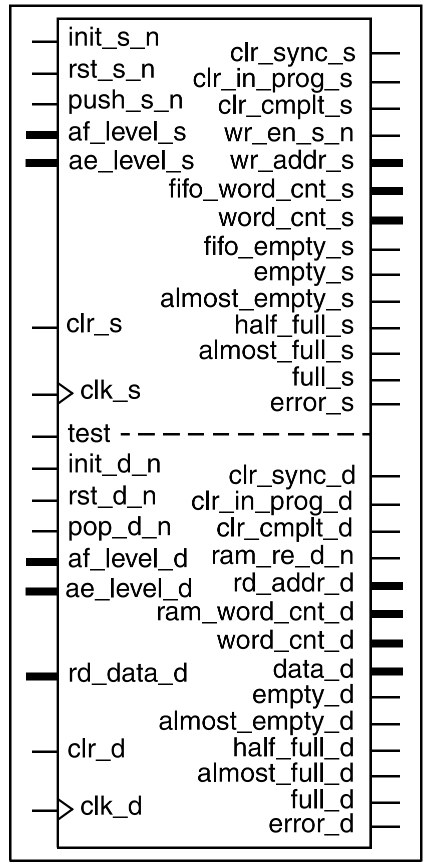
DW_fifoctl_2c_df

Dual Clock FIFO Controller with Dynamic Flags

Version, STAR, and myDesignWare Subscriptions: IP Directory

Features and Benefits

Revision History



- 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 data width
- 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 with the DesignWare-LP license (Get the minPower version of this datasheet)

Description

DW_fifoctl_2c_df is a dual independent clock 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.

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, full, and almost full) and operation error detection logic for both clock domains.

The FIFO controller provides parameterized data width, 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.

Table 1-1 Pin Description

Pin Name	Width (bits)	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($\log_2$[*ram_depth* + 1])	Input	Source domain almost empty level for the `almost_empty_s` output (the number of words in the RAM at or below which the `almost_empty_s` flag is active) (see *eff_depth* note below table)
af_level_s	ceil($\log_2$[*ram_depth* + 1])	Input	Source domain almost full level for the `almost_full_s` output (the number of empty memory locations in the RAM at which the `almost_full_s` flag is active).
push_s_n	1 bit	Input	Source domain push request (active low)
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($\log_2$[*ram_depth*])	Output	Source domain write address to RAM (registered)

Table 1-1 Pin Description (Continued)

Pin Name	Width (bits)	Direction	Function
fifo_word_cnt_s	ceil($\log_2$[*eff_depth* + 1])	Output	Source domain total word count in the RAM and cache (see *eff_depth* note below table)
word_cnt_s	ceil($\log_2$[*ram_depth* + 1])	Output	Source domain RAM word count (see Note on *ram_depth*)
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 RAM almost empty flag (determined by `ae_level_s` port)
half_full_s	1 bit	Output	Source domain RAM half full flag
almost_full_s	1 bit	Output	Source domain RAM full flag (determined by `af_level_s` port)
full_s	1 bit	Output	Source domain RAM almost full flag
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($\log_2$[*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($\log_2$[*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	*width* bit(s)	Input	Destination domain read data
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)
rd_addr_d	ceil($\log_2$(ram_depth))	Output	Destination domain read address to RAM (registered)
data_d	*width* bit(s)	Output	Destination domain data to pop
word_cnt_d	ceil($\log_2$[*eff_depth* + 1])	Output	Destination domain FIFO word count (see *eff_depth* note below table)
ram_word_cnt_d	ceil($\log_2$[*ram_depth* + 1])	Output	Destination domain RAM word count (see note on *ram_depth* parameter below)
empty_d	1 bit	Output	Destination domain FIFO empty flag

Table 1-1 Pin Description (Continued)

Pin Name	Width (bits)	Direction	Function
almost_empty_d	1 bit	Output	Destination domain FIFO almost empty flag (determined by `ae_level_d` parameter)
half_full_d	1 bit	Output	Destination domain FIFO half full flag
almost_full_d	1 bit	Output	Destination domain FIFO almost full flag (determined by `af_level_d` port)
full_d	1 bit	Output	Destination domain FIFO full flag
error_d	1 bit	Output	Destination domain push error flag (overrun)
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 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

The Source Domain status flags (such as `almost_empty_s` and `half_full_s`) are derived based on the *ram_depth* value. The Destination Domain status flags (such as a `half_full_s` and `full_s`) are calculated based on the *eff_depth* value.

Table 1-3 Parameters

Parameter	Values	Function
width	1 to 1024 Default: 8	Vector width of input `data_s` and output `data_d`
ram_depth[a]	4 to 16777216 Default: 8	Desired number of FIFO locations to be operated out of RAM not including the cache.
mem_mode	0 or 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

Table 1-3 Parameters (Continued)

Parameter	Values	Function
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 w/ 1st stage negative edge and 2nd stage positive edge capturing 2: 2-stage synchronization w/ both stages positive edge capturing 3: 3-stage synchronization w/ 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 & 2nd stage positive edge capturing 2: 2-stage synchronization w/ both stages positive edge capturing 3: 3-stage synchronization w/ all stages positive edge capturing
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 6. 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	0 to 2 Default: 0	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: 1	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 For a more information, see "Simulation Methodology" on page 9.

Table 1-3 Parameters (Continued)

Parameter	Values	Function
clr_dual_domain	0 or 1 Default: 1	Activity of `clr_s` and/or `clr_d` 0: Either `clr_s` or `clr_d` can be activated, but the other must be tied 'low' 1: Both `clr_s` and `clr_d` can be activated
arch_type[b]	0 or 1 Default: 0	Pre-fetch cache architecture type: 0: Pipeline style (PL cache) - 'rtl' implementation 1: Register File style (RF cache) - 'lpwr' implementation

a. Parameter *ram_depth* is not necessarily the number of RAM locations needed to operate the FIFO. See "Memory Depth Considerations and Setting ram_depth" on page 12.

b. Note: For *arch_type* equal to 1, the RF cache is used (lpwr implementation) when a Low-Power (DesignWare-LP) license is available and *mem_mode* is not 0 or 4. If a Low-Power license is not available or *mem_mode* is 0 or 4, the PL cache (rtl implementation) is always used not matter the setting of *arch_type*.

Table 1-4 Synthesis Implementations

Implementation Name	Function	License Required
rtl	Synthesis model	DesignWare

When Is It Necessary to Determine *clk_ratio*

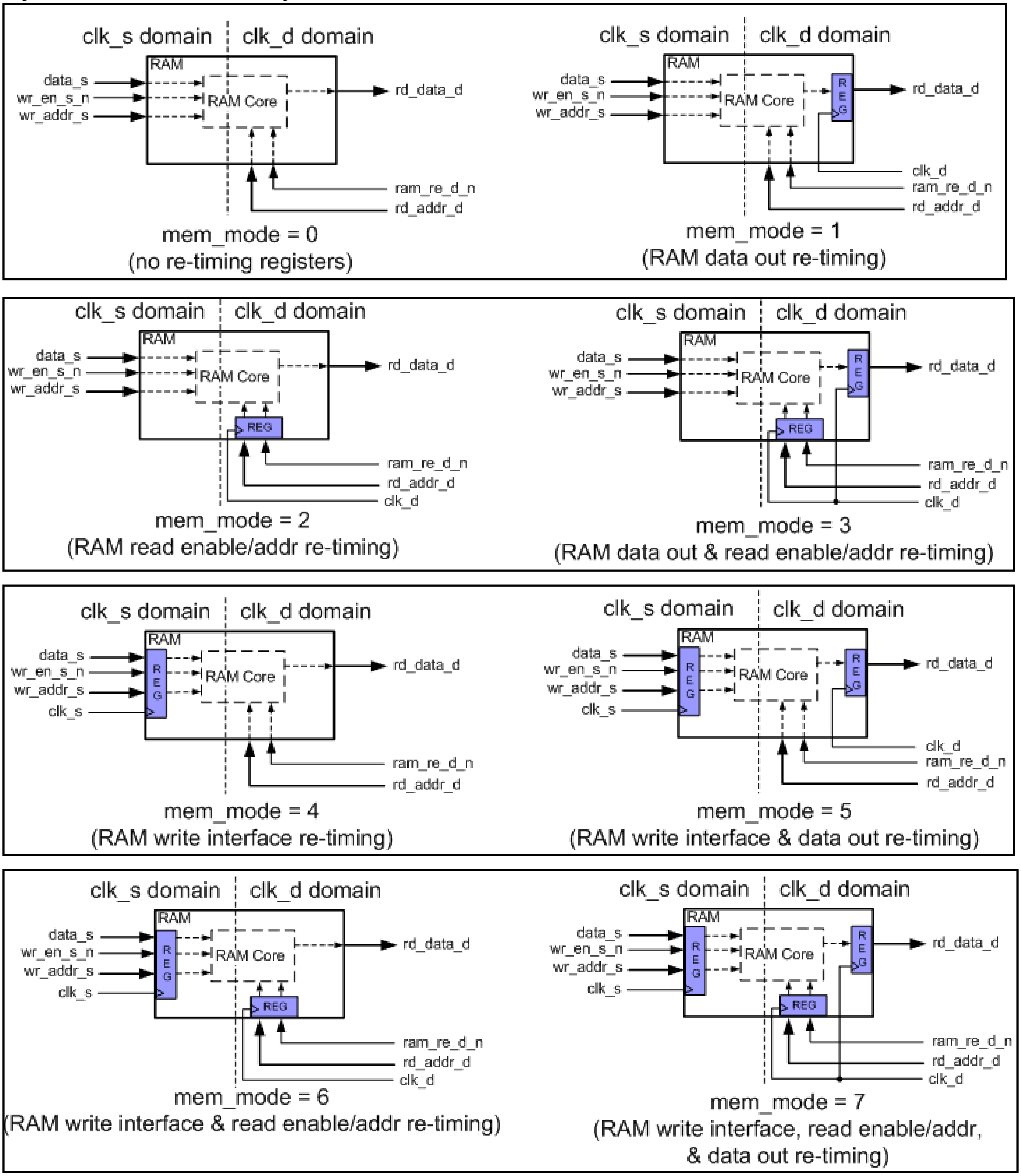
The parameter *clk_ratio* is only relevant when the parameter *mem_mode* indicates there are retiming registers on the input (write port or read port or both) of the RAM being used for the FIFO. When *mem_mode* is set to either 0 (no retiming registers in the RAM), or 1 (retiming registers only at the 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 being used with the DW_fifoctl_2c_df is needed. The following diagrams show the 8 possible RAM architectures that can interface with DW_fifoctl_2c_df and the required *mem_mode* setting for each.

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



Detailed Description of Parameter *arch_type*

The *arch_type* parameter is available for selection of the pre-fetch cache structure that will reside 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 1-10 on page 14), then a "register file" (RF) style of caching (lpwr implementation) is used provided a DesignWare-LP license is available.

When the RF Cache structure is desired and this component is configured accordingly, the lpwr implementation is automatically selected when a DesignWare-LP license is available. Only a "set implementation" to the rtl implementation will override the selection of the lpwr implementation. If no DesignWare-LP license is available, but the component is configured to attempt to use the RF Cache (i.e., 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 DesignWare-LP license is not consumed.

Table 1-5 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 1-5 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, 5-7	rtl, lpwr[a]

a. NOTE: The lpwr implementation is only available with a DesignWare-LP license. If a DesignWare-LP 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 DesignWare-LP license is not consumed.

Table 1-6 Simulation Models

Model	Function
DW03.DW_FIFOCTL_2C_DF_CFG_SIM	Design unit name for VHDL simulation
DW03.DW_FIFOCTL_2C_DF_CFG_SIM_MS	Design unit name for VHDL simulation with mis-sampling enabled.
dw/dw03/src/DW_fifoctl_2c_df_sim.vhd	VHDL simulation model source code (modeling RTL) - no missampling
dw/sim_ver/DW_fifoctl_2c_df.v	Verilog simulation model source code

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, no 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. Note: If `DW_MODEL_MISSAMPLES is not defined, the Verilog simulation model behaves as if *verif_en* is set to 0.

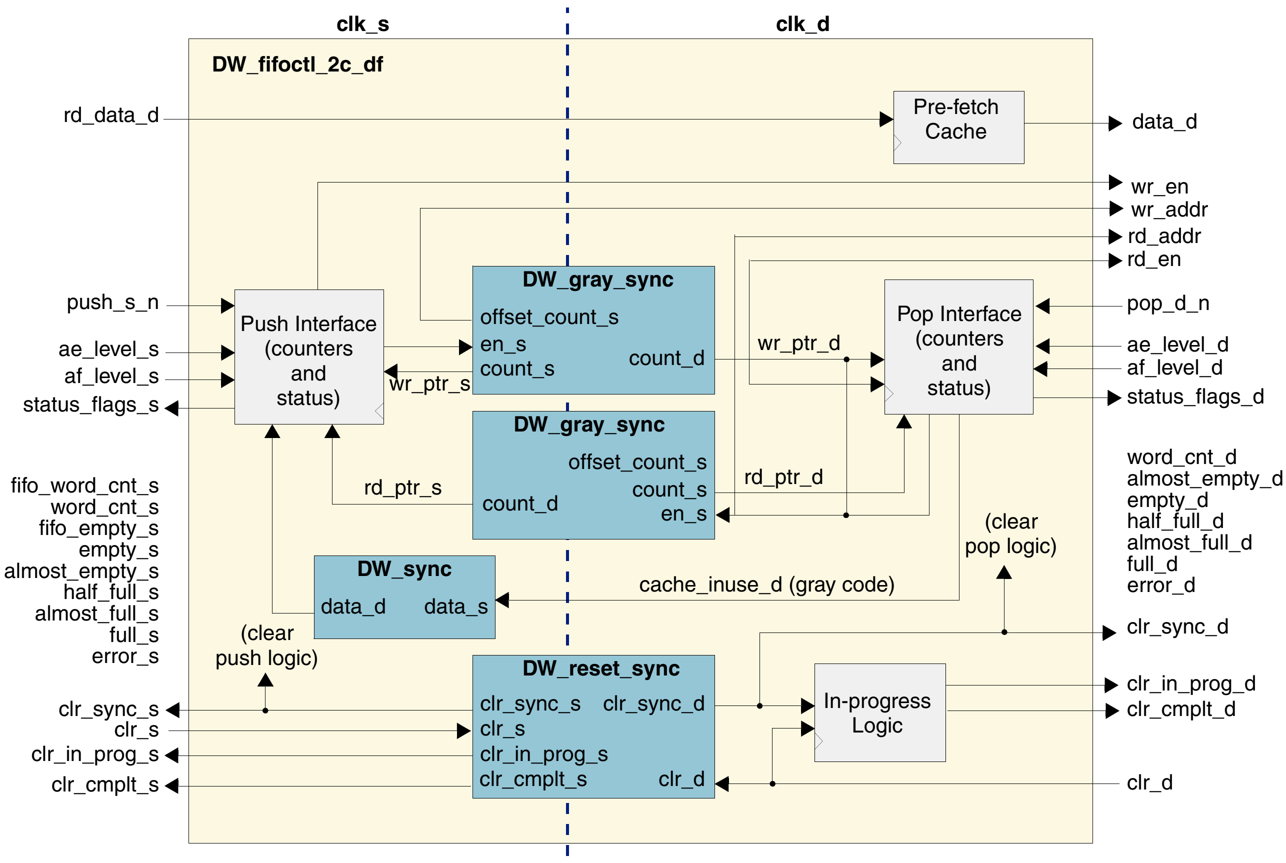
For VHDL simulation enabling missampling, an alternative simulation architecture is provided. This architecture is named sim_ms. The parameter *verif_en* only has meaning when utilizing the sim_ms. That is, when binding the "sim" simulation architecture the *verif_en* value is ignored and the model effectively behaves as though *verif_en* is set to '0'. See "HDL Usage Through Component Instantiation - VHDL" on page 45 for an example utilizing each architecture.

The following diagram is a basic block diagram of the DW_fifoctl_2c_df.

Block Diagram

Figure 1-2 shows the block diagram for the DW_fifoctl_2c_df component.

Figure 1-2 DW_fifoctl_2c_df Block Diagram



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 domains' 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 domains' clocks.

Besides satisfying simultaneous assertion of each domains 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, though not required, to release the source clock domain's reset last.

See Figure 1-15 on page 41 and Figure 1-16 on page 42 for examples of asynchronous and synchronous system reset assertion.

Clearing FIFO Controller (synchronous)

The DW_fifoctl_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 1-12 on page 38 and Figure 1-13 on page 39 show the clearing sequence for when `clr_s` and `clr_d`, respectively, are asserted. Additionally, Figure 1-14 on page 40 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. Figure 1-12 (`clr_s` initiated clearing) shows an example of this.

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 made. 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.

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 1-7 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 1-8 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 (4, 8, 16, 32, and so on), 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 1-9 Desired Number of FIFO Locations Is Even and Power of 2

FIFO Locations Desired	RAM Size	RAM Address Range	*ram_depth* value
32	32	0 to 31	32
128	128	0 to 127	128

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 (push)

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 `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.

Write Errors

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

- The `push_s_n` input is asserted (low), and
- The `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.

Destination Domain Caching (pop interface pre-fetching)

The popping interface contains output buffering (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 a 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 1-10 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

Reading from the Memory

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_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:

1. 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.
2. 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 `rd_data_d` 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 pipelined data words based on the *mem_mode* parameter as described in Figure 1-10 on page 14. When the head of the cache (the `data_d` output port) contains a valid entry the FIFO is considered not empty, i.e. the `empty_d` flag is not asserted, a legal pop of the FIFO is allowed (asserting `pop_d_n`). When `empty_d` is not asserted the `data_d` contents is the next valid word from the FIFO. The assertion of `pop_d_n` causes the cache pipeline to shift 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 closest vacated stage to the head of the cache. For example, if only the head stage of the cache contains valid data and `rd_data_d` is valid and `pop_d_n` is asserted, then on the next rising edge of `clk_d` the `rd_data_d` (data from RAM) is loaded to the head of the cache. This event would keep `empty_d` de-asserted and allow for another pop on the next rising edge of `clk_d`.

However, if only the head of the cache contains valid data 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 at the head of the cache (`data_d`) is 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 `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_s`, `almost_empty_s`, `empty_d`, and `almost_empty_d` are initialized to 1 (high)
- 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 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 DW_fifoctl_2c_df.

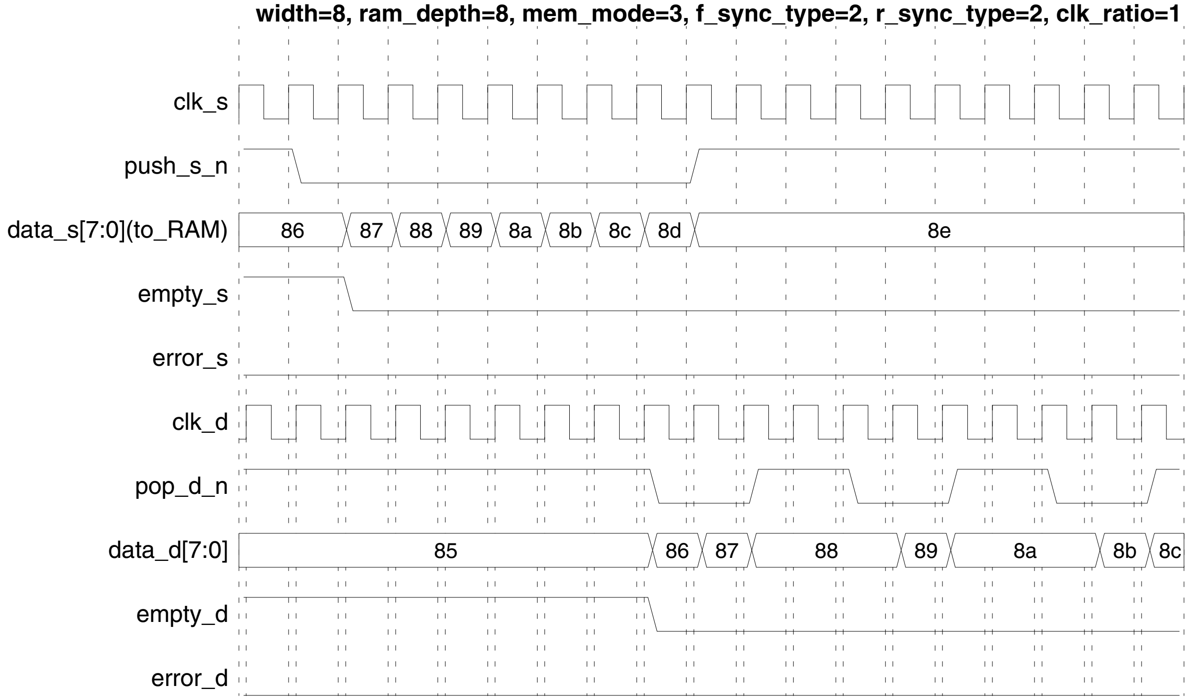
Generally, there's no specific rule of thumb in selecting which cache architecture will render the least power dissipation. This may require the design process to include some experimentation in using both cache styles to characterize behavior based on the system parameters.

Keep in mind, 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 would become relevant. When mem_mode is either 0 or 4, the pre-fetch cache depth is 1 and defaults to the rtl implementation (see Table 1-5 on page 8).

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

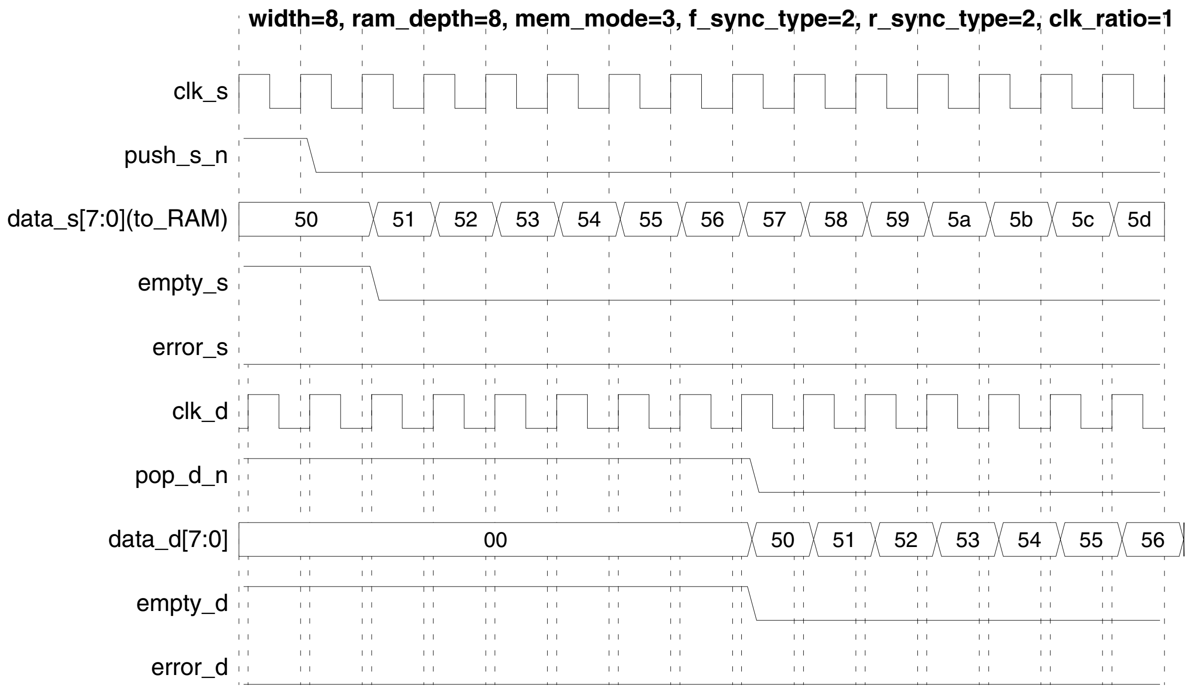
The key factor that determines which cache architecture will provide the least power consumption over the other hinges around the behavior of the pop requests (pop_d_n). If pop requests are issued in short bursts of 3 or less, the RF cache (lpwr implementation) will most likely yield the least power 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 two cases show the two extremes in data flow behavior.

Figure 1-3 CASE 1: Push packets and pop alternately with 2 cycles active then 2 cycles inactive



The data flow behavior for Figure X shows a packet of length *depth* (8 in this case) with words pushed contiguously before pushing halts. 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 of 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 *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 *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.

Figure 1-4 CASE 2: Popping in long contiguous bursts



When long bursts of contiguous pop requests are issued (Figure 1-4), this produces a type of data flow more conducive to using the PL cache architecture. 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.

Note that in CASE 1 above, the write and read address logic is always active as well. But the difference there is that the data flow through the cache caused by the bursting pop requests is such that the cache is full, almost full, or becoming full. Thus, all the cache locations are shifting in or out data on every cycle. Therefore, power consumption of the cache is predominantly the shifting of data (inherent to the PL cache) and not the write and read address logic. Thus, selecting the RF cache, in general, will provide best power consumption results in the CASE 1 scenario. Again, the differences in favor of the RF cache over the PL cache in this data flow behavior increase as data widths increase.

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. By Gray coding the address values that are synchronized across clock domains, this sampling uncertainty is limited to a single bit. Single bit sampling uncertainty results in only one of two possible Gray coded addresses being sampled: the previous address or the new address. The uncertainty in the bit that is changing near a sampling clock edge directly corresponds to an uncertainty in whether the new value will be captured by the sampling clock edge or whether the previous value will be captured (and the new value may be captured by a subsequent sampling clock edge). Thus, there are no errors in sampling Gray coded pointers, just a matter of whether a change of pointer value occurs in time to be captured by a given sampling clock edge or whether it must wait for the next sampling clock edge to be registered. To do this transporting of 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` (represents by *r_sync_type*) and internal Gray code write pointer to clk_d (represented by *f_sync_type*). A value of one (1) indicates single-stage synchronization; a value of two (2) indicates double-stage synchronization; a value of three (3) indicates triple-stage synchronization; a value of four (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.

Note 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. Then, 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. 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.

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.

Note `fifo_empty_s` reflects the contents of the RAM module and the 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 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`. 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 changed in the pop domain. Refer to the timing diagrams for more information.

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 the perspective of the destination domain.

Full to Not Full Transitional Operation

When the FIFO is full (RAM and cache full), both `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 `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 the timing diagrams for more information.

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

Not Full to Full Transitional Operation

When the RAM is almost full (with respect to the destination domain) both `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 `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 `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 `error_s` and `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.

error_s Output

The `error_s` output signal indicates that a push request was seen while the `full_s` output was active high (an overrun error). 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.

error_d Output

The `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 three 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. 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.

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 information about `fifo_empty_s` and `fifo_word_cnt_s`, see "Behavior of fifo_empty_s and fifo_word_cnt_s" on page 24.

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 with respect to the RAM module 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 with respect to the RAM module 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.

full_s

The full_s output, active high, is synchronous to the clk_s input. The full_s output indicates to the push interface that the RAM module is full. During the final push, the rising edge of clk_s causes the last word to be pushed, and 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 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.

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". In the block diagram in Figure 1-2 on page 10, these signals are "rd_ptr_d" and "cache_inuse_d", respectively.

The "rd_ptr_d" and "cache_inuse_d" signals are 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 values were used for Figure 1-5 on page 25 and Figure 1-6 on page 26:

width = 32

ram_depth = 16

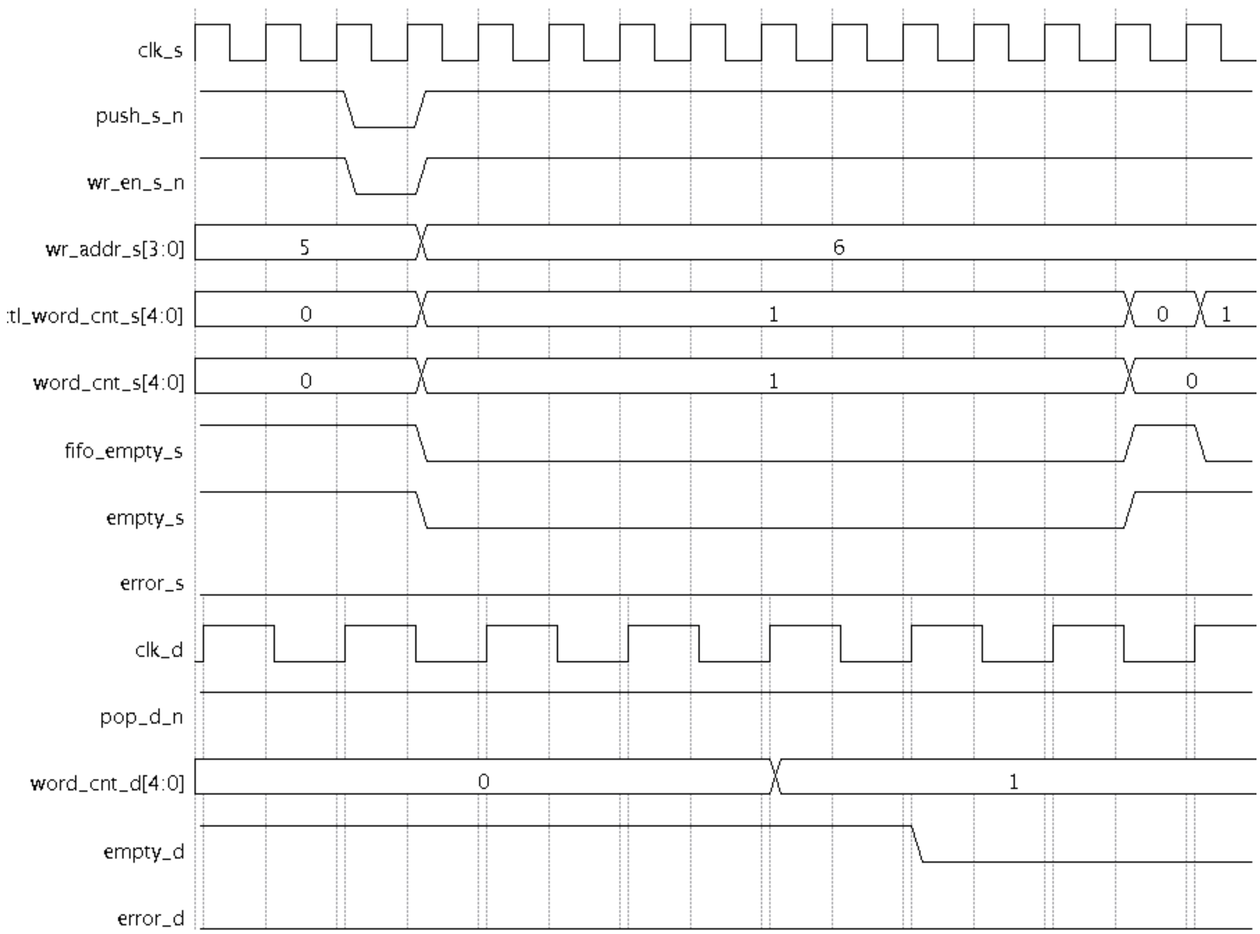
f_sync_type = 2

r_sync_type = 2

mem_mode = 0

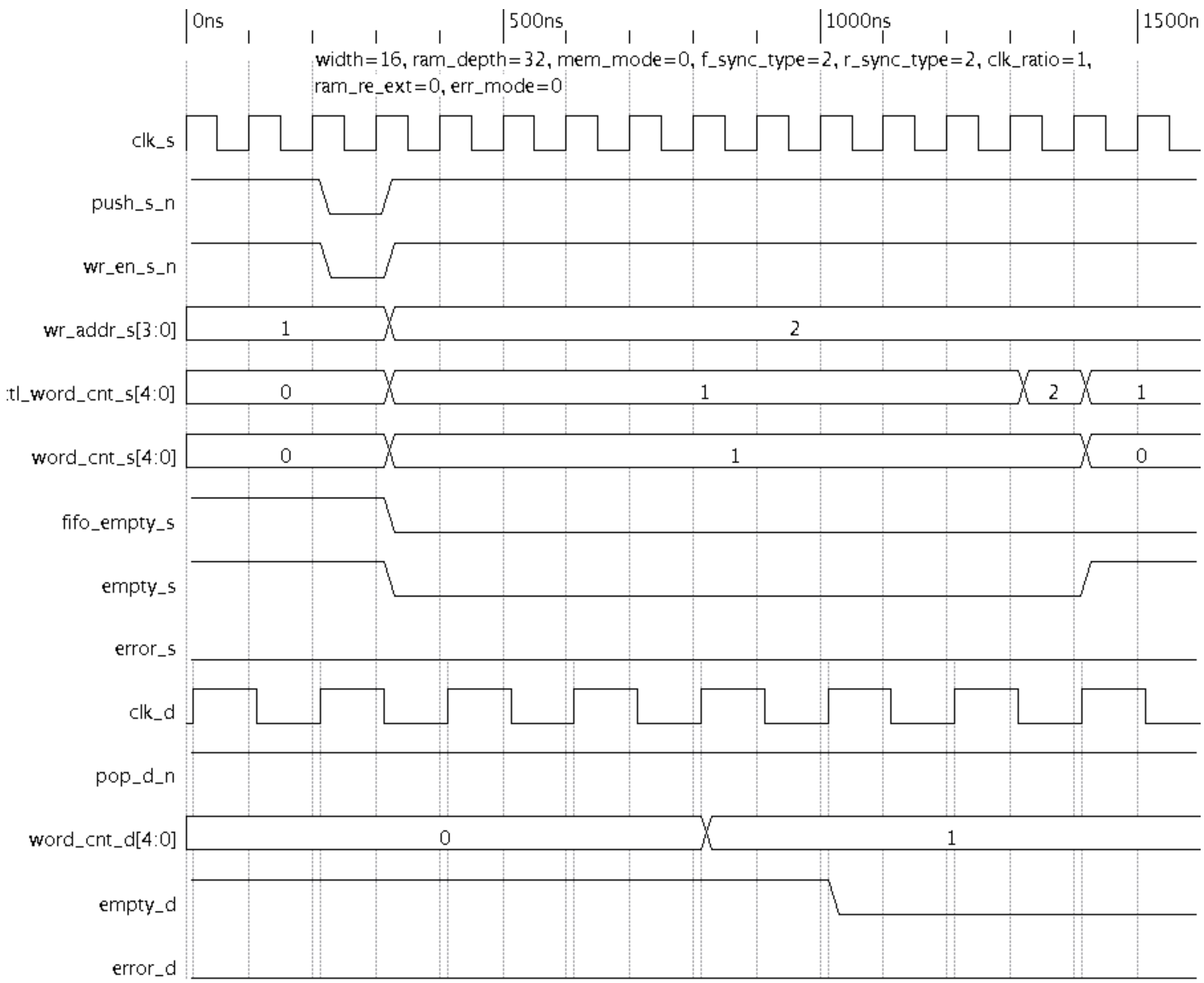
In Figure 1-5, the waveforms show a push request being made with `push_s_n` going to '0' for one `clk_s` cycle. Notice that no pop request is made throughout (`pop_n_d` stays at '1'). So, effectively, the FIFO has one valid word in it. As the pre-fetch to cache occurs (reading from the RAM), the read 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 1-5 Push Request Showing Instability On fifo_word_cnt_s (Example 1)



In Figure 1-6, 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 1-6 Push Request Showing Instability On fifo_word_cnt_s (Example 2)



Blind Pushing

Blind pushing is the operation of performing consecutive-cycle pushing without the risk of RAM overruns (overflows). The number of consecutive pushing cycles is predicated on the setting of level thresholds (`af_level_s` and/or `ae_level_s`) and the initiation of pushing is based on the state of the source domain status flags (`almost_full_s` and/or `almost_empty_s`, respectively, `half_full_s`, or `full_s`). In general, any pushing operation must rely on one or more of the provided source domain status flags (including `empty_s`) to know when pushing can begin and to prevent overrun. A common practice for implementing blind pushing operations is to use the `af_level_s` input value coupled with the `almost_full_s` status flag. For example, if the `af_level_s` is set to 2 and the source domain interface `almost_full_s` status flag is 0, then it would be safe to begin pushing (`push_s_n` to logic 0) consecutive operations of duration 3 (`af_level_s+1`) without overrunning the RAM after which the push request MUST be released (for example, `push_s_n` must go to logic 1).

empty_d

The `empty_d` output, active high, is synchronous to the `clk_d` input. `empty_d` indicates to the pop interface that the head of the cache does not contain relevant data. It does not mean, per se that the RAM module or other stages of the cache don'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 at the head of the cache pipeline.

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 with respect to the entire FIFO 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) then 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 1-2 on page 4. 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 with respect to the entire FIFO 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 1-2 on page 4. 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.

full_d

The `full_d` output, active high, is synchronous to the `clk_d` input. `full_d` indicates to the pop interface that the FIFO is 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 FIFO is controlled by the push 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`.

Blind Popping

Blind popping is the operation of performing consecutive-cycle popping without the risk of FIFO underruns (underflows). The number of consecutive popping cycles is predicated on the setting of level thresholds (`ae_level_d` and/or `af_level_d`) and the initiation of popping is based on the state of the destination domain status flags (`almost_empty_d` and/or `almost_full_d`, respectively, `half_full_d`, or `full_d`). In general, any popping operation must rely on one or more of the provided destination domain status flags (including `empty_d`) to know when popping can begin and to prevent underrun. A common practice for implementing blind popping operations is to use the ae_level_d input value coupled with the `almost_empty_d` status flag. For example, if the `ae_level_d` is set to 1 and the destination domain interface of the DW_fifoctl_2c_df calculates a `word_cnt_d` of 2 and the pre-fetch cache has a sufficient number of words installed to guarantee contiguously popped words, then the registered `almost_empty_d` status flag will go from 1 to 0. Once the `almost_empty_d` is at 0, pop operations can immediately start (`pop_d_n` of logic 0 sampled on the next rising-edge of `clk_d`). The duration of consecutive popping cycles (or *blind popping*) to guarantee no FIFO underruns would then be 2 (`ae_level_d+1`) after which the pop request MUST be released (i.e., `pop_d_n` must go to logic 1).

Note The word count outputs (`word_cnt_d` and `ram_word_cnt_d`) cannot be used for triggering blind popping operations.

Timing Waveforms

Figure 1-7 shows an 8-deep RAM configured with *mem_mode* as 0. The *mem_mode* setting implies that there is a one deep cache in the destination domain and, hence, defines a 9 deep FIFO. With the source clock (`clk_s`) slower by a factor of 2 compared to the destination clock (`clk_d`), 9 consecutive pushes (`push_s_n` asserted) can be issued without overrunning the RAM since the destination interface is able to read out from RAM at least one data packet.

Notice that after each `clk_s` cycle that samples `push_s_n` as 0, `word_cnt_s[3:0]` increments (`word_cnt_s[3:0]` represents the number of valid entries in RAM). The only exception is on the 6th push in which `word_cnt_s[3:0]` holds at a value of 5. This is due to the cache in the destination getting load with the first location in the RAM. Thus, freeing up that location and reducing the number of valid entries. So, if the 6th push did not happen, the `word_cnt_s[3:0]` would have been 4 due to the loading of the first RAM location to cache (which increments RAM read address from 0 to 1). But with the 6th push occurring where it did, the value of **`word_cnt_s[3:0]`** stays at '5'.

Furthermore, during the pushing activity the destination domain logic detects that the cache is empty and the RAM becomes non empty as indicated by the signal `ram_word_cnt_d[3:0]` registering a value 1 (non-zero value). This detection initiates are read operation of RAM indicated by ram_re_d_n going 'low' which on the next `clk_d` cycle causes the `ram_word_cnt_d[3:0]` change from 1 to 0 for only one `clk_d` cycle before going back to 1. This transition to '0' of `ram_word_cnt_d[3:0]` indicates the time at which the cache was loaded with the contents of the first RAM location and as a result the RAM becomes empty (from the destination domain perspective) for one `clk_d` cycle.

After 9 consecutive pushes are performed, a 10th push is issued that causes an overrun condition of the RAM indicated by `error_s` going to 1. `error_s` only stays at 1 for a single `clk_s` cycle in this case because the *err_mode* parameter is set to 0. When an overrun occurs, `wr_addr_s[2:0]` and the word counts do not change.

Note

1. `data_s[7:0]` is provided only as a reference to what is being written into the RAM during each `push_s_n` assertion. This timing diagram implies that a RAM device is connected to DW_fifoctl_2c_df in which `rd_data_d[7:0]` is supplied.
2. The signal Actual_FIFO_WordCnt[3:0] at the bottom of the waveform is provided only as reference and it does not exist in the DW_fifoctl_2c_df component. Actual_FIFO_WordCnt[3:0] is meant to provide an idealized value of the number valid entries in the FIFO at any one time independent of either interface.

Figure 1-7 Push Until Full and Error with clk_s slower than clk_d (RAM depth a power of 2)

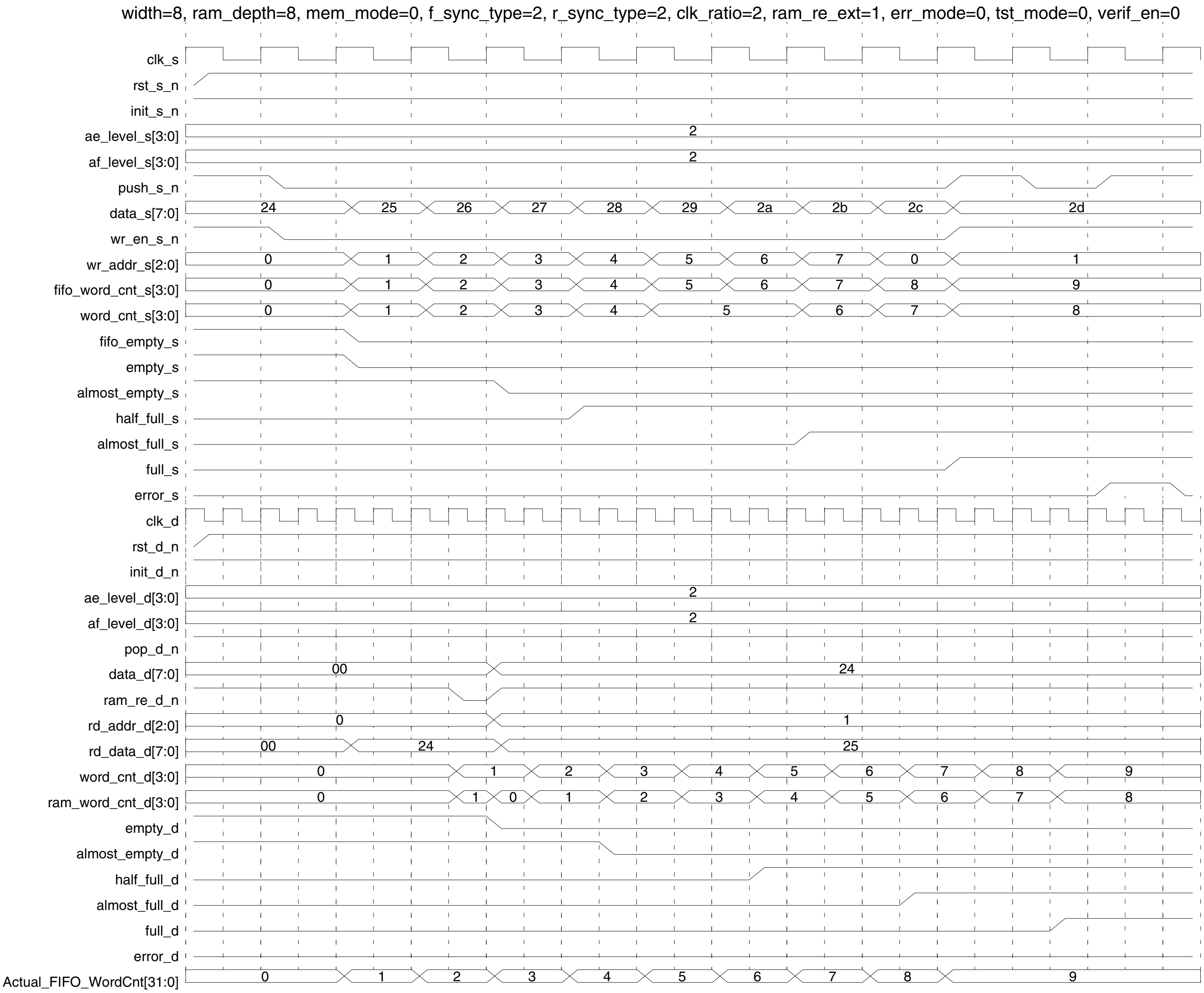


Figure 1-8 shows popping activity from the FIFO full condition to the FIFO empty condition. As long as `empty_d` is a 0, `pop_d_n` is asserted (low). `word_cnt_d[3:0]` represents the number of valid entries in the FIFO. In this example, before popping is performed the `word_cnt_d[3:0]` value is 9 and `empty_d` is 0. Therefore, popping 9 consecutive `clk_d` cycles empties the FIFO as shown in the waveform.

A 10th pop is performed when `empty_d` is 1 which causes an underrun of the cache and the `error_d` status signal to go high for a `clk_d` cycle. `error_d` is only active for one `clk_d` cycle because the parameter *err_mode* is set to 0.

Attention

1. `data_s[7:0]` is provided only as a reference to what is being written into the RAM during each `push_s_n` assertion. This timing diagram implies that a RAM device is connected to DW_fifoctl_2c_df in which `rd_data_d[7:0]` is supplied.
2. The signal, Actual_FIFO_WordCnt[3:0], at the bottom of the waveform is provided only as reference and it does not exist in the DW_fifoctl_2c_df component. Actual_FIFO_WordCnt[3:0] is meant to provide an idealized value of the number valid entries in the FIFO at any one time independent of either interface.

Figure 1-8 Pop Until Empty and Error with clk_s slower than clk_d (RAM depth a power of 2)

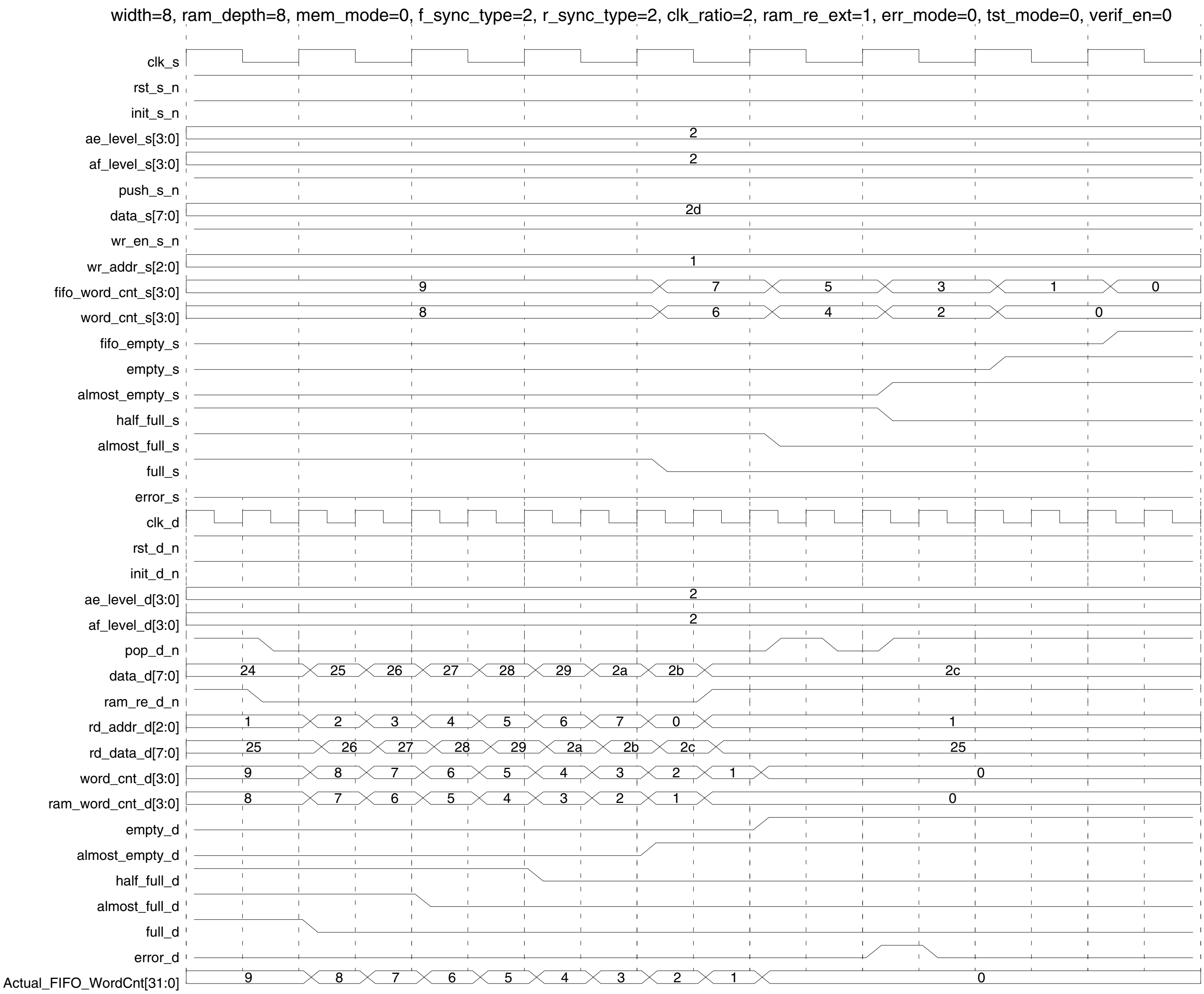


Figure 1-9 shows how the internal synchronization of pointers traverse between domains during single-word push and pop operations.

All the signals listed in the waveform with the suffix "REF" represent internal signals of DW_fifoctl_2c_df and are not visible at the component ports. The signals which names include "SYNC_Sx", where "x" is 1st stage or 2nd stage of synchronization, are internal synchronize representations of the pointers.

The pointers wr_ptr_s_REF[3:0] and rd_ptr_s_SYNC_S2_REF[3:0] are used to determine word counts and status for the source (push) domain. The pointers wr_ptr_d_SYNC_S2_REF[3:0] and rd_ptr_d_REF[3:0] are used to determine word counts and status for the destination (pop) domain.

Attention For clarity, the internal synchronized reference pointers represented here are not Gray coded. In the actual synthetic design Gray coded pointers are used in synchronization across clock boundaries.

Figure 1-9 Single Word Push/Pop Timing with Double Synchronization

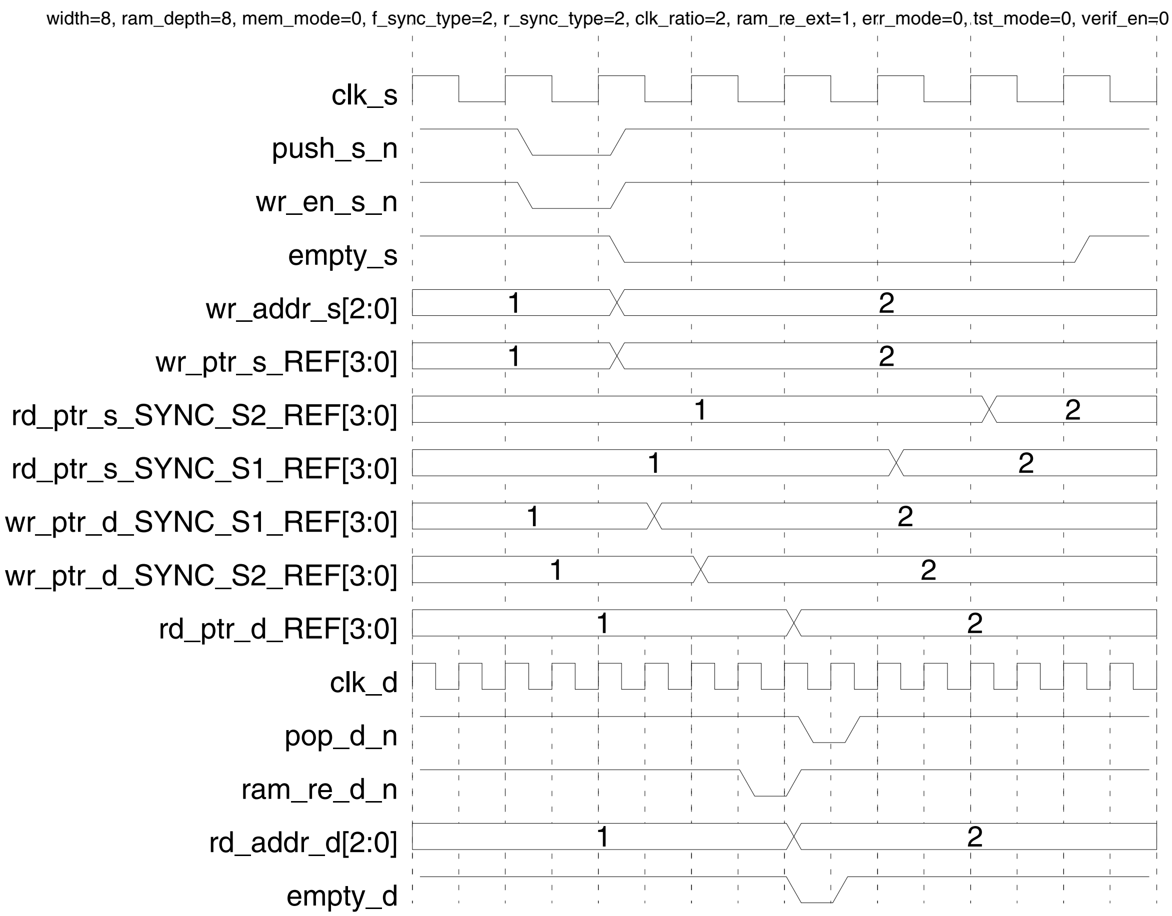


Figure 1-10 is similar to Figure 1-7 except that the parameters *ram_depth*, *mem_mode*, and *err_mode* have been changed.

With the combination of *ram_depth* being 6 and *mem_mode* set to 3, the FIFO depth derives to be 9. The *mem_mode* setting of 3 indicates that the RAM that is being used with the DW_fifoctl_2c_df contains re-timing of read address and control signals AND re-timing of the RAM data output. Having RAM configure this way requires a cache depth of 3 (automatically derived). So, with the RAM depth being 6 and the cache depth being 3, the overall FIFO depth is 9.

The parameter *err_mode* in this timing diagram is set to 1. Therefore, in the event of the overrun condition of the RAM, the `error_s` status flag only asserts upon occurrence of the error. If no overrun condition is present, then `error_s` de-asserts on the next clock cycle.

Attention

1. `data_s[7:0]` is provided only as a reference to what is being written into the RAM during each `push_s_n` assertion. This timing diagram implies that a RAM device is connected to DW_fifoctl_2c_df in which `rd_data_d[7:0]` is supplied.
2. The signal, Actual_FIFO_WordCnt[3:0], at the bottom of the waveform is provided only as reference and it does not exist in the DW_fifoctl_2c_df component. Actual_FIFO_WordCnt[3:0] is meant to provide an idealized value of the number valid entries in the FIFO at any one time independent of either interface.

Figure 1-10 Push Until RAM Full and Error with clk_s slower than clk_d (RAM depth a non-power of 2)

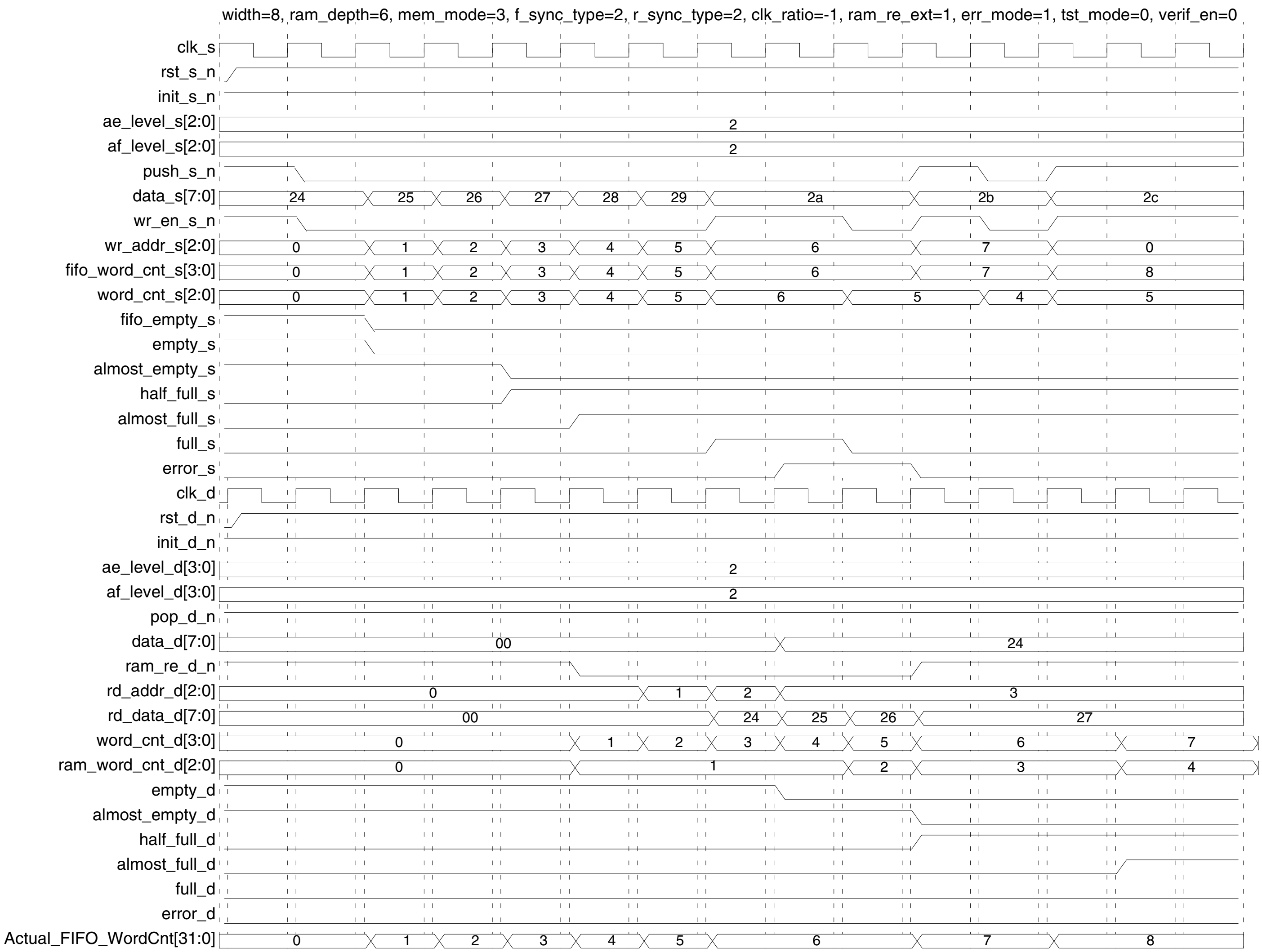


Figure 1-11 is similar to Figure 1-8 except that the parameters *ram_depth*, *mem_mode*, and *err_mode* have been changed.

With the combination of *ram_depth* being 6 and *mem_mode* set to 3, the FIFO depth derives to be 9. The *mem_mode* setting of 3 indicates that the RAM that is being used with the DW_fifoctl_2c_df contains re-timing of read address and control signals AND re-timing of the RAM data output. Having RAM configure this way requires a cache depth of 3 (automatically derived). So, with the RAM depth being 6 and the cache depth being 3, the overall FIFO depth is 9.

The parameter *err_mode* in this timing diagram is set to 0. Therefore, in the event of the underrun condition of the cache, the `error_d` status flag asserts and stays asserted until either a system reset or coordinated clearing sequence of the two domains.

Attention

1. `data_s[7:0]` is provided only as a reference to what is being written into the RAM during each `push_s_n` assertion. This timing diagram implies that a RAM device is connected to DW_fifoctl_2c_df in which `rd_data_d[7:0]` is supplied.
2. The signal, Actual_FIFO_WordCnt[3:0], at the bottom of the waveform is provided only as reference and it does not exist in the DW_fifoctl_2c_df component. Actual_FIFO_WordCnt[3:0] is meant to provide an idealized value of the number valid entries in the FIFO at any one time independent of either interface.

Figure 1-11 Pop Until Empty and Error with clk_s slower than clk_d (RAM depth a non-power of 2)

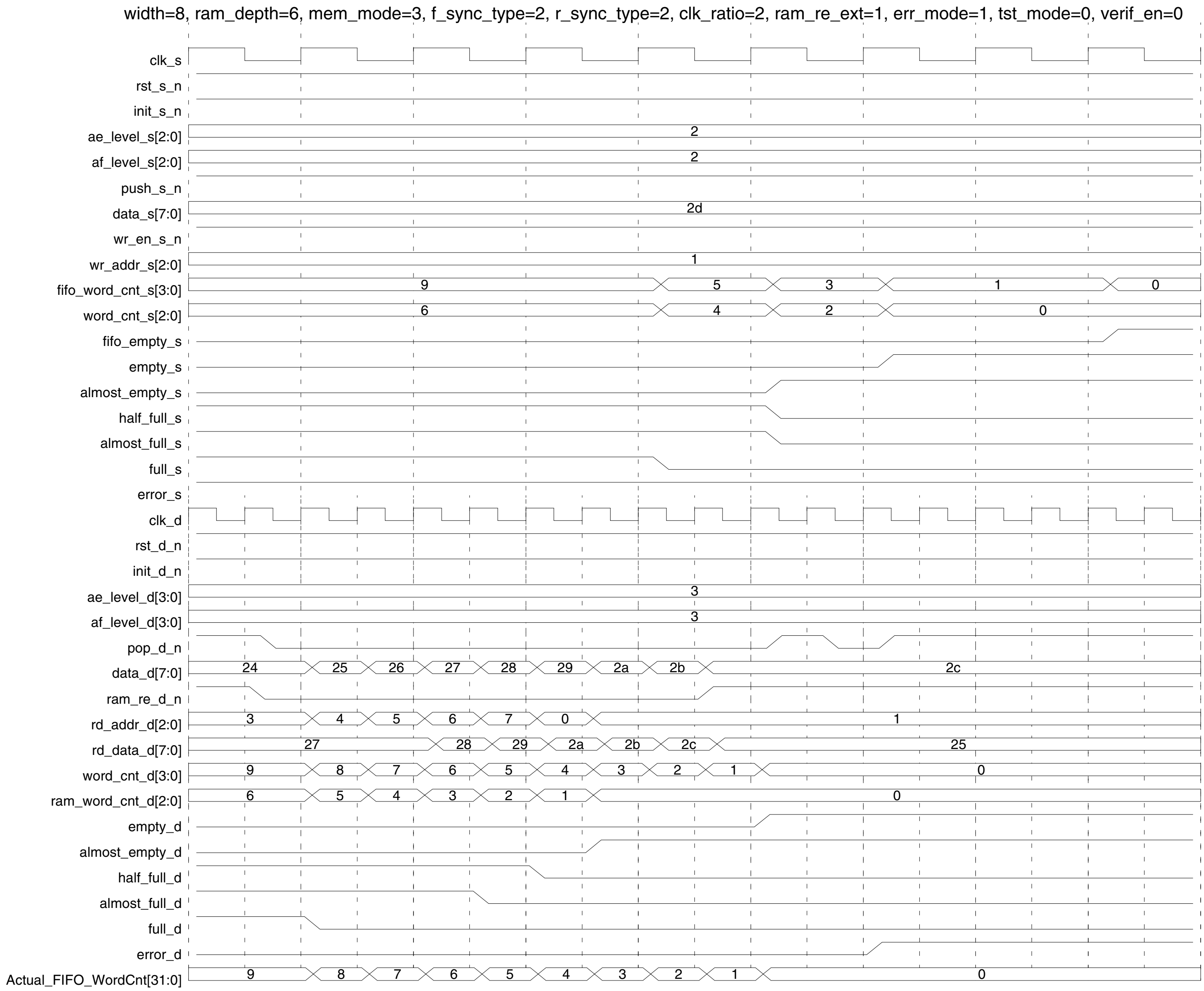


Figure 1-12 shows the initiation of the coordinated clearing sequence from the source domain via assertion of clr_s. In this case clr_s is a single clk_s cycle, but the length of clr_s assertions are not restricted. The clearing-related signals are grouped at the bottom of the timing diagram.

When clr_s is asserted it gets synchronized at the destination domain (based on *f_sync_type*) activates the 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's implied that no destination domain popping will be occurring 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 destination domain read pointer (used to calculate word counts and status flags) gets initialized to 0. The destination read pointer, which is always being synchronized to the source domain in the form of rd_ptr_s_SYNC_S2[3:0] (reference only signal) goes from 9 to 0. This change at the source domain compared to the unchanged source domain write pointer wr_ptr_s_REF[3:0] could cause word_cnt_s[3:0] and fifo_word_cnt[3:0] to change from 0 to 9 which, in turn, could produce state changes in output status signals fifo_empty_s, empty_s, almost_empty_s, half_full_s, and almost_full_s. In this example, no unwanted and incorrect state of these outputs occurs. But if the clk_d rate is faster than the clk_s rate beware that incorrect status may be reported during the clearing process.

Yet, 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.

Figure 1-12 clr_s Initiated Clearing Sequence

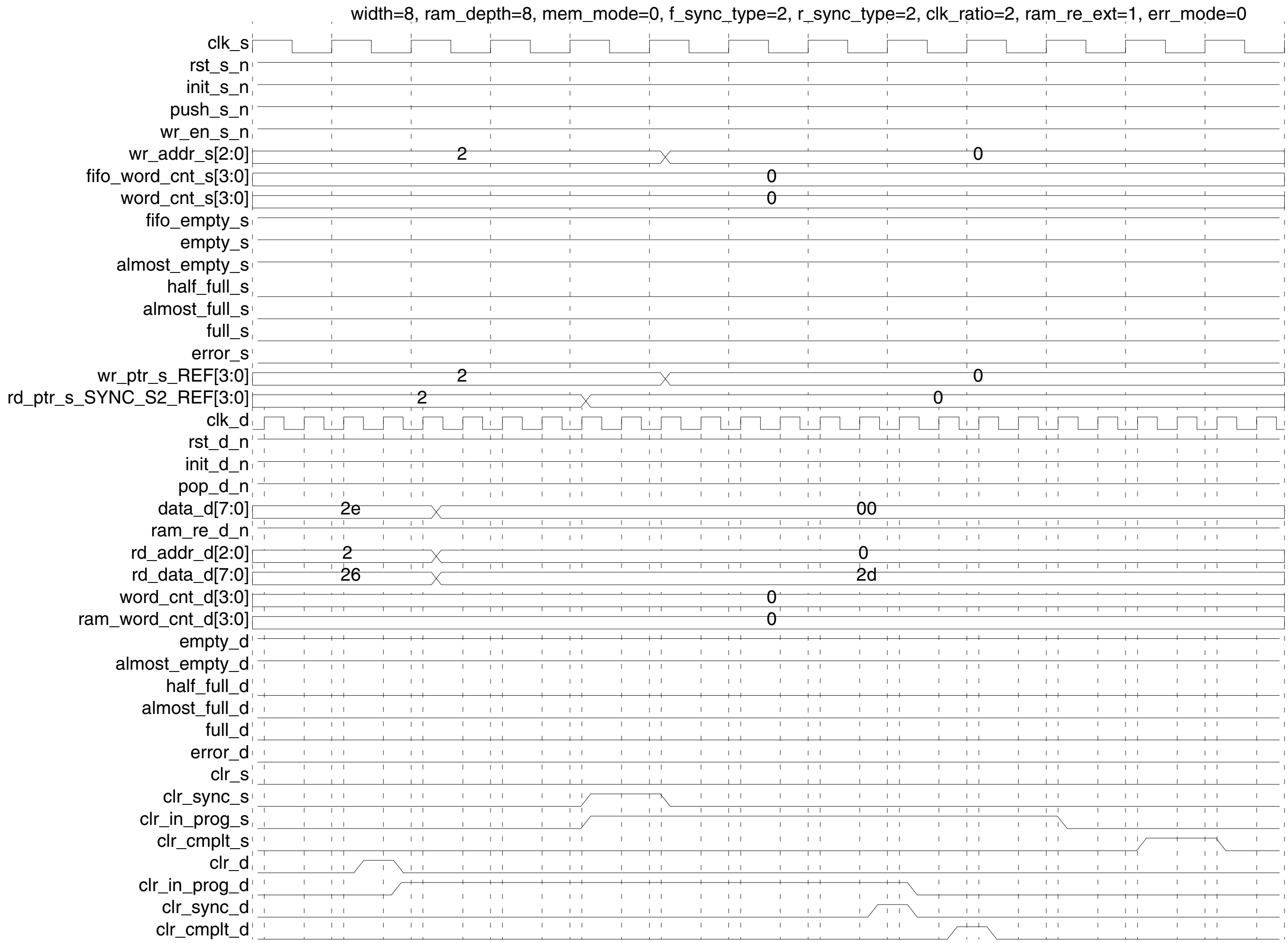


Figure 1-13 describes the timing from an initiated `clr_d` pulse. In this case, `clr_d` is a single `clk_d` cycle, but the length of `clr_d` assertions are not restricted. The `clr_d` initiated clearing sequence is similar to the `clr_s` initiated clearing sequence with fewer synchronization feedback paths from beginning to completion.

When '`clr_s`' is asserted it triggers the `clr_in_prog_s` 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 feedback, 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 feedback 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 destination domain read pointer (used to calculate word counts and status flags) gets initialized to 0. The destination read pointer, which is always being synchronized to the source domain in the form of rd_ptr_s_SYNC_S2[3:0] (reference only signal) goes from 2 to 0. This change at the source domain compared to the unchanged source domain write pointer wr_ptr_s_REF[3:0] causes `word_cnt_s[3:0]` and `fifo_word_cnt[3:0]` to change from 0 to 2 which, in turn, produces state changes in output status signals `fifo_empty_s` and `empty_s`. In this example, no temporary incorrect reporting of the source status outputs are made. But when cases of a big disparity of `clk_d` rate to `clk_s` rate exists, when `clk_d` has a much shorter period than `clk_s`, there's a possibility of incorrect source domain status to be reported during the clearing sequence.

Figure 1-13 clr_d Initiated Clearing Sequence

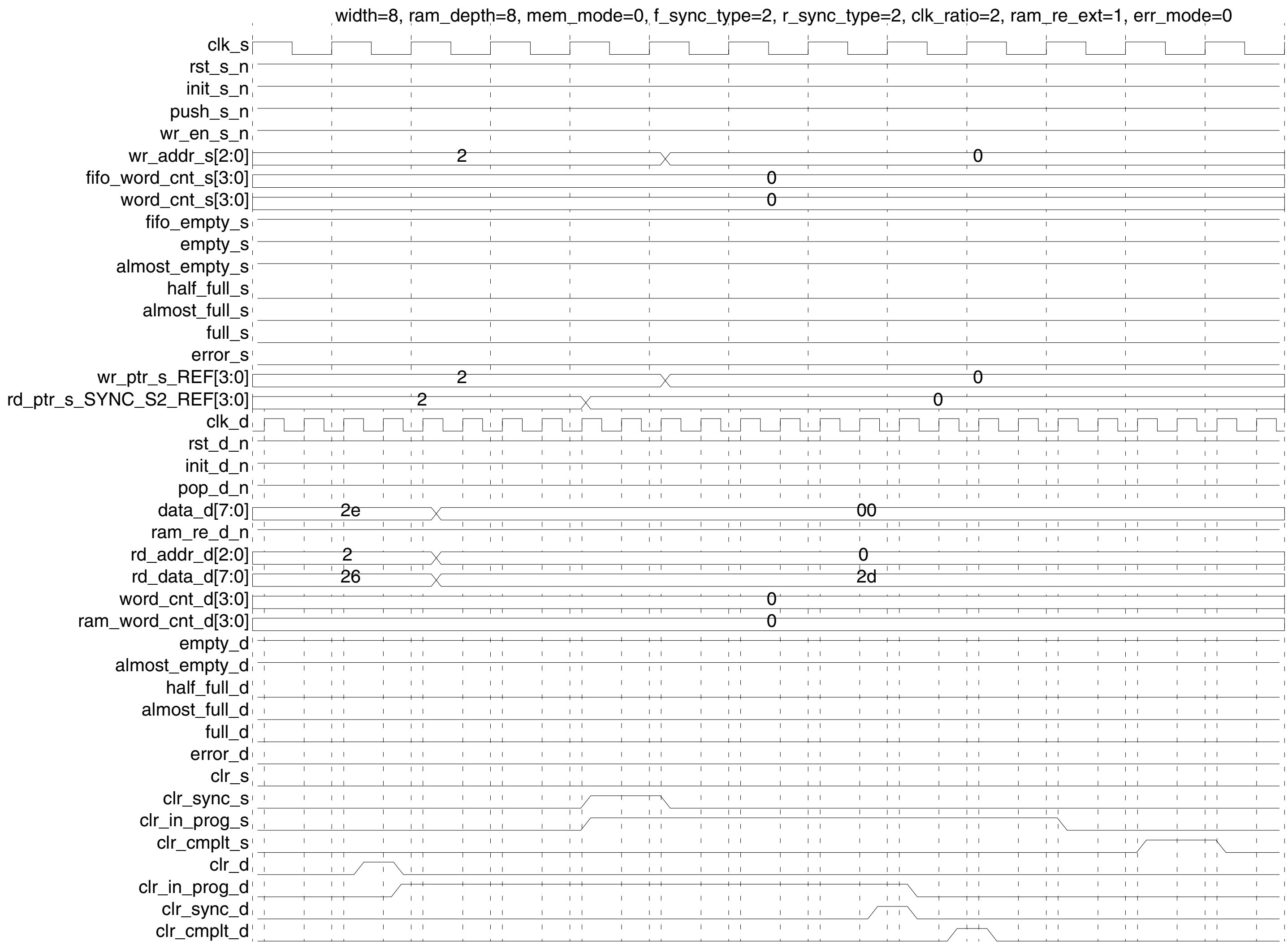


Figure 1-14 shows an initiation of a `clr_s` where its duration is much longer than one `clk_s` cycle. From this, the behavior of the `clr_in_prog_s` and `clr_in_prog_d` flags are sustained longer than those seen in Figure 1-12 on page 38 in which `clr_s` was only asserted a single `clk_s` cycle.

Figure 1-14 Initiation of clr_s with duration much longer than one clk_s cycle.

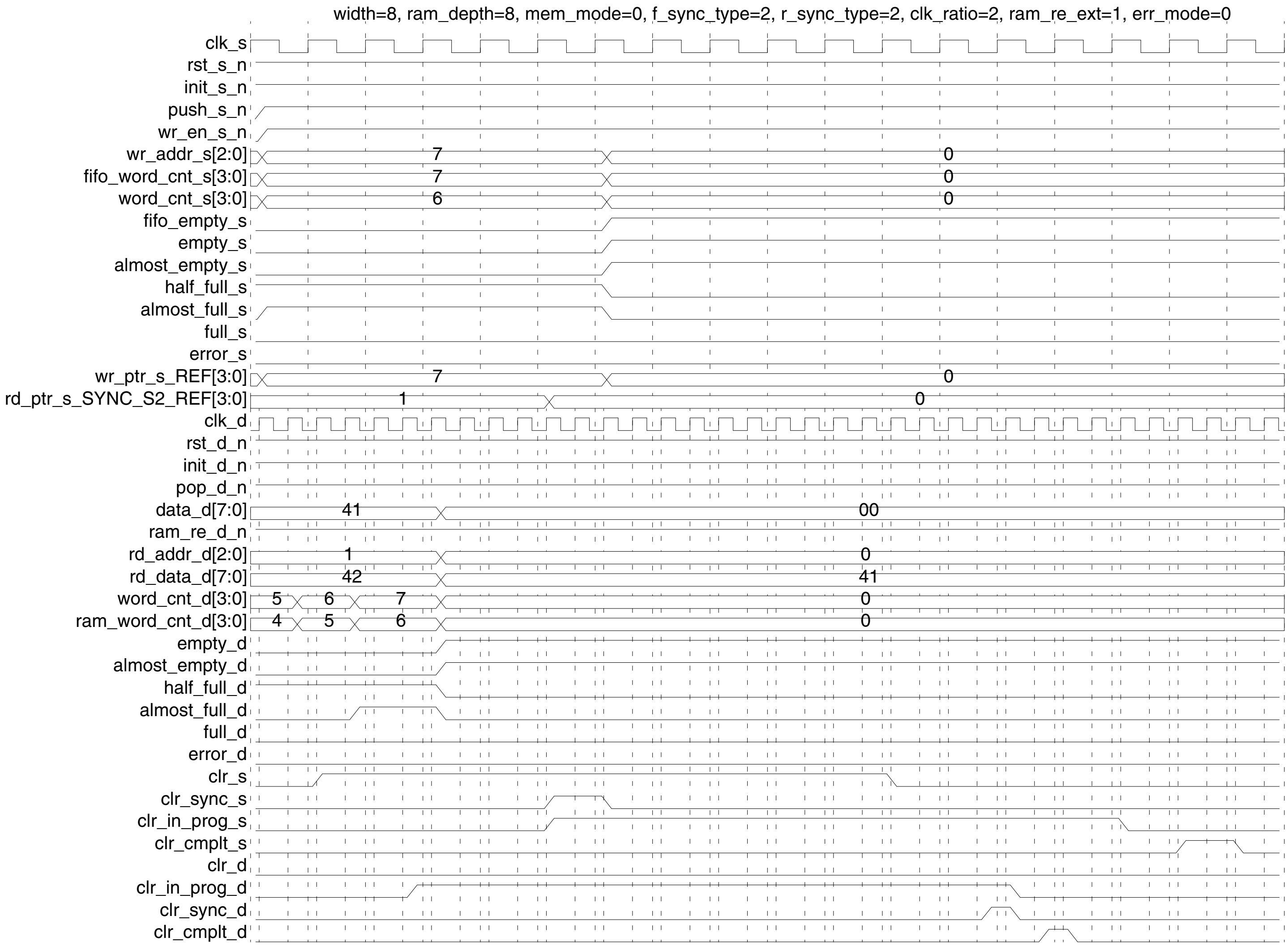


Figure 1-15 shows an example of a sequence where `rst_s_n` and `rst_d_n` are asserted. In this case, `clk_d` is faster than `clk_s` with `rst_d_n` asserted first followed by the assertion of `rst_s_n` within *r_sync_type*+1 `clk_s` cycles (*r_sync_type* is 2 in this example). Note that the word counts and addresses go to 0 and the status flags settle into the appropriate state.

Figure 1-15 Example of Asynchronous System Reset

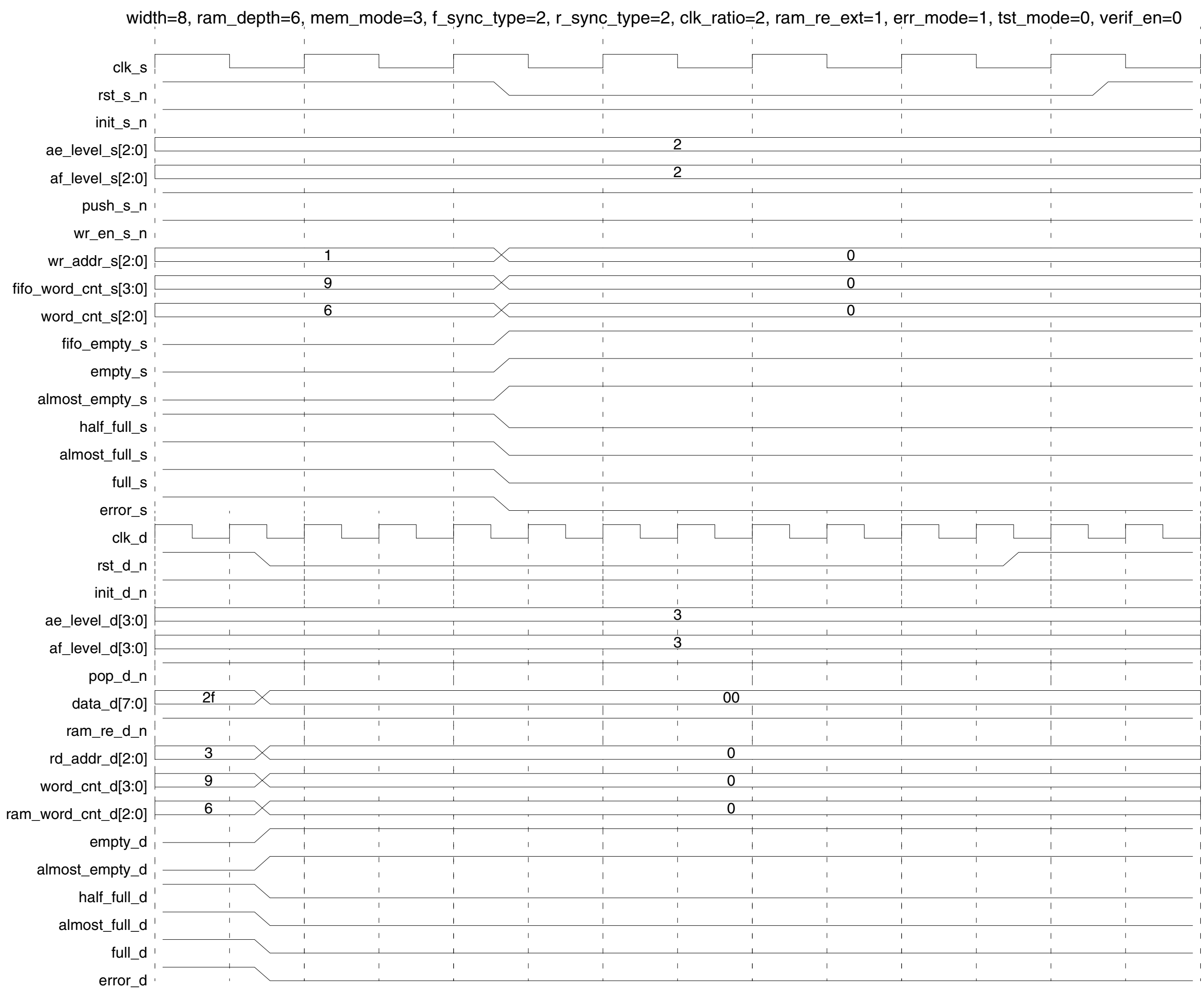
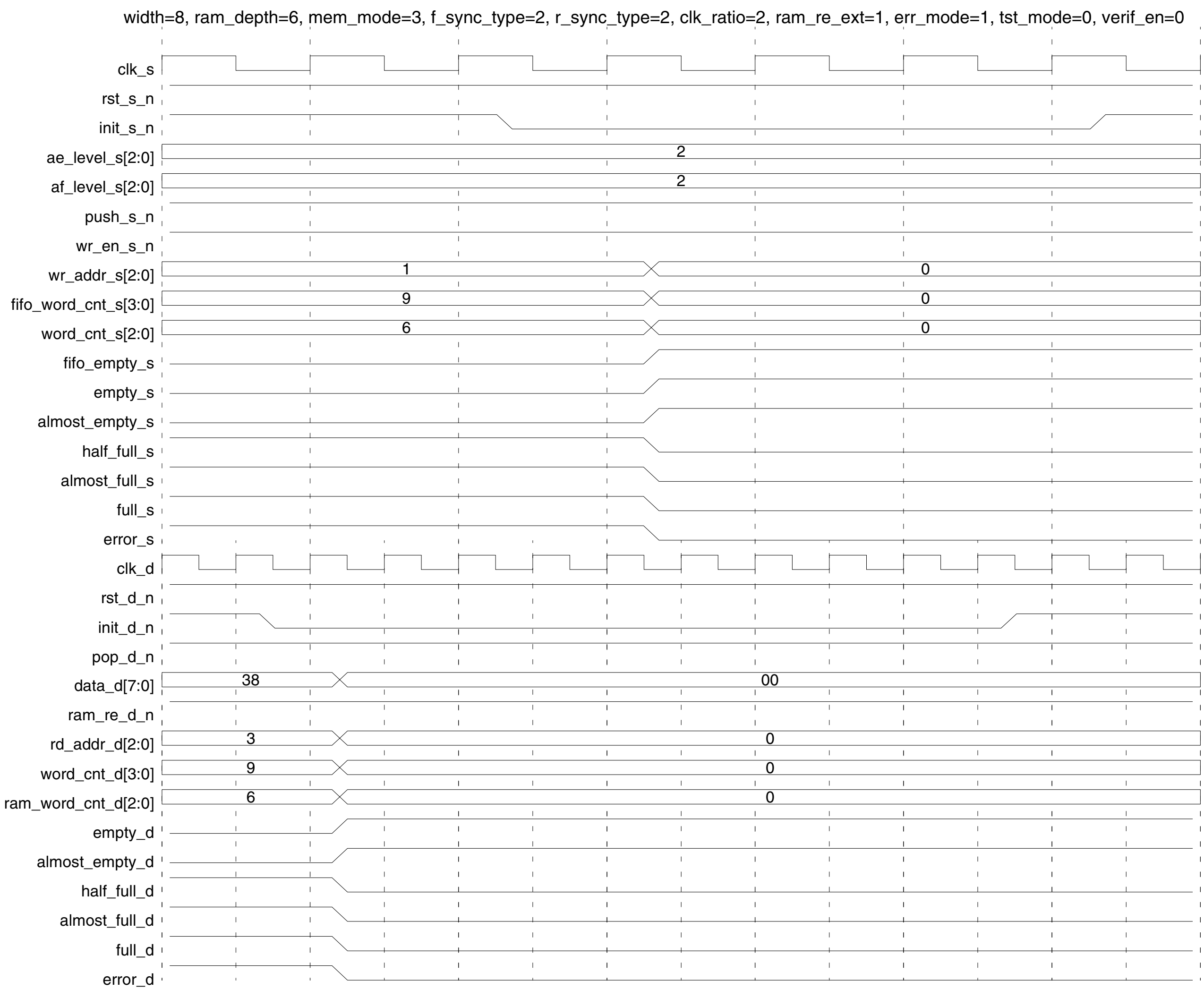


Figure 1-16 shows an example of a sequence where `init_s_n` and `init_d_n` are asserted. In this case, `clk_d` is faster than `clk_s` with `init_d_n` asserted first followed by the assertion of `init_s_n` within *r_sync_type*+1 `clk_s` cycles (*r_sync_type* is 2 in this example). Note that the word counts and addresses go to 0 and the status flags settle into the appropriate state.

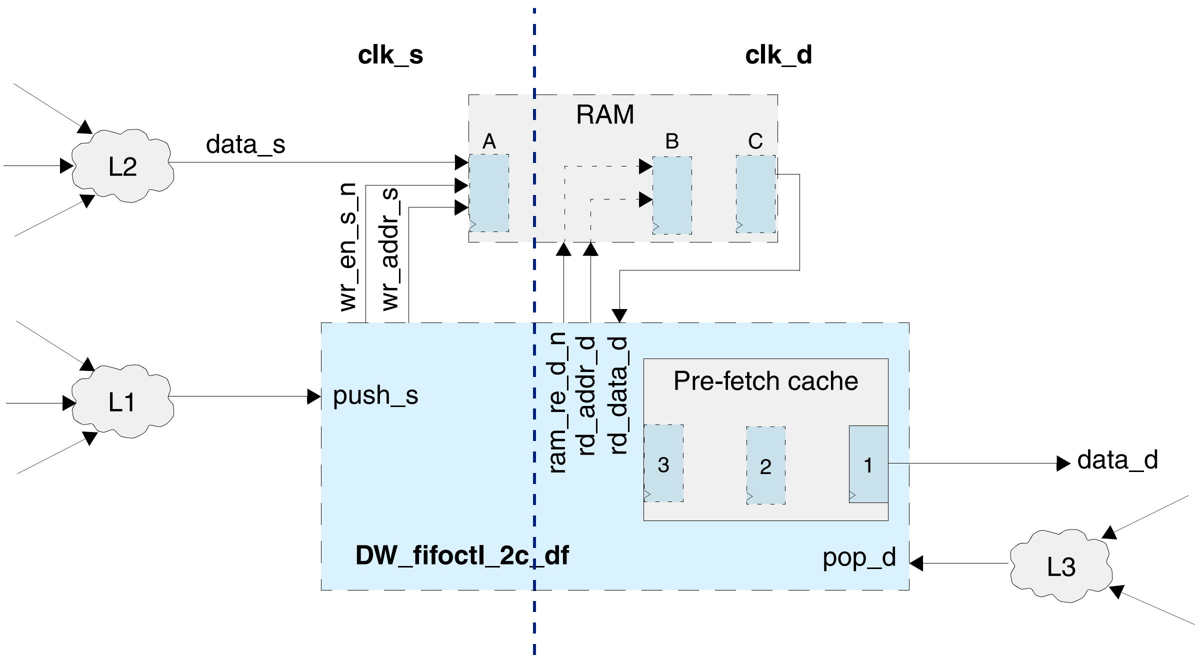
Figure 1-16 Example of Synchronous System Reset



Considerations for Setting the *mem_mode* Parameter

Depending on the logic being supplied to the RAM that interfaces with the DW_fifoctl_2c_df and the RAM architecture, the setting of the mem_mode parameter is determined. Refer to Figure 1-17 for the following discussion.

Figure 1-17 mem_mode Settings Based on System Design



If there are L1 and/or L2 logic clouds or delays that cause either `push_s` and/or `data_s`, respectively, to be late-arriving, then there potentially would be the need to re-time the write signals into the RAM as denoted by register A. If register A of the RAM is required in the design, the *mem_mode* parameter should be set to 4 or greater depending on the existence of registers B and C in the RAM.

Similarly, if there is an L3 logic cloud or delays that cause `pop_d` to be late-arriving enough to require a register B in the RAM, then *mem_mode* should be set to 2, 3, 6, or 7 depending on the existence of registers A and C in the RAM.

If delay is minimal through L1, L2, and L3 which does not require either register A or B in the RAM, then depending on the internal delay of the datapath through the RAM to its output a re-timing register C may or may not be needed. If needed, then the *mem_mode* must be set to either 1 (register C is exists) or 0 (register C does not exists).

The following is a table that lists the all possible supported RAM architectures and the required *mem_mode* setting along with the resulting structure of the pre-fetch cache.

Table 1-11 RAM Configuration Determines *mem_mode* Setting

Does RAM Register Exist?				
A	**B**	**C**	***mem_mode* Setting**	**Pre-fetch Cache Structure**
no	no	no	0	register 1
no	no	yes	1	register 1 and 2
no	yes	no	2	register 1 and 2
no	yes	yes	3	register 1, 2 and 3
yes	no	no	4	register 1
yes	no	yes	5	register 1 and 2
yes	yes	no	6	register 1 and 2
yes	yes	yes	7	register 1, 2 and 3

Related Topics

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

HDL Usage Through Component Instantiation - VHDL

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

entity DW_fifoctl_2c_df_inst is
      generic (
        inst_width           : POSITIVE := 8;
        inst_ram_depth       : POSITIVE := 8;
        inst_mem_mode        : NATURAL := 5;
        inst_f_sync_type     : NATURAL := 2;
        inst_r_sync_type     : NATURAL := 2;
        inst_clk_ratio       : INTEGER := 7;
        inst_ram_re_ext      : NATURAL := 1;
        inst_err_mode        : NATURAL := 0;
        inst_tst_mode        : NATURAL := 0;
        inst_verif_en        : NATURAL := 2;
        inst_clr_dual_domain : 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;
        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);
        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 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;
        full_s_inst          : out std_logic;
        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)-1 downto
0);
        inst_af_level_d        : in std_logic_vector(bit_width(inst_ram_depth+1)-1 downto
0);
        inst_pop_d_n           : in std_logic;
        inst_rd_data_d         : in std_logic_vector(inst_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_width-1 downto 0);
        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 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;
        error_d_inst           : out std_logic;
        inst_test              : in std_logic
        );
    end DW_fifoctl_2c_df_inst;


architecture inst of DW_fifoctl_2c_df_inst is
begin
    -- Instance of DW_fifoctl_2c_df
    U1 : DW_fifoctl_2c_df
    generic map ( width => inst_width, ram_depth => inst_ram_depth,
       mem_mode => inst_mem_mode, f_sync_type => inst_f_sync_type,
       r_sync_type => inst_r_sync_type, 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,
       clr_dual_domain => inst_clr_dual_domain )
    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, 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,
```

```
    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,
    full_s => full_s_inst, error_s => 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,
    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, error_d => error_d_inst, test => inst_test );

end inst;

-- Configuration for use with a VHDL simulator
-- pragma translate_off
library DW03;
configuration DW_fifoctl_2c_df_inst_cfg_inst of DW_fifoctl_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_fifoctl_2c_df_cfg_sim_ms;
end for;
  end for; -- inst
end DW_fifoctl_2c_df_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_fifoctl_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, clr_sync_s_inst, clr_in_prog_s_inst,
clr_cmplt_s_inst,
          wr_en_s_n_inst, wr_addr_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,
full_s_inst,
          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,
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, error_d_inst, inst_test );

parameter width = 8;
parameter ram_depth = 8;
parameter mem_mode = 5;
parameter f_sync_type = 2;
parameter r_sync_type = 2;
parameter clk_ratio = 3;
parameter ram_re_ext = 1;
parameter err_mode = 1;
parameter tst_mode = 0;
parameter verif_en = 1;
parameter clr_dual_domain = 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;

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 [`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 full_s_inst;
output 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 [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 [width-1:0] data_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 error_d_inst;

input inst_test;

    // Instance of DW_fifoctl_2c_df
    DW_fifoctl_2c_df #(width, ram_depth, mem_mode, f_sync_type, r_sync_type, clk_ratio,
ram_re_ext, err_mode, tst_mode, verif_en, clr_dual_domain)
```

```
      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), .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),
.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), .full_s(full_s_inst), .error_s(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),
.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), .error_d(error_d_inst), .test(inst_test) );

endmodule
```

Revision History

Date	Release	Updates
July 2017	M-2016.12-SP5	■ For STAR 9001209980, corrected width of the `wr_addr_s` port ■ Added this Revision History table

Copyright Notice and Proprietary Information

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