ASYNCHORNOUS FIFO

A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT
FOR THE COURSE OF

DIGITAL SYSTEMS DESIGN with FPGAs

In

Master of Technology

IN THE
FACULTY OF ENGINEERING

BY

GANESH KUMAR SHAW

GUIDED BY

PROF. KURUVILLA VARGHESE

DEPARTMENT OF ELECTRONIC SYSTEMS ENGINEERING

INDIAN INSTITUTE OF SCIENCE, BANGALORE

APRIL 2022

Notations

wdata : write data

waddr : write address

wptr : gray code of waddr

s_wptr : synchronised wptr in read clock domain

wclk : write clock

winc : write address increment signal

rdata : read data

raddr : read address

rptr : gray code of raddr

r_rptr : synchronised rptr in write clock domain

rclk : read clock

rinc : read address increment signal

Contents

Table of Contents **v**

List of Figures **vii**

1 Pre-study **1**

1.1 Introduction 1

2 Study **3**

2.1 Passing Multiple Asynchrnous Signal 3

2.1.1 Synchronous FIFO Pointer 3

2.1.2 Asynchronous FIFO Pointers 4

2.1.3 Empty Status 4

2.1.4 Full Status 5

2.1.5 How to differentiate between Empty and Full Status 5

2.1.6 Binary FIFO pointer consideration 6

2.2 Gray code counter 6

2.2.1 Gray code counter-Style #1 7

2.2.1.1 Dual n-bit Gray code cnounter 8

2.2.1.2 Additional Gray code counter consideration 9

2.2.2 Gray code counter - Style #2 9

2.3 Handling full & empty condition 10

2.3.1 Generating empty 10

2.3.2 Generating full 10

2.4 Different clock speeds 12

2.5 Pessimistic full & empty 13

2.6 Multi-bit asynchronous reset 14

2.7 Calculation of Fifo Depth 14

2.7.1 Case-1: $f_w > f_r$ with no idel cycles in both write and read 15

2.7.2 Case-2: $f_w > f_r$ with one clock cycle delay between two successive reads and writes 15

2.7.3 Case-3: $f_w > f_r$ with idel cycles in both write and read 15

2.7.4 Case-4: $f_w > f_r$ with duty cycles given for wr_enb and rd_enb. 16

2.7.5 Case-5: $f_w < f_r$ with no idel cycles in both write and read (i.e., the delay between two consecutive writes and reads is one clock cycle). . . 16

2.7.6 Case-6: $f_w < f_r$ with idel cycles in both write and read 16

2.7.7 Case-7: $f_w = f_r$ with no idel cycles in both write and read 17

2.7.8 Case-8: $f_w = f_r$ with idel cycles in both write and read 17

3 Design **19**

3.1 Verilog Codes 20

Bibliography **27**

List of Figures

1.1 Block Diagram 1

2.1 Empty Status Detection 4
2.2 Full Status Detection 5
2.3 Gray code pattern 7
2.4 Dual n bit code counter block diagram-style #1 8
2.5 Dual n bit code counter block diagram-style #2 9
2.6 Problems associated with extracting a 3-bit Gray code from a 4-bit Gray code . 11

3.1 FIFO BLOCK 19
3.2 Timing Simulation(when Wclk>Rclk) 26
3.3 Timing Simulation(when Wclk<Rclk) 26

Chapter 1

Pre-study

1.1 Introduction

[1]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 clocks domains are asynchronous to each other.

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

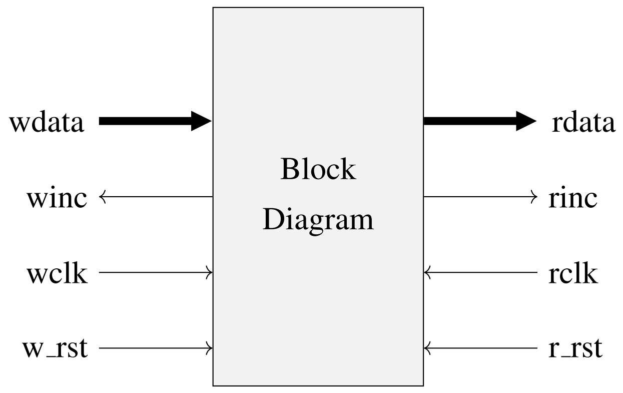

Figure 1.1: Block Diagram

Chapter 2

Study

2.1 Passing Multiple Asynchrnous Signal

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.*

2.1.1 Synchronous FIFO Pointer

For synchronous FIFO design ,a FIFO where writes to, and reads from the FIFO buffer are conducted in the same clock domain.To determine full and empty status for an synchronous FIFO design, the write and read address will have to be compared.

Unfortunately, for asynchronous FIFO design, the write-read addressed 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(synchronised read and write address).

2.1.2 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.*

2.1.3 Empty Status

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.

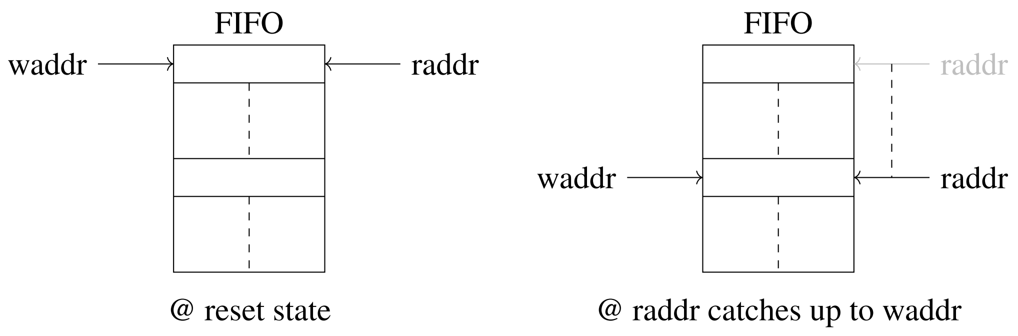

Figure 2.1: Empty Status Detection

2.1.4 Full Status

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 as well as pointers are reset to zero.

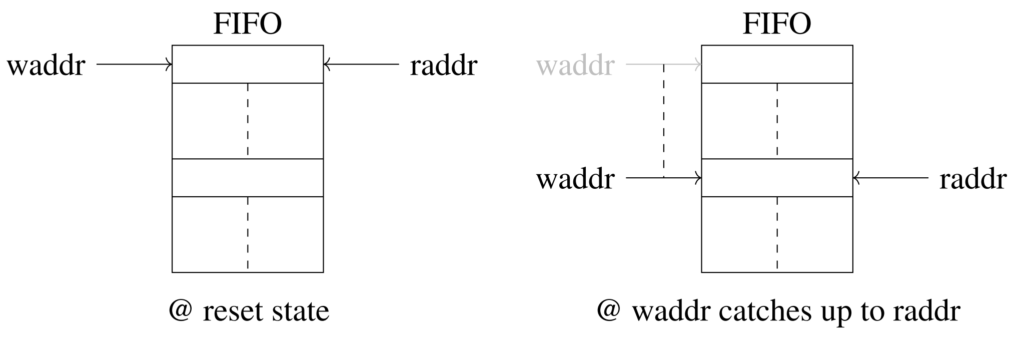

Figure 2.2: Full Status Detection

2.1.5 How to differentiate between Empty and Full Status

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 (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 which implies **FIFO is now Full**. If the MSBs of the two pointers are the same, it means that both pointers have wrapped the same number of times which implies **FFIFO is now Empty**.

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 uses n-bit pointers for a FIFO with $2^{(n-1)}$ write-able locations to help handle full and empty conditions.

2.1.6 Binary FIFO pointer consideration

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 $7 \rightarrow 8$ in binary numbers is $0111 \rightarrow 1000$, 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[2].

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[2].

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[2].

*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.*

2.2 Gray code counter

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 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 there are no odd-count-length Gray code sequences so ***one cannot make a 23-deep Gray code***.

Gray codes are named for the person who originally patented the code back in 1953, Frank Gray. There are multiple ways to design a Gray code counter.

2.2.1 Gray code counter-Style #1

The most common Gray code, as shown in 2.3, 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.** It

Decimal	Binary	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Figure 2.3: Gray code pattern

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 2^{nd} MSB to form an (n-1)-bit Gray code counter with shared LSBs. this is called a 'dual n-bit Gray code counter.'

2.2.1.1 Dual n-bit Gray code cnounter

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 .

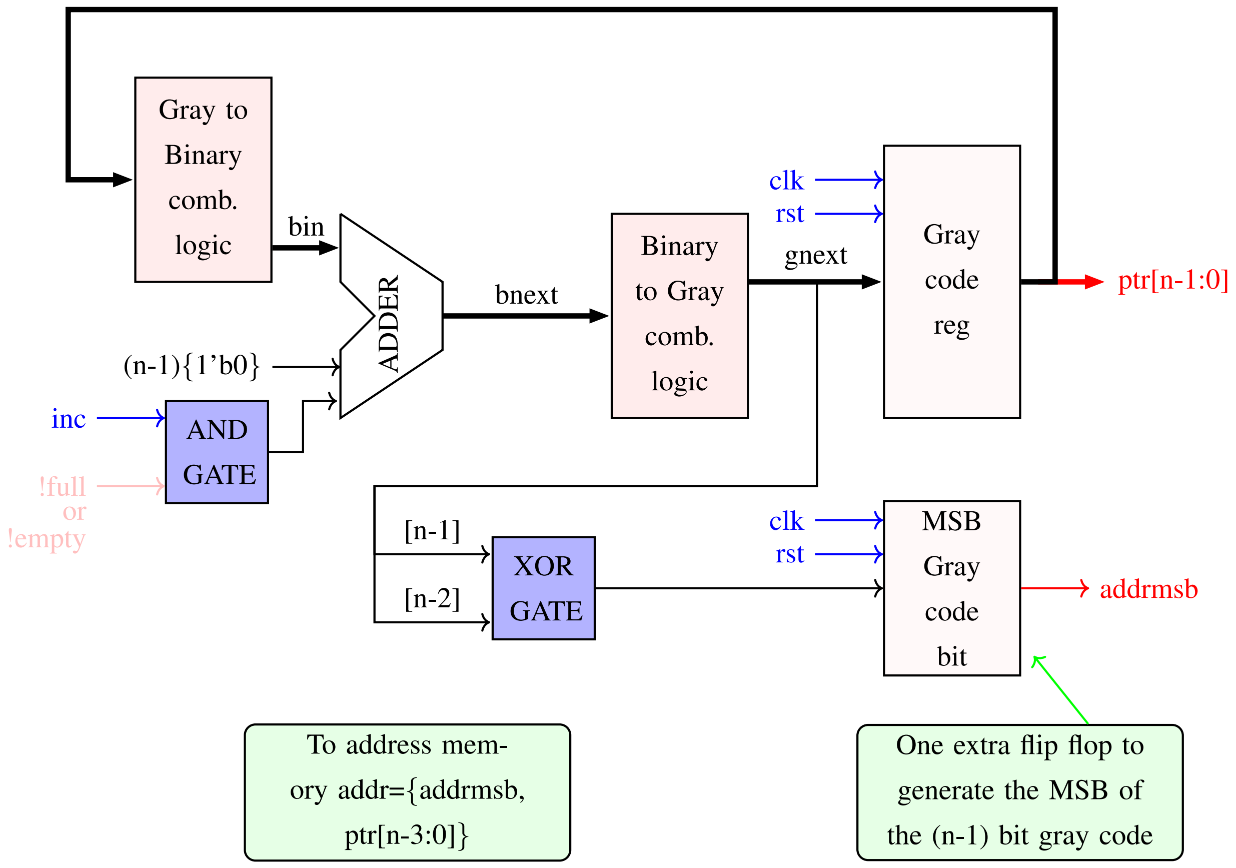
Gray to Binary comb. logic
bin
ADDER
bnext
(n-1){1'b0}
inc
!full or !empty
AND GATE
Binary to Gray comb. logic
clk
rst
gnext
Gray code reg
ptr[n-1:0]
[n-1]
[n-2]
XOR GATE
clk
rst
MSB Gray code bit
addrmsb
To address memory addr={addrmsb, ptr[n-3:0]}
One extra flip flop to generate the MSB of the (n-1) bit gray code

Figure 2.4: Dual n bit code counter block diagram-style #1

Fig 2.4 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 Fig 2.4 block diagram shows the described logic flow while the bottom half shows logic related to the second Gray code.

2.2.1.2 Additional Gray code counter consideration

The binary-value incrementer is conditioned with either an 'if not full' or 'if not empty' test as shown in Fig 2.4, 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.

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.

2.2.2 Gray code counter - Style #2

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.

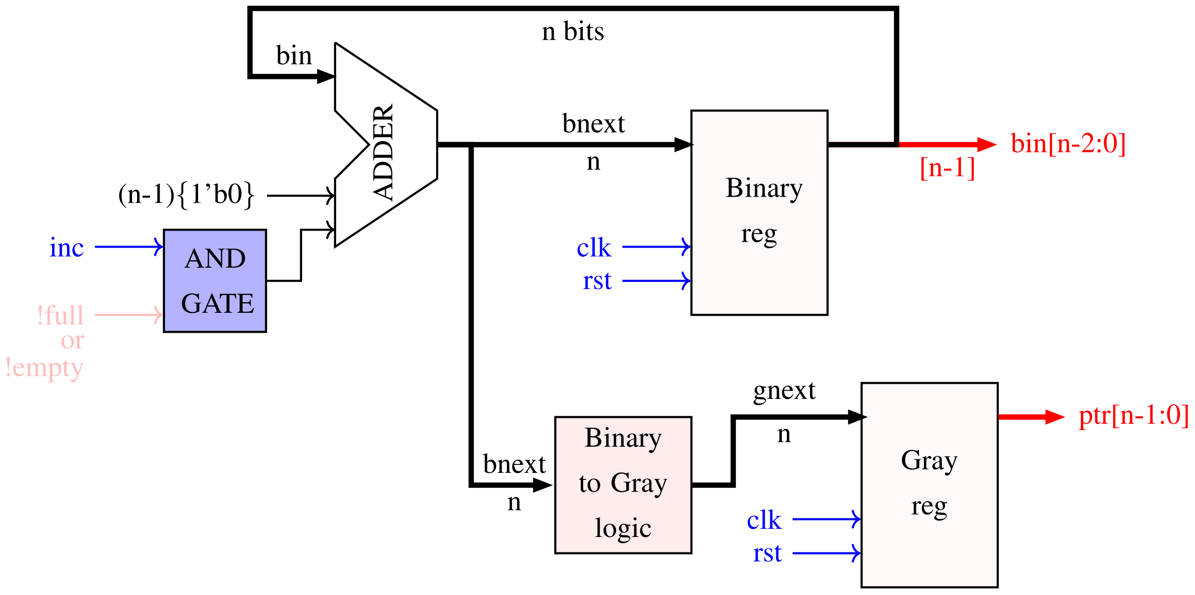

Figure 2.5: Dual n bit code counter block diagram-style #2

2.3 Handling full & empty condition

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

The FIFO design here 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, same MSBs indicate empty while different MSBs indicate full status).

Similarly 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).

2.3.1 Generating empty

As shown in fig 2.1, 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).

2.3.2 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 and converted into binray formate 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.

Consider the example shown in fig 2.6 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.

Decimal	Gray
0	0_000
1	0_001
2	0_011
3	0_010
4	0_110
5	0_111
6	0_101
7	0_100
8	1_100
9	1_101
10	1_111
11	1_110
12	1_010
13	1_011
14	1_001
15	1_000

Figure 2.6: Problems associated with extracting a 3-bit Gray code from a 4-bit Gray code

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 fig 2.5is used and for memory addressing we use binary address.

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 2 nd MSB from one pointer must be tested against the un-inverted 2 nd 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 wgraynext (the next Gray code that will be registered in the wptr).

2.4 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. 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.

2.5 Pessimistic full & empty

The FIFO 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.

2.6 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.*

2.7 Calculation of Fifo Depth

[3] The depth (size) of the FIFO should be in such a way that, the FIFO can store all the data which is not read by the slower module. FIFO will only work if the data comes in bursts; **we can't have continuous data in and out. If there is a continuous flow of data, then the size of the FIFO required should be infinite.** we need to know the burst rate, burst size, frequencies, etc. to determine the appropriate size of FIFO.

The logic in fixing the size of the FIFO is to find the no. of data items that are not read in a period in which the writing process is done. In other words, FIFO depth will be equal to the no. of data items that are left without reading.

2.7.1 Case-1: $f_w > f_r$ with no idel cycles in both write and read

Write frequency = f_w , Write time period = T_w

Read frequency = f_r , Read time period = T_r

Burst Length = No. of data items to be transferred = n

There are no idle cycles in both reading and writing which means that all the items in the burst will be written and read in consecutive clock cycles.

Sol.

Time required to write all the data in the burst = $n \times T_w$

The no. of data items can be read in the duration of $n \times T_w = (\frac{n \times T_w}{T_r})$

The remaining no. of bytes to be stored in the $FIFO, D = (n - (\frac{n \times T_w}{T_r}))$

So, the minimum depth of the FIFO should be 'D'.

2.7.2 Case-2: $f_w > f_r$ with one clock cycle delay between two successive reads and writes

Sol.

This is just, to create some sort of confusion. This scenario is no way different from the previous scenario (case -1), because, always, there will be one clock cycle delay between two successive reads and writes. So, the approach is same as the earlier one.

2.7.3 Case-3: $f_w > f_r$ with idel cycles in both write and read

Write frequency = f_w , Write time period = T_w

Read frequency = f_r , Read time period = T_r

Burst Length = No. of data items to be transferred = n

No. of idle cycles between two successive writes is = 1.

No. of idle cycles between two successive reads is = 3.

Sol.

The no. of idle cycles between two successive writes is 1 clock cycle. It means that, after writing one data, module A is waiting for one clock cycle, to initiate the next write. So, it can be understood that for every **two clock cycles**, one data is written.

The no. of idle cycles between two successive reads is 3 clock cycles. It means that, after reading one data, module B is waiting for 3 clock cycles, to initiate the next read. So, it can be understood that for every **four clock cycles**, one data is read.

Time required to write all the data in the burst = $n \times T_w \times \mathbf{2}$

The no. of data items can be read in the duration of $n \times T_w \times \mathbf{2} = (\frac{n \times T_w \times \mathbf{2}}{T_r \times \mathbf{4}})$

The remaining no. of bytes to be stored in the $FIFO, D = (n - (\frac{n \times T_w \times \mathbf{2}}{T_r \times \mathbf{4}}))$

So, the minimum depth of the FIFO should be 'D'.

2.7.4 Case-4: $f_w > f_r$ with duty cycles given for wr_enb and rd_enb.

Sol.

This scenario is no way different from the previous scenario (case - 3), because, in this case also, one data item will be written in 2 clock cycles and one data item will be read in 4 clock cycles.

2.7.5 Case-5: $f_w < f_r$ with no idel cycles in both write and read (i.e., the delay between two consecutive writes and reads is one clock cycle).

Sol. In this case, a FIFO of depth '1' will be sufficient because there will not be any data loss since the reading is faster than writing.

2.7.6 Case-6: $f_w < f_r$ with idel cycles in both write and read

Write frequency = f_w , Write time period = T_w

Read frequency = f_r , Read time period = T_r

Burst Length = No. of data items to be transferred = n

No. of idle cycles between two successive writes is = 1.

No. of idle cycles between two successive reads is = 3.

Sol.

The no. of idle cycles between two successive writes is 1 clock cycle. It means that, after writing one data, module A is waiting for one clock cycle, to initiate the next write. So, it can be understood that for every **two clock cycles**, one data is written.

The no. of idle cycles between two successive reads is 3 clock cycles. It means that, after reading one data, module B is waiting for 3 clock cycles, to initiate the next read. So, it can be understood that for every **four clock cycles**, one data is read.

Therefore, the calculation would be the same as case-3.

2.7.7 Case-7: $f_w = f_r$ with no idel cycles in both write and read

Sol.

FIFO is not required if there is no phase difference between clk_r and clk_w.

A FIFO of depth '1' will be sufficient if there is some phase difference between clk_r and clk_w.

2.7.8 Case-8: $f_w = f_r$ with idel cycles in both write and read

Read frequency = f_r , Read time period = T_r

Burst Length = No. of data items to be transferred = n

No. of idle cycles between two successive writes is = 1.

No. of idle cycles between two successive reads is = 3.

Sol.

The no. of idle cycles between two successive writes is 1 clock cycle. It means that, after writing one data, module A is waiting for one clock cycle, to initiate the next write. So, it can be understood that for every **two clock cycles**, one data is written.

The no. of idle cycles between two successive reads is 3 clock cycles. It means that, after reading one data, module B is waiting for 3 clock cycles, to initiate the next read. So, it can be understood that for every **four clock cycles**, one data is read.

Therefore, the calculation would be the same as case-3.

Chapter 3

Design

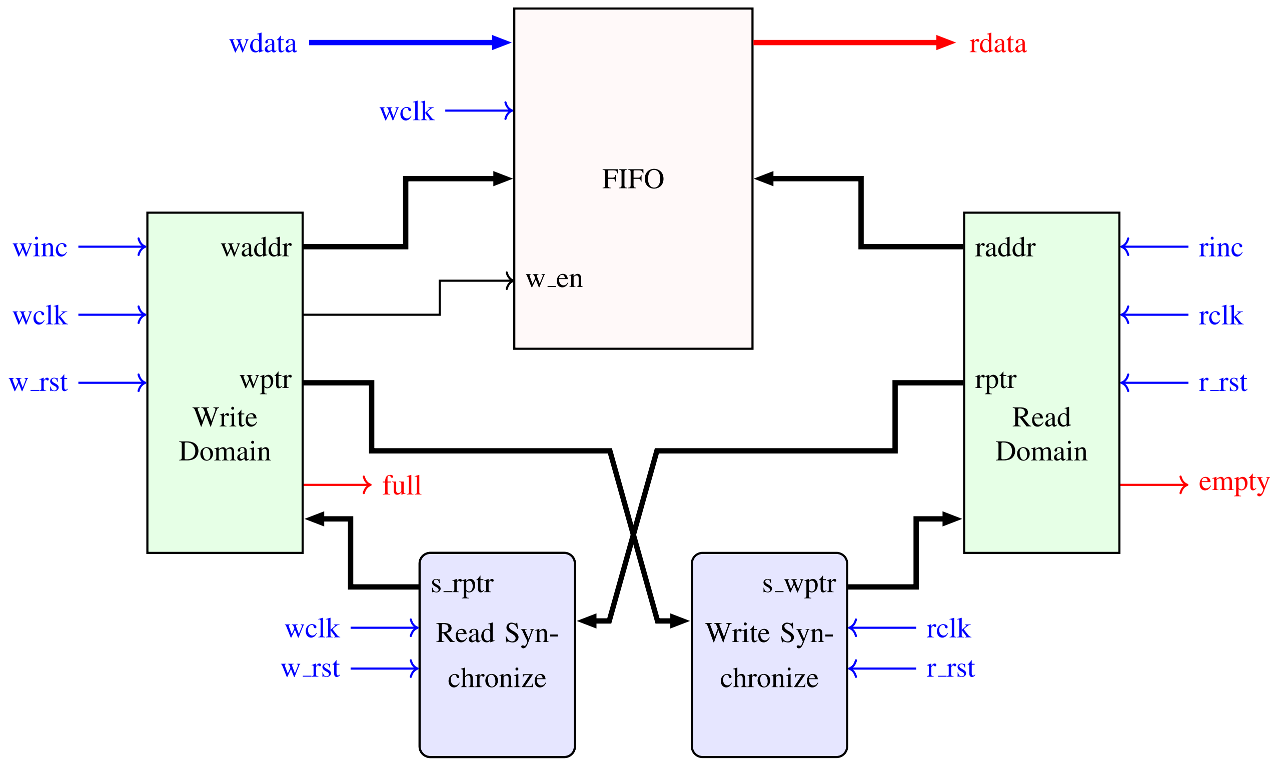

Figure 3.1: FIFO BLOCK

3.1 Verilog Codes

- Verilog Code for Top Module

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////////////////////


module TOP_MODULE(rdata,
                  empty,
                  full,
                  wclk,
                  winc,
                  w_rst,
                  rclk,
                  rinc,
                  r_rst,
                  wdata);
parameter DATASIZE=8;
parameter ADDRSIZE=4;
output[DATASIZE-1:0] rdata;
output empty,
       full;
input[DATASIZE-1:0] wdata;
input wclk,
      winc,
      w_rst,
      rclk,
      rinc,
      r_rst;
// fifo wire delcaration
wire[DATASIZE-1:0] fifo_rdata,
                   fifo_wdata;
wire fifo_wclk,
     fifo_w_en;
wire[ADDRSIZE-1:0] fifo_raddr,
                   fifo_waddr;
// wirte domain wire declaration
wire wd_full,
     wd_w_en,
     wd_wclk,
     wd_winc,
     wd_w_rst;
wire[ADDRSIZE-1:0] wd_waddr;
wire[ADDRSIZE:0] wd_wptr,
                 wd_s_rptr;
// Read domain wire declaration
wire rd_empty,
     rd_rclk,
     rd_rinc,
     rd_r_rst;
wire[ADDRSIZE-1:0] rd_raddr;
```

```
wire[ADDRSIZE:0] rd_rptr,
                 rd_s_wptr;
// read synchronise wire declaration
wire rs_clk,
     rs_rst;
wire[ADDRSIZE:0] rs_output,
                 rs_input;
// read synchronise wire declaration
wire ws_clk,
     ws_rst;
wire[ADDRSIZE:0] ws_output,
                 ws_input;
// fifo instantiation
FIFO  fifo(.rdata(fifo_rdata),
              .wclk(fifo_wclk),
              .w_en(fifo_w_en),
              .raddr(fifo_raddr),
              .wdata(fifo_wdata),
              .waddr(fifo_waddr));
assign fifo_wclk=wclk;
assign fifo_w_en=wd_w_en;
assign fifo_raddr=rd_raddr;
assign fifo_wdata=wdata;
assign fifo_waddr=wd_waddr;


// write domain instantiation
WDOMAIN wd(.full(wd_full),
               .w_en(wd_w_en),
               .waddr(wd_waddr),
               .wptr(wd_wptr),
               .wclk(wd_wclk),
               .winc(wd_winc),
               .w_rst(wd_w_rst),
               .s_rptr(wd_s_rptr));
assign wd_wclk=wclk;
assign wd_winc=winc;
assign wd_w_rst=w_rst;
assign wd_s_rptr=rs_output;

// Read domain instantiatio
RDOMAIN rd(.empty(rd_empty),
               .raddr(rd_raddr),
               .rptr(rd_rptr),
               .rclk(rd_rclk),
               .rinc(rd_rinc),
               .r_rst(rd_r_rst),
               .s_wptr(rd_s_wptr));
assign rd_rclk=rclk;
assign rd_rinc=rinc;
assign rd_r_rst=r_rst;
assign rd_s_wptr=ws_output;

```

```verilog

// read synchronise instantiation
SYNCHRONIZE rs(.ptr_out(rs_output),
                   .clk(rs_clk),
                   .ptr_in(rs_input),
                   .rst(rs_rst));
assign rs_clk=wclk;
assign rs_rst=w_rst;
assign rs_input=rd_rptr;


// write synchronise instantiation
SYNCHRONIZE ws(.ptr_out(ws_output),
                   .clk(ws_clk),
                   .ptr_in(ws_input),
                   .rst(ws_rst));
assign ws_clk=rclk;
assign ws_rst=r_rst;
assign ws_input=wd_wptr;
// output declaration
assign rdata=fifo_rdata;
assign empty=rd_empty;
assign full=wd_full;
endmodule
```

- Verilog Code for FIFO

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////


module FIFO   (rdata,
              wclk,
              w_en,
              raddr,
              wdata,
              waddr);
parameter DATASIZE=8;
parameter ADDRSIZE=4;
output[DATASIZE-1:0] rdata;
input wclk,
      w_en;
input[ADDRSIZE-1:0] waddr,
                    raddr;
input[DATASIZE-1:0] wdata;

//MEMORY mem(
//   .a(waddr),
//   .d(rdata),
//   .dpra(raddr),
//   .clk(wclk),
//   .we(w_en),
//   .dpo(rdata)
//);


localparam DEPTH=1<<ADDRSIZE;
reg[DATASIZE-1:0] mem[0:DEPTH-1];

always @(posedge wclk)
if(w_en)
 mem[waddr]<= #3 wdata;

assign rdata=mem[raddr];

endmodule
```

- Verilog Code for Writing Domain

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////
module WDOMAIN(full,
               w_en,
               waddr,
               wptr,
               wclk,
               winc,
               w_rst,
               s_rptr);
parameter ADDRSIZE=4;
parameter DATASIZE=8;
output reg full;
output     w_en;
output[ADDRSIZE-1:0] waddr;
output reg[ADDRSIZE:0] wptr;
input wclk,
      winc,
      w_rst;
input[ADDRSIZE:0] s_rptr;
reg [ADDRSIZE:0] wbin;
wire[ADDRSIZE:0] wbnext,
                 wgnext;
wire full_val;
assign wbnext=wbin+(winc & ~full);
assign wgnext=(wbnext>>1)^ wbnext;
assign full_val= (wgnext== {~s_rptr[ADDRSIZE:ADDRSIZE-1],s_rptr[ADDRSIZE-2:0]});
always @(posedge wclk)
 if(w_rst)
  begin
   wbin<= #3 0;
```

```
    wptr<= #3 0;
   end
  else
   begin
    wbin<= #3 wbnext;
    wptr<= #3 wgnext;
   end

// output declaration
assign waddr=wbin[ADDRSIZE-1:0];
always @(posedge wclk)
 begin
  if(w_rst)
   full<= #3 0;
  else
   full<= #3 full_val;
 end
assign w_en= (winc & ~full);   // CHECK
endmodule
```

- Verilog Code for Reading Domain

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////


module RDOMAIN(empty,
               raddr,
               rptr,
               rclk,
               rinc,
               r_rst,
               s_wptr);
parameter DATASIZE=8;
parameter ADDRSIZE=4;
output reg empty;
output[ADDRSIZE-1:0] raddr;
output reg[ADDRSIZE:0] rptr;
input rclk,
      rinc,
      r_rst;
input[ADDRSIZE:0] s_wptr;
wire[ADDRSIZE:0] rbnext,
                 rgnext;
wire empty_val;
reg[ADDRSIZE:0] rbin;
assign rbnext=rbin+(rinc & ~empty);
always @(posedge rclk)
begin
 if(r_rst)
```

```
  begin
   rbin<= #3 0;
   rptr<= #3 0;
  end
 else
  begin
   rbin<= #3 rbnext;
   rptr<= #3 rgnext;
  end
end

assign empty_val=(rgnext == s_wptr);     /// rst problem
assign rgnext=(rbnext>>1)^rbnext;
// output declaration
assign raddr=rbin[ADDRSIZE-1:0];

always @(posedge rclk)
if(r_rst)
 empty<= #3 1'b1;
else
 empty<= #3 empty_val;

endmodule
```

- Verilog Code for Synchronization

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////


module SYNCHRONIZE(ptr_out,
                   clk,
                   ptr_in,
                   rst);
parameter ADDRSIZE=4;
output reg[ADDRSIZE:0] ptr_out;
input clk,
      rst;
input[ADDRSIZE:0] ptr_in;
reg[ADDRSIZE:0] Q;
always @(posedge clk)
 if(rst)
  {ptr_out,Q}<= #3 0;
 else
  {ptr_out,Q}<= #3 {Q,ptr_in};

endmodule
```

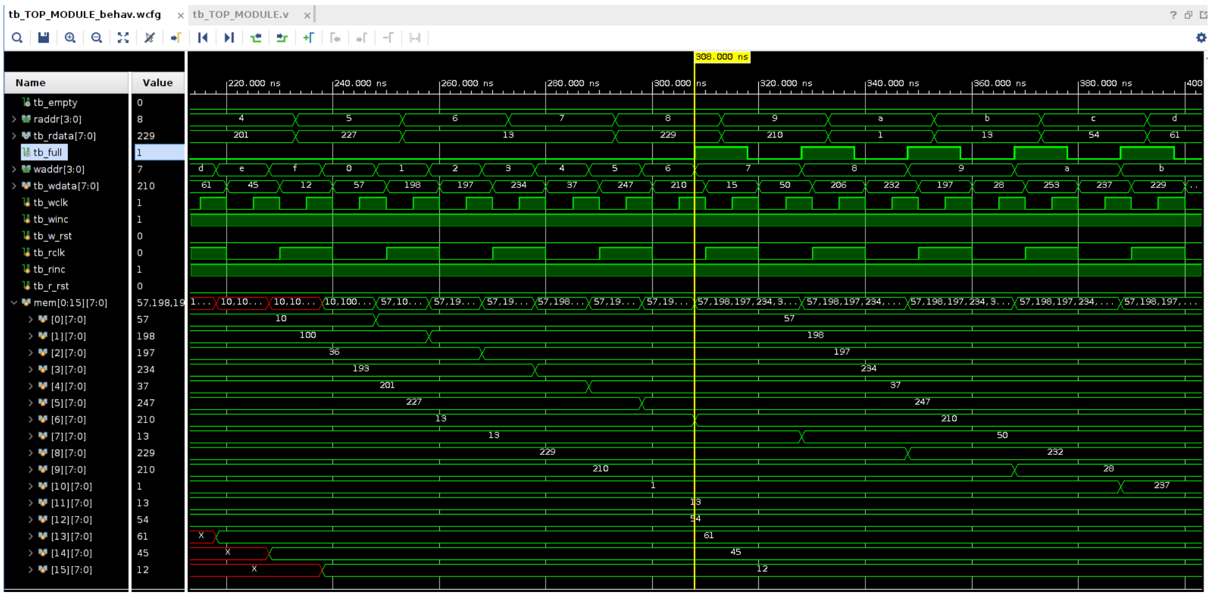

Figure 3.2: Timing Simulation(when Wclk>Rclk)

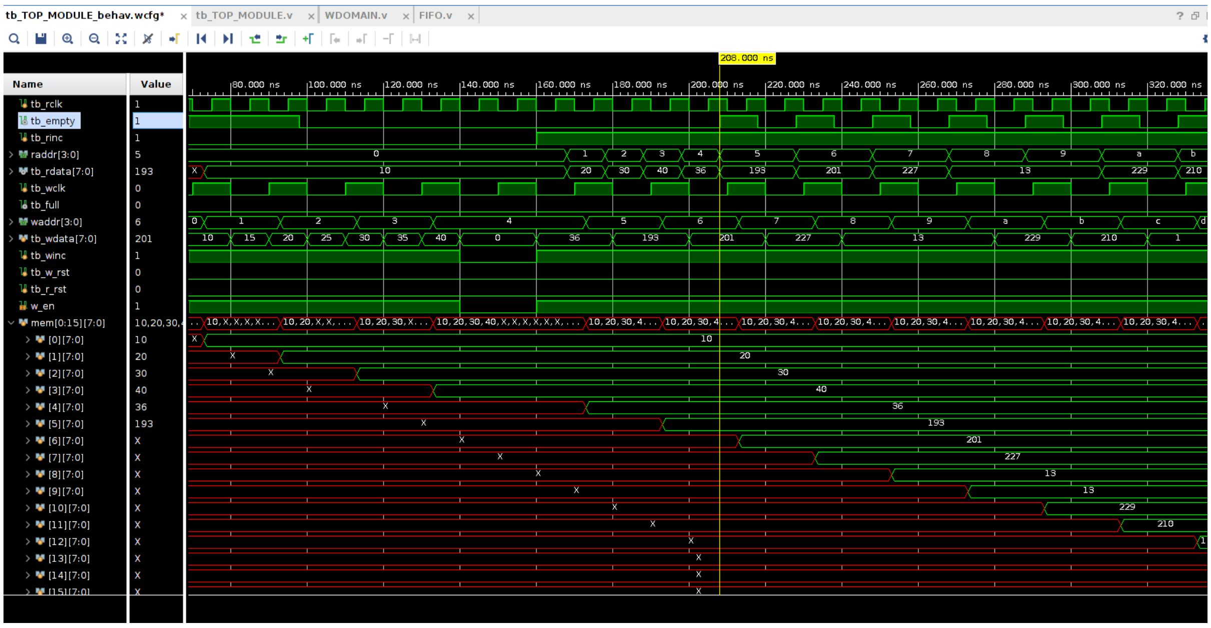

Figure 3.3: Timing Simulation(when Wclk<Rclk)

Bibliography

[1] Clifford Cummings and Peter Alfke. Simulation and synthesis techniques for asynchronous fifo design with asynchronous pointer comparisons. 01 2002.

[2] Steve Golson. Synchronization and metastability. *SNUG Silicon Valley*, 2014.

[3] https://hardwaregeeksblog.files.wordpress.com/2016/12/fifodepthcalculationmadeeasy2.pdf.