IOP 480/SFIFO AN

July 5, 2000

Version 2.0

IOP 480 to Synchronous FIFO Application Note

Features

- IOP 480 running at 66MHz Local bus.
- Synchronous FIFO, 16K x 36 x 2
- Logic code and testbench

General Description

This application note describes how to interface the PLX Technology IOP 480 to a Synchronous FIFO.

The IOP 480 has Direct Master, DMA, and Direct Slave data transfer capabilities. The Direct Master mode allows the IOP 480 on the Local bus to perform memory, I/O, and configuration cycles to the PCI bus. The Direct Slave mode allows a master device on the PCI bus to access memory on the Local bus. The IOP 480 allows the Local bus to run asynchronously to the PCI bus through the use of internal bi-directional FIFOs. The IOP 480 can operate up to 33MHz on the PCI bus and up to 66MHz on the Local bus.

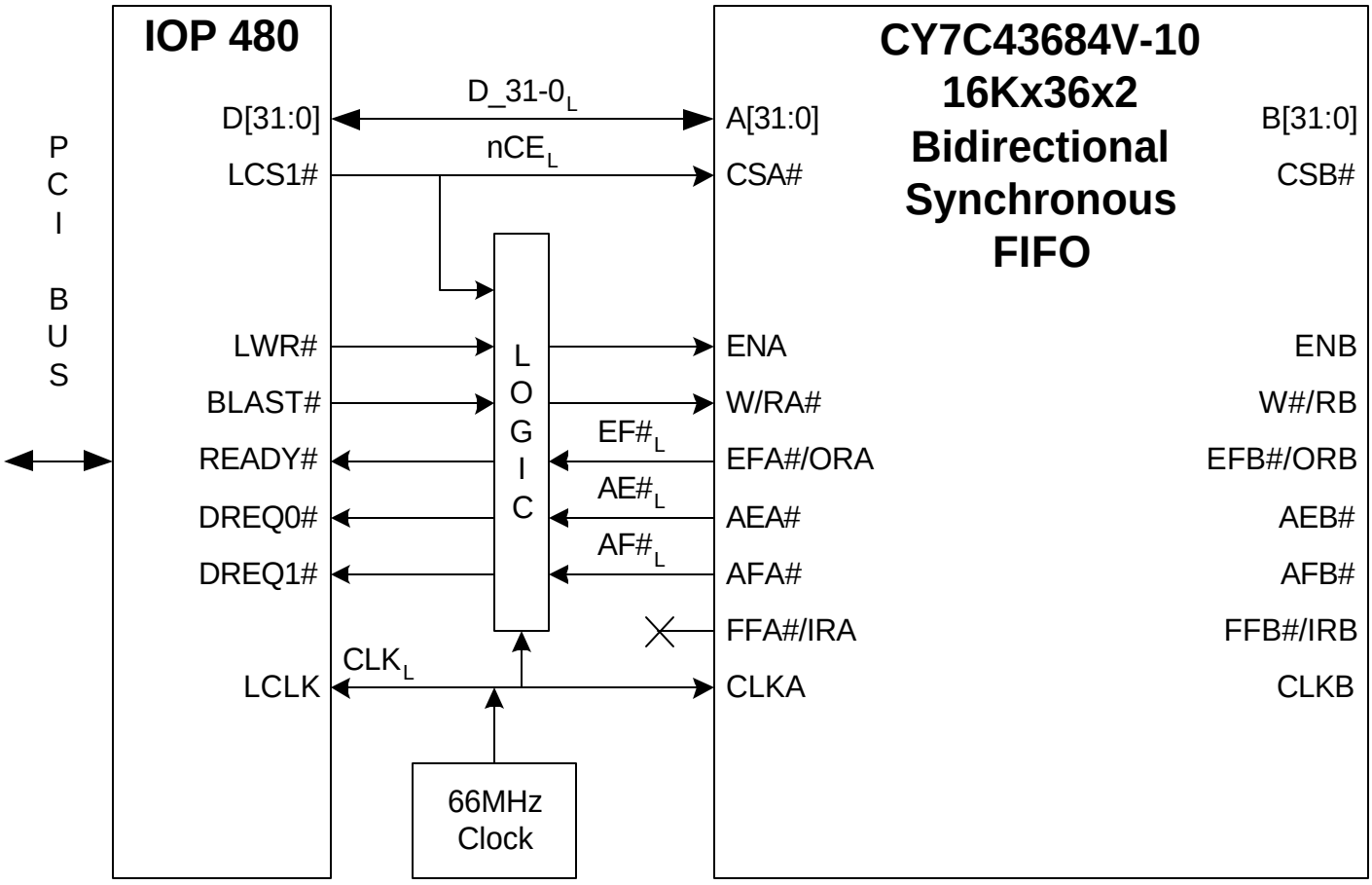

Figure 1. Block Diagram

PLX Technology, Inc, 390 Potrero Avenue, Sunnyvale, CA 94085, Phone 408-774-9060, Fax 408-774-2169

TABLE OF CONTENTS

1. BLOCK DIAGRAM COMPONENTS 2

1.1. IOP 480 2

1.2. FIFO 2

1.3. LOGIC 2

2. FUNCTIONAL DESCRIPTION 2

2.1. IOP 480 FUNCTIONS 2

2.1.1. IOP 480 Register Settings 2

2.2. FIFO FUNCTIONS 2

2.3. LOGIC FUNCTIONS 3

2.3.1. IOP 480 DMA Writes 3

2.3.1.1. Why are there Two Clocks to Every DMA Write Data? 3

2.3.2. IOP 480 DMA Reads 3

2.3.2.1. Why are there Three Clocks to Every DMA Read Data? 3

2.3.2.2. What Does Logic Do for the Three Clocks of a DMA Read? 3

3. SUMMARY 4

4. ASSUMPTIONS 4

5. REFERENCES 4

6. APPENDIX 5

6.1. LOGIC CODE 5

6.1.1. Test Bench. 9

6.2. LOGIC WAVEFORMS 13

LIST OF FIGURES

Figure 1. Block Diagram 1

1. Block Diagram Components

Figure 1 shows the application note components. The following subsections describe these components.

1.1. IOP 480

The IOP 480 is a high performance 32bit, 33MHz PCI interface chip. The IOP 480 has a 32bit, 66MHz SRAM Local bus interface that is used to access the FIFO. The IOP 480 has two flexible DMA channels for data transfers between the PCI and Local bus.

1.2. FIFO

The FIFO is the Cypress CY7C43684-10 a bi-directional 16K x 36 bit Synchronous FIFO. The FIFO has two internal independent unidirectional FIFOs. One buffer is used for IOP 480 writes and the other is used for IOP 480 reads. The IOP 480 is connected to the FIFOs port 'A' with a small amount of logic.

1.3. Logic

The logic inputs the FIFO flags and IOP 480 control signals and outputs FIFOs port 'A' control signals and IOP 480 DMA request signals.

2. Functional Description

The functions of each block are described in the following subsections.

2.1. IOP 480 Functions

The IOP 480 uses Demand Mode DMA to transfer data with the FIFO. Demand Mode DMA is used so that the DMA transfer can be 'paused' and restarted without the need to rewrite the descriptor blocks. This improves DMA performance.

The IOP 480 uses DMA channel #0 for reads. Channel #0 is configured for fast terminate mode. Fast terminate mode is required for a single DMA read. A single DMA read is needed to read a nearly empty FIFO without underflow.

The IOP 480 uses DMA channel #1 for writes. Channel #1 is configured for slow terminate mode. This allows the logic to detect the end of the write bus cycle by the IOP 480 asserting BLAST#.

2.1.1. IOP 480 Register Settings

The table below shows the IOP 480 SRAM Access Programmable Timing Parameter Register settings for this application.

IOP 480 Register Settings				
RAD=1	RDD=1	RDLYA=0	RDLYD=0	RRCV=1
WAD=1	WDD=0	WHLD=1	WDLY=0	WRCV=1

The IOP 480 requires that WHLD is set greater than or equal to one.

2.2. FIFO Functions

The Cypress CY7C43684-10 is a 32K x 36-bit Synchronous FIFO. This is a 3.3V device. The memory of the FIFO is 16K deep by 36 bits wide. The FIFO is operating in standard mode and all data written into the FIFO must be read out on the opposite side before it is driven on to the data bus.

This bi-directional FIFO has two ports. Each port provides four FIFO status flags. For port 'A' the flags are: Almost Empty 'A' (AEA#), Empty Flag 'A' (EFA#), Almost Full 'A' (AFA#), and Full Flag 'A' (FFA#). Port 'B' has matching flags. The threshold levels of the AEA# and AFA# flags are programmable by the user. The AEA# and AFA# flags

should be programmed to one. The FIFO serial programming and reset interfaces are not included in this applications note.

2.3. Logic Functions

As shown in the block diagram, a small amount of logic is required to control the FIFO control signals. The following subsections describe the DMA read and DMA write operations. Logic code (Verilog), testbench, and waveforms are in the appendix.

2.3.1. IOP 480 DMA Writes

The FIFO tells the logic that there is space for writing in the write FIFO buffer by negating AFA#. The logic then enables the IOP 480 to do a DMA write to the FIFO by asserting DREQ1#. The AFA# flag should be programmed to assert one data before the write FIFO buffer is full. This lets the logic negate DREQ1# in time to prevent the write FIFO buffer from overflowing. The FFA# flag is not used. The IOP 480 DMA channel #1 must be configured for slow termination mode. This allows the IOP 480 to indicate the end of the bus cycle to the logic by asserting BLAST#.

2.3.1.1. Why are there Two Clocks to Every DMA Write Data?

The IOP 480 requires that register bit WHLD is set greater than or equal to one. This results in two clocks for every DMA write data.

2.3.2. IOP 480 DMA Reads

The FIFO tells the logic that it has data in the read FIFO buffer by negating EFA#. The logic then enables the IOP 480 to do a DMA read from the FIFO by asserting DREQ0#. The AEA# flag should be programmed to assert one data before the read FIFO buffer is empty. This lets the logic negate DREQ0# and so pause any on going IOP 480 DMA read to prevent an invalid read from an empty read FIFO buffer. The IOP 480 DMA channel #0 must be configured for fast termination mode. This allows for DMA reads of the FIFO when the FIFO is almost empty. Once the IOP 480 begins a DMA read cycle from the FIFO, there are two possible operations.

The first possible operation is if EFA# is negated and AEA# is asserted. This means that there is only one data in the read FIFO buffer. The logic goes into single DMA read transfer mode. Once the IOP 480 asserts ADS# for a FIFO DMA read, the logic negates the IOP 480 DREQ0# input on the rising edge of the clock. This pauses the DMA read after just one data read.

The second possible operation is if both EFA# and AEA# are negated. This means that there is more than one data in the read FIFO buffer. The logic goes into DMA burst read mode. The logic continues to assert the IOP 480 DREQ0# input until the FIFO AEA# is asserted. The AEA# flag should be programmed to assert one data from empty. The IOP 480 does one more assured good read before the DMA is paused.

2.3.2.1. Why are there Three Clocks to Every DMA Read Data?

In DMA burst read mode, three clocks are used for each data. This is because the IOP 480 could terminate a DMA read bus cycle immediately after READY# is asserted without any prior warning. The BLAST# signal cannot be used for this since it will not assert in Demand Mode DMA with fast termination. There are many possible causes for such a bus cycle termination: terminal count is reached, Host asserting STOP#, BOFF# asserted, etc. For more information, see the IOP 480 Data Book, page 7-2.

2.3.2.2. What Does Logic Do for the Three Clocks of a DMA Read?

On the first clock after READY# negates, the logic checks LCS#. On the second clock, if LCS# has negated, the logic does not command the FIFO for a read and the logic goes into an idle state. If LCS has not negated, the logic commands the FIFO for a read. On the third clock, if the FIFO was commanded to read, the logic asserts READY# to strobe the FIFO data into the IOP 480. The logic repeats this three-clock cycle until the IOP 480 negates LCS#.

3. Summary

The PLX IOP 480 interfaces to external FIFOs. The IOP 480 block transfer DMA features and Scatter/Gather can be used. The FIFO transfers are 'paused' and restarted without rewriting the descriptor blocks. The write FIFO buffer will never overflow. The read FIFO buffer will always be emptied, even if only one data is in it. A DMA burst write takes two clocks per data. A DMA burst read takes three clocks per burst data. A small amount of logic is needed.

4. Assumptions

This application note is based on the following assumption:

- Some typically necessary design components and functions (i.e. boot and code memory, EEPROM, and System Reset) are not included in this application note.

The designers are expected to add such components and functions, as their design requires.

5. References

- IOP 480 Data Book v1.0
  PLX Technology, Inc.
  390 Potrero Avenue
  Sunnyvale, CA 94085 USA
  Tel: (408) 774-9060, 800 759-3735,
  Fax: (408) 774-2169,
  http://www.plxtech.com/

- Synchronous FIFO
  CY7C43684-10 Data Sheet
  Cypress
  3901 North First Street
  San Jose, CA 95134
  U.S.A.
  Tel: (408) 943-2600
  Fax: (408) 943-2741
  http://www.cypress.com/

6. Appendix

The logic code, test bench, and stimulation waveforms are in the following subsections.

6.1. LOGIC CODE

```
/*******************************************************************************/
/* MODULE:  fifo.v                                                             */
/* COMPANY: PLX TECHNOLOGY                                                     */
/* DATE:    03-25-2000                                                         */
/*                                                                             */
/* REVISION HISTORY:                                                           */
/* Rev 1.0:  Initial Development                                               */
/*                                                                             */
/* DESCRIPTION:                                                                */
/* Logic code for a PLX IOP 480 / Synchronous FIFO Interface.                  */
/*******************************************************************************/

`timescale 1ns/1ns

module  fifo(
            clk_i,
            ads_L_i,
            lcs1_L_i,
            lwr_L_i,
            blast_L_i,

            almost_e_L_i,
            empty_L_i,
            almost_f_L_i,

            dreq0_L_o,
            dreq1_L_o,
            ready_L_o,
            ena_o,
            w_ra_L_o
           );

//
// inputs and outputs
//

// Signals from IOP 480
input  clk_i;
input  ads_L_i;
input  lcs1_L_i;
input  lwr_L_i;
input  blast_L_i;

// Signals from FIFO
input almost_e_L_i;
input empty_L_i;
input almost_f_L_i;
```

```
// Signals to IOP 480
output dreq0_L_o;
output dreq1_L_o;
output ready_L_o;

// Signals to FIFO
output ena_o;
output w_ra_L_o;

wire  clk_i;
wire  ads_L_i;
wire  lcs1_L_i;
wire  lwr_L_i;

wire  almost_e_L_i;
wire  empty_L_i;
wire  almost_f_L_i;

parameter H = 1'b1;
parameter L = 1'b0;

// All state bits are outputs.
// The s4 bit has no output pin.

//  dreq0_L_o  ----------+
//  ready_L_o  ----------+|
//  ena_o      ----------+||
//  w_ra_L_o   ----------+|||
//  s4         --------+ ||||
//                     v_vvv
parameter state_a = 5'b0_1011; // Idle state DREQ0# and DREQ1# Negated.
parameter state_b = 5'b0_1010; // Assert DREQ0#.
parameter state_c = 5'b0_0000; // Read READY# Asserted, DREQ0# Asserted.
parameter state_d = 5'b0_0010; // Check if LCS# is still Asserted.
parameter state_e = 5'b0_0110; // CMD FIFO READ, DREQ0# Asserted.
parameter state_j = 5'b0_0111; // CMD FIFO READ, DREQ0# Negated.
parameter state_k = 5'b0_0001; // Read READY# Assserted, DREQ0# Negated.
parameter state_l = 5'b0_0011; // Check if LCS# is still Asserted.
parameter state_p = 5'b1_0111; // CMD FIFO READ (single), DREQ0# negated.
parameter state_q = 5'b0_1001; // Read READY# Asserted, DREQ0# Negated.
parameter state_w = 5'b0_1101; // Write READY# Asserted, DREQ0# Negated.
parameter state_x = 5'b0_1100; // Write READY# Asserted, DREQ0# Asserted.
parameter state_v = 5'b1_1011; // Write Hold, DREQ0# Negated.
parameter state_z = 5'b1_1010; // Write Hold, DREQ0# Asserted.

reg dreq1_L_o;
reg[4:0] state;

assign dreq0_L_o = state[0];
assign ready_L_o = state[1];
assign ena_o     = state[2];
assign w_ra_L_o  = state[3];
```

```
always @ (posedge clk_i)
begin
 case (state)

  // State_a : No FIFO flags are negated.
  state_a :
   if      ((ads_L_i == L) && (lcs1_L_i == L) && (lwr_L_i == H) && (empty_L_i == L))
                          #2 state = state_w;
   else if ((ads_L_i == L) && (lcs1_L_i == L) && (lwr_L_i == H) && (empty_L_i == H))
                          #2 state = state_x;
   else if (empty_L_i == H)    #2 state = state_b;
   else                        #2 state = state_a;

  // State_b : The EF# is negated.
  state_b :
   if      ((ads_L_i == L) && (lcs1_L_i == L) && (lwr_L_i == H) && (empty_L_i == L))
                                          #2 state = state_w;
   else if ((ads_L_i == L) && (lcs1_L_i == L) && (lwr_L_i == H) && (empty_L_i == H))
                                          #2 state = state_x;
   else if ((ads_L_i == H) || (lcs1_L_i == H))             #2 state = state_b;
   else if ((lwr_L_i == L) && (almost_e_L_i == H))          #2 state = state_e;
   else if ((lwr_L_i == L) && (almost_e_L_i == L))          #2 state = state_p;

  state_c : if (almost_e_L_i == H) #2 state = state_d; // Check if lcs1_L_1 is High.
            else                   #2 state = state_l; // Last read

  state_d : if      ((lcs1_L_i == H) && (empty_L_i == L))
                  #2 state = state_a; // IOP Ends Bus Cycle DREQ0# = H
            else if ((lcs1_L_i == H) && (empty_L_i == H))
                  #2 state = state_b; // IOP Ends Bus Cycle DREQ0# = L
            else  #2 state = state_e; // Do another read.

  state_e : #2 state = state_c; // Always Assert READY# after READ CMD to FIFO.

  state_l : if      ((lcs1_L_i == H) && (empty_L_i == L))
                  #2 state = state_a; // IOP Ends Bus Cycle DREQ0# = H
            else if ((lcs1_L_i == H) && (empty_L_i == H))
                  #2 state = state_b; // IOP Ends Bus Cycle DREQ0# = L
            else  #2 state = state_j; // Last FIFO read CMD.

  state_j : #2 state = state_k; // Always Assert READY# after READ CMD to FIFO.

  state_k : #2 state = state_a;

  state_p : #2 state = state_q; // Single FIFO CMD read.

  state_q : #2 state = state_a; // Single  READY# Assertion.

  state_w : #2 state = state_v; // Always go to hold state.

  state_v : if (blast_L_i == L) #2 state = state_a; // IOP ends cycle DREQ0=H
            else                #2 state = state_w;

  state_x : #2 state = state_z; // Always go to hold state.
```

```
  state_z : if (blast_L_i == L) #2 state = state_b; // IOP ends cycle DREQ0#=L
            else                #2 state = state_x;

  default : #2 state = state_a;

 endcase
end

always @ (posedge clk_i)
begin
  dreq1_L_o = almost_f_L_i;
end

endmodule
```

6.1.1. Test Bench

```
/*********************************************       *****/
/* MODULE:  fifo_tb.v                                  */
/* COMPANY: PLX TECHNOLOGY                             */
/* DATE:    03-07-2000                                 */
/*                                                     */
/* REVISION HISTORY:                                   */
/* Rev 1.0: Initial Development                        */
/*                                                     */
/* DESCRIPTION:                                        */
/* Provides Test Bench for IOP480                      */
/* Synchronous FIFO Interface.                         */
/************************************************       /

`timescale 1ns/1ns

module fifo_tb;

parameter H = 1'b1;
parameter L = 1'b0;
parameter CP = 14;

// register control
reg    clk_i;
reg    ads_L_i;
reg    lcs1_L_i;
reg    lwr_L_i;
reg    blast_L_i;

reg    almost_e_L_i;
reg    empty_L_i;
reg    almost_f_L_i;

wire   dreq0_L_o;
wire   dreq1_L_o;
wire   ready_L_o;
wire   ena_o;
wire   w_ra_L_o;

fifo fifo(
          .clk_i(clk_i),
          .ads_L_i(ads_L_i),
          .lcs1_L_i(lcs1_L_i),
          .lwr_L_i(lwr_L_i),
          .blast_L_i(blast_L_i),

          .almost_e_L_i(almost_e_L_i),
          .empty_L_i(empty_L_i),
          .almost_f_L_i(almost_f_L_i),

          .dreq0_L_o(dreq0_L_o),
          .dreq1_L_o(dreq1_L_o),
```

```
        .ready_L_o(ready_L_o),
        .ena_o(ena_o),
        .w_ra_L_o(w_ra_L_o)
       );

initial
  begin
        clk_i         = H;
    #2;  // Assures change is after clock edge.
           lcs1_L_i          = H;
        ads_L_i       = H;
        lwr_L_i    = H;
        blast_L_i     = H;

        almost_e_L_i  = L;
        empty_L_i     = L;
        almost_f_L_i  = H;
    #CP;

    // READ DMA Burst

    #CP   empty_L_i     = H;
    #CP;
    #CP;  almost_e_L_i  = H;
    #CP;
    #CP;
    #CP;  ads_L_i           = L;
          lcs1_L_i  = L;
             lwr_L_i           = L;
    #CP    ads_L_i          = H;
    #CP;
    #CP;
    #CP;
    #CP;
    #CP   almost_e_L_i  = L;
    #CP;
    #CP;
    #CP   lcs1_L_i          = H;
          empty_L_i     = L;

    // READ DMA burst ends wit no warning.

    #CP empty_L_i         = H;
    #CP almost_e_L_i      = H;
    #CP;
    #CP lcs1_L_i = L;
        ads_L_i               = L;
        lwr_L_i               = L;
           blast_L_i          = H;
    #CP ads_L_i               = H;
    #CP;
    #CP;
    #CP;
    #CP;
```

```
#CP lcs1_L_i        = H; // IOP 480 ends READ
#CP;

// READ DMA Only Two Data

#CP;
#CP   empty_L_i      = H;
#CP   almost_e_L_i   = L;
#CP;
#CP;  ads_L_i        = L;
      lcs1_L_i       = L;
      lwr_L_i        = L;
#CP   ads_L_i        = H;
#CP;
#CP   lcs1_L_i       = H;
#CP;  ads_L_i        = L;
      lcs1_L_i       = L;
      lwr_L_i        = L;
#CP   ads_L_i        = H;
#CP;
#CP   lcs1_L_i       = H;
      empty_L_i      = L;
#CP;

// WRITE DMA BURST with empty_L_i asserted.

#CP   almost_f_L_i   = H;
      almost_e_L_i   = L;
      empty_L_i      = L;
#CP;
#CP;
#CP;  ads_L_i        = L;
      lcs1_L_i       = L;
      lwr_L_i        = H;
#CP   ads_L_i        = H;
#CP;
#CP;
#CP   almost_f_L_i   = L;
#CP   blast_L_i      = L;
#CP;
#CP   lcs1_L_i       = H;
      blast_L_i      = H;
#CP;
#CP;

// WRITE DMA BURST with empty_L_i negated.

empty_L_i           = H;
almost_f_L_i        = H;
#CP;
#CP   almost_e_L_i   = H;
#CP;
#CP;  ads_L_i        = L;
      lcs1_L_i       = L;
```

```
        lwr_L_i       = H;
    #CP   ads_L_i       = H; // W Data
    #CP;                     // Hold
    #CP;                     // W Data
    #CP   almost_f_L_i  = L; // Hold
    #CP   blast_L_i     = L; // W Data
    #CP;                     // Hold
    #CP   lcs1_L_i      = H; // Recovery
        blast_L_i     = H;
    #CP;

    #CP  $finish;
  end

initial
  begin
   #10000 $finish;
  end

always #(CP/2)
  clk_i = !clk_i;

endmodule
```

6.2. LOGIC Waveforms