Design and Implementation of an Asynchronous FIFO

Arish Faiz
Branch: EE6 (ICS)
Roll No: 23M1179

Abstract

This report presents a detailed summary of the principles and methodologies for designing a robust asynchronous First-In, First-Out (FIFO) memory buffer. Asynchronous FIFOs are fundamental components for safely transferring data between modules operating in different and unrelated clock domains. The core challenge in their design is the reliable generation of "full" and "empty" status flags, which depends on safely comparing write and read pointers across these asynchronous boundaries. This report details a proven method that employs Gray code pointers to ensure reliable synchronization. Key aspects of the design are explored, including the pointer generation mechanism, the logic for full and empty detection, the benefits of a pessimistic timing approach for status flags, and a modular design partitioning strategy that facilitates synthesis and static timing analysis.

Contents

1 Introduction to Asynchronous FIFOs

An asynchronous FIFO is a memory buffer used to pass data from a write clock domain to a separate, asynchronous read clock domain. They are a critical component in complex SoCs and FPGAs where multiple clock sources are common. While the concept seems straightforward, implementing a functionally correct and robust asynchronous FIFO is a significant design challenge. Incorrectly implemented FIFOs often function correctly most of the time but are susceptible to rare failure conditions that are extremely difficult to detect during simulation and can lead to costly product recalls if not identified before production.

The primary difficulty arises from the need to determine the FIFO's status (full or empty). This requires comparing the write pointer, which operates in the write clock domain, with the read pointer, which operates in the read clock domain. Passing multi-bit pointer values across clock domains is inherently risky and requires specialized design techniques to avoid data corruption due to metastability.

This report details a robust methodology for asynchronous FIFO design that centers on using Gray code pointers to safely synchronize pointer values, enabling reliable status flag generation.

2 Pointer Design: Binary vs. Gray Codes

2.1 The Problem with Binary Pointers

A standard binary counter is not suitable for use as a pointer in an asynchronous FIFO. The issue arises when the pointer value is synchronized from its source clock domain to the destination clock domain for comparison. When a binary counter increments, multiple bits can change simultaneously. A classic example is the transition from 7 (‘0111‘) to 8 (‘1000‘), where all four bits flip at once.

When these multiple changing bits are sampled by an asynchronous clock, there is no guarantee that they will all be captured on the same clock edge in the destination domain. This can lead to a completely erroneous pointer value being used for the full/empty comparison, causing the FIFO to either overflow (writing to a full FIFO) or underflow (reading from an empty FIFO). While handshaking protocols can be used to pass binary pointers safely, they introduce latency.

2.2 Gray Codes: A Robust Solution

Gray codes solve this synchronization problem elegantly. A key property of a Gray code is that only a single bit changes between any two consecutive values. This property eliminates the multi-bit synchronization issue; since only one bit changes at a time, it can be safely passed to the destination clock domain through a simple two-stage synchronizer without the risk of capturing an incorrect value. For this reason, Gray code counters are a common and robust approach for asynchronous FIFO pointers.

3 Implementing Full and Empty Logic

A critical design challenge is distinguishing between an empty FIFO and a full FIFO. In a simple implementation, both conditions occur when the write and read pointers are equal. The solution is to make the pointers one bit wider than required for addressing the memory depth. For a FIFO with $2^{(n-1)}$ locations, n-bit pointers are used.

3.1 Empty Condition

The FIFO is considered empty when the entire n-bit write pointer and read pointer are identical. This check is performed in the read clock domain by comparing the read pointer (‘rptr‘) with the synchronized version of the write pointer (‘$rq2\_wptr$‘).

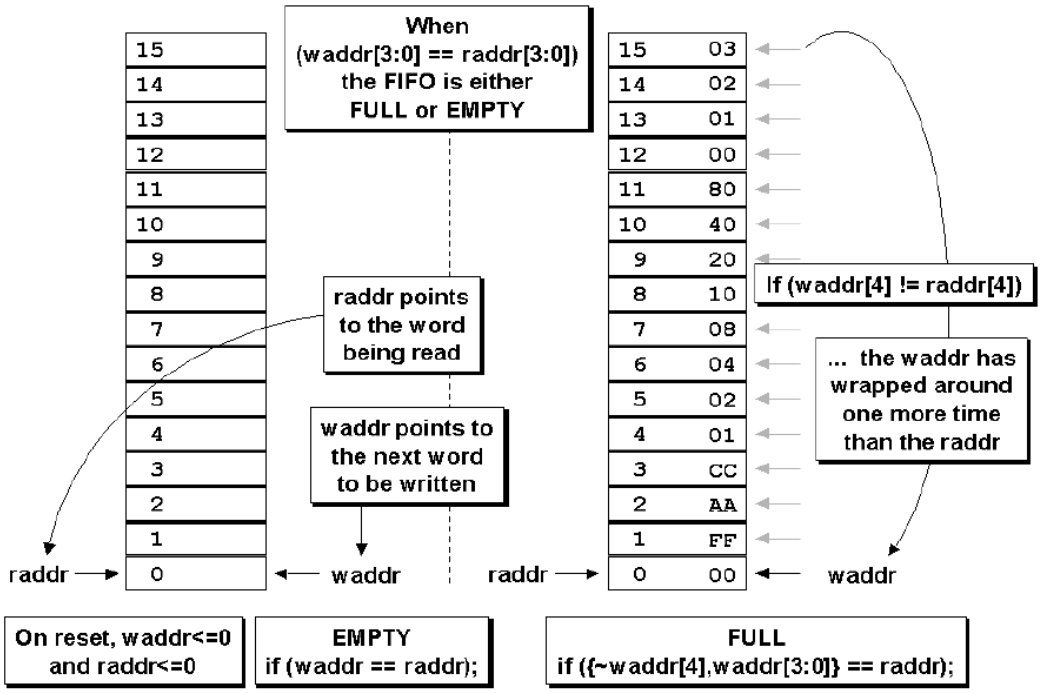

Figure 1: FIFO full and empty conditions using an extra pointer bit.

- An exact match (‘rptr == $rq2\_wptr$‘) means the pointers have wrapped the same number of times and are at the same location, indicating the FIFO is empty.
- The ‘rempty‘ flag is generated in the read clock domain to ensure the read logic stops immediately when the FIFO becomes empty.

3.2 Full Condition

The FIFO is considered full when the write pointer has wrapped around the address space one more time than the read pointer. This condition is detected when:

1. The most significant bits (MSBs) of the write and read pointers are different.
2. The remaining, lower bits of the pointers are identical.

The logic to check for the full condition is implemented in the write clock domain to ensure that write operations are halted immediately when the FIFO is full. A simplified version of the check compares the next value of the write pointer (‘wgraynext‘) with a modified version of the synchronized read pointer (‘$wq2\_rptr$‘): ‘assign wfull_val = (wgraynext == wq2_rptr[MSB], wq2_rptr[MSB-1], wq2_rptr[MSB-2:0]);‘ This logic accounts for the properties of Gray codes to correctly identify the full state.

4 Design Architecture and Partitioning

To facilitate synthesis and static timing analysis, the design is partitioned into distinct modules based on their clock domains. This ensures that logic within each module is fully synchronous to a single clock.

The main modules are:

- `wptr_full`: Operates in the write clock domain (‘wclk‘). It contains the write pointer (a Gray code counter) and the logic to generate the ‘wfull‘ flag.

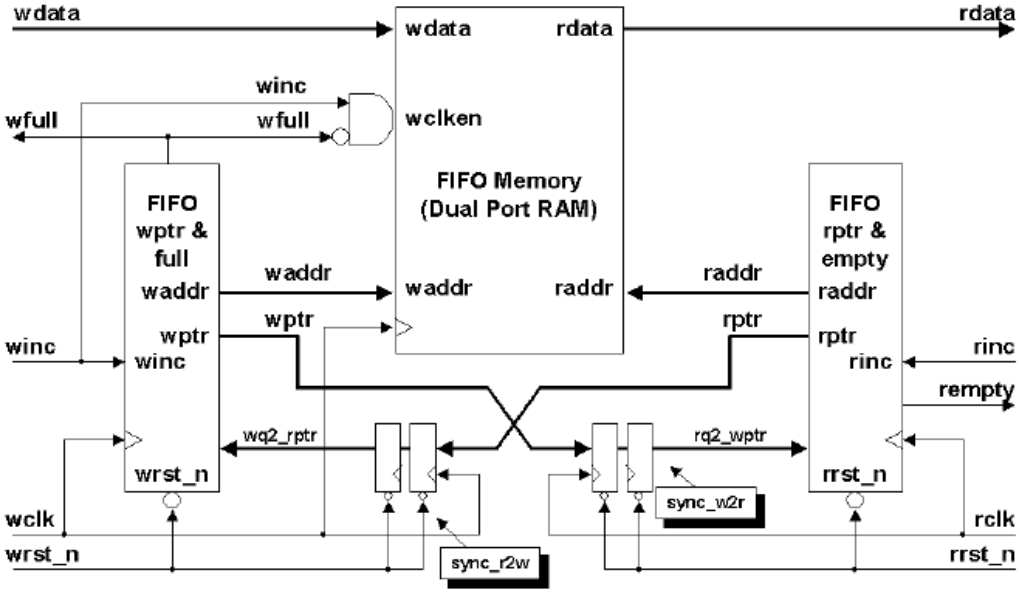

Figure 2: Block diagram showing the partitioned FIFO architecture.

- `rptr_empty`: Operates in the read clock domain (‘rclk‘). It contains the read pointer and the logic to generate the ‘rempty‘ flag.
- `sync_w2r`: A two-stage synchronizer that passes the write pointer from the ‘wclk‘ domain to the ‘rclk‘ domain.
- `sync_r2w`: A two-stage synchronizer that passes the read pointer from the ‘rclk‘ domain to the ‘wclk‘ domain.
- `fifomem`: A dual-port synchronous RAM that acts as the data buffer. It has a write port clocked by ‘wclk‘ and a read port that can be read from asynchronously.

This partitioning allows synthesis tools to optimize each clock domain independently and simplifies the process of applying timing constraints, as false paths between the asynchronous domains can be easily declared.

5 Conclusion

The design of a correct and robust asynchronous FIFO is a non-trivial task that requires deep understanding of clock domain crossing issues. The methodology presented by Clifford E. Cummings provides a reliable solution by addressing the core challenges head-on. The use of Gray code pointers is essential for safely synchronizing pointer values across asynchronous boundaries. A full/empty detection scheme based on an extended-width pointer, combined with pessimistic flag de-assertion, ensures that the FIFO will not overflow or underflow. Finally, a clean, modular partition of the design along clock domains is critical for successful hardware implementation, synthesis, and timing analysis. Following these principles results in a FIFO that is not only functionally correct but also verifiable and robust.

A Appendix: Verilog RTL Code

This appendix contains the complete Verilog RTL code for the asynchronous FIFO design, as presented in the reference paper.

A.1 Top-Level Module (`fifol.v`)

```
module fifol #(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 #(ASIZE) sync_r2w
              (.wq2_rptr(wq2_rptr), .rptr(rptr),
               .wclk(wclk), .wrst_n(wrst_n));

  sync_w2r #(ASIZE) 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),
               .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
```

Listing 1: Top-level Verilog code for the FIFO style #1 design

A.2 FIFO Memory (`fifomem.v`)

```
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);

// 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;

endmodule
```

Listing 2: Verilog RTL code for the FIFO buffer memory array

A.3 Read-to-Write Synchronizer (`sync_r2w.v`)

```
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
```

Listing 3: Verilog RTL code for the read-clock to write-clock domain synchronizer

A.4 Write-to-Read Synchronizer (`sync_w2r.v`)

```
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
```

Listing 4: Verilog RTL code for the write-clock to read-clock domain synchronizer

A.5 Read Pointer and Empty Logic (`rptr_empty.v`)

```
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
  assign raddr = rbin[ADDRSIZE-1:0];

  assign rbinnext = rbin + (rinc & ~rempty);
  assign rgraynext = (rbinnext>>1) ^ rbinnext;

  // FIFO empty when the next rptr == synchronized wptr
  assign rempty_val = (rgraynext == rq2_wptr);

  always @(posedge rclk or negedge rrst_n)
    if (!rrst_n) rempty <= 1'b1;
    else         rempty <= rempty_val;

endmodule
```

Listing 5: Verilog RTL code for the read pointer and empty flag logic

A.6 Write Pointer and Full Logic (`wptr_full.v`)

```
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
  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 = (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
```

Listing 6: Verilog RTL code for the write pointer and full flag logic