Code of asynch fifo  
  
`timescale 1ns / 1ps

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

//

// Create Date: 19.08.2024 23:34:43

// Design Name: Asynchronous FIFO

// Module Name: asynch\_fifo

// Target Devices: zedboard

// This module implements an Asynchronous FIFO with separate read and write clocks.

// It includes logic to handle data synchronization across different clock domains

// using Gray code pointers to avoid metastability and synchronization errors.

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

module cdc\_asynch\_fifo(

input wr\_clk, // Write clock

input rd\_clk, // Read clock

input rst, // Asynchronous reset

input wr\_en, // Write enable signal

input rd\_en, // Read enable signal

input [7:0] write\_data, // Data input for writing to FIFO

output reg [7:0] read\_data, // Data output for reading from FIFO

output wire empty, // Flag to indicate FIFO is empty

output wire full // Flag to indicate FIFO is full

);

parameter fifo\_depth = 8; // FIFO depth is 8, so address pointer will be 3 bits wide

parameter add\_size = 4; // Address size set to 4 bits to handle full/empty condition

// Define the read and write pointers with 4 bits each

reg [3:0] wptr, rptr;

wire [3:0] wptr\_gray, rptr\_gray;

// FIFO buffer memory of 8 entries, each 8 bits wide

reg [7:0] mem [7:0];

// Synchronized write and read pointers (Gray code)

reg [3:0] wptr\_gray\_sync;

reg [3:0] wptr\_gray\_ff1, wptr\_gray\_ff2;

reg [3:0] rptr\_gray\_sync;

reg [3:0] rptr\_gray\_ff1, rptr\_gray\_ff2;

// Write data into FIFO buffer

// This block is triggered by the write clock and checks for write enable and FIFO full conditions

always @(posedge wr\_clk) begin

if (rst) begin

// Reset write pointer to zero

wptr <= 4'b0000;

end else if (wr\_en && !full) begin

// Write data to the memory at the current write pointer position

mem[wptr] <= write\_data;

// Increment the write pointer

wptr <= wptr + 1;

end

end

// Read data from FIFO buffer

// This block is triggered by the read clock and checks for read enable and FIFO empty conditions

always @(posedge rd\_clk) begin

if (rst) begin

// Reset read pointer to zero

rptr <= 4'b0000;

end else if (rd\_en && !empty) begin

// Read data from the memory at the current read pointer position

read\_data <= mem[rptr];

// Increment the read pointer

rptr <= rptr + 1;

end

end

// Convert binary write and read pointers to Gray code

// Gray code minimizes the risk of synchronization errors by ensuring only one bit changes at a time

assign wptr\_gray = wptr ^ (wptr >> 1);

assign rptr\_gray = rptr ^ (rptr >> 1);

// Synchronize the write pointer to the read clock domain

// This prevents metastability when comparing pointers across different clock domains

always @(posedge rd\_clk) begin

if (rst) begin

// Reset synchronization flip-flops

wptr\_gray\_ff1 <= 0;

wptr\_gray\_ff2 <= 0;

wptr\_gray\_sync <= 0;

end else begin

// First stage of synchronization

wptr\_gray\_ff1 <= wptr\_gray;

// Second stage of synchronization

wptr\_gray\_ff2 <= wptr\_gray\_ff1;

// Final synchronized write pointer in read clock domain

wptr\_gray\_sync <= wptr\_gray\_ff2;

end

end

// Synchronize the read pointer to the write clock domain

// This prevents metastability when comparing pointers across different clock domains

always @(posedge wr\_clk) begin

if (rst) begin

// Reset synchronization flip-flops

rptr\_gray\_ff1 <= 0;

rptr\_gray\_ff2 <= 0;

rptr\_gray\_sync <= 0;

end else begin

// First stage of synchronization

rptr\_gray\_ff1 <= rptr\_gray;

// Second stage of synchronization

rptr\_gray\_ff2 <= rptr\_gray\_ff1;

// Final synchronized read pointer in write clock domain

rptr\_gray\_sync <= rptr\_gray\_ff2;

end

end

// Calculate the empty condition

// The FIFO is empty when the read pointer equals the synchronized write pointer

assign empty = (rptr\_gray == wptr\_gray\_sync);

// Calculate the full condition

// The FIFO is full when the write pointer is one position behind the read pointer, after wrapping around

assign full = ((wptr\_gray[3] != rptr\_gray\_sync[3]) && // MSB check to detect wrap-around

(wptr\_gray[2] == rptr\_gray\_sync[2]) && // Ensure pointers are aligned except MSB

(wptr\_gray[1:0] == rptr\_gray\_sync[1:0])); // Lower bits must match

endmodule

test bench

`timescale 1ns / 1ps

module cdc\_asynch\_fifo\_tb;

// Testbench signals

reg wr\_clk, rd\_clk, rst, wr\_en, rd\_en;

reg [7:0] write\_data;

wire [7:0] read\_data;

wire empty, full;

// Instantiate the FIFO

cdc\_asynch\_fifo dut (

.wr\_clk(wr\_clk),

.rd\_clk(rd\_clk),

.rst(rst),

.wr\_en(wr\_en),

.rd\_en(rd\_en),

.write\_data(write\_data),

.read\_data(read\_data),

.empty(empty),

.full(full)

);

// Clock generation (asynchronous)

initial wr\_clk = 0;

always #10 wr\_clk = ~wr\_clk; // 50 MHz

initial rd\_clk = 0;

always #17 rd\_clk = ~rd\_clk; // ~29.4 MHz

// Test variables

integer i;

reg [7:0] write\_mem [0:15]; // Data written to FIFO

reg [7:0] read\_mem [0:15]; // Data read from FIFO

integer wr\_count, rd\_count;

initial begin

// Initialize

rst = 1;

wr\_en = 0;

rd\_en = 0;

write\_data = 8'd0;

wr\_count = 0;

rd\_count = 0;

// Apply reset

#50;

rst = 0;

#30;

// Write 8 values into FIFO

for (i = 0; i < 8; i = i + 1) begin

@(posedge wr\_clk);

wr\_en = 1;

write\_data = i + 8'd100;

write\_mem[i] = write\_data;

@(posedge wr\_clk);

wr\_en = 0;

// Wait for full flag if FIFO is full

if (full) begin

$display("FIFO FULL at write %0d", i);

// break;

end

wr\_count = wr\_count + 1;

end

// Wait a bit before reading

#100;

// Read all values from FIFO

for (i = 0; i < wr\_count; i = i + 1) begin

@(posedge rd\_clk);

rd\_en = 1;

@(posedge rd\_clk);

rd\_en = 0;

read\_mem[i] = read\_data;

// Wait for empty flag if FIFO is empty

if (empty) begin

$display("FIFO EMPTY at read %0d", i);

end

rd\_count = rd\_count + 1;

end

// Compare written and read data

$display("\n--- FIFO Data Check ---");

for (i = 0; i < rd\_count; i = i + 1) begin

if (read\_mem[i] !== write\_mem[i])

$display("ERROR: Data mismatch at index %0d: wrote %0h, read %0h", i, write\_mem[i], read\_mem[i]);

else

$display("OK: Data matched at index %0d: %0h", i, read\_mem[i]);

end

// Test full and empty flags

$display("\n--- FIFO Flag Check ---");

if (empty)

$display("FIFO is EMPTY as expected after reading all data.");

else

$display("ERROR: FIFO should be empty after all reads.");

if (!full)

$display("FIFO is NOT FULL as expected after all reads.");

else

$display("ERROR: FIFO should not be full after all reads.");

$finish;

end

// Optional: dump VCD for waveform viewing

initial begin

$dumpfile("fifo\_tb.vcd");

$dumpvars(0, cdc\_asynch\_fifo\_tb);

end

endmodule