**ECE 6213 – Design of VLSI Circuits**

**Fall 2022 – The George Washington University – Dr. Jerry Wu**

**Osama Yousuf – HW 4**

**Problem 1 – Chapter 7 P2 from textbook – UART Clock Generator**

Diagram

Description automatically generated

The following is a breakdown of my implementation of the different modules for the UART\_Clock\_generator, as well as corresponding verification tests.

**Module 1:** Divide\_by\_N

For verification, I simply recreated the waveforms from the book (at N = 13). The N is a parameter which can be overridden to implement frequency division by any positive integer N.

The source file is: *dividebyN.v*, and the testbench is *dividebyN\_tb.v*

Diagram

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated

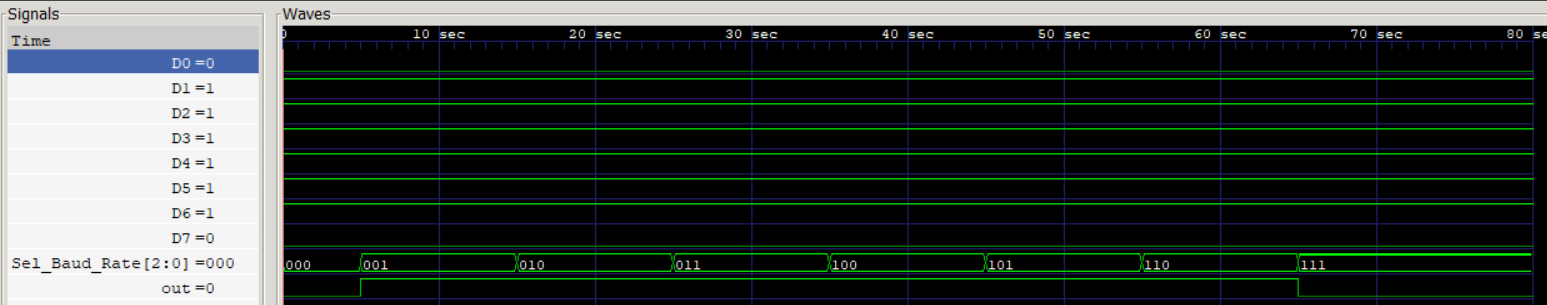
**Module 2:** 8 to 1 multiplexer

The source file is: *8to1mux.v*, and the testbench is *8to1mux \_tb.v*

For testing, I initialize each of the 8 inputs randomly, cycle through the select pins, and assert the correct output. I also implemented error monitoring, watchdog signals, as well as self-reporting in my testbench. The testbench executes with 0 errors.

Text

Description automatically generated



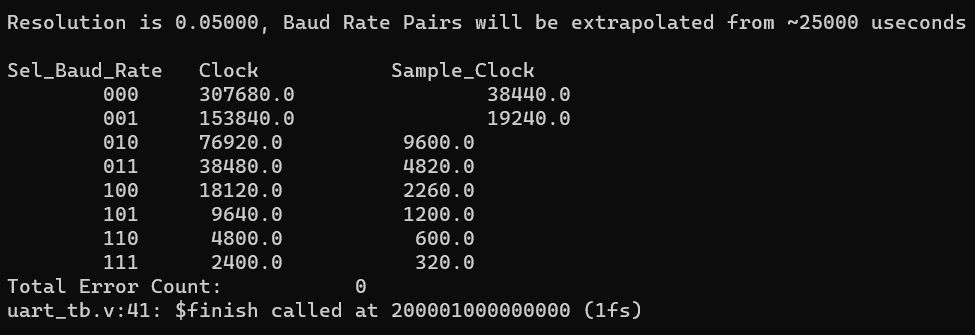
The waveform shows that the inputs are indeed correctly multiplexed on the out pin, based on the 3-bit Sel\_Baud\_Rate input pin.

**Overall Module:** UART\_Clock\_Generator

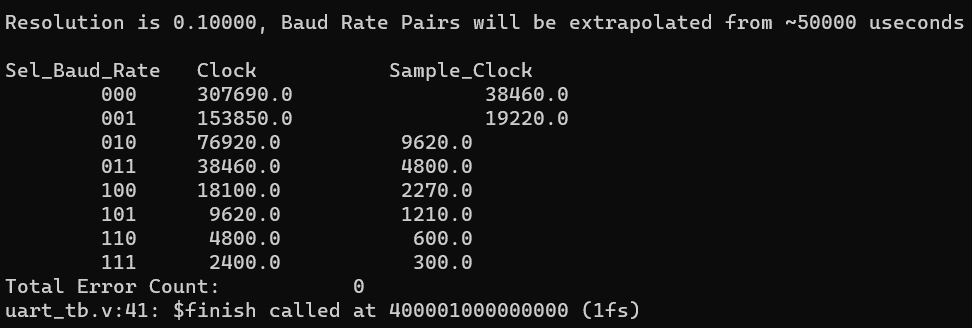
For verification, my testbench cycles through the 8 possible values of the “Sel\_Baud\_Rate” pin, and calculates the frequency of the two generated clock signals “Clock” and “Sample\_Clock” at a specified resolution parameter. This resolution parameter relates to how long the UART\_Clock\_Generator is allowed to run (in time cycles) at a given Sel\_Baud\_Rate value. For example, a resolution of 0.05 translates to a total time of 0.025 seconds per Sel\_Baud\_Rate, and a higher resolution of 2.00 translates to a total time of 1 seconds. The frequency of Clock and Sample\_Clock is simply extrapolated for a given resolution.

Verification results at different resolutions are presented below.

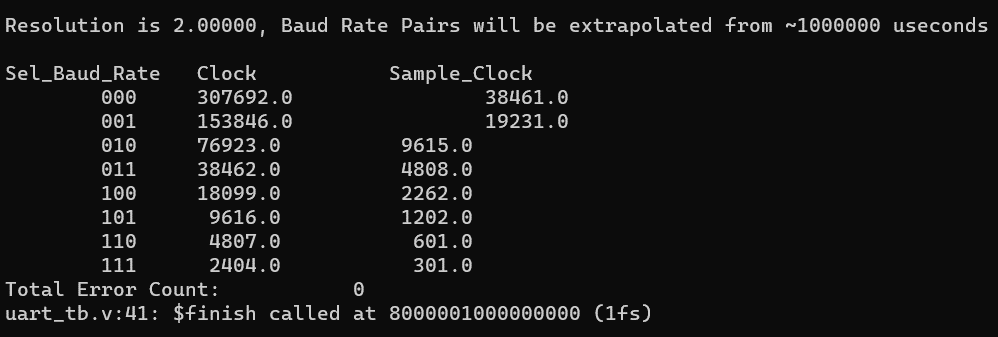
1. At very low resolution (0.025 second)



1. At medium resolution (0.05 second)



1. At high resolution (1.00 second):



As can be seen, the frequencies of “Clock” and “Sample\_Clock” both converge to the true frequencies described in Table P7-2 in the book, shown below. I did not have the computing resource to compute the values at a higher resolution (for example, 10.00 seconds). The reported frequency values of the baud rate pairs from my testbench at a resolution of 1.00 seconds are within a range of +-5% from the true values from the book, and so are acceptable to justify the verification for our purposes (as discussed).

Table

Description automatically generated

The source file is: *uart.v*, and the testbench is *uart \_tb.v*

**Note:** I have used error monitors, watchdog for finishing my simulations, self-checking, self-reporting, verilog events and tasks, and $random timings in my testbenches.

**Problem 2 – FIFO Data Structure (Queue) in Verilog**

**Block Diagram:**

Table

Description automatically generated with medium confidence

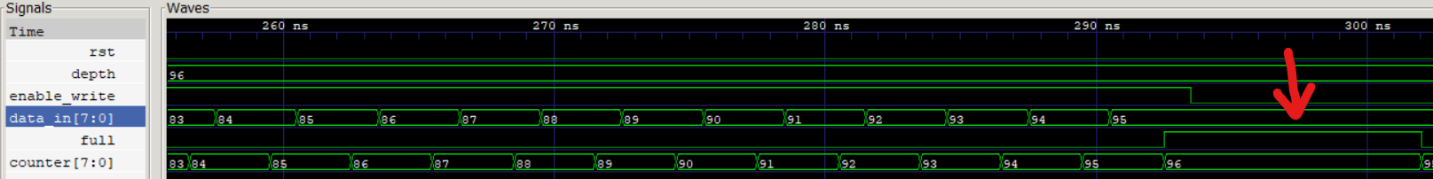
**Test Cases:**

|  |  |  |
| --- | --- | --- |
| **Test Case #** | **Name** | **Description** |
| 1 | Write until full | The full output of FIFO array should be asserted only when the array has been fully filled as a result of successive writes by the writer |
| 2 | Half Full | The Half Full output of FIFO array should be asserted only when the array has been at least half filled |
| 3 | Read until empty | The empty output of the FIFO array should be asserted when the array is empty as a result of either no data being filled, or the data being completely consumed by the reader |
| 4 | Empty Read | When read controller attempts to read an empty FIFO array, no data should be read |
| 5 | Full Write | When write controller attempts to write to a full FIFO array, the array should remain unchanged, and no data should be written |
| 6 | Write, Read Sequence Check | Successive samples of data should be written to the FIFO array by the write controller, followed by a sequence of reads until the array is empty. The test would check that the reader reads all the data in the correct sequence (first in, first out) |
| 7 | Write Controller Clock, Enable | The writing should only update the FIFO array in synchronization with its clock and when it’s enabled. |
| 8 | Read Controller Clock, Enable | The Read Controller should only update the FIFO array in synchronization with its clock – ClkR. |
| 9 | Simultaneous Read Write | Successive reading and writing should lead to correct behavior (with and without fork-join) |

**Test Case #1 -** Write until full

**Text

Description automatically generated**

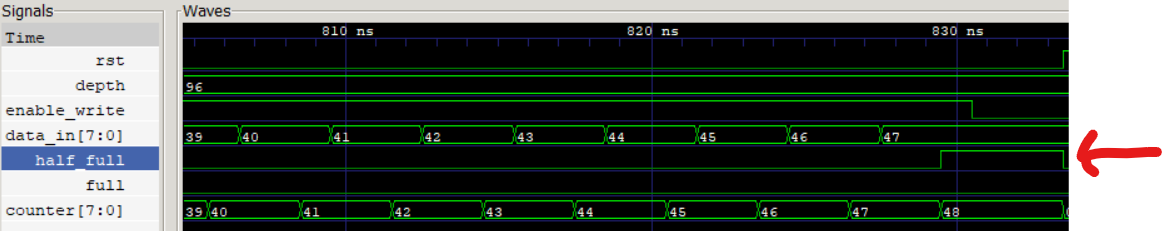
****

Starting from an empty queue, I push elements until it is full.

It can be seen that the “full” output of the queue is asserted when the queue has been completely filled i.e. the number of elements inside the queue, indicated by “counter”, are the same as the FIFO’s “depth”. My testbench also displays a warning sign for this case.

**Test Case #2 –** Half Full

Starting from an empty queue, I push elements until the queue is exactly half-filled.



It can be seen that the “half\_full” parameter is only asserted when the queue is exactly half filled. In this case, half-filled means that the queue has 48 elements written into it, since the depth is 96 and 96/2 = 48.

**Test Case #3 –** Read until empty

Starting from a filled queue, I read elements until the queue is empty.

Graphical user interface

Description automatically generated

It can be seen that whenever the queue is reset, the empty output is asserted, which is correct since the reset empties out memory contents of the FIFO.

Graphical user interface

Description automatically generated

It can be seen that when all data in the FIFO has been read, the counter becomes 0, and empty output is asserted correctly.

**Test Case #4 –** Empty Read

When the FIFO is empty, the testbench reports a warning that the read operation (dequeue) can not be done, since there is nothing to read in the queue. The “data\_out” remains unchanged (latches to previous value).

Graphical user interface, text

Description automatically generated

Graphical user interface, application

Description automatically generated

**Test Case #5 –** Full Write

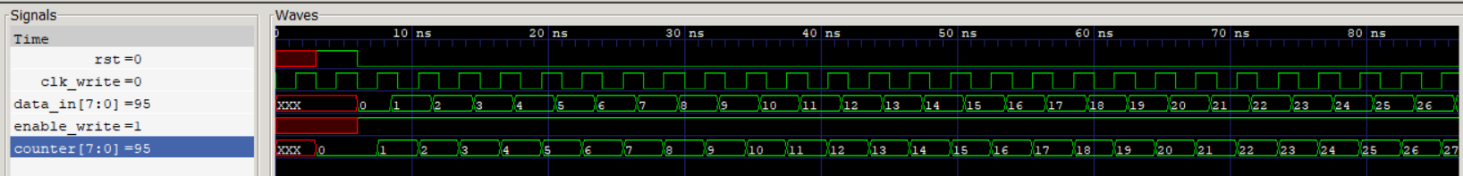
This has already been covered in Test Case #1.

**Test Case #6** – Write, Read Sequence Check

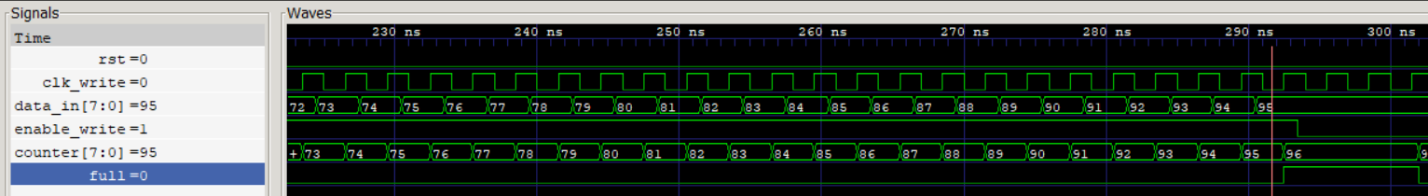
Starting from an empty FIFO, I write integers in the range [0, 95] in the queue till it’s full. I then perform a sequence of write operations until the queue is empty, and check that the elements are read in the order they were first written into the FIFO (first in-first out).

The test case passes, snippet of the waveform is attached below:

**Write operation Start**



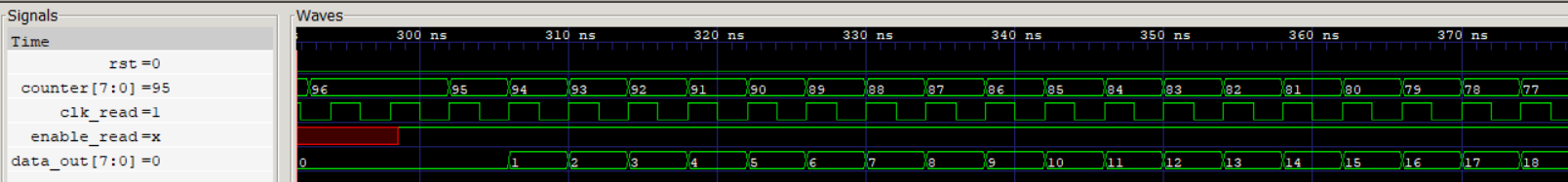
**Write operation End**



It can be seen that the integer present on “data\_in” is written into the FIFO at the positive edge of “clk\_write”, as indicated by the “counter”, and when the writer is enabled.

Once the FIFO is full, we now perform successive reads until the FIFO is empty.

**Read operation Start**



**Read operation End**

Graphical user interface

Description automatically generated

It can be seen that the “data\_out” reads the data in the same order as the data was written by the writer. It can also be seen that the reader reads only on the positive edge of its clock “clk\_read” and when the reader is enabled, which is independent of “clk\_write” (clock for the writer).

**Test Case #7 –** Write Controller Clock Enable

This has already been covered in Test Case #6.

**Test Case #8 –** Read Controller Clock Enable

This has already been covered in Test Case #6.

**Test Case #9 – Simultaneous Read & Write**

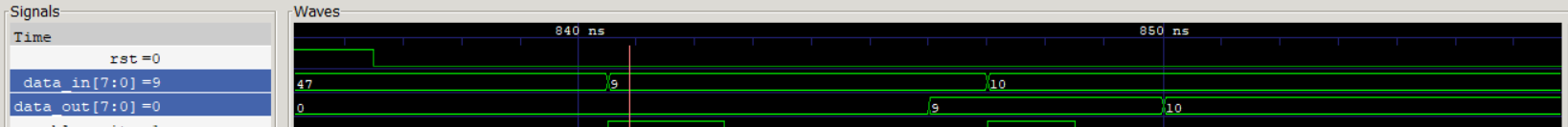
Firstly, starting from an empty queue, I write and read at the same time, and check if the correct value is read.

Secondly, I use fork-join to simulate simultaneous reading and writing to the FIFO in my testbench. For this test case to pass, the reader should read the simultaneously written data value on the second read operation, as coded below.

Text

Description automatically generated

My testbench passes without errors. This scenario is captured in the waveform as well:



The source file is: *fifo.v*, and the testbench is *fifo\_tb.v*

**Note:** I have used error monitors, watchdog for finishing my simulations, self-checking, self-reporting, verilog events (for assignments) and tasks (for enqueue, dequeue inside FIFO), and independent read and write clocks. My code for FIFO also supports separate widths for data\_in and data\_out (but they have to be multiple of one another), as indicated below:

From *fifo\_tb.v*

Text

Description automatically generated

The final self-reported output of my testbench is attached below, it finishes without any errors:

Text

Description automatically generated