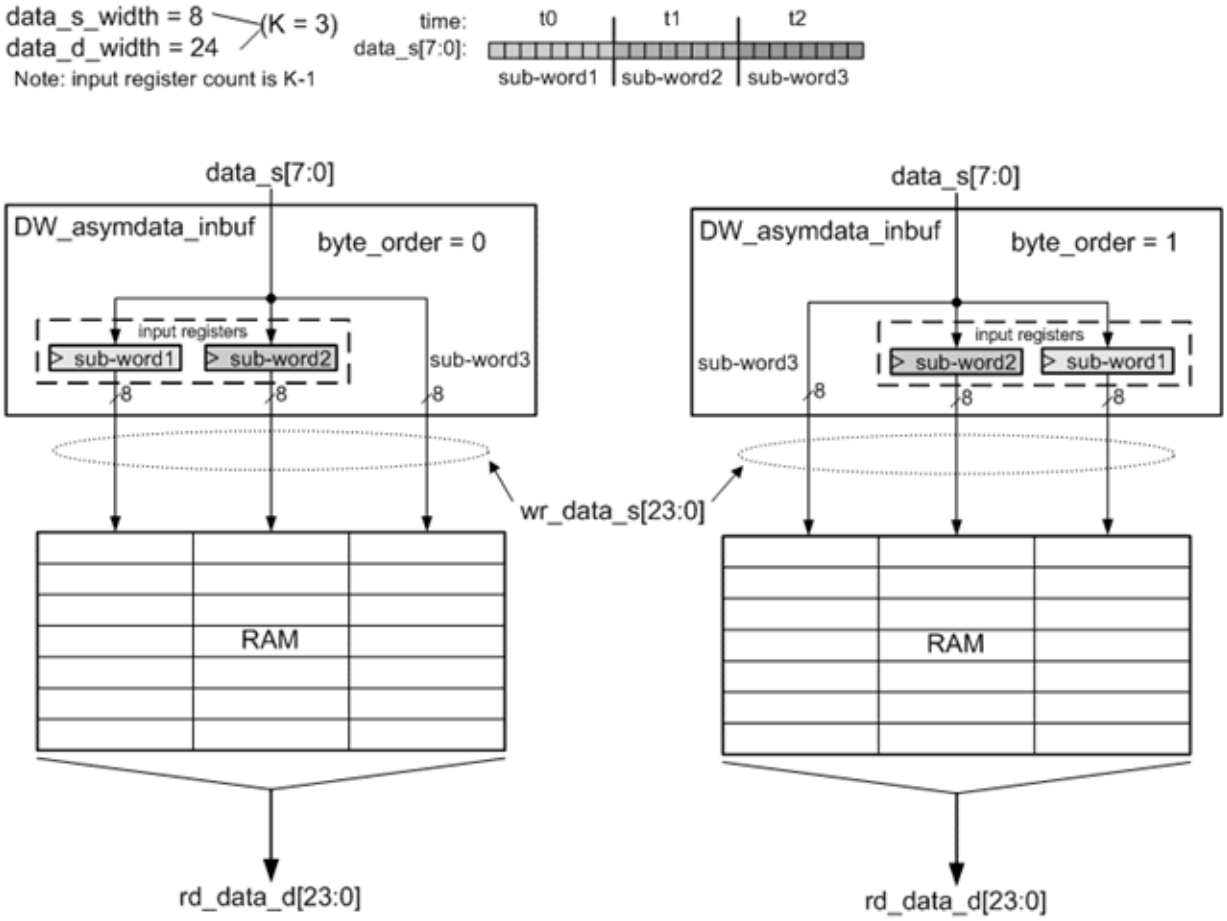
flush_s_n	push_s_n	inbuf_part_wd_s	inbuf_full_s	ram_full_s	Results/Action
1	0	x	0	x	Push only, no error
0	0	0	0	x	No flush, push only, no error
1	0	1	1	0	Generate wr_en_s_n, reset counters, no error
x	0	1	1	1	push error, last sub-word lost, hold counters
0	1	1	x	0	flush only, reset counters, no error*
0	1	0	0	x	do nothing
0	0	1	x	0	flush and push (`data_s` into sub-word1 input reg.), no error[a]
0	1	1	x	1	push error, no other action
0	0	1	0	1	no flush, push data, push error
1	1	x	x	x	do nothing
x	x	0	1	x	not possible

a. During a valid flush condition, `wr_en_s_n` is asserted.

Input Buffer Data-flow Details (DW_asymdata_inbuf)

When *data_s_width* is less than *data_d_width*, a data input buffer is put in place from the source domain to the RAM. The following figure shows how the input registers are organized and routed to the RAM element (via `wr_data_s`) based on the value of *byte_order* to a RAM element. Note that the number of sub-word input registers is equal to (*data_d_width* / *data_s_width*) - 1.

Figure 1-4 Buffer Data flow based on 'byte_order' value



Partial Words

When a partial word is in the input registers, output flag `inbuf_part_wd_s` is active (HIGH). After K pushes, where K = *data_d_width* / *data_s_width*, K sub-words are assembled into a full word (K-1 sub-words in the input register and the last sub-word on the `data_s` bus). This full word is then presented for writing into RAM. When a full word is sent from the input registers to RAM, `inbuf_part_wd_d` goes inactive (LOW) on the following cycle. The order of sub-words within a word is determined by the *byte_order* parameter (as shown in Figure 1-3).

Pushing Complete Words

As the Kth sub-word is being pushed (`push_s_n` asserted LOW), the complete word is presented to the `wr_data_s` output along with assertion of `wr_en_s_n` (LOW single `clk_s` cycle). The `wr_en_s_n` output is non-registered single-clock pulse and is a combinational result derived directly from `push_s_n` (and `flush_s_n`). So, be aware of the timing characteristics of `push_s_n`.

Flushing Input Registers

The flushing function, via `flush_s_n` assertion, pushes a partial word to RAM/FIFO. The input registers are pre-set to the *flush_value* value after a flush with the exception of one case. This case is when simultaneous assertion of `flush_s_n` and `push_s_n` when `inbuf_part_wd_s` is asserted and `ram_full_s` is not asserted. Under this scenario, the sub-word from `data_s` is placed into the sub-word1 input register with all other sub-word locations pre-set the *flush_value* value.

A complete list of results caused by `flush_s_n` assertion is shown in Table 11. Figure 1-15 is a timing waveform that shows an example of the flush operation.

Push Errors

A push error occurs if:

1. `push_s_n` active, `inbuf_part_wd_s` active, `inbuf_full_s` active, and `ram_full_s` active -OR-
2. `flush_s_n` active, `inbuf_part_wd_s` active, and `ram_full_s` active

The `push_error_s` output is registered and its behavior can either be 'sticky' or 'dynamic' based on the *err_mode* parameter. See Table 3 for the *err_mode* definition.

Writing to RAM when *data_s_width* >= *data_d_width*

Figure 1-3 shows the top-level block diagram for configurations when *data_s_width* > *data_d_width*. The DW_asymdata_inbuf is not present and the RAM width is determined by the large value setting of the parameters *data_s_width* and *data_d_width*. So, in this case, the width for RAM will be *data_s_width*. For *data_s_width* = *data_d_width* the following discussion of writing to RAM also applies.

The `wr_addr_s` and `wr_en_s_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_s_n` input is asserted (active low), and
- The `full_s` flag is inactive (low) at the rising edge of `clk_s`.

Asserting `push_s_n` when `ram_full_s` is inactive causes the following to occur:

- The `wr_en_s_n` is asserted immediately, preparing for a write to the RAM on the next rising `clk_s`, and
- On the next rising edge of `clk_s`, `wr_addr_s` is incremented (module depth).

Thus, the RAM is written and `wr_addr_s` (which always points to the address of the next word to be pushed) is incremented on the same rising edge of `clk_s` (the first clock after `push_s_n` is asserted). This

means that push_s_n must be asserted early enough to propagate through the FIFO controller to the RAM before the ensuing clock.

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

Write Errors

An error occurs if a push operation is attempted while the RAM is full. That is, the push_error_s output goes active if:

- The push_s_n input is asserted (low), and
- The ram_full_s flag is active (high) on the rising edge of clk_s.

When a push error occurs, wr_en_s_n stays inactive (HIGH) and the write address, wr_addr_s, does not advance. After a push error, although a data word was lost at the time of the error, the FIFO remains in a valid full state and can continue to operate properly with respect to the data that was contained in the FIFO before the push error occurred.

Reading from the Memory when *data_s_width* > *data_d_width*

When *data_s_width* is greater than *data_d_width*, the data output interface comes directly from the subcomponent DW_asymdata_outbuf as seen in Figure 1-3. In the read path from RAM contains the DW_fifoctl_2c_df pre-fetch cache (configured by the *arch_type* and *mem_mode* parameter settings) and the DW_asymdata_outbuf. The followings section describes the control of reading the RAM and the data flow through the DW_asymfifoctl_2c_df pre-fetch cache and DW_asymdata_outbuf.

Reading from RAM

The read port of the RAM must be asynchronous or synchronous with its own clock (unique from the write port's clock). All read data from RAM is first loaded into the pre-fetching cache. The rd_addr_d output port of the DW_fifoctl_2c_df provides the read address to the RAM. rd_addr_d points to (pre-fetches) the next word of RAM read data to be loaded to cache. Reading of RAM is initiated by two methods

1. The RAM is not empty and the cache is not full,
2. pop_wd_n from DW_asymdata_outbuf is asserted while the cache contains at least one valid data entry (and the RAM is not empty).

In detail, the RAM read operation occurs under two scenarios:

- The internally synchronized (to clk_d) write and read pointers indicate that the RAM is not empty and the pre-fetching cache is not full.
- The pop_wd_n from DW_asymdata_outbuf is asserted (low) and the RAM is not empty at the rising edge of clk_d.

Asserting pop_d_n of DW_fifoctl_2c_df (via pop_wd_n from DW_asymdata_outbuf) while empty_d is not active causes the internal read pointer to increment on the next rising edge of clk_d only if the RAM contains at least one valid entry. Therefore, for asynchronous read port memories, the RAM read data must be captured in the cache on the rising edge of clk_d following the assertion of pop_d_n.

For synchronous read port memories; when either rd_addr_d or RAM data out (rd_data_d) is buffered, data is captured by the cache on the rising edge of clk_d one cycle after the clk_d edge that directed the controller to read, or when both rd_addr_d and RAM data out are buffered then data is captured by the cache on the rising edge of clk_d two cycles after the initiating clk_d edge that directed the controller to read.

If the RAM is empty at the assertion of pop_d_n, the internal read pointer does not advance.

Destination Domain Caching (pop interface pre-fetching)

The popping interface contains output caching (pre-fetching cache) with the number of pipeline stages determined by the *mem_mode* parameter. When the cache is not fully populated with valid data and the RAM is detected as having valid entries, data is automatically pre-fetched into the cache to provide immediate data availability at pop requests no matter which mode of read port the RAM is using (asynchronous versus 1-deep synchronous versus 2-deep synchronous). An extra latency applies not only to when the first data word arrives at the pop interface but also to when FIFO 'fullness' information is delivered to the word_cnt_d and pop_* interface status flag ports.

At a minimum, there will always be at last one buffering stage in the cache which is seen at the pop_* interface. However, only the first word of a burst of words incurs one clk_d cycle latency before being read from the RAM and presented to the pop_* interface. Below is a list identifying the number of pre-fetching stages of the cache used based on the *mem_mode* parameter value.

Table 1-12 Pop Interface Cache Sizes

***mem_mode* values**	**Number of caching stages**
0 or 4	1
1, 2, 5, or 6	2
3 or 7	3

Popping from the Cache

The cache is the data interface of the FIFO and it is made up of data word entries based on the *mem_mode* parameter as described in Table 12. When the cache contains at least one valid data word it is considered not empty, meaning the empty_d flag is not asserted, and a legal pop of the cache is allowed (asserting DW_fifoctl_2c_df pop_d_n). The assertion of DW_fifoctl_2c_df pop_d_n (from DW_asymdata_outbuf pop_wd_n) causes the cache control mechanism advances to looks for the next available data from RAM on the next rising edge of clk_d. If active RAM data out (rd_data_d) is available, rd_data_d is loaded into the next available entry of the cache. Controls are in place to prevent the cache from overflowing.

However, if the cache contains only one valid data entry and rd_data_d is not valid and pop_d_n is asserted, then on the next rising edge of clk_d the data value selected by the cache control logic (data_d) is held AND the empty_d flag gets asserted. Thus, assertion of the empty_d flag declares the contents at the data_d port of DW_fifoctl_2c_df irrelevant.

It is worth noting that due to the pipelining nature of the read data path, the read control logic of the cache may not contain relevant data, hence the cache is declared 'empty', but data from previous read operations could be in transit and making their way for writing into the cache. So, 'empty' only means that the read

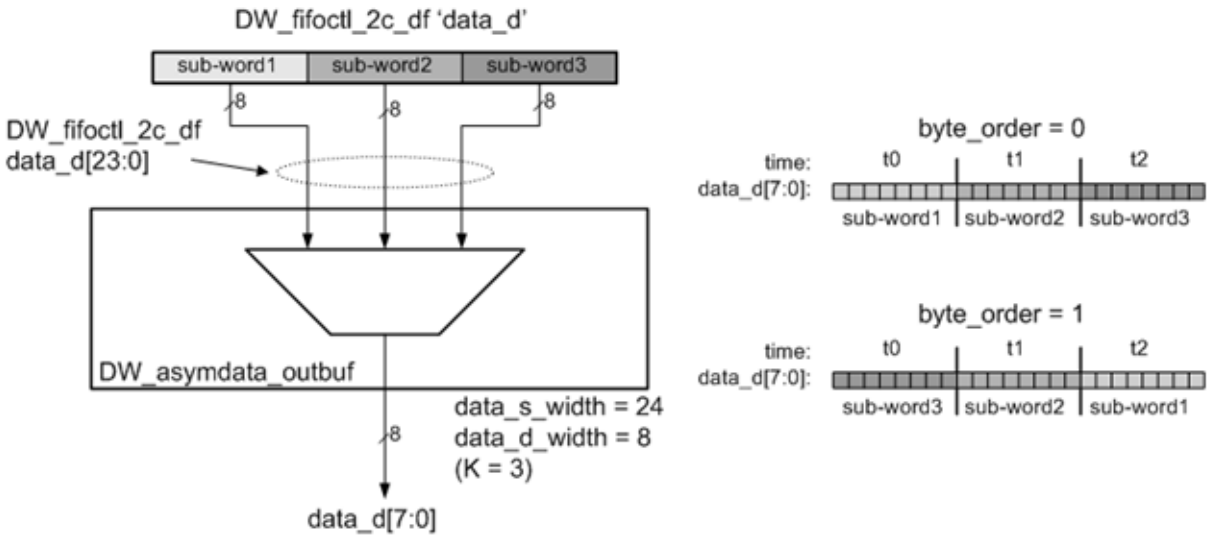
address entry of the cache does not contain relevant data, but there could still be relevant data being actively processed through the read data path.

Output Buffer Data Flow Details

The following figure shows how the output multiplexer routes `data_d` of DW_fifoctl_2c_df to `data_d` of DW_asymfifoctl_2c_df (based on the value of *byte_order*) via the DW_asymdata_outbuf.

When the DW_asymdata_outbuf requests a 'pop' from DW_fifoctl_2c_df by driving `pop_wd_n` low when `empty_d` is not asserted, the pre-fetch cache data of DW_fifoctl_2c_df (`data_d`) is captured by the output buffer for serialization out to `data_d` of the DW_asymfifoctl_2c_df.

Figure 1-5 Output Buffer Data Flow Based on *byte_order* Value



Partial Words

When the sub-words are being multiplexed to 'data_d' before the full word of 'DW_fifoctl_2c_df `data_d` is traversed, the output flag `outbuf_part_wd_d` is active (HIGH). After K pops, where K = *data_s_width* / *data_d_width*, K sub-words are presented to the `data_d` output. When all sub-words for a particular word from the cache of DW_fifoctl_2c_df are sent out, `outbuf_part_wd_d` goes inactive (LOW) on the following cycle. The order of how the sub-words are retrieved from a word is determined by the *byte_order* parameter (as shown in Figure 1-5).

Popping Complete Words

As the Kth sub-word is being popped (`pop_d_n` asserted LOW) to supply `data_d`, the `data_d` of DW_fifoctl_2c_df complete word has then been traversed, then an internal derived 'pop' signal asserts for a single `clk_d` cycle and drives the `pop_d_n` of the underlying DW_fifoctl_2c_df. That internal 'pop' signal is a non-registered single-clock pulse and is a combinational result derived directly from `pop_d_n`. So, be aware of the timing characteristics of `pop_d_n`'

Pop Errors

A pop error occurs if pop_d_n asserts during active empty_d. The pop_error_d output is registered and its behavior can either be 'sticky' or 'dynamic' based on the *err_mode* parameter. See Table 3 for the *err_mode* definition.

Reading from the Memory when 'data_s_width' <= 'data_d_width'

The read port of the RAM must be asynchronous or synchronous with its own clock (unique from the write port's clock). All read data from RAM is first loaded into the pre-fetching cache. The rd_addr_d output port of the underlying DW_fifoctl_2c_df provides the read address to the RAM. rd_addr_d points to (pre-fetches) the next word of RAM read data to be loaded to cache. Reading of RAM is initiated by two methods:

1. the RAM is not empty and the cache is not full,
2. pop_d_n is asserted while the cache contains at least one valid data entry (and the RAM is not empty).

In detail, the RAM read operation occurs under two scenarios:

- The internally synchronized (to clk_d) write and read pointers indicate that the RAM is not empty and the pre-fetching cache is not full.
- The pop_d_n is asserted (low), empty_d flag is not active (low) (the head of the cache contains a valid data entry), and the RAM is not empty at the rising edge of clk_d.

Asserting pop_d_n while empty_d is not active causes the internal read pointer to increment on the next rising edge of clk_d only if the RAM contains at least one valid entry. Therefore, for asynchronous read port memories, the RAM read data must be captured in the cache on the rising edge of clk_d following the assertion of pop_d_n.

For synchronous read port memories, when either rd_addr_d or RAM data out (rd_data_d) is buffered, data is captured by the cache on the rising edge of clk_d one cycle after the clk_d edge that directed the controller to read; -or- when both rd_addr_d and RAM data out are buffered, then data is captured by the cache on the rising edge of clk_d two cycles after the initiating clk_d edge that directed the controller to read.

If the RAM is empty at the assertion of pop_d_n, the internal read pointer does not advance.

Popping from the Cache

The cache is the data interface of the FIFO and it is made up of data words entries based on the *mem_mode* parameter as described in Table 12. Whenever the cache contains a valid entry the FIFO is considered not empty, for example the empty_d flag is not asserted, and a legal pop of the FIFO is allowed (asserting pop_d_n). When empty_d is not asserted the data_d contains the next valid word from the FIFO. The assertion of pop_d_n causes the cache control logic to advance to the next valid data on the next rising edge of clk_d. If active RAM data out (rd_data_d) is available, rd_data_d is loaded into the next location of cache behind the previous written data. Controls are in place to prevent the cache from overflowing.

However, if the cache contains only one valid data entry and rd_data_d is not valid when pop_d_n is asserted, then on the next rising edge of clk_d no data is written into the cache and its contents are held; -

but- the `empty_d` flag gets asserted. Thus, assertion of the `empty_d` flag declares the contents at `data_d` irrelevant.

It is worth noting that due to the pipelining nature of the read data path, the head of the cache may not contain relevant data, hence the FIFO is declared "empty," but data from previous read operations could be in transit and making their way to the head of the cache. So, "empty" only means that the head of the cache does not contain relevant data, but there could still be relevant data being actively processed through the read data path.

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_d` output goes active if:

- The `pop_d_n` input is active (low), and
- The `empty_d` flag is active (high) on the rising edge of `clk_d`.

When a pop error occurs, the read address `rd_addr_d` does not advance. After a pop error, the FIFO is still in a valid empty state and can continue to operate properly.

Error Outputs and Flag Status

The error outputs and flags are initialized as follows:

- `empty_d`, `almost_empty_s`, `empty_d`, and `almost_empty_d` are initialized to 1
- All other flags and the error outputs are initialized to 0 (low)

Pre-fetch Cache Architectures and Power Considerations

As mentioned earlier, there are two pre-fetch cache architectures residing in the underlying DWfifoctl_2c_df which are parameter selectable that allows for power optimization: pipelined (PL) and register file (RF) types.

The PL caching style (rtl implementation) is effectively a shift register of 1, 2, or 3 stages. Active switching through each stage occurs during shifting initiated by pop requests with pending valid data in either the RAM or cache stages behind the head location. In cases with a wide data bus and cache configurations of 2 or 3 deep, this could represent the majority of the register switching power consumption within the component.

As an alternative architecture for cache depths of 2 or 3, the pre-fetch cache is organized as a RF structure ('lpwr' implementation). For the RF cache structure, the shifting between pipelined cache entries is eliminated and replaced with write and read pointer manipulation to access cache elements; a mini-FIFO of sorts.

The two caching architectures are provided to give the designer flexibility in selecting the caching architecture that will yield the lowest total power consumption.

Knowing which pre-fetch cache architecture to choose is highly dependent on factors such as technology, clock rate, data width, pre-fetch cache depth, and data flow characteristics through the associated FIFO.

With all variables being equal, the advantage that either cache architecture provides in terms of optimal power dissipation is based particularly on the type of data flow behavior through the underlying DW_fifoctl_2c_df.

Generally, there's no specific rule of thumb in selecting which cache architecture will render the least power dissipation. This may require the design process to include some experimentation in using both cache styles to characterize behavior based on the system parameters.

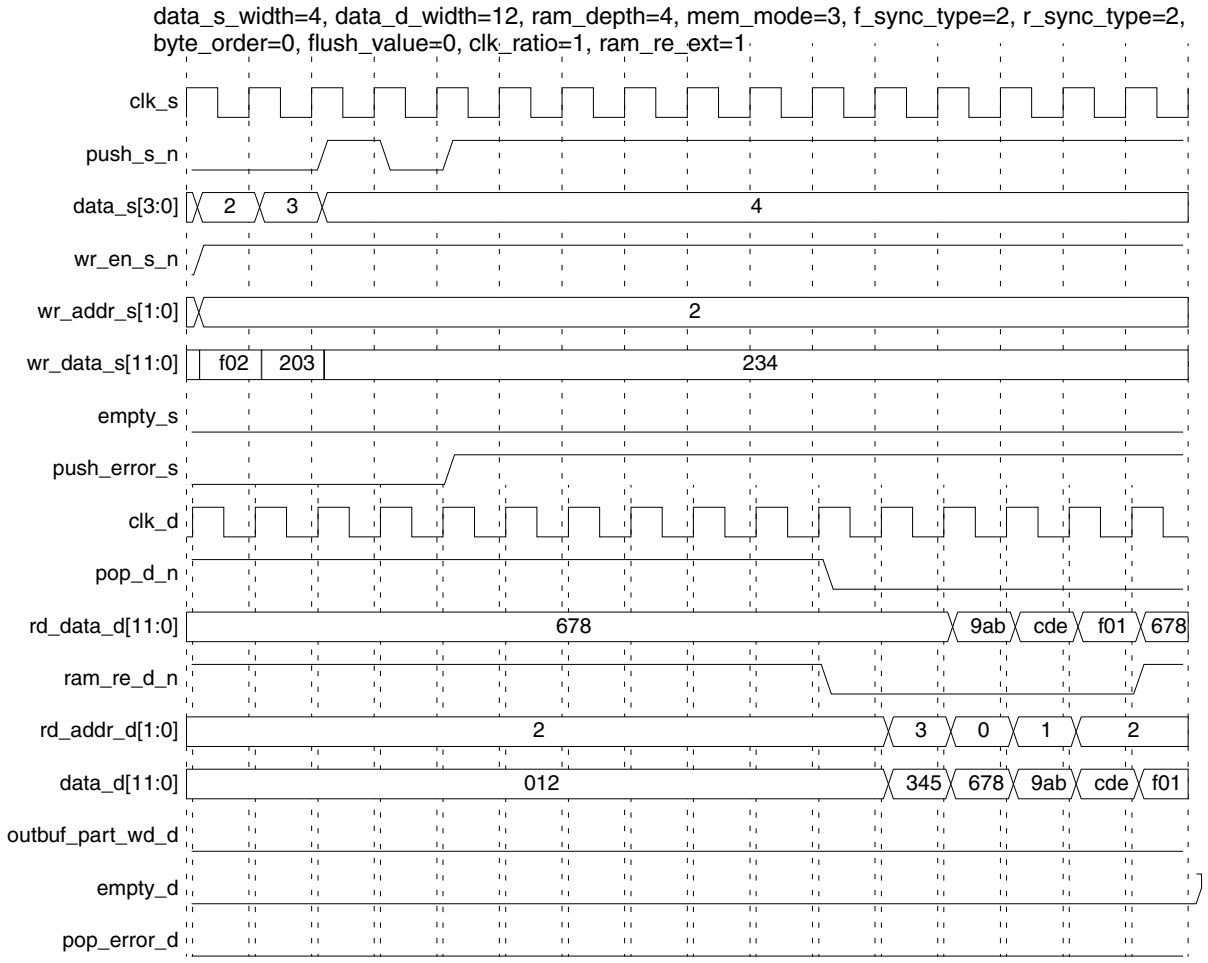
Keep in mind that criticality in choosing which pre-fetch cache architecture is only meaningful in a system when the parameter *mem_mode* is not 0 nor 4. That is, when the cache depth is 2 or 3, the selection of the cache architecture is relevant. When *mem_mode* is either 0 or 4, the pre-fetch cache depth is 1 and defaults to the 'rtl' implementation.

However, a starting point uses two data flow behaviors and the cache architecture that most likely will yield a better power result over the other. Of course, this assumes that this data flow is a significant power consumption factor through the DW_fifoctl_2c_df.

The key factor that determines which cache architecture will provide the least power consumption hinges around the behavior of the pop requests (`pop_d_n`) as the DW_fifoctl_2c_df sees it. That is, when *data_s_width* <= *data_d_width*, the behavior of `pop_d_n`. When *data_s_width* > *data_d_width*, the behavior of this internal “pop request” generated by the DW_asymdata_outbuf module which is initiated by `pop_d_n`. If pop requests are issued in short bursts of 3 or less, the RF cache (‘lpwr’ implementation) will most likely yield the least power consumption versus the PL cache architecture (‘rtl’ implementation). If however, pop requests that occur in longer contiguous bursts favor the PL cache architecture. The following three cases show the two extremes in data flow behavior.

CASE 1: When continuous bursts of contiguous pop requests are issued Figure 1-6 this produces a type of data flow more conducive to using the PL cache architecture.

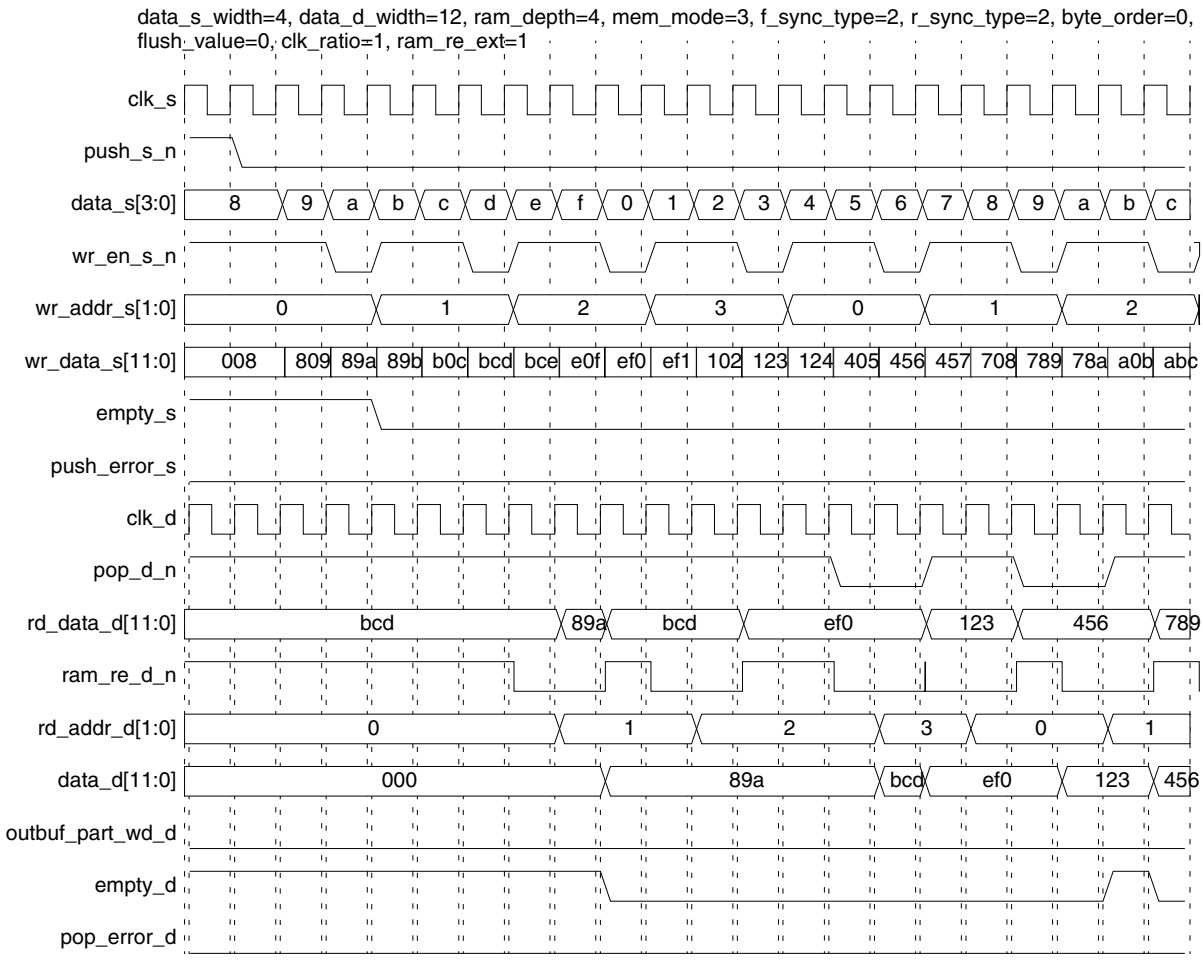
Figure 1-6 CASE 1: Popping in contiguous bursts



In this case *data_s_width* < *data_d_width* which mean `pop_d_n` is connected directly to the DW_fifoctl_2c_df. The power benefits of using the PL cache over the RF cache in this data flow condition comes from the fact that the front of the cache is continuously being written to and read from. From the PL cache perspective one stage of the cache is being used at a time and effectively the other stages of the cache are unused during this time. So, the dynamic power of shifting through many stages of the cache does not occur. In the RF cache however, data is being written to cache just like in the PL cache case but the write and read addressing logic is updating every cycle. This activity of the write and read addressing logic is the extra power consumption that the RF cache architecture has that the PL cache does not. Thus, the PL cache architecture would be optimal, in general, for this type of data flow.

CASE 2: With '*data_s_width* < *data_d_width*', the data flow behavior for Figure 1-7 shows a packet with words pushed contiguously.

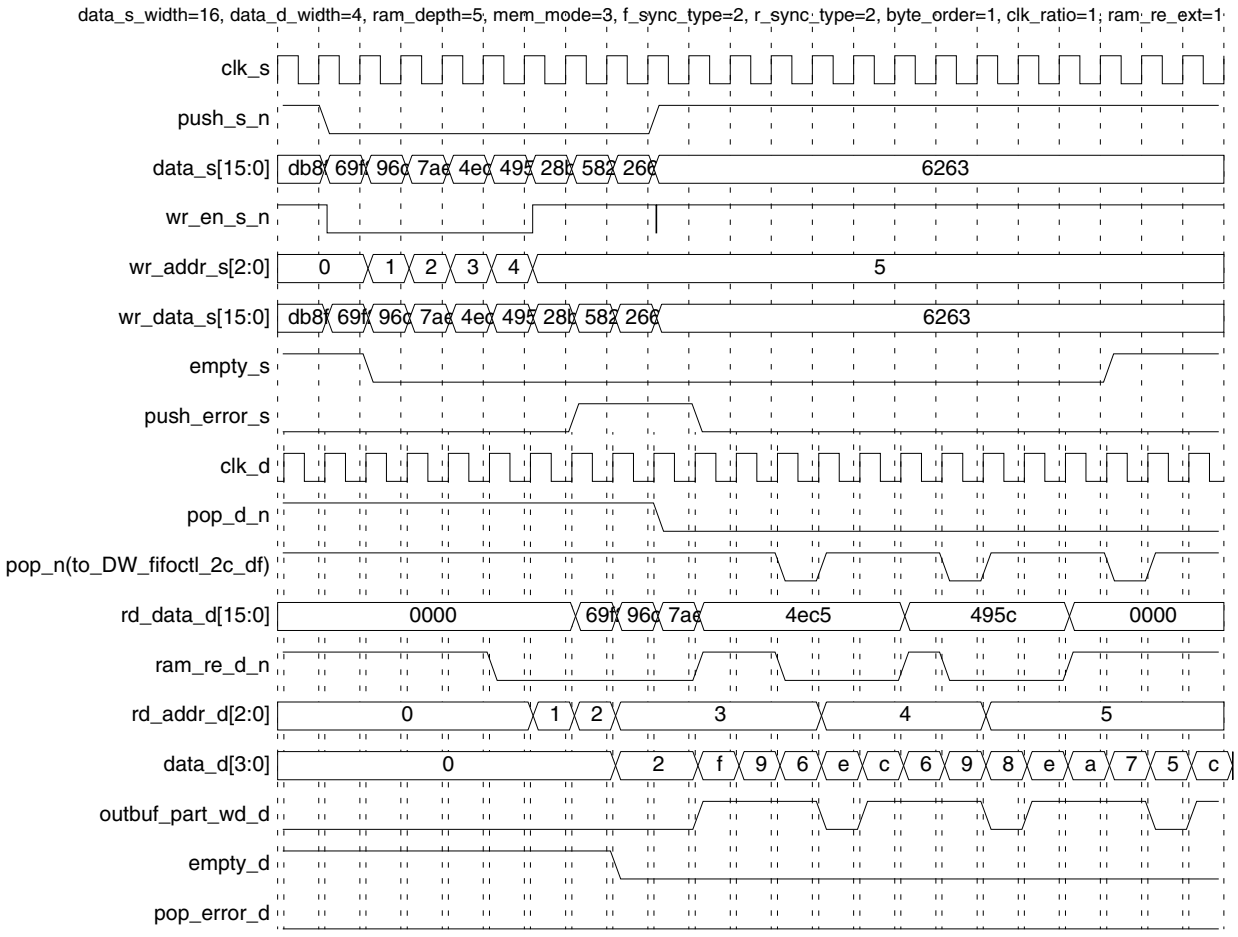
Figure 1-7 CASE 2: Push packets and pop alternately with 2 cycles active then 2 cycles inactive (*data_s_width* < *data_d_width*)



Based on the data width relationship, `pop_d_n` is connected directly to the underlying DW_fifoctl_2c_df. The pop activity begins as the FIFO is approximately half full and follows a progression of 2 cycles active and 2 cycles idle. This particular data flow, with a cache depth of 3, is the best-case behavior that favors the selection of the Register File (RF) cache architecture (*arch_type* of 1 and *mem_mode* of 3 or 7). Similarly, the pop request being active every other cycle for cache depth of 2 (*arch_type* of 1 and *mem_mode* of 1, 2, 5 or 6) would be the best-case data flow behavior geared to selecting the RF cache style. The disparity in power savings increases with larger *data_d_width* parameter values since a larger data path portion begins to dwarf the other portions of the component and, thus, the data flow plays a more prominent contributor to the overall dynamic power. That is, with larger *data_d_width* parameter values, the data flow through the component in the PL cache will be produce larger dynamic power numbers than the RF cache architecture. Thus, the advantage towards using the RF cache style is greater.

CASE 3: Figure 1-8 shows a case where *data_s_width* > *data_d_width* and `pop_d_n` is asserted continuously for multiple cycles.

Figure 1-8 CASE 3: Push packets and pops causing non-continuous pops to DW_fifoctl_2c_df (*data_s_width* > *data_d_width*)



However, since *data_s_width* > *data_d_width* an asymmetric output buffer, DW_asymdata_outbuf sits on the outputs of DW_fifoctl_2c_df. So even though the '`pop_d_n`' is asserted continuously, the `pop_n` (to_DW_fifoctl_2c_df) is the pop request going to the DW_fifoctl_2c_df. The `pop_n` is asserted every *data_s_width*/*data_d_width* (4) times in this case effectively following a similar behavior in CASE 2, but with longer durations of idle popping. So, based on this behavior for *data_s_width* > *data_d_width*, using the RF cache architecture would see power benefit over the PL cache architecture.

Synchronization Between Clock Domains

Each interface (source domain push and destination domain pop) operates synchronous to its own clock, `clk_s` and `clk_d`, respectively. Each interface is independent, containing its own state machine and flag logic. The pop interface has the primary read address counter and a synchronized copy of the write address counter. The push interface 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 (for example, `wr_addr_s` sampled by `clk_d`), sampling uncertainty can occur. Using Gray coding for 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. To transport Gray code addressing, the DesignWare component DW_gray_sync is instantiated for both directions.

f_sync_type and *r_sync_type*

The *f_sync_type* and *r_sync_type* parameters determine the number of register stages (1, 2, 3 or 4) used to synchronize the internal Gray code read pointer to `clk_s` (represented by *r_sync_type*) and internal Gray code write pointer to `clk_d` (represented by *f_sync_type*). A value of 1 indicates single-stage synchronization; a value of 2 indicates double-stage synchronization; a value of 3 indicates triple-stage synchronization; a value of 4 indicates quadruple-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 meta-stable synchronization events to stabilize and propagate to the pointer and flag registers.

Since timing slack and selection of register types is very difficult to control and meta-stability characteristics of registers are extremely difficult to ascertain, single-stage synchronization is not recommended, and thus, not available.

Double-stage synchronization is desirable when using relatively high clock rates. It allows an entire clock period for meta-stable 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 meta-stable events to settle at the first stage before being clocked into the second stage of the synchronizer. In the unlikely event that a meta-stable 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.

Quadruple-stage synchronization is desirable in extreme differences in clock rates between the two domains.

Empty to Not Empty Transitional Operation

When the FIFO is empty, `empty_s` and `empty_d` are active high. For *data_s_width* >= *data_d_width*, during the first push (`push_s_n` active low), the rising edge of `clk_s` writes the first word into the FIFO. The `empty_s` flag is driven low. For *data_s_width* < *data_d_width*, during the last sub-word of the first word pushed, the rising edge of `clk_s` writes the first word into the FIFO.

The *empty_d* flag does not go low until 1 to 3 cycles (of `clk_d`) after the new internal Gray code write pointer has been synchronized to `clk_d`. This could be as long as 2 to 7 cycles (depending on the values of the *f_sync_type* and *mem_mode* parameters). Refer to the timing diagrams for more information. The system design should allow for this latency in the depth budgeting of the FIFO design.

The `empty_d` flag is based on the validity of the data sitting at the head of the cache. It does not represent a count value of valid data entries in the RAM and cache pipeline. To identify precisely the number of valid data entries in the FIFO, refer to the `word_cnt_d` output port that gives the updated FIFO word count from the pop interface perspective.

`fifo_empty_s` reflects the contents of the RAM module and the pre-fetch cache whereas `empty_s` reflects the contents of RAM only.

Not Empty to Empty Transitional Operation

When the RAM module is almost empty, the `empty_s` is inactive (low), `almost_empty_d` is active (high), and the `empty_d` could be either state depending the cache state. When the `empty_d` goes inactive and during the pop (`pop_d_n` active (low) and assuming no pushes) that retrieves the last word of the FIFO, the next rising edge of `clk_d` causes the `empty_d` flag to be driven active (high).

The `empty_s` flag is not asserted (high) until one cycle (of `clk_s`) after the new internal Gray code read pointer has been synchronized to `clk_s` after the last entry of the RAM is loaded into the pre-fetch cache.

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

Note about Full Status

The concept of full with respect to each domain is different because of the cache that resides in the destination domain (pop interface). In the source domain (push interface), the concept of full is with respect to the RAM module contents. In the destination domain, the concept full is with respect to the RAM module and cache contents. In describing the dynamics of full in the following two sections, the starting point is always in perspective to the destination domain.

Full to Not Full Transitional Operation

When the FIFO is full (RAM and cache full), both `ram_full_s` and `full_d` are active high. During the first pop (`pop_d_n` active low), the rising edge of `clk_d` reads the first word out of the FIFO. The `full_d` flag is driven low.

The `ram_full_s` flag does not go low until one cycle (of `clk_s`) after the new internal Gray code read pointer has been synchronized to `clk_s`. This could be as long as 2 to 5 cycles (depending on the value of the *r_sync_type* parameter) from the time the read pointer in the destination is updated. Refer to "Timing Waveforms" on page 38 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 RAM is almost full (with respect to the destination domain) both `ram_full_s` and `full_d` are inactive (low) and `almost_full_s` is active (high). During the final push (`push_s_n` active low) and assuming no pops, the rising edge of `clk_s` writes the last word into the RAM. The `ram_full_s` flag is driven high.

The `full_d` flag is not asserted (high) until one cycle (of `clk_d`) after the new internal Gray code write pointer has been synchronized to `clk_d` (only after `ram_full_s` is asserted and the cache is full). This could be as long as 2 to 5 cycles (depending on the value of the *f_sync_type* parameter) from when the write pointer in the source domain got updated. Refer to the timing diagrams for more information.

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_s` and `pop_error_d` 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_s Output

The `push_error_s` output signal indicates that a push request was seen while the `ram_full_s` output was high (an overrun error) for cases with *data_s_width* >= *data_d_width*. For *data_s_width* < *data_d_width* configurations, the `push_error_s` output signal indicates that a push request was seen while both the `ram_full_s` and `ram_full_s` outputs were high. When an overrun condition occurs, the write address pointer (`wr_addr_s`) does not advance, and the RAM write enable (`wr_en_s_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_d Output

The `pop_error_d` output signal indicates that a pop request was seen while the `empty_d` output signal was active high (an underrun error). When an underrun condition occurs, the read address pointer (`rd_addr_d`) does not decrement, as there is no data in the FIFO to retrieve.

The FIFO timing is such that the logic controlling the `pop_d_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 `empty_d` 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 four clock cycles for proper synchronization between clock domains. Also, due to the cache present in the destination domain (pop interface) fullness will be based on RAM and cache contents whereas the source domain (push interface) only considers the RAM contents for its fullness. The same is true for the state of emptiness with one exception regarding `empty_d`.

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 and prefetch cache count to `clk_s`. 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_d`.

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.

empty_s

The `empty_s` output, active high, is synchronous to the `clk_s` input. `empty_s` indicates to the push interface that the RAM module is empty. If *data_s_width* >= *data_d_width*, during the first push, the rising edge of `clk_s` causes the first word to be written into the RAM and `empty_s` is driven low. If *data_s_width* < *data_d_width*, during the last sub-word of the first complete word push, the rising edge of `clk_s` causes the first word to be written into the RAM and `empty_s` is driven low.

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

Property of empty_s

If `empty_s` is active (high) then the RAM module is truly empty. This property does not apply to `empty_d`.

Attention

When using push status outputs to make decisions on writing into the FIFO, use `empty_s` and `word_cnt_s` instead `fifo_empty_s` and `fifo_word_cnt_s`. The `empty_s` and `word_cnt_s` signals provide accurate information about how much space is available for writing into the RAM of the FIFO. For detailed informatin about `fifo_empty_s` and `fifo_word_cnt_s`, see "Behavior of fifo_empty_s and fifo_word_cnt_s" on page 34.

almost_empty_s

The `almost_empty_s` output, active high, is synchronous to the `clk_s` input. The `almost_empty_s` output indicates to the push interface that the RAM module is almost empty when there are no more than `ae_level_s` (input port) words currently in the RAM module to be popped as perceived at the push interface.

The ae_level_s input port defines the almost empty threshold of the push interface independent of that of the pop interface. The almost_empty_s output is useful when it is desirable to push data into the RAM module in bursts (without allowing the RAM module to become empty).

Property of almost_empty_s

If almost_empty_s is active high then the RAM module has at least (*ram_depth* - ae_level_s) available locations. Therefore such status indicates that the push interface can safely and unconditionally push (*ram_depth* - ae_level_s) words into the RAM module. This property guarantees that such a 'blind push' operation will not overrun the RAM module.

half_full_s

The half_full_s output, active high, is synchronous to the clk_s input, and indicates to the push interface that the RAM module has at least half of its memory locations occupied as perceived by the push interface.

Property of half_full_s

If half_full_s is inactive (low) then the RAM module has at least half of its locations available. Thus such status indicates that the push interface can safely and unconditionally push (INT(*ram_depth*/2)+1) words into the RAM module. This property guarantees that such a 'blind push' operation will not overrun the RAM module.

almost_full_s

The almost_full_s output, active high, is synchronous to the clk_s input. almost_full_s indicates to the push interface that the RAM module is almost full when there are no more than af_level_s empty locations in the RAM module as perceived by the push interface.

The af_level_s input port defines the almost full threshold of the push interface independent of the pop interface. The almost_full_s output is useful when more than one cycle of advance warning is needed to stop the flow of data into the RAM module before it becomes full (to avoid a RAM module overrun).

Property of almost_full_s

If almost_full_s is inactive (low) then the RAM module has at least (af_level_s+1) available locations. Thus such status indicates that the push interface can safely and unconditionally push (af_level_s+1) words into the RAM module. This property guarantees that such a 'blind push' operation will not overrun the RAM module.

inbuf_part_wd_s (only when *data_s_width* < *data_d_width*)

The inbuf_part_wd_s output, active high, is synchronous to the clk_s input. The inbuf_part_wd_s output indicates to the push interface that at least one sub-word has been push in the input buffer.

The action of a complete word being written to the RAM module causes the inbuf_part_wd_s output to go low on the next rising edge of clk_s.

inbuf_full_s (only when *data_s_width* < *data_d_width*)

The `inbuf_full_s output`, active high, is synchronous to the `clk_s` input. The `inbuf_full_s` output indicates to the push interface that all sub-word entries in the input buffer have been pushed with valid data. The number of sub-words stored in the input buffer is (*data_d_width*/*data_s_width*)-1, call it "N". During the final Nth sub-word, the rising edge of `clk_s` causes the last word to be pushed, and `inbuf_full_s` is asserted.

The action of a complete word (Nth+1 sub-word) being written to the RAM module causes the `inbuf_full_s` output to go low on the next rising edge of `clk_s`.

ram_full_s

The `ram_full_s` output, active high, is synchronous to the `clk_s` input. The `ram_full_s` output indicates to the push interface that the RAM module is full. If *data_s_width* >= *data_d_width*, during the final push, the rising edge of `clk_s` causes the last word to be written and `ram_full_s` is asserted. If *data_s_width* < *data_d_width*, during the final sub-word of the last complete word push, the rising edge of `clk_s` causes the last word to be written and `ram_full_s` is asserted

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

empty_d

The `empty_d` output, active high, is synchronous to the `clk_d` input. `empty_d` indicates to the pop interface that the cache does not contain relevant data. It does not mean, per se, that the RAM module cache doesn't contain valid data. The action of the last word being popped from a nearly empty FIFO is controlled by the pop interface. Thus, the `empty_d` output is asserted at the rising edge of `clk_d` that causes the last word to be popped from the FIFO.

The last word in this context refers to when the RAM module is empty and the only relevant data word is sitting in the cache.

The action of pushing the first word into an empty FIFO is controlled by the push interface. That means `empty_d` goes low only after the new internal Gray code write pointer from the push interface is synchronized to `clk_d`, data is read from the RAM module, and then placed into the cache.

almost_empty_d

The `almost_empty_d` output (active high) is synchronous to the `clk_d` input. `almost_empty_d` indicates to the pop interface that the FIFO is almost empty when there are no more than `ae_level_d` (input port) words currently in the FIFO to be popped.

The `ae_level_d` input port defines the almost empty threshold of the pop interface independent of the push interface. The `almost_empty_d` 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 almost_empty_d

If `almost_empty_d` is inactive (low), there are at least (`ae_level_d` +1) words in the FIFO. Therefore such status indicates that the pop interface can safely and unconditionally pop (`ae_level_d` +1) words out of the FIFO. This property guarantees that such a 'blind pop' operation will not underrun the FIFO.

half_full_d

The `half_full_d` output (active high) is synchronous to the `clk_d` input. `half_full_d` indicates to the pop interface that the FIFO has at least half of its memory locations occupied.

Property of half_full_d

If `half_full_d` is active (high), then at least half of the words in the FIFO are occupied. Therefore such status indicates that the pop interface can safely and unconditionally pop INT((*eff_depth*+1)/2) words out of the FIFO, where *eff_depth* is defined in Table 1-5 on page 7. This property guarantees that such a 'blind pop' operation will not underrun the FIFO.

almost_full_d

The `almost_full_d` output (active high) is synchronous to the `clk_d` input. The `almost_full_d` output indicates to the pop interface that the FIFO is almost full when there are no more than `af_level_d` empty locations in the FIFO as perceived by the pop interface.

The `af_level_d` input port defines the almost full threshold of the pop interface independent of that of the push interface. The `almost_full_d` 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 almost_full_d

If `almost_full_d` is active (high) then there are at least (*eff_depth*- `af_level_d`) words in the FIFO, where *eff_depth* is defined in Table 1-5 on page 7. Therefore, such status indicates that the pop interface can safely and unconditionally pop (*eff_depth*- `af_level_d`) words out of the FIFO. This property guarantees that such a 'blind pop' operation will not underrun the FIFO.

outbuf_part_wd_d (only when *data_s_width* > *data_d_width*)

The `outbuf_part_wd_d` output (active high) is synchronous to the `clk_d` input. `outbuf_part_wd_d` indicates to the pop interface that the retrieval of sub-words (of a complete word) from RAM are in process. The action of popping the sub-words out from RAM is controlled by the pop interface. The `outbuf_part_wd_d` output goes low at the rising edge of `clk_d` following the popping of the last sub-word.

full_d

The `full_d` output, active high, is synchronous to the `clk_d` input. `full_d` indicates to the pop interface that the RAM and pre-fetch cache are full. The action of popping the first word out of a full FIFO is controlled by the pop interface. Thus, the `full_d` output goes low at the rising edge of `clk_d` that causes the first word to be popped.

The action of the last word being pushed into a nearly full RAM is controlled by the push interface. The action of loading the pre-fetch cache with RAM contents is controlled by the pop interface. This means the

`full_d` output is asserted only after the new write pointer from the pop interface is synchronized to `clk_d` and processed by the status flag state logic.

Property of full_d

If `full_d` is active (high), then the FIFO is truly full. This property does not apply to `full_s`.

Behavior of fifo_empty_s and fifo_word_cnt_s

The `fifo_empty_s` and `fifo_word_cnt_s` status outputs are meant to indicate the *general* state of the FIFO. For example, when the FIFO is nearing empty, there can be a one `clk_s` cycle window when the `fifo_empty_s` indicates “empty” when there is still one valid word stored in the FIFO. Also, the `fifo_word_cnt_s` output value can be off by a ±1 words for one `clk_s` cycle before achieving steady state. That is, when at the "close to empty state", the `fifo_word_cnt_s` could report a '0' or '2' when there is actually '1' valid word in the FIFO. The reason behind this stems from there being two pieces of information from the pop domain that are synchronized and merged into the push domain, the "pop read pointer" and "pop pre-fetch cache count" that reside (but are not shown) in the DW_fifoctl_2d_df block in Figure 1-2 on page 12.

The "read_pointer" and "pop pre-fetch cache count" information is then sent to the push domain (`clk_s`) in parallel for synchronization. Although, together, they represent the correct count (one word in the FIFO) in the pop domain, their values may incur sampling issues at the push domain synchronizers due to logic delay and meta-stability, and the sampling issues can cause momentary instability in the push domain. This momentary instability can result in inaccurate values on `fifo_word_cnt_s` and `fifo_empty_s` for one `clk_s` cycle. Steady state will occur, provided pushing and popping activities are suspended long enough for all the synchronization to stabilize.

Below are two sets of waveforms that show the possible inaccurate behavior of `fifo_word_cnt_s` and `fifo_empty_s` for a single word push and with no popping.

The following configuration was used for Figure 1-9 on page 35 and Figure 1-10 on page 36:

data_s_width = 32

data_d_width = 32

ram_depth = 16

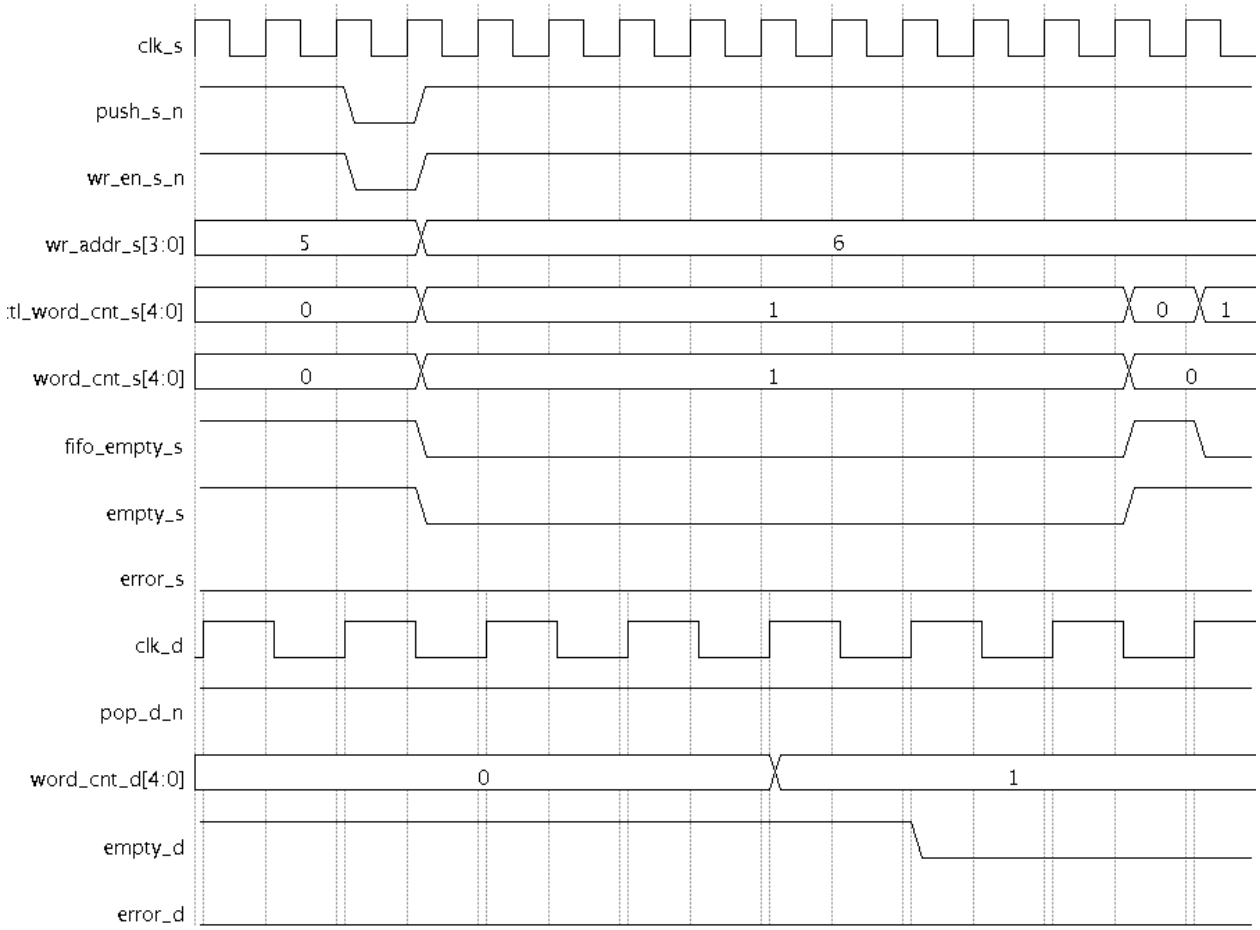
f_sync_type = 2

r_sync_type = 2

mem_mode = 0

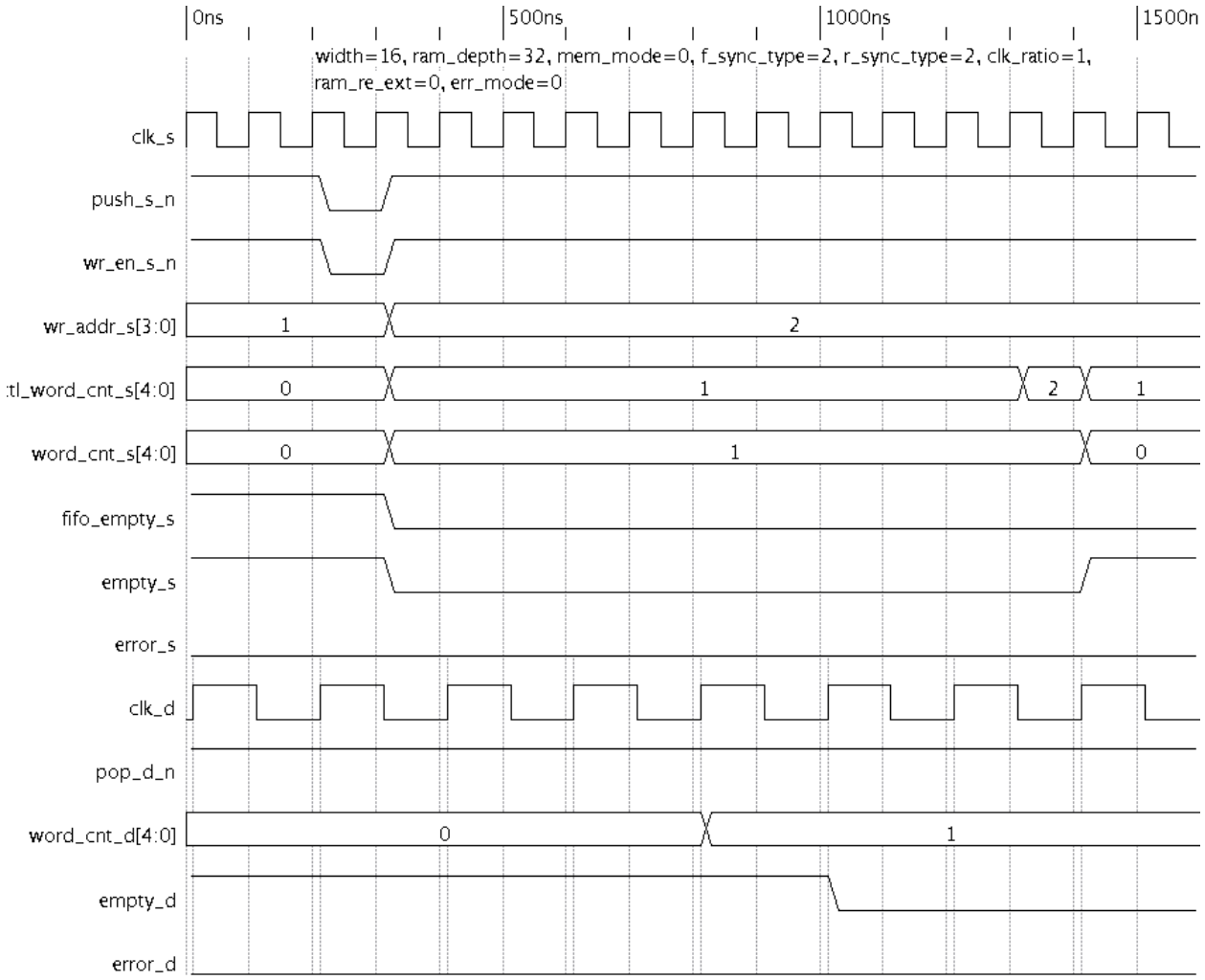
In Figure 1-9, the waveforms show a push request being made with `push_s_n` going to '0' for one `clk_s` cycle. Notice that no pop request is made throughout (`pop_n_d` stays at '1'). So, effectively, the FIFO has one valid word in it. As the pre-fetch to cache occurs (reading from the RAM), the read pointer and cache count from the pop domain are synchronized back to the push domain. In this case, `fifo_word_cnt_s` goes from '1' to '0' and then back to '1'. The transition to '0' is the instability, and is due to a synchronization sampling error causing a disparity between the read pointer and cache count in the push domain. This behavior is expected as `fifo_word_cnt_s` stabilizes at '1' after one `clk_s` cycle. Notice also that `fifo_empty_s` goes to '1' for a `clk_s` cycle coinciding with `fifo_word_cnt_s` being '0'.

Figure 1-9 Push Request Showing Instability On fifo_word_cnt_s (Example 1)



In Figure 1-10, the waveforms show a push request being made with `push_s_n` going to '0' for one `clk_s` cycle. Notice that no pop request is made throughout (`pop_n_d` stays at '1'). So, effectively, the FIFO has one valid word in it. As the pre-fetch to cache occurs (reading from the RAM), the read pointer and cache count from the pop domain are synchronized back to the push domain. In this case, `fifo_word_cnt_s` goes from '1' to '2' and then back to '1'. The transition to '2' is the instability, and is due to a synchronization sampling error causing a disparity between the read pointer and cache count in the push domain. This behavior is expected as `fifo_word_cnt_s` stabilizes at '1' after one `clk_s` cycle.

Figure 1-10 Push Request Showing Instability On fifo_word_cnt_s (Example 2)



Reset Considerations

System Resets (synchronous and asynchronous)

The system resets, `rst_s_n` and `init_s_n` for the source (push) domain and `rst_d_n` and `init_d_n` for the destination (pop) domain, work independently between the two domains. This inherently could cause data corruption and false status reporting when the activation of these resets is not coordinated between the two domains at the system level.

The following are some guidelines on how the system resets between the two domains should be coordinated.

For system reset conditions, if the assertion of the resets occurs in one clock domain, the other domain must also assert its reset so that both domains are eventually in reset. That is, at some time in the system reset sequence, both domains must be in the active reset condition simultaneously. The length of the system reset signal(s) assertion must be a minimum of 4 clock cycles of the slowest clock between the two domains. Both clock domain system reset signals, when asserted, should overlap for a minimum of *f_sync_type*+1 or *r_sync_type*+1 cycles (which ever is larger) of the slowest of the two domain clocks.

Besides satisfying simultaneous assertion of each domain system reset signals for a minimum number of cycles, the timing of the assertion between these signals needs some consideration. To prevent erroneous `clr_sync_s` and `clr_sync_d` pulses from occurring when system resets are asserted, it is recommended that:

- If the source domain system reset is asserted first, then the destination domain should assert its reset within *f_sync_type* +1 `clk_d` cycles from the time the assertion of the source domain reset occurred

OR

- If the destination domain system reset is asserted first, then the source domain should assert its within *r_sync_type*+1 `clk_s` cycles from the time the assertion of the destination domain reset occurred.

If both domains can tolerate a false `clr_sync_s` or `clr_sync_d` (whichever the case) during system reset conditions, then this recommendation can be ignored as long as both clock domains eventually have overlapping active reset conditions.

There are no restrictions on when to release the reset condition on either side. However, to be completely safe, it is recommended to release the source clock domain's reset last.

See Figure 1-19 on page 49 and Figure 1-20 on page 50 for examples of asynchronous and synchronous system reset assertions, respectively.

Clearing FIFO Controller (synchronous)

The DW_asymfifoctl_2c_df contains one coordinated clearing signal from each domain called `clr_s` and `clr_d`. A minimum of a single clock cycle pulse on either one of these clearing signals initiates a synchronized clearing sequence to each domain for resetting of its sequential devices. This clearing sequence is orchestrated to ensure that the destination domain interface is completely cleared and ready for more data before the source domain is permitted to begin sending.

Figure 1-16 on page 45 and Figure 1-17 on page 47 show the clearing sequence for when `clr_s` and `clr_d`, respectively, are asserted. Additionally, Figure 1-18 on page 48 shows another `clr_s` initiated clearing sequence with a `clr_s` that is asserted for longer than a single `clk_s` cycle.

It is imperative for data transfer integrity to cease pushing any packets after asserted `clr_s` (and while waiting for a subsequent `clr_cmplt_s` pulse) and/or when observing an active `clr_in_prog_s`. From the destination domain, reading data packets after `clr_d` is asserted and/or observing an active `clr_in_prog_d` would result in corrupt data packet retrieval. Bottom-line, it is very important to halt pushing and popping during the clearing sequence and only start pushing new data after the `clr_cmplt_s` pulse is observed. Also, during the clearing sequence from the `clr_s` is asserted to `clr_cmplt_s` assertion for the `clr_s` initiated case OR from the time `clr_d` is asserted to `clr_cmplt_s` assertion, it is important to realize that the values of all off status flags and word counts in both domains will not be reliable. Only after the completion of the coordinated clearing sequence are the status flags and word counts accurate.

There is no restriction on how often or how long `clr_s` and `clr_d` can be asserted. The clearing operation is maintained if in progress and subsequent `clr_s` and/or `clr_d` initiations are maid. Once the final assertion of `clr_s` and/or `clr_d` is made, the sustained clearing sequence eventually comes to completion and all in-progress flags de-assert.

Test

The synthesis parameter, *tst_mode*, controls the insertion of lock-up latches at the points where signals cross between the clock domains, `clk_s` and `clk_d`. Lock-up latches are used to ensure proper cross-domain operation during the capture phase of scan testing in devices with multiple clocks. When *tst_mode*=1, lock-up latches will be inserted during synthesis and will be controlled by the input, test.

With *tst_mode*=1, the input, `test`, controls the bypass of the latches for normal operation where `test`=0 bypasses latches and `test`=1 includes latches. In order to assist DFT compiler in the use of the lock-up latches, use the “set_test_hold 1” tst_mode command before using the “insert_scan” command.

When *tst_mode*=0 (which is its default value when not set in the design) no lock-up latches are inserted and the test input is not connected.

The insertion of lock-up latches requires the availability of an active low enable latch cell. If the target library does not have such a latch or if latches are not allowed (using “dont_use” commands for instance), synthesis of this module with *tst_mode*=1 will fail.

Timing Waveforms

The following section contains waveforms diagrams and descriptions for the DW_asymfifoctl_2c_df component.

Figure 1-11 shows an 4-deep RAM configured with *mem_mode* = 1 with *data_s_width* < *data_d_width*. The mem_mode setting implies that there is a 2-deep cache in the destination domain and, hence, defines a 6 deep FIFO. In this example `clk_s` and `clk_d` are asynchronous but the same frequency.

Pushing occurs (while no popping activity) until the RAM and input buffer (DW_asymdata_inbuf) are full. Since *data_s_width* < *data_d_width*, fullness of the push domain is made up `ram_full_s` and `inbuf_full_s`. When both are '1', the push interface is full. Under that condition in the waveform, one more push operation is issued causing `push_error_d` to go to '1'.

Figure 1-11 Push Until Full and Push Error (data_s_width < data_d_width)

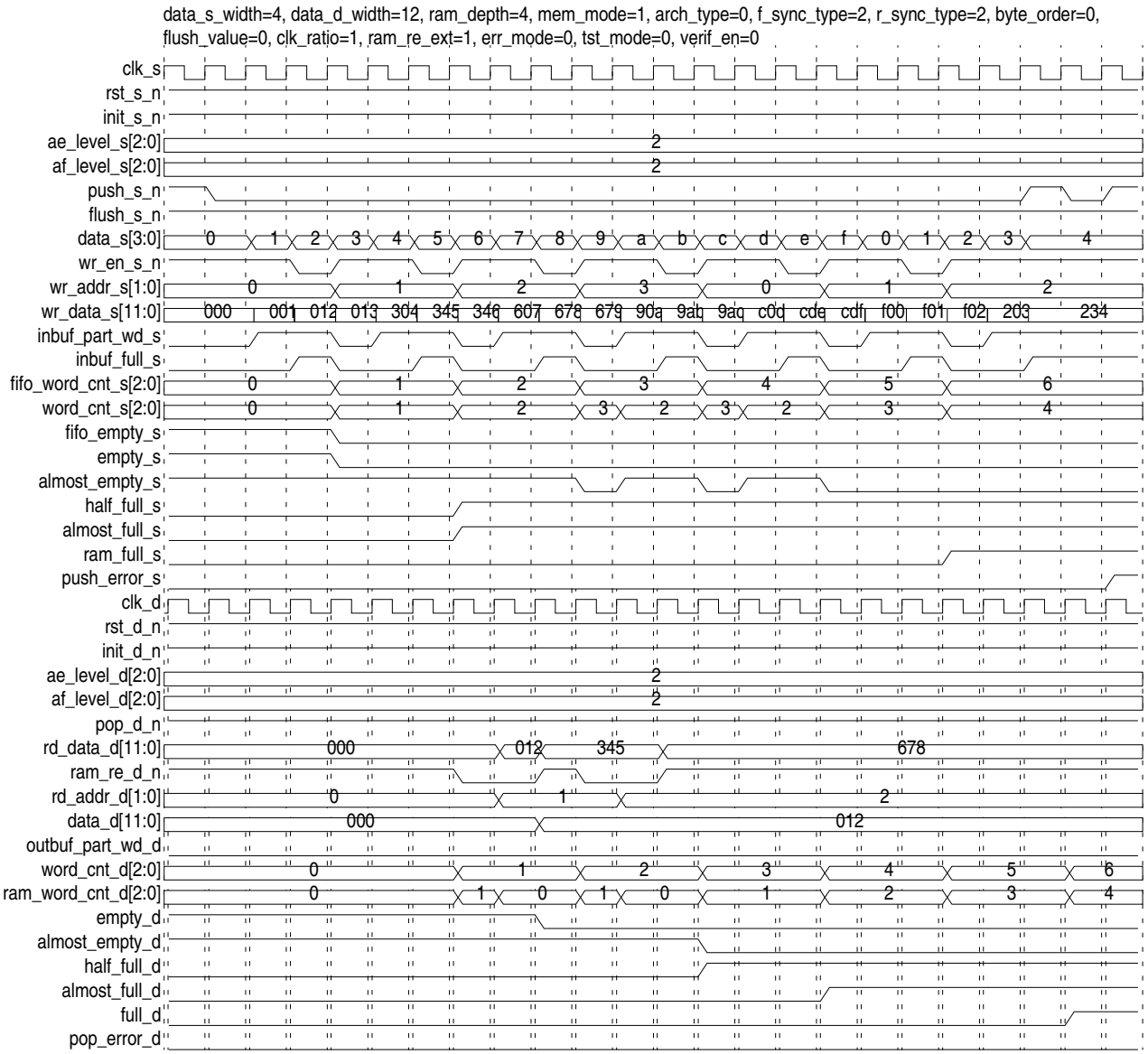


Figure 1-12 shows popping activity from the FIFO full condition to the FIFO empty condition for *data_s_width* < *data_d_width*. As long as `empty_d` is a 0, `pop_d_n` is asserted (low).

When the FIFO is empty (`empty_d` is a 1), another assertion of `pop_d_n` is made to show the resulting `pop_error_d` behavior.

Note: This timing waveform is a continuation of the one found in Figure 1-11 where the push operation is performed. Figure 1-12 shows how the 'data_d' output is organized based on the *byte_order* applied to the input buffer in the push domain. Configuration: same as Figure 1-11.

Figure 1-12 Pop Until Empty and Pop Error (*data_s_width* < *data_d_width*)

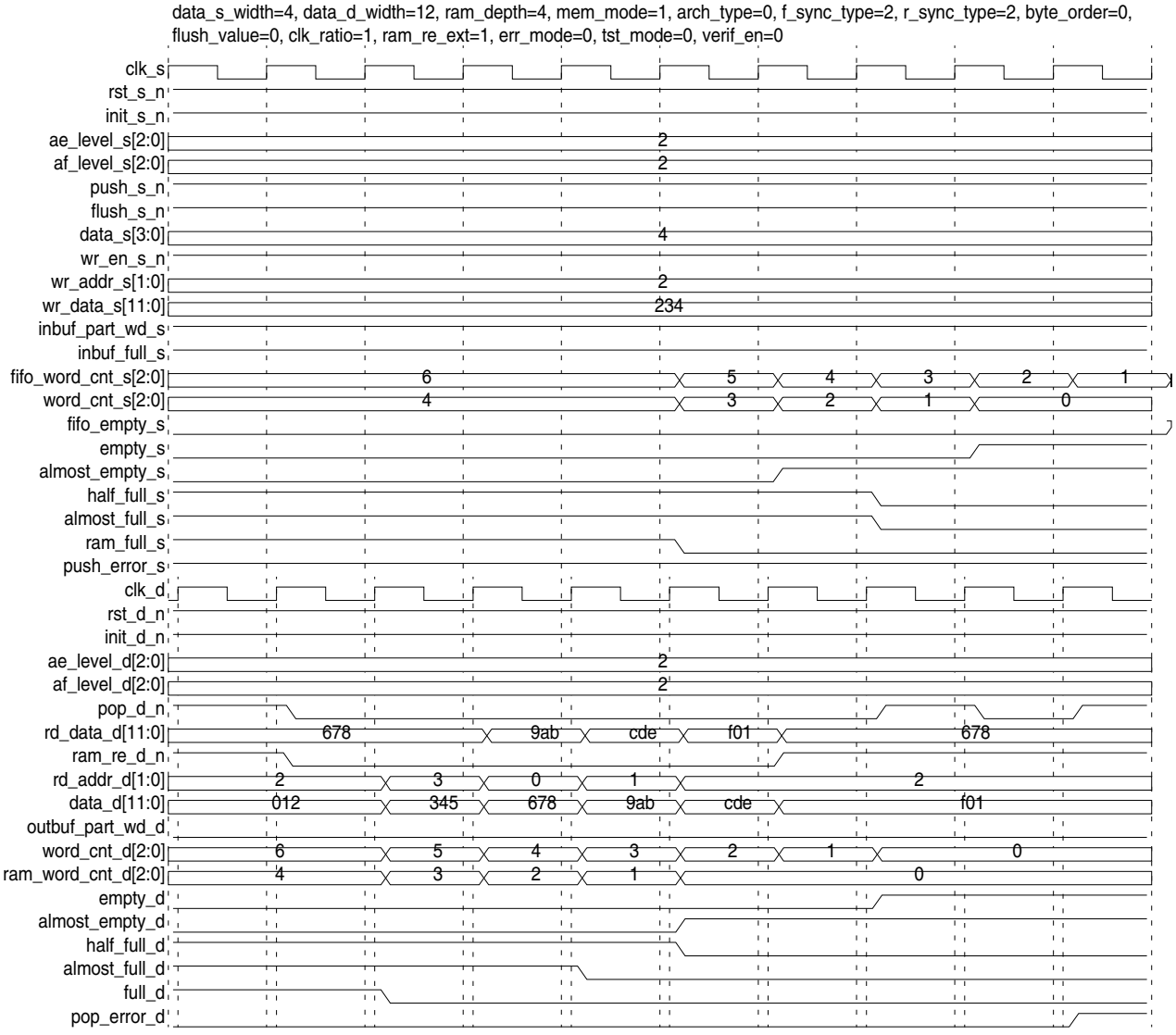


Figure 1-13 shows the DW_asymfifoctl_2c_df configured with *ram_depth* of 5, *mem_mode* as 4, and *data_s_width* > *data_d_width*. The *mem_mode* setting implies that there is a 1-deep cache in the destination domain and, hence, defines a 6-deep FIFO. In this example `clk_s` is slower than `clk_d` are they asynchronous to each other.

Pushing occurs (while no popping activity) until the RAM is full. Since *data_s_width* > *data_d_width*, there is no input buffer (DW_asymdata_inbuf) present. Therefore, fullness in the push domain is based on `ram_full_s`. Once the RAM is full (`ram_full_s` being 1), another push operation is issued causing `push_error_d` to go to 1. Note that since *err_mode* is 1, the `push_error_d` does not stay at 1.

Figure 1-13 Push Until Full and Push Error (*data_s_width* > *data_d_width*)

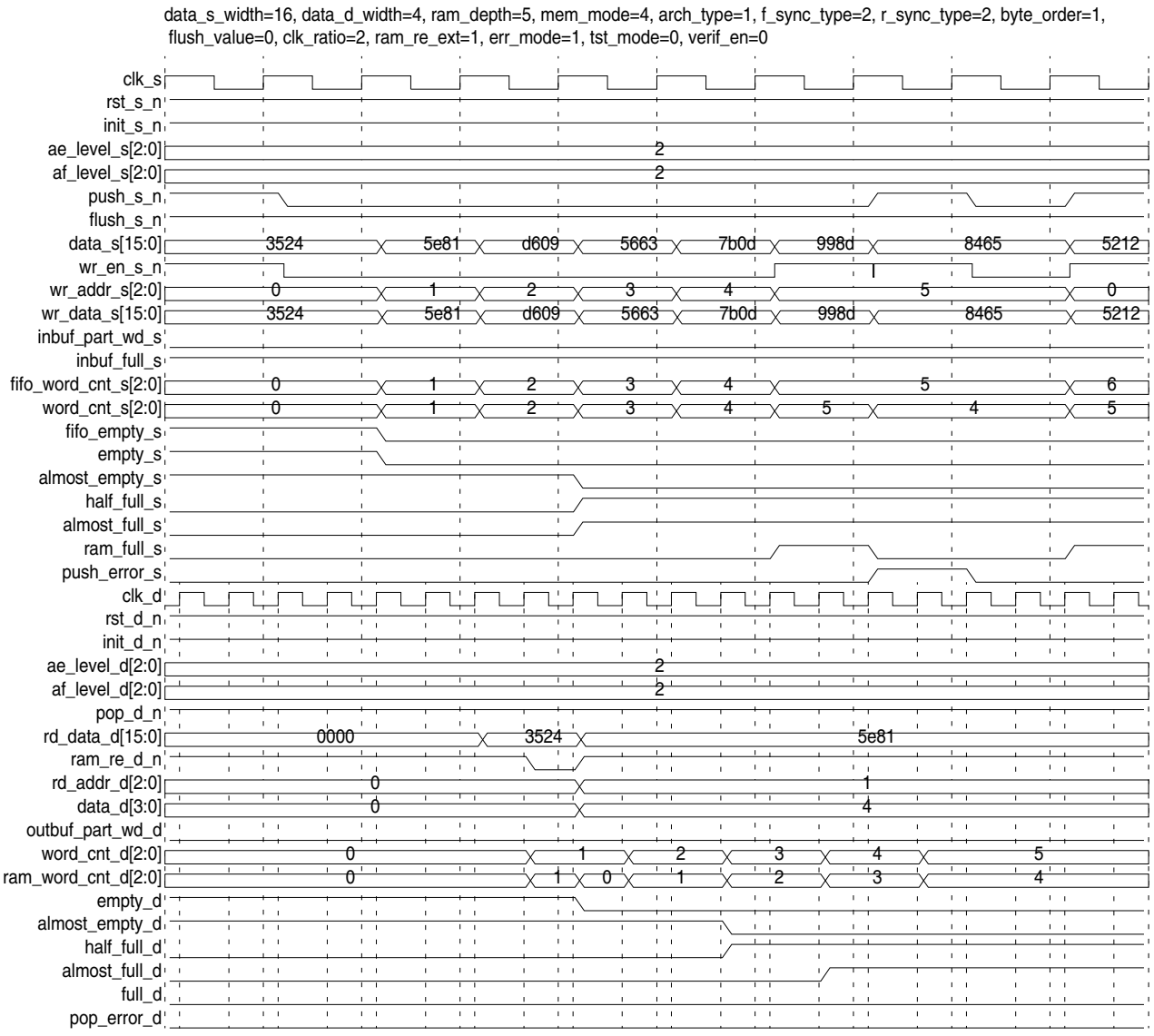


Figure 1-14 shows popping activity from the FIFO full condition to the FIFO empty condition for *data_s_width* > *data_d_width*. As long as `empty_d` is a 0, `pop_d_n` is asserted (low).

When the FIFO is empty (`empty_d` is a 1), another assertion of `pop_d_n` is made to show the resulting `pop_error_d` behavior. The `pop_error_d`, in this example, asserts dynamically on an occurrence of an error because *error_mode* is 1.

Note: This timing waveform is a continuation of the one found in Figure 1-13 where the push operation is performed. In Figure 1-14, it can be seen how the `data_d` output is organized based on the *byte_order* applied to the input buffer in the push domain. Configuration: same as Figure 1-13.

Figure 1-14 Pop Until Empty and Pop Error (data_s_width > data_d_width)

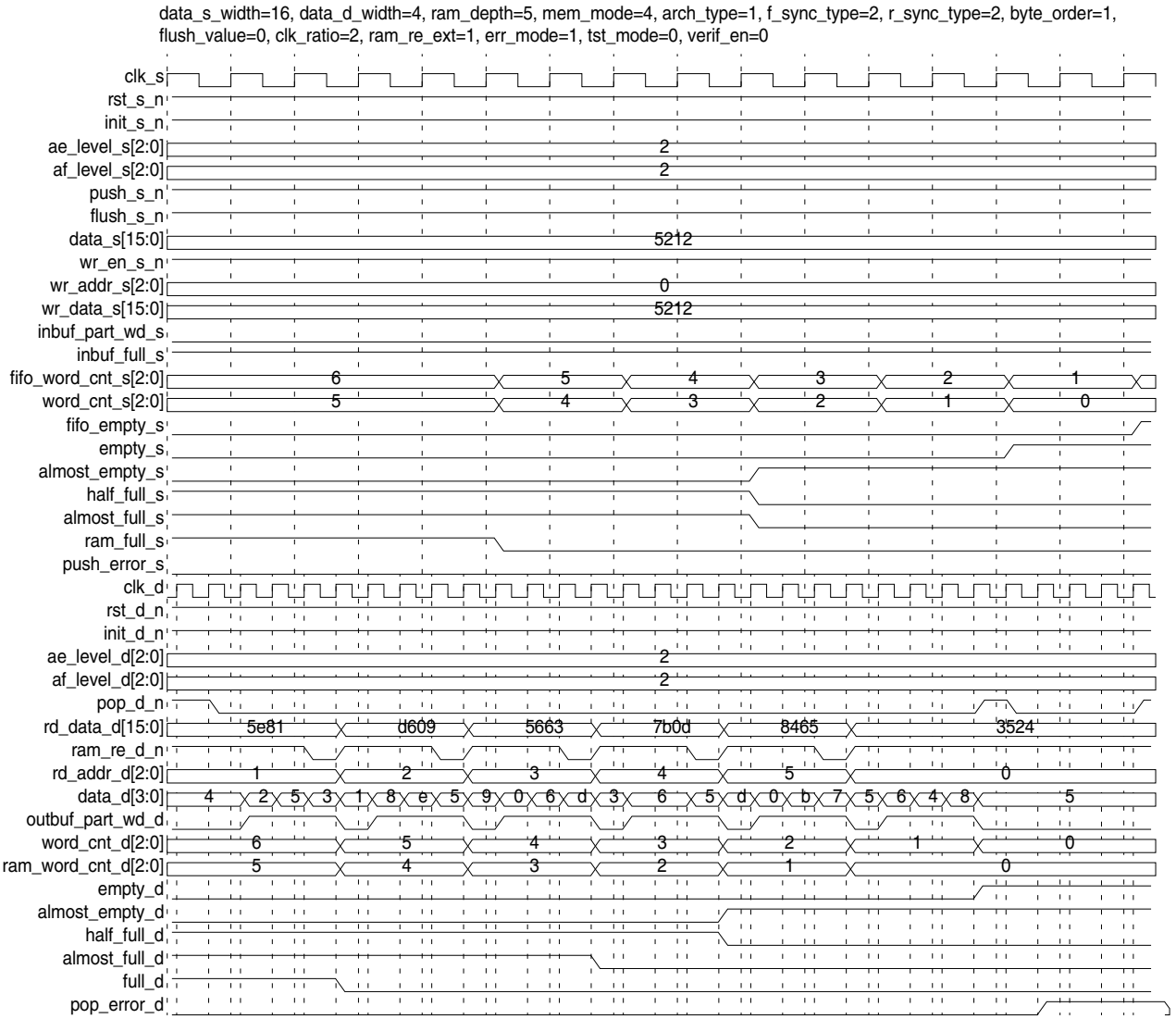


Figure 1-15 shows the behavior of DW_asymfifoctl_2c_df when the `flush_s_n` input is asserted. `flush_s_n` only has meaning when *data_s_width* < *data_d_width*. Here, the integer ratio of *data_s_width* to *data_d_width* is 3 (*data_d_width* / *data_s_width*).

The first `push_s_n` assertion captures `data_s`[3:0] of '0x5'. That operation causes the `inbuf_part_wd_s` to go to '1'. Three `clk_s` cycles later, `flush_s_n` is asserted and sampled. This causes the first sub-word if '0x5' to be loaded into the RAM (see `wr_en_s_n` go to '0' coincident with `flush_s_n` assertion) along with two other sub-words of '0x0'. The flushed partition of the word (two sub-words) are each '0x0' because *flush_value* is set to '0'. If *flush_value* was set to '1', the word written into RAM would have been '0x5ff'.

Figure 1-15 Flushing Operation (only when *data_s_width* < *data_d_width*)

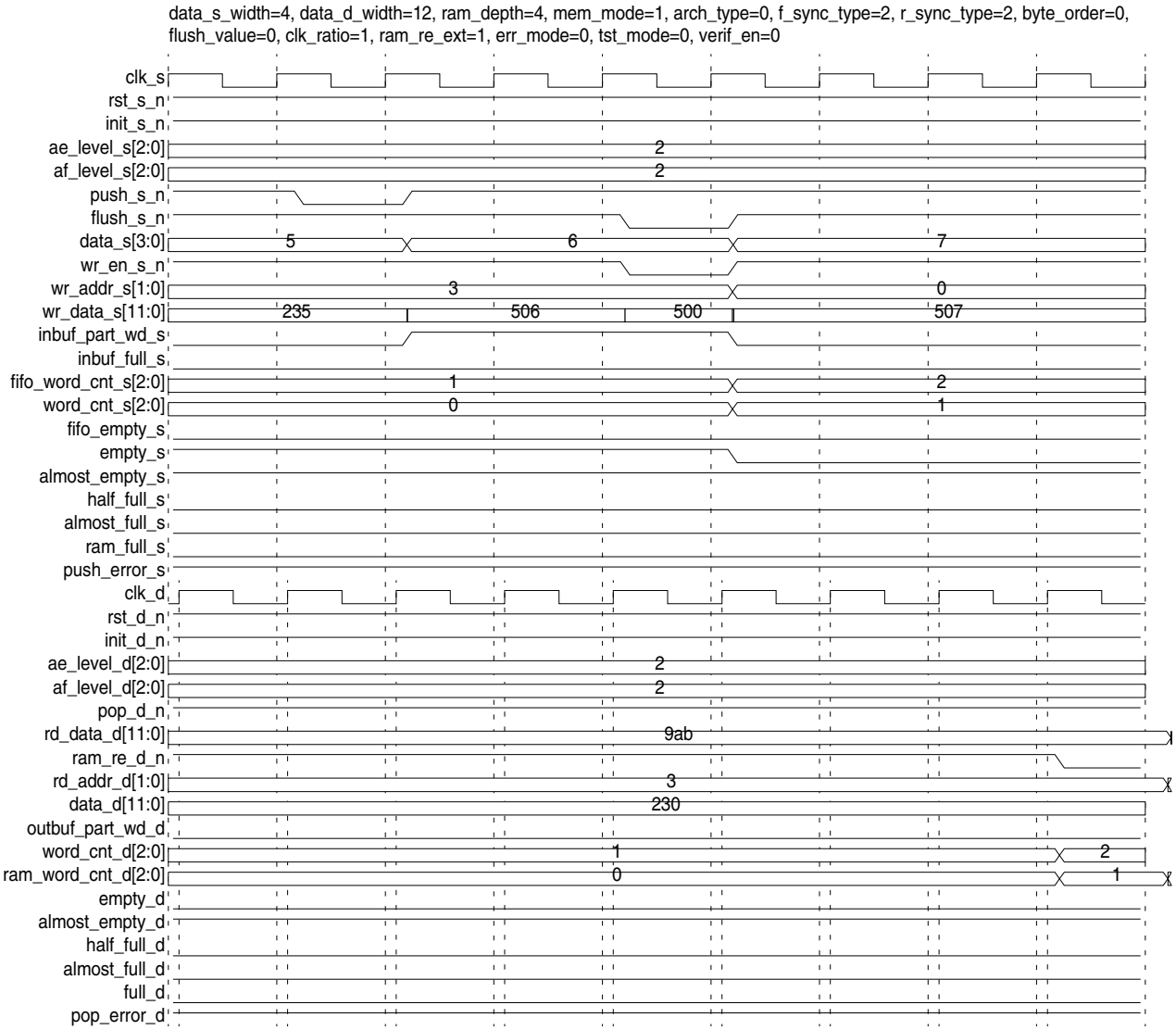


Figure 1-16 shows the initiation of the coordinated clearing sequence from the source domain via assertion of clr_s. In this case clr_s is a single clk_s cycle, but the length of clr_s assertions are not restricted. The clearing-related signals are grouped at the bottom of the timing diagram.

When clr_s is asserted it gets synchronized at the destination domain (based on *f_sync_type*) activates clr_in_prog_d. clr_in_prog_d is useful for destination sequential logic in that it can be used to 'initialize' circuits knowing that source domain is not scheduled to push any data packets until the clearing sequence is complete. The event that produces the clr_in_prog_d assertion is then fed back to the source domain where it is synchronized (based on *r_sync_type*) to generate the clr_sync_s pulse. On the heals of the clr_sync_s pulse, the clr_in_prog_s signals get activated. Similar to the clr_in_prog_d signal, clr_in_prog_s and/or clr_sync_s can be used to initialize source domain sequential logic since it is implied that no destination domain popping will occur until the clearing sequence is completed.

The clr_sync_s event is then sent back for synchronization in the destination domain to de-activate clr_in_prog_d and generate the clr_cmplt_d indicating to the destination domain that the source domain has been cleared and it can be in the waiting state for popping.

Now that the destination domain perceives that its clear sequence is done, that event is sent back to the source domain for synchronization which, in turn, de-activates clr_in_prog_s and produces a clr_cmplt_s pulse. The de-activation of clr_in_prog_s and subsequent clr_cmplt_s pulse indicates to the source domain that the destination domain logic has been cleared and all is ready for more pushing of data.

During the clearing process, there will be occasions when the status flags and word counts could report incorrectly. This situation occurs when clr_in_prog_d gets activated. Upon this event, the internal pointers (used to calculate word counts and status flags) could go between initialized and non-initialized states. This can be seen with fifo_word_cnt_s[2:0] going from 5 -> 0 -> 2 during the “in progress” portion of the clear sequence. This type of counter changes could produce state changes in output status signals fifo_empty_s, empty_s, almost_empty_s, half_full_s, and almost_full_s. But note by the time the clearing sequence completes in the source domain as indicated by clr_cmplt_s, all outputs and counters are in their initialized states. The same holds true in the destination domain for its counters, status outputs, and data with respect to an asserted clr_cmplt_d.

Therefore, having the word count and status flags report incorrectly during the clearing process is benign based on the knowledge that once the clr_s is initiated the source domain knows that the clearing sequence has begun and no credence should be put on the system state until both domains are notified of clearing completion based on clr_cmplt_s and clr_cmplt_d assertions.

Configuration: same as Figure 1-15.

Figure 1-16 Single-cycle clr_s Initiated Clearing Sequence

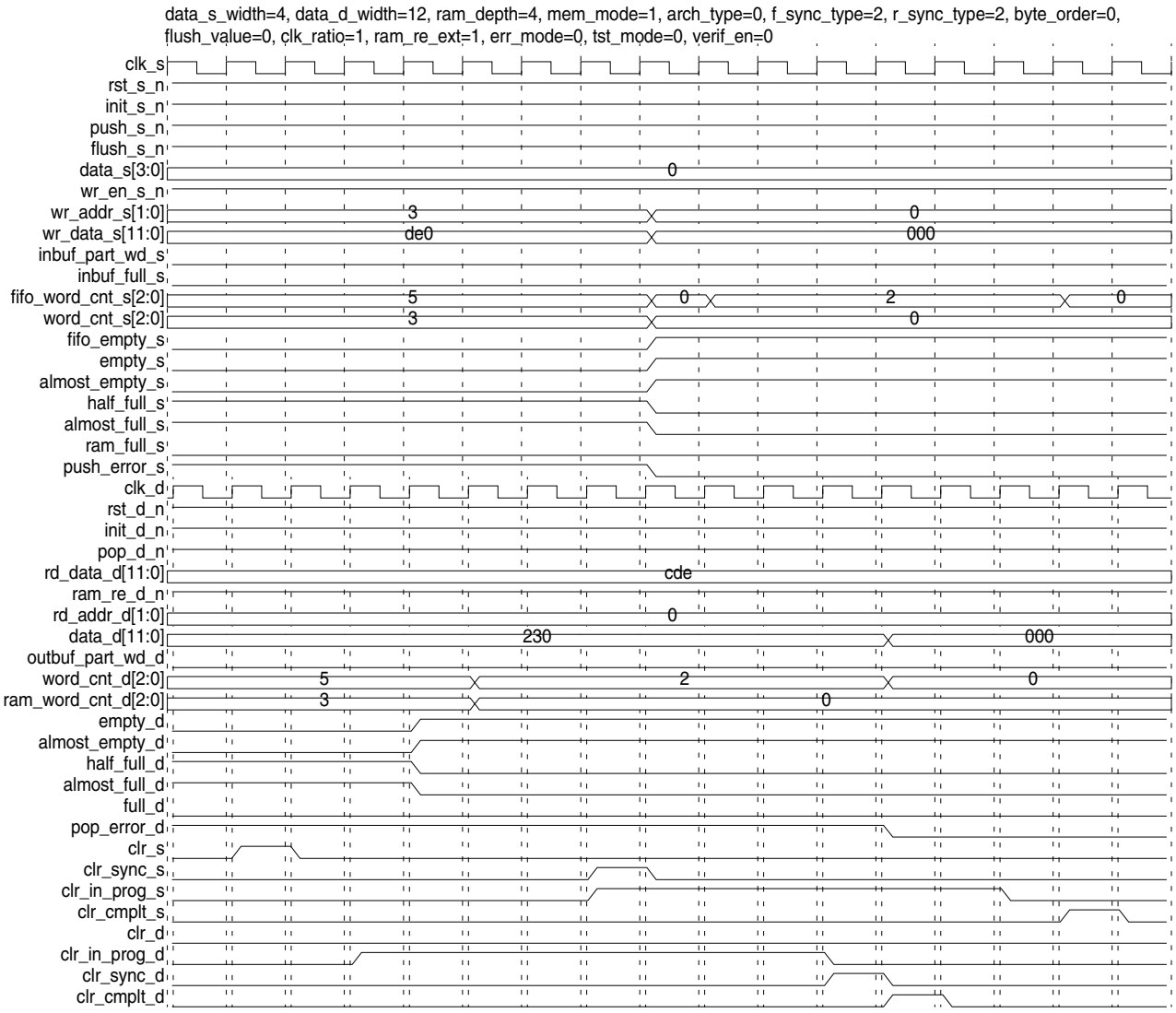


Figure 1-17 on page 47 describes the timing from an initiated clr_d pulse. In this case, clr_d is a single clk_d cycle, but the length of clr_d assertions are not restricted. The clr_d initiated clearing sequence is similar to the clr_s initiated clearing sequence with fewer synchronization feedback paths from beginning to completion.

When clr_d is asserted, it triggers the clr_in_prog_d to activate. This event then gets synchronized by the source domain (based on *r_sync_type*) and produces the clr_sync_s pulse and activation of clr_in_prog_s. The clr_sync_s pulse is then fed back, synchronized by the destination domain (based on *f_sync_type*), and de-activates the clr_in_prog_d signal. This is followed by active pulses of clr_sync_d and clr_cmplt_d. Finally, the clr_sync_d pulse is fed back to the source domain where it gets synchronized and de-activates the clr_in_prog_s signal followed by an asserted pulse of clr_cmplt_s.

During the clearing process there will be occasions when the status flags and word counts could report incorrectly just as in the clr_s initiated case. This situation occurs when clr_in_prog_d gets activated. Upon this event, the internal pointers (used to calculate word counts and status flags) could go between initialized and non-initialized states. This can be seen with fifo_word_cnt_s[2:0] going from 3 -> 5 -> 0 -> 2 during the "in progress" portion of the clear sequence. This type of counter changes could produce state changes in output status signals fifo_empty_s, empty_s, almost_empty_s, half_full_s, and almost_full_s. But note by the time the clearing sequence completes in the source domain as indicated by clr_cmplt_s all outputs and counters are in their initialized states. The same holds true in the destination domain for its counters, status outputs, and data with respect to an asserted clr_cmplt_d. Therefore, having the word count and status flags report incorrectly during the clearing process is benign based on the knowledge that once the clr_s is initiated the source domain knows that the clearing sequence has begun and no credence should be put on the system state until both domains are notified of clearing completion based on clr_cmplt_s and clr_cmplt_d assertions.

Configuration: same as Figure 1-15 on page 43.

Figure 1-17 clr_d Initiated Clearing Sequence

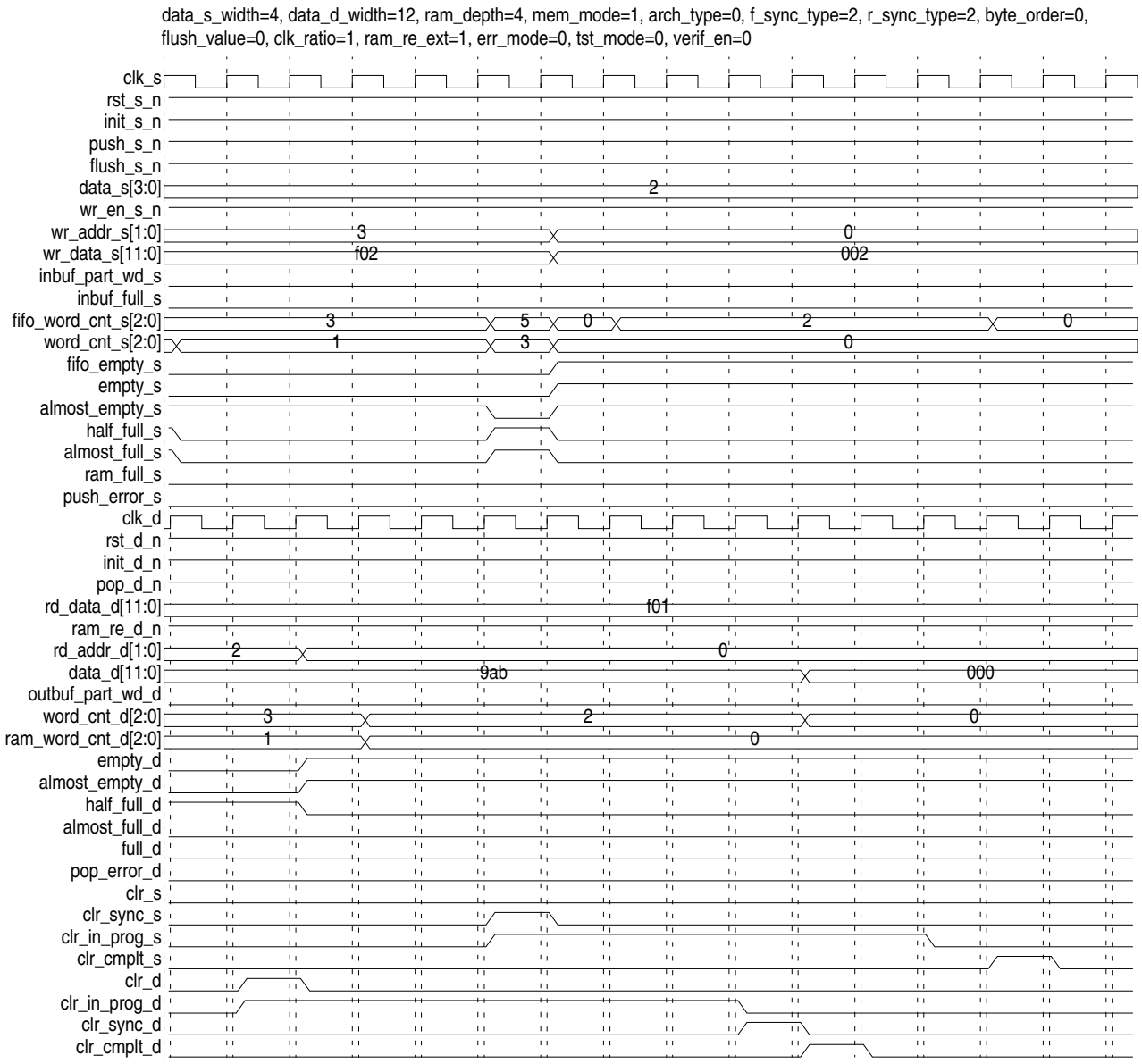


Figure 1-18 shows an initiation of a `clr_s` where its duration is much longer than one `clk_s` cycle. From this, the behavior of the `clr_in_prog_s` and `clr_in_prog_d` flags are sustained longer than those seen in Figure 1-16 in which `clr_s` was only asserted a single `clk_s` cycle.

Configuration: same as Figure 1-15.

Figure 1-18 Initiation of clr_s with duration much longer than one clk_s cycle

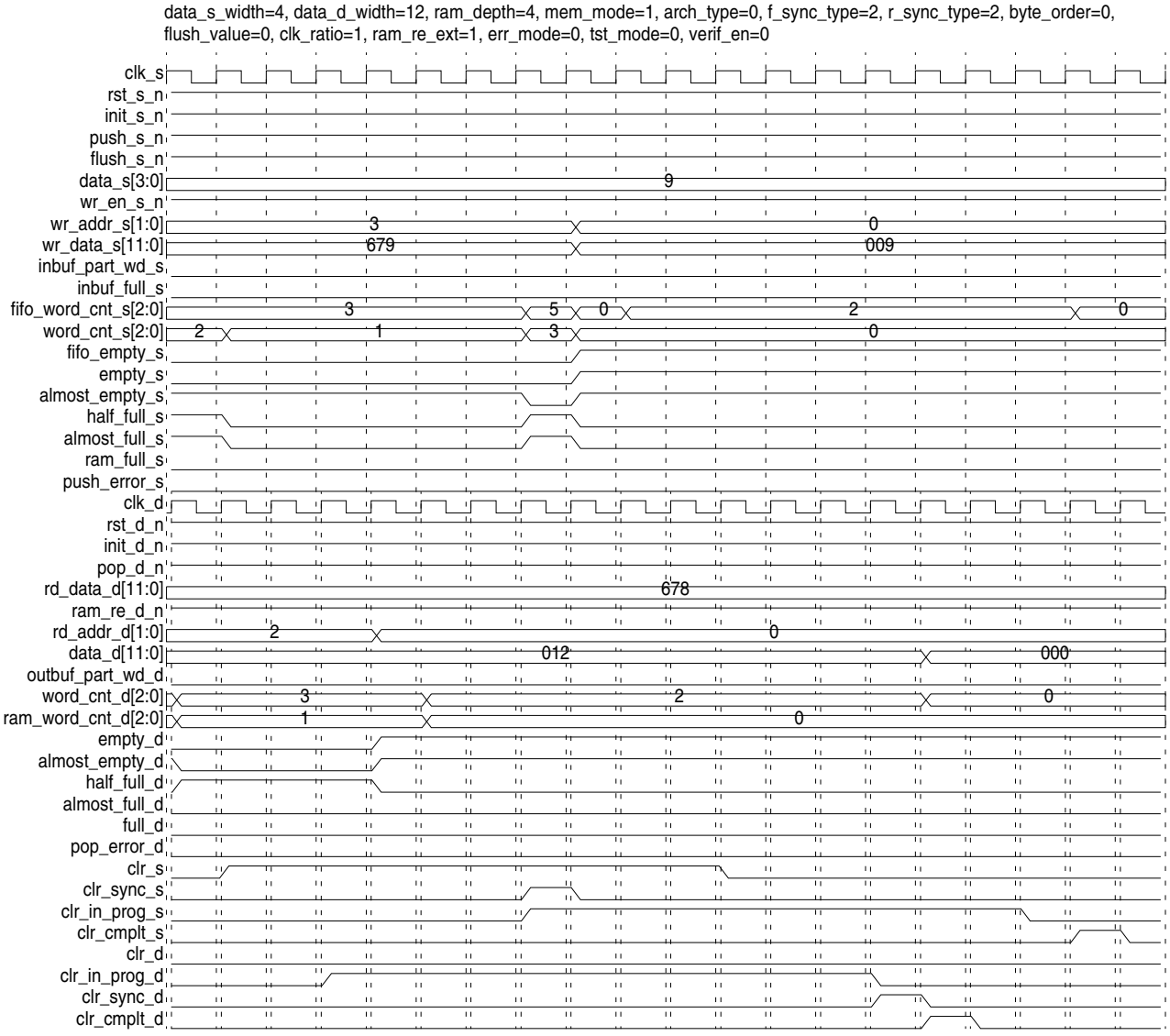


Figure 1-19 shows an example of a sequence where `rst_s_n` and `rst_d_n` are asserted. In this case, `clk_d` is faster than `clk_s` with `rst_s_n` asserted first followed by the assertion of `rst_d_n` within *f_sync_type*+1 `clk_d` cycles (*f_sync_type* is 2 in this example). Note that the word counts and addresses go to 0 and the status flags settle into the appropriate state. Note: *f_sync_type* is 2 and *r_sync_type* is 2

Figure 1-19 Example of Asynchronous System Reset

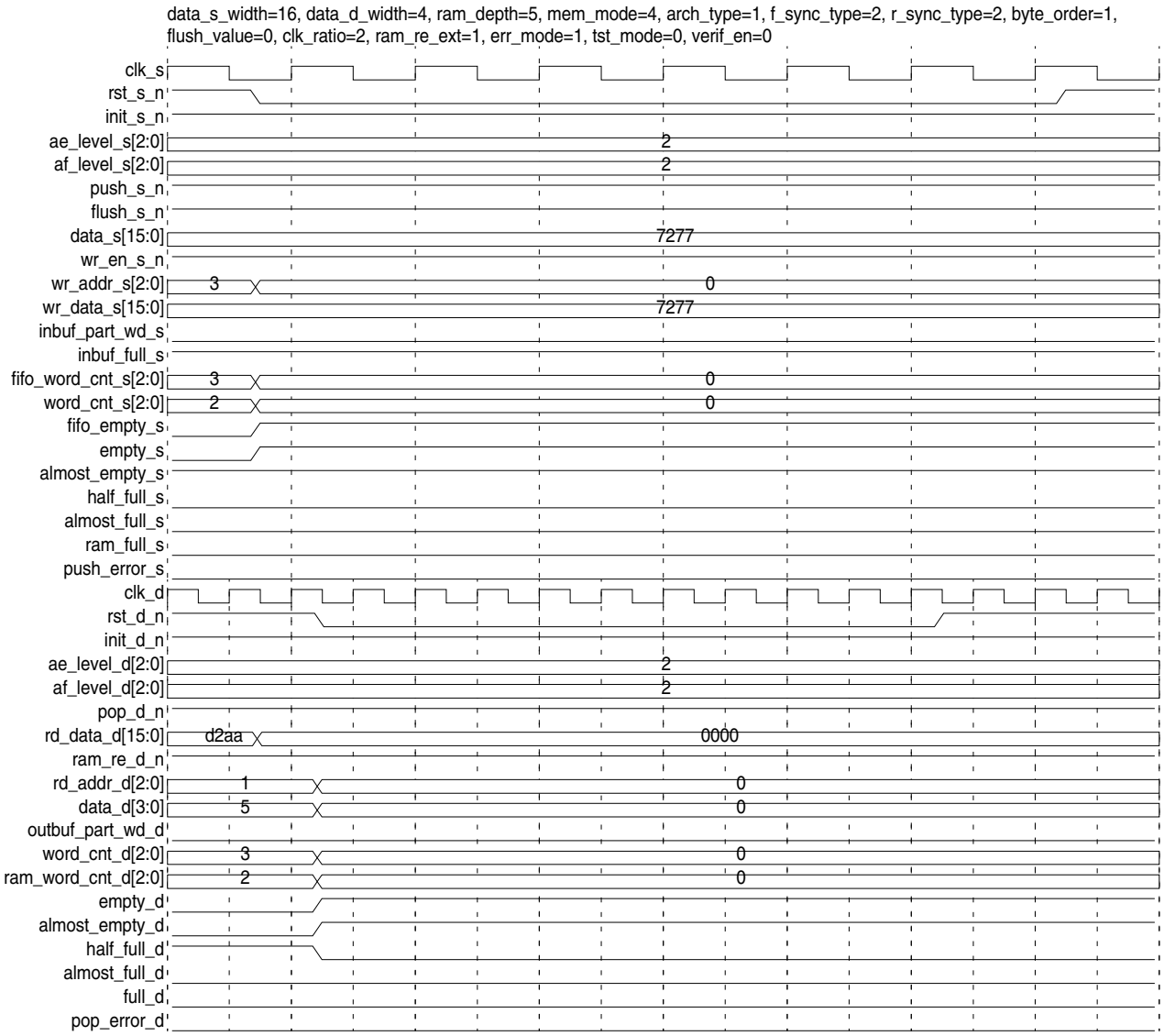
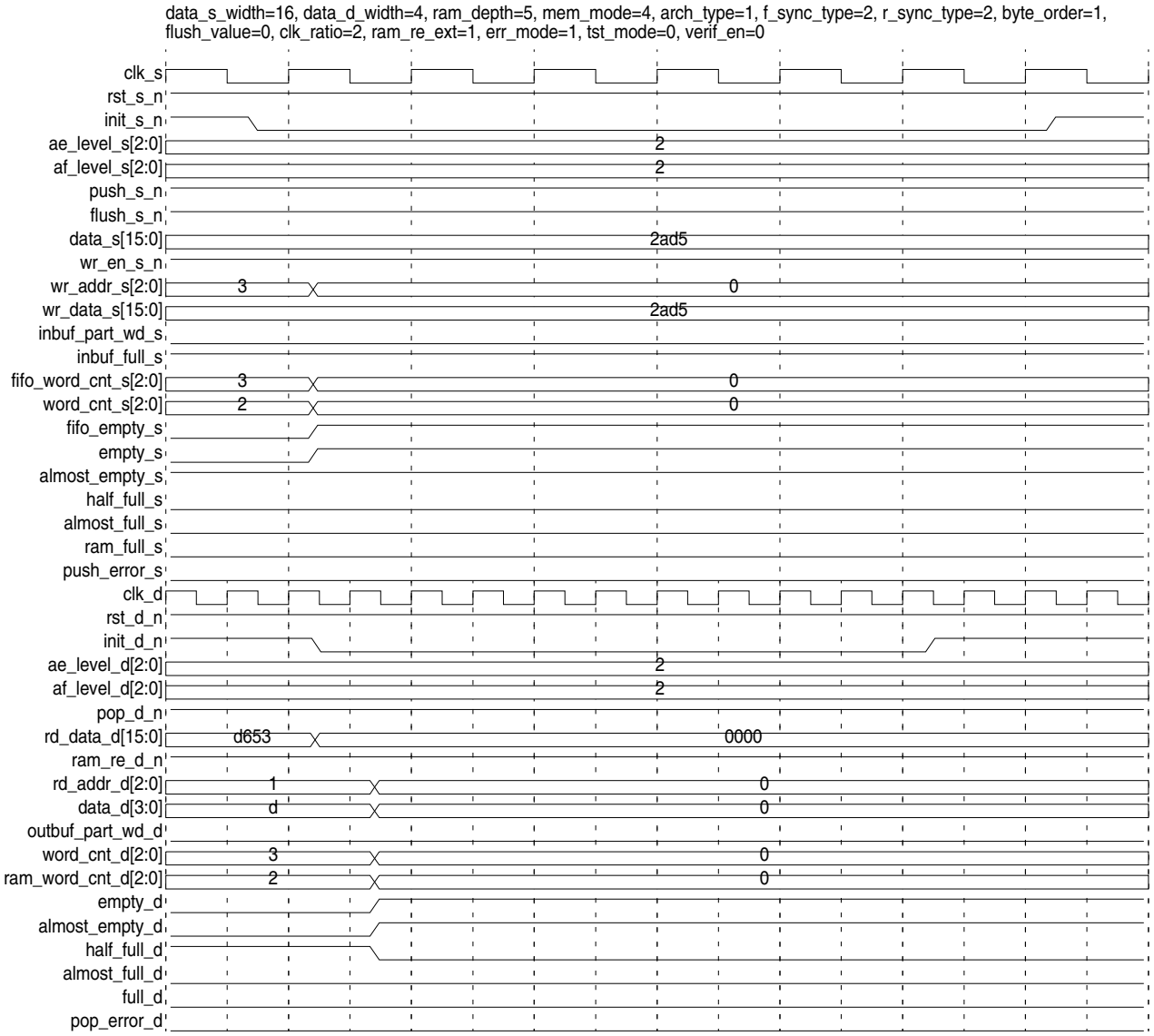


Figure 1-20 shows an example of a sequence where `init_s_n` and `init_d_n` are asserted. In this case, `clk_d` is faster than `clk_s` with `init_s_n` asserted first followed by the assertion of `init_d_n` within *f_sync_type*+1 `clk_d` cycles (*f_sync_type* is 2 in this example). Note that the word counts and addresses go to 0 and the status flags settle into the appropriate state. Note: *f_sync_type* is 2 and *r_sync_type* is 2

Figure 1-20 Example of Synchronous System Reset



Related Topics

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

HDL Usage Through Component Instantiation - VHDL

```
library IEEE,WORK,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DWARE.DW_Foundation_comp.all;

entity DW_asymfifoctl_2c_df_inst is
      generic (
        inst_data_s_width : POSITIVE := 16;
        inst_data_d_width : POSITIVE := 8;
        inst_ram_depth : POSITIVE := 8;
        inst_mem_mode : NATURAL := 5;
        inst_arch_type : NATURAL := 0;
        inst_f_sync_type : NATURAL := 2;
        inst_r_sync_type : NATURAL := 2;
        inst_byte_order : NATURAL := 0;
        inst_flush_value : NATURAL := 0;
        inst_clk_ratio : INTEGER := 2;
        inst_ram_re_ext : NATURAL := 1;
        inst_err_mode : NATURAL := 0;
        inst_tst_mode : NATURAL := 0;
        inst_verif_en : NATURAL := 1
        );
      port (
        inst_clk_s : in std_logic;
        inst_rst_s_n : in std_logic;
        inst_init_s_n : in std_logic;
        inst_clr_s : in std_logic;
        inst_ae_level_s : in std_logic_vector(bit_width(inst_ram_depth+1)-1 downto 0);
        inst_af_level_s : in std_logic_vector(bit_width(inst_ram_depth+1)-1 downto 0);
        inst_push_s_n : in std_logic;
        inst_flush_s_n : in std_logic;
        inst_data_s : in std_logic_vector(inst_data_s_width-1 downto 0);
        clr_sync_s_inst : out std_logic;
        clr_in_prog_s_inst : out std_logic;
        clr_cmplt_s_inst : out std_logic;
        wr_en_s_n_inst : out std_logic;
        wr_addr_s_inst : out std_logic_vector(bit_width(inst_ram_depth)-1 downto 0);
        wr_data_s_inst : out
std_logic_vector(maximum(inst_data_s_width,inst_data_d_width)-1 downto 0);
        inbuf_part_wd_s_inst : out std_logic;
        inbuf_full_s_inst : out std_logic;
        fifo_word_cnt_s_inst : out
std_logic_vector(bit_width((inst_ram_depth+1+(inst_mem_mode mod 2)+((inst_mem_mode/2)
mod 2))+1)-1 downto 0);
        word_cnt_s_inst : out std_logic_vector(bit_width(inst_ram_depth+1)-1 downto 0);
        fifo_empty_s_inst : out std_logic;
        empty_s_inst : out std_logic;
```

```
        almost_empty_s_inst : out std_logic;
        half_full_s_inst : out std_logic;
        almost_full_s_inst : out std_logic;
        ram_full_s_inst : out std_logic;
        push_error_s_inst : out std_logic;
        inst_clk_d : in std_logic;
        inst_rst_d_n : in std_logic;
        inst_init_d_n : in std_logic;
        inst_clr_d : in std_logic;
        inst_ae_level_d : in
std_logic_vector(bit_width((inst_ram_depth+1+(inst_mem_mode mod 2)+((inst_mem_mode/2)
mod 2))+1)-1 downto 0);
        inst_af_level_d : in
std_logic_vector(bit_width((inst_ram_depth+1+(inst_mem_mode mod 2)+((inst_mem_mode/2)
mod 2))+1)-1 downto 0);
        inst_pop_d_n : in std_logic;
        inst_rd_data_d : in
std_logic_vector(maximum(inst_data_s_width,inst_data_d_width)-1 downto 0);
        clr_sync_d_inst : out std_logic;
        clr_in_prog_d_inst : out std_logic;
        clr_cmplt_d_inst : out std_logic;
        ram_re_d_n_inst : out std_logic;
        rd_addr_d_inst : out std_logic_vector(bit_width(inst_ram_depth)-1 downto 0);
        data_d_inst : out std_logic_vector(inst_data_d_width-1 downto 0);
        outbuf_part_wd_d_inst : out std_logic;
        word_cnt_d_inst : out
std_logic_vector(bit_width((inst_ram_depth+1+(inst_mem_mode mod 2)+((inst_mem_mode/2)
mod 2))+1)-1 downto 0);
        ram_word_cnt_d_inst : out std_logic_vector(bit_width(inst_ram_depth+1)-1 downto
0);
        empty_d_inst : out std_logic;
        almost_empty_d_inst : out std_logic;
        half_full_d_inst : out std_logic;
        almost_full_d_inst : out std_logic;
        full_d_inst : out std_logic;
        pop_error_d_inst : out std_logic;
        inst_test : in std_logic
        );
    end DW_asymfifoctl_2c_df_inst;


architecture inst of DW_asymfifoctl_2c_df_inst is


begin

    -- Instance of DW_asymfifoctl_2c_df
    U1 : DW_asymfifoctl_2c_df
```

```
    generic map ( data_s_width => inst_data_s_width, data_d_width => inst_data_d_width,
ram_depth => inst_ram_depth, mem_mode => inst_mem_mode, arch_type => inst_arch_type,
f_sync_type => inst_f_sync_type, r_sync_type => inst_r_sync_type, byte_order =>
inst_byte_order, flush_value => inst_flush_value, clk_ratio => inst_clk_ratio,
ram_re_ext => inst_ram_re_ext, err_mode => inst_err_mode, tst_mode => inst_tst_mode,
verif_en => inst_verif_en )
    port map ( clk_s => inst_clk_s, rst_s_n => inst_rst_s_n, init_s_n => inst_init_s_n,
clr_s => inst_clr_s, ae_level_s => inst_ae_level_s, af_level_s => inst_af_level_s,
push_s_n => inst_push_s_n, flush_s_n => inst_flush_s_n, data_s => inst_data_s,
clr_sync_s => clr_sync_s_inst, clr_in_prog_s => clr_in_prog_s_inst, clr_cmplt_s =>
clr_cmplt_s_inst, wr_en_s_n => wr_en_s_n_inst, wr_addr_s => wr_addr_s_inst, wr_data_s
=> wr_data_s_inst, inbuf_part_wd_s => inbuf_part_wd_s_inst, inbuf_full_s =>
inbuf_full_s_inst, fifo_word_cnt_s => fifo_word_cnt_s_inst, word_cnt_s =>
word_cnt_s_inst, fifo_empty_s => fifo_empty_s_inst, empty_s => empty_s_inst,
almost_empty_s => almost_empty_s_inst, half_full_s => half_full_s_inst, almost_full_s
=> almost_full_s_inst, ram_full_s => ram_full_s_inst, push_error_s =>
push_error_s_inst, clk_d => inst_clk_d, rst_d_n => inst_rst_d_n, init_d_n =>
inst_init_d_n, clr_d => inst_clr_d, ae_level_d => inst_ae_level_d, af_level_d =>
inst_af_level_d, pop_d_n => inst_pop_d_n, rd_data_d => inst_rd_data_d, clr_sync_d =>
clr_sync_d_inst, clr_in_prog_d => clr_in_prog_d_inst, clr_cmplt_d => clr_cmplt_d_inst,
ram_re_d_n => ram_re_d_n_inst, rd_addr_d => rd_addr_d_inst, data_d => data_d_inst,
outbuf_part_wd_d => outbuf_part_wd_d_inst, word_cnt_d => word_cnt_d_inst,
ram_word_cnt_d => ram_word_cnt_d_inst, empty_d => empty_d_inst, almost_empty_d =>
almost_empty_d_inst, half_full_d => half_full_d_inst, almost_full_d =>
almost_full_d_inst, full_d => full_d_inst, pop_error_d => pop_error_d_inst, test =>
inst_test );


end inst;

-- Configuration for use with a VHDL simulator
-- pragma translate_off
library DW03;
configuration DW_asymfifoctl_2c_df_inst_cfg_inst of DW_asymfifoctl_2c_df_inst is
  for inst
    -- NOTE: If desiring to model missampling, uncomment the following
    -- line.  Doing so, however, will cause inconsequential errors
    -- when analyzing or reading this configuration before synthesis.
    -- for U1 : DW_fifoctl_2c_df use configuration
DW03.DW_asymfifoctl_2c_df_cfg_sim_ms;  end for;
  end for; -- inst
end DW_asymfifoctl_2c_df_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_asymfifoctl_2c_df_inst( inst_clk_s, inst_rst_s_n, inst_init_s_n, inst_clr_s,
inst_ae_level_s, inst_af_level_s, inst_push_s_n, inst_flush_s_n, inst_data_s,
clr_sync_s_inst, clr_in_prog_s_inst, clr_cmplt_s_inst, wr_en_s_n_inst, wr_addr_s_inst,
wr_data_s_inst, inbuf_part_wd_s_inst, inbuf_full_s_inst, fifo_word_cnt_s_inst,
word_cnt_s_inst, fifo_empty_s_inst, empty_s_inst, almost_empty_s_inst,
half_full_s_inst, almost_full_s_inst, ram_full_s_inst, push_error_s_inst, inst_clk_d,
inst_rst_d_n, inst_init_d_n, inst_clr_d, inst_ae_level_d, inst_af_level_d,
inst_pop_d_n, inst_rd_data_d, clr_sync_d_inst, clr_in_prog_d_inst, clr_cmplt_d_inst,
ram_re_d_n_inst, rd_addr_d_inst, data_d_inst, outbuf_part_wd_d_inst, word_cnt_d_inst,
ram_word_cnt_d_inst, empty_d_inst, almost_empty_d_inst, half_full_d_inst,
almost_full_d_inst, full_d_inst, pop_error_d_inst, inst_test );

parameter data_s_width = 4;
parameter data_d_width = 8;
parameter ram_depth = 8;
parameter mem_mode = 3;
parameter arch_type = 1;
parameter f_sync_type = 2;
parameter r_sync_type = 3;
parameter byte_order = 1;
parameter flush_value = 1;
parameter clk_ratio = 1;
parameter ram_re_ext = 1;
parameter err_mode = 0;
parameter tst_mode = 0;
parameter verif_en = 1;
`define addr_width     3 // ceil(log2(ram_depth))
`define ram_cnt_width  4 // ceil(log2(ram_depth+1))
`define fifo_cnt_width 4 // ceil(log2((ram_depth+1+(mem_mode % 2)+((mem_mode/2) %
2))+1))

input inst_clk_s;
input inst_rst_s_n;
input inst_init_s_n;
input inst_clr_s;
input [`ram_cnt_width-1:0] inst_ae_level_s;
input [`ram_cnt_width-1:0] inst_af_level_s;
input inst_push_s_n;
input inst_flush_s_n;
input [data_s_width-1:0] inst_data_s;
output clr_sync_s_inst;
output clr_in_prog_s_inst;
output clr_cmplt_s_inst;
output wr_en_s_n_inst;
output [`addr_width-1:0] wr_addr_s_inst;
output [data_d_width-1:0] wr_data_s_inst;
output inbuf_part_wd_s_inst;
```

```
output inbuf_full_s_inst;
output [`fifo_cnt_width-1:0] fifo_word_cnt_s_inst;
output [`ram_cnt_width-1:0] word_cnt_s_inst;
output fifo_empty_s_inst;
output empty_s_inst;
output almost_empty_s_inst;
output half_full_s_inst;
output almost_full_s_inst;
output ram_full_s_inst;
output push_error_s_inst;
input inst_clk_d;
input inst_rst_d_n;
input inst_init_d_n;
input inst_clr_d;
input [`fifo_cnt_width-1:0] inst_ae_level_d;
input [`fifo_cnt_width-1:0] inst_af_level_d;
input inst_pop_d_n;
input [data_d_width-1:0] inst_rd_data_d;
output clr_sync_d_inst;
output clr_in_prog_d_inst;
output clr_cmplt_d_inst;
output ram_re_d_n_inst;
output [`addr_width-1:0] rd_addr_d_inst;
output [data_d_width-1:0] data_d_inst;
output outbuf_part_wd_d_inst;
output [`fifo_cnt_width-1:0] word_cnt_d_inst;
output [`ram_cnt_width-1:0] ram_word_cnt_d_inst;
output empty_d_inst;
output almost_empty_d_inst;
output half_full_d_inst;
output almost_full_d_inst;
output full_d_inst;
output pop_error_d_inst;
input inst_test;

    // Instance of DW_asymfifoctl_2c_df
    DW_asymfifoctl_2c_df #(data_s_width, data_d_width, ram_depth, mem_mode, arch_type,
f_sync_type, r_sync_type, byte_order, flush_value, clk_ratio, ram_re_ext, err_mode,
tst_mode, verif_en)
```

```
      U1 ( .clk_s(inst_clk_s), .rst_s_n(inst_rst_s_n), .init_s_n(inst_init_s_n),
.clr_s(inst_clr_s), .ae_level_s(inst_ae_level_s), .af_level_s(inst_af_level_s),
.push_s_n(inst_push_s_n), .flush_s_n(inst_flush_s_n), .data_s(inst_data_s),
.clr_sync_s(clr_sync_s_inst), .clr_in_prog_s(clr_in_prog_s_inst),
.clr_cmplt_s(clr_cmplt_s_inst), .wr_en_s_n(wr_en_s_n_inst), .wr_addr_s(wr_addr_s_inst),
.wr_data_s(wr_data_s_inst), .inbuf_part_wd_s(inbuf_part_wd_s_inst),
.inbuf_full_s(inbuf_full_s_inst), .fifo_word_cnt_s(fifo_word_cnt_s_inst),
.word_cnt_s(word_cnt_s_inst), .fifo_empty_s(fifo_empty_s_inst), .empty_s(empty_s_inst),
.almost_empty_s(almost_empty_s_inst), .half_full_s(half_full_s_inst),
.almost_full_s(almost_full_s_inst), .ram_full_s(ram_full_s_inst),
.push_error_s(push_error_s_inst), .clk_d(inst_clk_d), .rst_d_n(inst_rst_d_n),
.init_d_n(inst_init_d_n), .clr_d(inst_clr_d), .ae_level_d(inst_ae_level_d),
.af_level_d(inst_af_level_d), .pop_d_n(inst_pop_d_n), .rd_data_d(inst_rd_data_d),
.clr_sync_d(clr_sync_d_inst), .clr_in_prog_d(clr_in_prog_d_inst),
.clr_cmplt_d(clr_cmplt_d_inst), .ram_re_d_n(ram_re_d_n_inst),
.rd_addr_d(rd_addr_d_inst), .data_d(data_d_inst),
.outbuf_part_wd_d(outbuf_part_wd_d_inst), .word_cnt_d(word_cnt_d_inst),
.ram_word_cnt_d(ram_word_cnt_d_inst), .empty_d(empty_d_inst),
.almost_empty_d(almost_empty_d_inst), .half_full_d(half_full_d_inst),
.almost_full_d(almost_full_d_inst), .full_d(full_d_inst),
.pop_error_d(pop_error_d_inst), .test(inst_test) );

endmodule
```

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