Comparison of Synchronization techniques in Pointer FIFOs

Shomit Das

***Abstract*—Use of FIFOs in digital designs for buffering and flow control has been widespread since many years. As feature sizes continue their south-bound journey, more and more complex circuitry has found its way onto the chip. The emergence of system on chip (SOC) and networks on chip (NOC) for internal connections have made it imperative to ensure correct flow of data across the chip especially in light of the performance shortcomings of global interconnects. One of the design challenges that arises is to ensure safety and fidelity for this data transfer. FIFO cells are a good way to accomplish this task. The FIFOs now require to serve two masters (clocks) instead of one. Such FIFOs are called Asynchronous FIFOs. There are many clever architectures and implementations of FIFOs as enlisted in the references. These aim to maximize benefits in latency or throughput or both. This work aims at comparing two pointer synchronization techniques in FIFOs.**

***Index Terms*— FIFO, Gray code, Token ring, pointer synchronization.**

I. Introduction

THE on-chip network concept becomes increasingly attractive in light of the interconnect problem faced by designers today. Interconnect is now the bottleneck in terms of performance, power and area in chip design. Large multi-clock systems on chip (SoC) manifest themselves as Globally Asynchronous Locally Synchronous (GALS) systems. Synchronous islands are connected together on a chip and they communicate via handshaking to asynchronous domains. Thus, data transfer has to take place between two independent clock domains. This is where asynchronous FIFOs come in.

A FIFO is nothing but a memory buffer between two asynchronous systems with simultaneous write and read access to and from the FIFO, these accesses being independent of one another. Data written into a FIFO is sequentially read out in a pipelined manner, such that the first data written into a FIFO will be the first data that is read out. The fundamental architecture of a FIFO has a write port, a read port and memory locations. Each port has its own associated pointer which points to a location in memory. After a reset, both write and read pointers will be at the first memory location within the FIFO. Every write operation will cause the write pointer to increment to the next address in memory, while every read operation will increment the read pointer. Write and read operations loop in a circular manner. It is important to know when the FIFO is full or empty in order to prevent an Overflow or Underflow condition. Thus extra logic is needed to generate the FULL and EMPTY status flags. An EMPTY flag is generated by the comparison of the read and write pointers and results in the number of memory locations between them being zero. If the number of memory locations between them is the maximum FIFO depth, the FULL flag is asserted. It is clear that an EMPTY FIFO cannot be read from and a FULL FIFO cannot be written into. Efficient generation and comparison of these status flags depends on the correct design of the FIFO and is the topic of this study. Sections II and III provide information about the two synchronization and comparison techniques used in this project. Section IV details the implementation of the FIFOs. Section V reports a few results. Conclusion and future work is outlined in Section VI.

· Author is with the Department of Electrical Engineering at the University of Utah.

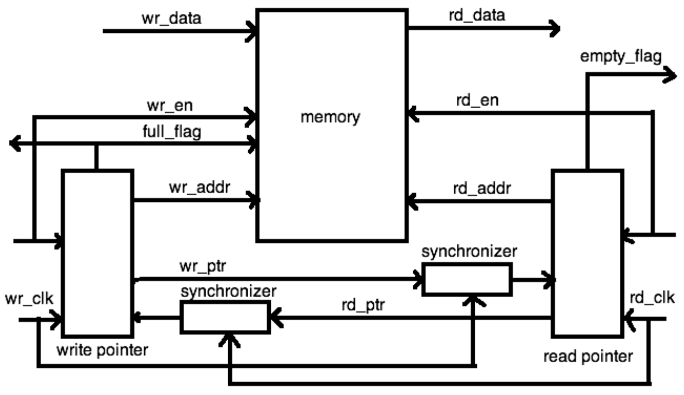

Fig 1: FIFO Architecture

II. Gray Code Pointer

A. The Code

The problem with trying to synchronize a binary count value from one clock domain to another is especially problematic because of the multiple bit changes possible. So, we need a code that has a Hamming distance of '1' between its adjacent vectors. Gray code is one such possibility.

The reflected binary code, or the Gray code was originally designed to prevent spurious output from electromechanical switches. While transferring pointer information between independent clock domains, we need to send each bit, new or old of the pointer. If more than one bit in the multi-bit pointer is changing at the sampling point, an incorrect binary value can be propagated. By guaranteeing that only one bit can be changing, Gray codes guarantee that the only possible sampled values are the new or old multi-bit value.

B. *The Implementation*

The beautiful thing about Synthesis is that you can specify the behavior of a circuit and (if the circuit complexity is within reason) let the tool worry about configuring it. All we need in this implementation is to convert the Gray value to binary, increment it, convert it back to Gray and store it. Fig. 2 shows how this is done.

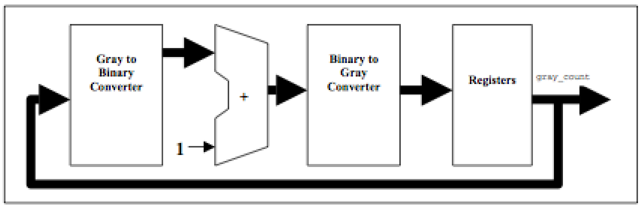

Fig 2: Gray code implementation

III. Token Ring

A. *Basics*

A token ring is a succession of nodes interconnected in a circular manner that contains tokens. It can be described with N registers (with Enable signal) interconnected in serial, like a cyclic shift register. If the Enable signal is true, the content of the register is shifted at the rising edge of the clock, otherwise the register maintains its data. A token is represented by the logic state 1 of the register. This arrangement can be used as a state machine. The position of the token defines the state of the state machine. It is better to have two tokens rather than one in order to eliminate error. If there are two consecutive tokens, at any given clock input, only one register changes state. It is easy to determine the token position even if one of the tokens is lost while sampling.

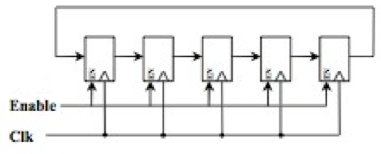

Fig 3: Token Ring

B. *Token Ring Counter*

A counter can be thought of as just a state machine, with the count value as the state code. So, as long as we keep track of the state, we do not need to know the exact count value. This makes it possible for the token ring to be used as a counter. Since the token ring is nothing more than a series of shift registers connected in a circular fashion, we can be positive that any change will not be multi-bit, thus avoiding pointer synchronization issues.

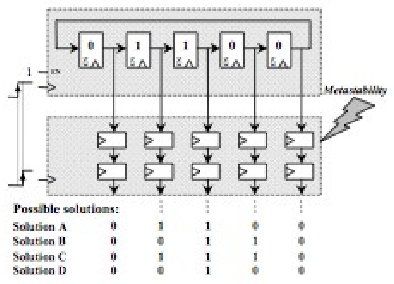

Fig 4: Avoiding Metastability

IV. FIFO Implementation

A. *Gray Code FIFO*

Fig. 5 shows the basic block diagram of the Gray code implementation of the pointer FIFO. The basic modules and their realizations are explained.

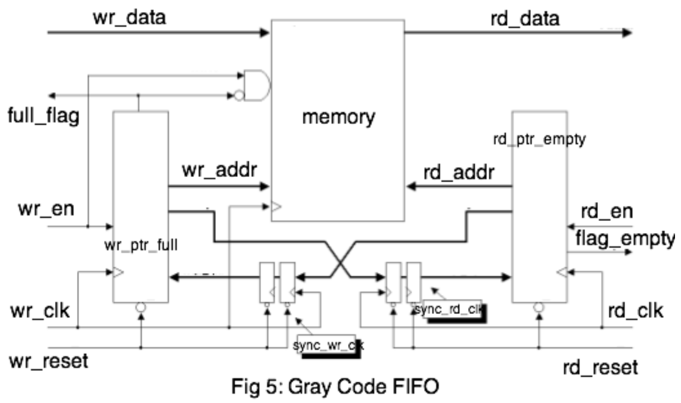

Fig 5: Gray Code FIFO

The memory module is comprised of nothing but synthesized register cells. It is dual ported and runs on the wr_clk frequency. The memory buffer puts out data on the rd_data bus. On every clock edge, it checks if it is enabled to write and if the FIFO is not already full. If these two conditions are met, it accepts input data from the wr_data bus and stores it to the FIFO memory location pointed to by the write pointer wr_addr.

We have two synchronizer modules in order to compare the read and write pointers correctly. The usual and reliable two-flop synchronizer is used for this module. The read pointer needs to be synchronized in the wr_clk domain and the write pointer similarly needs to be synchronized with the rd_clk. It is done as follows. At the positive edge of the read clock, the synchronizer checks to see whether the read reset signal is asserted. If not, the flop transfers its input to its output. After two flops, the write pointer synchronized with the read clock is obtained. Two flops are required to avoid exceptionally long response times when clock and data conflict.

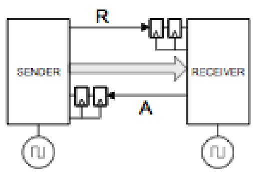

Fig 6: Two Flop Synchronizer

Similar circuit is required to synchronize the read pointer to the write clock domain.

Empty flag generation is done by comparing the read pointer to the synchronized write pointer. If the values are the same, the FIFO is empty because it means that both pointers have wrapped the same number of times and point to the same location, i.e. every data that has been written has also been read out.

Generation of the full flag is the most difficult part of this FIFO design. The FIFO is full when the pointers, save the MSB are the same. Since the MSB is different, the write pointer has looped one more time than the read pointer and now points to the same location, thereby indicating that the FIFO is full. This MSB difference makes the logic for full detection somewhat different than that of empty detection.

B. Token Ring FIFO

The token ring FIFO requires some different circuitry than the Gray code counter. The basic block diagram is shown in Fig. 7.

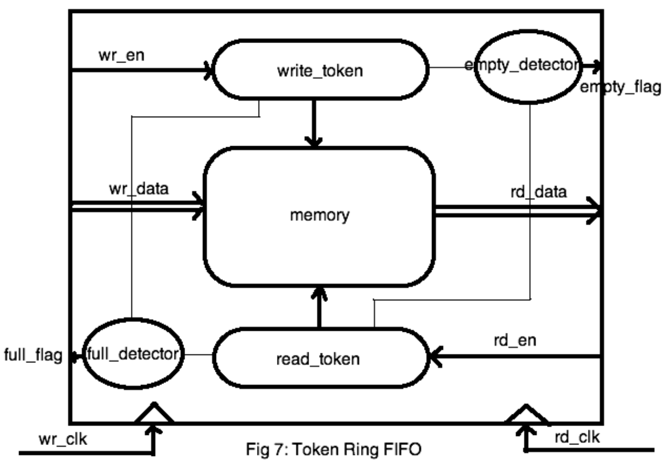

Fig 7: Token Ring FIFO

The memory module is quite similar to that of the Gray code FIFO. It is again comprised of shift registers.

The read and write pointers are implemented not as binary or gray code counters, but as token rings. The read token logic is as follows. At every positive edge of the read clock, or at asynchronous reset inputs, sample the signal. Keep a track of each register output. Define the set of register outputs as the read vector. At every clock edge, if the rd_en is asserted and the FIFO is not empty, shift the register values one bit to the right. Loop back the last register value to the first. Decode the read vector to correctly imply the read pointer at the address which is enabled by the state of the token ring. The position of the read pointer is defined by the position after the second token, starting from the left.

Similarly, the write pointer is implemented using a token ring. Here, the module conforms to the write clock instead of the read clock. The decoding of the write vector is different due to the different state coding of the write pointer. The position of the write pointer is defined by the position of the register containing the first token. The empty and full conditions are illustrated in Fig. 8.

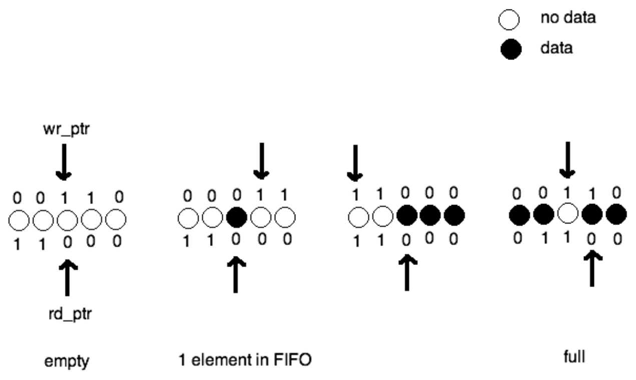

Fig 8: Pointers, tokens and flag conditions

The full and empty detection logic make use of combinational gates to derive the status conditions with the read and write vectors (defined earlier) as inputs. Of course, synchronization is essential as well.

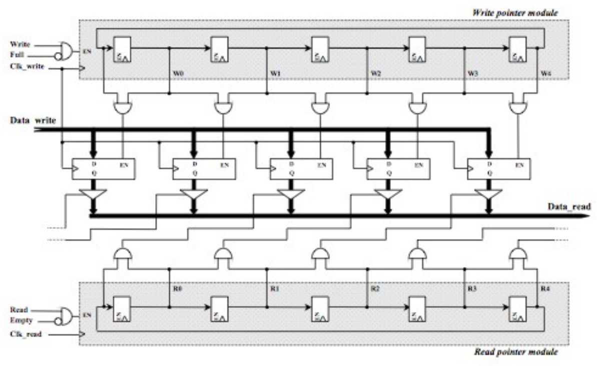

Fig 9: Details of the pointer

The full detector compares each bit of the two vectors to generate a full_flag signal. This is then synchronized to the wr_clk. The code for this module (and the others) is attached to this file. Similarly, the empty detector generates the rd_clk synchronized empty_flag. The implementation of the empty condition is a little more complicated than that of the full condition. This is in contrast to the case in the Gray code pointer FIFO. The difference arises due to the difference in the logic in the two designs.

Figures 10 and 11 show the gate level combinational circuit for the full and empty detectors respectively.

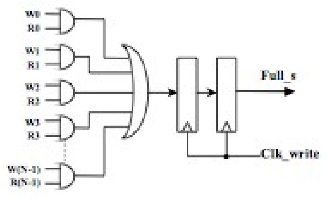

Fig 10: Full detector

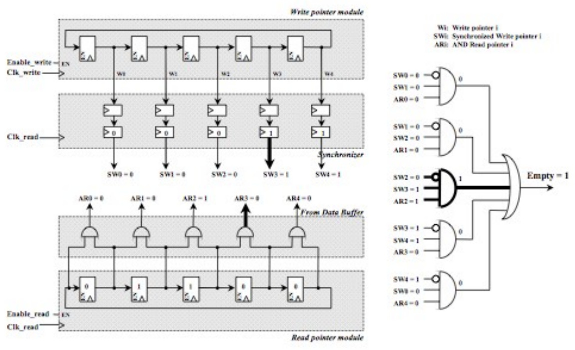

Fig 11: Empty detector

V. Results

The FIFOs were synthesized using Design Compiler with the UofU_Digital_v1_2 library (0.5u). The control circuitry for both the FIFOs showed remarkable similarity, e.g. the slowest component for both FIFOs had a minimum clock period of 5ps. Each module was synthesized using Design Compiler. After that, a high level structural Verilog called fifo.v was written, which defined the connections between modules. Each module was then individually placed and routed using the APR tool SOC Encounter. The tool used the synthesized structural netlist and the Synopsys Design Constraints (.sdc) files as input to place and route the module. After each module had a floorplan and post APR structural Verilog available, the top level module was sent to the SOC Encounter tool. The .lef files for each individual module were also provided as inputs for the placement and routing. After hand placing the modules for the entire fifo design and pin placement, global routing was employed to complete the design.

The generated files have been attached. A comparison of area was done which showed that the Gray code FIFO occupied a significantly larger area as compared to the token ring FIFO. Extensive timing analysis is the next step in this work. Also a FIFO depth of 8 was used. Designs with other depths need to be analyzed as well. Also, there is some research planned for FIFOs with variable data lengths. In this case, the single bit change of the Gray code and token ring arrangements cease to be useful. Performance of these techniques for that case and also an efficient coding scheme that has a Hamming distance of 2 needs to be evaluated.

VI. Conclusion

Two different encoding styles were implemented and evaluated for FIFO pointers. The token ring FIFO was shown to have a significant area advantage. The read and write frequencies for the FIFOs needs to be evaluated further, but is expected to be about the same. Further investigation includes more FIFO designs being synthesized and characterized.

References

[1] I. Panades, A. Greiner, "*Bi-Synchronous FIFO for Synchronous Circuit Communication Well Suited for Network-on-Chip in GALS Architectures*" IEEE Symposium on Networks on Chips, 2007.

[2] C. Cummings, P. Alfke, "*Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons" SNUG,* 2002.

[3] R. Ginosar, "*Fourteen Ways To Fool your Synchronizer" ASYNC 2003.*

[4] T. Chelcea, S. Nowick, *"Low Latency Asynchronous FIFOs using Token Rings", IEEE Transactions on VLSI Systems, 2004.*