Depth Expansion of Synchronous FIFOs

Introduction

Applications often require FIFO buffers deeper than those offered by discrete devices. By depth expanding multiple devices, a logically deeper FIFO can be constructed. The synchronous FIFO family offers two approaches to this common application. The CY7C42x5 family of x18 devices contain an on-chip expansion circuit, allowing very simple cascading. The CY7C42x1 family of x9 FIFOs take a very different approach since the on-chip expansion logic is not available. Both approaches will be discussed in detail, examining the advantages and disadvantages of each.

CY7C42x5 Approach

The CY7C42x5 family of Synchronous and Deep Sync FIFOs (CY7C4425, 4205, 4215, 4225, 4235, 4245, 4255, 4265, 4275, 4285) have an on-chip expansion circuit. Tokens are passed between devices to control which device is operational at any given time. There is only one write token and one read token, and only one device can be the owner of each token at any given time. This ensures that only one device accept write operations and one device accept read operations. See *Figure 1* for a block diagram showing how multiple devices are configured for depth expansion.

The first device in the serial chain must be designated by grounding the First Load ($\overline{FL}$) pin. This designates the device to be the owner of both the read and write tokens after Reset ($\overline{RS}$). All other devices must tie the ($\overline{FL}$) pin to VCC. Expansion pins are tied in a serial fashion. Write Expansion Output (WXO)of each device is tied to Write Expansion Input (WXI)of the next device. Likewise Read Expansion Out ($\overline{RXO}$)of each device is tied to Read Expansion Input ($\overline{RXI}$) of the next device. The Load pins ($\overline{LD}$) of all devices are tied together to allow programming of the flag offset registers in parallel.

The Empty and Full flags ($\overline{EF}$, $\overline{FF}$) are ORed together to generate composite flags. This ensures that all devices are empty before the composite $\overline{EF}$ is asserted, and that all devices are full before the composite $\overline{FF}$ is asserted. The Programmable Almost Empty and Almost Full ($\overline{PAE}$, $\overline{PAF}$) flags can also be ORed together to generate composite flags, however they are not as precise as in the Stand-alone mode. Each flag ($\overline{PAE}$, $\overline{PAF}$) will be an accurate representation of the number of words in that particular device. To understand why the composite ($\overline{PAE}$, $\overline{PAF}$) flags are not precise we must examine more closely how the read and write pointers are passed between devices.

The Cypress Sync FIFO family, as well as most other FIFOs, use internal address pointers to direct write/read operations to/from the device. After reset both address pointers are set to zero. The first write operation places data into location zero and the write address pointer is incremented. Each write operation thereafter also increments the write address pointer. Likewise the first read pulls data from address zero and increments the read address pointer. The read pointer essentially "chases" the write pointer. Flags are generated by subtracting the values inside the read and write address pointers. If the difference is zero, the read pointer has "caught up" the write pointer and the $\overline{EF}$ is asserted. If the difference equals the depth of the device, the device is full and the $\overline{FF}$ is asserted. $\overline{PAE}$ and $\overline{PAF}$ flags are generated by calculating the difference between the two address pointers.

In stand-alone operation, once the write address pointer reaches the highest possible location, it rolls over back to location zero. In depth cascade operation, once the write address pointer reaches its maximum value, the WXO pin is given a pulse passing the write token to the next device. Note, the first FIFO does not have to be full to pass the write token. The only requirement for passing the token is that the write address pointer reaches its maximum value. For example, a 1k FIFO in depth expansion will pass the write token after 1k writes, regardless of the number of reads that are performed. Once the write token is passed the write address pointer rolls over to zero in anticipation of the write token's return. Likewise the read token is passed in the RXO pin once the read address pointer reaches its maximum value.

Let us examine the case where two 4k FIFOs are expanded in depth to make an 8k buffer. Lets assume that the $\overline{PAE}$ and $\overline{PAF}$ flags are programmed to assert at 1k words and 3k words respectively. If the user were to perform 5k writes and 1k reads, the total array contains 4k words or half full. However, the first device contains 3k of those words and the second device contains the remaining 1k. Looking at the flags, the first device shows almost full ($\overline{PAF}$) while the second device shows almost empty ($\overline{PAE}$). By ORing the programmable flags together as shown in *Figure 1* it is difficult to determine the overall state of the 8k buffer.

By understanding how the FIFOs pass tokens and how the $\overline{PAE}$ and $\overline{PAF}$ flags are generated it is possible in many cases to understand the overall state of your buffer. In the above example, if you wanted a $\overline{PAF}$ flag to assert when the buffer received 6k writes after a reset, then the $\overline{PAF}$ flag of the upper device can be programmed to 2k from full. The desired result can be obtained by examining the $\overline{PAF}$ flag of the upper device.

The following is a list of advantages to using the token passing approach to depth expansion:

1. No external logic is required to perform the expansion.
2. There is no speed penalty associated with this approach. A depth cascaded FIFO buffer will operate as fast as a stand-alone device.
3. Overall power consumption is lower than other approaches since only one device is operational at a time for a given interface (read/ write).
4. The number of devices you can cascade is limited only by loading of the data buses. Buffers can be used on the data buses to expand even further.
5. There is no latency added to the overall data path as with some expansion methodologies.

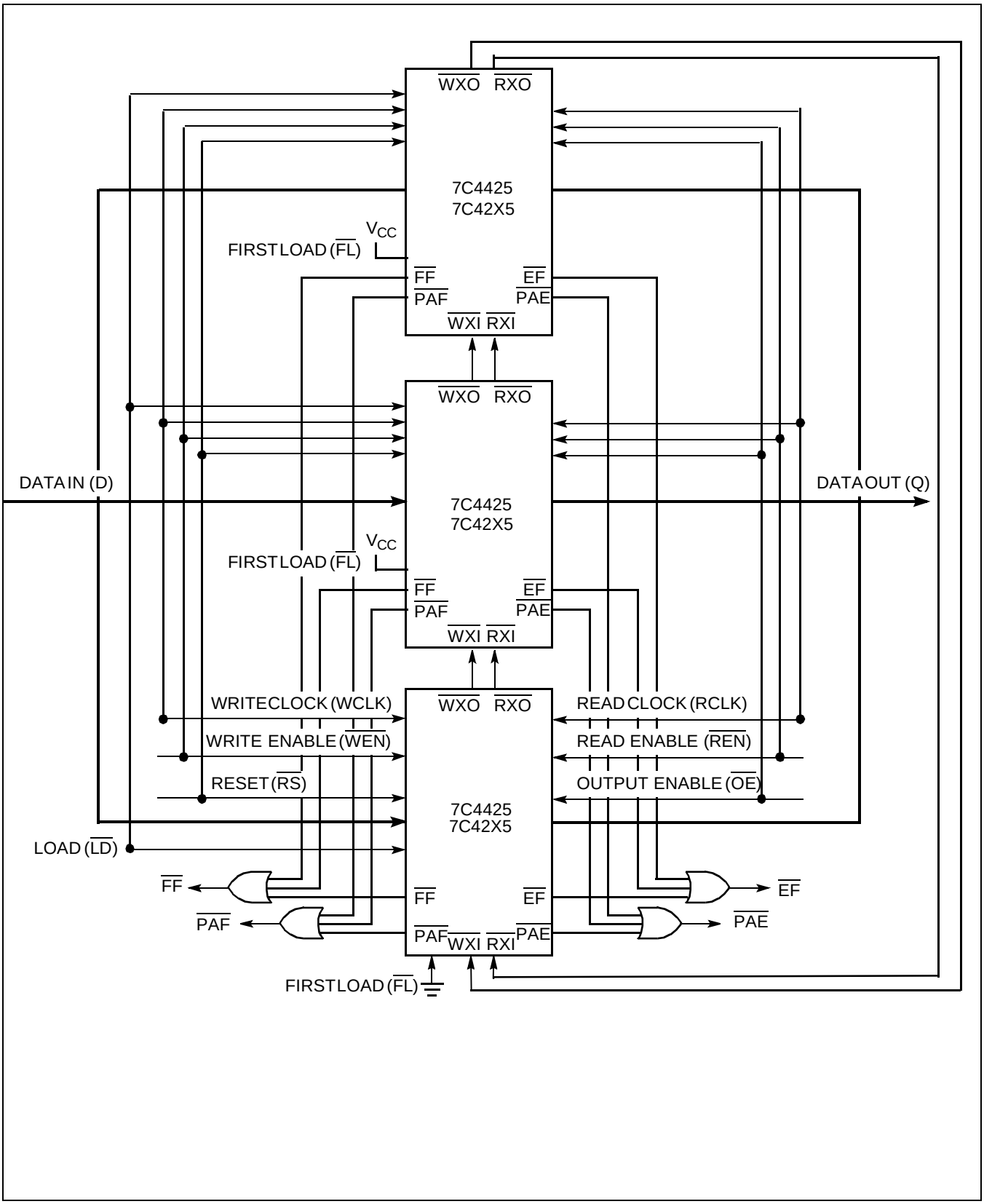


Figure 1. Depth Expansion of CY7C42x5 Synchronous FIFOs Using Token Passing

CY7C42x1 Approach

As mentioned above, the CY7C42x1 devices (CY7C4421, 4201, 4211, 4221, 4231, 4241, 4251, 4261, 4271, 4281, 4291) take a far different approach to depth expansion. Due to the small packages in which these devices are offered, there are not enough pins to support the token passing approach. Instead the devices have two read enable REN and two write enable WEN pins to allow external logic to accomplish the expansion. The following is a description of depth expansion using a "ping-pong" approach.

Writes are performed by rotating each transaction between devices. In other words, the first write is written to FIFO #1, the second write is written to FIFO #2, etc. If we assume that four FIFOs are being depth expanded, then the first four write operations will write one word in each device, in order. The fifth write "ping-pongs" back to FIFO #1 and so on. Read operations are handled in exactly the same manner.

You may already notice some of the advantages and disadvantages to this approach. One advantage is that you have much tighter control of the number of words in each device. All devices basically track each other, meaning if one device is almost full, all other devices will also be almost full. This allows the user to generate much more accurate PAE and PAF flags. A disadvantage to this approach is speed. I will go through a timing analysis later to calculate the maximum speed which this system will run.

Hardware

Figure 2 shows a diagram of the hardware used to support the "ping-pong" approach to depth expansion. Note that, while this diagram shows two devices being cascaded, many more devices could be cascaded using this approach. The limit to how many devices can be cascaded is determined only by loading of the data bus to/from the FIFOs and logic available to implement the controls

The external expansion logic in this example is implemented with a Cypress CPLD, the CY7C371. The CY7C371 has two clock inputs which will allow both the read and write control logic to fit into one device.

WEN2 and REN2 are tied both to the FIFOs and to the CY7C371. WEN1 and REN1 are generated by the CY7C371 and drive the FIFOs directly. All of the flags including FFA, FFB, EFA, EFB, PAFA, PAFB, PAEA, and PAEB, from the FIFOs, are fed back to the CY7C371 for generation of composite flags: EF, FF, PAE, and PAF. These composite flags are generated by simply ORing the flages from both FIFOs. Reset (RS) is driven to both the FIFOs and the CY7C371 for initialization. Output enables OE_A and OE_B are generated by the CY7C371 to ensure that only one FIFO is driving the output data bus (Q0–9) at a time.

Write Enable Controller

Write operations start when the global WEN signal is asserted and the WCLK transitions from low to high. WEN_A and WEN_B are asserted alternately which writes a word into FIFO A, then FIFO B, then FIFO A, then FIFO B, etc. After reset the first write enable asserted is WEN_A. Write operations can be stopped by deasserting the global WEN signal. If the FIFOs become full the write controller will deassert WEN_A and/or WEN_B until the appropriate FF is deasserted.

State "w_reset" is the initialization state which is entered anytime RS is asserted. After RS is deasserted and WEN is asserted the state machine transitions to "write_A." As long as WEN is asserted the write enable controller will "ping-pong" from "write_A" to "write_B" to "write_A" etc. on each rising edge of WCLK until either the WEN is deasserted or one of the devices becomes full. In either case the controller enters one of the "idle" states until both WEN is reasserted and the appropriate FF is deasserted.

Read Enable Controller

The Read operations work the same as the write. When the global read enable signal REN is asserted and the FIFOs are not empty, the REN_A and REN_B signals will alternately assert. This reads data from one device and then the other just as the writes are performed. Read operations can be stopped by deasserting the REN signal or by the devices becoming empty. The read enable controller knows which devices are empty and will not attempt to perform reads to those devices.

State "r_reset" is the initial state of the state machine after reset (RS). Once REN is asserted and the devices are not empty, REN_A is asserted to perform a read operation to FIFO A. Assuming the REN stays asserted and the EFs are not asserted, the state machine will continue to "ping-pong" between "read_A" and "read_B." If an empty flag is asserted the controller will enter into one of the "idle" states until the empty flag EF deasserts. Deasertion of the global REN will also place the controller into the appropriate "idle" state until REN is reasserted.

The output enables (OE_A and OE_B) follow the read enables (REN_A and REN_B) for each transaction to ensure that only one FIFO drives the data bus at a time. When REN_A is asserted, OE_A will assert as well if the global OE signal is active. Likewise when REN_B is asserted, OE_B will assert if the global OE is asserted. Using the global output enable OE, the user can read out words from the device without driving the data bus. In other words, the read control logic of the fifo is not dependent on the state of the OE signal. The OE simply controls the output drivers of the device irrespective of the read operations. This feature allows users to "discard" words from the fifos if necessary.

Timing Analysis

Table 1 lists the critical timing information regarding the CY7C371 CPLD and the CY7C42x1 Synchronous FIFO. To calculate the maximum rate at which the "ping-pong" approach to depth expansion will operate, we must examine t_{CO} and t_S of the CPLD and t_{DS}, t_{ENS}, t_A, t_{WFF}, and t_{RFF} of the FIFO. The maximum write clock frequency is calculated by adding t_{CO} of the CPLD with t_{DS} of the FIFO

Table 1. Critical Timing Parameters

Parameter	Description	Min	Max
t_{CO}	CPLD - Clock to Output		6 ns
t_S	CPLD - Setup time	5 ns	
t_{DS}	FIFO - Data set up time	3 ns	
t_{DH}	FIFO - Data hold time	.5 ns	
t_{ENS}	FIFO - Enable setup time	3 ns	
t_{ENH}	FIFO - Enable hold time	.5 ns	
t_{OE}	FIFO - OE LOW to output valid	3 ns	7 ns
t_{CLK}	FIFO - min. cycle time	10 ns	

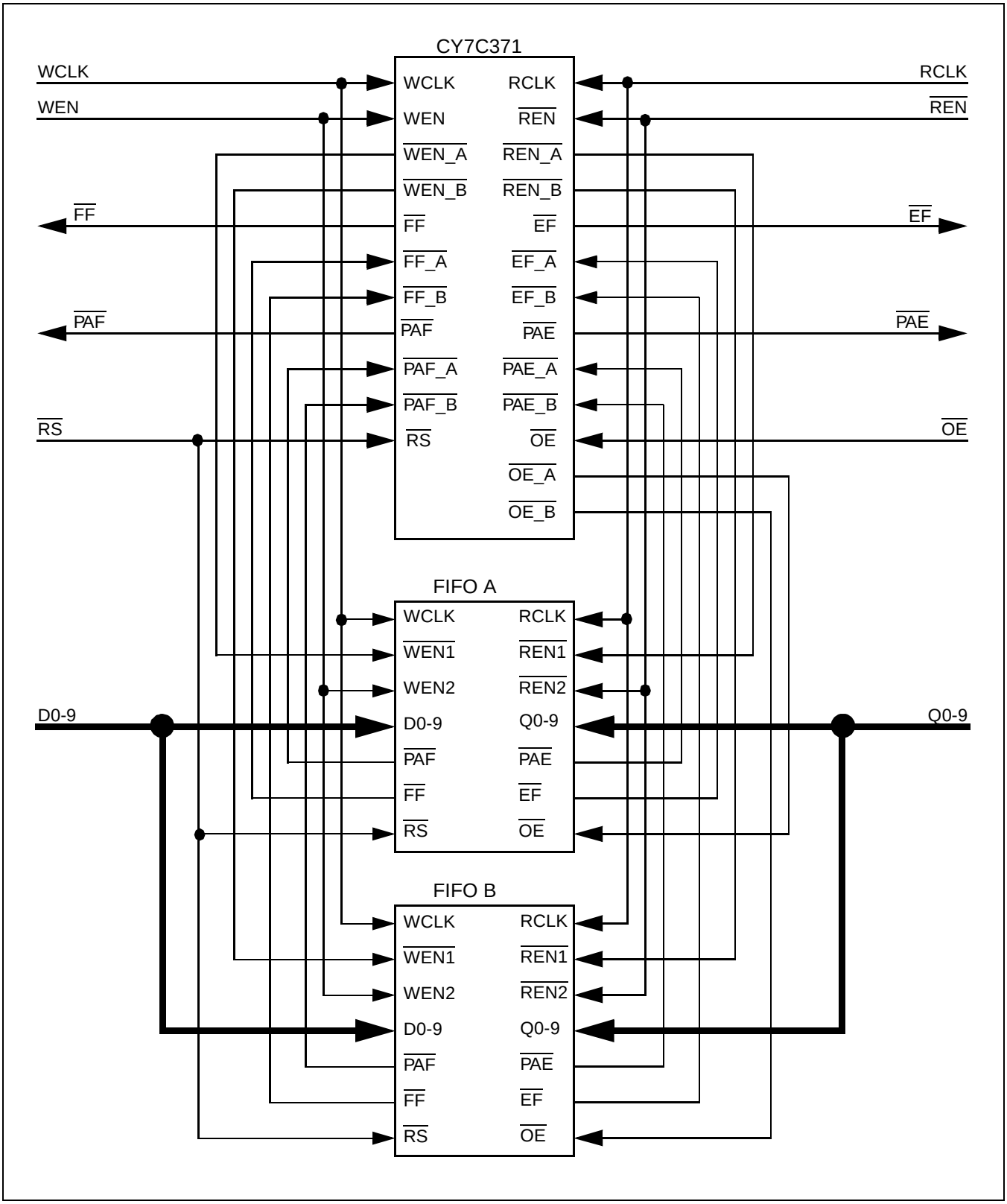


Figure 2. Depth Expansion of Two CY7C42x1 FIFOs Using a CY7C371

$$f_{max} = \frac{1}{t_{DS}(FIFO) + t_{CO}(CPLD)} = \frac{1}{9.5ns} = 110MHz$$

If we assume t_{CO} = 6.5 ns and t_{DS} = 3 ns, the max frequency equals 110 MHz. Since the FIFO is rated with a 10-ns cycle time (100 MHz), this spec. will actually limit writing to 100 MHz. In short, addition of the depth expansion logic does not affect the maximum frequency with which the depth expanded FIFO array can be written.

There is one concern with writing to the depth-expanded FIFO array at 100 MHz—overrun. Overrun is a condition where write operations are attempted to a full FIFO. The CY7C42x1 and CY7C42x5 fifos will discard any write attempts to a full device. If we cannot respond to a full flag assertion within one write clock cycle, there is a possibility that write operations may be performed to a full device. To calculate the maximum frequency which we can write to a depth expanded FIFO array without overrun we must add t_{WFF} of the FIFO and t_S of the CPLD.

$$f_{max} = \frac{1}{t_{WFF}(FIFO) + t_S(CPLD)} = \frac{1}{14ns} = 70MHz$$

We can therefore write to the depth expanded FIFO array with a maximum burst rate of 100 MHz. If we intend to use the full flag to restrict/qualify write operations the write operations are restricted to 70 MHz.

To calculate the maximum read frequency a similar approach is used. Since the output enables $\overline{OE}$_A and $\overline{OE}$_B are toggled for each transaction, read operations, are slightly slower. The maximum read clock frequency is calculated by:

$$f_{max} = \frac{1}{t_A(FIFO) + t_S(DESTINATION)} = \frac{1}{11ns} = 90MHz$$

or

$$f_{max} = \frac{1}{t_{CO}(CPLD) + t_{OE}(FIFO) + t_S(DEST)} = 60MHz$$

whichever is slower. If we assume t_S (destination) = 3 ns, the maximum read clock frequency is 60 MHz. Since this is slower than 70 MHz, underrun will not be an issue with using the empty flag ($\overline{EF}$).

$$f_{max} = \frac{1}{t_{REF}(FIFO) + t_S(CPLD)} = \frac{1}{14ns} = 70MHz$$

f_{max}: underrun

Other Design Concerns

1. The enable controller used in this application note is based on a synchronous state machine design. For proper operation the Read and Write clocks must be free running. This is not the case for normal operation of a stand-alone fifo which allows “gating” of the clocks to prevent read or write operations.
2. The above approach can be used to cascade devices of differing depths. To cascade an 8K device with a 4K device simply modify the VHDL to write 2 words to the 8K device for every one word to the 4K device. This will ensure that the devices fill at the same rate and that the status flags will track.
3. By using a CPLD with a faster t_{CO} or t_S the above system can be run faster. Often 22V10 PALs are offered with t_{CO} as low as 4 ns and t_S as low as 3 ns. The drawback is that two devices must be used to replace one CPLD.

Conclusion

Depth cascading of FIFOs to create logically larger devices is very common. The above discussion shows how the Cypress Synchronous fifos can be expanded using two different methods. Although the token passing approach shown for the CY7C42x5 devices is much simpler to implement, the CY7C42x1 approach using external logic allows tighter control over the number of words in the devices. This allows the user to generate more accurate $\overline{PAE}$ and $\overline{PAF}$ flags. With some modification, the VHDL code in Appendix A can be used to cascade the CY7C42x5 devices as well. Also, the code is intended to be scalable to allow cascading of any number of devices. The limitation to the amount of FIFOs which can be cascaded is based on available logic and bus loading to the data buses.

Appendix A. VHDL

```
ENTITY encntrl IS PORT (
   wclk:         IN           BIT;
   ff_A:         IN           BIT;
   ff_B:         IN           BIT;
   rs:           IN           BIT;
   wen:          IN           BIT;
   wen_A:        BUFFER       BIT;
   wen_B:        BUFFER       BIT;
   ff:           BUFFER       BIT;
   pae_A:        IN           BIT;
   pae_B:        IN           BIT;
   paf_A:        IN           BIT;
   paf_B:        IN           BIT;
   pae:          BUFFER       BIT;
   paf:          BUFFER       BIT;

   rclk:         IN           BIT;
   ef_A:         IN           BIT;
   ef_B:         IN           BIT;
   oe:           IN           BIT;
   oe_A:         BUFFER       BIT;
   oe_B:         BUFFER       BIT;
   ef:           BUFFER       BIT;
   ren:          IN           BIT;
   ren_A:        BUFFER       BIT;
   ren_B:        BUFFER       BIT );
   ren:          IN           BIT;
   ren_A:        BUFFER       BIT;
   ren_B:        BUFFER       BIT );


END encntrl;

use work.int_math.all;

ARCHITECTURE arch_encntrl OF encntrl IS

   type w_smachine is (w_reset, write_A, w_idle_A, write_B, w_idle_B);
   type r_smachine is (r_reset, read_A, r_idle_A, read_B, r_idle_B);
   signal w_present_state, w_next_state : w_smachine;
   signal r_present_state, r_next_state : r_smachine;
```

Appendix A. VHDL (continued)

```
BEGIN

w_state_comb: PROCESS (w_present_state, wen, rs, ff_A, ff_B)
    BEGIN
    case w_present_state is
          when w_reset  =>            wen_A <= '1';
                                      wen_B <= '1';
                                      if (rs = '1' and wen = '1') then
                                            w_next_state  <= write_A;
                                      else
                                            w_next_state <= w_reset;
                                      end if;

          when write_A  =>            wen_A <= '0';
                                      wen_B <= '1';
                                      if (rs = '0') then
                                            w_next_state <= w_reset;
                                      elsif (wen = '0') or (ff_A = '0') then
                                            w_next_state <= w_idle_A;
                                      elsif (ff_B = '0') then
                                            w_next_state <= w_idle_B;
                                      else
                                            w_next_state <= write_B;
                                      end if;

          when w_idle_A =>            wen_A <= '1';
                                      wen_B <= '1';
                                      if (rs = '0') then
                                            w_next_state <= w_reset;
                                      elsif  (wen = '1') and (ff_A = '1') then
                                            w_next_state <= write_A;
                                      else
                                            w_next_state <= w_idle_A;
                                      end if;

          when write_B  =>            wen_A <= '1';
                                      wen_B <= '0';
                                      if (rs = '0') then
                                            w_next_state <= w_reset;
                                      elsif (wen = '0') or (ff_B = '0') then
                                            w_next_state <= w_idle_B;
                                      elsif (ff_A = '0') then
                                            w_next_state <= w_idle_A;
```

Appendix A. VHDL (continued)

```
                                    else
                                          w_next_state <= write_A;
                                    end if;

          when w_idle_B =>          wen_A <= '1';
                                    wen_B <= '1';
                                    if (rs = '0') then
                                          w_next_state <= w_reset;
                                    elsif  (wen = '1') and (ff_A = '1') then
                                          w_next_state <= write_B;
                                    else
                                          w_next_state <= w_idle_B;
                                    end if;
    end case;

    ff <= ff_A OR ff_B;
    paf <= paf_A OR paf_B;

end PROCESS w_state_comb;

w_state_clocked: PROCESS (wclk)
    BEGIN
          IF (wclk'EVENT AND wclk = '1') THEN
                w_present_state <= w_next_state;
          END IF;
end process w_state_clocked;

r_state_comb: PROCESS (r_present_state, ren, rs, ef_A, ef_B)
    BEGIN
    case r_present_state is
          when r_reset  =>           ren_A <= '1';
                                    ren_B <= '1';
                                    if (rs = '1' and ren = '0') then
                                          r_next_state  <= read_A;
                                    else
                                          r_next_state <= r_reset;
                                    end if;

          when read_A   =>          ren_A <= '0';
                                    ren_B <= '1';
                                    if (rs = '0') then
                                          r_next_state <= r_reset;
                                    elsif (ren = '1') or (ef_A = '0') then
                                          r_next_state <= r_idle_A;
```

Appendix A. VHDL (continued)

```
                        elsif (ef_B = '0') then
                               r_next_state <= r_idle_B;
                        else
                               r_next_state <= read_B;
                        end if;

        when r_idle_A =>        ren_A <= '1';
                        ren_B <= '1';
                        if (rs = '0') then
                               r_next_state <= r_reset;
                        elsif  (ren = '0') and (ef_A = '1') then
                               r_next_state <= read_A;
                        else
                               r_next_state <= r_idle_A;
                        end if;

        when read_B  =>         ren_A <= '1';
                        ren_B <= '0';
                        if (rs = '0') then
                               r_next_state <= r_reset;
                        elsif (ren = '1') or (ef_B = '0') then
                               r_next_state <= r_idle_B;
                        elsif (ef_A = '0') then
                               r_next_state <= r_idle_A;
                        else
                               r_next_state <= read_A;
                        end if;

        when r_idle_B =>        ren_A <= '1';
                        ren_B <= '1';
                        if (rs = '0') then
                               r_next_state <= r_reset;
                        elsif  (ren = '0') and (ef_B = '1') then
                               r_next_state <= read_B;
                        else
                               r_next_state <= r_idle_B;
                        end if;
end case;

oe_A <= ren_A or oe;
oe_B <= ren_B or oe;
ef  <= ef_A or ef_B;
pae <= pae_A or pae_B;
```

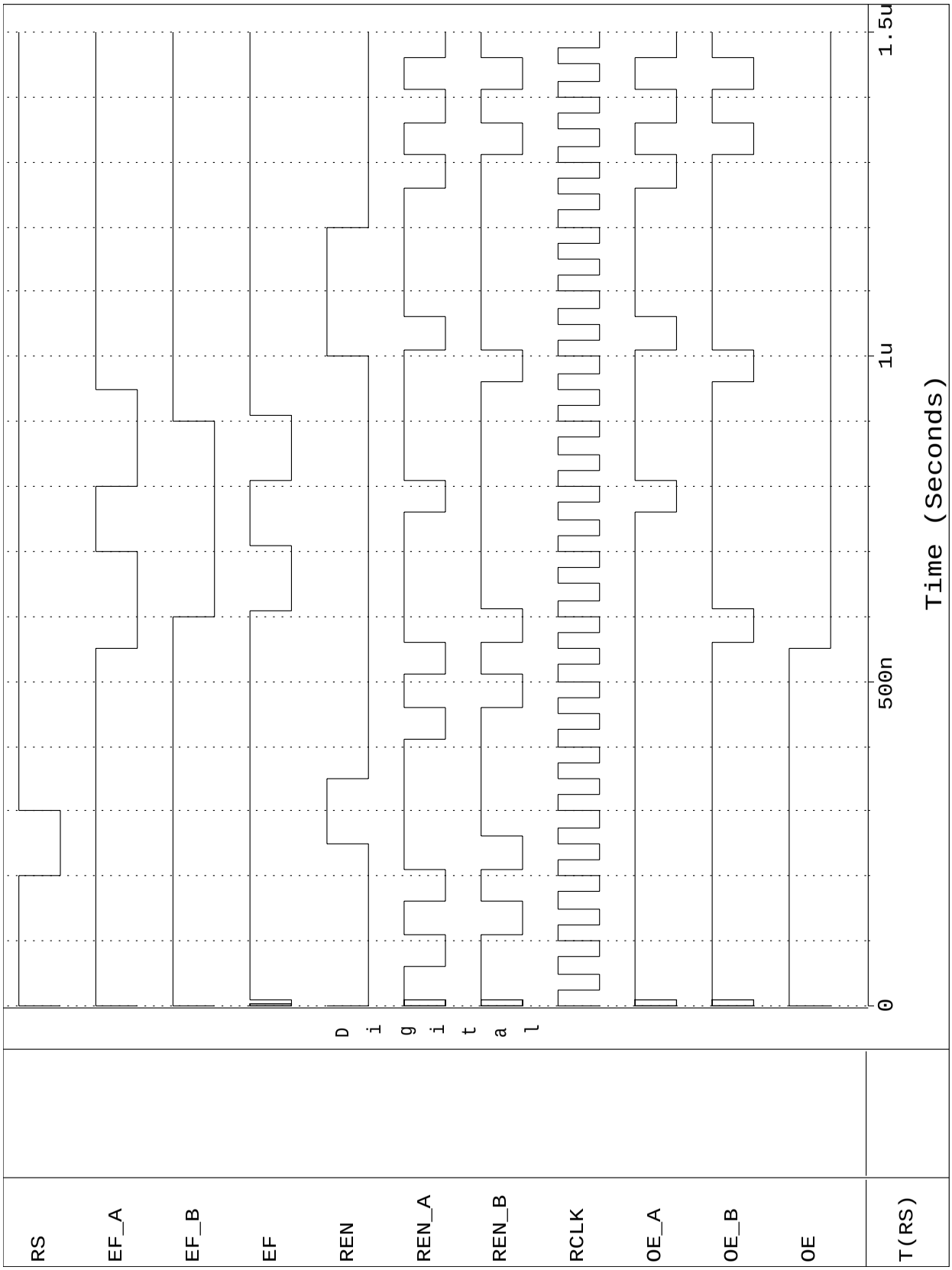
Appendix A. VHDL (continued)

```
end PROCESS r_state_comb;

r_state_clocked: PROCESS (rclk)
   BEGIN
          IF (rclk'EVENT AND rclk = '1') THEN
                r_present_state <= r_next_state;
          END IF;
end process r_state_clocked;

END arch_encntrl;
```

Appendix B. Read Timing



Appendix C. Write Timing