Design of Asynchronous FIFO

Submitted By

Harsh Gupta(22M1169)

Sanjoy Kumar Basu(22M1180)

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Powai, Mumbai - 400076

Contents

List of Figures

Chapter 1

Introduction

The concept of FIFO

The word FIFO stands for First In First Out. The concept of FIFO is used in many branches of life, namely, accounting, stocks rotation and memory management. In memory management, FIFO is used to capture data, essentially in applications where the rate at which it is captured(written) is different from the rate at which it is accessed(read). In many applications where a peripheral device connected to a processor captures and sends data to the processor at a rate, either more or less than the processor speed. For example, normal cameras "record" videos at 60 frames per second. The rate at which the camera sends data to the processor to "process" the data and store in the memory is significantly lower than the processor speed itself, hence the data sent by camera is essentially "asynchronous". For such applications, data needs to cross clock domains from the camera's clock domain to the processor's clock domain. This is done using an asynchronous FIFO of pre-defined "depth" and "width".

While the concept of FIFO is straight forward, the hardware implementation of FIFO is quite a challenge. Mainly because the read and write rates are different. To ensure the FIFO doesn't overflow or underflow and to ensure data integrity, the read domain pointers need to synchronise to write domain and vice-versa to generate a few status signals. These signals can be "full", "empty", "almost full" and "almost empty". The aim of this project is to design a dual clock FIFO with "full" and "empty" status signals and understand the working each block involved in the operations.

1.1 Background

Synchronous FIFO Pointers: -

A FIFO where writes to, and reads from the FIFO buffer are conducted in the same clock domain. For synchronous FIFO design, one implementation counts the number of writes to, and reads from the FIFO buffer to increment, decrement or hold the current fill value of the FIFO buffer. The FIFO is full when the FIFO counter reaches a predetermined full value and the FIFO is empty when the FIFO counter is zero. Unfortunately, for asynchronous FIFO design, the increment-decrement FIFO fill counter cannot be used, because two different and asynchronous clocks would be required to control the counter. To determine full and empty status for an asynchronous FIFO design, the write and read pointers will have to be compared.

Asynchronous FIFO Pointers: -

In order to understand FIFO design, one needs to understand how the FIFO pointers work. The write pointer always points to the next word to be written. Therefore, on reset, both pointers are set to zero, which also happens to be the next FIFO word location to be written. On a FIFO-write operation, the memory location that is pointed to by the write pointer is written, and then the write pointer is incremented to point to the next location to be written. Similarly, the read pointer always points to the current FIFO word to be read. Again, on reset, both pointers are reset to zero, the FIFO is empty and the read pointer is pointing to invalid data (because the FIFO is empty and the empty flag is asserted). As soon as the first data word is written to the FIFO, the write pointer increments, the empty flag is cleared, and the read pointer that is still addressing 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.

Gray Code Counter: -

A common approach to FIFO counter-pointers, is to use Gray code counters. Gray codes only allow one bit to change for each clock transition, eliminating the problem associated with trying to synchronize multiple changing signals on the same clock edge. The first fact to remember about a Gray code is that the code distance between any two adjacent words is just 1 (only one bit can change from one Gray count to the next). The second fact to remember about a Gray code counter is that most useful Gray code counters must have power-of-2 counts in the sequence. It is possible to make a Gray code counter that counts an even number of sequences but conversions to and from these sequences are generally not as simple to do as the standard

Gray code. Also note that there are no odd-count-length Gray code sequences. This Gray code counter utilize the binary carry structure, simplify the Gray-to-binary conversion, reduce combinational logic, and increase the upper frequency limit of the Gray code counter. The binary counter conditionally increments the binary value, which is passed to both the inputs of the binary counter as the next-binary-count value, and is also passed to the simple binary-to-Gray conversion logic, consisting of one 2-input XOR gate per bit position. The converted binary value is the next Gray-count value and drives the Gray code register inputs.This implementation requires twice the number of flip-flops, but reduces the combinatorial logic and can operate at a higher frequency.

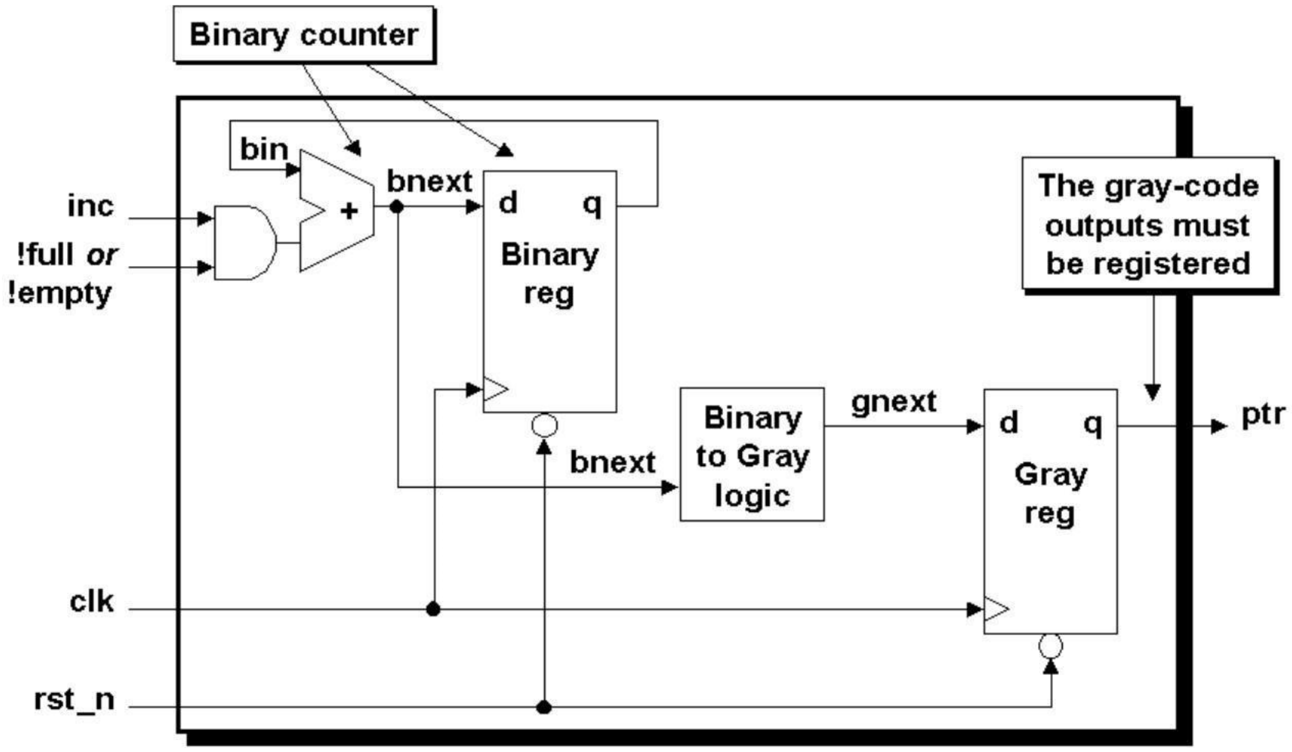


Figure 1.1: Gray code counter

FIFO Full empty detection :-

The FIFO is empty when the read pointer and the synchronized write pointer are equal. The empty comparison is simple to do. Pointers that are one bit larger than needed to address the FIFO memory buffer are used. If the extra bits of both pointers (the MSBs of the pointers) are equal, the pointers have wrapped the same number of times and if the rest of the read pointer equals the synchronized write pointer, the FIFO is empty. The Gray code write pointer must be synchronized into the read-clock domain through a pair of synchronizer registers found in the *syncw*2*r* module. Since only one bit change at a time using a Gray code pointer, there is no prob-lem synchronizing multi-bit transitions between clock domains. In order to efficiently register the rempty output, the synchronized write pointer is actually compared against the rgraynext (the next Gray code that will be registered into the rptr).

Since the full flag is generated in the write-clock domain by running a comparison between the write and read pointers, one safe technique for doing FIFO design requires that the read pointer be synchronized into the write clock domain before doing pointer comparison. 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.

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: -

1. The wptr and the synchronized rptr MSB’s are not equal (because the wptr must have wrapped one more time than the rptr).
2. 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.
3. 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 wgnext (the next Gray code that will be registered in the wptr).

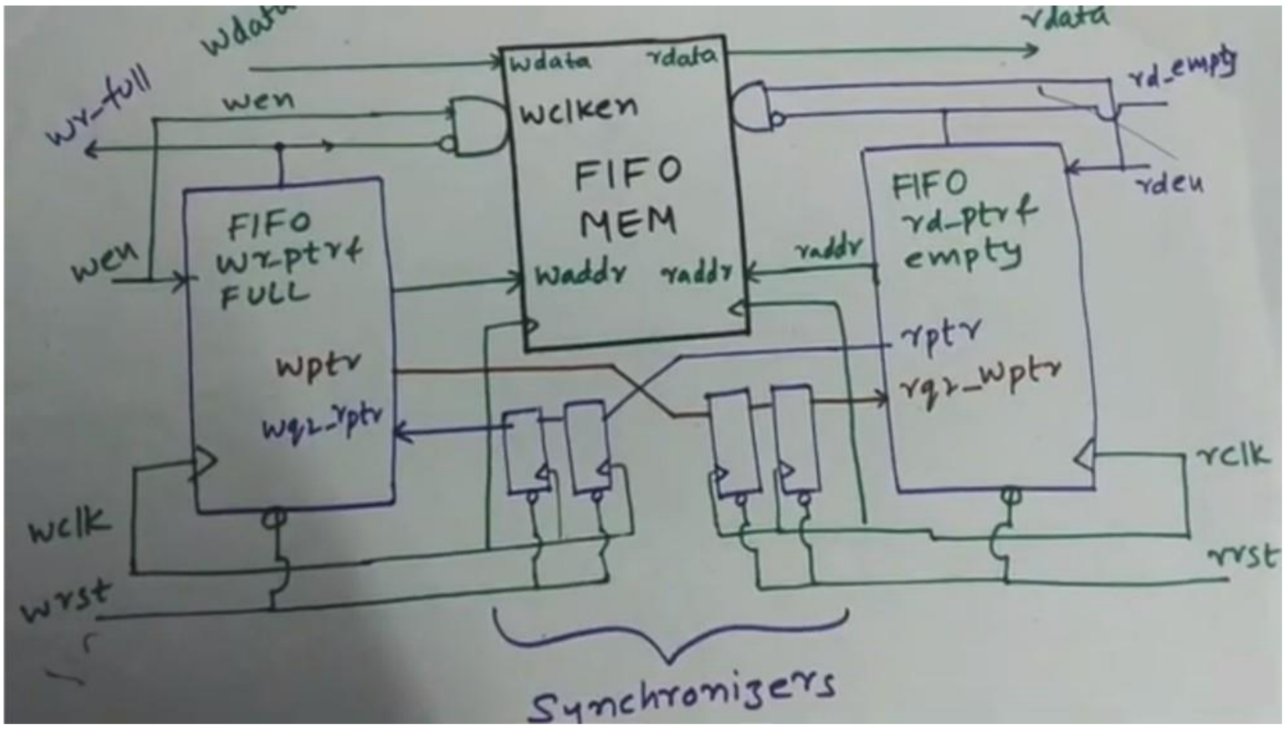


Figure 1.2: Block diagram of FIFO

Chapter 2

Simulation Results

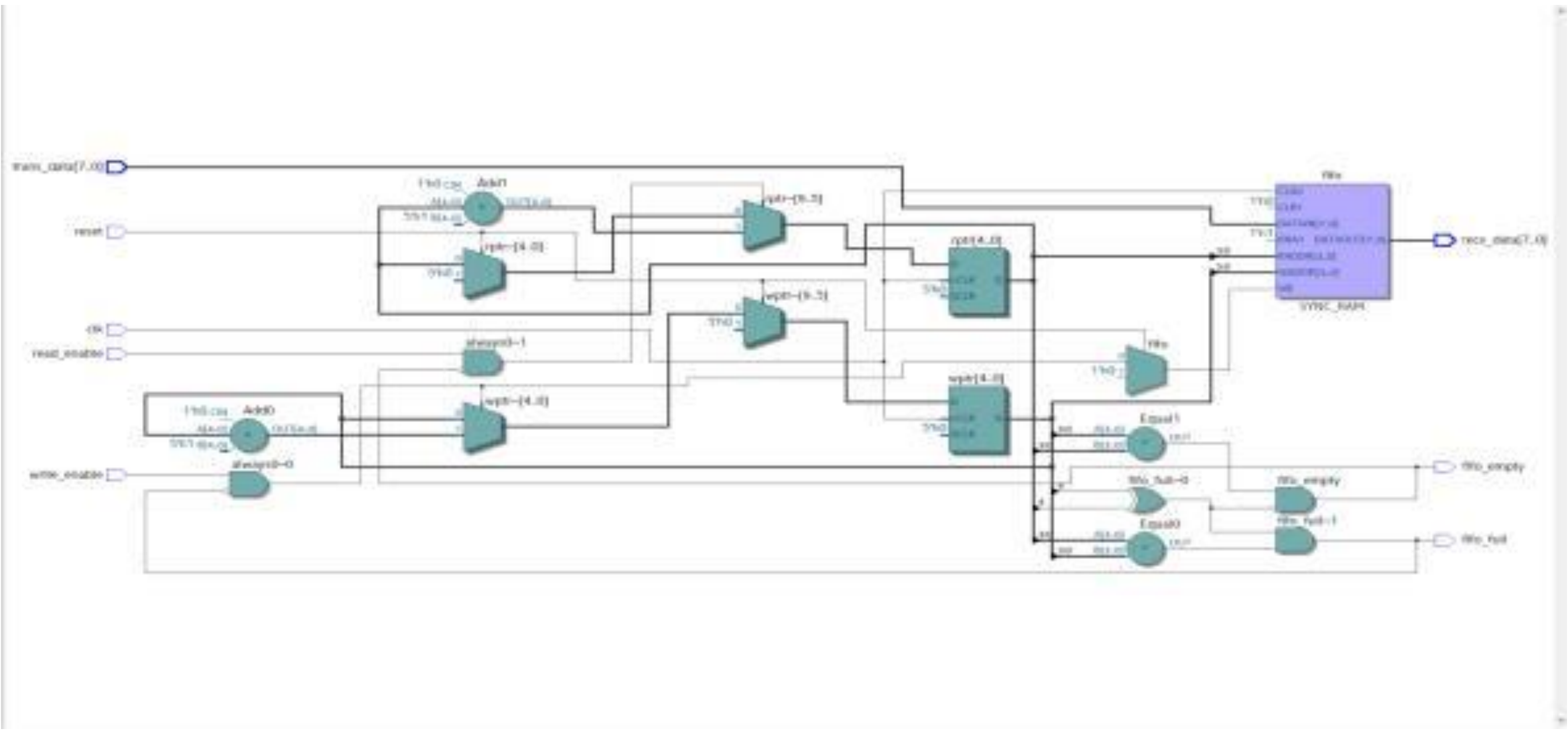


Figure 2.1: Netlist view of Synchronous FIFO

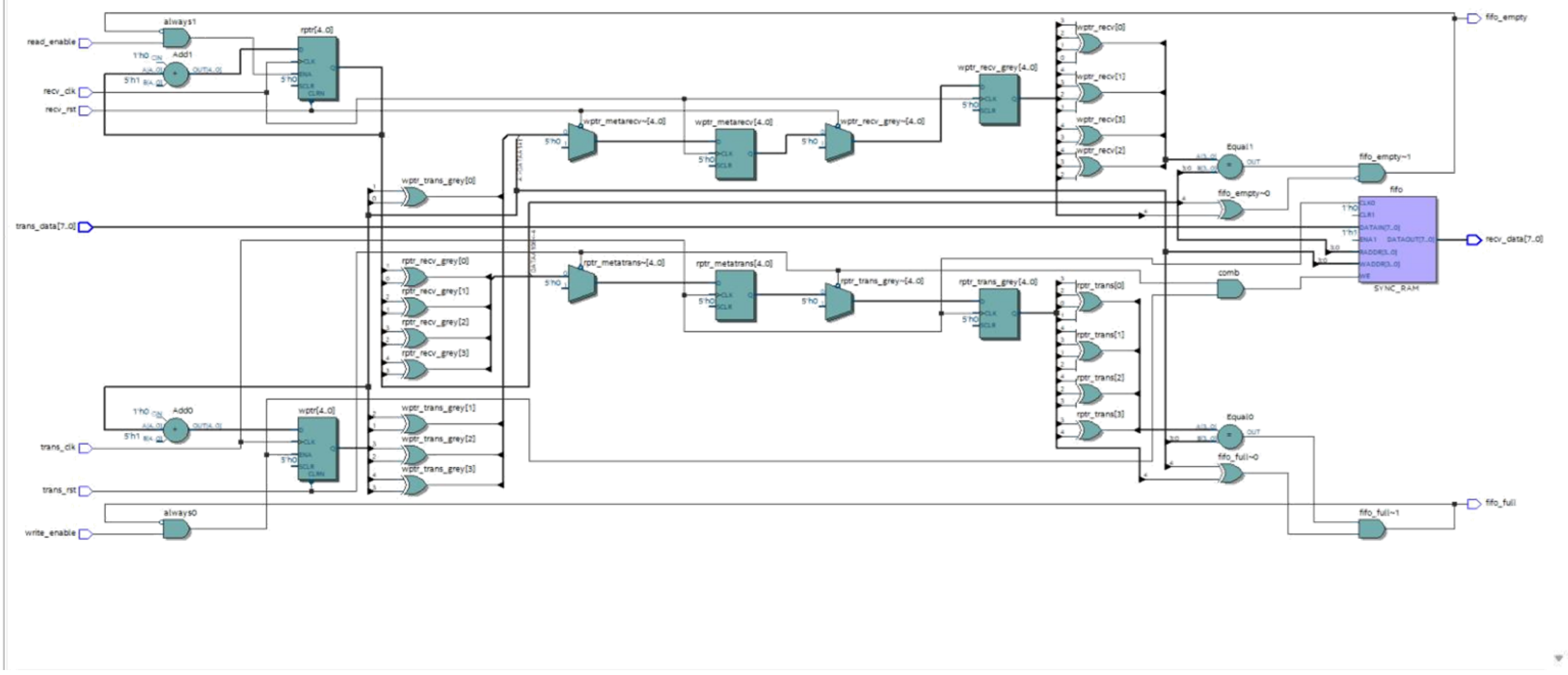
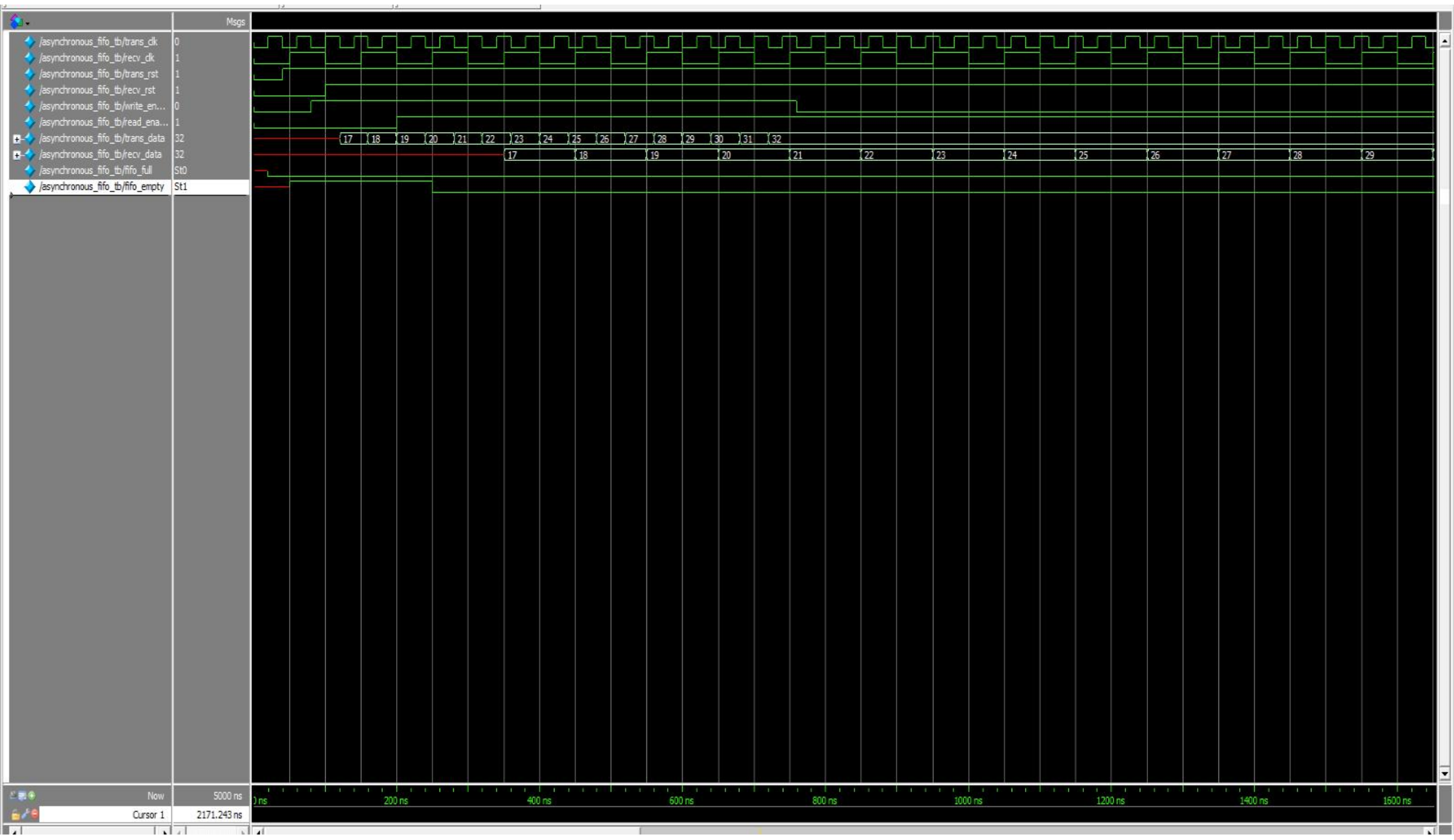


Figure 2.1: Netlist view of Asynchronous FIFO



/asynchronous_fifo_tb/trans_clk
/asynchronous_fifo_tb/recv_clk
/asynchronous_fifo_tb/trans_rst
/asynchronous_fifo_tb/recv_rst
/asynchronous_fifo_tb/write_en...
/asynchronous_fifo_tb/read_ena...
/asynchronous_fifo_tb/trans_data
/asynchronous_fifo_tb/recv_data
/asynchronous_fifo_tb/fifo_full
/asynchronous_fifo_tb/fifo_empty
Msgs
Now 5000 ns
Cursor 1 2171.243 ns

Figure 2.3: Simulation Result

We used below frequency of clock for this simulation,

Transmitter frequency = (1/40ns) = 25 MHz

Receiver frequency = (1/100ns) = 10 MHz

Chapter 3

Conclusion

3.1 Conclusion and Outlook

Asynchronous FIFO design requires careful attention to details from pointer generation tech-niques to full and empty generation. Synchronization of FIFO pointers into the opposite clock domain is safely accomplished using Gray code pointers. Generating the FIFO-full status is perhaps the hardest part of a FIFO design. Dual n-bit Gray code counters are valuable to syn-chronize and n-bit pointer into the opposite clock domain and to use an (n-1)-bit pointer to do “full” comparison. Generating the FIFO-empty status is easily accomplished by comparinge-qual the n-bit read pointer to the synchronized n-bit write pointer. Careful partitioning of the FIFO modules along clock boundaries with all outputs registered can facilitate synthesis and static timing analysis within the two asynchronous clock domains.