ASYNCHRONOUS FIFO DESIGN

Achyuth Krishna, Amrutha Regalla, Sathwik Reddy, Sai Sri Harsha Atmakuri

Abstract

In modern integrated circuits and systems-on-chip (SoCs), asynchronous FIFOs (First-In-First-Out) play a crucial role in enabling reliable communication and data transfer across clock domain boundaries. As SoCs incorporate multiple modules operating at diverse clock frequencies, asynchronous FIFOs provide a robust solution for clock domain crossing, decoupling read and write operations. These components offer several advantages, including elastic buffering to handle data rate mismatches, promotion of modular design practices, improved timing closure, and power optimization through techniques like clock gating and dynamic voltage and frequency scaling (DVFS). This paper will detail one method that is used to design, synthesize and analyse a Asynchronous FIFO between different clock domains using Gray code pointers that are synchronized into a different clock domain before testing for "FIFO full" or "FIFO empty" conditions.

Keywords: Asynchronous FIFO; Empty - full signal; Pointer – Synchronization, Metastability.

1. INTRODUCTION

In modern digital systems, managing data transfer between different clock domains is a critical challenge due to varying clock frequencies and phases. Asynchronous FIFOs (First In, First Out) buffers provide a robust solution for this issue, enabling reliable communication between components operating in distinct clock domains. However, designing asynchronous FIFOs presents several challenges, particularly in generating and synchronizing FIFO pointers, and accurately determining the full and empty status of the buffer. The primary difficulties arise from ensuring reliable pointer synchronization between the two clock domains and mitigating the effects of metastability, which can lead to data corruption or loss.

To overcome these challenges, advanced techniques are employed, such as using Gray code for pointers to minimize synchronization errors and implementing dual-clock domain synchronizers to safely transfer pointer values across clock domains. Additionally, precise logic is used to compare pointers and generate full and empty status flags accurately, ensuring robust operation even under varying operating conditions. This paper delves into these challenges and solutions, providing a comprehensive understanding of asynchronous FIFO design for reliable data transfer in complex digital systems.

2. Basic Knowledge of Asynchronous FIFO

2.1 Asynchronous FIFO Basic Module

First In First Out, or FIFO, refers to a first-in-first-out data cache. It differs from conventional memory in that there is no external read-write address line, making it very simple to use, but it has the drawback of only supporting sequential write and sequential read operations, with the

data address being automatically added by the internal read-write pointer to complete rather than being determined by the address line to read or write, as in conventional memory. The location is set.

Compared with synchronous FIFO read and write in the same clock signal, asynchronous FIFO reads data and writes data in different clock signals, so asynchronous FIFO is mainly applied to realize data transmission between different clock domains or as a data interface of different data widths. Asynchronous FIFO includes four modules: dual port RAM, write control module, read control module, and clock synchronization module.

Dual Port RAM: Dual port RAM (DPRAM) has two ports, allowing the read controller and the write controller to access the storage unit asynchronously at the same time. In the asynchronous FIFO, data can be written and read simultaneously, and real-time cache of written data can be realized. Dual port RAM can be read and written at any time and is very fast.

***Write Control Module*:** The function of the write control module is to control the write data and determine whether the DPRAM is full. The write address pointer is generated by the simultaneous action of the write clock and the write enable signal, and data is sequentially written to the dual port RAM. After the clock synchronization module delays two beats, the read address pointer is compared with the write address pointer, which generates a full write signal in the write control module (write clock domain).

Read Control Module: The function of the read control module is to control the read data and judge whether the DPRAM is empty. The read address pointer is created by the simultaneous operation of the read clock and the read enable signal, and it reads data sequentially from the DPRAM. After the clock synchronization module delays two beats, the write address pointer is compared with the read address pointer, and then the read control module (read clock domain) generates an empty read signal.

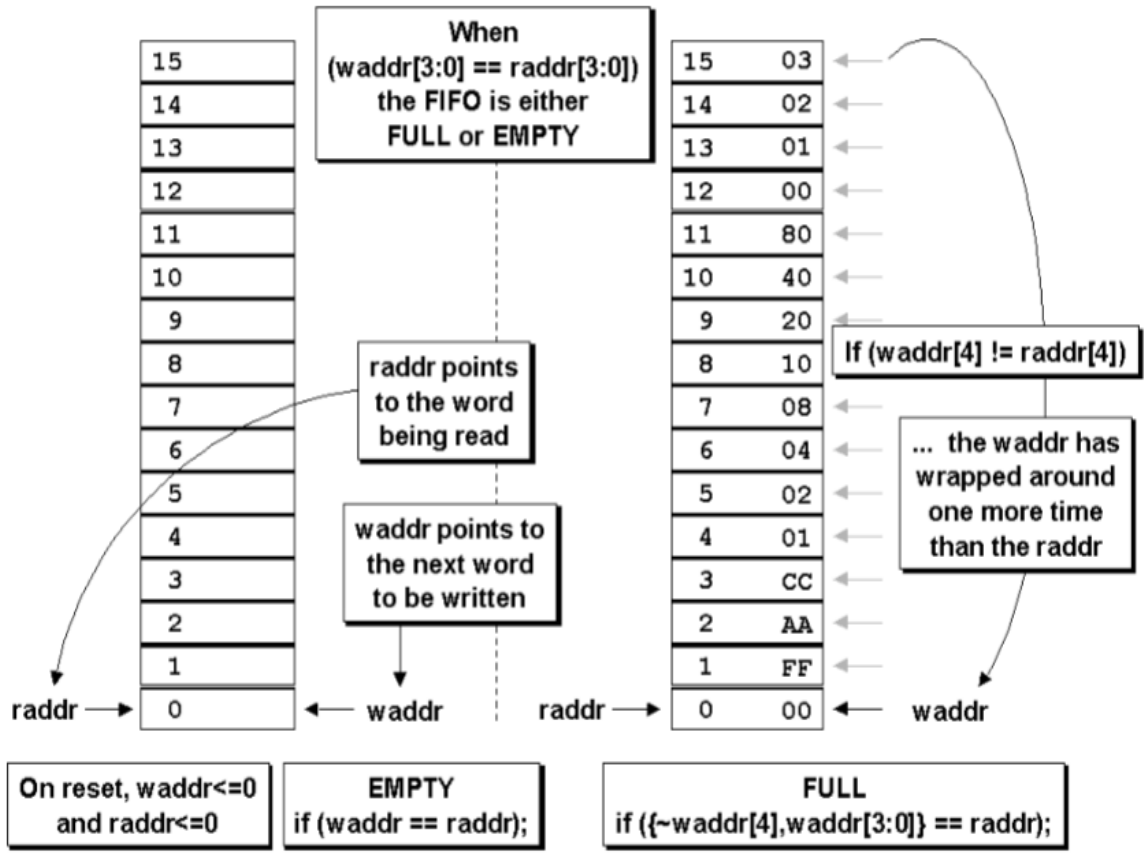
Clock Synchronization Module: This module compares the read address pointer with the write address pointer in the write clock domain after a delay of two beats and generates a full write signal. The write full signal is fed back to the write control module. The write enable function is lowered to stop data writing. The write address pointer is delayed by two beats. Comparing the read clock domain with the read address pointer generates an empty read signal. The empty read signal is fed back to the read control module, and the read enable is lowered to stop reading data.

2.2 Asynchronous FIFO Pointer

For comprehension of the FIFO (First-In, First-Out) architecture, it is necessary to understand how FIFO pointer's function. The write pointer always shows the next word to be written; so, on reset, both pointers are set to zero, indicating the next FIFO word position to be written. During a FIFO-write operation, the memory address indicated to by the write pointer is written first, followed by an increment to the next place to be written. Similarly, the read pointer is always set to the current FIFO word to be read. On reset, both pointers are reset to zero, resulting in an empty FIFO, and the read pointer links to incorrect data.

The read pointer, which still addresses the contents of the first FIFO memory word, immediately drives that first valid word onto the FIFO data output port to be read by the receiver logic as soon as the first data word is written to the FIFO. This is followed by the write pointer incrementing and the empty flag clearing. Because of this configuration, the read pointer always points to the next FIFO word that has to be read, thus the receiver logic may read the data word without requiring two clock periods. The receiver would clock once to output the data word from the FIFO and clock twice to capture the data word into memory if it was necessary to first increase the read pointer before reading a FIFO data word.

The FIFO is empty when the read and write pointers are equal, which occurs when both pointers are reset to zero via a reset operation or when the read pointer catches up to the write pointer after receiving the final word from the FIFO. The FIFO is full when the write pointer wraps around and catches up with the read pointer. This condition presents a problem since while the pointers are equal, the FIFO might be empty or filled.



3. Gray Code Counter

To understand Gray codes, keep in mind that the coding gap between neighboring words is only one bit. The most effective Gray code counters require power-of-2 counts in their sequence. While it is feasible to create a Gray code counter that counts an even number of sequences, converting to and from these sequences is not as straightforward as using the conventional Gray code. Gray code sequences cannot be of odd lengths, making it impossible to create a 23-deep code. This implies that the approach described in this paper is utilized to create 2^n FIFO.

The style #1 Gray code counter expects that the register bits output the Gray code value (ptr, wptr or rptr). The Gray code outputs are passed to a Gray-to-binary converter (bin), which is then passed to a conditional binary-value incrementer to generate the next-binary-count-value (binnext). This is then passed to a binary-to-Gray converter, which generates the next-Gray-count-value (graynext), which is then passed to the registers.

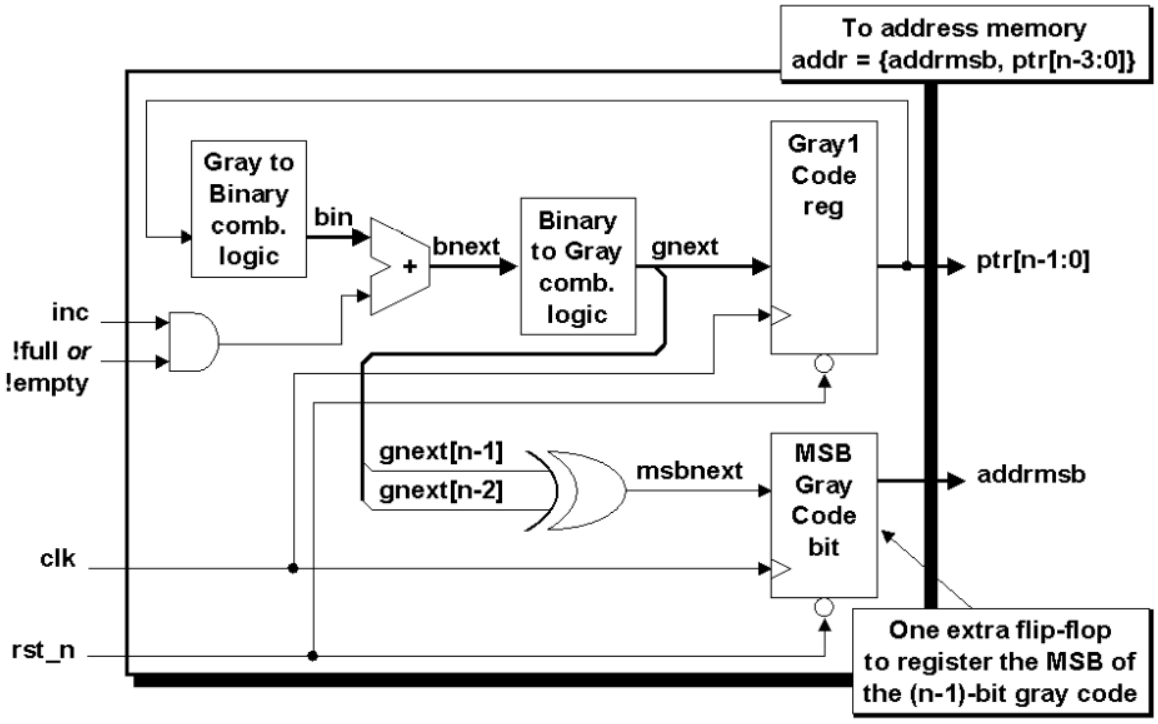


Fig 2: Dual n-bit Gray code Counter Block diagram

Figure 2 shows that the binary-value incrementer uses a "if not full" or "if not empty" test to prevent overflow or underflow of the FIFO buffer. Removing the full-testing logic from the FIFO write pointer can simplify the FIFO design if the logic block properly stops delivering data when the FIFO reaches full capacity. The FIFO pointer does not protect the buffer against overwriting. However, extra conditioning logic may be implemented to prevent write_enable signals from being activated during a full FIFO state.

To indicate error conditions that can only be cleared during reset, the pointer design could include a "sticky" status bit, either ovf (overflow) or unf (underflow), indicating additional FIFO write or read operations.

4. FIFO Pointer-Synchronization

To analyse the static timing of the FIFO design, the design has been partitioned into the following six SystemVerilog modules with the following functionality and clock domains:

- **Fifo.sv -** This is the top-level wrapper module for all clock domains. The top module serves as a wrapper for all FIFO modules in the architecture. If this FIFO is used in a bigger ASIC or FPGA design, the top-level wrapper may be removed to arrange the other FIFO modules into clock domains for better synthesis and timing analysis.
- **Fifomem.sv-** this is the FIFO memory buffer that is accessed by both the write and read clock domains. This buffer is most likely an instantiated, synchronous dual-port RAM. Other memory styles can be adapted to function as the FIFO buffer.
- **sync_r2w.sv** - synchronizes the read pointer to the write-clock domain. The wptr_full module will use the synchronized read pointer to create the FIFO full state. This module solely includes flip-flops synced to the writing clock. There is no other logic in this module.

- **sync_w2r.sv** -a synchronizer module that synchronizes the write pointer with the read-clock domain. The rptr_empty module will use the synchronized write pointer to create the FIFO empty state. This module solely includes flip-flops synced to the read clock. There is no other logic in this module.
- **rptr_empty.sv** - This module is totally synchronous to the read-clock domain and contains the FIFO read pointer and empty-flag logic.
- **wptr_full.sv** - This module is totally synchronous to the write-clock domain and contains the FIFO write pointer and full-flag logic.

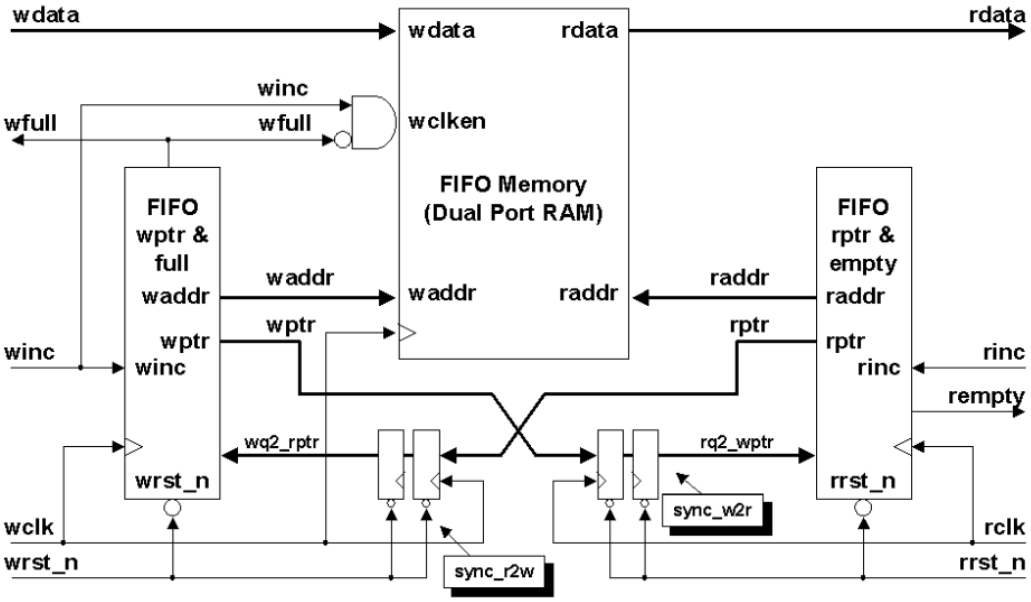


Fig 3: FIFO1 partitioning with synchronized pointer comparison

To execute FIFO full and FIFO empty tests with this FIFO approach, the read and write pointers must be transferred to the opposite clock domain for comparison.
As with previous FIFO systems, because the two pointers are produced from separate clock domains, they must be "safely" sent to the opposite clock domain. This paper demonstrates a strategy for synchronizing Gray code pointers so that only one pointer bit can change at a time.

5. Handling Full or Empty Conditions

The exact implementation of FIFO full and FIFO empty depends on the architecture. This paper's FIFO architecture generates the empty flag in the read-clock domain to detect it instantly when the FIFO buffer is empty. This occurs when the read pointer catches up to the write pointer, including the pointer MSBs. This paper's FIFO architecture generates the full flag in the write-clock domain, ensuring that it is noticed instantly when the FIFO buffer is full. This occurs when the write pointer catches up to the read pointer (except for differing pointer MSBs).

5.1 Empty Condition

The FIFO is empty when the read and synchronized write pointers are equal.
The empty comparison is straightforward to complete. FIFO memory buffers are addressed

using pointers that are one bit bigger than required. If both pointers' additional bits (MSBs) are equal, they have wrapped the same number of times. If the rest of the read pointer matches the synchronized write pointer, the FIFO is empty.
To synchronize the Gray code write pointer into the read-clock domain, use the **sync_w2r** module's pair of synchronizer registers. Gray code pointers allow for synchronized multi-bit transitions across clock domains by changing only one bit at a time.

```
  // FIFO empty when the next rptr == synchronized wptr or on reset

  assign rEmpty_val = (rgraynext == wptr_s);

  always_ff @(posedge rclk or negedge rrst)
    if (!rrst)
      rEmpty <= 1'b1;
    else
      rEmpty <= rEmpty_val;

endmodule
```

5.2 Full Condition

The full comparison is not as simple to do as the empty comparison. Pointers that are one bit larger than needed to address the FIFO memory buffer are still used for the comparison, but simply using Gray code counters with an extra bit to do the comparison is not valid to determine the full condition. The problem is that a Gray code is a symmetric code except for the MSBs.

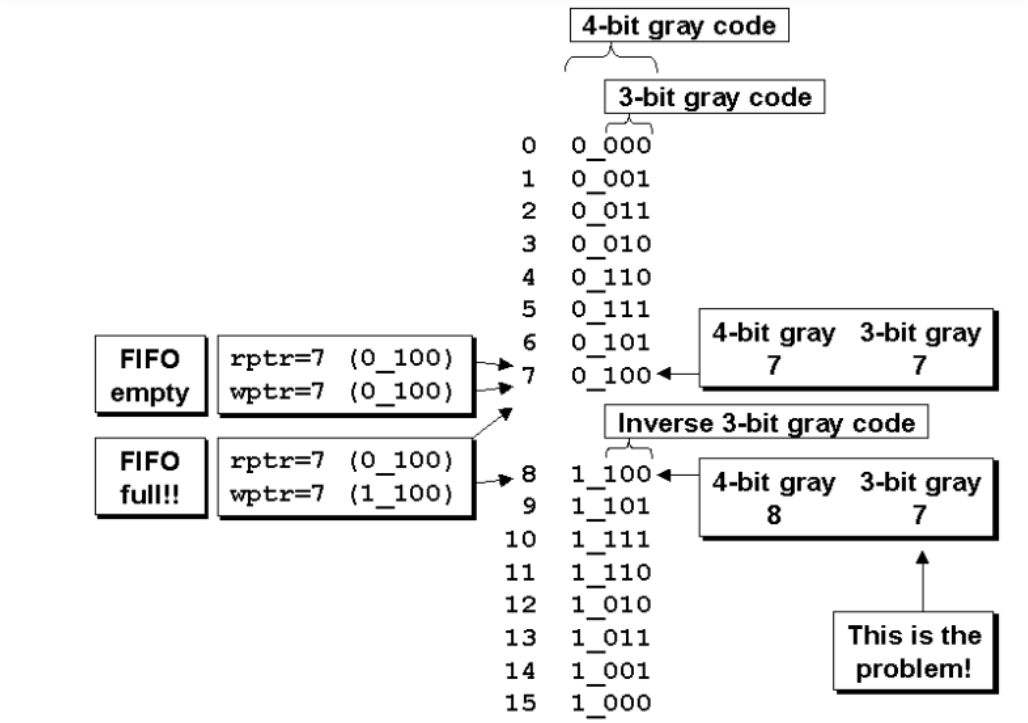


Fig 5: Problems associated with extracting a 3-bit Gray code from a 4-bit Gray code

Figure 5 illustrates an 8-deep FIFO. This example uses a 3-bit Gray code pointer to address memory and adds an additional bit (the MSB of a 4-bit Gray code) to test for full and empty situations. If the FIFO is allowed to fill the first seven places (words 0-6) and then emptied by reading back the same seven words, both pointers will be identical and point to address Gray-7 (indicating the FIFO is empty).

During the next write operation, the write pointer will increment the 4-bit Gray code pointer (only the 3 LSBs are used to address memory), resulting in different MSBs on the 4-bit pointers. However, the remaining write pointer bits will match the read pointer bits, indicating the FIFO full flag. This is incorrect! The FIFO is not full, and the three LSBs remain unchanged, indicating that the targeted memory location will overwrite the previously written location. This too is incorrect! This is one of the reasons for using the dual n-bit Gray code counter.

The correct method to perform the full comparison is accomplished by synchronizing the rptr into the wclk domain and then there are three conditions that are all necessary for the FIFO to be full:

- The wptr and the synchronized rptr MSB's are not equal (because the wptr must have wrapped one more time than the rptr).
- The wptr and the synchronized rptr 2nd MSB's are not equal (because an inverted 2nd MSB from one pointer must be tested against the un-inverted 2nd MSB from the other pointer, which is required if the MSB's are also inverses of each other - see Figure 5 above).
- All other wptr and synchronized rptr bits must be equal.

In order to efficiently register the wfull output, the synchronized read pointer is actually compared against the wgraynext (the next Gray code that will be registered in the wptr).

```
//Full when grey pointer is not equal to MSB and MSB-1 and is equal to the rest of the synchronized read pointer
 assign wFull_val = (wgraynext=={~rptr_s[ADDRSIZE:ADDRSIZE-1], rptr_s[ADDRSIZE-2:0]});


 always_ff @(posedge wclk or negedge wrst)
   if (!wrst)
     wFull <= 1'b0;
   else
     wFull <= wFull_val;
```

6. RTL Codes for FIFO Design

6.1 fifo.sv FIFO Top Level Module

The top-level FIFO module is parameterized and sub-blocks are instantiated with named port connections, as suggested. A frequent coding approach is to give top-level module instantiations the same name as the module itself. Using instance names that match module names in a hierarchical route simplifies debugging.

```
// Top level wrapper
//
module FIFO
#(
  parameter DSIZE = 8,
  parameter ASIZE = 8 //size 256 for fifo depth
 )
(
  input  logic winc, wclk, wrst,
  input  logic rinc, rclk, rrst,
  input  logic [DSIZE-1:0] wData,

  output logic [DSIZE-1:0] rData,
  output logic wFull,
  output logic rEmpty
);

  logic [ASIZE-1:0] waddr, raddr;
  logic [ASIZE:0] wptr, rptr, rptr_s, wptr_s;

  sync_r2w sync_r2w (.*);
  sync_w2r sync_w2r (.*);
  fifomem #(DSIZE, ASIZE) fifomem (.*);
  rptr_empty #(ASIZE) rptr_empty (.*);
  wptr_full #(ASIZE) wptr_full (.*);

endmodule
```

Example1: Top-level System Verilog code for the FIFO design

6.2 fifomem.sv - FIFO memory buffer

A FIFO memory buffer is usually an instantiated ASIC or FPGA dual-port, synchronous memory device. This module's RTL code allows for synthesis of the memory buffer into ASIC or FPGA registers. The Synopsys Design Ware team analysed the difference between instantiated vendor RAM and Verilog-declared RAM and discovered that for sizes up to 256 bits, Verilog-declared RAM had no performance or area loss compared to instantiated vendor RAM.

```
// FIFO memory
//
module fifomem
#(
  parameter DATASIZE = 8, // Memory data word width
  parameter ADDRSIZE = 8  // Number of mem address bits
)
(
  input  logic winc, wFull, wclk, rinc, rEmpty, rclk,
  input  logic [ADDRSIZE-1:0] waddr, raddr,
  input  logic [DATASIZE-1:0] wData,
  output logic [DATASIZE-1:0] rData
);

  // memory model
  localparam DEPTH = 1<<ADDRSIZE;

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

  always @(posedge rclk) //write if not empty and read increment is high
    if (rinc && !rEmpty)
      rData = mem[raddr];

  always @(posedge wclk)
    if (winc && !wFull)
     mem[waddr] <= wData;
     // mem[waddr] <= 1; use this for error injection

endmodule
```

Example 2 - Verilog RTL code for the FIFO buffer memory array

6.3 sync_r2w.sv - Read-domain to write-domain synchronizer

This is a simple synchronizer module, used to pass an n-bit pointer from the read clock domain to the write clock domain, through a pair of registers that are clocked by the FIFO write clock.

```
//
// Read pointer to write clock synchronizer
//
module sync_r2w
#(
  parameter ADDRSIZE = 8
)
(
  input  logic wclk, wrst,
  input  logic [ADDRSIZE:0] rptr,
  output logic [ADDRSIZE:0] rptr_s
);

  logic [ADDRSIZE:0] wq1_rptr;

  always_ff @(posedge wclk or negedge wrst) //2 cycle sync
    if (!wrst) {rptr_s,wq1_rptr} <= 0;
    else {rptr_s,wq1_rptr} <= {wq1_rptr,rptr};

endmodule
```

Example 3 -RTL code for the read-clock domain to write-clock domain synchronizer module

6.4 sync_w2r.sv - Write-domain to read-domain synchronizer

This is a simple synchronizer module, used to pass an n-bit pointer from the write clock domain to the read clock domain, through a pair of registers that are clocked by the FIFO read clock.

```
// Write pointer to read clock synchronizer
//
module sync_w2r
#(
  parameter ADDRSIZE = 8
)
(
  input  logic rclk, rrst,
  input  logic [ADDRSIZE:0] wptr,
  output logic [ADDRSIZE:0] wptr_s
);

  logic [ADDRSIZE:0] rq1_wptr;

  always_ff @(posedge rclk or negedge rrst) //2 cycle sync
    if (!rrst)
      {wptr_s,rq1_wptr} <= 0;
    else
      {wptr_s,rq1_wptr} <= {rq1_wptr,wptr};

endmodule
```

Example 4 -RTL code for the write-clock domain to read-clock domain synchronizer module

6.5 rptr_empty.sv - Read pointer & empty generation logic

This module encloses all of the FIFO logic that is generated within the read clock domain (except synchronizers). The read pointer is a dual n-bit Gray code counter. The n-bit pointer (

rptr) is passed to the write clock domain through the sync_r2w module. The (n-1)-bit pointer (raddr) is used to address the FIFO buffer. The FIFO empty output is registered and is asserted on the next rising rclk edge when the next rptr value equals the synchronized wptr value. All module outputs are registered for simplified synthesis using time budgeting. This module is entirely synchronous to the rclk for simplified static timing analysis.

```
// Read pointer and empty generation
//
module rptr_empty
#(
  parameter ADDRSIZE = 8
)
(
  input  logic rinc, rclk, rrst,
  input  logic [ADDRSIZE :0] wptr_s,
  output logic rEmpty,
  output logic [ADDRSIZE-1:0] raddr,
  output logic [ADDRSIZE :0] rptr
);

  logic [ADDRSIZE:0] rbin;
  logic [ADDRSIZE:0] rgraynext, rbinnext;

  always_ff @(posedge rclk or negedge rrst)
    if (!rrst)
      {rbin, rptr} <= '0;
    else
      {rbin, rptr} <= {rbin + (rinc & ~rEmpty), 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 or on reset

  assign rEmpty_val = (rgraynext == wptr_s);

  always_ff @(posedge rclk or negedge rrst)
    if (!rrst)
      rEmpty <= 1'b1;
    else
      rEmpty <= rEmpty_val;

endmodule
```

Example 5 - RTL code for the read pointer and empty flag logic

6.6 wptr_full.sv - Write pointer & full generation logic

This module encloses all of the FIFO logic that is generated within the write clock domain (except synchronizers). The write pointer is a dual n-bit Gray code counter. The n-bit pointer (wptr) is passed to the read clock domain through the sync_w2r module. The (n-1)-bit pointer (waddr) is used to address the FIFO buffer. The FIFO full output is registered and is asserted on the next rising wclk edge when the next modified wgnext value equals the synchronized and modified wrptr2 value (except MSBs). All module outputs are registered for simplified synthesis using time budgeting. This module is entirely synchronous to the wclk for simplified static timing analysis.

```
// Write pointer and full generation
//
module wptr_full
#(
  parameter ADDRSIZE = 8
)
(
  input  logic winc, wclk, wrst,
  input  logic [ADDRSIZE :0] rptr_s,
  output logic wFull,
  output logic [ADDRSIZE-1:0] waddr,
  output logic [ADDRSIZE :0] wptr
);

  logic [ADDRSIZE:0] wbin;
  logic [ADDRSIZE:0] wgraynext, wbinnext;


  always_ff @(posedge wclk or negedge wrst)
    if (!wrst)
      {wbin, wptr} <= '0;
    else
      {wbin, wptr} <= {wbin + (winc & ~wFull), wgraynext};

  // Memory write-address pointer (okay to use binary to address memory)
  assign waddr = wbin[ADDRSIZE-1:0];
  assign wbinnext = wbin + (winc & ~wFull);
  assign wgraynext = (wbinnext>>1) ^ wbinnext;
 //Full when grey pointer is not equal to MSB and MSB-1 and is equal to the rest of the synchronized read pointer
  assign wFull_val = (wgraynext=={~rptr_s[ADDRSIZE:ADDRSIZE-1], rptr_s[ADDRSIZE-2:0]});


  always_ff @(posedge wclk or negedge wrst)
    if (!wrst)
      wFull <= 1'b0;
    else
      wFull <= wFull_val;

endmodule
```

Example 6- RTL code for the write pointer and full flag logic

7. Simulation

8. Conclusion

9. References Uses / Citations/Acknowledgements

- S. Cummings, "FIFOs: Fast, predictable, and deep," in Proceedings of SNUG, 2002. [Online].Available:http://www.sunburstdesign.com/papers/CummingsSNUG2002SJ_FIFO1.pdf.
- S. Cummings, "FIFOs: Fast, predictable, and deep (Part II)," in Proceedings of SNUG, 2002.[Online].Available:http://www.sunburstdesign.com/papers/CummingsSNUG2002SJ_FIFO2.pdf.
- Putta Satish, "FIFO Depth Calculation Made Easy," [Online]. Available:https://hardwaregeeksblog.files.wordpress.com/2016/12/fifodepthcalculationmadeeasy2.pdf.
- Author et al., "Title of the Paper," in Proceedings of the Conference, 2015, pp. 123-456. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7237325.
- M. Last Name et al., "Designing Asynchronous FIFO," [Online]. Available: https://d1wqtxts1xzle7.cloudfront.net/56108360/EC109-libre.pdf.
- A. Author et al., "Title of the Paper," in Proceedings of the Conference, 2011, pp. 789-012.[Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6041338
- Author, "Title of the Video," [Online]. Available: https://www.youtube.com/watch?v=UNoCDY3pFh0
- Professor Slides & sample Academic paper
- Open AI, "Chat GPT," [Online].