NEW ASYNCHRONOUS FIFO DESIGN

Asynchronous FIFO - General Working

Verilog code for Asynchronous FIFO and its verilog test bench code are already given in previous posts.

Let us have a small recap of asynchronous FIFO working and then we will go to new asynchronous FIFO design.

The general block diagram of asynchronous FIFO is shown in Figure (1). Functionality wise mainly we can distinguish four blocks in this diagram. They are: dual port RAM, read pointer logic, write pointer logic and synchronizer. Dual port RAM has two ports-one is for reading and the other one is for writing operation. These two accesses of the FIFO are independent of each other and are completely controlled by read pointer logic and write pointer logic. Number of memory locations of the FIFO varies from 8 locations to some kilobytes. The data width of each location is also varying from one to 256 bits depending on the applications and technology. Modern day FIFOs provide options to program both of the above parameters as per requirements.

Data is written sequentially into the FIFO and read sequentially such that the first data written is the first data read out and so on with the remaining sequential data. Thus architecture of FIFO is completely characterized by these two independent operations. Dual port RAM and read-write logic circuits with synchronizers accomplish this task. Read port has its associated memory addressing logic called as ‘read pointer’ logic and write port has ‘write pointer’ logic. When FIFO is reset both read and write pointers point to first memory location of the FIFO. As and when data is written to FIFO write pointer gets incremented and points to next memory location. Similarly when read operation takes place read pointer gets incremented for every read. Both pointer works in circular fashion i.e. after reaching the last position it will jump to first location of the FIFO.

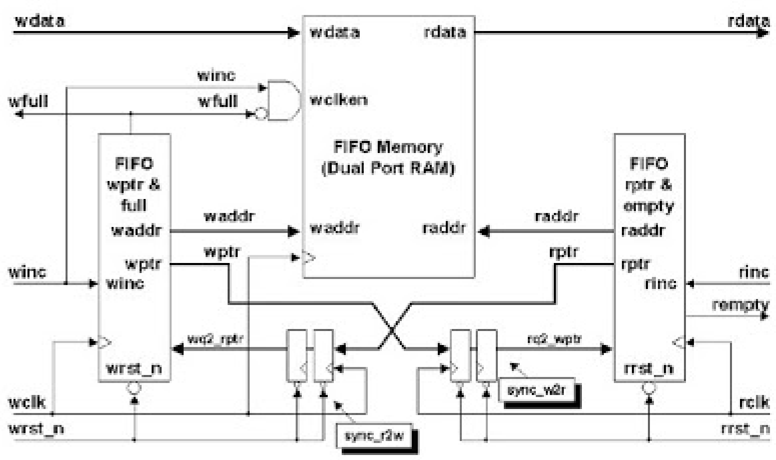

Figure (1) General approach of FIFO design [1]

‘Full flag’ and ‘empty flag’ are used to detect the status of the FIFO. These two flags are generated depending on the comparison result of FIFO pointers. Full flag is asserted when FIFO is completely full. Empty flag is asserted when FIFO is empty. Assertion of full flag indicates that no data can be written further unless at least one data is read out of the FIFO. Assertion of empty flag indicates the condition that no more data can be read from the FIFO unless until at least one data is written to the FIFO.

Even after the assertion of full flag, if data is written to FIFO ‘overflow’ condition occurs. Similarly after the assertion of empty flag if read operation is performed then ‘underflow’ occurs. Either overflow or underflow condition causes the data corruption or data loss. Safe and reliable FIFO designs always avoid both extreme conditions.

New asynchronous FIFO Design

A new asynchronous FIFO design is presented here. The concept of using pointer difference for determining the FIFO status is already used in synchronous FIFO designs. [3]. Here same concept is extended to asynchronous FIFO. The corresponding block diagram is shown in the Figure (2). The block diagram consists of a dual port RAM, two 4 bit binary up-counters, address pointer gap generation logic, full and empty condition generation logic, next read control logic and next write control logic.

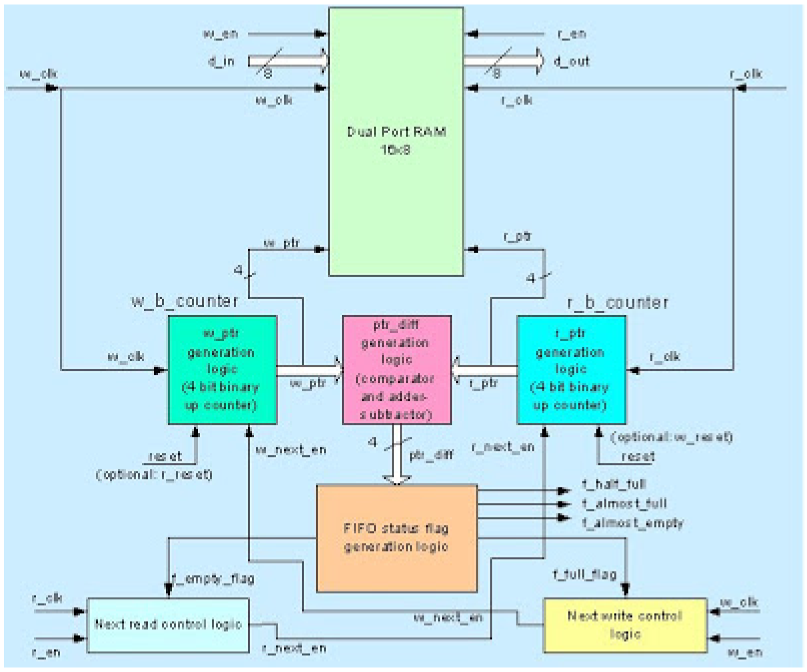

Figure (2) new proposed asynchronous FIFO

The naming convention used is explained below:

- **d_in**: input data; 8 bit width is considered
- **d_out**: output data; 8 bit width is considered
- **w_en**: write enable signal
- **r_en**: read enable signal
- **r_next_en**: read next enable
- **w_next_en**: write next enable
- **w_clk:** write clock; 10 MHz for this design
- **r_clk:** read clock; 50 MHz for this design
- **w_ptr:** write address pointer; 4 bit to address depth of 16
- **r_ptr:** read address pointer; 4 bit to address depth of 16
- **ptr_diff:** address pointer difference; 4 bit width
- **f_full_flag:** FIFO full flag; asserted when FIFO is full
- **f_empty_flag:** FIFO empty flag; asserted when FIFO is empty

- **f_half_full_flag:** FIFO half full flag; asserted when FIFO is half full
- **f_almost_full_flag**: FIFO almost full flag; asserted when FIFO is almost full(configurable)
- **f_almost_empty_flag**: FIFO almost empty flag; asserted when FIFO is almost empty (configurable**)**

Dual port RAM

For this design depth of the RAM is considered to be 16 and the width is 8. . The RTL code synthesizes to a distributed dual port RAM as shown in Figure (3). LUTs are used as dual port RAM.

```
//-----------------------------------------------

always @(posedge w_clk) //write clock domain

begin

if(w_en) begin //if write enabled and FIFO

if(!f_full_flag) //is not full write data to

f_memory[w_ptr]<=d_in; end //to FIFO

end

//----------------------------------

always @(posedge r_clk) //read clock domain

begin

if(reset) //if reset make output zero

d_out<=0;//f_memory[r_ptr];

else if(r_en) begin //if read enabled and FIFO is

if(!f_empty_flag) //not empty read data from

d_out<=f_memory[r_ptr]; end //FIFO

else d_out<=0;

end
```

//----------------------------------

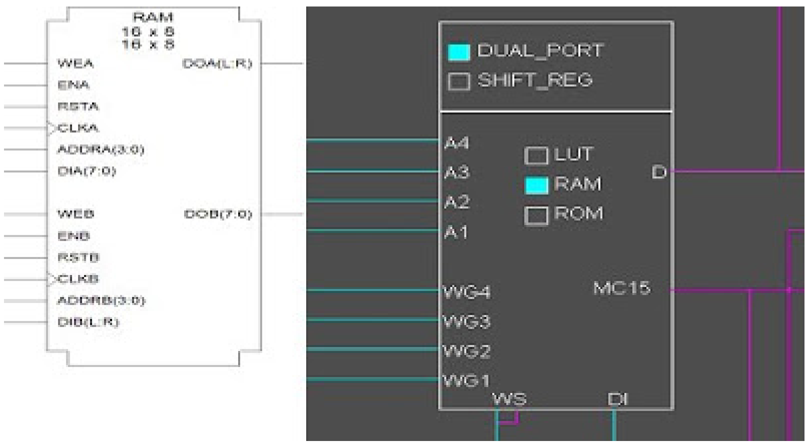

Figure (3) Distributed Dual Port RAM-RTL schematic and Xilinx FPGA editor

Data is written to FIFO memory only if FIFO is not full and write enable signal w_en is enabled. Similarly data is read out of memory location only if FIFO is not empty and read enable signal r_en is active

Binary Counters

Four bit binary up counters are used to generate address for read and write port. These address generators have external reset and enable signals called as w_next_en and r_next_en which are generated and controlled by next write control logic and next read control logic. Presently single reset signal resets the both counters. Individual resets also can be easily provided which is elaborated below.

Four bit up counter is designed as a separate module. This is instantiated in main module to realize read binary counter (i.e. read address generator) r_b_counter and write binary counter (i.e. write address generator) w_b_counter. Single reset input is mapped for both counters which resets both read and write pointers. **If individual read reset and write reset are required in the design then simply map these separate reset inputs to instantiated counter reset.** RTL code for the counter is given below. RTL schematic is showed in the Figure (4).

```
//------------------------------------------------
module b_counter(c_out,c_reset,c_clk,en); //separate module for counter
parameter c_width=4; //counter width
output [c_width-1:0] c_out; reg [c_width-1:0] c_out;
input c_reset,c_clk,en;
always @(posedge c_clk or posedge c_reset)
if (c_reset)
c_out <= 0; //if reset counter output is zero
else if(en)
c_out <= c_out + 1; //if count enabled count up
endmodule
//----------------------------------------------
```

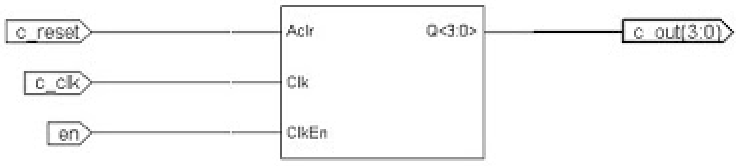

Figure (4) binary up counter-RTL schematic

Binary counter is instantiated in the a_fifo5.v file to obtain read address generator r_b_counter and write address generator w_b_counter. The instantiation code is given below.

```
//-------------------------------------
b_counter //instantiation in main
coder_b_counter(.c_out(r_ptr),.c_reset(reset),.c_clk(r_clk),.en(r_next_en));
b_counter //instantiation in main
```

```
codew_b_counter(.c_out(w_ptr),.c_reset(reset),.c_clk(w_clk),.en(w_next_en)
);
//-------------------------------------
```

Address pointer gap generation logic

The blocks compare the read and write addresses and gives out the difference of two address pointers. If w_ptr>r_ptr, it finds the gap using relation, ptr_diff=w_ptr-r_ptr. If w_ptrThen ptr_diff=10-5=5. If w_ptr=5 and r_ptr=10, then ptr_diff = (16-10) + 5=11. From the RTL schematic we can find that this block consists of comparators and adder-subtractor. These modules always calculate the difference between r_ptr and w_ptr whenever there is a change in r_ptr and w_ptr. Thus ptr_diff dynamically reflects the status of the FIFO. This flexibility of the design allows us to use any read and write clock frequencies, within the maximum operating frequency.

```
//-------------------------------------

always @(*) //pointer difference is evaluated for both clock edges

begin

if(w_ptr>r_ptr)

ptr_diff<=w_ptr-r_ptr;

else if(w_ptr)

ptr_diff<=((f_depth-r_ptr)+w_ptr);

else ptr_diff<=0; end

//-------------------------------------
```

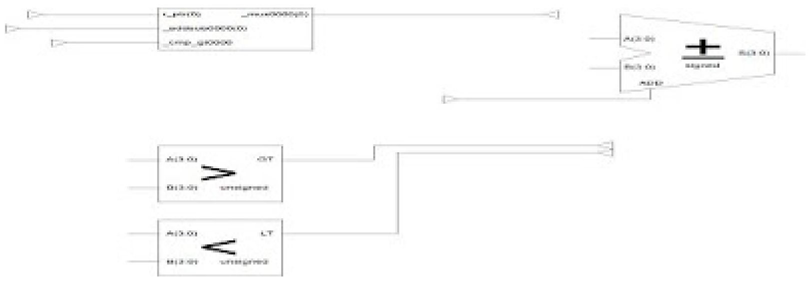

Figure (5) RTL schematic of adder-subtracter and comparators

The above given RTL code synthesizes into adder-subtracter and two comparators. Maximum operating frequency of the design is limited by the carry propagation delay of the adder-sub tractor. Keeping 'speed' as the optimization goal, maximum achievable operating frequency is limited to 112.3MHz. This is one of the main drawbacks of this design compared to the Figure (1) implementation of the FIFO.

Full and Empty Generation logic

This logic takes ptr_diff as input and generates the required condition. If ptr_diff=0, empty condition is generated and if ptr_diff =15 full condition is generated. Similarly almost full, almost empty, half full conditions are generated for required values. Since pointer differences is calculated with respect to both read and write clocks, FIFO status assertion is immediate with zero clock delay.

RTL code given uses continuous assign statement to activate FIFO status flags.

```
//-----------------------------------

assign f_full_flag=(ptr_diff==(f_depth-1));

assign f_empty_flag=(ptr_diff==0);

assign f_half_full_flag=(ptr_diff==f_half_full_value);

assign f_almost_full_flag=(ptr_diff==f_almost_full_value);

assign f_almost_empty_flag=(ptr_diff==f_almost_empty_value);
```

```
//------------------------------------
```

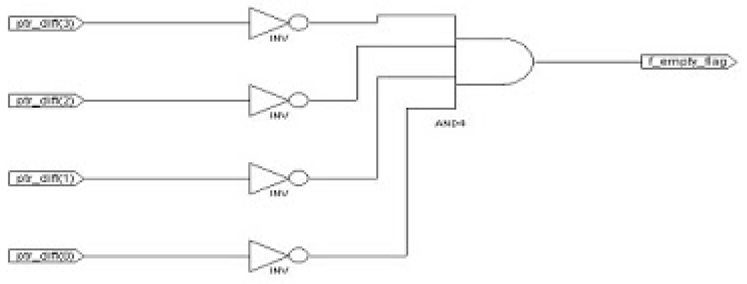

Figure (6) FIFO empty condition generation logic

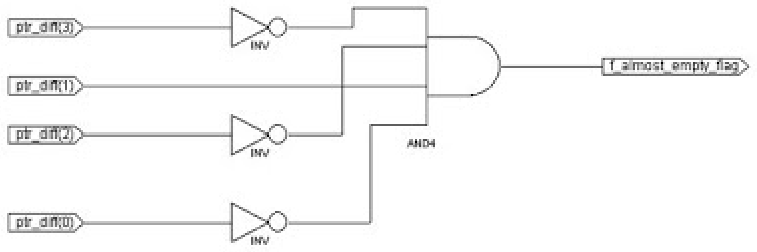

Figure (7) FIFO almost empty condition generation logic

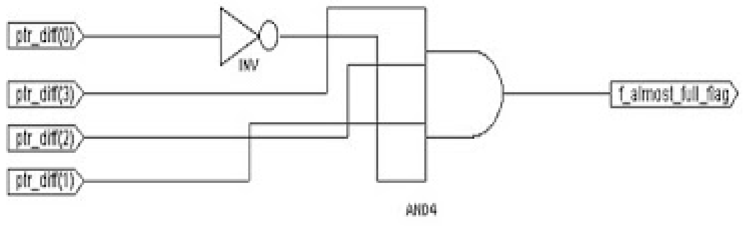

Figure (8) FIFO almost full condition generation logic

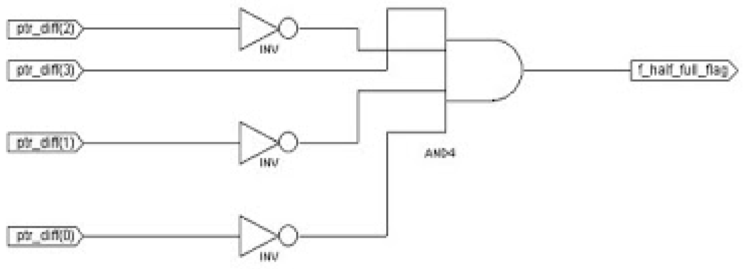

Figure (9) FIFO half full condition generation logic

Figure (6) to Figure (9) shows the FIFO status flag generation logic. Since the ptr_diff is calculated, generation of FIFO status condition becomes very easy with AND and NOT gates.

Next read and write control logic

These control blocks decide the enabling of read and write once the empty and full conditions are asserted. After the assertion of f_empty_flag r_ptr should not increment unless and until there is at least one data is written. Thus once empty flag is asserted next read control logic looks into the write domain for any write activity. It disables r_next_en signal till it finds that data has been written to FIFO. Immediately after the write operation (i.e. posedge of w_clk) control logic enables r_next_en.

```
//------------------------------------
always @(*) //if FIFO empty read counter should not increment
begin
if(r_en && (!f_empty_flag))
r_next_en=1;
else r_next_en=0;
end
//------------------------------------
always @(*) //if FIFO full write counter should not increment
begin
if(w_en && (!f_full_flag))
w_next_en=1;
else w_next_en=0;
end
//------------------------------------
```

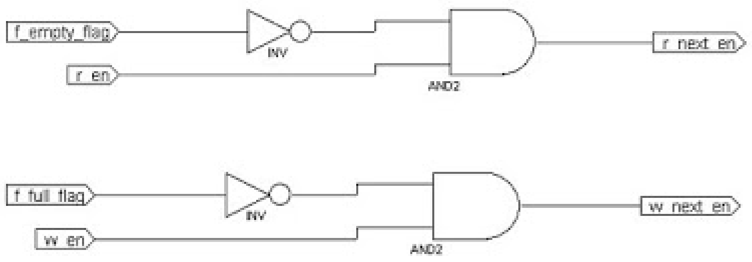

Figure (10) simplified r_next_en and w_next_en generation logic

Similarly after the assertion of f_full_flag, w_ptr should not increment unless and until there is at least one data has been read. Once f_full_flag is asserted write next control logic looks into the read domain for any read activity. It disables w_next_en signal till it finds that data has been read out from FIFO. Immediately after the read operation (i.e. posedge of r_clk) control logic enables w_next_en.

Source : http://asic-soc.blogspot.in/2007/12/new-asynchronous-fifo-design.html