Asynchronous FIFO Architectures

Vijay A. Nebhrajani

Designing a FIFO is one of the most common problems an ASIC designer comes across. This series of articles is aimed at looking at how FIFOs may be designed -- a task that is not as simple as it seems.

At the onset, note that FIFOs are usually used for domain crossing, and are therefore dual clock designs. In other words, the design works off two clocks, and so the most general case is a FIFO that is designed with no relationship assumed between these two clocks. However, we will not start with such an architecture -- we will start with the trivial case of a FIFO that runs on just one clock. I imagine that such a circuit will have limited use in practice, but it serves the very useful purpose of setting the stage for more complex designs.

This series of articles is designed to follow the course of development of FIFO designs from the trivial to the general. This will be the series of articles:

- Single clock architecture
- Dual clock architectures - Architecture 1.
- Dual clock architectures - Architecture 2.
- Dual clock architectures - Architecture 3.
- Pulse mode FIFO.

The trivial case of a single clock FIFO

There are several architectures that are possible for a FIFO; and these include ripple FIFOs, shift registers and other such architectures that we will not care much about. We will concentrate on architectures that involve random access memory arrays. Such an architecture is shown in Fig (vijayn?)

Analyzing, we see that there is a RAM array with separate read and write ports. This is chosen for convenience. If you has a single port memory, you would have to include an arbiter that would grant access to one operation (read or write) at one time. We choose Dual Port RAMs (not necessarily true dual port, because we simply want a separate read and write port) because these illustrate more realistic situations.

The read and write ports have separate read and write addresses, generated by two counters of width log_2(`ARRAY_SIZE`), where `ARRAY_SIZE` is the depth of the DPRAM array in locations (not bytes). We don't care much about the data width right now, but this does become an important parameter in choosing architectures later. For consistency we will refer to these counters as the "read pointer" and the "write pointer". The write pointer points to the location that will be written next, and the read pointer points to the location that will be read next. A write increments the write pointer, and a read increments the read pointer.

The last block we see is the "status" block. The responsibility of this block is to generate the "empty" and "full" signals to the FIFO. These signals tell the outside world that the FIFO has reached a terminal condition: If "full" is active, then the FIFO has reached a terminal condition for write and if "empty" is active, the FIFO has reached a terminal condition for read. A terminal condition for write implies that the FIFO has no more space to accommodate more data and a terminal condition for read implies that the FIFO has no more data available for readout. The status block may also report the number of empty or full locations in the FIFO, and this is accomplished by an arithmetic operation on the pointers.

The actual count of empty or full locations does not play much of a part in the FIFO itself; it is used as a reporting mechanism to the outside world. However, the empty and full signals play a very important role within the FIFO – they block access to further reads and writes respectively. The importance of this blocking does not lie in the fact that data may be overwritten (or read out twice); the critical importance lies in the fact that pointer positions are the only control we have over the FIFO, and writes or reads change the pointers. If we do not block the pointers from changing state at terminal conditions, we will have a FIFO that either "eats" data or "generates" data, and that is quite unacceptable.

Further analysis: The DPRAM could have "registered" reads -- this means that the output data from the array is registered. If this is so, the read pointers will have to be designed as "read and increment". This means that you will have to provide an explicit read signal before the data at the output of the FIFO is valid. On the other hand, if the DPRAM does not have registered outputs, valid data is available as soon as it is written; you read this data first, and increment the pointer later. This affects the logic that reads data out from the FIFO and the logic that performs the empty/full calculation. For the sake of simplicity we will only treat the case where the DPRAM does not provide registered outputs. It is not very complex to extend the same reasoning (that we will use) to the case of a registered output DPRAM.

Functionally the FIFO works as follows: At reset, the pointers are both 0. This is the empty condition of the FIFO, and empty is pulled high (we will use the active high convention) and full is low. At empty, reads are blocked and so the only operation possible is write. A write loads location 0 of the array and increments the write pointer to 1. This causes the empty signal to go low. Assuming that there are no reads and subsequent cycles only write to the FIFO, there will be a time when the write pointer will equal ARRAY_SIZE - 1. This means that the last location in the array is the next location that will be written to. At this condition, a write will cause the write pointer to become 0, and set full.

Note that in this condition the write and read pointers are equal, but the FIFO is full, and not empty. This implies that the full/empty decision is not based on the pointer values alone, but on the operation that caused the pointers to become equal.

Now assume that we begin a series of reads. Each read will increment the read pointer, to the point where the read pointer equals ARRAY_SIZE - 1. At this point, the data from this location is available on the output bus of the FIFO. Succeeding logic reads this data and provides a read signal (active for one clock). This causes the read pointer to become equal

to the write pointer again (after both pointers have completed one cycle through the array). However, since this equality came about because of a read, empty is set.

Thus, we have the following for the empty flag:

- A write unconditionally clears empty.
- Read Pointer = `(ARRAY_SIZE - 1)` and a read sets empty.

and the following for the full flag:

- A read unconditionally clears full.
- Write Pointer = `(ARRAY_SIZE - 1)` and a write sets full.

However, this is a special case, since in general reads may start as soon as the FIFO is not empty (the reading logic need not wait for the FIFO to become full), so these conditions have to be modified to accommodate any read pointer and write pointer values.

A little thought shows that we have organized the array as a circular list. Thus, if the write pointer is numerically greater than the read pointer by 1 and a read occurs, the FIFO is empty. This works just fine for the boundary case described above as long as we use unsigned (n-bit) arithmetic. Similarly, if the read pointer is numerically greater than the write pointer by 1 and a write occurs, the FIFO is full.

This leads to the final conditions:

- A write unconditionally clears empty.
- (write pointer = read pointer + 1) and a read sets empty.

- A read unconditionally clears full.
- (read pointer = write pointer + 1) and a write sets full.

Note that a simultaneous read and write increments both pointers, but does not alter the state of the empty and full flags. A simultaneous read and write is not allowed at the full and empty boundaries.

With this we can now define the status block of the FIFO. I am providing the code in VHDL, but since this is synthesizable, translation to Verilog is easy.

```
library IEEE, STD;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity status is
  port     (reset            : in   std_logic;
            clk              : in   std_logic;
            fifo_wr          : in   std_logic;
            fifo_rd          : in   std_logic;
            valid_rd         : out  std_logic;
```

```
            valid_wr         : out  std_logic;
            rd_ptr           : out  std_logic_vector(4 downto 0);
            wr_ptr           : out  std_logic_vector(4 downto 0);
            empty            : out  std_logic;
            full             : out  std_logic
           );
end status;

architecture status_A of status is
  signal rd_ptr_s     : std_logic_vector(4 downto 0);
  signal wr_ptr_s     : std_logic_vector(4 downto 0);
  signal valid_rd_s   : std_logic;
  signal valid_wr_s   : std_logic;

begin
  empty_P : process(clk, reset)

  begin
    if (reset = '1') then
      empty <= '1';

    elsif (clk'event and clk = '1') then
      if (fifo_wr = '1' and fifo_rd = '1') then
        -- do nothing
        null;

      elsif (fifo_wr = '1') then
        -- write unconditionally clears empty
        empty <= '0';

      elsif (fifo_rd = '1' and (wr_ptr_s = rd_ptr_s + 1)) then
        -- set empty
        empty <= '1';

      end if;
    end if;

  end process;

  full_P : process(clk, reset)

  begin
    if (reset = '1') then
      full <= '0';

    elsif (clk'event and clk = '1') then
      if (fifo_rd = '1' and fifo_wr = '1') then
```

```
      -- do nothing
      null;

    elsif (fifo_rd = '1') then
      -- read unconditionally clears full
   full <= '0';

    elsif (fifo_wr = '1' and (rd_ptr_s = wr_ptr_s + 1)) then
      -- set full
      full <= '1';

    end if;
  end if;

end process;

valid_rd_s <= '1' when (empty = '0' and fifo_rd = '1');
valid_wr_s <= '1' when (full  = '0' and fifo_wr = '1');

wr_ptr_s_P : process(clk, reset)

begin
  if (reset = '1') then
    wr_ptr_s_P <= (others => '0');

  elsif (clk'event and clk = '1') then
    if (valid_wr_s = '1') then
      wr_ptr_s <= wr_ptr_s + '1';

    end if;
  end if;

end process;

rd_ptr_s_P : process(clk, reset)

begin
  if (reset = '1') then
    rd_ptr_s_P <= (others => '0');

  elsif (clk'event and clk = '1') then
    if (valid_rd_s = '1') then
      rd_ptr_s <= rd_ptr_s + '1';

    end if;
  end if;
```

```
  end process;

  rd_ptr <= rd_ptr_s;
  wr_ptr <= wr_ptr_s;

end status_A;
```

The circuit for this is shown in Fig (vijayn?).

The observant reader will notice that generating the full or empty flags requires the use of both pointers. In the case of a dual clock design, the read pointer is expected to work off the read clock and the write pointer off the write clock. This raises unexpectedly thorny issues -- you may try and see for yourself. These issues and some solutions will be covered in future articles in this series.