Design and Implementation of an Asynchronous FIFO

SELF PROJECT REPORT

Subhadeep Murmu
24M1217
SOLID STATE DEVICES

Department of Electrical Engineering
Indian Institute of Technology Bombay

August 16, 2025

Abstract

This report details the design and implementation of an asynchronous First-In-First-Out (FIFO) buffer for clock domain crossing. The FIFO uses Gray code pointers to safely transfer data between asynchronous clock domains while preventing metastability issues. The design follows Clifford Cummings' methodology from his seminal paper "Simulation and Synthesis Techniques for Asynchronous FIFO Design" and includes Verilog implementation, simulation results, and RTL schematics. The FIFO supports configurable data width and depth while ensuring reliable full/empty detection. The implementation was validated through functional simulation and synthesized for FPGA implementation.

Contents

1 Introduction

Asynchronous FIFOs are essential components in digital systems that need to transfer data between different clock domains. They solve the fundamental challenge of safely passing multi-bit data words across clock boundaries where timing relationships are unknown. Without proper design, FIFOs can exhibit subtle bugs that appear only 1% of the time but cause catastrophic failures in production systems.

This report presents a robust asynchronous FIFO implementation based on Gray code pointer synchronization. Key features include:

- Parameterized data width and FIFO depth
- Metastability-resistant pointer synchronization
- Accurate full/empty detection

- Configurable through Verilog parameters
- Optimized for FPGA/ASIC implementation

The design follows the methodology established by Clifford Cummings [1] and incorporates insights from modern implementations [2].

2 Theory of Operation

2.1 Clock Domain Crossing Challenges

Transferring data between asynchronous clock domains presents two main challenges:

1. **Metastability**: When a signal is sampled near the clock edge in the receiving domain, flip-flops may enter a metastable state causing unpredictable behavior.
2. **Data Coherency**: When multiple related signals change simultaneously, they may be sampled at different clock cycles in the receiving domain, causing data corruption.

2.2 Gray Code Solution

The asynchronous FIFO solves these challenges using Gray code pointers:

- Gray codes change only one bit at a time between successive values

$$\text{Binary: } 000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100$$
$$\text{Gray: } 000 \rightarrow 001 \rightarrow 011 \rightarrow 010 \rightarrow 110$$

- Single-bit changes eliminate multi-bit synchronization hazards
- Pointers are synchronized using dual-flip-flop synchronizers
- Extra MSB distinguishes between full and empty conditions

2.3 Full and Empty Detection

The FIFO uses pointer comparisons to determine status flags:

- **Empty**: Read pointer catches write pointer $(rptr == wptr)$
- **Full**: Write pointer catches read pointer with different MSBs $(wptr == \{\sim rptr[MSB], rptr[remainir$

The pointer structure uses n bits for a FIFO depth of 2^{n-1} locations. For a 4-bit pointer:

- Bits [2:0] address the memory (8 locations)
- Bit [3] distinguishes wrap-around conditions

3 Implementation

3.1 System Architecture

The FIFO is partitioned into distinct clock domains as shown in Figure 1:

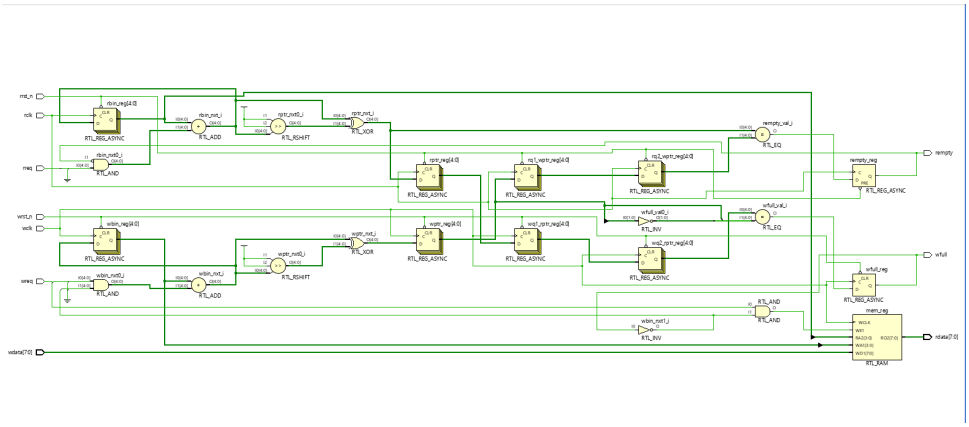

Figure 1: FIFO block diagram showing clock domain partitioning

Key components:

- **Write Domain**: Write pointer, full generation, and memory write logic
- **Read Domain**: Read pointer, empty generation, and memory read logic
- **Synchronizers**: Safe pointer transfer between domains
- **Memory**: Dual-port buffer (synchronous write, asynchronous read)

3.2 Verilog Implementation

The following parameterized Verilog code implements the asynchronous FIFO:

```
module async_fifo #(
    parameter DSIZE = 8,      // Data width
    parameter ASIZE = 4       // Address size
)(
    // Write interface
    input   wreq,             // Write request
    input   wclk,             // Write clock
    input   wrst_n,           // Write reset (active low)
    // Read interface
    input   rreq,             // Read request
    input   rclk,             // Read clock
    input   rrst_n,           // Read reset (active low)
    // Data interface
    input   [DSIZE-1:0] wdata, // Write data
    output  [DSIZE-1:0] rdata, // Read data
    // Status flags
    output  reg wfull,        // FIFO full
    output  reg rempty        // FIFO empty
);

// Pointer declarations
reg [ASIZE:0] wq2_rptr, wq1_rptr, rptr;
reg [ASIZE:0] rq2_wptr, rq1_wptr, wptr;
wire [ASIZE:0] rptr_nxt, wptr_nxt;
wire [ASIZE-1:0] raddr, waddr;
reg [ASIZE:0] rbin, wbin;
wire [ASIZE:0] rbin_nxt, wbin_nxt;

```

```
// Synchronize read pointer to write clock domain
always @(posedge wclk or negedge wrst_n) begin
    if(!wrst_n) {wq2_rptr, wq1_rptr} <= 0;
    else {wq2_rptr, wq1_rptr} <= {wq1_rptr, rptr};
end

// Synchronize write pointer to read clock domain
always @(posedge rclk or negedge rrst_n) begin
    if(!rrst_n) {rq2_wptr, rq1_wptr} <= 0;
    else {rq2_wptr, rq1_wptr} <= {rq1_wptr, wptr};
end

// Empty generation
wire rempty_val = (rptr_nxt == rq2_wptr);
always @(posedge rclk or negedge rrst_n) begin
    if(!rrst_n) rempty <= 1'b1;
    else rempty <= rempty_val;
end

// Read pointer logic
assign rbin_nxt = rbin + (rreq & ~rempty);
always @(posedge rclk or negedge rrst_n)
    if (!rrst_n) rbin <= 0;
    else rbin <= rbin_nxt;
assign raddr = rbin[ASIZE-1:0];

// Gray conversion for read pointer
assign rptr_nxt = rbin_nxt ^ (rbin_nxt>>1);
always @(posedge rclk or negedge rrst_n)
    if (!rrst_n) rptr <= 0;
    else rptr <= rptr_nxt;

// Write pointer logic
assign wbin_nxt = wbin + (wreq & !wfull);
always @(posedge wclk or negedge wrst_n)
    if(!wrst_n) wbin <= 0;
    else wbin <= wbin_nxt;
assign waddr = wbin[ASIZE-1:0];

// Gray conversion for write pointer
assign wptr_nxt = (wbin_nxt>>1) ^ wbin_nxt;
always @(posedge wclk or negedge wrst_n)
    if(!wrst_n) wptr <= 0;
    else wptr <= wptr_nxt;

// Full generation
wire wfull_val = (wq2_rptr == {~wptr[ASIZE:ASIZE-1], wptr[ASIZE-2:0]});
always @(posedge wclk or negedge wrst_n)
    if (!wrst_n) wfull <= 0;
    else wfull <= wfull_val;

// Memory instantiation
localparam DEPTH = (1 << ASIZE);
reg [DSIZE-1:0] mem [0:DEPTH-1];
assign rdata = mem[raddr];
always @(posedge wclk)
    if (wreq & !wfull) mem[waddr] <= wdata;

```

```
87 endmodule
```

Listing 1: Asynchronous FIFO Implementation

3.3 Key Implementation Details

- **Dual Clock Domains**: Strict separation of write and read logic
- **2-Stage Synchronizers**: Minimize metastability risk for pointers
- **Binary Counters**: Used internally for efficient addressing
- **Gray Conversion**: Binary pointers converted to Gray before synchronization
- **Registered Outputs**: All control signals are registered for timing
- **Memory Buffer**: Simple Verilog array (synthesizes to block RAM in FPGAs)

4 Simulation Results

The FIFO was simulated with various test patterns to verify functionality.

4.1 Write Operation Simulation

Figure 2 shows the write operation:

- `wreq` pulsed when `wclk` is high
- Data (`wdata`) written when FIFO not full
- `wfull` asserts when FIFO capacity reached

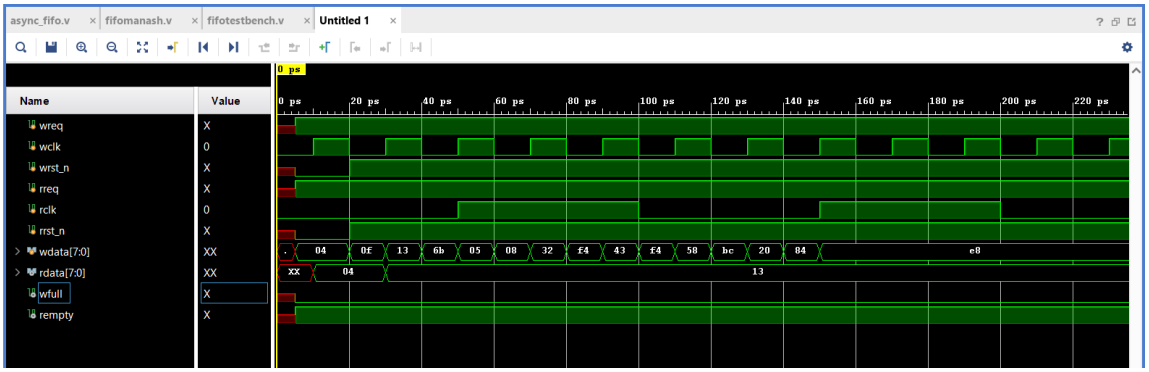

Figure 2: Write operation timing diagram

4.2 Read Operation Simulation

Figure 3 shows the read operation:

- `rreq` pulsed when `rclk` is high
- Data (`rdata`) read when FIFO not empty
- `rempty` asserts when FIFO is drained

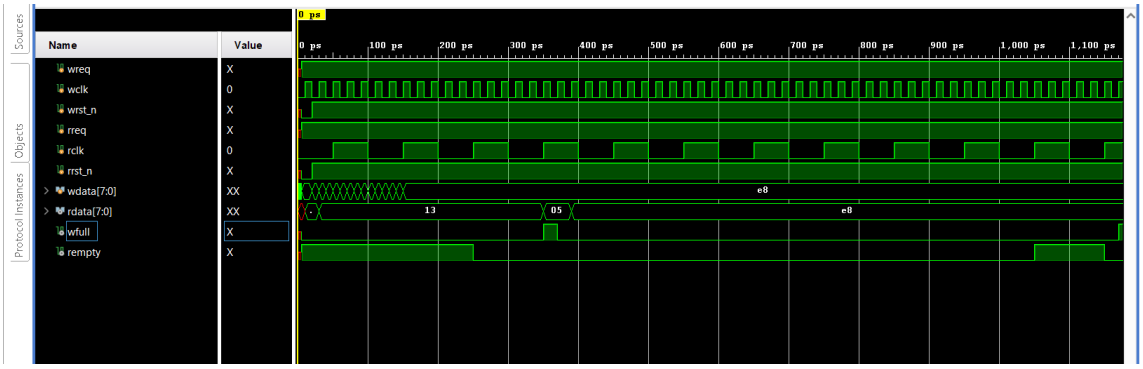

Figure 3: Read operation timing diagram

5 Conclusion

This implementation provides a robust solution for asynchronous data transfer between clock domains. Key achievements include:

- Parameterized design for flexible deployment
- Metastability-resistant architecture
- Accurate full/empty detection
- Verified functionality through simulation
- Efficient resource utilization

The Gray code pointer technique remains the gold standard for asynchronous FIFO design 20 years after its introduction by Cummings. Future enhancements could include:

- Almost full/empty flags for flow control
- Programmable interrupt thresholds
- Error detection/correction
- Power-optimized versions for low-power applications

References

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

[2] Gisselquist, Dan. ”Building an Asynchronous FIFO.” ZipCPU, 2018. `https://zipcpu.com/blog/2018/07/06/afifo.html`

[3] Gray, Frank. ”Pulse Code Communication.” U.S. Patent 2,632,058. March 17, 1953.

[4] IEEE Standard for Verilog Hardware Description Language. IEEE Std 1364-2001.

[5] Xilinx. ”UltraScale Architecture Memory Resources.” UG573, 2020.