FIFO Dipstick Using *Warp2*™ VHDL and the CY7C371

Introduction

Programmable FIFO flags can often simplify the design of a digital system by automatically indicating a status that can prevent overrun or underrun in an elastic FIFO buffer. Although many FIFOs are available with on-chip programmable flag functions, these features are not available on industry-standard asynchronous FIFOs. Of those FIFOs that do have programmable flags, some do not allow the almost-empty and almost-full values to be programmed independently, or in some cases, for these values to be programmed to any specific word boundary. This application note presents a method by which FIFOs of any size may be monitored by an external Programmable Logic Device which will then generate all of the flags necessary for most FIFO applications. The FIFO Dipstick PLD behaves like a measuring device that can observe the level of data within a FIFO.

Application Description

A variable-length up-down counter is implemented with VHDL to measure the exact level of data within a FIFO. The number of bits required for the dipstick counter is dependent on the size of the FIFO and must satisfy the following equation:

2^n = FIFO Depth; Where: n = number of counter bits required

For example, a 2K FIFO would require an 11-bit counter. The n^{th} bit is necessary to prevent the dipstick counter from rolling over to zero when the last byte is written into the FIFO. In other words, the n^{th} bit will only be set when the FIFO is completely full.

Due to the truly asynchronous nature of the read and write ports of a FIFO, a state machine must be implemented to control the operation of the dipstick counter. This state machine must resolve the overlapping and nesting conditions that may occur with the FIFO_READ_L and FIFO_WRITE_L signals to the FIFO. For instance, multiple read pulses may occur within a single write pulse, read and write pulses may occur simultaneously, or read and write pulses may overlap by any amount of time.

The status of the almost-full and almost-empty flags is determined by simply comparing the dipstick counter value to pre-programmed levels and generating the appropriate combinatorial outputs. This method allows for the generation of any flag outputs required for a given application. The almost-full and almost-empty flags are the most typical levels required and are used to determine greater-than-or-equal-to and less-than-or-equal-to specified levels, respectively. Many possibilities exist, however, such as an approx-half-full flag, which could be used to add hysteresis to the half-full value of a FIFO.

Synchronous FIFO Ports

The VHDL/FLASH370™ implementation in this application note is based upon the following assumption. Both the read and write ports of the FIFO are controlled by clocked circuitry and the clocks for each port are synchronous to each other. This assumption allows a single clock to be used for the state machine and the counter. It also provides for the read, write, and reset inputs to be used without any chance of a metastable event occurring. As a result of this synchronous implementation, the almost-flags will change state combinatorially within

three clock cycles after the clock cycle that initiated the read or write. For instance, if a FIFO read is held active for two clock cycles followed by one clock cycle for read-recovery time, the updated almost-empty flag will be available during the read-recovery cycle.

Asynchronous FIFO Ports

The read and write ports of a FIFO may be controlled by clocked circuitry with clocks that are asynchronous to each other. In this case, the state machine and counter should be controlled by the one clock that best suits the application. If it is imperative that the write port of the FIFO receives the almost-full flag immediately, the write port clock should be used. If it is imperative that the read port of the FIFO receives the almost-empty flag immediately, the read port clock should be used. In either case, the read or write input from the opposite port needs to be synchronized to the dipstick's clock before it is used as a state machine input. The CY7C371 is ideally suited for this because of its dedicated inputs, which can be configured as single- or double-registered which will achieve a guaranteed 10-year MTBF. In addition, the port that is asynchronous to the dipstick's clock must also synchronize the almost-flags before use to prevent metastability problems. A negative aspect of using the FIFO dipstick in this application is that additional delays are introduced between a FIFO access and the almost-flags status change. These additional delays may or may not be tolerable, depending on the application.

State Machine Design

The finite state machine observes the FIFO_READ_L and FIFO_WRITE_L inputs in order to control the operation of the dipstick counter (see *Figure 1*). There are eight states required: an idle state, four counter-enabled states, and three counter-disabled states. The counter-enabled states are further categorized into count-up states (write and rd_hold_wr) and count-down states (read and wr_hold_rd). The counter-disabled states (rd_hold, wr_hold, rd_hold_wr_hold) are required for the FIFO_READ_L and FIFO_WRITE_L pulses that are active for greater than one clock cycle.

Within each state, all four permutations of FIFO_READ_L and FIFO_WRITE_L are evaluated to determine the next state. If neither signal is active, the state machine always returns to the idle state. If a single signal goes active or stays active, the state machine will progress to the appropriate state such that the counter-enabled states are active for a single clock cycle only, during each FIFO_READ_L and FIFO_WRITE_L pulse to the FIFO. If both FIFO_READ_L and FIFO_WRITE_L are observed going active on the same clock cycle, the counter-enabled states are avoided completely, allowing the dipstick counter to remain constant. The FIFO_RESET_L signal is not required as an input to the state machine because the dipstick counter will remain cleared if FIFO_RESET_L is active.

Warp2™ VHDL Implementation

The VHDL design used for the FIFO Dipstick is completely behavioral. This high-level design methodology eliminates any need to describe device specific implementations and it also allows for the most readability. The *Warp2* VHDL Compiler will synthesize the design into low-level components necessary for a CY7C371 automatically.

The design entity defines all the inputs and outputs of the design and assigns a type to these signals. The architecture describes the behavior of the circuit. See Appendix A for a listing of the code.

The entity declaration is comprised of a port map, a generic statement, and an attribute statement. The port map defines the FIFO Dipstick inputs, outputs, and the bidirectional counter bits for a variable-length up-down counter. The FIFO_READ_L, FIFO_WRITE_L, and FIFO_RESET_L inputs are written with a _l suffix to indicate that they are active LOW signals, all others are active HIGH. The generic statement is used as a convenient way to define the actual size of the dipstick counter. Simply defining the counter size once in this statement allows the entire design to be modified accordingly, including the number of bidirectional pins defined for the dipstick counter. The attribute statement has been included to define the CY7C371 as the PLD for which

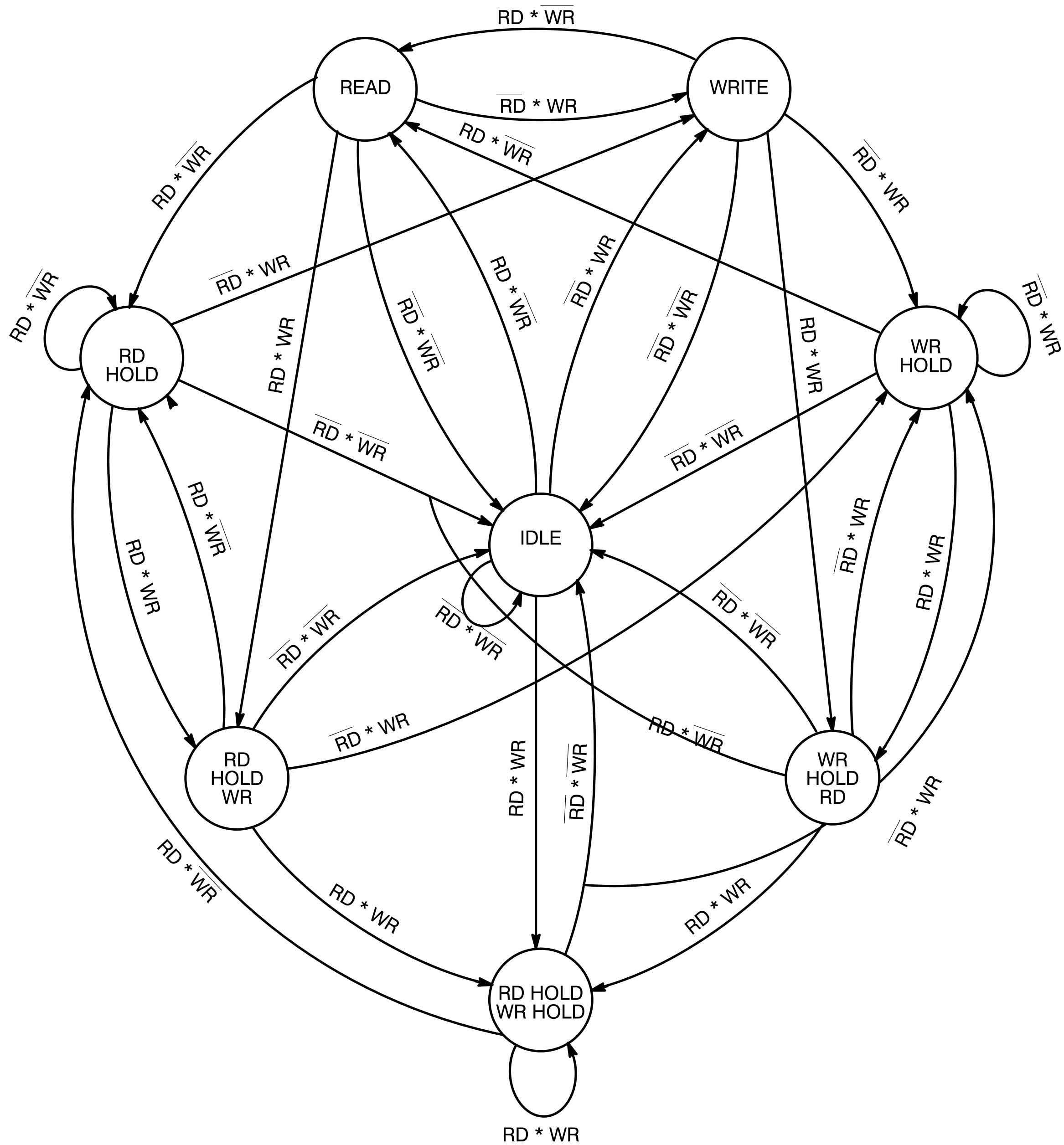

Figure 1. FIFO Dipstick State Machine Bubble Chart

Warp2 will generate a JEDEC file. This statement could be deleted if the “C371” is chosen from the device options menu of *Warp2* instead of the default device.

The architecture body of the design is comprised of three separate processes which will execute in parallel. The process titled *outputs* defines the output flags as a function of the dipstick counter level. Relational operators are used to compare the counter bit_vector to the integer constants defined at the beginning of the architecture. Comparing a bit_vector to an integer is typically not allowed in VHDL, however, *Warp2*’s int_math package provides this capability. Since there is no wait statement included, the afull and aempty signals are combinatorial outputs from this process.

The process titled *counter* controls the operation of the FIFO dipstick counter. If the FIFO_RESET_L input is asserted, then the counter is cleared on the next clock edge. This is accomplished by using a *Warp2* function-call titled i2bv, which converts the integer constant "zero_count" to a bit_vector of the appropriate length. The result is then assigned as the new counter value which is a bit_vector of all zeros. If the FIFO_RESET_L input is not active, the counter operation is then determined by the state of the dipstick state machine and the current counter value. The count-up states will increment the counter unless the counter's MSB is set indicating that the maximum count has been reached. The count-down states will decrement the counter unless it is currently equal to zero. If none of the conditions described above exist, then the counter will maintain its current value as indicated in the else statement. The inc_bv and dec_bv functions, used to increment and decrement the counter bit_vector respectively, are provided by *Warp2* as part of the bv_math package in the work library.

The process titled *state_machine* implements the finite state machine, as represented in the bubble chart of *Figure 1*. This behavioral description makes use of the enumerated-type form. The major advantage of this form is that the state encoding can be easily changed by the user. The current encoding options available are sequential, gray, one-hot, and user-defined and are determined by the attribute state_encoding. The enumerated type is defined at the beginning of the architecture body and is comprised of the eight state names. The case statement within the process defines the state machine transitions based on the current state and the FIFO_READ_L and FIFO_WRITE_L inputs.

Differences From Programmable FIFOs

The following two differences between the FIFO Dipstick design and the use of a FIFO with programmable flags must be understood. First, the latency incurred between a FIFO access and the update of flag status may be prohibitive; refer to the synchronous and asynchronous FIFO ports sections above. Second, the flag outputs of a FIFO will always go inactive based on a FIFO strobe going inactive, whereas the FIFO Dipstick solution will always change flag states based on the strobes going active.

Summary

This application note provides the information required to implement programmable flags for any size FIFO by simply changing the values in the VHDL statements of Appendix A, which are noted as application specific in the source code. For applications that require dynamically alterable flags, a microprocessor port is easily adaptable. The design in Appendix A is also easily adaptable to different FIFO applications, i.e., clocked FIFOs, BiFIFOs, FIFOs with asynchronously clocked ports, etc.

Appendix A. FIFO Dipstick *Warp2* VHDL Source Code

```
USE work.bv_math.all;
USE work.int_math.all;

ENTITY dipstick IS
GENERIC (counter_size: INTEGER := 16);-- APPLICATION SPECIFIC
   PORT (clock, fifo_reset_l, fifo_rd_l, fifo_wr_l: IN BIT;
      afull, aempty: OUT BIT := '0';
      counter: INOUT BIT_VECTOR(counter_size DOWNTO 0));
   ATTRIBUTE part_name OF dipstick: ENTITY IS "C371";
END dipstick;

ARCHITECTURE behavior OF dipstick IS
   CONSTANT afull_value: INTEGER := 32000;-- APPLICATION SPECIFIC
   CONSTANT aempty_value: INTEGER := 07;-- APPLICATION SPECIFIC

   TYPE fifostate IS
      (idle, read, rd_hold, rd_hold_wr, rd_hold_wr_hold,
       write, wr_hold, wr_hold_rd);
   SIGNAL nextstate: fifostate;
   ATTRIBUTE state_encoding OF fifostate: TYPE IS SEQUENTIAL;

BEGIN
   outputs: PROCESS BEGIN
      IF (counter >= afull_value) THEN
         afull <= '1';
         aempty <= '0';
      ELSIF (counter <= aempty_value) THEN
         afull <= '0';
         aempty <= '1';
      ELSE
         afull <= '0';
         aempty <= '0';
      END IF;
   END PROCESS;

   counter: PROCESS
      CONSTANT zero_count: INTEGER := 0;
   BEGIN
   WAIT UNTIL (clock = '1');
      IF (fifo_reset_l = '0') THEN
                counter <= I2BV(zero_count, counter_size);
      ELSIF ((nextstate = write) OR (nextstate = rd_hold_wr)) AND
            (counter(counter_size-1) = '0') THEN
                counter <= inc_bv(counter);
      ELSIF ((nextstate = read) OR (nextstate = wr_hold_rd)) AND
            (counter /= zero_count) THEN
                counter <= dec_bv(counter);
      ELSE      counter <= counter;
      END IF;
   END PROCESS;
```

Appendix A. FIFO Dipstick *Warp2* VHDL Source Code (continued)

```
state_machine: PROCESS
BEGIN
WAIT UNTIL (clock = '1');

CASE nextstate IS
   WHEN idle =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= read;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= write;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr_hold;
      END IF;
   WHEN read =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= rd_hold;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= write;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr;
      END IF;
   WHEN rd_hold =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= rd_hold;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= write;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr;
      END IF;
   WHEN rd_hold_wr =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= rd_hold;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= wr_hold;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr_hold;
      END IF;
   WHEN rd_hold_wr_hold =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
```

Appendix A. FIFO Dipstick *Warp2* VHDL Source Code (continued)

```
         ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
            nextstate <= rd_hold;
        ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
            nextstate <= wr_hold;
         ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
            nextstate <= rd_hold_wr_hold;
         END IF;
      WHEN write =>
         IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
            nextstate <= idle;
         ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
            nextstate <= read;
         ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
            nextstate <= wr_hold;
         ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
            nextstate <= wr_hold_rd;
         END IF;
      WHEN wr_hold =>
         IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
            nextstate <= idle;
         ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
            nextstate <= read;
         ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
            nextstate <= wr_hold;
         ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
            nextstate <= wr_hold_rd;
         END IF;
      WHEN wr_hold_rd =>
         IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
            nextstate <= idle;
         ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
            nextstate <= rd_hold;
         ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
            nextstate <= wr_hold;
         ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
            nextstate <= rd_hold_wr_hold;
         END IF;
      WHEN OTHERS =>
          nextstate <= idle;
   END CASE;
   END PROCESS;
END behavior;
```