Expert **Verilog, SystemVerilog & Synthesis Training**

Simulation and Synthesis Techniques for Asynchronous FIFO Design

Clifford E. Cummings, Sunburst Design, Inc.

cliffc@sunburst-design.com

ABSTRACT

FIFOs are often used to safely pass data from one clock domain to another asynchronous clock domain. Using a FIFO to pass data from one clock domain to another clock domain requires multi-asynchronous clock design techniques. There are many ways to design a FIFO wrong. There are many ways to design a FIFO right but still make it difficult to properly synthesize and analyze the design.

This paper will detail one method that is used to design, synthesize and analyze a safe 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. The fully coded, synthesized and analyzed RTL Verilog model (FIFO Style #1) is included.

Post-SNUG Editorial Comment

A second FIFO paper by the same author was voted "Best Paper - 1st Place" by SNUG attendees, is listed as reference [3] and is also available for download.

1.0 Introduction

An asynchronous FIFO refers to a FIFO design where data values are written to a FIFO buffer from one clock domain and the data values are read from the same FIFO buffer from another clock domain, where the two clock domains are asynchronous to each other.

Asynchronous FIFOs are used to safely pass data from one clock domain to another clock domain.

There are many ways to do asynchronous FIFO design, including many wrong ways. Most incorrectly implemented FIFO designs still function properly 90% of the time. Most almost-correct FIFO designs function properly 99%+ of the time. Unfortunately, FIFOs that work properly 99%+ of the time have design flaws that are usually the most difficult to detect and debug (if you are lucky enough to notice the bug before shipping the product), or the most costly to diagnose and recall (if the bug is not discovered until the product is in the hands of a dissatisfied customer).

This paper discusses one FIFO design style and important details that must be considered when doing asynchronous FIFO design.

The rest of the paper simply refers to an "asynchronous FIFO" as just "FIFO."

2.0 Passing multiple asynchronous signals

Attempting to synchronize multiple changing signals from one clock domain into a new clock domain and insuring that all changing signals are synchronized to the same clock cycle in the new clock domain has been shown to be problematic[1]. FIFOs are used in designs to safely pass multi-bit data words from one clock domain to another. Data words are placed into a FIFO buffer memory array by control signals in one clock domain, and the data words are removed from another port of the same FIFO buffer memory array by control signals from a second clock domain. Conceptually, the task of designing a FIFO with these assumptions seems to be easy.

The difficulty associated with doing FIFO design is related to generating the FIFO pointers and finding a reliable way to determine full and empty status on the FIFO.

2.1 Synchronous FIFO pointers

For synchronous FIFO design (a FIFO where writes to, and reads from the FIFO buffer are conducted in the same clock domain), one implementation counts the number of writes to, and reads from the FIFO buffer to increment (on FIFO write but no read), decrement (on FIFO read but no write) or hold (no writes and reads, or simultaneous write and read operation) 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.

2.2 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. The fact that the read pointer is always pointing to the next FIFO word to be read means that the receiver logic does not have to use two clock periods to read the data word. If the receiver first had to increment the read pointer before reading a FIFO data

word, the receiver would clock once to output the data word from the FIFO, and clock a second time to capture the data word into the receiver. That would be needlessly inefficient.

The FIFO is empty when the read and write pointers are both equal. This condition happens when both pointers are reset to zero during a reset operation, or when the read pointer catches up to the write pointer, having read the last word from the FIFO.

A FIFO is full when the pointers are again equal, that is, when the write pointer has wrapped around and caught up to the read pointer. This is a problem. The FIFO is either empty or full when the pointers are equal, but which?

One design technique used to distinguish between full and empty is to add an extra bit to each pointer. When the write pointer increments past the final FIFO address, the write pointer will increment the unused MSB while setting the rest of the bits back to zero as shown in Figure 1 (the FIFO has wrapped and toggled the pointer MSB). The same is done with the read pointer. If the MSBs of the two pointers are different, it means that the write pointer has wrapped one more time that the read pointer. If the MSBs of the two pointers are the same, it means that both pointers have wrapped the same number of times.
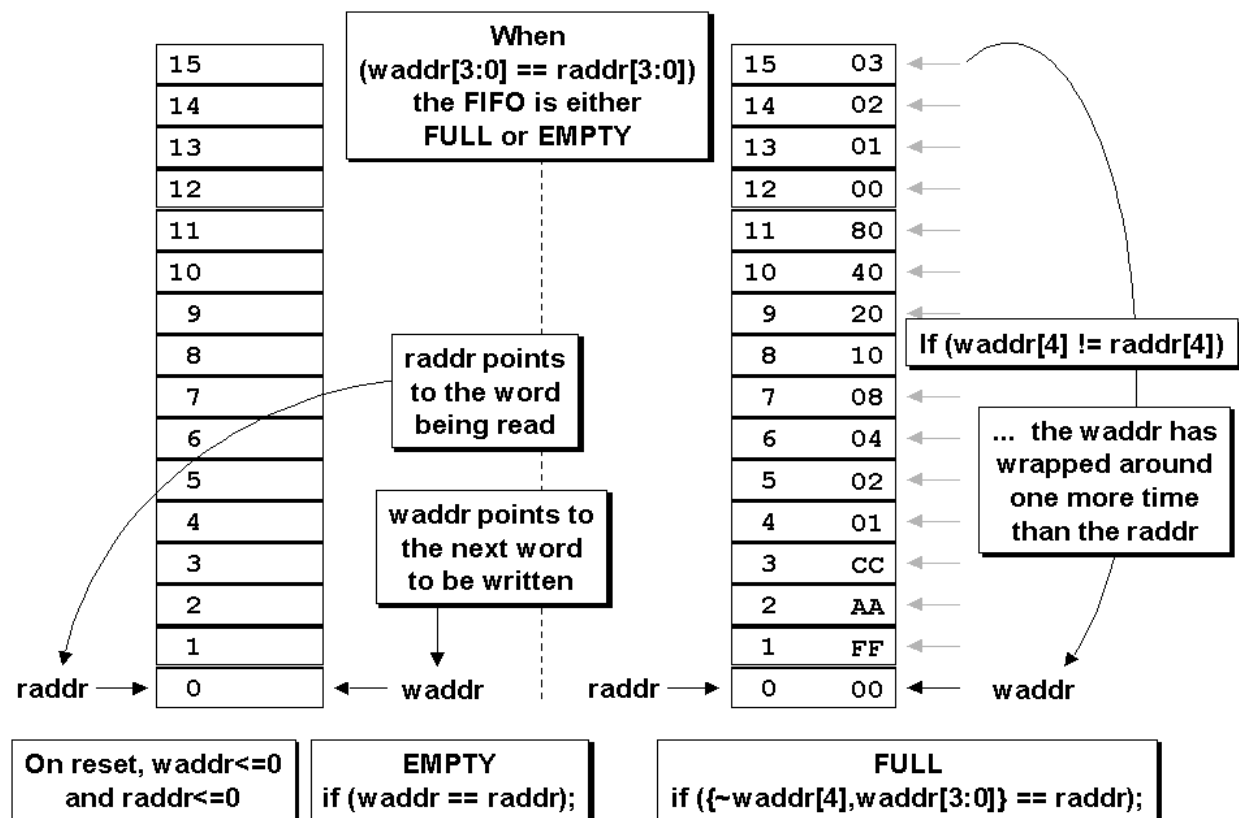

Figure 1 - FIFO full and empty conditions

Using n-bit pointers where (n-1) is the number of address bits required to access the entire FIFO memory buffer, the FIFO is empty when both pointers, including the MSBs are equal. And the FIFO is full when both pointers, except the MSBs are equal.

The FIFO design in this paper uses n-bit pointers for a FIFO with $2^{(n-1)}$ write-able locations to help handle full and empty conditions. More design details related to the full and empty logic are included in section 5.0.

2.3   Binary FIFO pointer considerations

Trying to synchronize a binary count value from one clock domain to another is problematic because every bit of an n-bit counter can change simultaneously (example 7->8 in binary numbers is 0111->1000, all bits changed). One approach to the problem is sample and hold periodic binary count values in a holding register and pass a synchronized ready signal to the new clock domain. When the ready signal is recognized, the receiving clock domain sends back a synchronized acknowledge signal to the sending clock domain. A sampled pointer must not change until an acknowledge signal is received from the receiving clock domain. A count-value with multiple changing bits can be safely transferred to a new clock domain using this technique. Upon receipt of an acknowledge signal, the sending clock domain has permission to clear the ready signal and re-sample the binary count value.

Using this technique, the binary counter values are sampled periodically and not all of the binary counter values can be passed to a new clock domain The question is, do we need to be concerned about the case where a binary counter might continue to increment and overflow or underflow the FIFO between sampled counter values? The answer is no[8].

FIFO full occurs when the write pointer catches up to the synchronized and sampled read pointer. The synchronized and sampled read pointer might not reflect the current value of the actual read pointer but the write pointer will not try to count beyond the synchronized read pointer value. Overflow will not occur[8].

FIFO empty occurs when the read pointer catches up to the synchronized and sampled write pointer. The synchronized and sampled write pointer might not reflect the current value of the actual write pointer but the read pointer will not try to count beyond the synchronized write pointer value. Underflow will not occur[8].More observations about this technique of sampling binary pointers with a synchronized ready-acknowledge pair of handshaking signals are detailed in section 7.0, after the discussion of synchronized Gray[6] code pointers.

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.

2.4   FIFO testing troubles

Testing a FIFO design for subtle design problems is nearly impossible to do. The problem is rooted in the fact that FIFO pointers in an RTL simulation behave ideally, even though, if incorrectly implemented, they can cause catastrophic failures if used in a real design.

In an RTL simulation, if binary-count FIFO pointers are included in the design all of the FIFO pointer bits will change simultaneously; there is no chance to observe synchronization and comparison problems. In a gate-level simulation with no backannotated delays, there is only a slight chance of observing a problem if the gate transitions are different for rising and falling edge signals, and even then, one would have to get lucky and have the correct sequence of bits changing just prior to and just after a rising clock edge. For higher speed designs, the delay differences between rising and falling edge signals diminishes and the probability of detecting problems also diminishes. Finding actual FIFO design problems is greatest for gate-level designs with backannotated delays, but even doing this type of simulation, finding problems will be difficult to do and again the odds of observing the design problems decreases as signal propagation delays diminish.

Clearly the answer is to recognize that there are potential FIFO design problems and to do the design correctly from the start.

The behavioral model that I sometimes use for testing a FIFO design is a FIFO model that is simple to code, is accurate for behavioral testing purposes and would be difficult to debug if it were used as an RTL synthesis model. This FIFO model is only recommended for use in a FIFO testbench. The model accurately determines when FIFO full and empty status bits should be set and can be used to determine the data values that should have been stored into a working FIFO. THIS FIFO MODEL IS NOT SAFE FOR SYNTHESIS!

```
        module beh_fifo (rdata, wfull, rempty, wdata,
                         winc, wclk, wrst_n, rinc, rclk, rrst_n);
```

```verilog
    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;


    reg    [ASIZE:0]   wptr, wrptr1, wrptr2, wrptr3;
    reg    [ASIZE:0]   rptr, rwptr1, rwptr2, rwptr3;


    parameter MEMDEPTH = 1<<ASIZE;

    reg [DSIZE-1:0] ex_mem [0:MEMDEPTH-1];

    always @(posedge wclk or negedge wrst_n)
      if      (!wrst_n)         wptr <= 0;
      else if (winc && !wfull) begin
        ex_mem[wptr[ASIZE-1:0]] <= wdata;
        wptr                    <= wptr+1;
      end

    always @(posedge wclk or negedge wrst_n)
      if (!wrst_n) {wrptr3,wrptr2,wrptr1} <= 0;
      else         {wrptr3,wrptr2,wrptr1} <= {wrptr2,wrptr1,rptr};

    always @(posedge rclk or negedge rrst_n)
      if      (!rrst_n)         rptr    <= 0;
      else if (rinc && !rempty) rptr <= rptr+1;

    always @(posedge rclk or negedge rrst_n)
      if (!rrst_n) {rwptr3,rwptr2,rwptr1} <= 0;
      else         {rwptr3,rwptr2,rwptr1} <= {rwptr2,rwptr1,wptr};

    assign rdata = ex_mem[rptr[ASIZE-1:0]];
    assign rempty = (rptr == rwptr3);
    assign wfull  = ((wptr[ASIZE-1:0] == wrptr3[ASIZE-1:0]) &&
                     (wptr[ASIZE]     != wrptr3[ASIZE]     ));
  endmodule
```

Example 1 - Behavioral FIFO model for testbench use only - SHOULD NOT BE USED FOR SYNTHESIS!

In the behavioral model of Example 1, it is okay to use binary-count pointers, a Verilog array to represent the FIFO memory buffer, multi-asynchronous clocks in the same module and non-registered outputs. THIS MODEL IS NOT INTENDED FOR SYNTHESIS! (Hopefully enough capital letters have been used in this section to discourage anyone from trying to synthesize this model!)

Two of the always blocks in the module (the always blocks with concatenations) are included to behaviorally represent the synchronization that will be required in the actual RTL FIFO design. They are not important to the testing of the data transfer through the FIFO, but they are important to the testing of the correctly timed full and empty flags in the FIFO model. The exact number of synchronization stages required in the behavioral model is FIFO-design dependent. This model can be used to help test the FIFO design described in this paper.

3.0   Gray code counter - Style #1

Gray codes are named for the person who originally patented the code back in 1953, Frank Gray[6]. There are multiple ways to design a Gray code counter. This section details a simple and straight forward method to do the design. The technique described in this paper uses just one set of flip-flops for the Gray code counter. A second method that uses two sets of flip-flops to achieve higher speeds is detailed in shown in section 4.0.

3.1   Gray code patterns

For reasons that will be described later, it is desirable to create both an n-bit Gray code counter and an (n-1)-bit Gray code counter. It would certainly be easy to create the two counters separately, but it is also easy and efficient to create a common n-bit Gray code counter and then modify the 2nd MSB to form an (n-1)-bit Gray code counter with shared LSBs. In this paper, this will be called a "dual n-bit Gray code counter."
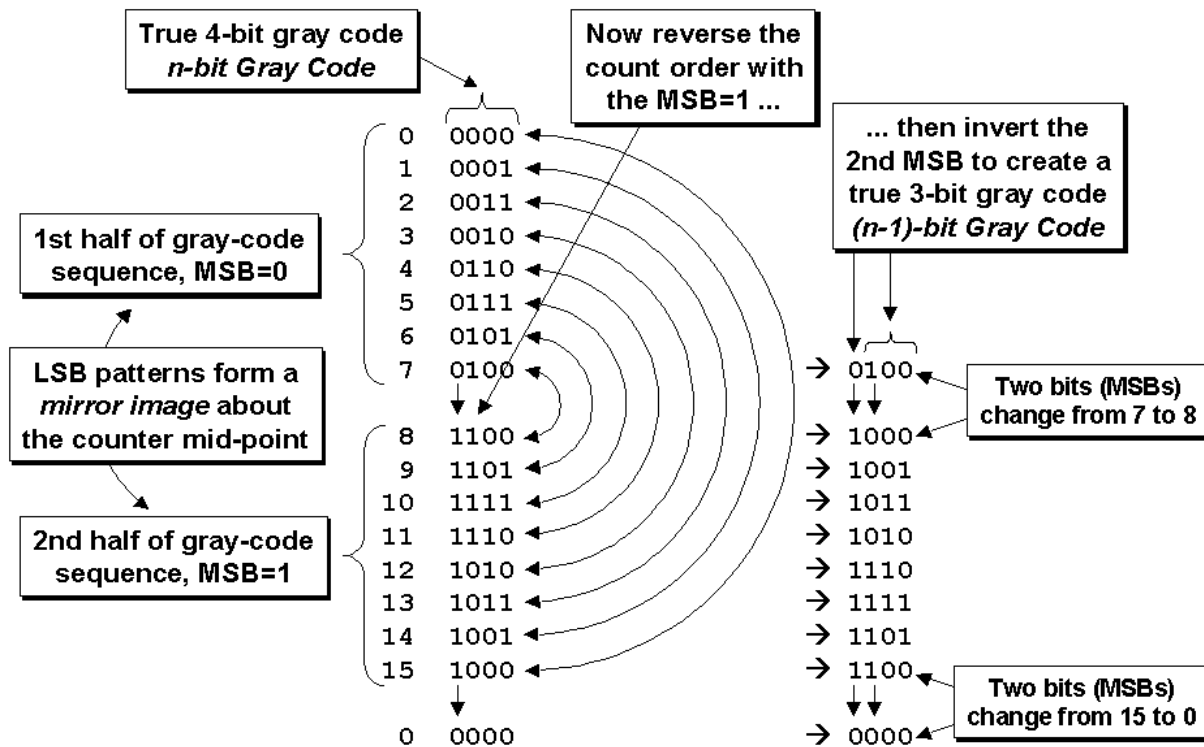

Figure 2 - n-bit Gray code converted to an (n-1)-bit Gray code

To better understand the problem of converting an n-bit Gray code to an (n-1)-bit Gray code, consider the example of creating a dual 4-bit and 3-bit Gray code counter as shown in Figure 2.

The most common Gray code, as shown in Figure 2, is a reflected code where the bits in any column except the MSB are symmetrical about the sequence mid-point[6]. This means that the second half of the 4-bit Gray code is a mirror image of the first half with the MSB inverted.

To convert a 4-bit to a 3-bit Gray code, we do not want the LSBs of the second half of the 4-bit sequence to be a mirror image of the LSBs of the first half, instead we want the LSBs of the second half to repeat the 4-bit LSB-sequence of the first half.

Upon closer examination, it is obvious that inverting the second MSB of the second half of the 4-bit Gray code will produce the desired 3-bit Gray code sequence in the three LSBs of the 4-bit sequence. The only other problem is that the 3-bit Gray code with extra MSB is no longer a true Gray code because when the sequence changes from 7 (Gray 0100) to 8 (~Gray 1000) and again from 15 (~Gray 1100) to 0 (Gray 0000), two bits are changing instead of just one bit. A true Gray code only changes one bit between counts.

3.2   Gray code counter basics

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 so one cannot make a 23-deep Gray code. This means that the technique described in this paper is used to make a FIFO that is 2^n deep.

Figure 3 is a block diagram for a style #1 dual n-bit Gray code counter. The style #1 Gray code counter assumes that the outputs of the register bits are the Gray code value itself (**ptr**, either **wptr** or **rptr**). The Gray code outputs are then passed to a Gray-to-binary converter (**bin**), which is passed to a conditional binary-value incrementer to generate the next-binary-count-value (**bnext**), which is passed to a binary-to-Gray converter that generates the next-Gray-count-value (**gnext**), which is passed to the register inputs. The top half of the Figure 3 block diagram shows the described logic flow while the bottom half shows logic related to the second Gray code counter as described in the next section.
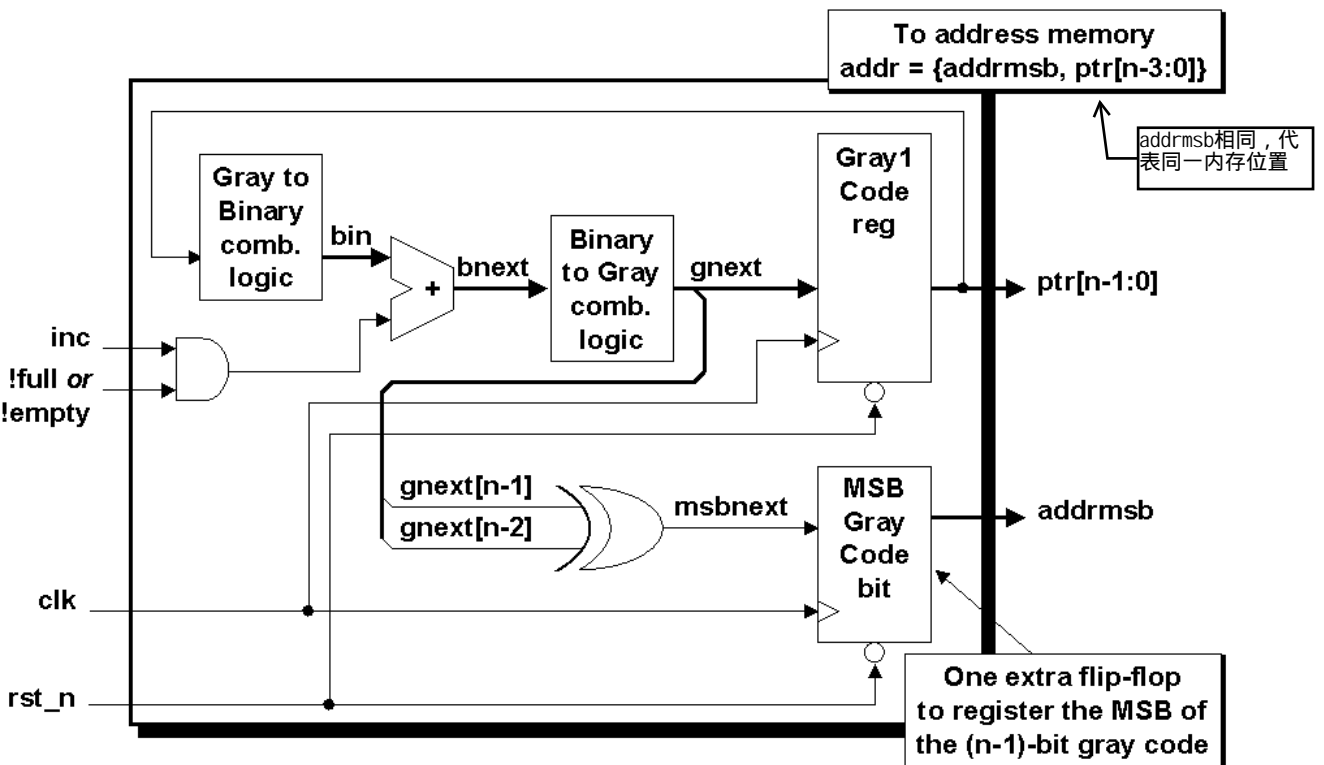

Figure 3 - Dual n-bit Gray code counter block diagram - style #1

3.3   Dual n-bit Gray code counter

A dual n-bit Gray code counter is a Gray code counter that generates both an n-bit Gray code sequence (described in section 3.2) and an (n-1)-bit Gray code sequence.

The (n-1)-bit Gray code is simply generated by doing an exclusive-or operation on the two MSBs of the n-bit Gray code to generate the MSB for the (n-1)-bit Gray code. This is combined with the (n-2) LSBs of the n-bit Gray code counter to form the (n-1)-bit Gray code counter[5].

3.4   Additional Gray code counter considerations

The binary-value incrementer is conditioned with either an "if not full" or "if not empty" test as shown in Figure 3, to insure that the appropriate FIFO pointer will not increment during FIFO-full or FIFO-empty conditions that could lead to overflow or underflow of the FIFO buffer.

If the logic block that sends data to the FIFO reliably stops sending data when a FIFO full condition is asserted, the FIFO design might be streamlined by removing the full-testing logic from the FIFO write pointer.

The FIFO pointer itself does not protect the FIFO buffer from being overwritten, but additional conditioning logic could be added to the FIFO memory buffer to insure that a write_enable signal could not be activated during a FIFO full condition.

An additional "sticky" status bit, either ovf (overflow) or unf (underflow), could be added to the pointer design to indicate that an additional FIFO write operation occurred during full or an additional FIFO read operation occurred during empty to indicate error conditions that could only be cleared during reset.

A safe, general purpose FIFO design will include the above safeguards at the expense of a slightly larger and perhaps slower implementation. This is a good idea since a future co-worker might try to copy and reuse the code in another design without understanding all of the important details that were considered for the current design.

4.0   Gray code counter - Style #2

Starting with version 1.2 of this paper, the FIFO implementation uses the Gray code counter style #2, which actually employs two sets of registers to eliminate the need to translate Gray pointer values to binary values. The second set of registers (the binary registers) can also be used to address the FIFO memory directly without the need to translate memory addresses into Gray codes. The n-bit Gray-code pointer is still required to synchronize the pointers into the opposite clock domains, but the n-1-bit binary pointers can be used to address memory directly. The binary pointers also make it easier to run calculations to generate "almost-full" and "almost-empty" bits if desired (not shown in this paper).
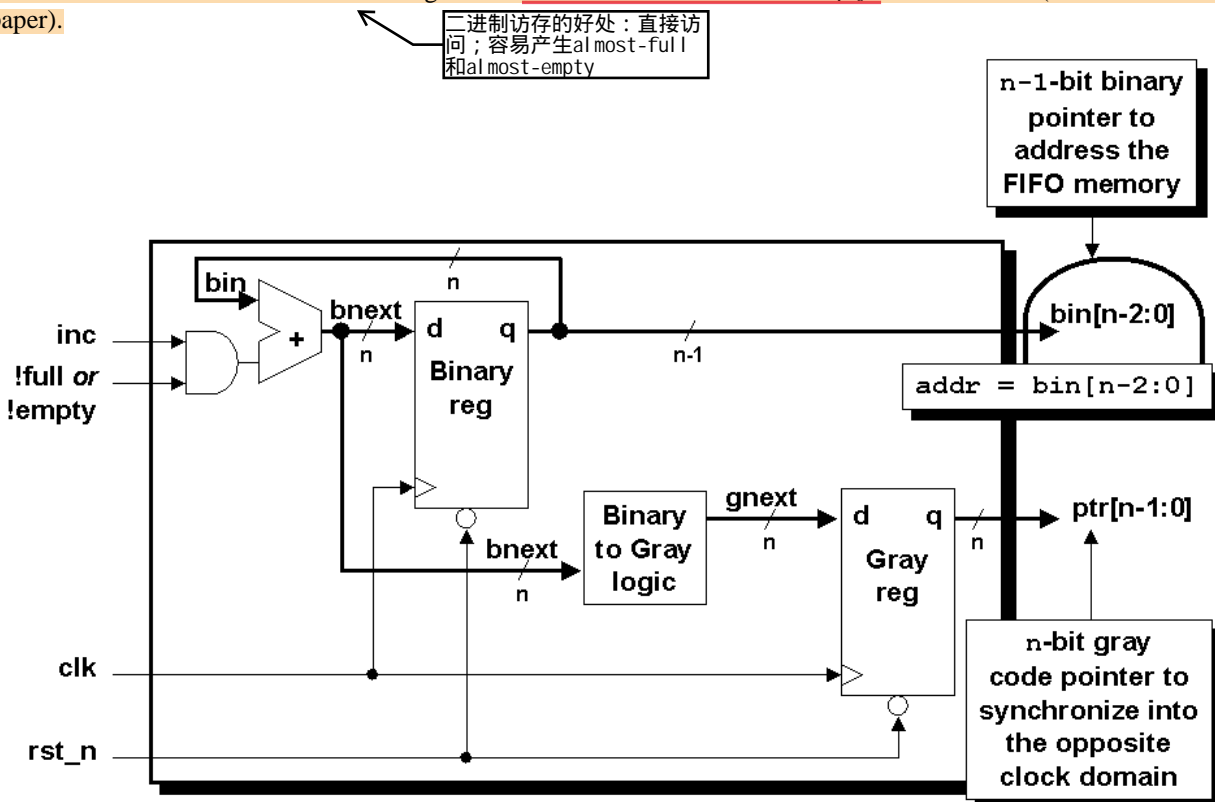

Figure 4 - Dual n-bit Gray code counter block diagram - style #2

FIFO style #1

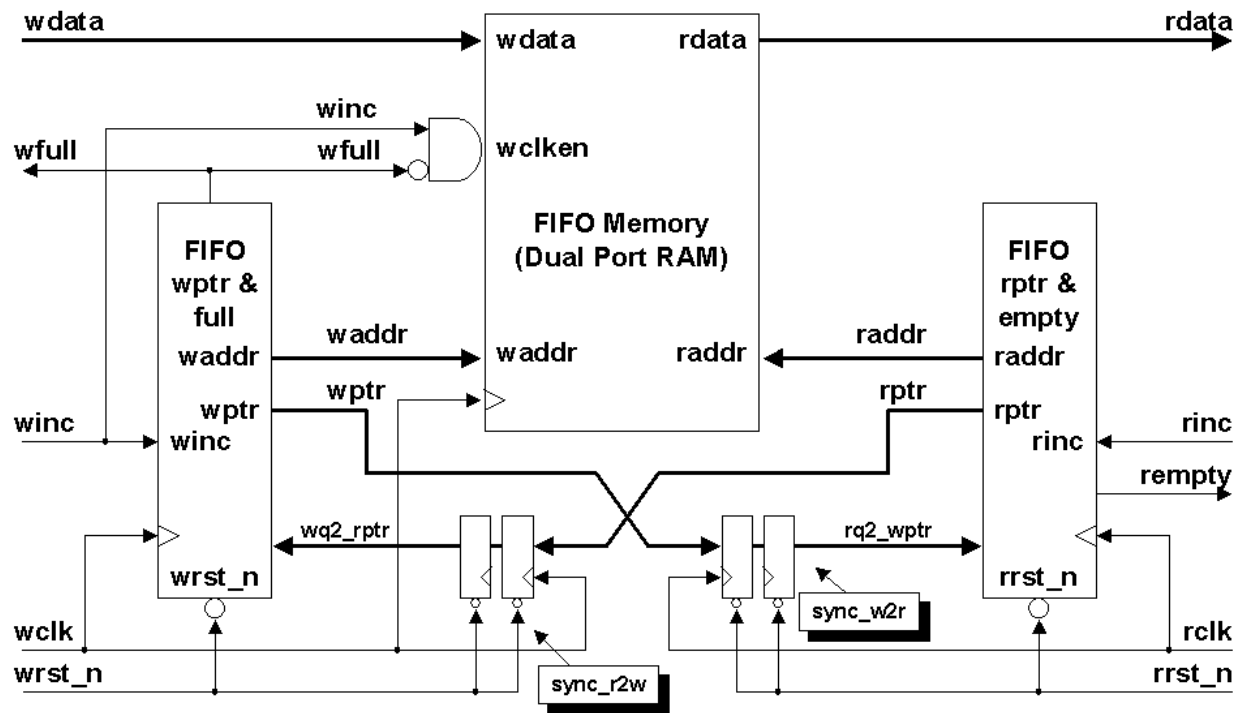The block diagram for FIFO style #1 is shown in Figure 5.

Figure 5 - FIFO1 partitioning with synchronized pointer comparison

To facilitate static timing analysis of the style #1 FIFO design, the design has been partitioned into the following six Verilog modules with the following functionality and clock domains:

- **fifo1.v** - (see Example 2 in section 6.1) - this is the top-level wrapper-module that includes all clock domains. The top module is only used as a wrapper to instantiate all of the other FIFO modules used in the design. If this FIFO is used as part of a larger ASIC or FPGA design, this top-level wrapper would probably be discarded to permit grouping of the other FIFO modules into their respective clock domains for improved synthesis and static timing analysis.

- **fifomem.v** - (see Example 3 in section 6.2) - 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.v** - (see Example 4 in section 6.3) - this is a synchronizer module that is used to synchronize the read pointer into the write-clock domain. The synchronized read pointer will be used by the **wptr_full** module to generate the FIFO full condition. This module only contains flip-flops that are synchronized to the write clock. No other logic is included in this module.

- **sync_w2r.v** - (see Example 5 in section 6.4) - this is a synchronizer module that is used to synchronize the write pointer into the read-clock domain. The synchronized write pointer will be used by the **rptr_empty** module to generate the FIFO empty condition. This module only contains flip-flops that are synchronized to the read clock. No other logic is included in this module.

- **rptr_empty.v** - (see Example 6 in section 6.5) - this module is completely synchronous to the read-clock domain and contains the FIFO read pointer and empty-flag logic.

- **`wptr_full.v`** - (see Example 7 in section 6.6) - this module is completely synchronous to the write-clock domain and contains the FIFO write pointer and full-flag logic.

In order to perform FIFO full and FIFO empty tests using this FIFO style, the read and write pointers must be passed to the opposite clock domain for pointer comparison.

As with other FIFO designs, since the two pointers are generated from two different clock domains, the pointers need to be "safely" passed to the opposite clock domain. The technique shown in this paper is to synchronize Gray code pointers to insure that only one pointer bit can change at a time.

5.0   Handling full & empty conditions

Exactly how FIFO full and FIFO empty are implemented is design-dependent.

The FIFO design in this paper assumes that the empty flag will be generated in the read-clock domain to insure that the empty flag is detected immediately when the FIFO buffer is empty, that is, the instant that the read pointer catches up to the write pointer (including the pointer MSBs).

The FIFO design in this paper assumes that the full flag will be generated in the write-clock domain to insure that the full flag is detected immediately when the FIFO buffer is full, that is, the instant that the write pointer catches up to the read pointer (except for different pointer MSBs).

5.1   Generating empty

As shown in Figure 1, 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 **sync_w2r** module. Since only one bit changes at a time using a Gray code pointer, there is no problem 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**). The empty value testing and the accompanying sequential always block has been extracted from the **rptr_empty.v** code of Example 6 and is shown below:

```
    assign rempty_val = (rgraynext == rq2_wptr);

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

5.2   Generating full

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.
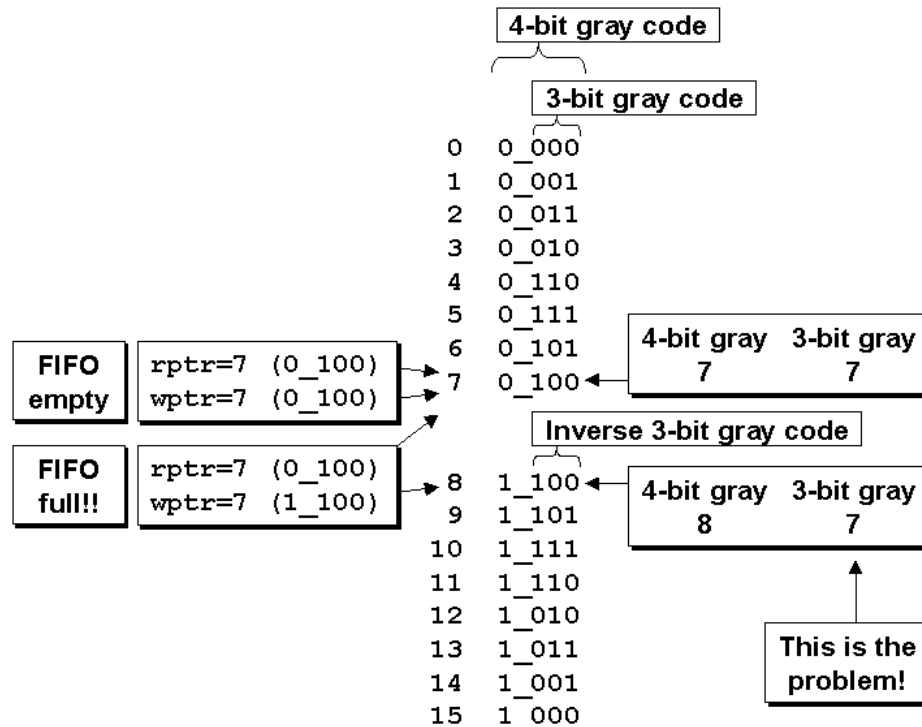
Figure 6 - Problems associated with extracting a 3-bit Gray code from a 4-bit Gray code

Consider the example shown in Figure 6 of an 8-deep FIFO. In this example, a 3-bit Gray code pointer is used to address memory and an extra bit (the MSB of a 4-bit Gray code) is added to test for full and empty conditions. If the FIFO is allowed to fill the first seven locations (words 0-6) and then if the FIFO is emptied by reading back the same seven words, both pointers will be equal and will point to address Gray-7 (the FIFO is empty). On the next write operation, the write pointer will increment the 4-bit Gray code pointer (remember, only the 3 LSBs are being used to address memory), making the MSBs different on the 4-bit pointers but the rest of the write pointer bits will match the read pointer bits, so the FIFO full flag would be asserted. This is wrong! Not only is the FIFO **_not_** full, but the 3 LSBs did not change, which means that the addressed memory location will over-write the last FIFO memory location that was written. This too is wrong!

This is one reason why the dual n-bit Gray code counter of Figure 4 and Section 4.0 is used.

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 - see Figure 6 above).

(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`). This is shown below in the sequential always block that has been extracted from the `wptr_full.v` code of Example 7:

```
      assign wfull_val = ((wgnext[ADDRSIZE]     !=wq2_rptr[ADDRSIZE]   ) &&
                          (wgnext[ADDRSIZE-1]  !=wq2_rptr[ADDRSIZE-1]) &&
                          (wgnext[ADDRSIZE-2:0]==wq2_rptr[ADDRSIZE-2:0]));

      always @(posedge wclk or negedge wrst_n)
        if (!wrst_n) wfull  <= 1'b0;
        else         wfull  <= wfull_val;
```

In the above code, the three necessary conditions to check for FIFO-full are tested and the result is assigned to the **wfull_val** signal, which is then registered in the subsequent sequential always block.

The continuous assignment to **wfull_val** can be further simplified using concatenations as shown below:

```
      assign wfull_val = (wgraynext=={~wq2_rptr[ADDRSIZE:ADDRSIZE-1],
                                       wq2_rptr[ADDRSIZE-2:0]});
```

5.3   Different clock speeds

Since asynchronous FIFOs are clocked from two different clock domains, obviously the clocks are running at different speeds. When synchronizing a faster clock into a slower clock domain, there will be some count values that are skipped due to the fact that the faster clock will semi-periodically increment twice between slower clock edges. This raises discussion of the two following questions:

First question. Noting that a synchronized Gray code that increments twice but is only sampled once will show multi-bit changes in the synchronized value, will this cause multi-bit synchronization problems?

The answer is no. Synchronizing multi-bit changes is only a problem when multiple bits are changing near the rising edge of the synchronizing clock. The fact that a Gray code counter could increment twice (or more) between slower synchronization clock edges means that the first Gray code change will occur well before the rising edge of the slower clock and only the second Gray code transition could change near the rising clock edge. There is no multi-bit synchronization problem with Gray code counters.

Second question. Again noting that a faster Gray code counter could increment more than once between the rising edge of a slower clock signal, is it possible that the Gray code counter from the faster clock domain could increment to a full-state and to a full+1-state before full is detected, causing the FIFO to overflow without recognizing that the FIFO was ever full? (This question similarly applies to FIFO empty).

Again, the answer is no using the implementation described in this paper. Consider first the generation of FIFO full. The FIFO goes full when the write pointer catches up to the synchronized read pointer and the FIFO-full state is detected in the write clock domain. If the **wclk**-domain is faster than the **rclk**-domain, the write pointer will eventually catch up to the synchronized read pointer, the FIFO will be full, the **wfull** bit will be set and the FIFO will quit writing until the synchronized read pointer advances again. The write pointer cannot advance past the synchronized read pointer in the **wclk**-domain.

A similar examination of the empty flag shows that the FIFO goes empty when the read pointer catches up to the synchronized write pointer and the FIFO-empty state is detected in the read clock domain. If the **rclk**-domain is faster than the **wclk**-domain, the read pointer will eventually catch up to the synchronized write pointer, the FIFO will be empty, the **rempty** bit will be set and the FIFO will quit reading until the synchronized write pointer advances again. The read pointer cannot advance past the synchronized write pointer in the **rclk**-domain.

Using this implementation, assertion of "full" or "empty" happens exactly when the FIFO goes full or empty. Removal of "full" and "empty" status is pessimistic.

5.4 Pessimistic full & empty

The FIFO described in this paper has implemented full-removal and empty-removal using a "pessimistic" method. That is, "full" and "empty" are both asserted exactly on time but removed late.

Since the write clock is used to generate the FIFO-full status and since FIFO-full occurs when the write pointer catches up to the synchronized read pointer, full-detection is "accurate" and immediate. Removal of "full" status is pessimistic because "full" comparison is being done with a synchronized read pointer. When the read pointer does increment, the FIFO is no longer full, but the full-generation logic will not detect the change until two rising `wclk` edges synchronize the updated `rptr` into the `wclk` domain. This is generally not a problem, since it means that the data-sending hardware is being "held-off" or informed that the FIFO is still full for a couple of extra `wclk` edges. The important detail is to insure that the FIFO does not overflow. Signaling the data-sender to not send more data for a couple of extra `wclk` edges merely gives time for the FIFO to make room to receive more data.

Similarly, since the read clock is used to generate the FIFO-empty status and since FIFO-empty occurs when the read pointer catches up to the synchronized write pointer, empty-detection is "accurate" and immediate. Removal of "empty" status is pessimistic because "empty" comparison is being done with a synchronized write pointer. When the write pointer does increment, the FIFO is no longer empty, but the empty-generation logic will not detect the change until two rising `rclk` edges synchronize the updated `wptr` into the `rclk` domain. This is generally not a problem, since it means that the data-receiving logic is being "held-off" or informed that the FIFO is still empty for a couple of extra `rclk` edges. The important detail is to insure that the FIFO does not underflow. Signaling the data-receiver to stop removing data from the FIFO for a couple of extra `rclk` edges merely gives time for the FIFO to be filled with more data.

5.4.1 "Accurate" setting of full & empty

Note that setting either the full flag or empty flag might not be quite accurate if both pointers are incrementing simultaneously. For example, if the write pointer catches up to the synchronized read pointer, the full flag will be set, but if the read pointer had incremented at the same time as the write pointer, the full flag will have been set early since the FIFO is not really full due to a read operation occurring simultaneous to the "write-to-full" operation, but the read pointer had not yet been synchronized into the write-clock domain. The setting of the full flag was slightly too early and slightly pessimistic. This is not a design problem.

5.5 Multi-bit asynchronous reset

Much attention has been paid to insuring that the FIFO pointers only change one bit at a time. The question is, will there be a problem associated with an asynchronous reset, which generally causes multiple pointer bits to changes simultaneously?

The answer is no. A reset indicates that the FIFO has also been reset and there is no valid data in the FIFO. On assertion of the reset, all of the synchronizing registers, `wclk`-domain logic (including the registered full flag), and `rclk`-domain logic are simultaneously and asynchronously reset. The registered empty flag is also set at the same time. The more important question concerns orderly removal of the reset signals.

Note that the design included in this paper uses different reset signals for the `wclk` and `rclk` domains. The resets used in this design are intended to be asynchronously set and synchronously removed using the techniques describe in Mills and Cummings[2].

Asynchronous reset of the FIFO pointers is not a problem.

5.6 Almost full and almost empty

Many designs require notification of a pending full or empty status with the generation of "almost full" and "almost empty" status bits. There are many ways to implement these two status bits and each implementation is dependent upon the specified design requirements.

Some FIFO designs require programmable FIFO-full and FIFO-empty difference values, such that when the difference between the two pointers is smaller than the programmed difference, the corresponding almost full or almost empty bit is asserted. Other FIFOs may be implemented with a fixed difference to generate almost full or empty. Other FIFOs may be satisfied with almost full and empty being loosely generated when the MSBs of the FIFO pointers are close. And yet other designs might only require knowing when the FIFO is more, or less than half full.

Remembering that the FIFO is full when the **wptr** catches up to the synchronized **rptr**, the almost full condition could be described as the condition when (**wptr**+4) catches up to the synchronized **rptr**. The (**wptr**+4) value could be generated in the Gray code pointer logic shown in
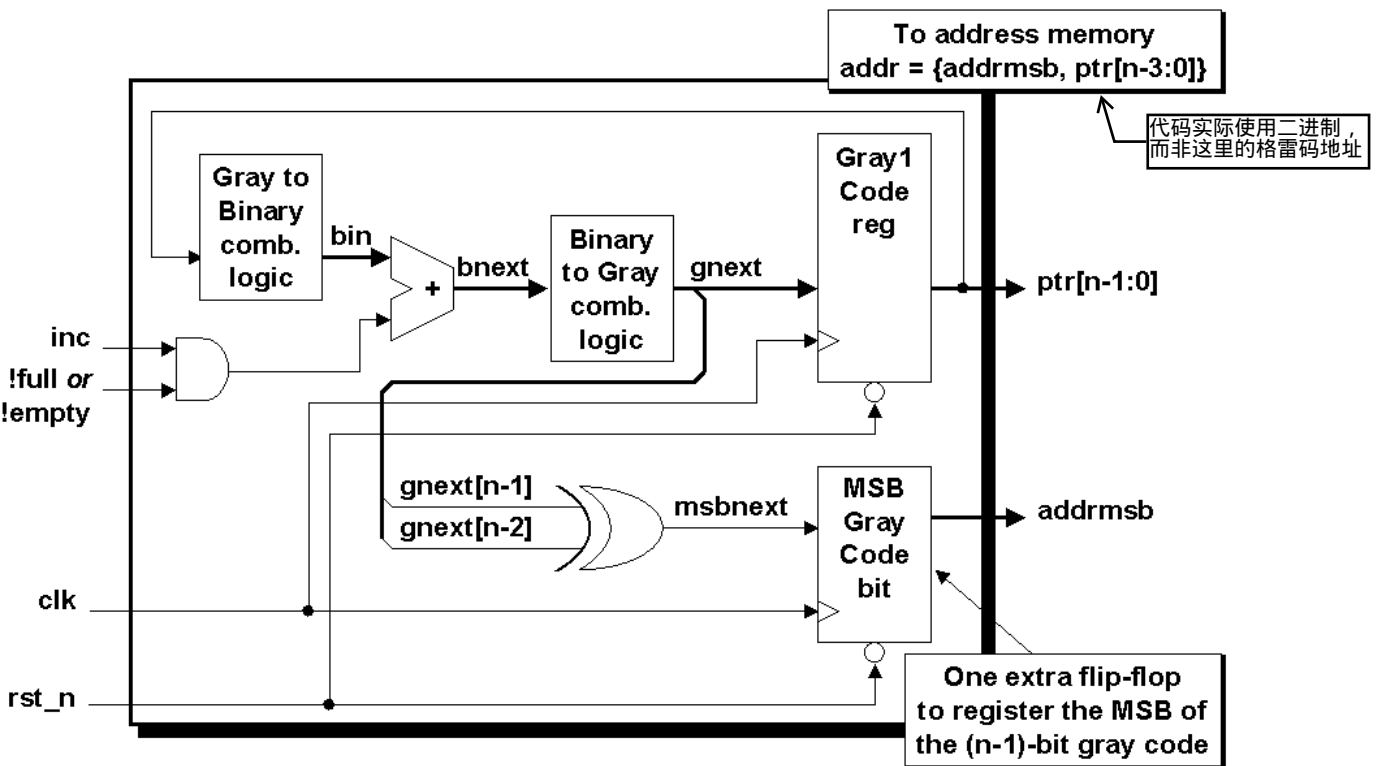

Figure 3 by placing a second adder after the Gray-to-binary combinational logic to add four to the binary value and register the result. This registered value would then be used to do subtraction against the synchronized **rptr** after it has been converted to a binary value in the **wclk** domain, and if the difference is less than four, an **almost_full** bit could be set. A less-than operation insures that the **almost_full** bit is set for the full range when the **wptr** is within 0-4 counts of catching up to the synchronized **rptr**. Similar logic could be used in the **rclk**-domain to generate the **almost_empty** flag.

Almost full and almost empty have not been included in the Verilog RTL code shown in this paper.

6.0 RTL code for FIFO Style #1

The Verilog RTL code for the FIFO style #1 model is listed in this section.

6.1 `fifo1.v` - FIFO top-level module

The top -level FIFO module is a parameterized FIFO design with all sub-blocks instantiated using the recommended practice of doing named port connections. Another common coding practice is to give the top-level module instantiations the same name as the module name. This is done to facilitate debug, since referencing module names in a hierarchical path will be straight forward if the instance names match the module names.

```verilog
module fifo1 #(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      sync_r2w  (.wq2_rptr(wq2_rptr), .rptr(rptr),
                           .wclk(wclk), .wrst_n(wrst_n));

  sync_w2r      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(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
```

Example 2 - Top-level Verilog code for the FIFO style #1 design

6.2 `fifomem.v` - FIFO memory buffer

The FIFO memory buffer is typically an instantiated ASIC or FPGA dual-port, synchronous memory device. The memory buffer could also be synthesized to ASIC or FPGA registers using the RTL code in this module.

About an instantiated vendor RAM versus a Verilog-declared RAM, the Synopsys DesignWare team did internal analysis and found that for sizes up to 256 bits, there is no lost area or performance using the Verilog-declared RAM compared to an instantiated vendor RAM[4].

If a vendor RAM is instantiated, it is highly recommended that the instantiation be done using named port connections.

```
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);

  `ifdef VENDORRAM
    // instantiation of a vendor's dual-port RAM
    vendor_ram mem (.dout(rdata), .din(wdata),
                    .waddr(waddr), .raddr(raddr),
                    .wclken(wclken),
                    .wclken_n(wfull), .clk(wclk));
  `else
    // 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;
  `endif
endmodule
```

Example 3 - Verilog RTL code for the FIFO buffer memory array

6.3 `sync_r2w.v` - 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. Notice the simplicity of the always block that concatenates the two registers together for reset and shifting. The synchronizer always block is only three lines of code.

All module outputs are registered for simplified synthesis using time budgeting. All outputs of this module are entirely synchronous to the **wclk** and all asynchronous inputs to this module are from the **rclk** domain with all signals named using an "**r**" prefix, making it easy to set a false path on all "**r***" signals for simplified static timing analysis.

```
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
```

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

6.4 `sync_w2r.v` - 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. Notice the simplicity of the always block that concatenates the two registers together for reset and shifting. The synchronizer always block is only three lines of code.

All module outputs are registered for simplified synthesis using time budgeting. All outputs of this module are entirely synchronous to the **rclk** and all asynchronous inputs to this module are from the **wclk** domain with all signals named using an "**w**" prefix, making it easy to set a false path on all "**w***" signals for simplified static timing analysis.

```
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
```

Example 5 - Verilog RTL code for the write-clock domain to read-clock domain synchronizer module

6.5 `rptr_empty.v` - 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.

```
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 (okay to use binary to address memory)
  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 == rq2_wptr);

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

Example 6 - Verilog RTL code for the read pointer and empty flag logic

6.6 `wptr_full.v` - 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.

```verilog
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 (okay to use binary to address memory)
  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=((wgnext[ADDRSIZE]    !=wq2_rptr[ADDRSIZE]  ) &&
  //                   (wgnext[ADDRSIZE-1]  !=wq2_rptr[ADDRSIZE-1]) &&
  //                   (wgnext[ADDRSIZE-2:0]==wq2_rptr[ADDRSIZE-2:0]));
  //------------------------------------------------------------------
  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
```

Example 7 - Verilog RTL code for the write pointer and full flag logic

7.0   Comparing Gray code pointers to binary pointers

As mentioned in section 2.3, binary pointers can be used to do FIFO design if the pointers are sampled and handshaking control signals are used between the two clock domains to safely pass the sampled binary count values.

Some advantages of using binary pointers over Gray code pointers:

- The technique of sampling a multi-bit value into a holding register and using synchronized handshaking control signals to pass the multi-bit value into a new clock domain can be used for passing ANY arbitrary multi-bit value across clock domains. This technique can be used to pass FIFO pointers or any multi-bit value.

- Each synchronized Gray code pointer requires 2n flip-flops (2 per pointer bit). The sampled multi-bit register requires 2n+4 flip-flops (1 per holding register bit in each clock domain, 2 flip-flops to synchronize a ready bit and 2 flip-flops to synchronize an acknowledge bit). There is no appreciable difference in the chance that either pointer style would experience metastability.

- The sampled multi-bit binary register allows arbitrary pointer changes. Gray code pointers can only increment and decrement.

- The sampled multi-bit register technique permits arbitrary FIFO depths; whereas, a Gray code pointer requires power-of-2 FIFO depths. If a design required a FIFO depth of at least 132 words, using a standard Gray code pointer would employ a FIFO depth of 256 words. Since most instantiated dual-port RAM blocks are power-of-2 words deep, this may not be an issue.

- Using binary pointers makes it easy to calculate "almost-empty" and "almost-full" status bits using simple binary arithmetic between the pointer values.

One small disadvantage to using binary pointers over Gray code pointers is:

- Sampling and holding a binary FIFO pointer and then handshaking it across a clock boundary can delay the capture of new samples by at least two clock edges from the receiving clock domain and another two clock edges from the sending clock domain. This latency is generally not a problem but it will typically add more pessimism to the assertion of full and empty and might require additional FIFO depth to compensate for the added pessimism. Since most FIFOs are typically specified with excess depth, it is not likely that extra registers or a larger dual-port FIFO buffer size would be required.

The above comparison is worthy of consideration when selecting a method to implement a FIFO design.

8.0   Conclusions

Asynchronous FIFO design requires careful attention to details from pointer generation techniques to full and empty generation. Ignorance of important details will generally result in a design that is easily verified but is also wrong. Finding FIFO design errors typically requires simulation of a gate-level FIFO design with backannotation of actual delays and a whole lot of luck!

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 synchronize and n-bit pointer into the opposite clock domain and to use an (n-1)-bit pointer to do "full" comparison. Synchronizing binary FIFO pointers using techniques described in section 7.0 is another worthy technique to use when doing FIFO design.

Generating the FIFO-empty status is easily accomplished by comparing-equal the n-bit read pointer to the synchronized n-bit write pointer.

The techniques described in this paper should work with asynchronous clocks spanning small to large differences in speed.

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.

9.0 DesignWare FIFOs

It should be mentioned that DesignWare (DW) has a number of FIFO implementations that can be instantiated into a design. It should also be noted that the DW FIFOs have not always been bug-free.

For additional documentation, go to SolvNet and search on "FIFO STAR" and you will find STAR 104287 and STAR 105016 related to the FIFO DW components and the DW_16550 UART. All of these bugs had to do with the DW FIFOs and FIFO sections of the UART. The DesignWare-110.html says that the bugs are fixed in the 1299-3 patch (December 1999).

There are too many ways to do a FIFO design wrong and I consider relying on the DW FIFO components to be absolutely correct without more details on how they were designed to be very risky. Unless I could verify that IP designers followed the important FIFO design guidelines outlined in this paper, I would be inclined to code my own FIFO designs.

10.0 Acknowledgements

I am grateful to Ben Cohen for his willingness to discuss FIFO design issues with me in preparation for writing this paper. I would also like to thank Peter Alfke of Xilinx for also discussing with me alternate interesting approaches to FIFO design.

A special thanks to Steve Golson for doing a great review of the paper on short notice and adding the valuable information, techniques and advantages related to using binary pointers in FIFO design in place of the Gray code pointers. Also for finding the original patent information on Frank Gray's "Pulse Code Communication."

11.0 Additional Post-SNUG Editorial Comments

A second FIFO paper, voted "Best Paper - 1st Place" by SNUG attendees, is listed as reference [3] and is also available for download.

Many of the techniques used in the second FIFO paper[3] can also be used in the FIFO1 design. In particular, the "dual n-bit counter" of the FIFO1 design can be replaced with the quadrant detection logic described in the second FIFO paper. The FIFO1 Gray code counter style #1 can also be replaced with the faster Gray code counter style #2 described in the second FIFO paper.

12.0 Errata and Changes

Readers are encouraged to send email to Cliff Cummings (cliffc@sunburst-design.com) any time they find potential mistakes or if they would like to suggest improvements. Cliff is always interested in other techniques that engineers are using.

12.1 Revision 1.1 (2002) - What Changed?

Version 1.1 was the first release version of this paper on the sunburst-design.com web page and included the Post-SNUG Editorial Comments.

12.2 Revision 1.2 (June 2005) - What Changed?

Full flag detection - the first version of this paper sent the full flag back to the sending logic, which meant that the sending logic had to use the full flag to generate the **winc** signal (used to enable memory writes) using combinational logic. The updated version of this FIFO design shows that the full signal is also sent to the FIFO memory to help determine if the memory should be written. This modification allows the full signal in the FIFO design and the **winc** signal from the sending logic to both be registered, which is a good design and synthesis coding practice, plus it simplifies the sending logic required to generate the **winc** signal. The updated block diagram can be seen in Figure 5.

Full flag testing - the full flag testing and generation as described in section 5.2 has been simplified. The simplification came from the realization that the 2nd MSB did not have to be generated using an exclusive-or operation, but that the inverse 2nd MSB bits could be tested directly and the three conditions that are necessary to detect full are easily described and implemented.

Errata - A colleague, Mario da Costa, pointed out that the combinational sensitivity lists contained in both the **wptr_full** and **rptr_empty** logic listings were missing the **wfull** and **rempty** signals respectively. Mario was correct, but the pointers in this version of the paper have been replaced with Gray code counter style #2 pointers, which also fixes the problems present in earlier versions of this paper.

Binary pointers to address memory - binary pointers can be safely used to address the FIFO memory buffer and since the binary values are readily available in the **wptr_full** and **rptr_empty** logic blocks, they have been used to address memory without conversion to Gray codes.

Naming convention - the naming convention for the pointers synchronized into the opposite clock domains was somewhat confusing to many of my Advanced Verilog Students, so I changes the naming convention to make the intent more clear. For example, the **wptr** synchronized into the **rclk** domain was **rwptr**, but the new name reflects the double synchronization more clearly by using the name **rq2_wptr**, etc.

Verilog-2001 coding style - many examples were updated with more efficient Verilog-2001 coding styles.

Errata - A colleague, Zenja Chao, pointed out labeling errors in Figure 3. The figure referenced **ptr[n:0]** and called it an **n**-bit pointer. The appropriate labels have been corrected. Zenja also found a typo in the second to last paragraph of section 5.3. When the FIFO is empty, the FIFO will quit **reading** (instead of writing). Zenja also found a typo in version 1.1 of this paper in the third paragraph after Figure 5 (**rwptr2** should be **wrptr2**). This code was fixed by replacing the full-testing functionality, which now does not require inversion of the second MSB of either pointer.

References

[1] Clifford E. Cummings, "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs," *SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001) User Papers*, March 2001, Section MC1, 3rd paper. Also available at www.sunburst-design.com/papers

[2] Clifford E. Cummings and Don Mills, "Synchronous Resets? Asynchronous Resets? I am so confused! How will I ever know which to use?," *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, March 2002, Section TB2, 1st paper. Also available at www.sunburst-design.com/papers

[3] Clifford E. Cummings and Peter Alfke, "Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons," *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, March 2002, Section TB2, 3rd paper. Also available at www.sunburst-design.com/papers

[4] Dinesh Tyagi, former CAE Manager for Synopsys DesignWare product, personal communication

[5] Edward Paluch, personal communication

[6] Frank Gray, "Pulse Code Communication." United States Patent Number 2,632,058. March 17, 1953.

[7] John O'Malley, *Introduction to the Digital Computer*, Holt, Rinehart and Winston, Inc., 1972, pg. 190.

[8] Steve Golson, personal communication

[9] Synopsys SolvNet, Doc Name: DesignWare-110.html, "Functional Bugs in DesignWare Components," Updated: 11/30/2000

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 23 years of ASIC, FPGA and system design experience and 13 years of Verilog, SystemVerilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, is the only Verilog and SystemVerilog trainer to co-develop and co-author every IEEE 1364 Verilog Standard, the IEEE 1364.1 Verilog RTL Synthesis Standard, every Accellera SystemVerilog Standard, and the IEEE 1800 SystemVerilog Standard.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers Verilog, Verilog Synthesis and SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.

Email address: cliffc@sunburst-design.com

An updated version of this paper can be downloaded from the web site: www.sunburst-design.com/papers

 (Last updated June 16th, 2005)