Design, Simulation and Realization of a Parametrizable, Configurable and Modular Asynchronous FIFO

Haytham Ashour
Mentor Graphics, Emulation Division
Cairo, Egypt
haytham_ashour@mentor.com

***Abstract*—Asynchronous FIFOs had become an important block in the new system designs. High speed IOs elasticity buffers, interfacing with system components like processor and memory and in general data transfers between two un-correlated clock domains are example of applications that needs asynchronous FIFOs. This paper presents the design and simulation of an asynchronous FIFO that is paramterizable in data interface width and memory depth. The FIFO flag thresholds are re-configurable at run time and are using a modular design approach in its implementation and in interfacing with other system components.**

Keywords—Asynchronous; FIFO; configurable; modular paramterizable; FPGA

I. INTRODUCTION

An asynchronous FIFO is a FIFO design where data values are written sequentially into a FIFO memory using one clock domain, and the data values are sequentially read from the same FIFO memory using another clock domain. Asynchronous FIFOs are used to safely pass data from one clock domain to another clock domain. [1]. For example, asynchronous FIFO are used in rate matching video interface, speed bridges, bulk data transfer by DMA across a chip, interfacing with processor and bus system and communicating to off-chip components. FIFO full and empty flags are generated by the FIFO design to indicate the internal status of the FIFO buffer. In asynchronous FIFO design, this would require to run a comparison between the read pointer and the write pointer which are clocked at different clock domains.

One common technique for designing an asynchronous FIFO is to use gray code pointers that are got synchronized to the opposite clock domain before generating synchronous FIFO full or empty status signals [2] [3] [4].

This paper would use the same design techniques for asynchronous FIFO design that used in [1]. This paper would implement the asynchronous FIFO to be parametrizable in memory depth and data size. This would need to have both the read and write pointers to be parametrizable as well. Also the generation of the FIFO full and empty flags should accommodate this. The asynchronous FIFO presented in this paper would allow re-configuration of FIFO threshold values at run time. These values are the threshold values at which FIFO near-full and near-empty status signals are generated. These status signals are important in designs that are interested to avoid overflow and underflow conditions on the FIFO .Generation of signals prior in time is important for accommodation of pipe-line latency in digital design till the read / write control block is being able to stop the read / write operation on the FIFO [4].

The rest of this paper is organized as follows: Section II describes the interface signals and system architecture of asynchronous FIFO presented in this work. Section III describes the implementation techniques and presents most important parts of the HDL code used to describe the functionality of the different modules. Section IV presents the simulations results at different operating modes of the asynchronous FIFO presented in this paper. Section V presents realization of this asynchronous FIFO design on FPGA.

II. ASYNCRNOUS FIFO INTERFACE AND ARCHITECTURE

A. Asyncrnous FIFO Interface

Table I lists the interface of the asynchronous FIFO described in this paper. The data signals “wdata” and rdata”, threshold signals “near_full_mrgn” and “near_empty_mrgn” values are all paramterized. Clock and reset of the write port “wclk” and “wrst_n” and that of the read port “rclk” and “rrst_n” are assumed to be asynchronous to each other. Write port input control signals “wen” and “fifowr_clr” are assumed to be synchronous to write clock “wclk”. Read port input control signals “ren” and “fiford_clr” are assumed to be synchronous to read clock “rclk”. Output status signals of write port “near_full”, “full” and “overflow” and output status signals of read port “near_empty”, “empty” and “underflow” are assumed to be generated synchronous to the “wclk” and “rclk” respectively.

Table II lists the HDL parameters that are used to determine the address and data sizes of the FIFO. The address size is used also to determine the depth of the FIFO.

These HDL parameters can be assigned when the FIFO is instantiated inside the system.

B. Asynchronous FIFO Architecture

Fig. 1 shows the architecture of the asynchronous FIFO described in this work. Details of the architecture and functionality of each block is described in Section III

TABLE I. INTERFACE SIGNALS

Signal	Direction	Width	Description
wclk	Input	1 bit	Write clock
wrst_n	Input	1 bit	Write domain reset
wen	Input	1 bit	Write enable
fifowr_clr	Input	1 bit	Write pointer clear
wdata	Input	parameter	Write data
rclk	Input	1 bit	Read clock
rrst_n	Input	1 bit	Read domain reset
ren,	Input	1 bit	Read enable
fiford_clr	Input	1 bit	Read pointer clear
near_full_mrgn	Input	parameter	FIFO near full margin from the full limit
near_empty_mrgn	Input	parameter	FIFO near empty margin from the empty condition
rdata	Output	parameter	Read data
full	Output	1 bit	FIFO full
empty	Output	1 bit	FIFO empty
near_full	Output	1 bit	FIFO is about to be full
near_empty	Output	1 bit	FIFO is about to be empty
over_flow	Output	1 bit	FIFO overflow
under_flow	Output	1 bit	FIFO underflow

TABLE II. PARAMETERS

Parameter	Description	Notes
DATA_SIZE	Determines the width of the wdata /rdata ports	
ADDR_SIZE	Determines the width of the write pointer, read pointer, near full threshold value and near empty threshold value	FIFO memory depth is calculated to be two to the power of value of this parameter

III. ASYNCRNOUS FIFO IMPLEMENTATION

This section describes the implementation of the different blocks inside the asynchronous FIFO design presented in this paper.

A. *async_fifo*

This is the top-level wrapper that integrates the asynchronous FIFO building blocks. This module is the one that shall be instantiated inside the system that uses the FIFO.

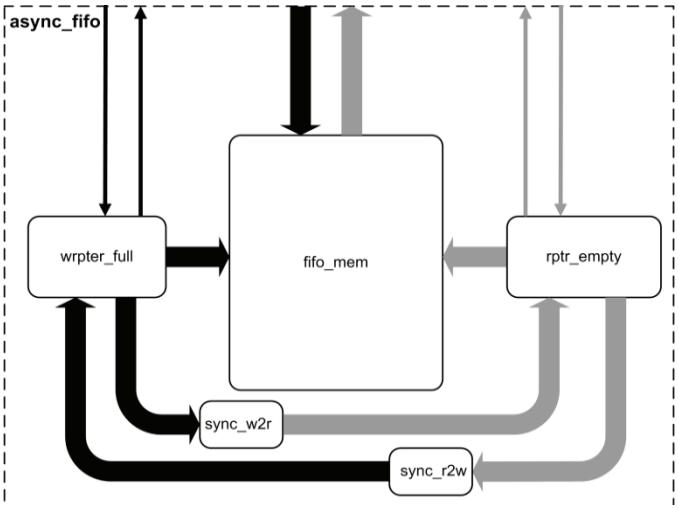

Fig. 1. Asynchrnous FIFO Architecture

B. *sync_r2w / sync_w2r*

These are two instances of a synchronizer module build from two D-FFs that are clocked by the destination clock. As shown in Fig.1, the *sync_r2w* is clocked with the write clock and is used to synchronize the read pointer to the write clock domain while sync_*w2r* is clocked with the read clock and is used to synchronize the write pointer to the read clock domain. Since both pointers are paramterizable, the synchronizer module implementation should be paramterizable as well. The following HDL code shows this

```
parameter  SIZE = 20;
input  [SIZE-1:0]   async_in;
output [SIZE-1:0]  sync_out;
reg    [SIZE-1:0]  sync_1, sync_out;
    always @(posedge clk or negedge rst_n)
    begin
    if (!rst_n) begin
       sync_1    <= {SIZE+1{1'b0}};
      sync_out <= {SIZE+1{1'b0}}; end
else  begin
   sync_1      <= async_in;
   sync_out <= tmp_1;
  end
end
```

At instantiation of these synchronizers inside the "async_fifo" top-level, the parameter "SIZE" will be assigned to "ADDR_SIZE" parameter plus one as follow

```
sync_module  #(.SIZE   (ADDR_SIZE+1))
                  sync_r2w ( <io_list> );
```

C. *fifo_mem*

This is the FIFO storage memory. This memory would be typically a dual-port, asynchronous read and synchronous write memory. However, other memory options can also work. Data width of the memory is paramterizable by "DATA_SIZE" parameter. Memory size calculation is based on "ADDR_SIZE" parameter. The following HDL code shows memory size calculation and declarations of memory signal

```
localparam   DEPTH = 1 << ADDR_SIZE;
 reg    [DATA_SIZE-1:0]  mem [0:DEPTH-1];
```

D. *rptr_empty*

This is the module that handles the read pointer updates and the generation of the "empty", "near_empty" and "underflow" flags. This module is clocked with the read clock. Read pointer is one-bit larger in size than the "fifo_mem" read address size. This is according to the technique in [1] to detect the full and empty conditions. A binary counter is implemented in this module to set the value of the "fifo_mem" read address. Another gray-code counter is implemented also and passed to the "sync_r2w" module as shown in Fig.1. The HDL implementation of these parts is as follow

```
    always @(posedge rclk or negedge rrst_n)
    begin
  if (!rrst_n) begin
```

```
    rbin <= {ADDR_SIZE+1{1'b0}};
    rptr <= {ADDR_SIZE+1{1'b0}};
   end
   else if (fiford_clr) begin
    rbin <= {ADDR_SIZE+1{1'b0}};
    rptr <= {ADDR_SIZE+1{1'b0}};
   end
   else if (ren && ~empty) begin
    rbin <= rbin_next;
    rptr <= rgray_next;
   end
 end
assign raddr      = rbin[ADDR_SIZE-1:0];
assign rbin_next  = rbin + 1'b1;
assign rgray_next = (rbin_next>>1) ^ rbin_next;
```

The gray-coded write pointer value is input to this module after being synchronized to the read clock by the "sync_w2r" as shown in Fig1. This value should be converted back to binary to be compared against binary value of read pointer to detect "empty" and "near_empty" conditions. The HDL code is as follow

```
genvar i;
generate for (i=0;i<ADDR_SIZE;i=i+1) begin: rd_gray2bin
   assign rbin_rq2_wptr[ADDR_SIZE-1-i] =
                          rbin_rq2_wptr [ADDR_SIZE-i] ^
                          rq2_wptr [ADDR_SIZE-1-i];
end
endgenerate
assign  rbin_rq2_wptr [ADDR_SIZE] =
                            rq2_wptr [ADDR_SIZE];
always @ (*)
begin
  near_empty_val <= ~empty  &&
  ((rbin_rq2_wptr - rbin) <= (near_empty_mrgn+1'b1));
end
always @(posedge rclk or negedge rrst_n)
begin
  if (!rrst_n)
    near_empty <= 1'b0;
  else if(fiford_clr)
    near_empty <= 1'b0;
  else if (ren && near_empty_val)
    near_empty <= 1'b1;
  else if (!near_empty_val)
    near_empty <= 1'b0;
 end
always @(posedge rclk or negedge rrst_n)
begin
  if (!rrst_n)
    empty <= 1'b0;
  else if(fiford_clr)
    empty <= 1'b0;
  else if (rbin == rbin_rq2_wptr)
    empty <= 1'b1;
end
always @(posedge rclk or negedge rrst_n)
begin
  if (!rrst_n)
    under_flow <= 1'b0;
  else if(fiford_clr)
    under_flow <= 1'b0;
  else if (empty && ren)
    under_flow <= 1'b1;
end
```

The AND logic with "ren" in generation of "near_empty" condition is to avoid generation of a misleading flag. This condition would happen if the FIFO is empty and started to be filled. Typically at this time, there are no read requests coming. Though this, "near_empty" will be generated if AND logic with "ren" is not there

E. wptr_full

This is the module that handles the write pointer updates and the generation of the "full", "near_full" and "overflow" flags. This module is clocked with the write clock. Write pointer is one-bit larger in than the "fifo_mem" write address size. The same implementation technique described in detail before in the "rptr_empty" module is followed to generate "fifo_mem" writes address, "near_full" and "overflow" flags. As per [1], three conditions are necessary for the FIFO to be full. The HDL code of the "near_ful"l and full logic is below

```
always @ (*)
 begin
    near_full_val <=~full  &&
    ((wbin - wbin_wq2_rptr) >=
              ({ADDR_SIZE{1'b1}} - (near_full_mrgn+1'b1))) ;
end
always @(posedge wclk or negedge wrst_n)
 begin
  if (!wrst_n)
    near_full <= 1'b0;
  else if (fifowr_clr)
    near_full <= 1'b0;
  else if (wen && near_full_val)
    near_ full <= 1'b1;
  else if (!near_ full _val)
    near_ full <= 1'b0;
 end
assign full_val =
((wptr[ADDR_SIZE]     != wq2_rptr[ADDR_SIZE] )    &&
(wptr[ADDR_SIZE-1]   != wq2_rptr[ADDR_SIZE-1]) &&
(wptr[ADDR_SIZE-2:0] = = wq2_rptr[ADDR_SIZE-2:0]));
always @(posedge wclk or negedge wrst_n)
 begin
  if (!wrst_n)
    full <= 1'b0;
  else if (fifowr_clr)
    full <= 1'b0;
  else
    full <= full_val;
 end
```

IV. SIMULATION RESULT

This section presents simulation results for the asynchronous FIFO. All simulation snapshots that are presented here shows the setting of the "near_full_mrgn" and "near_empty_mrgn" input signals, the most important write/read control signals, the write / read related operational flags and the binary version of the read and write pointers. The

clock generator used in the test environment is using random techniques to set the phase and frequency of the generated clock. The read and write clocks generated in the test environment are un-correlated in both phase and frequency.

Fig.2 shows simulation of simple operation on the FIFO. It is being filled first by the write domain then it is emptied later by the read domain. In simulations, the FIFO address size parameter is set to 4 which imply a FIFO memory depth of 16. As "near_full_mrgn" is set to 4, the FIFO "near_full flag" should be asserted when the difference between the write and read pointers becomes equal or greater than 11. As "near_empty_mrgn" is set to 4, the FIFO "near_empty" flag should be asserted when the difference between the write and read pointers becomes equal or less than 4.

Fig.3 shows simulation of typical operation on the FIFO. It is being filled and emptied simultaneously. A successful generation of "near_full", "full", "near_empty" and "empty" flags is presented for the assigned value of "near_full_mrgn" and "near_empty_mrgn".

Fig.4 shows simulation of the re-configurability of the FIFO at run time. The "near_full_mrgn" and "near_empty_mrgn" changed to be 6 at run-time. With this update, the "near_full" is generated when the difference between write and read pointers is equal or greater than 9. The "near_empty" is generated when the difference between write and read pointers is equal or less than 6. The change of "near_full_mrgn" and "near_empty_mrgn" input signals should be synchronized with the write and read clock domains respectively.

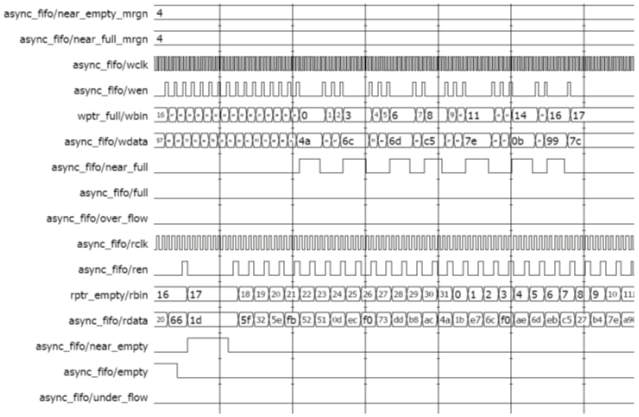

Fig. 2. Simulation of Simple FIFO Write /Read operation

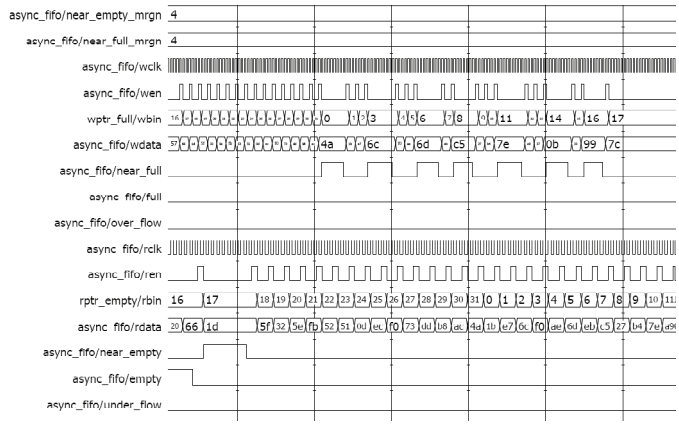

Fig. 3. Simulation of Typical FIFO Write /Read operation

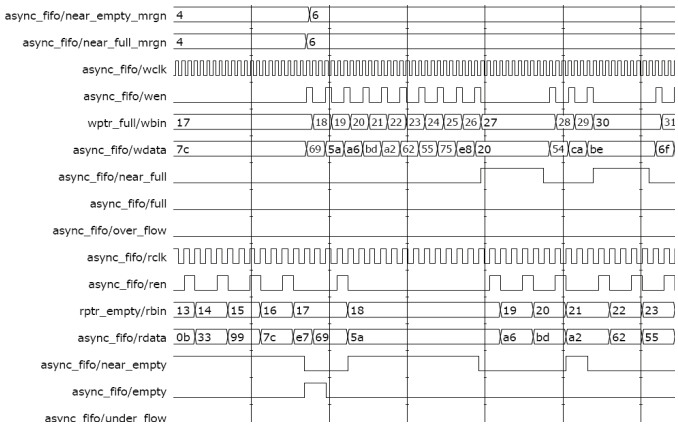

Fig. 4. Simulation of Run-time Reconfigurability of the Asynchronous FIFO

V. FPGA REALIZATION

This section presents FPGA realization results of the asynchronous FIFO design presented in this paper. An Altera Stratix IV GX FPGA chip is used in this realization. The on-chip scope (SignalTap II logic analyzer) is used to capture the asynchronous FIFO signals. Two different PLLs are used to generate the read / write clocks. These PLLS are fed from different clock sources on the FPGA bard. Two asynchronous reset synchronizers [5] are used to synchronize the external reset signal to the read / write clock domains. A simple data generator is implemented to generate an incremental data pattern.

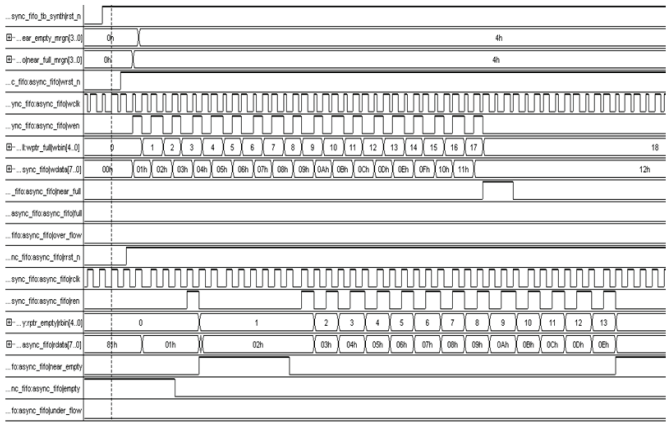

Fig. 5. FPGA Realization of Typical FIFO Write /Read operation

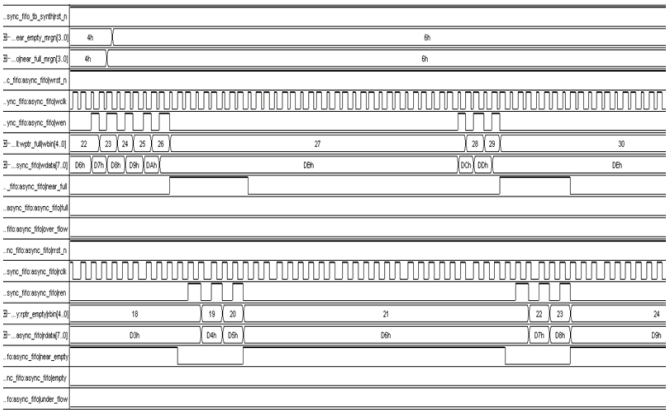

Fig. 6. FPGA Realization of Run-time Reconfigurability of the Asynchronous FIFO

Control logic is built to generate write enable strobes to the FIFO synchronized with the FIFO write clock. These write strobes are continually generated as long as both "full" and "near_full" flags are not asserted. Control logic is used to generate read enable strobes signals to the FIFO synchronized with the FIFO read clock. These read strobes are continually generated as long as both "empty" and "near_empty" flags are not asserted.

Fig.5 shows typical operation of the FIFO on the FPGA. It is being filled and emptied simultaneously. A successful generation of "near_full", "full", "near_empty" and "empty" flags is presented for the assigned value of "near_full_mrgn" and "near_empty_mrgn" as described earlier.

Fig.6 shows re-configurability of the FIFO at run time on FPGA. A successful generation of "near_full", "full", "near_empty" and "empty" flags is presented for the assigned value of "near_full_mrgn" and "near_empty_mrgn" as described earlier.

VI. CONCLUSIONS

A parametrizable and re-configurable asynchronous FIFO design using a modular design approach is presented in this paper. The asynchronous FIFO presented is parametrized in address and data sizes. Early full and empty flag threshold values are re-configured at run-time. The design approach followed is modular in its architecture which means each module in the architecture is implementing an isolated and self-contained set of functions that can be re-used in any other system.

This approach resulted in that the asynchronous FIFO itself becomes a modular design offering a modular interface to the other system level components. The re-configurability of the FIFO at run time finds its applications in applications like rate matching, re-configurable hardware and data streaming applications.

REFERENCES

[1] Clifford E. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design" SNUG-2002, San Jose, CA

[2] Clifford E. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons" SNUG-2002, San Jose, CA.

[3] Frank Gray, "Pulse Code Communication." United States Patent Number 2,632,058. March 17, 1953. K. Elissa,

[4] Ryan William Apperson "A dual-clock FIFO for the reliable transfer of high- throughput data between unrelated clock domains" B.S.E.E. University of Washington, March 2002.

[5] Clifford E. Cummings, Don Mills, Steve Golson "Asynchronous & Synchronous Reset Design Techniques – Part Deux". http://www.sunburst-desig