Expert **Verilog, SystemVerilog & Synthesis Training**

Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons

SNUG-2002
San Jose, CA
Voted Best Paper
1st Place

Clifford E. Cummings
Sunburst Design, Inc.

Peter Alfke
Xilinx, Inc.

ABSTRACT

An interesting technique for doing FIFO design is to perform asynchronous comparisons between the FIFO write and read pointers that are generated in clock domains that are asynchronous to each other. The asynchronous FIFO pointer comparison technique uses fewer synchronization flip-flops to build the FIFO. The asynchronous FIFO comparison method requires additional techniques to correctly synthesize and analyze the design, which are detailed in this paper.

To increase the speed of the FIFO, this design uses combined binary/Gray counters that take advantage of the built-in binary ripple carry logic.

The fully coded, synthesized and analyzed RTL Verilog model (FIFO Style #2) is included.

This FIFO design paper builds on information already presented in another FIFO design paper where the FIFO pointers are synchronized into the opposite clock domain before running "FIFO full" or "FIFO empty" tests. The reader may benefit from first reviewing the FIFO Style #1 method before proceeding to this FIFO Style #2 method.

Post-SNUG Editorial Comment (by Cliff Cummings)

Although this paper was voted "Best Paper - 1st Place" by SNUG attendees, this paper builds off of a second FIFO paper listed as reference [1]. The first FIFO paper laid the foundation for some of the content of this paper; therefore, it is highly recommended that readers download and read the FIFO1 paper[1] to acquire background information already assumed to be known by the reader of this paper.

1.0 Introduction

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

One common technique for designing an asynchronous FIFO is to use Gray[4] code pointers that are synchronized into the opposite clock domain before generating synchronous FIFO full or empty status signals[1]. An interesting and different approach to FIFO full and empty generation is to do an asynchronous comparison of the pointers and then asynchronously set the full or empty status bits[6].

This paper discusses the FIFO design style with asynchronous pointer comparison and asynchronous full and empty generation. Important details relating to this style of asynchronous FIFO design are included. The FIFO style implemented in this paper uses efficient Gray code counters, whose implementation is described in the next section.

2.0 Gray code counter - style #2

One Gray code counter style uses a single set of flip-flops as the Gray code register with accompanying Gray-to-binary conversion, binary increment, and binary-to-Gray conversion[1].

A second Gray code counter style, the one described in this paper, uses two sets of registers, one a binary counter and a second to capture a binary-to-Gray converted value. The intent of this Gray code counter style #2 is to 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.

Figure 1 shows the block diagram for an n-bit Gray code counter (style #2).

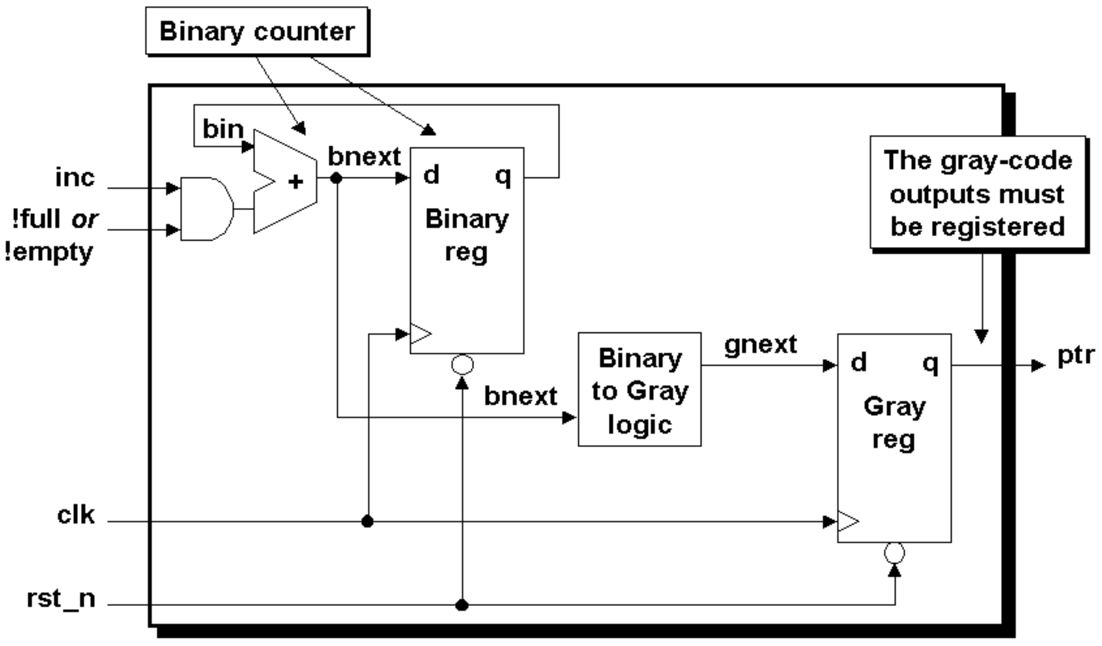


Figure 1 - Dual n-bit Gray code counter style #2

This implementation requires twice the number of flip-flops, but reduces the combinatorial logic and can operate at a higher frequency. In FPGA designs, availability of extra flip-flops is rarely a problem since FPGAs typically contain far more flip-flops than any design will ever use. In FPGA designs, reducing the amount of combinational logic frequently translates into significant improvements in speed.

The **`ptr`** output of the block diagram in Figure 1 is an n-bit Gray code pointer.

Note: since the MSB of a binary sequence is equal to the MSB of a Gray code sequence, this design can be further simplified by using the binary MSB-flip-flop as the Gray code MSB-flip-flop. The Verilog code in this paper did not implement this additional optimization. This would save one flip-flop per pointer.

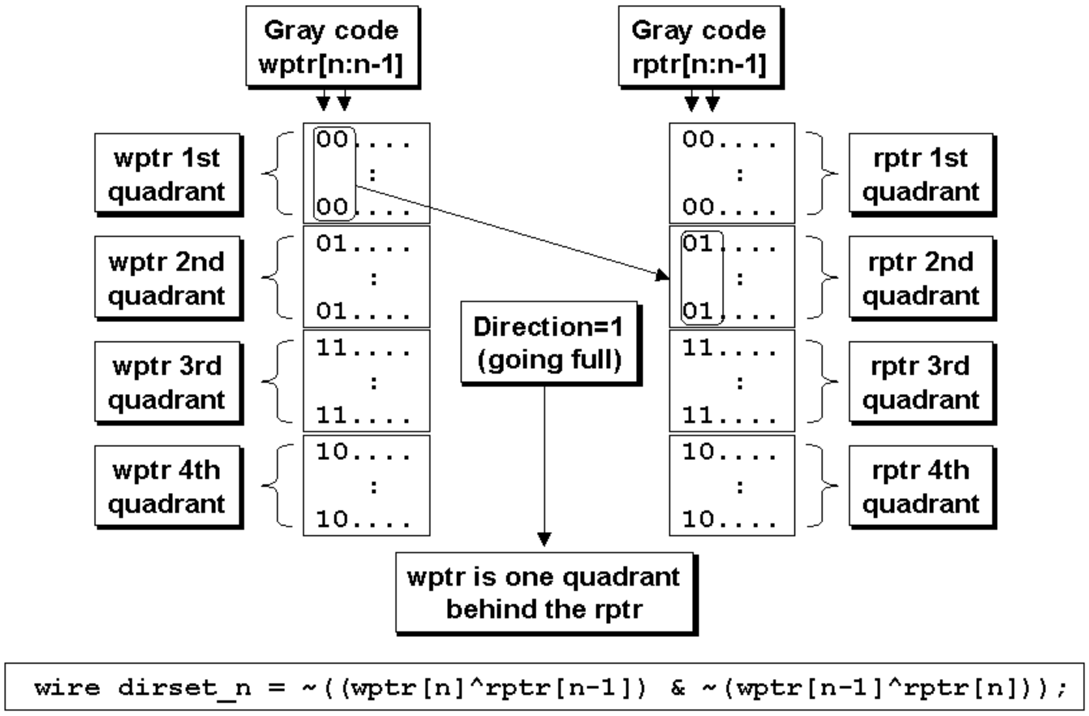
3.0 Full & empty detection

As with any FIFO design, correct implementation of full and empty is the most difficult part of the design.

There are two problems with the generation of full and empty:

First, both full and empty are indicated by the fact that the read and write pointers are identical. Therefore, something else has to distinguish between full and empty. One known solution to this problem appends an extra bit to both pointers and then compares the extra bit for equality (for FIFO empty) or inequality (for FIFO full), along with equality of the other read and write pointer bits[1].

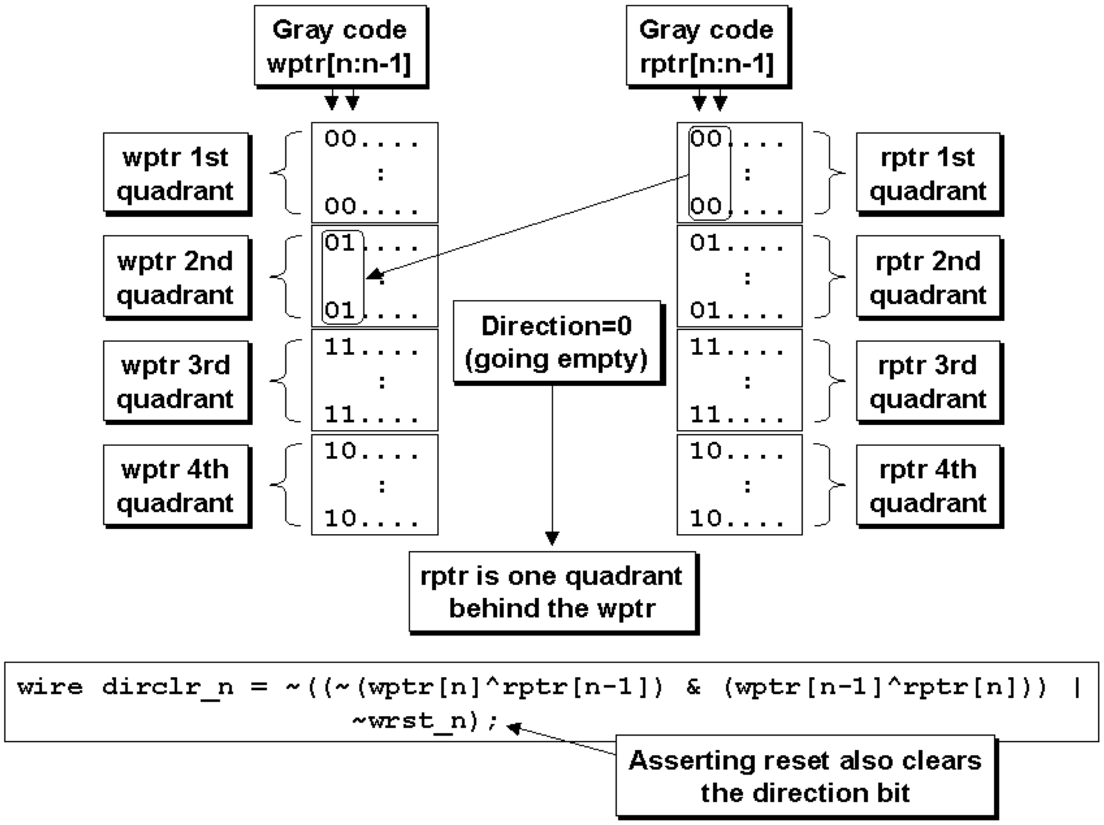
Another solution, the one described in this paper, divides the address space into four quadrants and decodes the two MSBs of the two counters to determine whether the FIFO was going full or going empty at the time the two pointers became equal.



```
wire dirset_n = ~((wptr[n]^rptr[n-1]) & ~(wptr[n-1]^rptr[n]));
```

Figure 2 - FIFO is going full because the **`wptr`** trails the **`rptr`** by one quadrant

If the write pointer is one quadrant behind the read pointer, this indicates a "possibly going full" situation as shown in Figure 2. When this condition occurs, the **`direction`** latch of Figure 4 is set.



```
wire dirclr_n = ~((~(wptr[n]^rptr[n-1]) & (wptr[n-1]^rptr[n])) |
                  ~wrst_n);
```

Figure 3 - FIFO is going empty because the **`rptr`** trails the **`wptr`** by one quadrant

If the write pointer is one quadrant ahead of the read pointer, this indicates a "possibly going empty" situation as shown in Figure 3. When this condition occurs, the **`direction`** latch of Figure 4 is cleared.

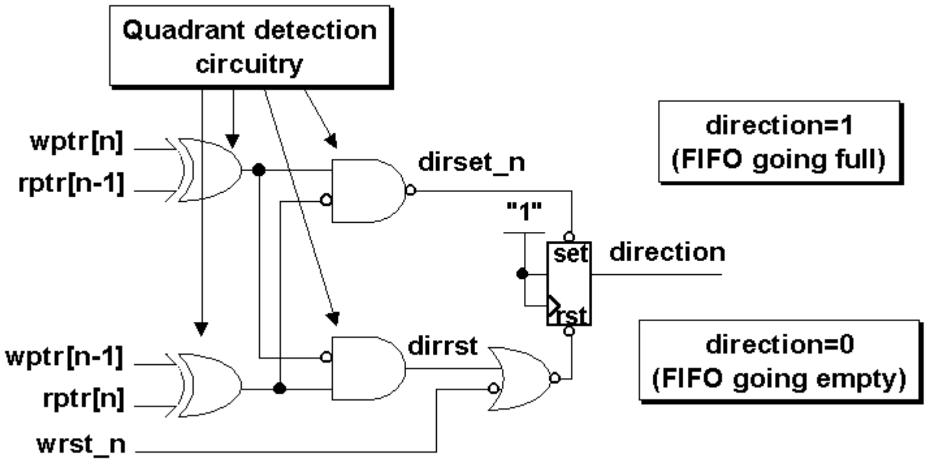

Quadrant detection circuitry
wptr[n]
rptr[n-1]
dirset_n
direction=1 (FIFO going full)
"1"
set
direction
rst
wptr[n-1]
rptr[n]
wrst_n
dirrst
direction=0 (FIFO going empty)

Figure 4 - FIFO direction quadrant detection circuitry

When the FIFO is reset the **`direction`** latch is also cleared to indicate that the FIFO "is going empty" (actually, it *is* empty when both pointers are reset). Setting and resetting the **`direction`** latch is not timing-critical, and the direction latch eliminates the ambiguity of the address identity decoder.

The Xilinx FPGA logic to implement the decoding of the two **`wptr`** MSBs and the two **`rptr`** MSBs is easily implemented as two 4-input look-up tables.

The second, and more difficult, problem stems from the asynchronous nature of the write and read clocks. Comparing two counters that are clocked asynchronously can lead to unreliable decoding spikes when either or both counters change multiple bits more or less simultaneously. The solution described in this paper uses a Gray count sequence, where only one bit changes from any count to the next. Any decoder or comparator will then switch only from one valid output to the next one, with no danger of spurious decoding glitches.

4.0 FIFO style #2

For the purposes of this paper, FIFO style #1 refers to a FIFO implementation style that synchronizes pointers from one clock domain to another before generating full and empty flags [1].

The FIFO style described in this paper (FIFO style #2) does asynchronous comparison between Gray code pointers to generate an asynchronous control signal to set and reset the full and empty flip-flops.

The block diagram for FIFO style #2 is shown in Figure 5.

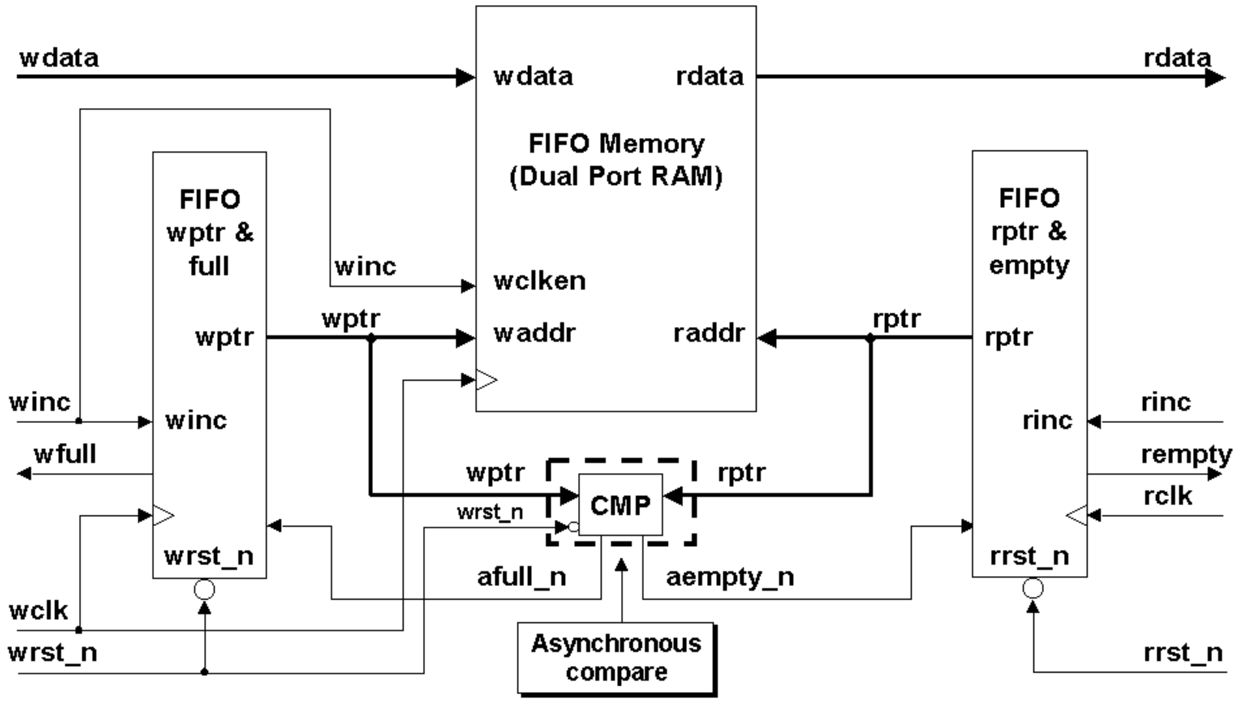


Figure 5 - FIFO2 partitioning with asynchronous pointer comparison logic

To facilitate static timing analysis of the style #2 FIFO design, the design has been partitioned into the following five Verilog modules with the following functionality and clock domains:

- **`fifo2.v`** - (see Example 1 in section 5.1) - 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 2 in section 5.2) - 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.
- **async_cmp.v** - (see Example 3 in section 5.3) - this is an asynchronous pointer-comparison module that is used to generate signals that control assertion of the asynchronous "full" and "empty" status bits. This module only contains combinational comparison logic. No sequential logic is included in this module.
- **rptr_empty.v** - (see Example 4 in section 5.4) - this module is mostly synchronous to the read-clock domain and contains the FIFO read pointer and empty-flag logic. Assertion of the **aempty_n** signal (an input to this module) is synchronous to the **rclk**-domain, since **aempty_n** can only be asserted when the **rptr** incremented, but de-assertion of the **aempty_n** signal happens when the **wptr** increments, which is asynchronous to **rclk**.
- **wptr_full.v** - (see Example 5 in section 5.5) - this module is mostly synchronous to the write-clock domain and contains the FIFO write pointer and full-flag logic. Assertion of the **afull_n** signal (an input to this module) is synchronous to the **wclk**-domain, since **afull_n** can only be asserted when the **wptr** incremented (and **wrst_n**), but de-assertion of the **afull_n** signal happens when the **rptr** increments, which is asynchronous to **wclk**.

5.0 RTL code for FIFO style #2

The Verilog RTL code for the FIFO style #2 model is listed in this section.

5.1 `fifo2.v` - FIFO top-level module

The fifo2 top-level module is a parameterized module with all sub-blocks instantiated following safe coding practices using named port connections.

```
module fifo2 (rdata, wfull, rempty, wdata,
            winc, wclk, wrst_n, rinc, rclk, rrst_n);
  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] wptr, rptr;
  wire   [ASIZE-1:0] waddr, raddr;

  async_cmp  #(ASIZE)     async_cmp
                          (.aempty_n(aempty_n), .afull_n(afull_n),
                           .wptr(wptr), .rptr(rptr), .wrst_n(wrst_n));

  fifomem #(DSIZE, ASIZE) fifomem
                          (.rdata(rdata), .wdata(wdata),
                           .waddr(wptr), .raddr(rptr),
                           .wclken(winc), .wclk(wclk));

  rptr_empty #(ASIZE)     rptr_empty
                          (.rempty(rempty), .rptr(rptr),
                           .aempty_n(aempty_n), .rinc(rinc),
                           .rclk(rclk), .rrst_n(rrst_n));

  wptr_full  #(ASIZE)     wptr_full
                          (.wfull(wfull), .wptr(wptr),
                           .afull_n(afull_n), .winc(winc),
                           .wclk(wclk), .wrst_n(wrst_n));
endmodule
```

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

5.2 `fifomem.v` - FIFO memory buffer

The FIFO memory buffer could be 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.

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

```
module fifomem (rdata, wdata, waddr, raddr, wclken, wclk);
  parameter DATASIZE = 8;        // Memory data word width
  parameter ADDRSIZE = 4;        // Number of memory address bits
  parameter DEPTH = 1<<ADDRSIZE; // DEPTH = 2**ADDRSIZE
  output [DATASIZE-1:0] rdata;
  input  [DATASIZE-1:0] wdata;
  input  [ADDRSIZE-1:0] waddr, raddr;
  input                 wclken, wclk;

  `ifdef VENDORRAM
    // instantiation of a vendor's dual-port RAM
    VENDOR_RAM MEM (.dout(rdata), .din(wdata),
                    .waddr(waddr), .raddr(raddr),
                    .wclken(wclken), .clk(wclk));
  `else
    reg [DATASIZE-1:0] MEM [0:DEPTH-1];

    assign rdata = MEM[raddr];

    always @(posedge wclk)
      if (wclken) MEM[waddr] <= wdata;
  `endif
endmodule
```

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

5.3 `async_cmp.v` - Asynchronous the full/empty comparison logic

The logic used to determine the full or empty status on the FIFO is the most distinctive difference between FIFO style #1 and FIFO style #2.

Async_cmp is an asynchronous comparison module, used to compare the read and write pointers to detect full and empty conditions.

```
module async_cmp (aempty_n, afull_n, wptr, rptr, wrst_n);
  parameter ADDRSIZE = 4;
  parameter N = ADDRSIZE-1;
  output        aempty_n, afull_n;
  input  [N:0] wptr, rptr;
  input         wrst_n;

  reg           direction;
  wire          high = 1'b1;

  wire dirset_n = ~(  (wptr[N]^rptr[N-1]) & ~(wptr[N-1]^rptr[N]));
  wire dirclr_n = ~((~(wptr[N]^rptr[N-1]) &  (wptr[N-1]^rptr[N])) |
                     ~wrst_n);

  always @(posedge high or negedge dirset_n or negedge dirclr_n)
    if      (!dirclr_n) direction <= 1'b0;
    else if (!dirset_n) direction <= 1'b1;
    else                direction <= high;

  //always @(negedge dirset_n or negedge dirclr_n)
    //if      (!dirclr_n) direction <= 1'b0;
    //else                direction <= 1'b1;

  assign aempty_n = ~((wptr == rptr) && !direction);
  assign afull_n  = ~((wptr == rptr) &&  direction);
endmodule
```

Example 3 - Verilog RTL code for the asynchronous comparator module

Three of the last seven lines of the Verilog code of Example 3 have been commented out in this model. In theory, a synthesis tool should be capable of inferring an RS-flip-flop from the comment-removed code, but the LSI_10K library that is included with the default installation of the Synopsys tools did not infer a correct RS-flip-flop with this code when tested, so the always block immediately preceding the commented code was added to infer an RS-flip-flop.

5.3.1 Asynchronous generation of full and empty

In the **`async_cmp`** code of Example 3, and shown in Figure 6, **`aempty_n`** and **`afull_n`** are the asynchronously decoded signals. The **`aempty_n`** signal is asserted on the rising edge of an **`rclk`**, but is de-asserted on the rising edge of a **`wclk`**. Similarly, the **`afull_n`** signal is asserted on a **`wclk`** and removed on an **`rclk`**.

The empty signal will be used to stop the next read operation, and the leading edge of **`aempty_n`** is properly synchronous with the read clock, but the trailing edge needs to be synchronized to the read clock. This is done in a two-stage synchronizer that generates **`rempty`**.

The **`wfull`** signal is generated in the symmetrically equivalent way.

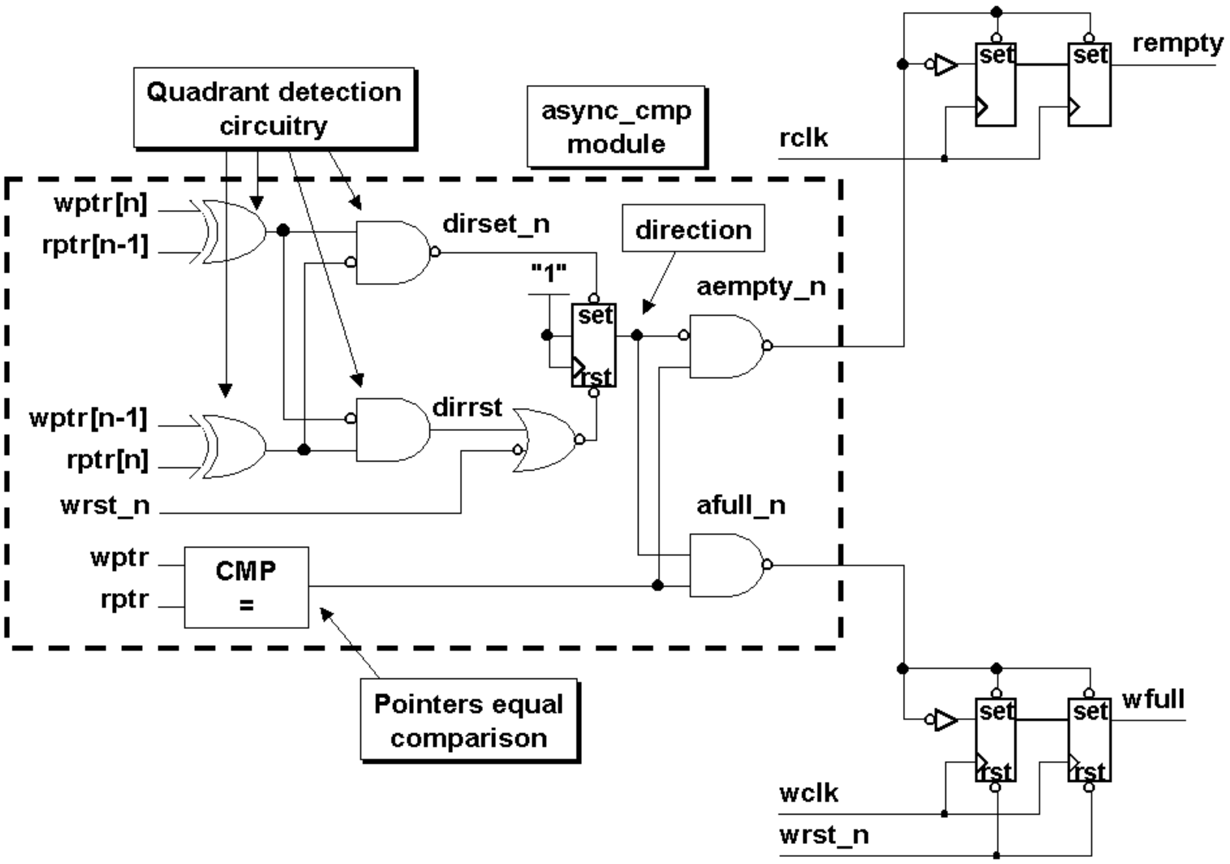


Figure 6 - Asynchronous pointer comparison to assert full and empty

5.3.2 Resetting the FIFO

The first FIFO event of interest takes place on a FIFO-reset operation. When the FIFO is reset, four important things happen within the **`async_cmp`** module and accompanying full and empty synchronizers of the **`wptr_full`** and **`rptr_empty`** modules (the connections between the **`async_cmp`**, **`wptr_full`** and **`rptr_empty`** modules are shown in Figure 7):

1. The reset signal directly clears the **`wfull`** flag. The **`rempty`** flag is not cleared by a reset.
2. The reset signal clears both FIFO pointers, so the pointer comparator asserts that the pointers are equal.
3. The reset clears the **`direction`** bit.
4. With the pointers equal and the **`direction`** bit cleared, the **`aempty_n`** bit is asserted, which presets the **`rempty`** flag.

5.3.3 FIFO-writes & FIFO full

The second FIFO operational event of interest takes place when a FIFO-write operation takes place and the **`wptr`** is incremented. At this point, the FIFO pointers are no longer equal so the **`aempty_n`** signal is de-asserted, releasing the preset control of the **`rempty`** flip-flops. After two rising edges on **`rclk`**, the FIFO will de-assert the **`rempty`** signal. Because the de-assertion of **`aempty_n`** happens on a rising **`wclk`** and because the **`rempty`** signal is clocked by the **`rclk`**, the two-flip-flop synchronizer as shown in Figure 8 is required to remove metastability that could be generated by the first **`rempty`** flip-flop.

The second FIFO operational event of interest takes place when the **`wptr`** increments into the next Gray code quadrant beyond the **`rptr`** (see section 3.0 for a discussion of Gray code quadrants). The **`direction`** bit is cleared (but it was already clear).

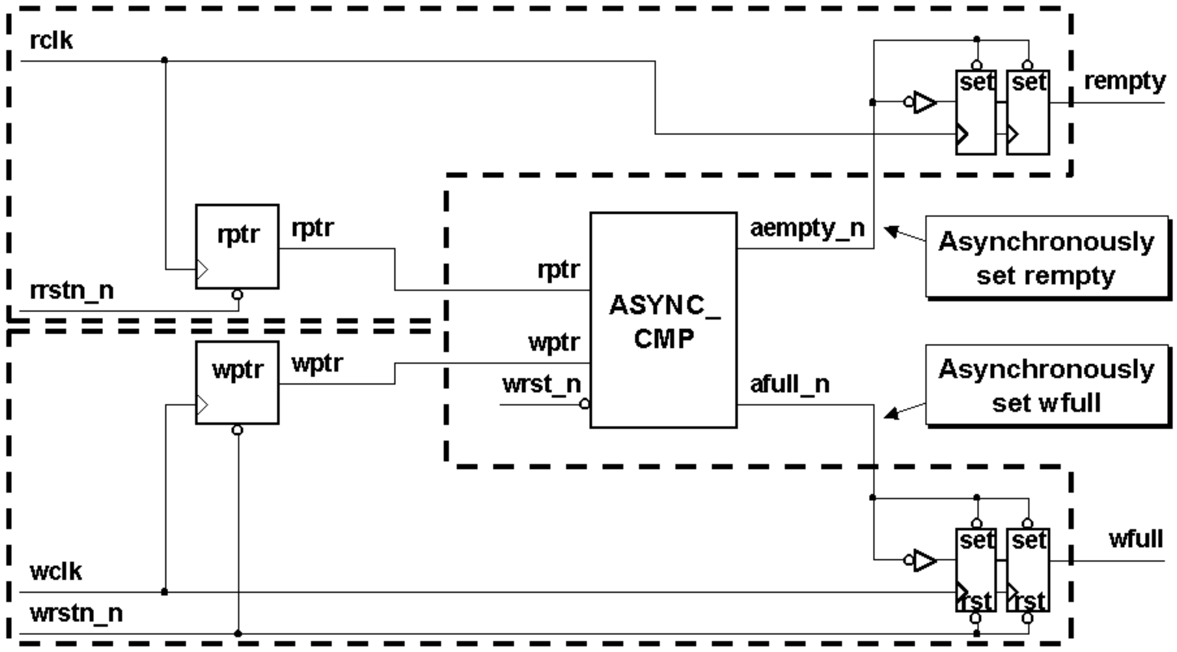


Figure 7 - **`async_cmp`** module connection to **`rptr_empty`** and **`wptr_full`** modules

The third FIFO operational event of interest occurs when the **`wptr`** is within one quadrant of catching up to the **`rptr`** as described in section 3.0. When this happens, the **`dirset_n`** bit of Figure 6 is asserted low, which sets the **`direction`** bit high. This means that the **`direction`** bit is set long before the FIFO is full and is not timing-critical to assertion of the **`afull_n`** signal.

The fourth FIFO operational event of interest is when the **`wptr`** catches up to the **`rptr`** (and the **`direction`** bit is set). When this happens, the **`afull_n`** signal presets the **`wfull`** flip-flops. The **`afull_n`** signal is asserted on a FIFO-write operation and is synchronous to the rising edge of the **`wclk`**; therefore, asserting full is synchronous to the **`wclk`**. See section 5.3.6 for a discussion of the critical timing path associated with assertion of the **`wfull`** signal.

The fifth FIFO operational event of interest is when a FIFO-read operation takes place and the **`rptr`** is incremented. At this point, the FIFO pointers are no longer equal so the **`afull_n`** signal is de-asserted, releasing the preset control of the **`wfull`** flip-flops. After two rising edges on **`wclk`**, the FIFO will de-assert the **`wfull`** signal. Because the de-assertion of **`afull_n`** happens on a rising **`rclk`** and because the **`wfull`** signal is clocked by the **`wclk`**, the two-flip-flop synchronizer, shown in Figure 8, is required to remove metastability that could be generated by the first **`wfull`** flip-flop capturing the inverted and asynchronously generated **`afull_n`** data input.

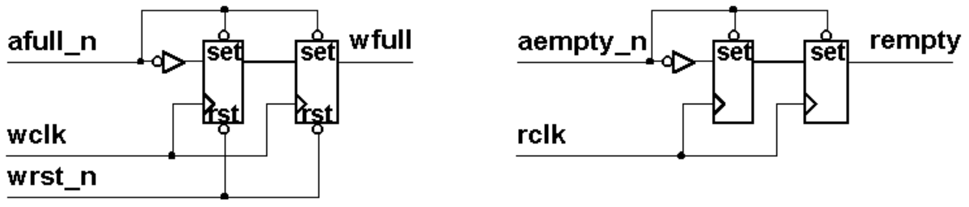


Figure 8 - Asynchronous empty and full generation

During operation, **`wfull`** is generated synchronous to the write clock, in a similar way that **`rempty`** is generated synchronous to the read clock. The **`afull_n`** signal is asserted as a result of a write clock, and the leading (falling) edge is thus naturally synchronous to the write clock. The trailing (rising) edge is, however caused by the read clock, and must, therefore be synchronized to the write clock. The same timing issues related to the setting of the full flag also apply to the setting of the empty flag.

5.3.4 FIFO-reads & FIFO empty

The sixth FIFO operational event of interest takes place when the **rptr** increments into the next Gray code quadrant beyond the **wptr**. The **direction** bit is again set (but it was already set).

The seventh FIFO operational event of interest occurs when the **rptr** is within one quadrant of catching up to the **wptr**. When this happens, the **dirrst** bit of Figure 6 is asserted high , which clears the **direction** bit. This means that the **direction** bit is cleared long before the FIFO is empty and is not timing critical to assertion of the **aempty_n** signal.

The eighth FIFO operational event of interest is when the **rptr** catches up to the **wptr** (and the **direction** bit is zero). When this happens, the **aempty_n** signal presets the **rempty** flip-flops. The **aempty_n** signal is asserted on a FIFO-read operation and is synchronous to the rising edge of the **rclk**; therefore, asserting empty is synchronous to the **rclk**. See section 5.3.6 for a discussion of the critical timing path associated with assertion of the **rempty** signal.

Finally, when a FIFO-write operation takes place and the **wptr** is incremented. At this point, the FIFO pointers are no longer equal so the **aempty_n** signal is de-asserted, releasing the preset control of the **rempty** flip-flops. After two rising edges on **rclk**, the FIFO will de-assert the **rempty** signal. Because the de-assertion of **aempty_n** happens on a rising **wclk** and because the **rempty** signal is clocked by the **rclk**, the two-flip-flop synchronizer as shown in Figure 8 is required to remove metastability that could be generated by the first **rempty** flip-flop.

5.3.5 Alternate method to preset the full & empty flags

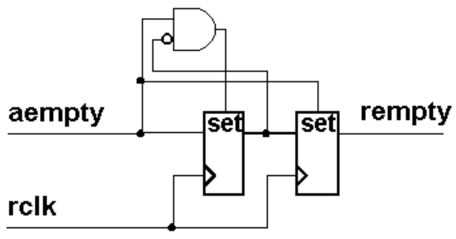


Figure 9 - Self-timed preset assertion circuit

Another method for setting the **rempty** or **wfull** flags is to use a self-timed differentiating circuit as shown in Figure 9. In this figure, the flip-flops are shown with high-true presets, similar to what is found on Xilinx FPGAs. (equivalent circuitry could also be designed using low-true presets). When the **aempty** signal goes high, the **rempty** output flip-flop is preset and assuming that the signal between the flip-flops was low, this signal combined with **aempty**-high will drive the output of the and gate high and set the first flip-flop. When the first flip-flop is set, the and gate will quit driving the preset signal to the first flip-flop. This is a self-timed preset signal that releases preset immediately after preset occurs, well before the **aempty** signal goes low.

5.3.6 Full and empty critical timing paths

Using the asynchronous comparison technique described in this paper, there are critical timing paths associated with the generation of both the **rempty** and **wfull** signals.

The **rempty** critical timing path, shown in Figure 10, consists of (1) the **rclk**-to-q incrementing of the **rptr**, (2) comparison logic of the **rptr** to the **wptr**, (3) combining the comparator output with the direction latch output to generate the **aempty_**n signal, (4) presetting the **rempty** signal, (5) any logic that is driven by the **rempty** signal, and (6) resultant signals meeting the setup time of any down-stream flip-flops clocked within the

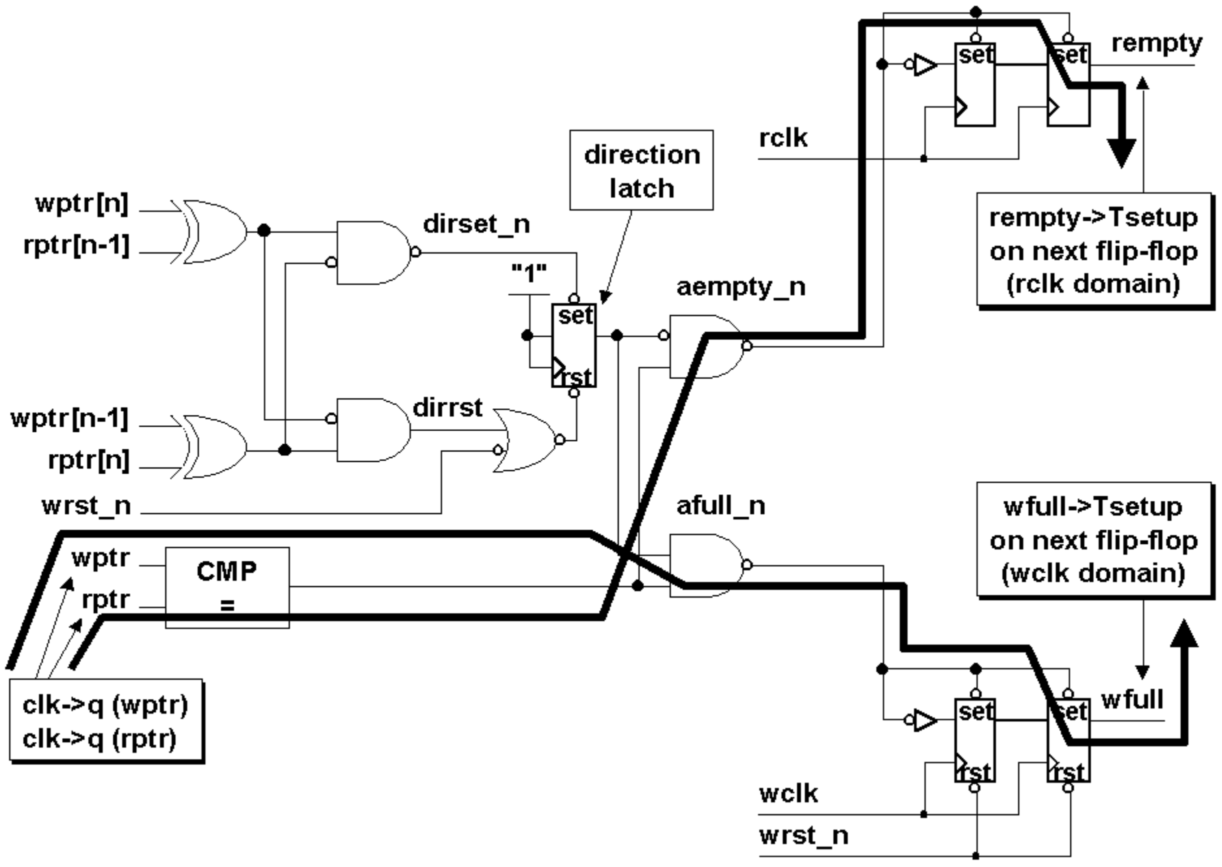


Figure 10 - Critical timing paths for asserting **rempty** and **wfull**

rclk domain. This critical timing path has a symmetrically equivalent critical timing path for the generation of the **wfull** signal, also shown in Figure 10.

5.3.7 Asynchronous concerns, questions and answers

While writing this paper, the authors asked and answered numerous questions to address concerns over the highly asynchronous nature of the generation and removal of the full and empty bits for the FIFO style described in this paper. This section captures a number of the questions, concerns and answers that lead both authors to believe this coding style does indeed work.

Generation of the **aempty_n** control signal is straightforward. Whenever the read pointer (**rptr**) equals the write pointer (**wptr**), and the **direction** latch is clear, the FIFO is empty.

The empty flag is used only in the read clock domain and since the read pointer, incremented by a read clock, causes the empty flag to be set, assertion of the empty flag is always synchronous in the read clock domain. As long as the empty flag meets the critical empty-assertion timing path described in section 5.3.6, there is no synchronization problems associated with asserting the empty flag.

The de-assertion of **aempty_n** is caused by the write clock incrementing the write pointer, and is thus unrelated to the read clock. The de-assertion of **aempty_n** must, therefore, be synchronized in a dual-flip-flop synchronizer, clocked by the read clock. The first flip-flop is subject to metastability but the second flip-flop is included to wait for the metastability to subside, just like any other multi-clock synchronizer[2].

Since **aempty_n** is started by one clock and terminated by the other, it has an undefined duration, and might even be a runt pulse. A runt pulse is a Low-High-Low signal transition where the transition to High may or may not pass through the logic-“1” threshold level of the logic family being used.

If the **aempty_n** control signal is a runt pulse, there are four possible scenarios that should be addressed:

(1) the runt signal is not recognized by the **rempty** flip-flops and empty is not asserted. This is not a problem.

(2) The runt pulse might preset the first synchronizer flip-flop, but not the second flip-flop. This is highly unlikely, but would result in an unnecessary, but properly synchronized **rempty** output, that will show up on the output of the second flip-flop one read clock later. This is not a problem.

(3) The runt pulse might preset the second synchronizer flip-flop, but not the first flip-flop. This is highly unlikely, but would result in an unnecessary, but properly synchronized **rempty** output (as long as the empty critical timing is met), that will be set on the output of the second flip-flop until the next read clock, when it will be cleared by the zero from the first flip-flop. This is not a problem.

(4) The most likely case is that the runt pulse sets both flip-flops, thus creating a properly synchronized rempty output that is two read-clock periods long. The longer duration is caused by the two-flip-flop synchronizer (to avoid metastable problems as described below). This is not a problem.

The runt pulse cannot have any effect on the synchronizer data-input, since an **aempty_n** runt pulse can only occur immediately after a read clock edge, thus long before the next read clock edge (as long as critical timing is met).

The **aempty_n** signal might also stay high longer and go low at any moment, even perhaps coincident with the next read clock edge. If it goes low well before the set-up time of the first synchronize flip-flop, the result is like scenario (4) above. If it goes low well after the set-up time, the synchronizer will stretch **rempty** by one more read clock period.

If **aempty_n** goes low within the metstability-catching set-up time window, the first synchronizer flip-flop output will be indeterminate for a few nanoseconds, but will then be either high or low. In either case, the output of the second synchronizer flip-flop will create the appropriate synchronized **rempty** output.

The next question is, what happens if the write clock de-asserts the **aempty_n** signal coincident with the rising **rclk** on the dual synchronizer? The first flip-flop could go metastable, which is why there is a second flip-flop in the dual synchronizer.

But the removal of the setting signal on the second flip-flop will violate the recovery time of the second flip-flop. Will this cause the second flip-flop to go metastable? The authors do not believe this can happen because the preset to the flip-flop forced the output high and the input to the same flip-flop is already high, which we believe is not subject to a recovery time instability on the flip-flop.

Challenge: if anyone can prove that a flip-flop that is set high, and is also driven by a high-data-input signal, can go metastable if the preset signal is removed coincident with the rising edge of the clock to the same flip-flop, the authors would like to be made aware of any such claim. The authors believe that recovery time parameters are with respect to removing a preset when the data input value is zero. The authors could not find any published reference to discount the possibility of metastability on the output of the second flip-flop but we believe that metastability in this case is not possible.

Last question. Can a runt-preset pulse, where the trailing edge of the runt pulse is caused by the **wclk**, preset the second synchronizer flip-flop in close proximity to a rising **rclk**, violate the preset recovery time and cause metastability on the output of the second flip-flop? The answer is no as long as the **aempty_n** critical timing path is met. Assuming that critical timing is met, the **aempty_n** signal going low should occur shortly after a rising **rclk** and well before the rising edge of the second flip-flop, so runt pulses can only occur well before the rising edge of an **rclk**.

Again, symmetrically equivalent scenarios and arguments can be made about the generation of the **wfull** flag.

5.4 `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 an n-bit Gray code counter. The FIFO **`rempty`** output is asserted when the **`aempty_n`** signal goes low and the **`rempty`** output is de-asserted on the second rising **`rclk`** edge after **`aempty_n`** goes high (a rare metastable state could cause the **`rempty`** output to be de-asserted on the third rising **`rclk`** edge). This module is completely synchronous to the **`rclk`** for simplified static timing analysis, except for the **`aempty_n`** input, which is de-asserted asynchronously to the **`rclk`**.

```
module rptr_empty (rempty, rptr, aempty_n, rinc, rclk, rrst_n);
  parameter ADDRSIZE = 4;
  output                rempty;
  output [ADDRSIZE-1:0] rptr;
  input                 aempty_n;
  input                 rinc, rclk, rrst_n;
  reg    [ADDRSIZE-1:0] rptr, rbin;
  reg                   rempty, rempty2;
  wire   [ADDRSIZE-1:0] rgnext, rbnext;

  //---------------------------------------------------------------
  // GRAYSTYLE2 pointer
  //---------------------------------------------------------------
  always @(posedge rclk or negedge rrst_n)
    if (!rrst_n) begin
      rbin     <= 0;
      rptr     <= 0;
    end
    else begin
      rbin     <= rbnext;
      rptr     <= rgnext;
    end

  //---------------------------------------------------------------
  // increment the binary count if not empty
  //---------------------------------------------------------------
  assign rbnext = !rempty ? rbin + rinc : rbin;
  assign rgnext = (rbnext>>1) ^ rbnext; // binary-to-gray conversion

  always @(posedge rclk or negedge aempty_n)
    if (!aempty_n) {rempty,rempty2} <= 2'b11;
    else           {rempty,rempty2} <= {rempty2,~aempty_n};
endmodule
```

Example 4 - Verilog RTL code for the read pointer and empty flag logic

The last always block in this module is the asynchronously preset **`rempty`** signal generation. The presetting signal is the **`aempty_n`** input , which is asserted when the **`rptr`** is incremented by the **`rclk`** (synchronous to this block) as long as the **`rempty`** critical timing path (described in section 5.3.6) is satisfied. Removal of the **`rempty`** signal occurs when the write pointer increments, which is asynchronous to the **`rclk`** domain. Because reset removal is asynchronous to the **`rclk`** domain, a two-flip-flop synchoronizer is required to synchronize **`aempty_n`** removal to the **`rclk`** domain.

5.5 `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 an n-bit Gray code counter. The FIFO **`wfull`** output is asserted when the **`afull_n`** signal goes low and the **`wfull`** output is de-asserted on the second rising **`wclk`** edge after **`afull_n`** goes high (a rare metastable state could cause the **`wfull`** output to be de-asserted on the third rising **`wclk`** edge). This module is completely synchronous to the **`wclk`** for simplified static timing analysis, except for the **`afull_n`** input, which is de-asserted asynchronously to the **`wclk`**.

```
module wptr_full (wfull, wptr, afull_n, winc, wclk, wrst_n);
  parameter ADDRSIZE = 4;
  output                wfull;
  output [ADDRSIZE-1:0] wptr;
  input                 afull_n;
  input                 winc, wclk, wrst_n;
  reg    [ADDRSIZE-1:0] wptr, wbin;
  reg                   wfull, wfull2;
  wire   [ADDRSIZE-1:0] wgnext, wbnext;

  //---------------------------------------------------------------
  // GRAYSTYLE2 pointer
  //---------------------------------------------------------------
  always @(posedge wclk or negedge wrst_n)
    if (!wrst_n) begin
      wbin     <= 0;
      wptr     <= 0;
    end
    else begin
      wbin     <= wbnext;
      wptr     <= wgnext;
    end

  //---------------------------------------------------------------
  // increment the binary count if not full
  //---------------------------------------------------------------
  assign wbnext = !wfull ? wbin + winc : wbin;
  assign wgnext = (wbnext>>1) ^ wbnext; // binary-to-gray conversion

  always @(posedge wclk or negedge wrst_n or negedge afull_n)
    if      (!wrst_n ) {wfull,wfull2} <= 2'b00;
    else if (!afull_n) {wfull,wfull2} <= 2'b11;
    else               {wfull,wfull2} <= {wfull2,~afull_n};
endmodule
```

Example 5 - Verilog RTL code for the write pointer and full flag logic

The last always block in this module is the asynchronously preset **`wfull`** signal generation. The presetting signal is the **`afull_n`** input , which is asserted when the **`wptr`** is incremented by the **`wclk`** (synchronous to this block) as long as the **`wfull`** critical timing path (described in section 5.3.6) is satisfied. Removal of the **`wfull`** signal occurs when the read pointer increments, which is asynchronous to the **`wclk`** domain. Because reset removal is asynchronous to the **`wclk`** domain, a two-flip-flop synchoronizer is required to synchronize **`afull_n`** removal to the **`wclk`** domain. The **`wfull`** signal must also go low when the FIFO is reset.

6.0 Conclusion

This paper describes an efficient technique to implement a high-speed asynchronous FIFO, using dual-port RAMs addressed by Gray counters This design uses an asynchronous comparator for detecting full and empty status.

The technique described implements an asynchronous assertion of the full and empty flags that requires more effort to analyze for static timing verification.

The technique described also does not have registered full and empty status flags, so care must be taken to insure that the generation of these flags meets the required timing to recognize assertion of full and empty in the rest of the system.

This efficient and interesting approach to FIFO design worthy of consideration.

7.0 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.

7.1 Revision 1.1 (2002) - What Changed?

Additional Post-SNUG Editorial Comments (by Cliff Cummings) - Although this paper was voted "Best Paper - 1st Place" by SNUG attendees, this paper describes a FIFO design style that is generally incompatible with static timing analysis (STA) and design-for-test (DFT). Readers should also reference the FIFO design style described by reference [1] if a more STA-friendly and DFT-friendly style is desired.

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

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.

7.2 Revision 1.2 (June 2005) - What Changed?

There were some minor formatting changes to this paper, along with additional changes noted below.

Full flag detection - this paper sends the full flag back to the sending logic, which means that the sending logic has to use the full flag to generate the **`winc`** signal (used to enable memory writes) using combinational logic. An updated version of this FIFO style #1 design[1] shows that the full signal can also be 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 logic in this paper does not include this useful update and readers are encouraged to examine the changes in the FIFO style #1 paper[1].

This paper uses some very asynchronous techniques to generate full and empty status flags and Cliff Cummings does not plan to update this version of the FIFO design.

The clever quadrant techniques described in this paper have been incorporated into a new synchronous FIFO design style and are described in a paper using SystemVerilog coding styles (easily converted to plain Verilog coding styles) that will also shortly be available on the www.sunburst-design.com/papers web page.

Errata - A colleague, Zenja Chao, pointed out that there was a typo at the end of section 5.5. Removal of the **`wfull`** signal should happen when the READ POINTER (**`rptr`**) increments, not when the **`wptr`** increments.

References

[1] Clifford E. Cummings, “Simulation and Synthesis Techniques for Asynchronous FIFO Design,” *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, March 2002, Section TB2, 2nd paper. Also available at www.sunburst-design.com/papers

[2] 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

[3] 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

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

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

[6] Peter Alfke, “Asynchronous FIFO in Virtex-II™ FPGAs,” Xilinx techXclusives, downloaded from www.xilinx.com/support/techXclusives/fifo-techX18.htm

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 20th, 2005)

Peter Alfke, Director, Applications Engineering, Xilinx, Inc, San Jose, CA. Email address: peter.alfke@xilinx.com

Peter Alfke came to the US in 1966, with a German MSEE degree and nine years experience in digital systems and circuit design at LM Ericsson and Litton Industries in Sweden. He has been manager, later director of applications engineering for 34 years, at Fairchild, Zilog, AMD, and, since 1988, at Xilinx.

He holds fifteen patents, has written many Application Notes, presented at numerous design conferences, and has given many applications-oriented seminars in the US and in Europe. He is an active participant in the best newsgroup for FPGA users, comp.arch.fpga.

(Data accurate as of April 19th, 2002)