PROJECT

EE705: - VLSI Design Lab

Department of Electrical Engineering

(IIT Bombay)

Asynchronous FIFO Design

Submitted By: -

Mahesh Shahaji Patil (20307R010)

Yash Soni (203070109)

Aniket Srivastava (203070093)

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.

Attempting to synchronize multiple changing signals from one clock domain into a new clock domain and ensuring that all changing signals are synchronized to the same clock cycle in the new clock domain has been shown to be problematic. 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.

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.

Synchronous FIFO Pointers: -

A FIFO where writes to, and reads from the FIFO buffer are conducted in the same clock domain. For synchronous FIFO design, one implementation counts the number of writes to, and reads from the FIFO buffer to increment, decrement or hold the current fill value of the FIFO buffer. The FIFO is full when the FIFO counter reaches a predetermined full value and the FIFO is empty when the FIFO counter is zero.

Unfortunately, for asynchronous FIFO design, the increment-decrement FIFO fill counter cannot be used, because two different and asynchronous clocks would be required to control the counter. To determine full and empty status for an asynchronous FIFO design, the write and read pointers will have to be compared.

Asynchronous FIFO Pointers: -

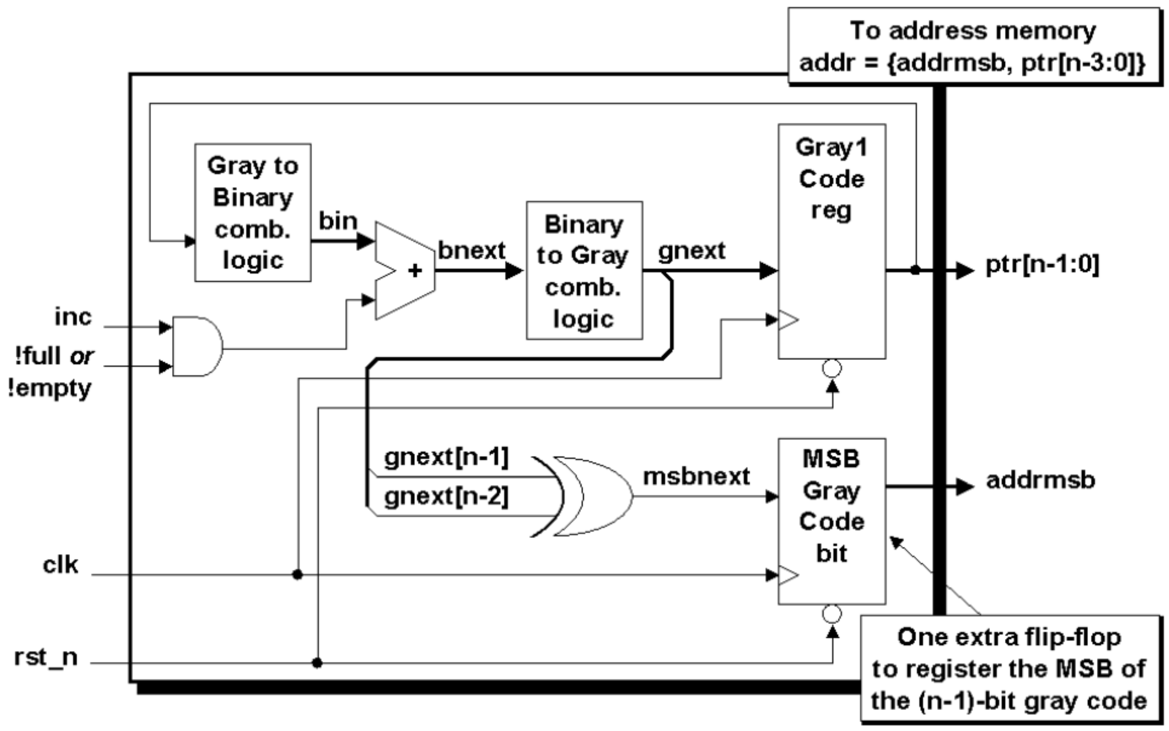
In order to understand FIFO design, one needs to understand how the FIFO pointers work. The write pointer always points to the next word to be written. Therefore, on reset, both pointers are set to zero, which also happens to be the next FIFO word location to be written. On a FIFO-write operation, the memory location that is pointed to by the write pointer is written, and then the write pointer is incremented to point to the next location to be written.

Similarly, the read pointer always points to the current FIFO word to be read. Again, on reset, both pointers are reset to zero, the FIFO is empty and the read pointer is pointing to invalid data (because the FIFO is empty and the empty flag is asserted). As soon as the first data word is written to the FIFO, the write pointer increments, the empty flag is cleared, and the read pointer that is still addressing the contents of the first FIFO memory word, immediately drives that first valid word onto the FIFO data output port, to be read by the receiver logic.

Gray Code Counter: -

A common approach to FIFO counter-pointers, is to use Gray code counters. Gray codes only allow one bit to change for each clock transition, eliminating the problem associated with trying to synchronize multiple changing signals on the same clock edge.

The first fact to remember about a Gray code is that the code distance between any two adjacent words is just 1 (only one bit can change from one Gray count to the next). The second fact to remember about a Gray code counter is that most useful Gray code counters must have power-of-2 counts in the sequence. It is possible to make a Gray code counter that counts an even number of sequences but conversions to and from these sequences are generally not as simple to do as the standard Gray code. Also note that there are no odd-count-length Gray code sequences.

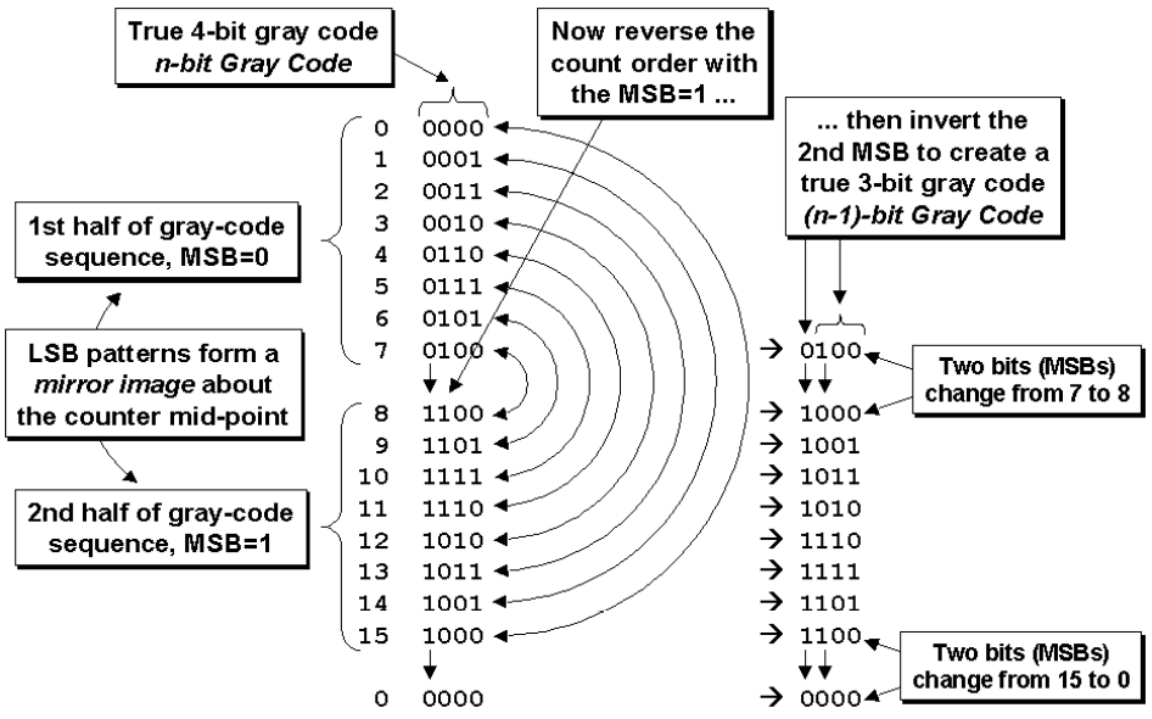

To address memory
addr = {addrmsb, ptr[n-3:0]}
Gray to Binary comb. logic
bin
bnext
+
Binary to Gray comb. logic
gnext
Gray1 Code reg
ptr[n-1:0]
inc
!full *or* !empty
gnext[n-1]
gnext[n-2]
msbnext
MSB Gray Code bit
addrmsb
clk
rst_n
One extra flip-flop to register the MSB of the (n-1)-bit gray code

This figure is a block diagram for dual n-bit Gray code counter. This 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 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.

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 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.

The n-bit Gray-code pointer is required to synchronize the pointers into the opposite clock domains and the (n-1)-bit Gray-code pointer can be used to address memory directly.


True 4-bit gray code
n-bit Gray Code
Now reverse the count order with the MSB=1 ...
... then invert the 2nd MSB to create a true 3-bit gray code
(n-1)-bit Gray Code
1st half of gray-code sequence, MSB=0
LSB patterns form a mirror image about the counter mid-point
2nd half of gray-code sequence, MSB=1
Two bits (MSBs) change from 7 to 8
Two bits (MSBs) change from 15 to 0
0 0000
1 0001
2 0011
3 0010
4 0110
5 0111
6 0101
7 0100
8 1100
9 1101
10 1111
11 1110
12 1010
13 1011
14 1001
15 1000
0 0000
0100
1000
1001
1011
1010
1110
1111
1101
1100
0000

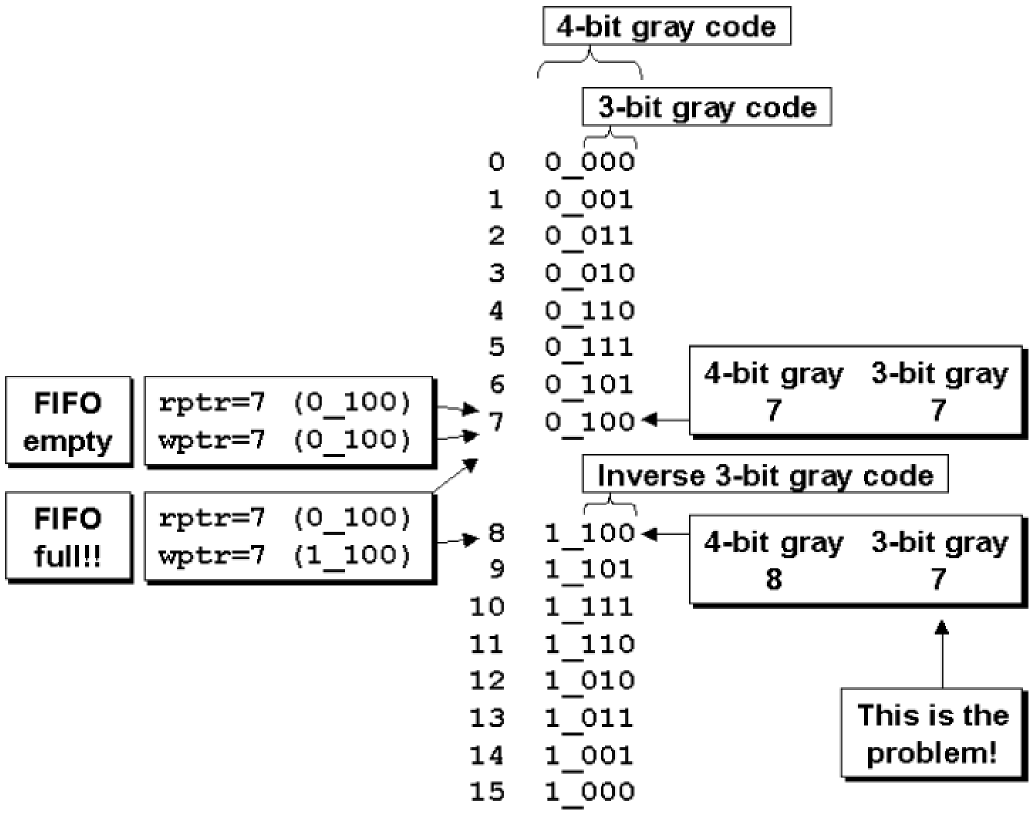
To better understand the problem of converting an n-bit Gray code to an (n-1)-bit Gray code, consider the example of creating a dual 4-bit and 3-bit Gray code counter as shown in above figure. The most common Gray code, as shown in above figure, is a reflected code where the bits in any column except the MSB are symmetrical about the sequence mid-point. 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.

<u>Problems with dual n-bit Gray code counter implemented using above technique: -</u>

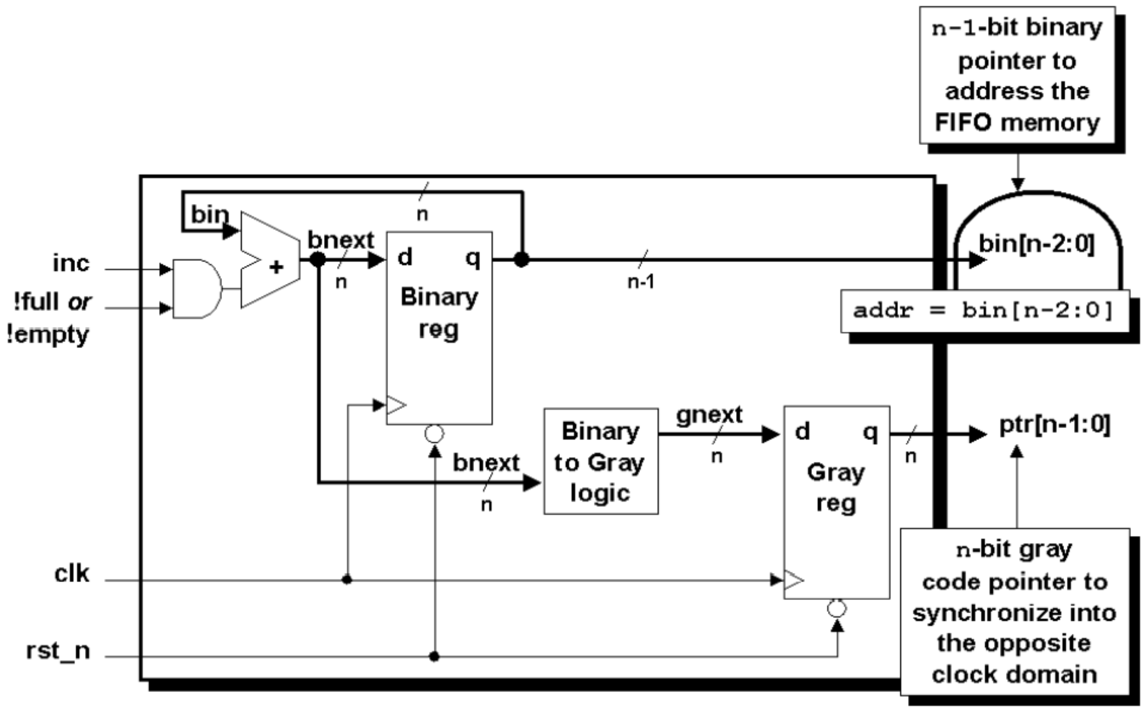


Consider the example shown in figure above of an 8-deep FIFO. In this example, a 3-bit Gray code pointer is used to address memory and an extra bit (the MSB of a 4-bit Gray code) is added to test for full and empty conditions. If the FIFO is allowed to fill the first seven locations (words 0-6) and then 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!

Hence, this is one reason why the dual n-bit Gray code counter of below figure is used.



This one is also another way of implementation of the dual n-bit Gray code counter, 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.

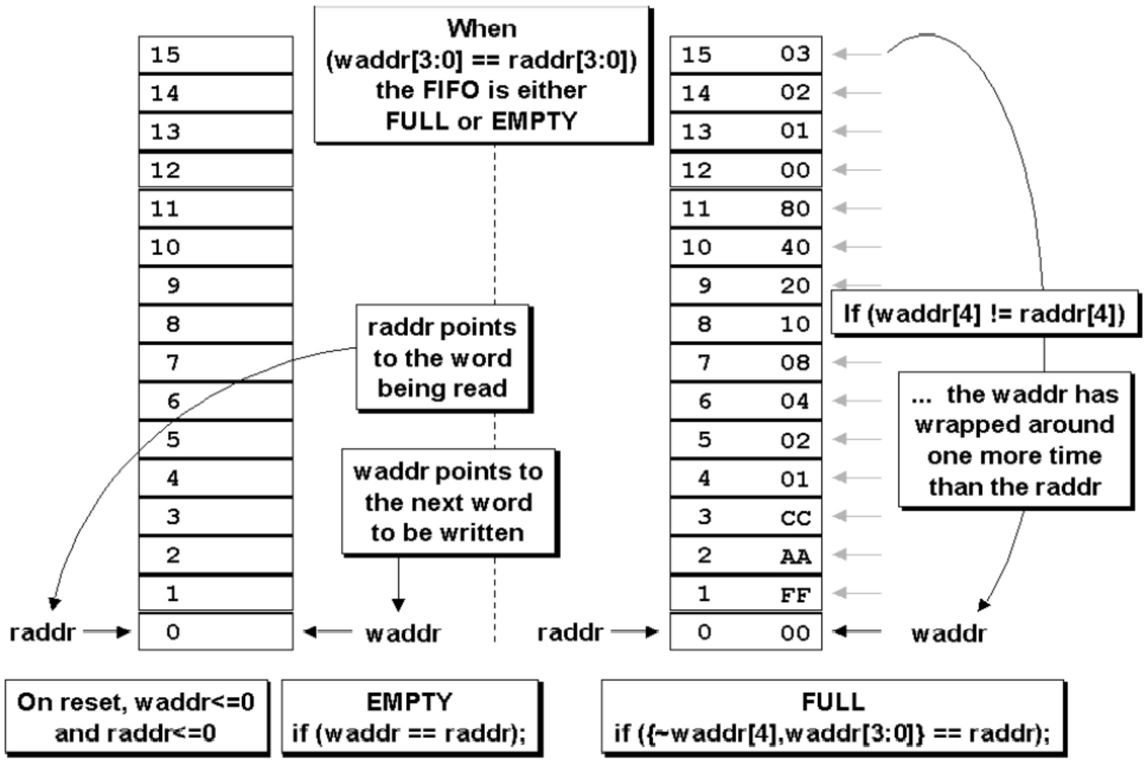
FIFO Full and Empty Condition: -

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 below figure (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.


When (waddr[3:0] == raddr[3:0]) the FIFO is either FULL or EMPTY
raddr points to the word being read
waddr points to the next word to be written
raddr
waddr
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
15 03
14 02
13 01
12 00
11 80
10 40
9 20
8 10
7 08
6 04
5 02
4 01
3 CC
2 AA
1 FF
0 00
If (waddr[4] != raddr[4])
... the waddr has wrapped around one more time than the raddr
On reset, waddr<=0 and raddr<=0
EMPTY if (waddr == raddr);
FULL if ({~waddr[4],waddr[3:0]} == raddr);

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.

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.

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.

The FIFO design in this project assumes that the empty flag will be generated in the read-clock domain to ensure 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).

Similarly, for full condition it assumes that the full flag will be generated in the write-clock domain to ensure 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).

FIFO Empty Condition: -

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 change 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).

FIFO Full Condition: -

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

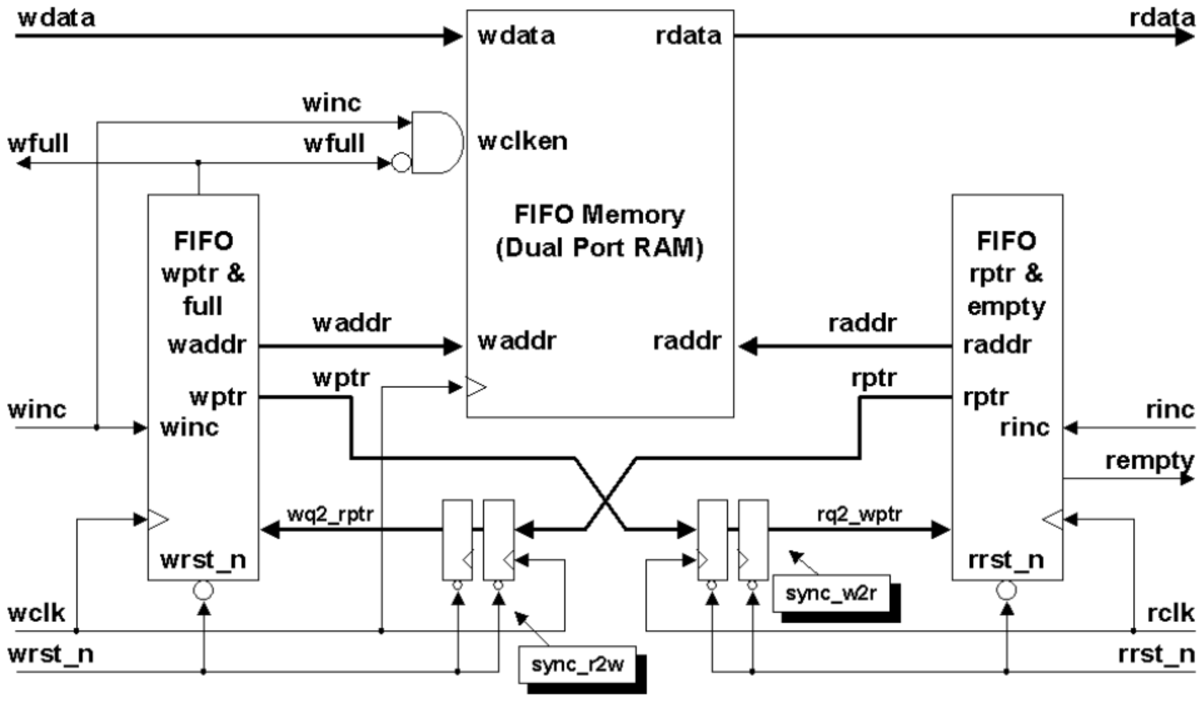
The full comparison is not as simple to do as the empty comparison. Pointers that are one bit larger than needed to address the FIFO memory buffer are still used for the comparison, but simply using Gray code counters with an extra bit to do the comparison is not valid to determine the full condition.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

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

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

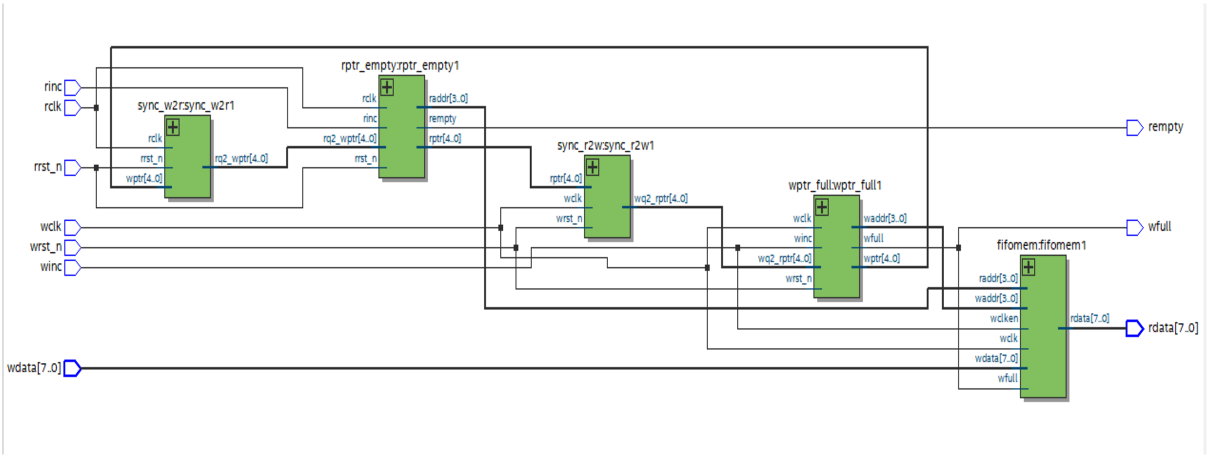
In order to efficiently register the wfull output, the synchronized read pointer is actually compared against the wgnext (the next Gray code that will be registered in the wptr).

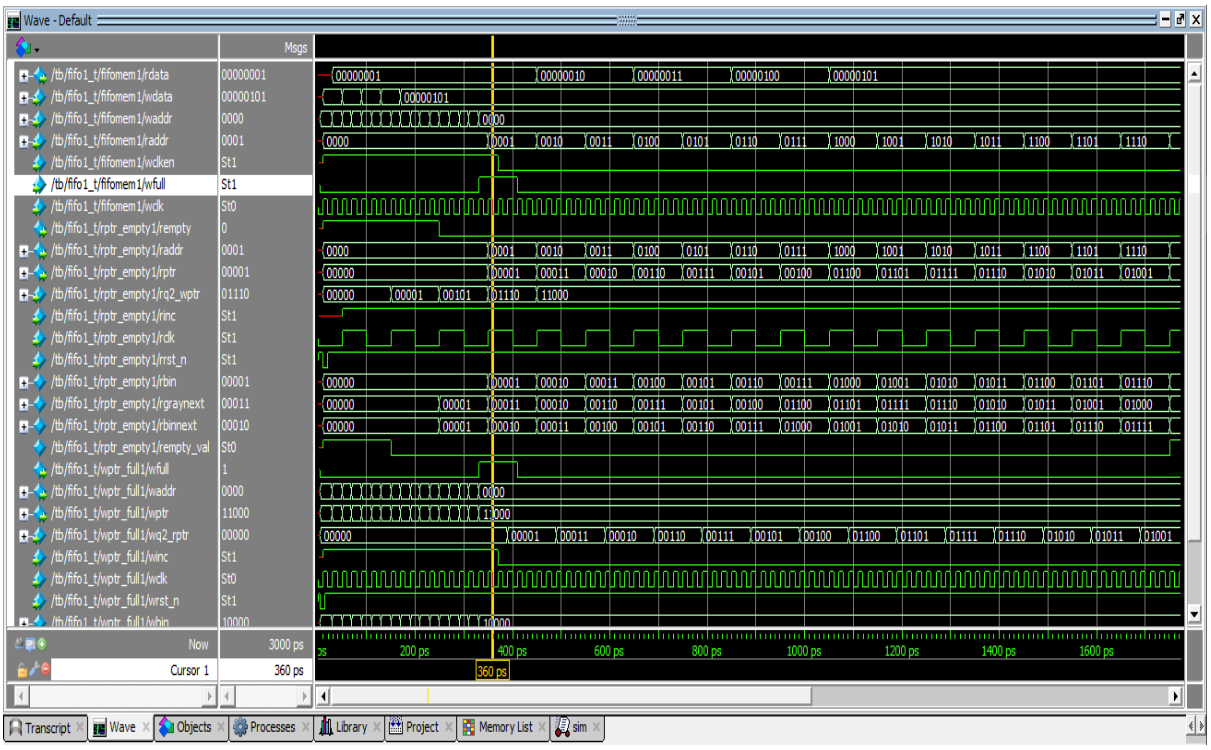
The Datapath formed from the discussion can be seen as: -



Simulation Results: -

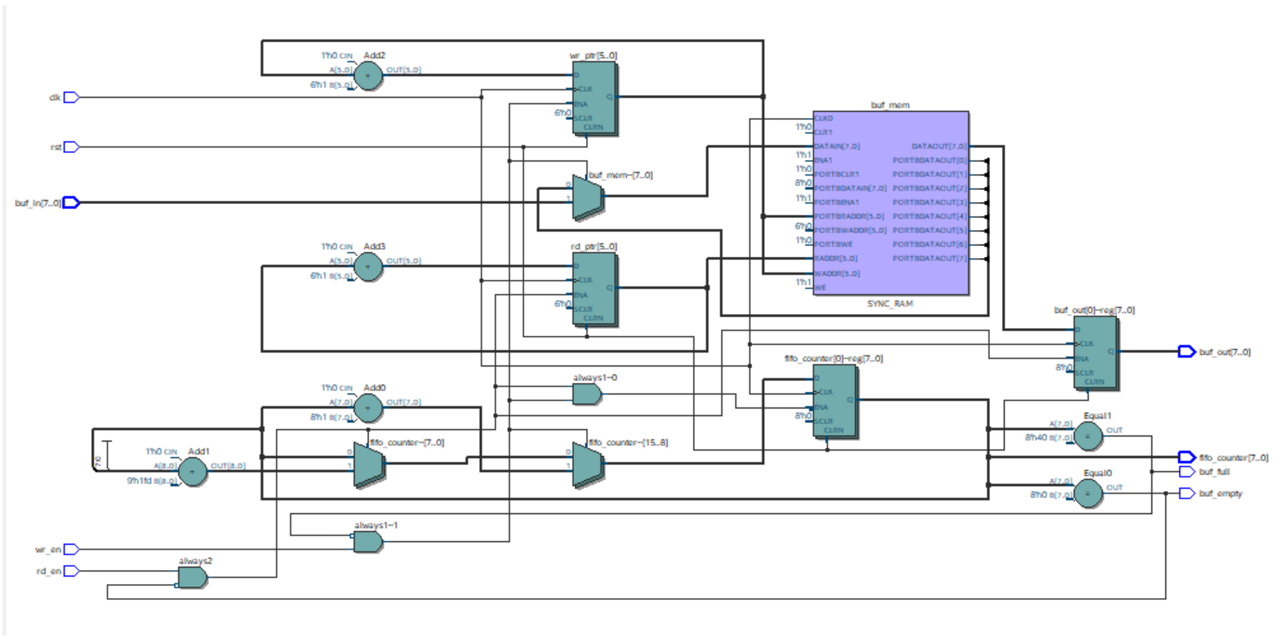
For Asynchronous FIFO: -

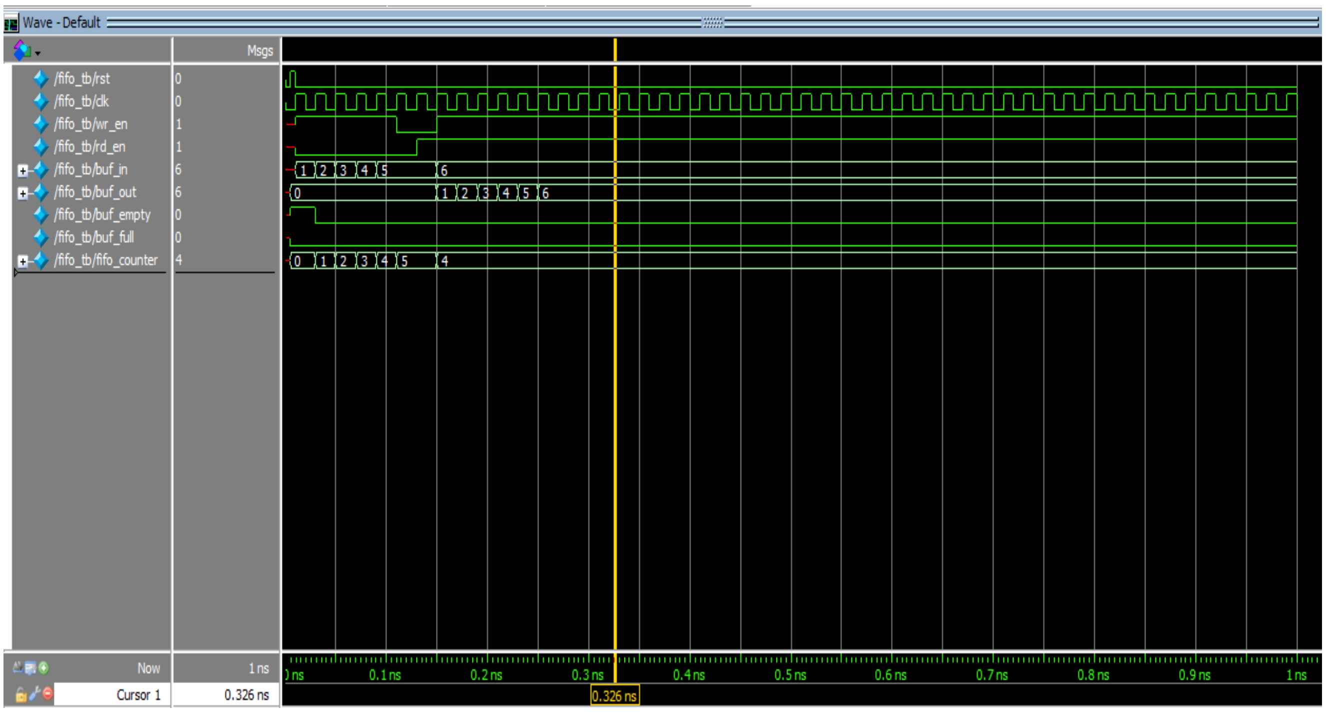



Wave - Default

	Msgs
/tb/fifo1_t/fifomem1/rdata	00000001
/tb/fifo1_t/fifomem1/wdata	00000101
/tb/fifo1_t/fifomem1/waddr	0000
/tb/fifo1_t/fifomem1/raddr	0001
/tb/fifo1_t/fifomem1/wclken	St1
/tb/fifo1_t/fifomem1/wfull	St1
/tb/fifo1_t/fifomem1/wclk	St0
/tb/fifo1_t/rptr_empty1/rempty	0
/tb/fifo1_t/rptr_empty1/raddr	0001
/tb/fifo1_t/rptr_empty1/rptr	00001
/tb/fifo1_t/rptr_empty1/rq2_wptr	01110
/tb/fifo1_t/rptr_empty1/rinc	St1
/tb/fifo1_t/rptr_empty1/rclk	St1
/tb/fifo1_t/rptr_empty1/rrst_n	St1
/tb/fifo1_t/rptr_empty1/rbin	00001
/tb/fifo1_t/rptr_empty1/rgraynext	00011
/tb/fifo1_t/rptr_empty1/rbinnext	00010
/tb/fifo1_t/rptr_empty1/rempty_val	St0
/tb/fifo1_t/wptr_full1/wfull	1
/tb/fifo1_t/wptr_full1/waddr	0000
/tb/fifo1_t/wptr_full1/wptr	11000
/tb/fifo1_t/wptr_full1/wq2_rptr	00000
/tb/fifo1_t/wptr_full1/winc	St1
/tb/fifo1_t/wptr_full1/wclk	St0
/tb/fifo1_t/wptr_full1/wrst_n	St1
/tb/fifo1_t/wptr_full1/wbin	10000
Now	3000 ps
Cursor 1	360 ps

For Synchronous FIFO (Just For learning): -





Conclusion: -

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

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 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

Video Link: -

EE705_Project.mp4