**README: FIFO Module**

**Overview**

This document provides a comprehensive explanation of the FIFO (First-In-First-Out) module implemented in SystemVerilog. The module is designed for dual-clock domain operation and is parameterized to support flexible FIFO depth requirements. The FIFO efficiently manages data transfer between two clock domains using Gray code synchronization to mitigate metastability issues.

**Module Declaration**

module fifo #(parameter DEPTH = 70) (

input logic write\_clk,

input logic read\_clk,

input logic rst, // Active-high reset

input logic write\_en,

input logic read\_en,

input logic [79:0] data\_in,

output logic [79:0] data\_out,

output logic full,

output logic empty

);

**Purpose**

The FIFO buffer is used to:

* Decouple the producer (writer) and consumer (reader) logic.
* Safely transfer data across different clock domains.
* Manage data flow with backpressure using full and empty status signals.

**Port Descriptions**

| **Port** | **Direction** | **Description** |
| --- | --- | --- |
| write\_clk | Input | Clock signal for write domain. |
| read\_clk | Input | Clock signal for read domain. |
| rst | Input | Active-high reset that clears internal pointers and resets FIFO state. |
| write\_en | Input | Write enable signal; controls data writing. |
| read\_en | Input | Read enable signal; controls data reading. |
| data\_in | Input | 80-bit input data bus for FIFO writes. |
| data\_out | Output | 80-bit output data bus for FIFO reads. |
| full | Output | High when FIFO is full; prevents overwriting data. |
| empty | Output | High when FIFO is empty; prevents invalid reads. |

**Key Features**

* **Depth Parameterization**: Configurable depth using the DEPTH parameter.
* **Dual Clock Support**: Independent read and write clock domains for improved flexibility.
* **Gray Code Synchronization**: Ensures stable pointer synchronization across clock domains.
* **Full/Empty Handling**: Robust control logic to manage buffer state.
* **Efficient Memory Design**: Uses SystemVerilog's array construct for optimal storage.

**Design Details**

**Pointer Width Calculation**

localparam PTR\_WIDTH = $clog2(DEPTH);

* The pointer width is derived from the ceiling logarithm ($clog2) of the FIFO depth, ensuring proper addressing.

**Memory Storage**

logic [79:0] mem [DEPTH-1:0];

* Implements the FIFO storage as a 2D array where each entry stores 80-bit data.

**Binary to Gray Code Conversion**

function automatic logic [PTR\_WIDTH:0] bin\_to\_gray(input logic [PTR\_WIDTH:0] bin);

return (bin >> 1) ^ bin;

endfunction

* Gray code minimizes bit transitions, reducing the chance of metastability in multi-clock domain designs.

**Gray Code to Binary Conversion**

function automatic logic [PTR\_WIDTH:0] gray\_to\_bin(input logic [PTR\_WIDTH:0] gray);

logic [PTR\_WIDTH:0] bin;

bin = gray[PTR\_WIDTH];

for (int i = PTR\_WIDTH-1; i >= 0; i--)

bin[i] = bin[i+1] ^ gray[i];

return bin;

endfunction

* This function accurately decodes Gray code pointers for internal logic comparisons.

**Write Operation**

always\_ff @(posedge write\_clk or posedge rst) begin

if (rst) begin

write\_ptr <= 0;

write\_ptr\_gray <= 0;

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

mem[write\_ptr[PTR\_WIDTH-1:0]] <= data\_in;

write\_ptr <= write\_ptr + 1;

write\_ptr\_gray <= bin\_to\_gray(write\_ptr + 1);

end

end

* Data is written on the rising edge of write\_clk if write\_en is active and the FIFO is not full.

**Read Operation**

always\_ff @(posedge read\_clk or posedge rst) begin

if (rst) begin

read\_ptr <= 0;

read\_ptr\_gray <= 0;

data\_out <= 0;

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

data\_out <= mem[read\_ptr[PTR\_WIDTH-1:0]];

read\_ptr <= read\_ptr + 1;

read\_ptr\_gray <= bin\_to\_gray(read\_ptr + 1);

end

end

* Data is read on the rising edge of read\_clk if read\_en is active and the FIFO is not empty.

**Full and Empty Conditions**

always\_ff @(posedge write\_clk or posedge rst) begin

if (rst)

full <= 0;

else

full <= ((write\_ptr - read\_ptr) == DEPTH);

end

always\_ff @(posedge read\_clk or posedge rst) begin

if (rst)

empty <= 1;

else

empty <= (write\_ptr == read\_ptr);

end

* full is asserted when the FIFO reaches maximum capacity.
* empty is asserted when the FIFO has no valid data to read.

**Example Usage**

fifo #(70) my\_fifo (

.write\_clk(wclk), .read\_clk(rclk), .rst(reset),

.write\_en(wen), .read\_en(ren),

.data\_in(wdata), .data\_out(rdata),

.full(full\_flag), .empty(empty\_flag)

);

**Simulation Tips**

1. **Reset Properly:** Ensure the reset signal is asserted long enough to clear both the write and read pointers.
2. **Clock Synchronization:** Verify that write\_clk and read\_clk are correctly defined in the testbench with realistic timing constraints.
3. **Monitor Full/Empty Signals:** These flags should be actively monitored to avoid data corruption.

**Potential Improvements**

* Add a half\_full signal for better flow control.
* Implement burst read/write support for improved performance.
* Integrate a data integrity check mechanism for error detection.

**Conclusion**

This FIFO module offers a robust and efficient solution for data buffering across clock domains. By using Gray code synchronization and clear full/empty conditions, it ensures stable and reliable operation in complex digital systems.