Designing Asynchronous FIFO

Dadhania Prashant C.

Department of Electronics Engineering,
Gujarat Technological University,
Gandhinagar, Gujarat, India.

Prashant3730@gmail.com

ABSTRACT: FIFOs are often used to safely pass data from one clock domain to another asynchronous clock domain. Using a FIFO to pass data from one clock domain to another clock domain requires multi-asynchronous clock design techniques. There are many ways to design a FIFO wrong. There are many ways to design a FIFO right but still makeit difficult to properly synthesize and analyze the design.
This paper will detail one method that is used to design, synthesize and analyze a safe FIFO between different clockdomains using Gray code pointers that are synchronized into a different clock domain before testing for "FIFO full"or "FIFO empty" conditions. The fully coded, synthesized and analyzed RTL Verilog isincluded.

KEYWORDS:*FIFO, Asynchronous FIFO, Gray Counter*

1. Introduction

An asynchronous FIFO refers to a FIFO design where data values are written to a FIFO buffer from one clockdomain and the data values are read from thesame FIFO buffer from another clock domain, where the two clockdomains are asynchronous to eachother.

Asynchronous FIFOs are used to safely pass datafrom one clock domain to another clock domain.

There are many ways to do asynchronous FIFOdesign, including many wrong ways. Most incorrectly implementedFIFO designs still function properly 90% of the time. Most almost-correct FIFO designs function properly 99%+ ofthe time. Unfortunately, FIFOs that work properly 99%+ of the time have design flaws that are usually the mostdifficult to detect and debug (if you are lucky enough to notice the bug before shipping the product), or the mostcostly to diagnose and recall (if the bug is not discovered until the product is in the hands of a dissatisfied customer).

This paper discusses one FIFO design style and important details that must be considered when doing asynchronous
FIFO design.

2. Asynchronous FIFO Pointers

In order to understand FIFO design, one needs to understand how the FIFO pointers work.
There are mainly two pointers.

1. Write Pointer
2. Read Pointer

The write pointer alwayspoints to the next word to be written; therefore, on reset, both pointers are set to zero, which also happens to be thenext FIFO wordlocation to be written. On a FIFO-write operation, the memory location that is pointed to by the write pointer is written, and then the write pointer isincremented to point to the next location to bewritten.
Similarly, the read pointer always points to the current FIFO word to be read. Again on reset, both pointers are set tozero, the FIFO is empty and the read pointer is pointing to invalid data (because the FIFO is empty and the emptyflag is asserted). As soon as the first data word is written to the FIFO, the write pointer increments, the empty flag iscleared, and the read pointer that is still addressing the contents of the first FIFO memory word, immediately drivesthat first valid word onto the FIFO data output port, to be read by the receiver logic. The fact that the read pointer isalways pointing to the next FIFO word to be read means that the receiver logic does not have to use two clockperiods to read the data word. If the receiver first had to increment the read pointer before reading a FIFO data word,the receiver would clock once to output the data word from the FIFO, and clock a second time to capture the dataword into the receiver. That would be needlessly inefficient.

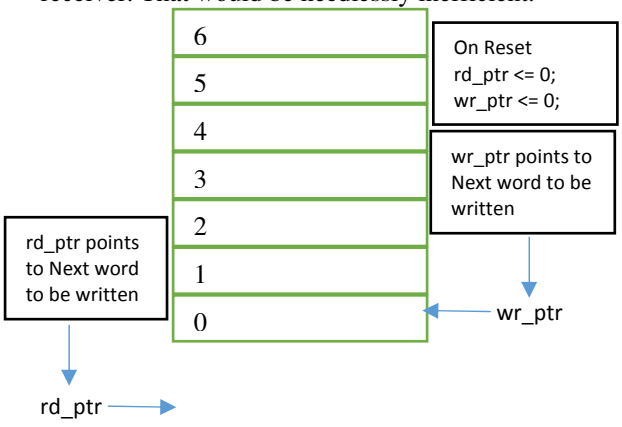

Figure 1FIFO Pointers

3. Handling Full and Empty Conditions

The FIFO design in this paper ensures that the empty flag will be generated in the read-clock domain to insure thatthe empty flag is detected immediately when the FIFO buffer is empty and similarly full flag will be generated in the write-clock domain to insure that thefull flag is detected immediately when the FIFO buffer is full

Here, in this FIFO design a status counter (status_cnt) will be take care of FIFO full and empty conditions. This status counter will be incremented on every write and will be decremented on every read transaction. When the status counter reaches the maximum FIFO depth it will assert FIFO full signal and when its value is zero it will assert FIFO empty signal. This status counter also ensure that there is no loss of data i.e. it ensures the "write before read conditions" and "read before wire condition". On reset the status counter is set to zero value and empty signal will be asserter.

The write enable and read enable will be dependent on write request and read request and empty and full bit.
Write enable == write req. and ~ (not of) full
Read enable == read req. and ~ (not of) empty

4. Use of Grey Counter

A common approach to FIFO counter-pointers, is to use Gray code counters. Gray codes only allow one bit tochange for each clock transition, eliminating the problem associated with trying to synchronize multiple changingsignals on the same clock edge.

The first fact to remember about a Gray code is that the code distance between any two adjacent words is just 1 (onlyone bit can change from one Gray count to the next). The second fact to remember about a Gray code counter is thatmost useful Gray code counters must have power-of-2 counts in the sequence. It is possible to make a Gray codecounter that counts an even number of sequences but conversions to and from these sequences is generally not assimple to do as the standard Gray code. Also note that there are no odd-count-length Gray code sequences so onecannot make a 23-deep Gray code. This means that the technique described in this paper is used to make a FIFO thatis 2^n deep.

Now to compare Mealy and Moore machine one example is taken. Consider the case of a circuit to detect a pair of 1's or 0's in the single bit input.
If two one's or two zero's comes one after another, output should go high. Otherwise output should be low.

Here is a Moore type state transition diagram for the circuit:

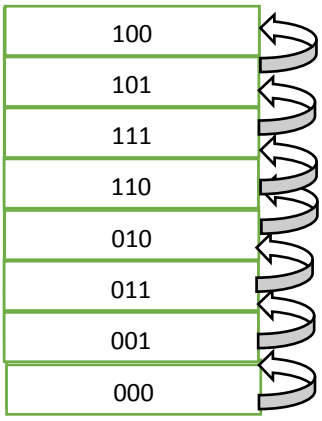

Figure 2 Gray counter sequence

5. RTL CODE Async FIFO

```
module async_fifo
#(parameter DATA_WIDTH = 32,
parameter ADDR_WIDTH = 4,
parameter FIFO_DEPTH= (1 << ADDR_WIDTH))
    //Reading port
(output reg  [DATA_WIDTH-1:0]       data_out,
          // Data Output
output wire                         empty,
          // FIFO Empty
input wire                          pop_req_en,// Read Enable
input wire                          rclk,       // Read clock
    //Writing port.
input wire  [DATA_WIDTH-1:0]        data_in,
          // Data Input
output wire                         full,
          // FIFO Full
input wire                                  push_req_en,// Write
Enable
input wire                          wclk,       // Write clock
input wire                          rst);       // Reset signal


//-----------Internal variables-------------------
wire [ADDR_WIDTH-1:0] wr_pointer;
wire [ADDR_WIDTH-1:0] rd_pointer;
reg [ADDR_WIDTH :0] status_cnt;
wire                                write_en, read_en;
reg [DATA_WIDTH-1:0] mem [0:FIFO_DEPTH-1];

//-----------Variable assignments---------------
assign full = (status_cnt == (FIFO_DEPTH-1));
assign empty = (status_cnt == 0);

//Addreses (Gray counters) logic:
gray_count gray_count_w
      (.graycount(wr_pointer),
       .enable(write_en),
       .rst(rst),
       .clk(wclk)
```

```
      );


gray_count gray_count_r
      (.graycount(rd_pointer),
       .enable(read_en),
       .rst(rst),
       .clk(rclk)
      );


  //////////////Code///////////////

   //Data ports logic:
   //(Uses a dual-port RAM).
   //'data_out' logic:
always @ (posedge rclk)
          begin
                    if (rst)
                    begin
                              status_cnt <= 0;
                    end
                    else if (pop_req_en & !empty)
                    begin
                    data_out<= mem[rd_pointer];
                    status_cnt <= status_cnt - 1;
                    end
          end
   //'data_in' logic:
always @ (posedge wclk)
          begin
                    if (rst)
                    begin
                              status_cnt <= 0;
                    end
                    else if (push_req_en & !full)
                    begin
                    mem[wr_pointer] <= data_in;
                    status_cnt <= status_cnt + 1;
                    end
          end
   //Fifo addresses support logic:
   //'Next Addresses' enable logic:
assign write_en = push_req_en & ~full;
assign read_en  = pop_req_en  & ~empty;
endmodule
```

4. Results:

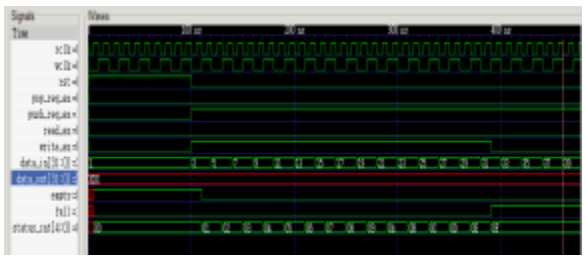

Figure 3 Write operation

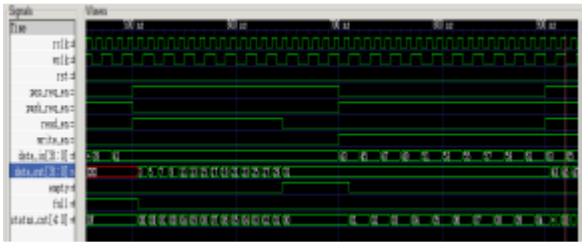

Figure 4 Read Operation

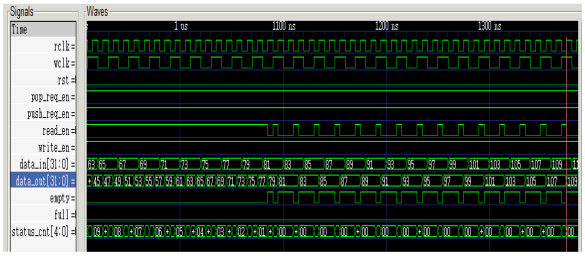

Figure 5 Write/Read Operation simultaneously

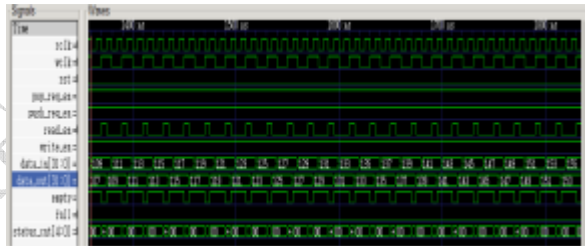

Figure 6 Write/Read Operation simultaneously

7. Conclusion:

Asynchronous FIFO design requires careful attention to details from pointer generation techniques to full and emptygeneration. Ignorance of important details will generally result in a design that is easily verified but is also wrong.Finding FIFO design errors typically requires simulation of a gate-level FIFO design with backannotation of actualdelays and a whole lot of luck!

Synchronization of FIFO pointers into the opposite clock domain is safely accomplished using Gray code pointers.The techniques described in this paper should work with asynchronous clocks spanning small to large differences in speed.

8. Reference:

1. IEEE paper "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs,"
2. IEEE standard Verilog Reference Manual , IEEE standard 2001
3. Design Compiler User Guide from Synopsys
4. Verilog HDL by Samir Palnitkar