Self Project by Praveen Sangani (22M1140), IITB

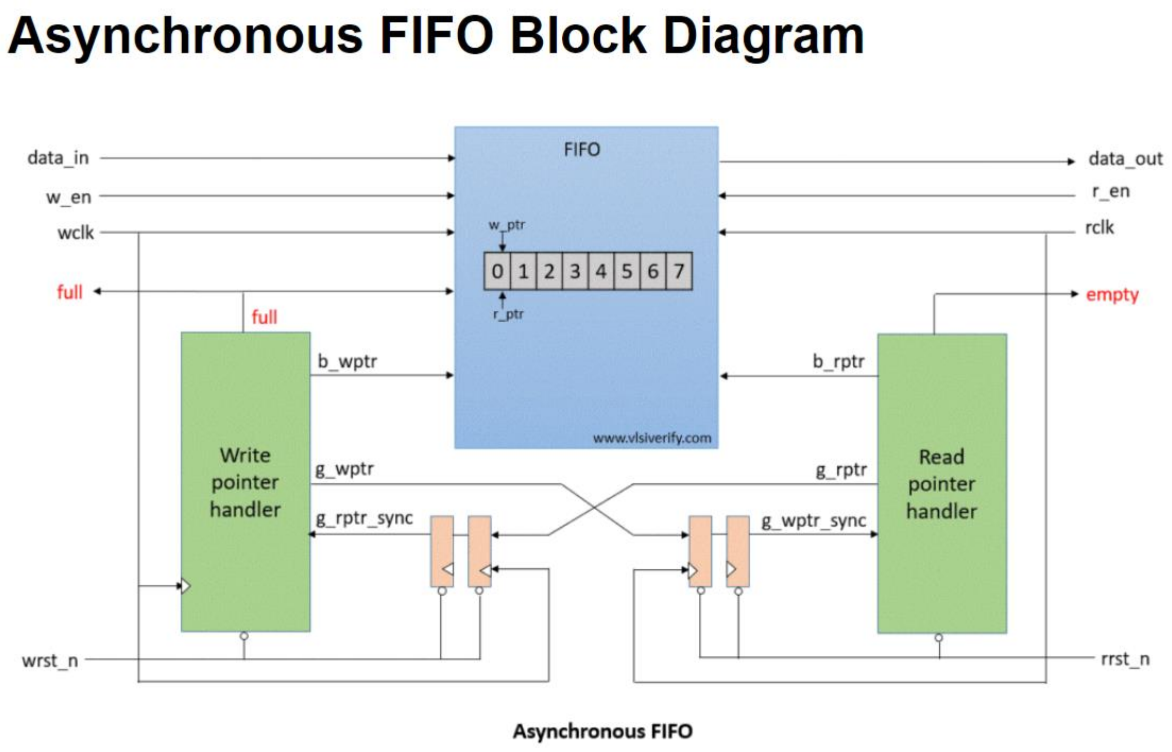
Asynchronous FIFO

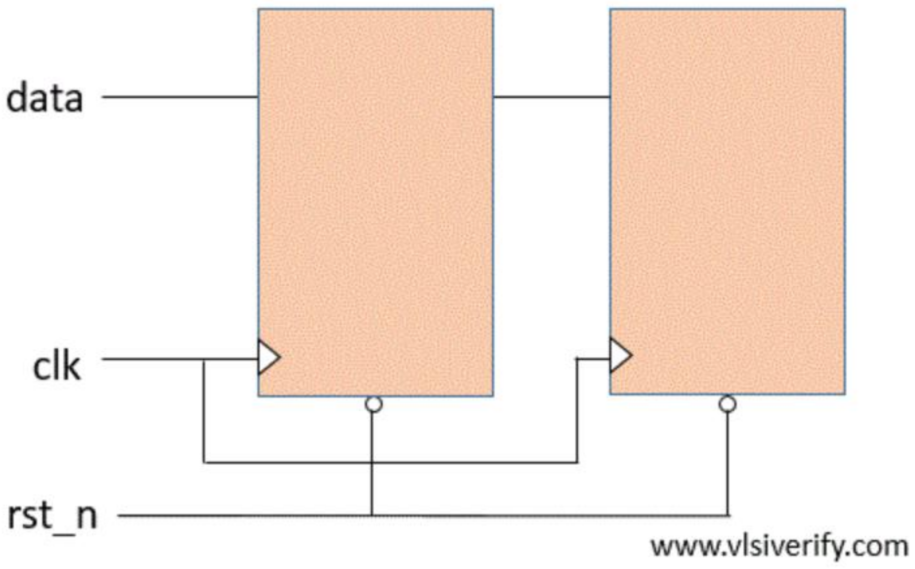
In asynchronous FIFO, data read and write operations use different clock frequencies. Since write and read clocks are not synchronized, it is referred to as asynchronous FIFO. Usually, these are used in systems where data need to pass from one clock domain to another which is generally termed as 'clock domain crossing'.

Efforts to synchronize multiple signals from one clock domain to a new clock domain, ensuring the synchronization of all these signals in the new domain, has proven to be problematic. In designs, First-In-First-Out (FIFO) structures are used in designs to securely transfer multi-bit data chunks from one clock domain to other. These data chunks are inserted into a FIFO buffer memory array through control signals in one clock domain, and the removal of data chunks occurs through another port of the same FIFO buffer memory array using control signals from a different clock domain.

The challenges associated with FIFO design arise primarily from devising the FIFO pointers and identifying a dependable method for ascertaining the "full" and "empty" status of the FIFO.

Block Diagram of Asynchronous FIFO:





2 flip-flop synchronizer

Introduction

In the realm of digital design, data communication and synchronization play pivotal roles. Asynchronous FIFOs, also known as First-In-First-Out buffers, emerge as indispensable components that address the challenge of transferring data seamlessly between different clock domains. This report delves into the architecture, design considerations, applications, challenges, and future trends associated with asynchronous FIFOs.

1. Asynchronous FIFO Architecture

An asynchronous FIFO is essentially a data buffer that manages the orderly transfer of information between different clock domains. Its architecture encompasses several key components:

Read and Write Pointers: These pointers indicate the positions for reading and writing data. They ensure that data is processed in the order it was received.

Data Storage: The FIFO employs memory cells to store data in an organized manner. Depending on the design, this storage could be linear or dual-ported.

Control Logic: The control logic governs the read and write operations, manages the conditions of full and empty states, and synchronizes the data flow across asynchronous clock domains.

2. Design Considerations

Designing an asynchronous FIFO involves navigating through challenges inherent to its asynchronous nature:

Metastability Mitigation: The asynchronicity of clock domains can lead to metastability, where signals can become ambiguous. Techniques like multi-stage synchronization and Gray coding are employed to mitigate this issue.

Synchronization Strategies: Synchronizing signals between different clock domains is crucial. This typically involves using multiple flip-flops and possibly Gray-coded state machines.

Handshaking Mechanisms: Handshake signals such as read and write enables are essential to prevent data corruption. These signals ensure that both sides are ready for data transfer.

3. Asynchronous FIFO Design Steps

Creating an asynchronous FIFO involves a systematic approach:

Requirements Specification: Determine the FIFO's parameters such as data width, depth, and the clock domains involved.

Pointer Logic Implementation: Develop and manage the read and write pointers to ensure proper data flow.

Data Storage Arrangement: Organize memory cells to store data efficiently while considering read and write operations.

Control Logic Development: Design control logic to manage read and write operations, handle full and empty conditions, and synchronize data flow.

Addressing Metastability: Integrate techniques to mitigate metastability challenges across asynchronous domains.

Timing Analysis: Perform timing analysis to ensure data capture and transfer within the required time constraints.

4. Applications of Asynchronous FIFOs

Asynchronous FIFOs find practical utility across a wide spectrum of applications:

Data Communication: They are integral to serial communication interfaces such as UART, SPI, and I2C, where different clock domains need to communicate seamlessly.

Memory Interfaces: Asynchronous FIFOs bridge the gap between processors and memory units operating at different clock frequencies.

Digital Signal Processing: They facilitate data buffering in DSP pipelines, enabling efficient signal processing.

Multimedia Systems: Asynchronous FIFOs ensure smooth data transfer between various components in multimedia applications, like video and audio processing units.

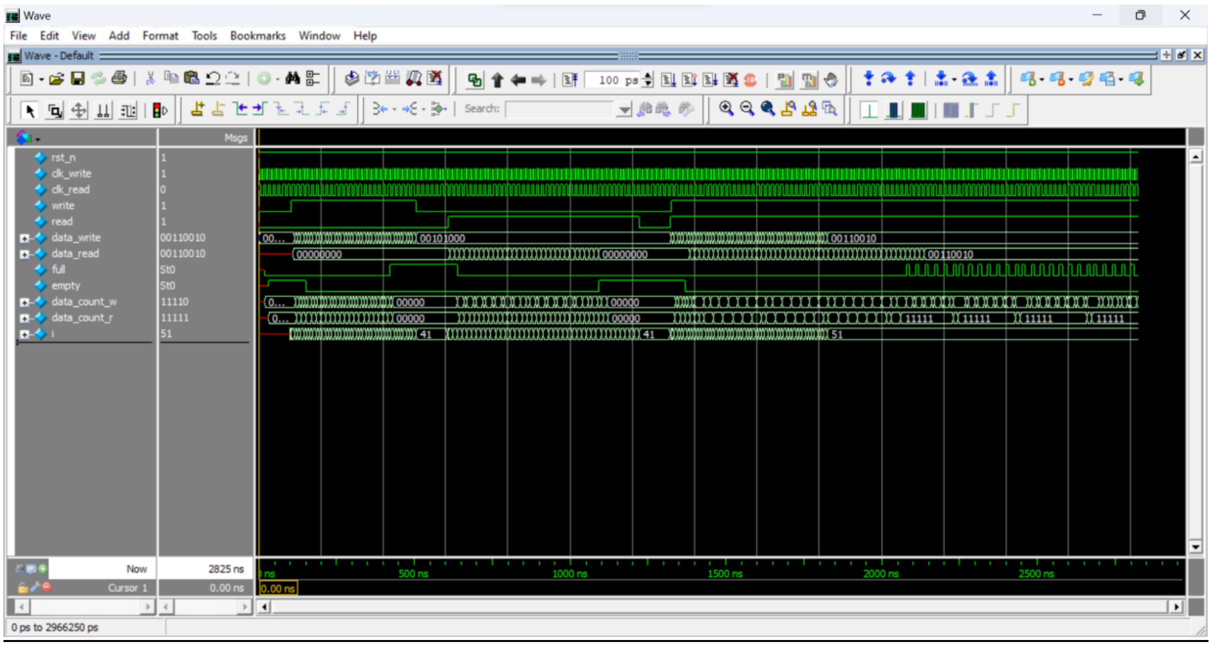
5. Challenges and Future Trends

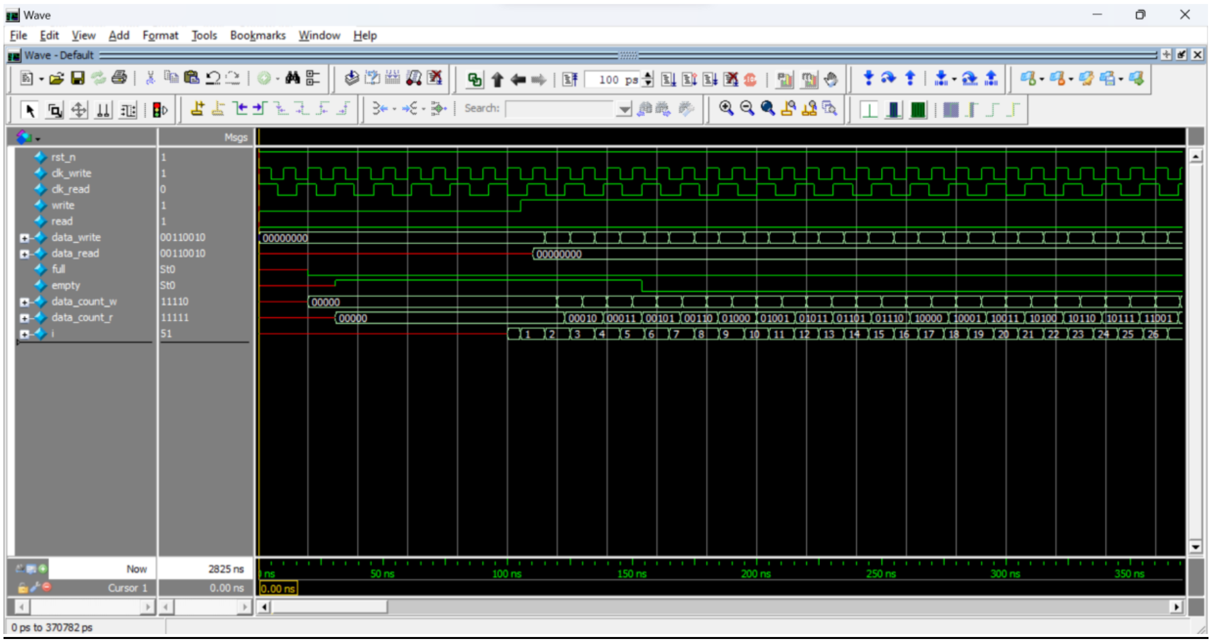
Asynchronous FIFO design is not without its challenges:

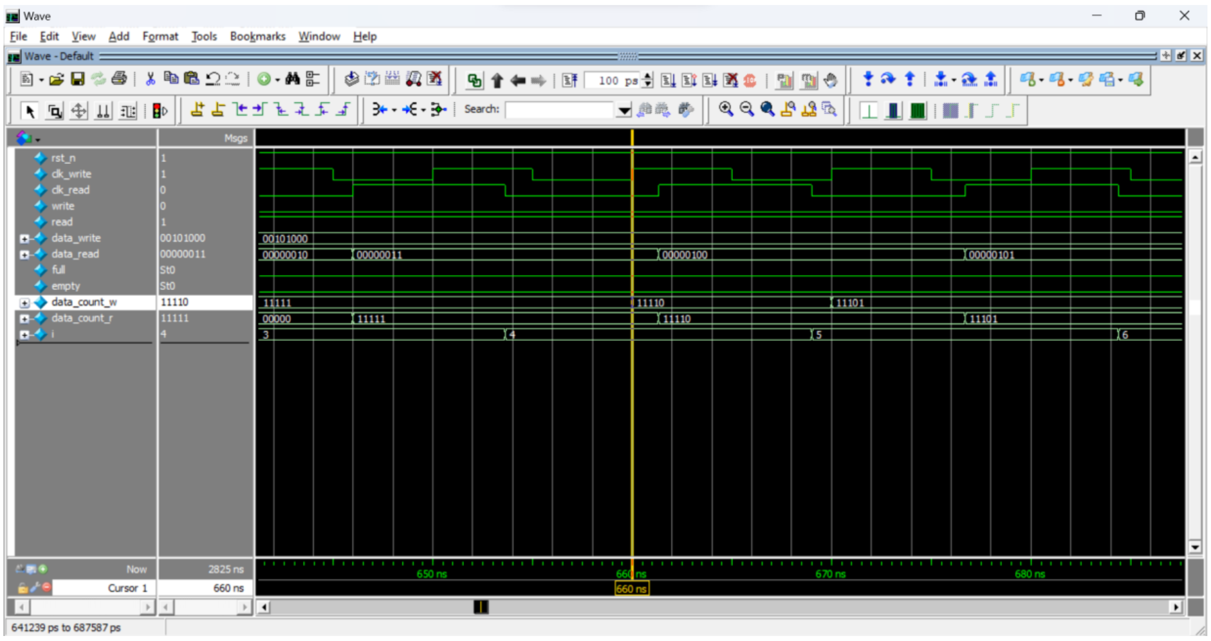
Metastability Management: Handling metastability continues to be a crucial challenge. Research into novel techniques for better metastability handling is ongoing.

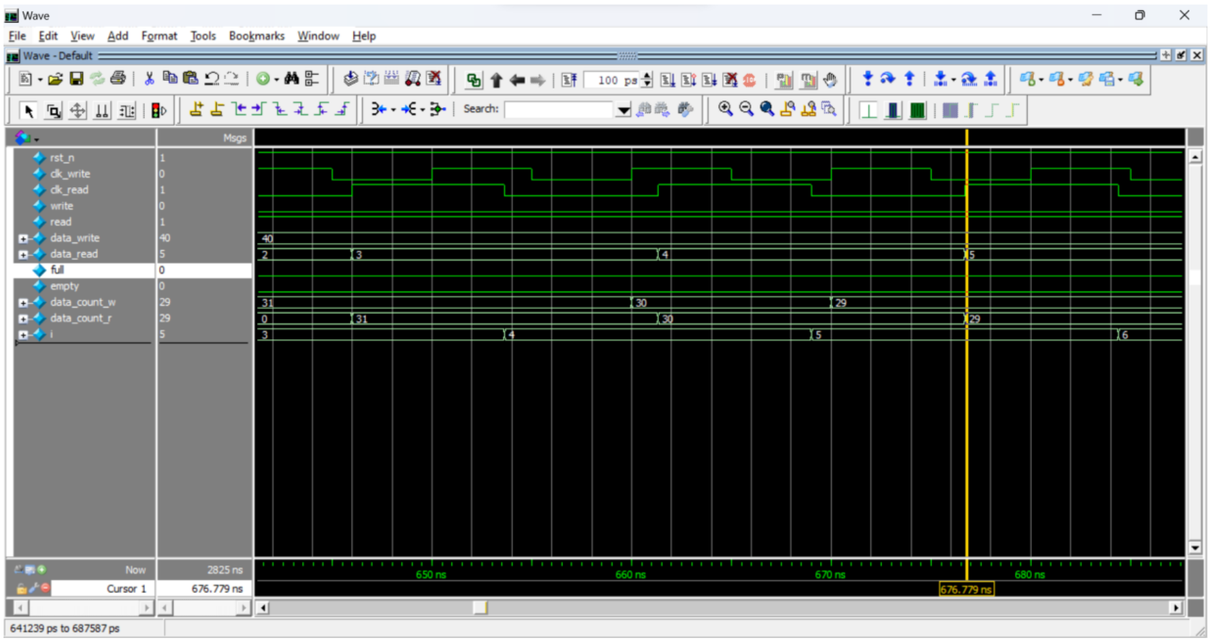
Power Efficiency: Optimizing power consumption in asynchronous designs remains a key concern.

RTL Simulation :









Code :

```
`timescale 1ns / 1ps


module async_fifo
        #(
                parameter DATA_WIDTH=8,
                                        FIFO_DEPTH_WIDTH=11  //total depth will then be
2**FIFO_DEPTH_WIDTH
```

```
            )
            (
            input wire rst_n,
            input wire clk_write,clk_read, //clock input from both domains
            input wire write,read,
            input wire [DATA_WIDTH-1:0] data_write, //input FROM write clock domain
            output [DATA_WIDTH-1:0] data_read, //output TO read clock domain
            output reg full,empty, //full=sync to write domain clk , empty=sync to read domain clk
            output reg[FIFO_DEPTH_WIDTH-1:0] data_count_w,data_count_r //counts number of data
left in fifo memory(sync to either write or read clk)
   );




            localparam FIFO_DEPTH=2**FIFO_DEPTH_WIDTH;

            //////////////////WRITE CLOCK DOMAIN//////////////////////////////
            reg[FIFO_DEPTH_WIDTH:0] w_ptr_q=0; //binary counter for write pointer
            reg[FIFO_DEPTH_WIDTH:0] r_ptr_sync; //binary pointer for read pointer sync to write clk
            reg[FIFO_DEPTH_WIDTH:0] r_grey_sync; //grey counter for the read pointer synchronized
to write clock
            reg[3:0] i; //log_2(FIFO_DEPTH_WIDTH)

            wire[FIFO_DEPTH_WIDTH:0] w_grey,w_grey_nxt; //grey counter for write pointer
            wire we;

            assign w_grey=w_ptr_q^(w_ptr_q>>1); //binary to grey code conversion for current write
pointer
            assign w_grey_nxt=(w_ptr_q+1'b1)^((w_ptr_q+1'b1)>>1);  //next grey code
            assign we= write && !full;

            //register operation
```

```
        always @(posedge clk_write,negedge rst_n) begin

                if(!rst_n) begin

                        w_ptr_q<=0;

                        full<=0;

                end

                else begin

                        if(write && !full) begin //write condition

                                w_ptr_q<=w_ptr_q+1'b1;

                                full <= w_grey_nxt ==
{~r_grey_sync[FIFO_DEPTH_WIDTH:FIFO_DEPTH_WIDTH-1],r_grey_sync[FIFO_DEPTH_WIDTH-2:0]};
//algorithm for full logic which can be observed on the grey code table

                        end

                        else full <= w_grey ==
{~r_grey_sync[FIFO_DEPTH_WIDTH:FIFO_DEPTH_WIDTH-1],r_grey_sync[FIFO_DEPTH_WIDTH-2:0]};


                        for(i=0;i<=FIFO_DEPTH_WIDTH;i=i+1) r_ptr_sync[i]=^(r_grey_sync>>i);
//grey code to binary converter

                        data_count_w <= (w_ptr_q>=r_ptr_sync)? (w_ptr_q-
r_ptr_sync):(FIFO_DEPTH-r_ptr_sync+w_ptr_q); //compares write pointer and sync read pointer to
generate data_count

                end

        end


        //////////////////////////////////////////////////////////////////////




         //////////////////READ CLOCK DOMAIN//////////////////////////////

        reg[FIFO_DEPTH_WIDTH:0] r_ptr_q=0; //binary counter for read pointer

        reg[FIFO_DEPTH_WIDTH:0] w_ptr_sync; //binary counter for write pointer sync to read clk

        reg[FIFO_DEPTH_WIDTH:0] w_grey_sync; //grey counter for the write pointer synchronized
to read clock


        wire[FIFO_DEPTH_WIDTH:0] r_grey,r_grey_nxt; //grey counter for read pointer
```

```
    wire[FIFO_DEPTH_WIDTH:0] r_ptr_d;



    assign r_grey= r_ptr_q^(r_ptr_q>>1);  //binary to grey code conversion
    assign r_grey_nxt= (r_ptr_q+1'b1)^((r_ptr_q+1'b1)>>1); //next grey code
    assign r_ptr_d= (read && !empty)? r_ptr_q+1'b1:r_ptr_q;

    //register operation
    always @(posedge clk_read,negedge rst_n) begin
        if(!rst_n) begin
            r_ptr_q<=0;
            empty<=1;
        end
        else begin
            r_ptr_q<=r_ptr_d;
            if(read && !empty) empty <= r_grey_nxt==w_grey_sync;//empty condition
            else empty <= r_grey==w_grey_sync;

            for(i=0;i<=FIFO_DEPTH_WIDTH;i=i+1) w_ptr_sync[i]=^(w_grey_sync>>i);
//grey code to binary converter
            data_count_r = (w_ptr_q>=r_ptr_sync)? (w_ptr_q-r_ptr_sync):(FIFO_DEPTH-
r_ptr_sync+w_ptr_q); //compares read pointer to sync write pointer to generate data_count
        end
    end
    ////////////////////////////////////////////////////////////////////////



    /////////////////////CLOCK DOMAIN CROSSING//////////////////////////////
    reg[FIFO_DEPTH_WIDTH:0] r_grey_sync_temp;
    reg[FIFO_DEPTH_WIDTH:0] w_grey_sync_temp;
    always @(posedge clk_write) begin //2 D-Flipflops for reduced metastability in clock domain
crossing from READ DOMAIN to WRITE DOMAIN
```

```
                r_grey_sync_temp<=r_grey;

                r_grey_sync<=r_grey_sync_temp;

         end

         always @(posedge clk_read) begin //2 D-Flipflops for reduced metastability in clock domain
crossing from WRITE DOMAIN to READ DOMAIN

                w_grey_sync_temp<=w_grey;

                w_grey_sync<=w_grey_sync_temp;

         end


        //////////////////////////////////////////////////////////////////////////




        //instantiation of dual port block ram

        dual_port_sync #(.ADDR_WIDTH(FIFO_DEPTH_WIDTH) , .DATA_WIDTH(DATA_WIDTH)) m0

        (

                .clk_r(clk_read),

                .clk_w(clk_write),

                .we(we),

                .din(data_write),

                .addr_a(w_ptr_q[FIFO_DEPTH_WIDTH-1:0]), //write address

                .addr_b(r_ptr_d[FIFO_DEPTH_WIDTH-1:0] ), //read address ,addr_b is already
buffered inside this module so we will use the "_d" ptr to advance the data(not "_q")

                .dout(data_read)

        );


endmodule

        //inference template for dual port block ram

module dual_port_sync

        #(

                parameter ADDR_WIDTH=11, //2k by 8 dual port synchronous ram(16k block ram)

                                        DATA_WIDTH=8
```

```
	)
	(
		input wire clk_r,
		input wire clk_w,
		input wire we,
		input wire[DATA_WIDTH-1:0] din,
		input wire[ADDR_WIDTH-1:0] addr_a,addr_b, //addr_a for write, addr_b for read
		output wire[DATA_WIDTH-1:0] dout
	);

	reg[DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
	reg[ADDR_WIDTH-1:0] addr_b_q;

	always @(posedge clk_w) begin
		if(we) ram[addr_a]<=din;
	end
	always @(posedge clk_r) begin
		addr_b_q<=addr_b;
	end
	assign dout=ram[addr_b_q];

endmodule
```