
Name: Kerolos Youssef Ghobrial

# LAB 1: CDC

## Q9.

In this lab there are some errors such as unsynchronized crossing, data loss, no gray encoding that are solved all by putting an asynchronous FIFO with gray encoded memory as will be explained.

First: Verilog codes of design and FIFO and top module:

**Design module: write clock domain.**

```
module cdc_Q9(input clk1,rst1,input [7:0] din1, output reg [7:0]do1);

always @(posedge clk1 or negedge rst1)
begin

if(~rst1)
begin
do1 <= 8'b0;
end

else
begin
do1 <= din1;
end
end

endmodule
```

**FIFO module:**

```
module sync_r2w #(parameter ADDRSIZE = 4)
 (output reg [ADDRSIZE:0] wq2_rptr,
 input [ADDRSIZE:0] rptr,
```

```
 input wclk, wrst_n);

 reg [ADDRSIZE:0] wq1_rptr;

 always @(posedge wclk or negedge wrst_n)
 if (!wrst_n) {wq2_rptr,wq1_rptr} <= 0;
 else {wq2_rptr,wq1_rptr} <= {wq1_rptr,rptr};

endmodule


module sync_w2r #(parameter ADDRSIZE = 4)
 (output reg [ADDRSIZE:0] rq2_wptr,
 input [ADDRSIZE:0] wptr,
 input rclk, rrst_n);

 reg [ADDRSIZE:0] rq1_wptr;

 always @(posedge rclk or negedge rrst_n)
 if (!rrst_n) {rq2_wptr,rq1_wptr} <= 0;
 else {rq2_wptr,rq1_wptr} <= {rq1_wptr,wptr};

endmodule


module fifomem #(parameter DATASIZE = 8, // Memory data word width
 parameter ADDRSIZE = 4) // Number of mem address bits
 (output [DATASIZE-1:0] rdata,
 input [DATASIZE-1:0] wdata,
 input [ADDRSIZE-1:0] waddr, raddr,
 input wclken, wfull, wclk);

 `ifdef VENDORRAM
 // instantiation of a vendor's dual-port RAM
 vendor_ram mem (.dout(rdata), .din(wdata),
 .waddr(waddr), .raddr(raddr),
 .wclken(wclken),
 .wclken_n(wfull), .clk(wclk));
 `else

 // RTL Verilog memory model
 localparam DEPTH = 1<<ADDRSIZE;
 reg [DATASIZE-1:0] mem [0:DEPTH-1];
 assign rdata = mem[raddr];

 always @(posedge wclk)
 if (wclken && !wfull) mem[waddr] <= wdata;
 `endif

endmodule


module rptr_empty #(parameter ADDRSIZE = 4)
 (output reg rempty,
 output [ADDRSIZE-1:0] raddr,
 output reg [ADDRSIZE :0] rptr,
```

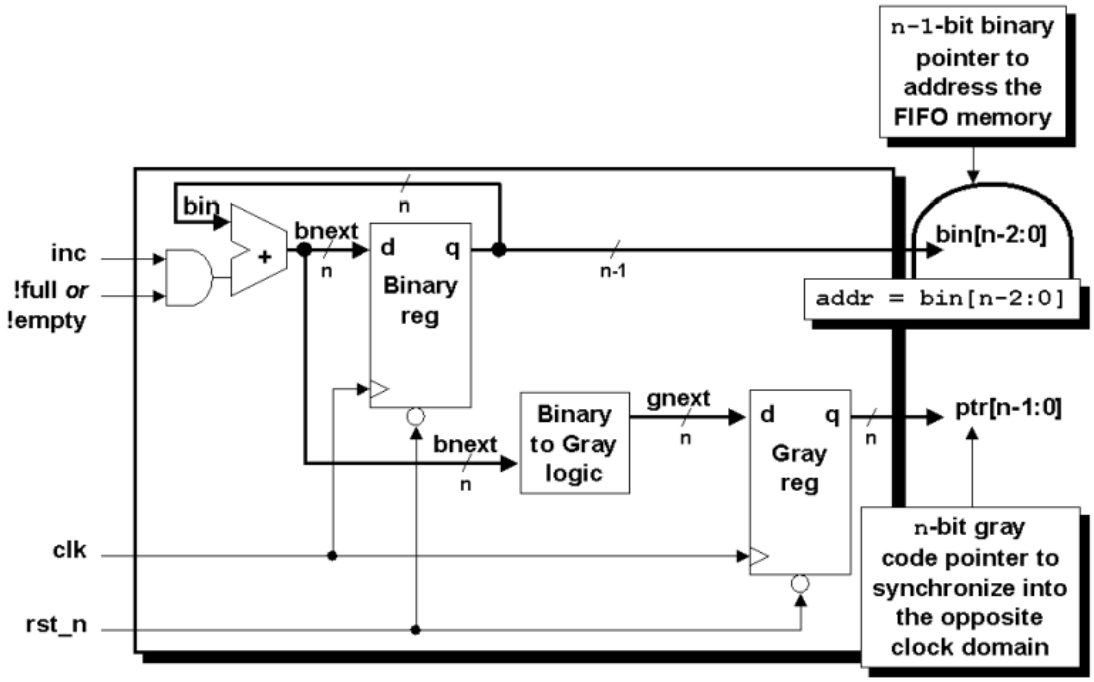
```
 input [ADDRSIZE :0] rq2_wptr,
 input rinc, rclk, rrst_n);
 reg [ADDRSIZE:0] rbin;
 wire [ADDRSIZE:0] rgraynext, rbinnext;

 //------------------
 // GRAYSTYLE2 pointer
 //------------------
 always @(posedge rclk or negedge rrst_n)
 if (!rrst_n) {rbin, rptr} <= 0;
 else {rbin, rptr} <= {rbinnext, rgraynext};
 // Memory read-address pointer (okay to use binary to address memory)
 assign raddr = rbin[ADDRSIZE-1:0];
 assign rbinnext = rbin + (rinc & ~rempty);
 assign rgraynext = (rbinnext>>1) ^ rbinnext;

 //---------------------------------------------------------------
 // FIFO empty when the next rptr == synchronized wptr or on reset
 //---------------------------------------------------------------
 assign rempty_val = (rgraynext == rq2_wptr);
 always @(posedge rclk or negedge rrst_n)
 if (!rrst_n) rempty <= 1'b1;
 else rempty <= rempty_val;

endmodule


module wptr_full #(parameter ADDRSIZE = 4)
 (output reg wfull,
 output [ADDRSIZE-1:0] waddr,
 output reg [ADDRSIZE :0] wptr,
 input [ADDRSIZE :0] wq2_rptr,
 input winc, wclk, wrst_n);

 reg [ADDRSIZE:0] wbin;
 wire [ADDRSIZE:0] wgraynext, wbinnext;

 // GRAYSTYLE2 pointer
 always @(posedge wclk or negedge wrst_n)
 if (!wrst_n) {wbin, wptr} <= 0;
 else {wbin, wptr} <= {wbinnext, wgraynext};
 // Memory write-address pointer (okay to use binary to address memory)
 assign waddr = wbin[ADDRSIZE-1:0];
 assign wbinnext = wbin + (winc & ~wfull);
 assign wgraynext = (wbinnext>>1) ^ wbinnext;

 //------------------------------------------------------------------
 // Simplified version of the three necessary full-tests:
 // assign wfull_val=((wgnext[ADDRSIZE] !=wq2_rptr[ADDRSIZE] ) &&
 // (wgnext[ADDRSIZE-1] !=wq2_rptr[ADDRSIZE-1]) &&
 // (wgnext[ADDRSIZE-2:0]==wq2_rptr[ADDRSIZE-2:0]));
 //------------------------------------------------------------------

 assign wfull_val = (wgraynext=={~wq2_rptr[ADDRSIZE:ADDRSIZE-1],
 wq2_rptr[ADDRSIZE-2:0]});
 always @(posedge wclk or negedge wrst_n)
```

```
 if (!wrst_n) wfull <= 1'b0;
 else wfull <= wfull_val;
endmodule


module fifo #(parameter DSIZE = 8,
 parameter ASIZE = 4)
 (output [DSIZE-1:0] rdata,
 output wfull,
 output rempty,
 input [DSIZE-1:0] wdata,
 input winc, wclk, wrst_n,
 input rinc, rclk, rrst_n);

 wire [ASIZE-1:0] waddr, raddr;
 wire [ASIZE:0] wptr, rptr, wq2_rptr, rq2_wptr;

 sync_r2w sync_r2w (.wq2_rptr(wq2_rptr), .rptr(rptr),
 .wclk(wclk), .wrst_n(wrst_n));

 sync_w2r sync_w2r (.rq2_wptr(rq2_wptr), .wptr(wptr),
 .rclk(rclk), .rrst_n(rrst_n));

 fifomem #(DSIZE, ASIZE) fifomem
 (.rdata(rdata), .wdata(wdata),
 .waddr(waddr), .raddr(raddr),
 .wclken(winc), .wfull(wfull),
 .wclk(wclk));

 rptr_empty #(ASIZE) rptr_empty
 (.rempty(rempty),
 .raddr(raddr),
 .rptr(rptr), .rq2_wptr(rq2_wptr),
 .rinc(rinc), .rclk(rclk),
 .rrst_n(rrst_n));

 wptr_full #(ASIZE) wptr_full
 (.wfull(wfull), .waddr(waddr),
 .wptr(wptr), .wq2_rptr(wq2_rptr),
 .winc(winc), .wclk(wclk),
 .wrst_n(wrst_n));

endmodule
```

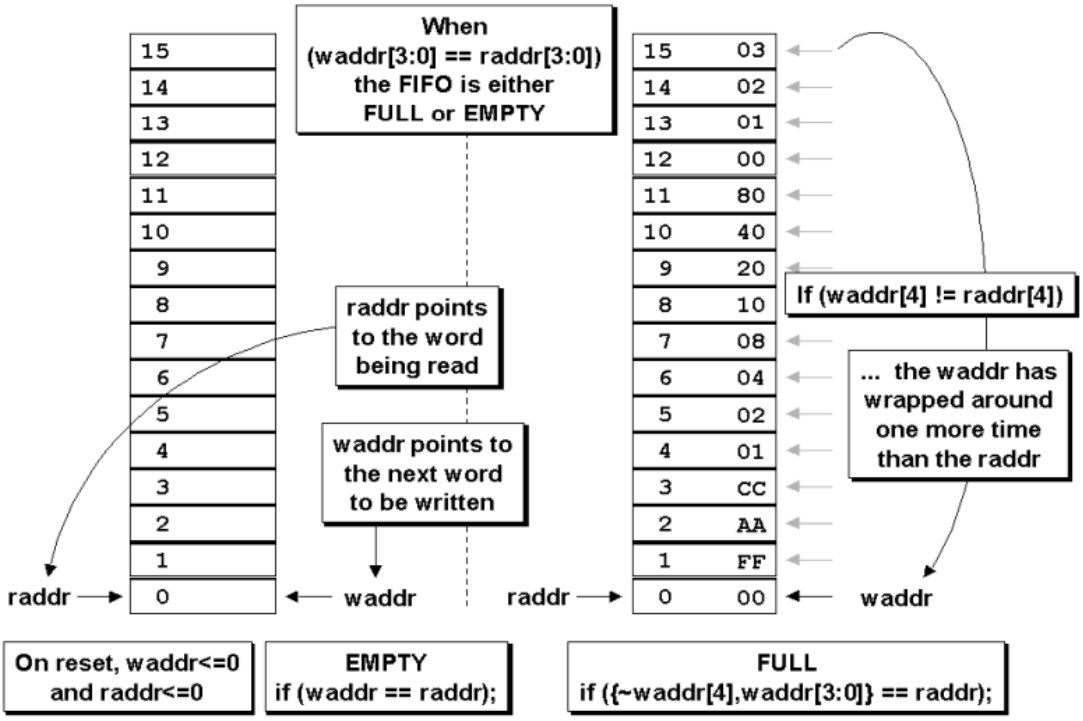
## Top module:

```
module top #(parameter DSIZE = 8,
 parameter ASIZE = 4)
(input wclk,rclk,wrst,rrst,rinc,winc,output [7:0]rdata,output
wfull,rempty, input [7:0] wdata);

wire [7:0] rdata1;
```

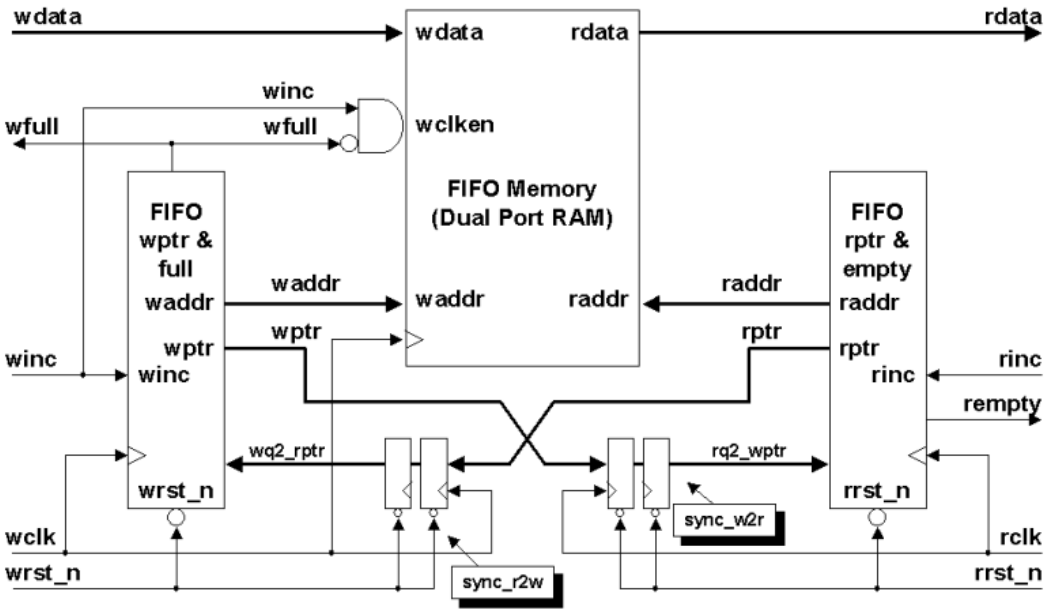
```
fifo #(DSIZE, ASIZE)
f1(rdata,wfull,rempty,rdata1,winc,wclk,wrst,rinc,rclk,rrst);

cdc_Q9 cdc(wclk,wrst,wdata,rdata1);

endmodule
```




The above figure for gray code counter used in memory and that can solve the problem of gray encoding for data array.




The above figure explains the conditions of full and empty memory of FIFO.




The above figure explains FIFO partitioning with synchronized pointer comparison.

## Second: SGDC constraints:

current_design top

clock -name wclk -edge {"0" "10"} -period 20 -domain wclk

clock -name rclk -edge {"0" "5"} -period 10 -domain rclk

reset -name wrst -value 0 -async

reset -name rrst -value 0 -async

input -name wdata -clock wclk

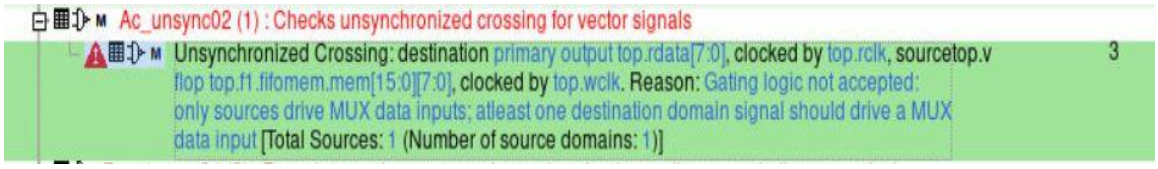
output -name rdata -clock rclk

cdc_false_path -from top.f1.fifomem.mem -to top.rdata -to_type data  -from_type data

## Third: results:

In beginning after solving unsynchronized crossing between pointers of memory using double flip-flop synchronizers as was shown in figure of FIFO partitioning.

There was an unsynchronized crossing for the read data controlled by a multiplexer which its selector signals found in the write data domain as in the next image.




I solved that by the constraint of false path in the end of SGDC file.

Finally, the issues were all solved as in the next image.