Using FIFOs in Delta39K™ CPLDs

Introduction

The purpose of this application note is to provide instruction for all aspects of implementing synchronous First-In, First-Out buffers (FIFOs) in Delta39K™ Complex Programmable Logic Devices (CPLDs). The Delta39K is a family of high density CPLDs containing embedded SRAM. Topics of discussion will include a description of FIFO operation, static timing analysis, expanding FIFOs across multiple blocks of memory, interfacing the FIFO through the CPLDs routing channels, software support, and a design example.

FIFOs are ideally suited to applications that require data to be buffered between two buses. As the name implies, data is read out of the FIFO in the same order as it was written. FIFO reads and writes can occur asynchronously to each other because FIFOs are built with dual-ported memory cells. A FIFO's largest benefit is its ability to pass data between two data buses that are asynchronous to one another. This includes buses running at different rates, as well as same rate buses whose clocks are generated from different sources.

FIFOs are used in systems that receive data faster than it can be processed. Typical disciplines which employ FIFOs are telecommunications, networking, mass storage and video. The two basic classes of FIFOs are asynchronous FIFOs and synchronous FIFOs. Designers prefer synchronous FIFOs to their asynchronous predecessors due to their speed and ease of operation. Asynchronous FIFOs require read and write pulses to be generated, which is difficult to do and timing critical at high speed. By comparison, synchronous FIFOs are operated by a free-running clock, which makes them easier to use at high speed.

Delta39K CPLDs contain true dual-ported memory cells from which FIFOs can be implemented. Read and write address pointers independently track the sequentially stored data and update at valid FIFO operations. Since the logic required to make a synchronous FIFO is included within each dual-port memory block, no general purpose logic (such as status flags or expansion logic) is used to create the FIFO. In other words, instantiating a FIFO incurs no resource overhead.

Due to these factors, synchronous FIFOs can be easily and efficiently implemented in Delta39K CPLDs. In addition to FIFO resources, there exists a large amount of general purpose logic and single port SRAM memory in Delta39K CPLDs. This additional logic and memory allows processing of the synchronous FIFO data stream. The Delta39K's ability to realize efficient FIFOs and make available any necessary logic to process the data make it an ideal one chip solution for many FIFO applications.

Delta39K FIFO Implementation

Channel Memory Blocks

The Delta39K CPLD family contains three main architectural components: logic block clusters, channel memory blocks and I/O blocks. The logic block clusters and channel memory blocks are arranged in rows and columns connected together by a series of horizontal and vertical routing channels. Each cluster contains 128 macrocells divided into 8 logic blocks together with two 8192-bit single port cluster memory blocks totalling 16,384 bits of user-accessible SRAM.

One block of channel memory is placed at every intersection of these horizontal and vertical routing channels. Each channel memory block is 4096 bits in size and can be configured to be used as asynchronous or synchronous dual-port RAM, or synchronous FIFO. The channel memory organization is configurable in block sizes of 4Kx1, 2Kx2, 1Kx4 and 512x8. This application note focusses on configuring and using these channel memory blocks as synchronous FIFOs.

FIFO Ports

Figure 1 shows the internal input and output ports of a FIFO memory in a Delta39K CPLD. The input port, Data, is controlled by a write clock (writeclock) and an active HIGH write enable signal (ENW). Writeclock is selected from one of the four global clocks or a local clock supplied by either the horizontal or vertical routing channel. When ENW is asserted, data is written into the FIFO on the rising edge of writeclock. While ENW is held active, data is continually written into the FIFO on each rising edge of writeclock. The output port, Q, is controlled in a similar manner by a read clock (readclock) and an active HIGH read enable signal (ENR). Readclock and writeclock may be tied together for single-clock operation or the two clocks may be run independently for asynchronous read/write applications.

Flag Operation

In addition, there are three output flags that indicate the current status of the FIFO: $\overline{EF}$, $\overline{HF}$ and $\overline{PAFE}$. These flags should be used by user FIFO control logic to prevent writing when full or reading when empty. The $\overline{EF}$ flag is active LOW and indicates that the FIFO is either completely full or completely empty. The $\overline{HF}$ flag is active LOW and indicates that the FIFO is greater than or equal to half full. The $\overline{PAFE}$ flag is active LOW and indicates that the FIFO is currently either almost-full

Table 1. Decoding FIFO Status from Flag Outputs

$\overline{EF}$	$\overline{PAFE}$	$\overline{HF}$	FIFO Status
0	0	0	Full
0	0	1	Empty
1	0	0	Almost Full
1	0	1	Almost Empty
1	1	0	Greater than Half Full
1	1	1	Less than Half Full

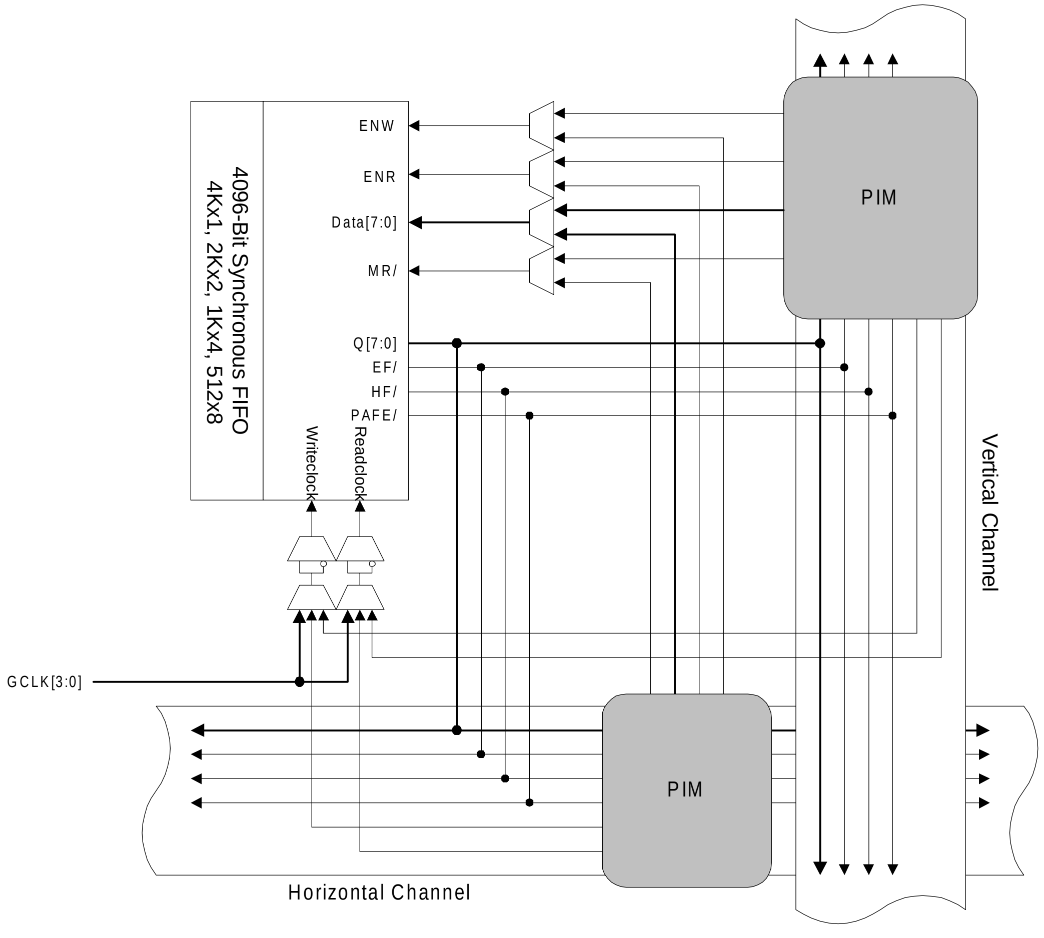


Figure 1. FIFO Block Diagram showing Interface to Routing Channels

or almost-empty. The combination of the three flags can be decoded into the six valid states shown in *Table 1*.

The $\overline{\text{PAFE}}$ range is programmable and can be set for any distance from full or empty using an internal 11-bit configuration register. This register is initialized when the device is configured at power-up. During a master reset cycle, the flag logic will read the value of this register to determine the distance for the $\overline{\text{PAFE}}$ flag. For example, if the $\overline{\text{PAFE}}$ register is set to 16, then the $\overline{\text{PAFE}}$ flag will be active when the FIFO is 16 or fewer words from being either empty or full. If the user does not specify a value for the programmable flag register, the software will initialize it to be equal to half of the FIFO size.

The three flag outputs are encoded from five internal flag registers as shown in ***Figure 2***. This allows the FIFO to use only three routing channels for the flags while, if needed, still allowing any of the five original flags to be decoded by user logic outside the FIFO block. The internal empty and full flags shown in ***Figure 2*** are used by the read and write circuitry to prevent false reads and writes when the FIFO is empty or full.

Four of the five FIFO flags are synchronous, meaning that they change state relative to either readclock or writeclock. The internal flags denoting empty and almost-empty are updated exclusively by readclock. The internal flags denoting full and almost full are updated exclusively by writeclock. The synchronous flag architecture guarantees that the flags maintain their status for a minimum of one read or write cycle, depending on which clock the flag is synchronized to. The internal half full flag, however, is asynchronously updated by either an enabled readclock or writeclock.

Since the FIFO flags are synchronous, they require an edge from their respective clock to update their most current status. The clock to which a flag is not synchronized will be referred to as the opposite clock. If a flag boundary is crossed due to an operation from the opposite clock, a flag update cycle is

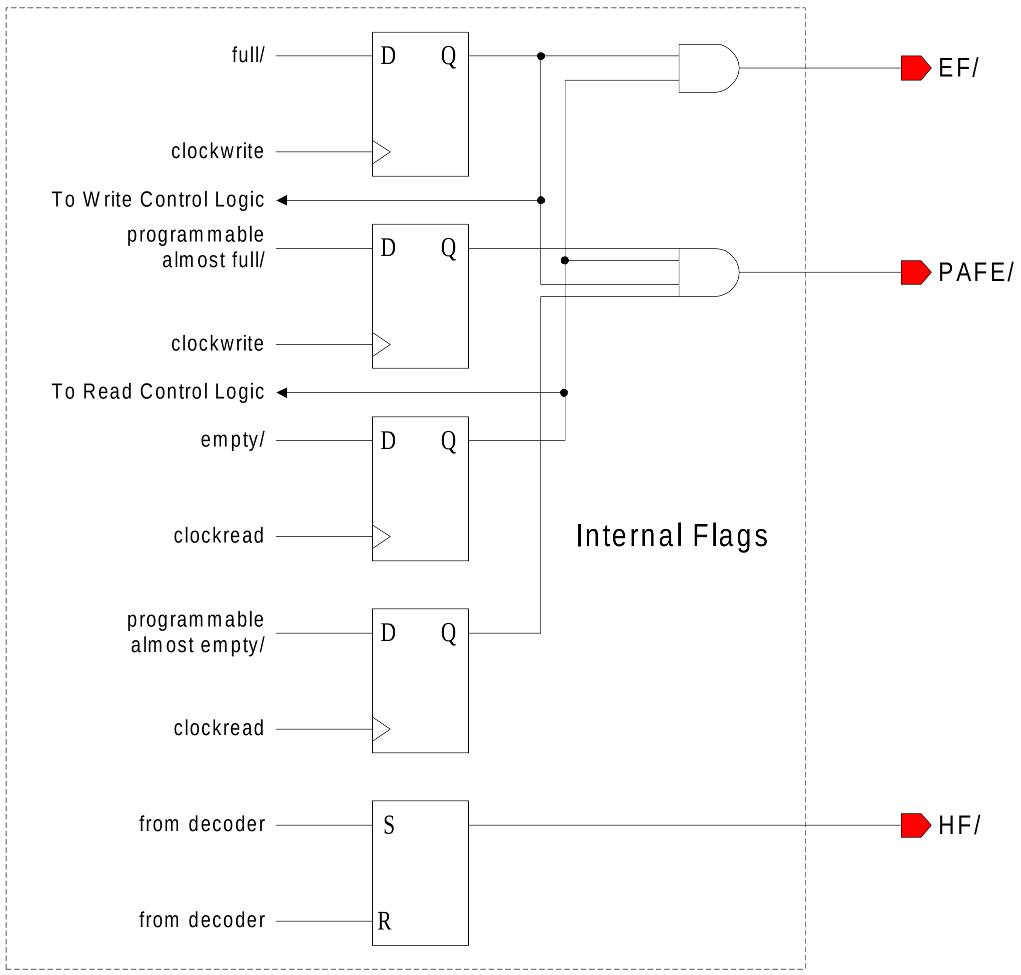


Figure 2. Flag Output Encoding

necessary to represent the new state of the FIFO flags. For instance, if the FIFO is full and a read is performed, the full flag is not updated until the next active edge of writeclock. In this case, even though a read was performed, the FIFO remains in the full state until the active writeclock edge following the read. This period of latency until the flag is updated is called the flag update or latent cycle.

Clock timings can also effect this latency. When an enabled FIFO operation takes place that changes the status of a boundary condition (full or empty), there is a minimum period of time that must lapse before the opposite clock can register the operation and subsequently update the flags. If this skew time parameter, which states a minimum skew between writeclock and readclock, is violated, then the update may happen on the following active updating clock. This adds an additional flag update latency. This extra latency cycle will occur when both writeclock and readclock are tied together for single-clock operation.

The empty and full flags mark the absolute bounds of the FIFO and are therefore referred to as boundary flags. The half full, almost full, and almost empty flags are called non-boundary flags because they are used to indicate the FIFO's status between the bounds of the FIFO. The ramification of the flag update cycle is different for boundary versus non-boundary flags.

The difference occurs because boundary flags are used by the read and write control logic to prevent reading invalid data or overwriting valid data in the FIFO memory. For example, reads are suppressed by the read control logic while the empty flag is active. Though data written into the FIFO is controlled by writeclock, the empty flag is only updated by readclock. When the empty flag is active, two read cycles are required to read data out of the FIFO after new data has been written to the FIFO. The first read serves only to update the empty flag to inactive, while the second read retrieves the data. The flag is updated regardless of the state of ENR so an enabled read is not necessary to update the empty flag. The FIFO operates in a similar manner when the full flag is asserted. After data has been read from a full FIFO, two cycles of writeclock are necessary to write data to the FIFO again.

Non-boundary flags require a similar flag update cycle. For example, if the FIFO just reaches the almost full state and two words are then read from the FIFO, a rising edge on writeclock is required to update the status of the almost full flag. However, since the non-boundary flags do not affect the read and write control logic, reads and writes can proceed normally during the flag update cycle. This means the state of the enable inputs will affect the FIFO operation during the flag update cycle. In the example above, if ENW is active on the rising edge of writeclock, data will be written to the FIFO and the write pointer will be updated at the same time that the status of the almost full flag is refreshed. If ENW is not active, only the almost full flag will be updated.

Resetting the FIFO

When the Master Reset ($\overline{\text{MR}}$) becomes active (LOW), the FIFO enters the empty condition signified by an active $\overline{\text{EF}}$ and $\overline{\text{PAFE}}$ and an inactive $\overline{\text{HF}}$. The read and write pointers are both reset to point to the same location in the FIFO. All data outputs transition LOW after $\overline{\text{MR}}$ is asserted. In order for the FIFO to properly reset to its default state, the user must not read or write to the FIFO while $\overline{\text{MR}}$ is active. A Master Reset is mandatory to guarantee proper FIFO block initialization.

Expandability

The FIFO blocks in the Delta39K are made up of 4096 bits of memory. Multiple blocks of channel memory can be used in conjunction to form deeper and wider FIFO blocks. FIFO memory blocks are usually classified in terms of the width and depth of the block where: FIFO Block Size = Depth x Width.

In Delta39K CPLDs, FIFOs can be configured to one of four depths, 4096, 2048, 1024, and 512, limited by the possible configurations supported by the channel memory. Wider FIFOs are formed by chaining multiple FIFO blocks of the same depth in parallel. FIFOs can be configured in sizes given by 4096xN, 2048x2N, 1024x4N, or 512x8N, where N is the number of individual FIFO blocks used. If the number of output bits required from a certain channel memory block is less than the number available, the extra bits are left unconnected. For instance, an implemented 2048x3 FIFO will span two 2048x2 channel memories with one data output bit left unused. This width expansion is performed automatically.

The total number of FIFO blocks is limited by the total number of channel memory blocks available, varying by device. For example, to implement an expanded FIFO block of size 1024x8, two channel memory blocks, each configured as 1024x4 FIFOs are used. The lower four data bits are supplied by one FIFO block and the upper four data bits from the other. No additional circuitry or multiplexing is needed to perform this expansion.

Built-in logic to support chaining multiple FIFO blocks together to form depths greater than 4096 is not included in the channel memory blocks. Therefore declaring FIFOs with a depth greater than 4096 is not allowed in Cypress's *Warp*™ software. As a result, FIFOs with depths greater than 4096 are not automatically supported by Delta39K CPLDs.

When using multiple channel memory blocks to expand a FIFO in width, the blocks must stay synchronized with each other. Blocks could possibly get out of synchronization when a read operation occurs nearly simultaneous to a write operation near a boundary condition of the FIFO. For instance, consider the case of two FIFO blocks linked together in width expansion, each block containing one word. If a read and write occur at the same time, the first block may perform the read just before the write. In this block, the last word is read from the FIFO causing the block's empty flag to be asserted. When a word is written to the FIFO a moment later, the block is no longer empty, but the empty flag is still asserted awaiting a flag update cycle. Meanwhile, the second block performs the write first, and thus momentarily has two words when the read is processed. After the read is processed, the second block also contains one word, but this block does not require a flag update cycle. Since only one of the blocks is waiting for its flags to update, the blocks are now out of synchronization.

In Delta39K CPLDs, the synchronization problem has been solved by placing FIFO blocks into either a master or slave mode of operation. In master mode, the FIFO read, write and flag operation is normal and performs as previously described. Any stand-alone channel memory block that is configured for FIFO operation will be in master mode.

In slave mode, the FIFO will accept two of the three flags as inputs rather than outputs. In this case, the FIFO accepts the $\overline{\text{EF}}$ and $\overline{\text{HF}}$ from the master FIFO as inputs and will not generate internal flags to determine the boundary conditions. Instead, it decodes the empty and full states from the master's $\overline{\text{EF}}$ and $\overline{\text{HF}}$ flags. The slave FIFO will execute a write and update the write pointer as long as the $\overline{\text{EF}}$ and $\overline{\text{HF}}$ inputs do not indicate the full state. Likewise, the slave FIFO will execute a read and update the read pointer as long as the $\overline{\text{EF}}$ and $\overline{\text{HF}}$ inputs do not indicate the empty state. This allows multiple FIFO blocks to easily be expanded in parallel without the possibility of one FIFO block becoming out of synchronization with the others.

Even though the slave FIFO blocks do not generate flag outputs, all of the FIFO blocks used in the expansion must use the same $\overline{\text{MR}}$ and clock signals. All FIFO blocks must be reset with a Master Reset cycle before operation can begin so that the read and write pointers of all of the FIFO blocks are synchronized.

FIFO Interface to Routing Channels

A FIFO is interfaced to other CPLD resources through the Delta39K's routing channels. Each block of channel memory can interface to one horizontal and one vertical routing channel from which it can route its inputs and outputs. All of the ports for the FIFO may interface to just one of these channels. This is significant because it allows expanded FIFOs to be implemented along a single routing channel without incurring extra routing delay. The delay incurred for routing signals to multiple FIFO blocks from the same routing channel is the same as the delay for routing to a single FIFO block.

On the other hand, if a FIFO is implemented using multiple blocks that are not on the same routing channel, additional timing delays do apply. The extra delay is due to the extra time needed to route common control signals to each FIFO block. When all the FIFO blocks are interfaced to a common channel, then all of the control signals can be routed to the FIFO through that channel. However, when the FIFO blocks are interfaced to different routing channels, then the control signals have to be routed between horizontal and vertical routing channels, each time adding additional delay.

Inputs to a FIFO Block

Inputs to a FIFO block are routed from a routing channel through a Programmable Interconnect Matrix (PIM). This PIM, specially designed to facilitate routing inputs to the channel memory, is called the Channel-to-Memory PIM. The PIM has been designed to be highly routable, so that any of the signals available in the routing channel can be made available to the channel memory. These signals come from I/O block inputs, cluster outputs, data coming from other channel memory blocks, flags coming from other channel memory blocks, and signals coming from either H-to-V or V-to-H PIMs which route signals between routing channels.

The Channel-to-Memory PIMs redirect all the data and control signals needed to implement any of the channel memory's functions. To implement a FIFO block, the channel memory may need a total of 18 signals. These signals include 8 data lines, 4 local clocks, ENR, ENW, $\overline{\text{MR}}$, and two flag inputs for slave mode FIFOs. In addition, the PIM can route 12 address lines that are used to implement dual or single port memory, but that are not needed to implement a FIFO. Since each input to the FIFO has a dedicated purpose and needs to be routed to a specific location, signals routed to the channel memory need to be routed to a particular PIM output. Therefore, the PIM has been designed so that a signal routed through it can be routed to any channel memory input.

Outputs from a FIFO Block

The channel memory block outputs do not require the use of a PIM to connect to the horizontal and vertical channels. Rather, each output of the channel memory block drives a dedicated track in the horizontal and vertical channels. Therefore, eleven lines are added to a routing channel for each channel memory block connected to it. Eight of the eleven tracks are used for the channel memory data output and the remaining three tracks are used for the channel memory flag outputs. The same data and flag outputs are driven to both routing channels when the channel memory is configured as a FIFO.

Static Timing Analysis

Detailing the entire timing model of Delta39K CPLDs is beyond the scope of this application note. Instead, the specific timing of the channel memory block configured as a FIFO will be detailed. This serves as a supplement to the timing diagrams and parameters found in the Delta39K data sheet.

Relevant waveforms showing the timing of all modes of FIFO operation are included within ***Appendix A: Timing Diagrams***. For these waveforms, the specified timing parameters (see Delta39K datasheet) are relative to the pins of the device. Hence the input and output signals shown are representative of signals at the pins, driven to or from the channel memory block through dedicated lines in the Delta39K routing channels.

Figure 6 shows the basic FIFO timings for a read and write operation to and from I/O pins. An enabled read means that the ENR signal is asserted at the rising edge of readclock. Likewise, an enabled write indicates that ENW is asserted at the writeclock active edge, provided that the data and enable signals meet the set-up and hold requirements of the input stage. These timings illustrate that a valid FIFO operation synchronized to the same clock as the applicable FIFO flag is reflected in the status of the FIFO flags after an internal delay.

Figure 7 shows the read and write FIFO timings for the empty and full boundary conditions. The displayed word count represents the number of words stored in the FIFO at any given time. An ignored read (write) will occur when an enabled read (write) takes place while the FIFO is already empty (full). Since the empty flag is synchronized to readclock, a write operation on the opposite clock (writeclock) will induce a latent cycle for flag update since the empty flag status must be updated on the first readclock active edge, followed by the desired data on the next edge. If the skew between the readclock and writeclock is not large enough then the read/write clock skew time for flag parameter will be violated. The result is that the flag update will occur on the following active clock edge. The same can be said for both full and empty border conditions.

Figure 8 displays the almost full and almost empty timings as determined by the setting of the programmable flag register. As previously mentioned, the $\overline{\text{PAFE}}$ flag is asserted whenever the number of words stored in the FIFO range by a set number from the empty or full state. Crossing an almost full or empty boundary has no effect on the operation of the FIFO other than to update the $\overline{\text{PAFE}}$ flag (i.e. further reads or writes to the FIFO are not inhibited).

Figure 9 shows the timings for read and write operations around the half full boundary. The $\overline{\text{HF}}$ is asserted whenever the word count lies above half way. However, this flag is asynchronous since it is updated by either a read or write operation. If the FIFO reaches the half full state, then the next enabled write will assert the $\overline{\text{HF}}$ flag. If the FIFO is half full plus one, then the next enabled read will deactivate the $\overline{\text{HF}}$ flag.

Figure 10 shows the FIFO timing for a Master Reset operation. The $\overline{\text{MR}}$ (or mrb) signal must be applied for a minimum pulse width to guarantee a proper reset. During this time, no enabled reads or writes to the FIFO should be made by the user. The first valid read or write operation can only take place after the master reset recovery time has been satisfied.

For specific FIFO implementations, delays associated with the other parameters in the Delta39K timing model should be included with the information shown in the waveforms from *Appendix A*. For example, if the output data or FIFO flag outputs are routed to macrocells, there will be an associated read clock to macrocell clock timing parameter to satisfy.

More information on Delta39K timing is available in other Cypress application notes. Detailed information on the channel memory blocks configured as dual-ports is available in "Delta39K and Quantum38K™ Dual-Port RAM."

Software Support

The Delta39K devices are supported by Cypress's *Warp* software Release 6.0 and above. The *Warp* software package is a complete CPLD development tool that allows users to enter designs in either VHDL or Verilog. Once the design description is written, the HDL is synthesized and fit into a target device. The fitter algorithms allow designs to be efficiently fitted to a programmable logic device. One of the key components of this fitting for the Delta39K family is to make use of the channel memory resources as dual-port and FIFO memories.

Warp users specify FIFOs through instantiations of the CY_FIFO component in their VHDL or Verilog code. The CY_FIFO component is parameterized to support all possible size configurations and FIFO features. ***Figure 3*** illustrates the port signals of the CY_FIFO component, whereas *Table 2* lists the names, types, and characteristics for all these ports. *Table 3* lists the properties used to configure the size and programmable flag length of the CY_FIFO component.

To use the FIFO in VHDL designs for the Delta39K, an instance of the CY_FIFO component (located in the Templates menu of *Warp*) must be declared and the required ports connected. Additionally, required generic parameters must be given values. Optional ports or generics of the CY_FIFO component may also be mapped to take advantage of other FIFO logic functions. A VHDL example design using the CY_FIFO component is shown in ***Figure 4***. Here, the FIFO is used to interface two processors operating with different bus widths and clock domains.

To enable the use the CY_FIFO component in VHDL code, the "LPMPKG" package must be linked in the design file.

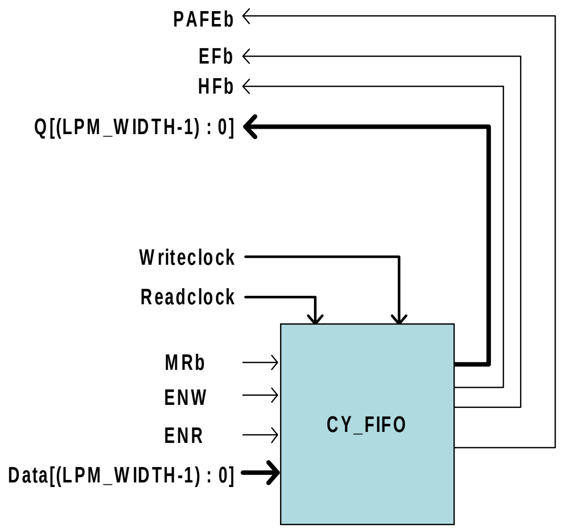


Figure 3. Ports of the CY_FIFO Component

Table 2. CY_FIFO Ports Function Description

Port Name	Type	Usage	Description	Comments
Data	In	Required	Data input to FIFO	Vector, LPM_WIDTH wide
Q	Out	Required	Data output from FIFO	Vector, LPM_WIDTH wide
ENR	In	Required	Read Enable	
ENW	In	Required	Write Enable	
Readclock	In	Required	Read Clock (CKR)	
Writeclock	In	Required	Write Clock (CKW)	
MRb	In	Required	Master reset	Active Low
EFb	Out	Optional	Empty/full flag	Active Low
HFb	Out	Optional	Half full flag	Active Low
PAFEb	Out	Optional	Programmable almost-empty/full flag	Active Low

Table 3. CY_FIFO Properties

LPM Property	Usage	Value	Comments
LPM_WIDTH	Required	LPM value > 0	Width of input and output data vectors
LPM_NUMWORDS	Required	LPM value > 0. Must be a valid configuration depth: 4096, 2048, 1024, or 512	Depth of the the FIFO.
LPM_PAFE_LENGTH	Optional	0 < LPM value < LPMNUMWORDS / 2	Distance in words from full or empty which results in the PAFE flag being set
LPM_HINT	Optional	Not used	Not used

The use of FIFOs in Verilog designs is similar. The CY_FIFO module must be instantiated with the required ports connected. In Verilog, the properties of the FIFO are specified by default parameters.

These default parameters can be changed by using the defparam override technique supported by *Warp*. Unspecified parameters are assigned their default values. Additional features of the CY_FIFO module can be used by connecting the appropriate ports and specifying the necessary parameters. An example of using the CY_FIFO component in Verilog is shown in ***Figure 5***. This is the same example application as previously described.

To enable the use the CY_FIFO module in Verilog code, the "LPM.V" source file must be linked to the design file.

With Delta39K devices, the channel memory can be configured to operate as a FIFO. However, the dedicated logic that is built-in to support all of the FIFO functions is not available in the Quantum38K family. For these latter devices, while one may still implement FIFOs using dual-port memory instantiations, all of the surrounding flag and address pointer logic must be specified by the user. This logic would be implemented within the macrocells contained in the device logic blocks. Hence, the CY_FIFO component is not available for Quantum38K CPLDs.

Conclusions

The Delta39K family combines a high density CPLD architecture with powerful embedded features like single and dual-port RAM, built-in FIFO support, Phase-Locked Loop (PLL), and flexible I/O standards. The Delta39K helps integrate common system components to thereby decrease design turnaround, improve system speeds, and reduce board area.

To that aim, the Delta39K contains built-in logic that allows the embedded dual-port memory blocks to have true FIFO functionality. This functionality comes without overhead due to no additional demands for logic resources. All of the necessary FIFO read/write address pointers and flag logic are contained within every channel memory block.

Several FIFO configuration options are offered, including four depth selections as well as flexible FIFO data widths by expansion. The width expansion capability is provided automatically without risk of synchronization problems. As well, there is no additional timing penalty for expanding across a horizontal or vertical routing channel.

The Delta39K CPLD family offers a single-chip solution for low-density synchronous FIFO applications.

```
-- Delta39K sample code showing use of internal FIFO
-- Unidirectional data transfer between two processors using 512x32 FIFO.  Processor1 has a 32-bit bus independently  writing
-- to FIFO.  Transmission stops when FIFO has only 4 words free space.  Processor2 has an 8-bit data bus and outputs a
-- x4 clock to allow multiplexing of the 32-bit FIFO data words.

library ieee;
use ieee.std_logic_1164.all;
library cypress;
use cypress.lpmpkg.all;
use cypress.std_arith.all;

ENTITY bus_matcher IS PORT (
        proc1_data: in std_logic_vector (31 downto 0);      -- input data from processor1
        proc2_data: out std_logic_vector (7 downto 0);      -- output data to processor2
        proc1_clk,                                          -- processor1 clock
        proc2_clk_x4: in std_logic;                         -- processor2 clock x 4
        proc1_we,                                           -- write enable from processor1
        proc2_re: in std_logic;                             -- read enable from processor2
        proc1_tx_wait,                                      -- halt tx from proc1 if '1'
        no_data: out std_logic;                             -- halt read to proc2 if '1'
        FIFO_resetb: in std_logic                           -- master reset for FIFO
 );
END bus_matcher;

ARCHITECTURE arch_bus_matcher OF bus_matcher IS

signal internal: std_logic_vector (31 downto 0);
signal clk_divide, temp_clk_divide : std_logic_vector (1 downto 0);
signal temp_efb, temp_hfb, temp_pafeb : std_logic;                          -- FIFO flags

BEGIN

proc1_tx_wait <= temp_efb and not(temp_pafeb) and not(temp_hfb);    -- shows if FIFO is almost full
no_data        <= not(temp_efb) and temp_hfb;                       -- shows if FIFO is empty

mux:process (proc2_clk_x4)                                          -- multiplex FIFO data onto smaller data bus
begin
        if (proc2_clk_x4'event and proc2_clk_x4 = '1') then
                case clk_divide is
                        when "10" => proc2_data <= internal(31 downto 24);
                        when "11" => proc2_data <= internal(23 downto 16);
                        when "00" => proc2_data <= internal(15 downto 8);
                        when others => proc2_data <= internal(7 downto 0);
                end case;
                temp_clk_divide <= clk_divide + 1;
        end if;
end process;

divide:process (FIFO_resetb, temp_clk_divide)
begin
        if (FIFO_resetb = '0') then             -- reset count if FIFO_resetb = ‘0’
                clk_divide <= "00";
        else
                clk_divide <= temp_clk_divide;
        end if;
end process;

U1: cy_fifo-- Clocked FIFO
        generic map(
                lpm_width          => 32,
                lpm_numwords       => 512,
                lpm_pafe_length    => 4,        -- optional
                lpm_hint           => speed     -- optional
        )
        port map(
                data        => proc1_data,
                q           => internal,
                enr         => proc2_re,
                enw         => proc1_we,
                readclock  => clk_divide(1),
                writeclock => proc1_clk,
                mrb         => FIFO_resetb,  -- optional
                efb         => temp_efb,     -- optional
                hfb         => temp_hfb,     -- optional
                pafeb       => temp_pafeb    -- optional
        );
END arch_bus_matcher;
```

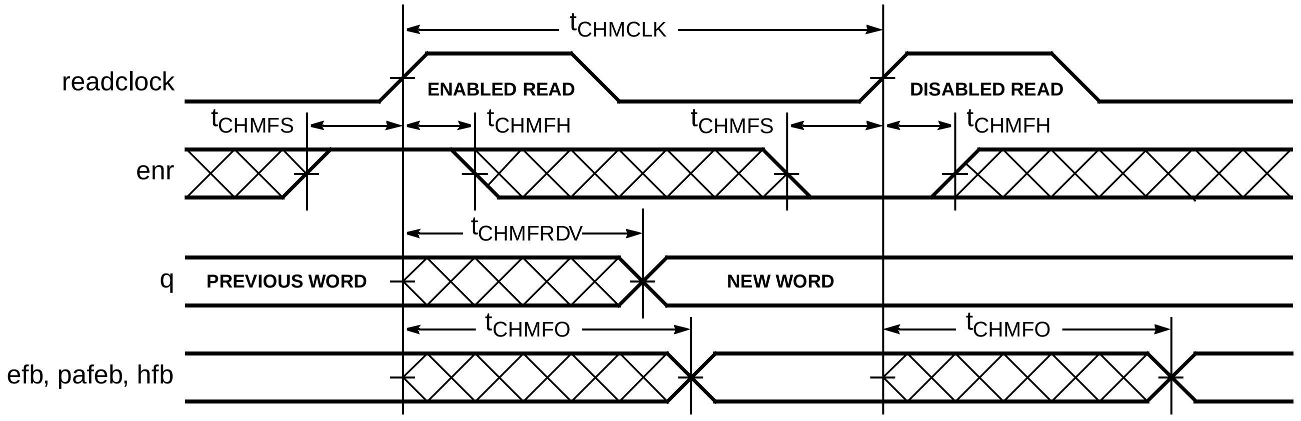
Figure 4. VHDL Sample Code With CY_FIFO Instantiation

```
// Delta39K sample code showing use of internal FIFO
// Unidirectional data transfer between two processors using 512x32 FIFO.  Processor1 has a 32-bit bus independently  writing
// to FIFO.  Transmission stops when FIFO has only 4 words free space.  Processor2 has an 8-bit data bus and outputs a
// x4 clock to allow multiplexing of the 32-bit FIFO data words.

`include "lpm.v"
`include "rtl.v"

module bus_matcher (proc1_data, proc2_data, proc1_clk, proc2_clk_x4, proc1_we,
                    proc2_re, proc1_tx_wait, no_data, FIFO_resetb);

input [31:0] proc1_data;        // input data from processor1
output [7:0] proc2_data;        // output data to processor2
reg   [7:0] proc2_data;
input proc1_clk;                // processor1 clock
input proc2_clk_x4;             // processor2 clock x 4
input proc1_we;                 // write enable from processor1
input proc2_re;                 // read enable from processor2
output proc1_tx_wait;           // halt tx from proc1 if '1'
output no_data;                 // halt read to proc2 if '1'
input FIFO_resetb;              // master reset for FIFO

wire  [31:0] internal;
wire  temp_efb, temp_hfb, temp_pafeb;     // FIFO flags
reg   [1:0] clk_divide, temp_clk_divide;  // counter for multiplexor

defparam U0.lpm_width= 32;
defparam U0.lpm_numwords= 512;
defparam U0.lpm_pafe_length= 4;       // optional
defparam U0.lpm_hint= `SPEED;         // optional
cy_fifo  U0(
    .data( proc1_data ),
    .q        ( internal ),
    .enr  ( proc2_re ),
    .enw ( proc1_we ),
    .readclock( clk_divide[1] ),
    .writeclock( proc1_clk ),
    .mrb ( FIFO_resetb ),          // optional
    .efb  ( temp_efb ),            // optional
    .hfb  ( temp_hfb ),            // optional
    .pafeb( temp_pafeb ));         // optional

assign proc1_tx_wait = temp_efb & !temp_pafeb & !temp_hfb;
assign no_data = !temp_pafeb & temp_hfb;

always @(posedge proc2_clk_x4)    //multiplex FIFO data onto smaller data bus
    begin
        case (clk_divide)
            2'b10:      proc2_data <= internal[31:24];
            2'b11:      proc2_data <= internal[23:16];
            2'b00:      proc2_data <= internal[15:8];
            default :   proc2_data <= internal[7:0];
        endcase

        temp_clk_divide <= clk_divide + 1;
    end

always @(FIFO_resetb or temp_clk_divide)
        if (FIFO_resetb == 1'b0)      // asynchronous FIFO reset if 0
            clk_divide <= 2'b00;
        else
            clk_divide <= temp_clk_divide;

endmodule
```

Figure 5. Verilog Sample Code With CY_FIFO Instantiation

Appendix A: Timing Diagrams

a) FIFO Read Timing



b) FIFO Write Timing

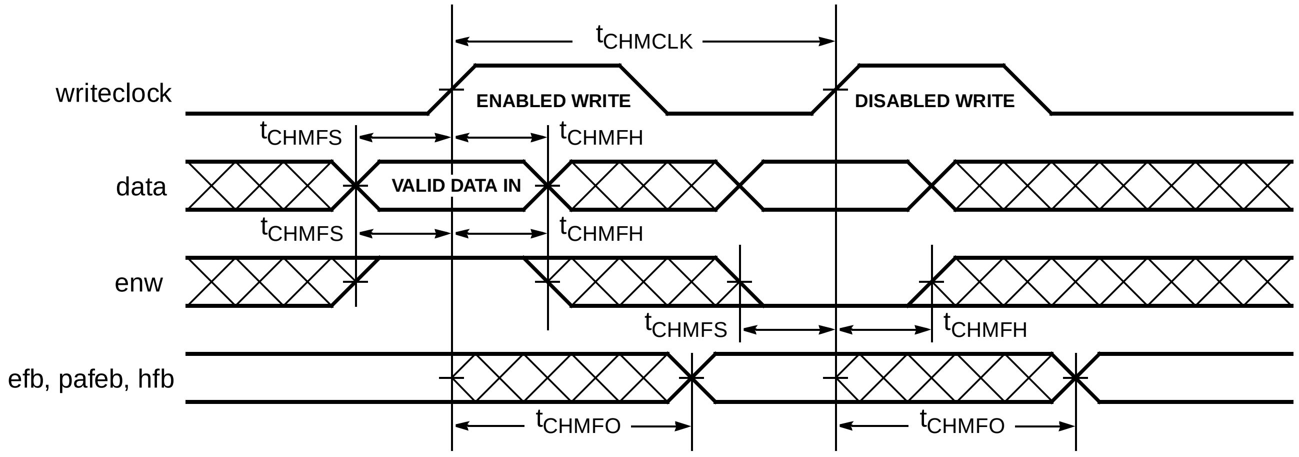
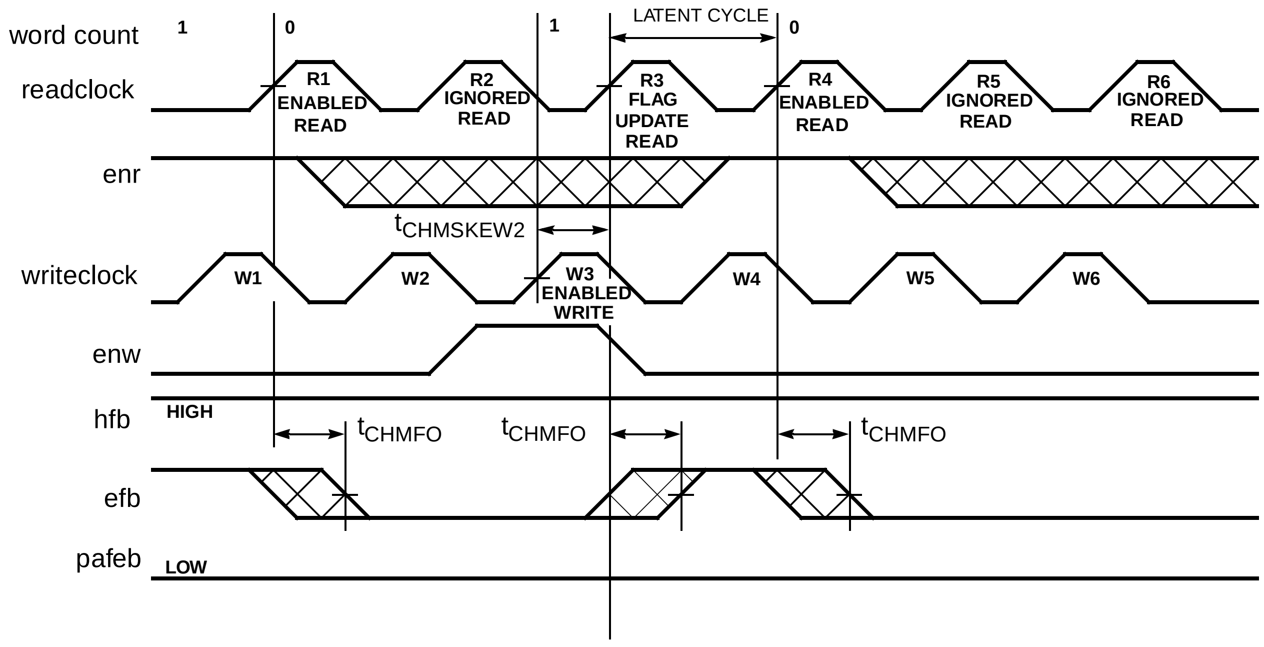


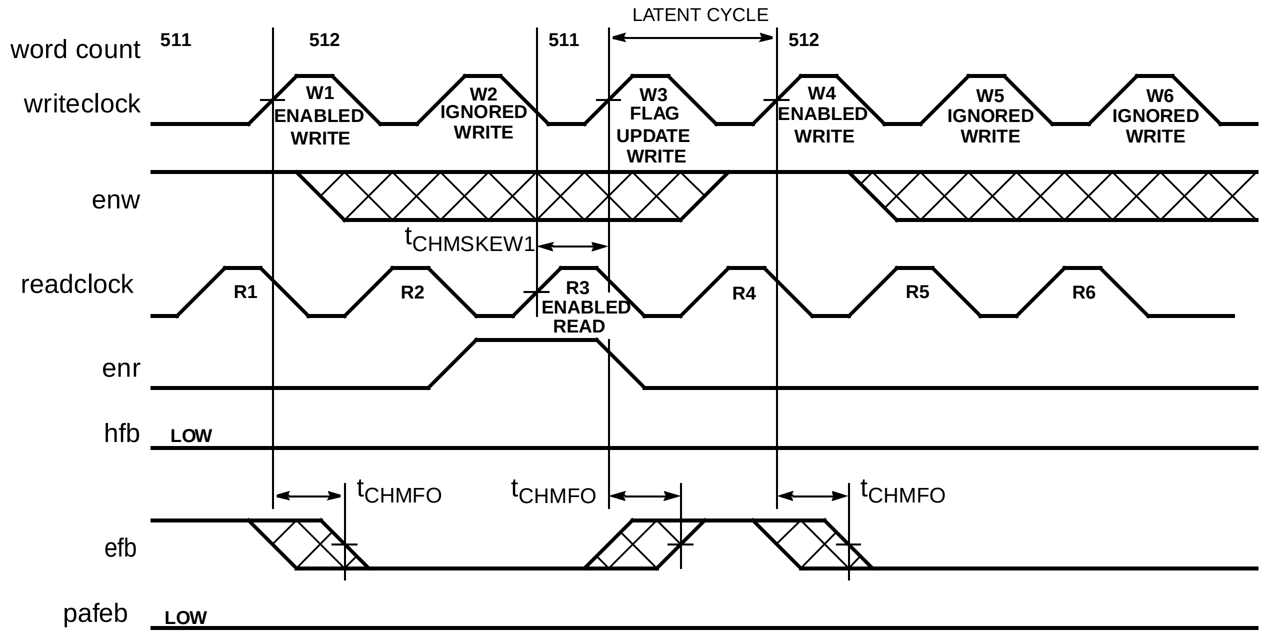
Figure 6. FIFO Timing for Basic Read and Basic Write Operation

Figure 7. FIFO Timing for Read to Empty and Write to Full Operations

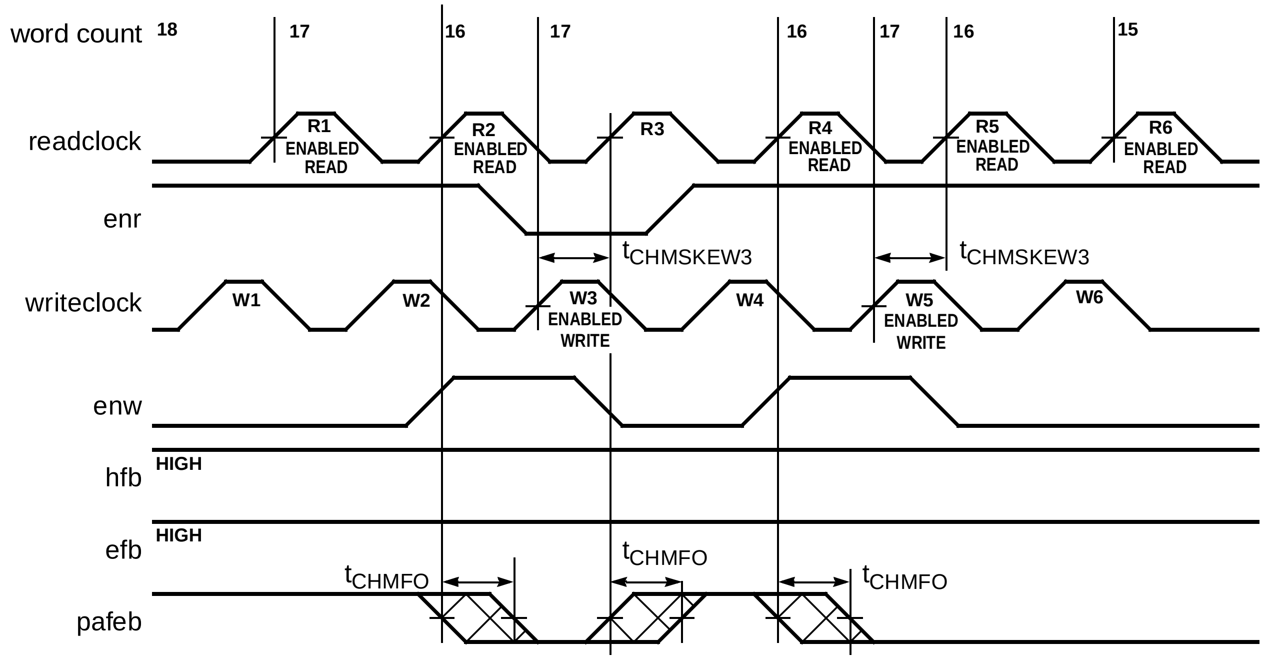
a) Read to Empty Timing Diagram [1,2,3,4]



b) Write to Full Timing Diagram [1,2,3,4]



a) Read to Almost Empty Timing Diagram [1,3,5]



b) Write to Almost Full Timing Diagram [1,3,5]

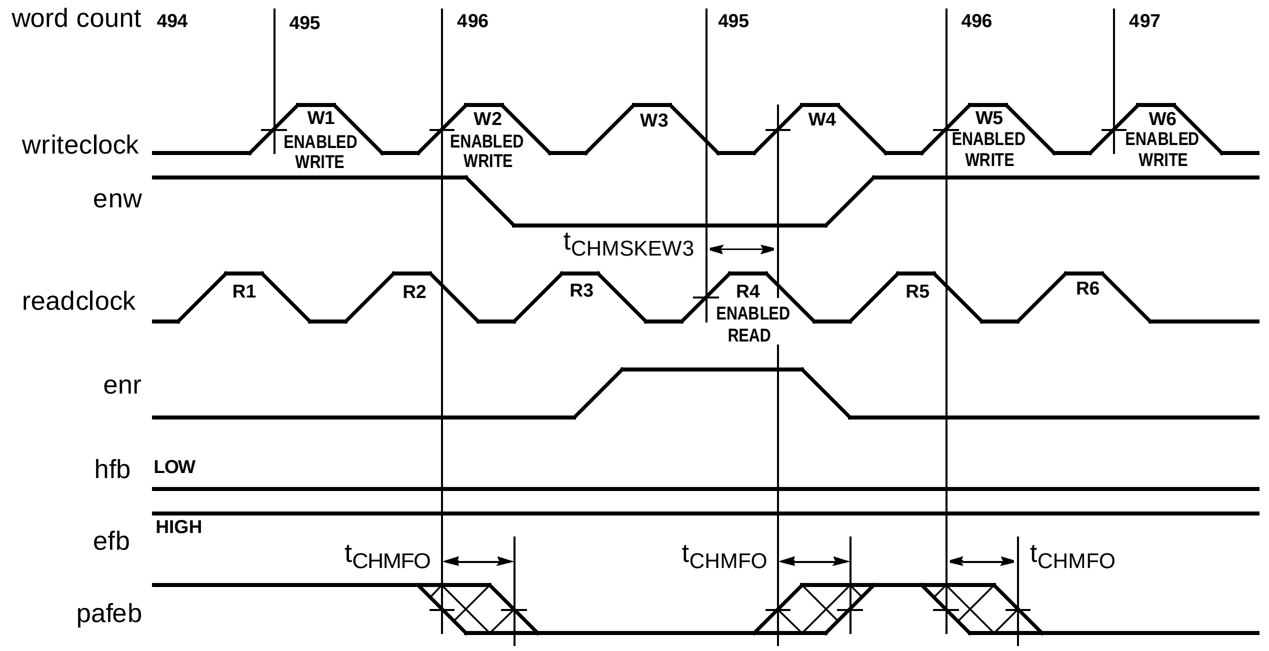


Figure 8. FIFO Timings for Read to Almost Empty and Write to Almost Full Operations

Write/Read to Half Full Timing Diagram [1,6]

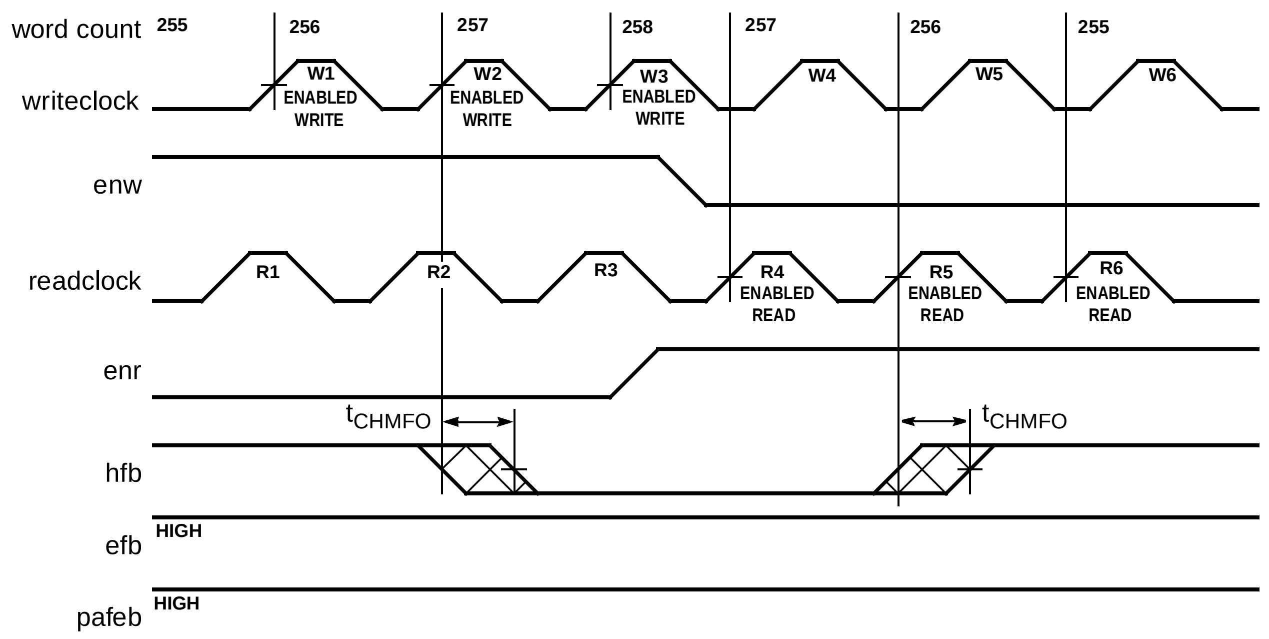
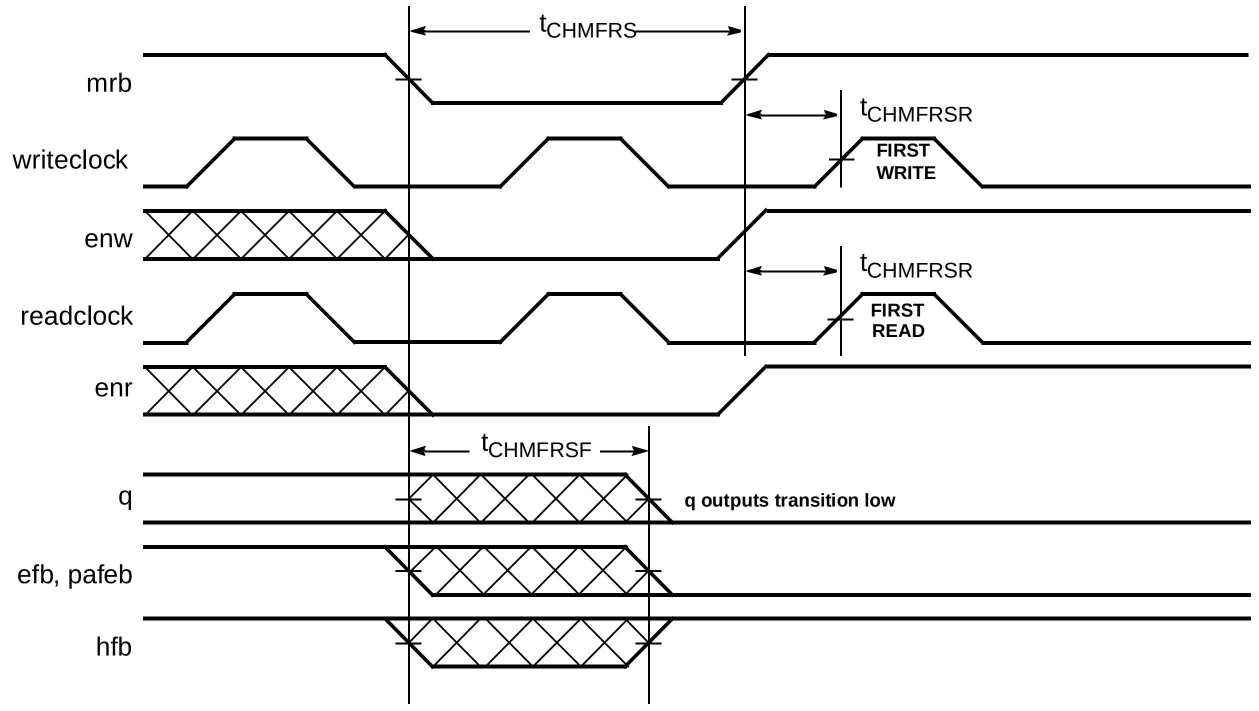


Figure 9. FIFO Timing for Read and Write to Half Full Operations

Figure 10. FIFO Timing for Master Reset Operation

Master Reset Timing [see Note 7, 8]



Notes:
1. FIFO block is assumed to be configured to depth of 512 words.
2. During the Latent Cycle, only the flags are updated. While the flags indicate the FIFO is full, no data can be written to the FIFO. While the flags indicate the FIFO is empty, no data can be read from the FIFO.
3. If the CKR to CKW skew parameter is violated, the associated flag update will occur at the next active clock edge.
4. LPM_PAFE_LENGTH is assumed to have the default value of half the FIFO depth.
5. The FIFO in this example is assumed to have LPM_PAFE_LENGTH set to 16.
6. The $\overline{\text{HF}}$ is asynchronous and can be updated by either a read or write operation.
7. To ensure a proper reset, the user must not read or write to the FIFO while $\overline{\text{MR}}$ is asserted.
8. The first valid read/write operation must follow the deassertion of $\overline{\text{MR}}$ after satisfying the reset recovery time parameter.