Gisselquist Technology, LLC

**WBUART32** **SPECIFICATION**

Dan Gisselquist, Ph.D. dgisselq (at) ieee.org

November 2, 2018

Copyright (C) 2016–2017, Gisselquist Technology, LLC.

This project is free software (firmware): you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY, without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License and this program. If not, see http://www.gnu.org/licenses/ for a copy.

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev.** | **Date** | **Author** | **Description** |
| 1.01 | 6/02/2017 | D. Gisselquist | Clarified register descriptions |
| 1.0 | 2/20/2017 | D. Gisselquist | Added Hardware Flow Control |
| 0.2 | 1/03/2017 | D. Gisselquist | Added test-bench information |
| 0.1 | 8/26/2016 | D. Gisselquist | Initial Draft Specification |

**Table of Contents**

[List of Figures iv](#_Toc189036505)

[List of Tables iv](#_Toc189036506)

[Preface vi](#_Toc189036507)

[1.0 Introduction 1](#_Toc189036508)

[2.0 Architecture 2](#_Toc189036509)

[3.0 Operation 4](#_Toc189036510)

[4.0 Registers 6](#_Toc189036511)

[4.1 Setup Register 6](#_Toc189036512)

[4.2 FIFO Register 7](#_Toc189036513)

[4.3 RX DATA Register 9](#_Toc189036514)

[4.4 TX DATA Register 9](#_Toc189036515)

[5.0 Receiver’s Module (rxuart) State Machine 10](#_Toc189036516)

[5.1 RXU\_RESET\_IDLE 11](#_Toc189036517)

[5.2 RXU\_IDLE 12](#_Toc189036518)

[5.3 RXU\_BREAK 12](#_Toc189036519)

[5.4 RXU\_BIT\_ZERO 13](#_Toc189036520)

[5.5 RXU\_BIT\_ONE 13](#_Toc189036521)

[5.6 RXU\_BIT\_TWO 14](#_Toc189036522)

[5.7 RXU\_BIT\_THREE 14](#_Toc189036523)

[5.8 RXU\_BIT\_FOUR 15](#_Toc189036524)

[5.9 RXU\_BIT\_FIVE 15](#_Toc189036525)

[5.10 RXU\_BIT\_SIX 16](#_Toc189036526)

[5.11 RXU\_BIT\_SEVEN 16](#_Toc189036527)

[5.12 RXU\_PARITY 17](#_Toc189036528)

[5.13 RXU\_STOP 17](#_Toc189036529)

[5.14 RXU\_SECOND\_STOP 18](#_Toc189036530)

[6.0 Clock 18](#_Toc189036531)

[7.0 Clock Domains 19](#_Toc189036532)

[8.0 Reset Domains 19](#_Toc189036533)

[9.0 Wishbone Datasheet 19](#_Toc189036534)

[10.0 Module Hierarchy 20](#_Toc189036535)

[11.0 IO Ports 22](#_Toc189036536)

[12.0 RTL Parameters 24](#_Toc189036537)

# List of Figures

[Figure 1 SETUP Register fields. 7](#_Toc180003886)

[Figure 2 RXDATA Register fields. 8](#_Toc180003887)

[Figure 3 RXDATA Register fields. 8](#_Toc180003888)

[Figure 4 TXDATA Register fields 8](#_Toc180003889)

[Figure 5 Receiver's module (rxuart) state machine diagram. 11](#_Toc180003890)

# List of Tables

[Table 1 UART Registers 6](#_Toc180003891)

[Table 2 Parity Setup 7](#_Toc180003892)

[Table 3 Clock Requirements 19](#_Toc180003893)

[Table 4 Wishbone Datasheet 19](#_Toc180003894)

[Table 5 ufifo module IO ports 22](#_Toc180003895)

[Table 6 rxuart module IO ports 23](#_Toc180003896)

[Table 7 txuart module IO ports 23](#_Toc180003897)

[Table 8 wbuart module IO ports 23](#_Toc180003898)

[Table 9 ufifo module default RTL parameters 24](#_Toc180003899)

[Table 10 rxuart module default RTL parameters 25](#_Toc180003900)

[Table 11 txuart module default RTL parameters 25](#_Toc180003901)

[Table 12 wbuart module default RTL parameters 25](#_Toc180003902)

# Preface

Building a UART may be a mandatory coming-of-age task for any HDL designer. The task is simple and easy, and there is little to it. This project is based on some of my first experiences with Verilog.

Since then, it has been augmented with several useful capabilities for simulating a UART connection when using Verilator. This unusual addition to the core makes this core worth noting. I hope you find it useful.

Dan Gisselquist, Ph.D.

# Introduction

Universal Asynchronous Serial Transport, or UART, has become a common protocol between devices. It is simple to wire up, easy to use, and easy to process. This core provides one implementation of the logic necessary to use such a communications scheme.

While you are likely to find many UART examples out there, this UART implementation offers something many of these other examples do not: a Verilator simulation capability that allows the user to connect via a TCP/IP port or a telnet application to the UART of their desired chip. As a result, full two-way interaction can be had between a simulation and a terminal or other port. Indeed, this may even be sufficient to connect a CPU capable of running Linux to a terminal to verify that it can truly run Linux–all within Verilator.

As a final addition, the test bench section contains four files that can be used as top-level design files to prove whether the serial port on a given circuit board works.

# Architecture

The HDL portion of the core itself consists of four basic files: rxuart.v, txuart.v, ufifo.v and wbuart.v. These are the receive UART code, the transmit UART code, a generic FIFO, and a fully wishbone-compliant UART peripheral. This latter file demonstrates how the receiver, transmitter, and a pair of FIFOs may be connected to a Wishbone bus. A fifth file, wbuart-insert.v, demonstrates how the rxuart.v and txuart.v files may be included in a module implementing a simpler wishbone interface without the FIFO.

Each core file, rxuart.v, and txuart.v, is fully capable. Both files each accept a 30-bit setup value specifying baud rate, the number of bits per byte (between 5 and 8), whether hardware flow control is off, or whether parity is used, and if so, whether that parity is even, odd, or fixed mark or fixed space. This setup register will be discussed further in Chapter 4.1.

A further note on the rxuart.v module is in order. This module double latches the input in the proper two buffer fashions to avoid problems with metastability. Then, upon detecting the start bit (i.e., a high-to-low transition), the port waits for half of a baud and then starts its baud clock to sample in the middle of every baud following. The result of this is a timing requirement. After *N* + 2 baud intervals (*N* + 3 if parity is used), where *N* is the number of bits per byte, this calculated middle sample must remain within the associated bit period. This requirement leads to the following criteria:

Equation 1:

where *f*SYS is the system clock frequency, is the baud rate or frequency, CKS is the number of clocks per baud as set in the configuration register, and *N* is the number of bits per byte. When transmission rates where *f*BAUD approaches *f*SYS, the number of data rates that can be synthesized becomes limited.

Connecting to either txuart.v or rxuart.v is quite simple. Both files have a data port and a strobe. To transmit, set the data and strobe lines. Drop the strobe line on the clock after the busy line is low. Likewise, to connect to the rxuart.v port, there is a data and a strobe. This time, though, these two wires are outputs of the receive module instead of inputs. When the strobe is high, the data is valid. It will only be high for a one-clock period. If you wish to connect this output to a bus, a register will be needed to hold the strobe high until the data is read, as in wbuart-insert.v. Also, while the strobe is high, the o\_frame\_err will indicate a framing error (i.e., no stop bit), and o\_parity\_err will indicate whether the parity matched. Finally, the o break line will indicate whether the receiver is in a “break” state,

The tx\_busy line may be inverted and connected to a transmit interrupt line. Similarly, the rx\_stb line, or the bus equivalent of rx\_ready, may receive interrupt lines–although it will need to be latched as both wbuart.v and wbuart-insert.v demonstrate.

A simple example of putting this configuration together is in wbuart-insert.v. In this example, the rx\_data register will have only the lower eight bits set if the data is valid; higher bits will be set upon error conditions and cleared automatically upon the next byte read. Similarly, the tx\_data register can be written with a byte to transmit that byte. Writing bit ten will place the transmitter into a “break” condition, which will only be cleared by writing a zero to that bit later. Reading from the tx\_data register can also determine if the transmitter is busy (via polling), whether it is in a break condition, or even what bit is being placed to the output port.

A more comprehensive example of how these UART modules may be used together can be found in wbuart.v. This file provides a full wishbone interface allowing interaction with the core using four registers: a setup register, receive register, and transmit register as before, and a FIFO health register through which the size and fill of the FIFO can be queried.

The C++ simulation portion of the code revolves around the file bench/cpp/uartsim.cpp and its associated header. This file defines a class, UARTSIM, which can connect the UART to a TCP/IP stream. When initialized, this class takes, as input, the TCP/IP port number that the class is to connect with. Setting the port to zero connects the UART to the standard input and output file facilities. Once connected, using this simulator is as simple as calculating the receive input bit from the transmit output bit when the clock is low, and the core takes care of everything else.

Finally, a series of example files can be found in the bench/Verilog directory. helloworld.v presents an example of a simple UART transmitter repeatedly sending the “Hello, World \r\n” message. This example uses only the txuart.v module and can be simulated in Verilator. A second test file, echotest.v, works by echoing every received character to the transmit port, testing both txuart.v and rxuart.v. A third test file, linetest.v, works by waiting for a line of data to be received, then parrots that line back to the terminal. A fourth test file, speechfifo.v tests the wishbone interface and the FIFO by filling in the UART, 10 samples simultaneously, with text from Abraham Lincoln’s Gettysburg address. All three example files may be stand-alone top-level design files to verify your UART hardware functionality.

# Operation

A couple of steps are required to use the core. First, wire it up, including wiring the i\_uart and o\_uart ports and any i\_cts\_n and/or o\_rts\_n hardware flow control. The rxuart.v and txuart.v files may be wired up for use individually or as part of a large module, such as the example in wbuart-insert.v. Alternatively, the wbuart.v file may be connected to a straight 32-bit wishbone bus. Second, set the UART configuration register. Ideally, it is set by setting the INITIAL SETUP parameter of rxuart, txuart, or even wbuart. Alternatively, you can write to the setup register later, as is done within the speechfifo.v bench test.

From a simulation standpoint, it must also be “wired” up inside your C++ main Verilator file. Somewhere, internal to the top-level Verilator C++ simulation file, you will want to have some setup lines like

#include “uartsim.h” *// Tell compiler about UARTSIM*

*.*

*.*

*.*

UARTSIM \*uartsim; *// Declare a variable to hold the simulator.*

uartsim = new UARTSIM(ip port); *// Create/initialize it with your TCP/IP port #* uartsim->setup(setup register value); *// Tell it the line coding to expect*

and then another set of lines within your clocked section that look something like,

if (!clk)

tb->i\_uart\_rx= uartsim(tb->o uart tx);

You should be able to find several examples of this in the helloworld.cpp, linetest.cpp, and speechtest.cpp files. These C++ implementations, though, are also complicated by the need for a self–contained testing program to be able to capture and know what was placed onto the standard input and output streams; hence, many of them fork() into two processes so that one process can verify the output of the other. Both speechtest.cpp and linetest.cpp allow a *-i* option to run in an interactive mode without forking. Either way, forking the simulation program should not be needed for normal usage of these techniques, but you may find it helpful to know if you should examine this code or if you wish to build your test file that proves its output.

To use the transmitter, set the i\_stb and i\_data wires. Drop the strobe line any time after

(i\_stb)&&(!o\_busy).

To use the receiver, grab the data any time o\_stb is true.

From the standpoint of the bus, there are two ways to handle receiving and transmitting: polling and interrupt-based, although both work one character at a time. To poll, repeatedly read the receive data register until only bits from the bottom eight are set. It is an indication that the byte is valid. Alternatively, you could wait until an interrupt line is set and read. In the wbuart-insert.v example, as well as the wbuart.v implementation, the o\_uart\_rx\_int line will be set (rx\_int for wbuart-insert.v) and automatically cleared upon any read. To write, one can read from the transmit data register until the eighth bit, the tx\_busy bit, is cleared and transmitted. Alternatively, this negation of this bit may be connected to an interrupt line, o\_uart\_tx\_int. Writing to the port while the transmitter is idle will start transmitting. Writing to the port while it is busy will fill a one-word buffer that will get sent as soon as the port is idle for one clock.

# Registers

The wbuart core supports four registers, shown in Table 1. We will cover the format of all these registers here, as they are defined by wbuart.v.

## Setup Register

The setup register is the most critical of all the registers. It is shown in Figure 1. It is designed so that, for any 8N1 protocol (eight data bits, no parity, one stop bit, hardware flow control), all the upper bits will be set to zero so that only the number of clocks per baud interval needs to be set. The top bit is unused, making this a 31-bit number. The other field is H,which, when set, turns off any hardware flow control. *N* sets the number of bits per word. A zero value corresponds to 8-bit words, one to seven-bit words, and so forth, up to three for five-bit words. *S* determines the number of stop bits. Set this to one for two stop bits or leave it at zero for a single stop bit. *P* determines whether a parity bit is used (1 for parity, 0 for no parity), while *F* determines whether the parity is fixed. Table 2 lists how *P*, *F*, and *T* affect which parity is used.

The final portion of this register is the baud CLKS, which specifies the number of ticks your system clock has per baud interval.

Equation 2:

Rounding to the nearest integer is recommended. Hence, if you have a system clock of 100 MHz and wish to achieve 115,200 Baud, you will set CLKS to

Equation 3:

Table UART Registers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Address** | **Width** | **Access** | **Description** |
| SETUP | 2’b00 | 30 | R/W | UART configuration/setup register. |
| FIFO | 2’b01 | 32 | R | Returns size and status of the FIFOs |
| RX DATA | 2’b10 | 13 | R | Read data, read from the UART. |
| TX DATA | 2’b11 | 15 | R/W | Transmit data: writes send out the UART. |

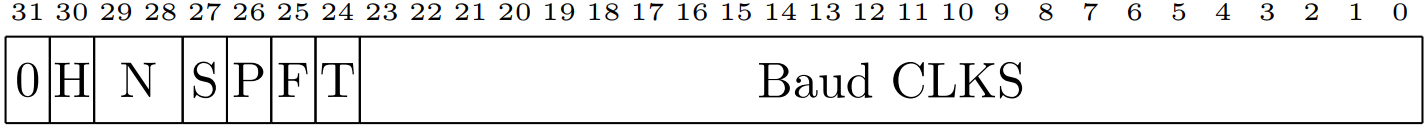


Figure SETUP Register fields.

Table Parity Setup

|  |  |  |  |
| --- | --- | --- | --- |
| **P** | **F** | **T** | **Setting** |
| 1 | 0 | 0 | Odd parity |
| 1 | 0 | 1 | Even parity |
| 1 | 1 | 0 | Parity bit is Space (1’b0) |
| 1 | 1 | 1 | Parity bit is Mark (1’b1) |
| 0 |  |  | No parity |

Changes to this setup register will occur in the transmitter as soon as it is idle and ready to accept another byte.

Changes to this setup register in rxuart.v also take place between bytes. However, within the wbuart.v context, any changes to the setup register will reset both the receiver and receive FIFO. Once reset, the receiver will insist on a minimum of sixteen idle baud intervals before receiving the next byte.

## FIFO Register

The FIFO register is a read-only register containing information about the status of both receive and transmit FIFOs within it. The transmit FIFO information is kept in the upper 16–bits, and the receiver FIFO information in the lower 1 bit, as shown in Figure 2. We will discuss each of these bits individually.

The LGLN field indicates the log base two of the FIFO length. Hence, an LGLN field of four would indicate a FIFO length of sixteen values. The FIFO fill for the transmitter indicates the number of available spaces within the transmit FIFO, while the FIFO fill in the receiver indicates the current number of spaces within the FIFO having valid data. The *H* bit will be true if the high-order FIFO fill bit is set. Finally, the *Z* bit will be true for transmitting if at least one open space in the FIFO and true in the receiver if at least one value needs to be read.

A close up of a piece of paper

Description automatically generated

Figure 2 RXDATA Register fields.

A white rectangular object with numbers

Description automatically generated

Figure 3 RXDATA Register fields.

A screenshot of a computer

Description automatically generated

Figure TXDATA Register fields

The *H* and *Z* bits also mirror the interrupt bits generated by wbuart.v. Interrupts will be generated whenever the FIFO is half full (on receive) or less than half full (on transmit). The same logic applies to the *Z* bit. An interrupt will be generated whenever the FIFO is non-empty (on receive) or not full (on transmit).

Writes to this FIFO status register are quietly ignored.

## RX DATA Register

Figure 3 breaks out the various bit fields of the receive data register used in wbuart.v. In particular, the *B* field indicates that the receiving line is in a break condition. The *F* and *P* fields indicate that a frame error or parity error has been detected. These bits are not self-clearing but are cleared by writing to 1’s to them. The *S* field will be false when the RWORD is valid. Hence, a word can be received without error if (RWORD & 0x0ff) is zero.

The *E* bit is an error bit. When set, it indicates that the FIFO has overflowed sometime since the last reset. This bit is also a reset bit. In other words, writing a 1’b1 to this bit will command a receive reset, clearing the FIFO and waiting for the line to be idle before receiving another byte. This bit is not implemented in wbuart-insert.v but exists in the wbuart.v implementation.

## TX DATA Register

Figure 4 breaks out the various bit fields of the transmit data register used in wbuart.v. The *C* field indicates whether the received data line is high or low, and the *O* field indicates the same for the transmit line. These are not particularly useful or valuable, but the *C* bit does not fit in the receive data register since it would violate the error condition detector. These two bits are thrown in here for whatever useful purpose one might find. The *B* field, when set, transmits a break condition.

Further, writing to the TXDATA register while in a break condition and with the B field clear will clear the transmitter from any break condition without transmitting anything. The *S* field is like the RXDATA strobe register. It is a read-only bit that will be true whenever the transmitter is busy. It will be clear only when the transmitter is idle. Finally, the upper *R* bit at the top of the register is the instantaneous value of the ready-to-send (RTS) value received.

The final three bits, *H*, *Z*, and *E*, are present only in wbuart.v. These bits indicate *H* if the FIFO is at least half full, *Z* if the FIFO is not full, and *E* if the FIFO has experienced an overflow condition since the last reset. Writing a 1’b1 to the *E* bit will reset the transmit FIFO, clearing any error indication in the FIFO and the FIFO.

To use the transmitter, write a byte to the TXDATA register with the upper 24–bits clear to transmit.

# Receiver’s Module (rxuart) State Machine

The rxuart module is a critical component of the WBUART32 core, responsible for receiving serial data in UART communication. It converts the serial input from the i\_uart\_rx line into parallel data that the system can process. Central to its operation is a finite state machine (FSM) that manages the reception process, including synchronization, data bit sampling, parity checking, and handling special conditions like breaks. This chapter provides a detailed description of the rxuart state machine, its states, and transitions. Figure 5 illustrates the flow of state machine.

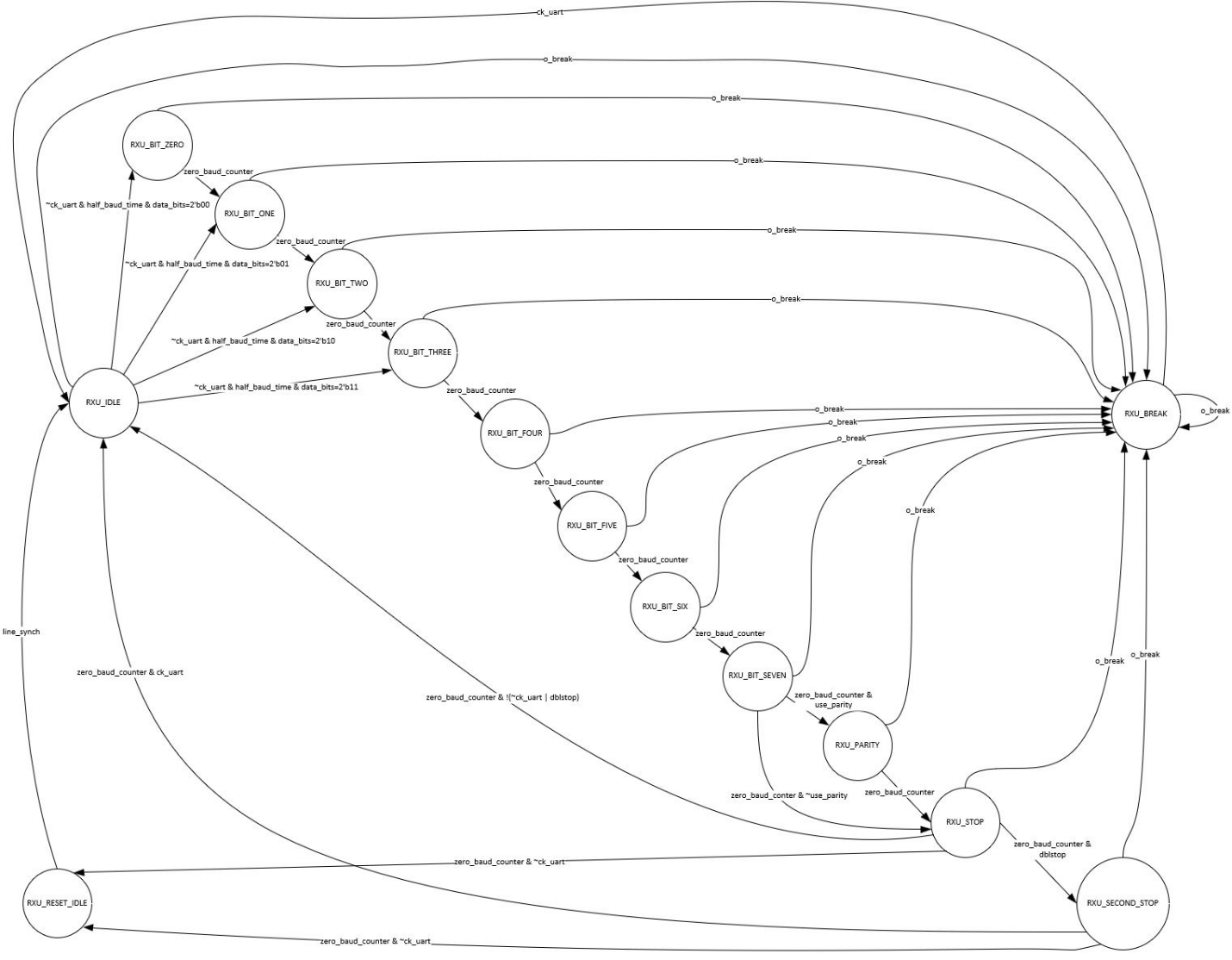


Figure Receiver's module (rxuart) state machine diagram.

## RXU\_RESET\_IDLE

* **Purpose:** Initialize the receiver and wait for line synchronization.
* **Actions:**
  + Resets internal counters and registers.
  + Monitors the line for an idle condition (line held high for a duration exceeding the break condition).
* **Transition Conditions:**
  + **From RXU\_RESET\_IDLE to RXU\_IDLE:** line\_synch is high (when line\_synch is high, indicate that the line is idle, and synchronization is achieved).
  + **From RXU\_RESET\_IDLE to RXU\_RESET\_IDLE:** line\_synch is low (if line\_synch remains low, wait for the line to become idle).

## RXU\_IDLE

* **Purpose:** Wait for the start bit indicating the beginning of a new data frame.
* **Actions:**
  + - Monitors the ck\_uart line for a falling edge (high to low transition), signaling a start bit.
    - When no start bit is detected, ck\_uart is high.
    - Ensures the transition occurs in the middle of a baud interval (half\_baud\_time).
* **Transition Conditions:**
  + - **From RXU\_IDLE to RXU\_BREAK:** o\_break is high (if a break condition is detected).
    - **From RXU\_IDLE to RXU\_BIT\_ZERO:** half\_baud\_time is high and ck\_uart is low and o\_break is low and data\_bits=2’b00 (transitions to the appropriate state to start sampling data bits after detecting a valid start bit, in the middle of a baud interval and out of break condition).
    - **From RXU\_IDLE to RXU\_BIT\_ONE:** half\_baud\_time is high and ck\_uart is low and o\_break is low and data\_bits=2’b01 (transitions to the appropriate state to start sampling data bits after detecting a valid start bit, in the middle of a baud interval and out of break condition).
    - **From RXU\_IDLE to RXU\_BIT\_TWO:** half\_baud\_time is high and ck\_uart is low and o\_break is low and data\_bits=2’b10 (transitions to the appropriate state to start sampling data bits after detecting a valid start bit, in the middle of a baud interval and out of break condition).
    - **From RXU\_IDLE to RXU\_BIT\_THREE:** half\_baud\_time is high and ck\_uart is low and o\_break is low and data\_bits=2’b11 (transitions to the appropriate state to start sampling data bits after detecting a valid start bit, in the middle of a baud interval and out of break condition).
    - **From RXU\_IDLE to RXU\_IDLE:** o\_break is low and ck\_uart is high or half\_baud\_time is low (if no start bit is detected and out of break condition).

## RXU\_BREAK

* **Purpose:** Handle a break condition where the line is held low for an extended period. This state ensures that the receiver properly handles this condition and does not process invalid data during the break.
* **Actions:**
  + Indicates a break condition to the system by setting o\_break to high.
  + Continuously monitors the ck\_uart line for changes.
  + Waits for the line to return high before resuming normal operation.
* **Transition Conditions:**
  + **From RXU\_BREAK to RXU\_IDLE:** ck\_uart is high (when ck\_uart is high, indicate the end of the break condition. At this point, the line is idle again, allowing the system to transition back to normal operation).
  + **From RXU\_BREAK to RXU\_BREAK:** ck\_uart is low (the system stays in its own state, maintaining the current state and continuing to monitor for line changes).

## RXU\_BIT\_ZERO

These states handle the sampling of each data bit in the serial input.

* **Purpose:** Sample data bits from the serial input.
* **Actions:**
  + For each data bit, wait for one baud interval (zero\_baud\_counter becomes high).
  + Samples the ck\_uart line at the center of the baud interval to capture the data bit.
  + Shifts the sampled bit into the data\_reg register.
* **Transition Conditions:**
  + **From RXU\_BIT\_ZERO to RXU\_BREAK:** o\_break is high (if a break condition is detected during data bit sampling).
  + **From RXU\_BIT\_ZERO to RXU\_BIT\_ONE:** zero\_baud\_counter is high and o\_break is low (after sampling the current bit, transitions to RXU\_BIT\_ONE state).
  + **From RXU\_BIT\_ZERO to RXU\_BIT\_ZERO:** o\_break is low and zero\_baud\_counter is low (the system remains in its own state).

## RXU\_BIT\_ONE

These states handle the sampling of each data bit in the serial input.

* **Purpose:** Sample data bits from the serial input.
* **Actions:**
  + For each data bit, wait for one baud interval (zero\_baud\_counter becomes high).
  + Samples the ck\_uart line at the center of the baud interval to capture the data bit.
  + Shifts the sampled bit into the data\_reg register.
* **Transition Conditions:**
  + **From RXU\_BIT\_ONE to RXU\_BREAK:** o\_break is high (if a break condition is detected during data bit sampling).
  + **From RXU\_BIT\_ONE to RXU\_BIT\_TWO:** zero\_baud\_counter is high and o\_break is low (after sampling the current bit, transitions to RXU\_BIT\_TWO state).
  + **From RXU\_BIT\_ONE to RXU\_BIT\_ONE:** o\_break is low and zero\_baud\_counter is low (the system remains in its own state).

## RXU\_BIT\_TWO

These states handle the sampling of each data bit in the serial input.

* **Purpose:** Sample data bits from the serial input.
* **Actions:**
  + For each data bit, wait for one baud interval (zero\_baud\_counter becomes high).
  + Samples the ck\_uart line at the center of the baud interval to capture the data bit.
  + Shifts the sampled bit into the data\_reg register.
* **Transition Conditions:**
  + **From RXU\_BIT\_TWO to RXU\_BREAK:** o\_break is high (if a break condition is detected during data bit sampling).
  + **From RXU\_BIT\_TWO to RXU\_BIT\_THREE:** zero\_baud\_counter is high and o\_break is low (after sampling the current bit, transitions to RXU\_BIT\_THREE state).
  + **From RXU\_BIT\_TWO to RXU\_BIT\_TWO:** o\_break is low and zero\_baud\_counter is low (the system remains in its own state).

## RXU\_BIT\_THREE

These states handle the sampling of each data bit in the serial input.

* **Purpose:** Sample data bits from the serial input.
* **Actions:**
  + For each data bit, wait for one baud interval (zero\_baud\_counter becomes high).
  + Samples the ck\_uart line at the center of the baud interval to capture the data bit.
  + Shifts the sampled bit into the data\_reg register.
* **Transition Conditions:**
  + **From RXU\_BIT\_THREE to RXU\_BREAK:** o\_break is high (if a break condition is detected during data bit sampling).
  + **From RXU\_BIT\_THREE to RXU\_BIT\_FOUR:** zero\_baud\_counter is high and o\_break is low (after sampling the current bit, transitions to RXU\_BIT\_FOUR state).
  + **From RXU\_BIT\_THREE to RXU\_BIT\_THREE:** o\_break is low and zero\_baud\_counter is low (the system remains in its own state).

## RXU\_BIT\_FOUR

These states handle the sampling of each data bit in the serial input.

* **Purpose:** Sample data bits from the serial input.
* **Actions:**
  + For each data bit, wait for one baud interval (zero\_baud\_counter becomes high).
  + Samples the ck\_uart line at the center of the baud interval to capture the data bit.
  + Shifts the sampled bit into the data\_reg register.
* **Transition Conditions:**
  + **From RXU\_BIT\_FOUR to RXU\_BREAK:** o\_break is high (if a break condition is detected during data bit sampling).
  + **From RXU\_BIT\_FOUR to RXU\_BIT\_FIVE:** zero\_baud\_counter is high and o\_break is low (after sampling the current bit, transitions to RXU\_BIT\_FIVE state).
  + **From RXU\_BIT\_FOUR to RXU\_BIT\_FOUR:** o\_break is low and zero\_baud\_counter is low (the system remains in its own state).

## RXU\_BIT\_FIVE

These states handle the sampling of each data bit in the serial input.

* **Purpose:** Sample data bits from the serial input.
* **Actions:**
  + For each data bit, wait for one baud interval (zero\_baud\_counter becomes high).
  + Samples the ck\_uart line at the center of the baud interval to capture the data bit.
  + Shifts the sampled bit into the data\_reg register.
* **Transition Conditions:**
  + **From RXU\_BIT\_FIVE to RXU\_BREAK:** o\_break is high (if a break condition is detected during data bit sampling).
  + **From RXU\_BIT\_FIVE to RXU\_BIT\_SIX:** zero\_baud\_counter is high and o\_break is low (after sampling the current bit, transitions to RXU\_BIT\_SIX state).
  + **From RXU\_BIT\_FIVE to RXU\_BIT\_FIVE:** o\_break is low and zero\_baud\_counter is low (the system remains in its own state).

## RXU\_BIT\_SIX

These states handle the sampling of each data bit in the serial input.

* **Purpose:** Sample data bits from the serial input.
* **Actions:**
  + For each data bit, wait for one baud interval (zero\_baud\_counter becomes high).
  + Samples the ck\_uart line at the center of the baud interval to capture the data bit.
  + Shifts the sampled bit into the data\_reg register.
* **Transition Conditions:**
  + **From RXU\_BIT\_SIX to RXU\_BREAK:** o\_break is high (if a break condition is detected during data bit sampling).
  + **From RXU\_BIT\_SIX to RXU\_BIT\_SEVEN:** zero\_baud\_counter is high and o\_break is low (after sampling the current bit, transitions to RXU\_BIT\_SEVEN state).
  + **From RXU\_BIT\_SIX to RXU\_BIT\_SIX:** o\_break is low and zero\_baud\_counter is low (the system remains in its own state).

## RXU\_BIT\_SEVEN

These states handle the sampling of the final data bit in the serial input stream.

* **Purpose:** Sample the 7th (most significant) data bit from the serial input.
* **Actions:**
  + Wait for one baud interval (zero\_baud\_counter becomes high).
  + Sample the ck\_uart line at the center of the baud interval to capture the 7th data bit.
  + Shifts the sampled bit into the data\_reg register.
  + After the 7th bit is sampled, if parity is enabled, the state transitions to RXU\_PARITY. If parity is not enabled, the state transitions to RXU\_STOP.
* **Transition Conditions:**
  + **From RXU\_BIT\_SEVEN to RXU\_BREAK:** o\_break is high (if a break condition is detected during data bit sampling).
  + **From RXU\_BIT\_SEVEN to RXU\_PARITY:** o\_break is low and zero\_baud\_counter is high and use\_parity is high (after the last data bit (7th bit) is sampled at RXU\_BIT\_SEVEN and parity is enabled).
  + **From RXU\_BIT\_SEVEN to RXU\_STOP:** o\_break is low and zero\_baud\_counter is high and use\_parity is low (after the last data bit (7th bit) is sampled at RXU\_BIT\_SEVEN and parity is not used).
  + **From RXU\_BIT\_SEVEN to RXU\_BIT\_SEVEN:** o\_break is low and zero\_baud\_counter is low (the system remains in its own state).

## RXU\_PARITY

* **Purpose:** Sample the parity bit if parity checking is enabled.
* **Actions:**
  + Waits for one baud interval (zero\_baud\_counter becomes high).
  + Samples the ck\_uart line to capture the parity bit.
  + Compares the sampled parity bit with the calculated parity (calc\_parity) to detect parity errors.
* **Transition Conditions:**
  + **From RXU\_PARITY to RXU\_BREAK:** o\_break is high (if a break condition is detected during parity bit sampling).
  + **From RXU\_PARITY to RXU\_STOP:** zero\_baud\_counter is high and o\_break is low (after parity bit sampling is complete, transitions to RXU\_STOP state).
  + **From RXU\_PARITY to RXU\_PARITY:** o\_break is low and zero\_baud\_counter is low (the system remains in its own state).

## RXU\_STOP

* **Purpose:** Sample the first stop bit.
* **Actions:**
  + Waits for one baud interval (zero\_baud\_counter becomes high).
  + Samples the ck\_uart line to verify the presence of a valid stop bit (should be high).
  + Detects framing errors if the stop bit is not high (ck\_uart is low).
* **Transition Conditions:**
  + **From RXU\_STOP to RXU\_BREAK:** o\_break is high (if a break condition is detected during stop bit sampling).
  + **From RXU\_STOP to RXU\_RESET\_IDLE:** zero\_baud\_counter is high and ck\_uart is low and o\_break is low (this indicates a framing error or an incomplete data frame, causing the system to reset to idle state for re-initialization).
  + **From RXU\_STOP to RXU\_SECOND\_STOP:** ck\_uart is high and dblstop is high and zero\_baud\_counter is high and o\_break is low (this transition happens when the receiver expects a second stop bit after the first).
  + **From RXU\_STOP to RXU\_IDLE:** zero\_baud\_counter is high and ck\_uart is high and dblstop is low and o\_break is low (this transition occurs when the single stop bit is successfully detected, and the system returns to the idle state to wait for the next frame).
  + **From RXU\_STOP to RXU\_STOP:** o\_break is low and zero\_baud\_counter is low (the system remains in its own state).

## RXU\_SECOND\_STOP

* **Purpose:** Sample the second stop bit when two stop bits are configured.
* **Actions:**
  + Similar to RXU\_STOP, waits for one baud interval and samples the ck\_uart line.
  + Verifies the presence of a valid second stop bit.
* **Transition Conditions:**
  + **From RXU\_SECOND\_STOP to RXU\_BREAK:** o\_break is high (if a break condition is detected during second stop bit sampling).
  + **From RXU\_SECOND\_STOP to RXU\_RESET\_IDLE:** zero\_baud\_counter is high and ck\_uart is low and o\_break is low (if a framing error is detected).
  + **From RXU\_SECOND\_STOP to RXU\_IDLE:** ck\_uart is high and zero\_baud\_counter is high and o\_break is low (if the second stop bit is valid).
  + **From RXU\_SECOND\_STOP to RXU\_SECOND\_STOP:** o\_break is low and zero\_baud\_counter is low (if none of the above conditions are met, the system remains in its own state, waiting for zero\_baud\_counter or a break condition to occur).

# Clock

The UART has been tested with a clock as fast as 200 MHz (Table 3). It should be able to use slower clocks, but only if it can properly set the baud rate, as shown in Equation 1.

I do not recommend using this core with a baud rate greater than a quarter of the system clock rate.

Table Clock Requirements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Source** | **Rates (MHz)** | | **Description** |
|  |  | **Max** | **Min** |  |
| i\_clk | (System) | 200 MHz |  | System clock |

# Clock Domains

In digital design, clock domains play a crucial role in defining how different components within a system synchronize and interact. In the case of the WBUART system, the clocking mechanism is relatively simple yet effective, as it operates primarily within a single clock domain. All modules namely wbuart, rxuart, txuart, and ufifo—are driven by a single input clock i\_clk. This common clock source propagates through all the connected sub-modules, ensuring synchronized operation throughout the UART system.

# Reset Domains

The UART system operates under a single reset domain, ensuring all modules namely rxuart, txuart, ufifo, and wbuart are reset simultaneously for synchronized operation. The rxuart and txuart modules use the i\_reset signal, while the ufifo and wbuart modules use the i\_rst signal. Despite the different names, these signals serve the same purpose, ensuring consistent reset behavior across the design. This unified reset ensures that the receiver, transmitter, FIFO, and Wishbone interface are initialized together, preventing data misalignment or corruption. The single reset domain supports reliable data transmission and enhances system stability, ensuring smooth error recovery.

# Wishbone Datasheet

Table 4 is required by the Wishbone specification to declare the core as Wishbone compliant.

Table Wishbone Datasheet

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Specification** | | |
| Revision level of Wishbone | WB B4 spec | | |
| Type of interface | Slave, Read/Write, pipeline reads supported | | |
| Port size | 32–bit | | |
| Port granularity | 32–bit | | |
| Maximum Operand Size | 32–bit | | |
| Data transfer ordering | (Irrelevant) | | |
| Clock constraints | None | | |
| Signal Names | **wbuart.v** | **wbuart-insert.v** | **WB Equivalent** |
| i\_clk | i\_wb\_clk | CLK\_I |
| i\_rst |  | RST\_I |
| i\_wb\_cyc | i\_wb\_cyc | CYC\_I |
| i\_wb\_stb | i\_wb\_stb | STB\_I |
| i\_wb\_we | i\_wb\_we | WE\_I |
| i\_wb\_addr | i\_wb\_addr | ADR\_I |
| i\_wb\_data | i\_wb\_data | DAT\_I |
| o\_wb\_ack | o\_wb\_ack | ACK\_O |
| o\_wb\_stall | o\_wb\_stall | STALL\_O |
| o\_wb\_data | o\_wb\_data | DAT\_O |

and so it is included here. It references the connections used in wbuart.v and those exemplified by wbuart-insert.v. The important thing to notice is that this core acts as a wishbone slave and that all accesses to the core registers are 32–bit reads and writes to this interface—not the 8–bit reads or writes that might be expected from any other 8-bit serial interface.

This table does not show that all accesses to the port take a single clock for wbuart-insert.v, or two clocks for wbuart.v. If the i\_wb\_stb line is high on one clock, the i\_wb\_ack line will be high the next for single clock access or the clock after that for two clock access. Further, the o\_wb\_stall line is tied to zero.

Also, this wishbone implementation assumes that if i\_wb\_stb, then i\_wb\_cyc will also be high. Hence, it only checks whether i\_wb\_stb is true to determine whether a transaction has taken place. If your bus does not meet this requirement, you must AND i\_wb\_stb with i\_wb\_cyc before using the core.

# Module Hierarchy

The UART design features the top-level module wbuart, which interfaces with the Wishbone bus to manage UART communication. Within wbuart, there are four key instances:

* Instance rx of module rxuart: Handles the reception of serial data, converting it into parallel format for the system.
* Instance rxfifo of module ufifo: Serves as a receive buffer (FIFO) that temporarily stores incoming parallel data from rxuart.
* Instance tx of module txuart: Manages the transmission of data by converting parallel data from the system into serial format for output.
* Instance txfifo of module ufifo: Acts as a transmit buffer (FIFO) that queues outgoing parallel data before it's sent by txuart.

This hierarchical structure allows for modular and efficient handling of UART communication, with ufifo being utilized for both transmitting and receiving data buffers to streamline the design.

# IO Ports

In its simplest form, the UART offers simply two IO ports: the i\_uart\_rx line to receive and the o\_uart\_tx line to transmit. These lines need to be brought outside of your design. Within Verilator, they need to be connected inside your Verilator test bench, as in:

if (!clk)

tb->i\_uart\_rx = uartsim(tb->ouart\_tx);

For those interested in hardware flow control, the core also offers an i\_cts\_n input to control the flow out of our transmitter and an o\_rts\_n output to indicate when the receiver is full. Both wires are active low.

There are three possibilities for those not interested in flow control. First, one can set the module parameter HARDWARE\_FLOW\_CONTROL\_PRESENT to zero, disabling hardware flow control and permanently setting the hardware flow control to off bit in the setup register. Second, hardware flow control can be disabled by connecting a 1’b0 wire to i\_cts\_n and ignoring the o\_rts\_n output. In this case, the hardware flow control setup bit becomes an unused flip-flop within the driver. The third way to disable hardware flow control is to disable it within the setup register. These approaches will only affect the transmitter’s operation and how the o\_rts\_n bit gets set.

A more detailed discussion of the connections associated with these modules can begin with Table 5, detailing the IO ports of synchronous FIFO; Table 6, detailing the IO ports of the UART receiver; Table 7, detailing the IO ports of the UART transmitter; and Table 8, detailing the IO ports of full UART module with wishbone interface.

Table ufifo module IO ports

|  |  |  |  |
| --- | --- | --- | --- |
| **Port** | **Width** | **Direction** | **Description** |
| i\_clk | 1 | Input | Clock |
| i\_rst | 1 | Input | Reset |
| i\_wr | 1 | Input | Write enable |
| i\_data | 8 | Input | Data input |
| o\_empty\_n | 1 | Output | Empty flag |
| i\_rd | 1 | Input | Read enable |
| o\_data | 8 | Output | Data output |
| o\_status | 16 | Output | Status register |
| o\_err | 1 | Output | Error flag |

Table rxuart module IO ports

|  |  |  |  |
| --- | --- | --- | --- |
| **Port** | **Width** | **Direction** | **Description** |
| i\_clk | 1 | Input | The system clock |
| i\_reset | 1 | Input | A positive, synchronous reset |
| i\_setup | 31 | Input | The 31–bit setup register |
| i\_uart\_rx | 1 | Input | The input wire from the outside world. |
| o\_wr | 1 | Output | True if a word was received. At this time, o data, o break, o\_parity\_err, and o\_frame\_err will also be valid. |
| o\_data | 8 | Output | The received data is valid if o\_wr |
| o\_break | 1 | Output | True in the case of a break condition |
| o\_parity\_err | 1 | Output | True if a parity error was detected |
| o\_frame\_err | 1 | Output | True if a frame error was detected |
| o\_ck\_uart | 1 | Output | A synchronized copy of the i\_uart |

Table txuart module IO ports

|  |  |  |  |
| --- | --- | --- | --- |
| **Port** | **Width** | **Direction** | **Description** |
| i\_clk | 1 | Input | The system clock |
| i\_reset | 1 | Input | A positive, synchronous reset |
| i\_setup | 31 | Input | The 31-bit setup register |
| i\_break | 1 | Input | Set to true to place the transmit channel into a break condition |
| i\_wr | 1 | Input | An input strobe. Set it to one when you wish to transmit data and clear it once it has been accepted. |
| i\_data | 8 | Input | The data to be transmitted is ignored unless  (i\_wr)&&(!o\_busy) |
| i\_cts\_n | 1 | Input | A hardware flow control wire, true if the transmitter is cleared to send, active low |
| o\_uart\_tx | 1 | Output | The wire to be connected to the external port |
| o\_busy | 1 | Output | True if the transmitter is busy, false if it will receive data |

Table wbuart module IO ports

|  |  |  |  |
| --- | --- | --- | --- |
| **Port** | **Width** | **Direction** | **Description** |
| i\_clk | 1 | Input | System clock input |
| i\_rst | 1 | Input | Synchronous reset signal |
| i\_wb\_cyc | 1 | Input | Wishbone bus cycle signal |
| i\_wb\_stb | 1 | Input | Wishbone bus strobe signal |
| i\_wb\_we | 1 | Input | Wishbone write enable signal |
| i\_wb\_addr | 2 | Input | Wishbone address bus |
| i\_wb\_data | 32 | Input | Wishbone data input bus |
| o\_wb\_ack | 1 | Output | Wishbone bus acknowledgment signal |
| o\_wb\_stall | 1 | Output | Wishbone bus stall signal |
| o\_wb\_data | 32 | Output | Wishbone data output bus |
| i\_uart\_rx | 1 | Input | The received wire coming from the external port |
| o\_uart\_tx | 1 | Output | The transmit wire to be connected to the external port |
| i\_cts\_n | 1 | Input | The hardware flow control clear-to-send input for the transmitter, active low |
| o\_rts\_n | 1 | Output | The hardware flow control ready-to-send (receive) output, also active low |
| o\_uart\_rx\_int | 1 | Output | True if a byte may be read from the receiver |
| o\_uart\_tx\_int | 1 | Output | True if a byte may be sent to the transmitter |
| o\_uart\_rxfifo\_int | 1 | Output | True if the receive FIFO is half full |
| o\_uart\_txfifo\_int | 1 | Output | True if the transmit FIFO is half empty |

# RTL Parameters

RTL parameters play a crucial role in defining the module’s behavior and configuration. These parameters provide a way to customize the operation of the WBUART for specific applications, allowing flexibility in setting communication protocols, buffer sizes, and flow control options. By adjusting the default values of these parameters, designers can tailor the WBUART to work with different baud rates and data formats without the need for extensive modifications to the core logic. This flexibility ensures that the WBUART can be adapted to various hardware environments and use cases, making the design versatile and scalable for a wide range of communication needs. The following tables describe all the default RTL parameters that can be found in all modules.

Table ufifo module default RTL parameters

|  |  |  |
| --- | --- | --- |
| **RTL Parameter Name** | **Default RTL Parameter Value** | **Description** |
| BW | 8 | Specifies the byte/data width for the FIFO. This parameter defines the size of each data word handled by the FIFO. In this case, each data word is 8 bits (1 byte). |
| LGFLEN | 4 | Determines the log base 2 of the FIFO length. This value defines the total number of entries in the FIFO as 2^LGFLEN, which means the FIFO can hold 16 entries. |
| RXFIFO | 1'b0 | Indicates whether the FIFO is configured as a receiver FIFO (1'b0 for disabled or 1'b1 for enabled). In this case, the receiver FIFO is disabled. |

Table rxuart module default RTL parameters

|  |  |  |
| --- | --- | --- |
| **RTL Parameter Name** | **Default RTL Parameter Value** | **Description** |
| INITIAL\_SETUP | 31'd868 | Configures key communication settings such as baud rate, data bits, stop bits, and parity. In the case of the 31’d868 configuration, it specifies that 868 clock cycles are used per baud, setting the baud rate to approximately 115,200 bps with a 100 MHz clock, along with specific settings for data and parity bits. |

Table txuart module default RTL parameters

|  |  |  |
| --- | --- | --- |
| **RTL Parameter Name** | **Default RTL Parameter Value** | **Description** |
| INITIAL\_SETUP | 31'd868 | Configures key communication settings such as baud rate, data bits, stop bits, and parity. In the case of the 31’d868 configuration, it specifies that 868 clock cycles are used per baud, setting the baud rate to approximately 115,200 bps with a 100 MHz clock, along with specific settings for data and parity bits. |

Table wbuart module default RTL parameters

|  |  |  |
| --- | --- | --- |
| **RTL Parameter Name** | **Default RTL Parameter Value** | **Description** |
| INITIAL\_SETUP | 31'd25 | Configures the baud rate, data bits, stop bits, parity, and flow control. With 31'd25, it sets a 4 megabaud rate using a 100 MHz clock in 8N1 mode which means of 8 data bits, no parity, and 1 stop bit. |
| LGFLEN | 4 | Determines the log base 2 of the FIFO length. This value defines the total number of entries in the FIFO as 2^LGFLEN, which means the FIFO can hold 16 entries. |
| HARDWARE\_FLOW\_CONTROL\_PRESENT | 1'b1 | Controls whether hardware flow control is used. When set to 1, it enables flow control using i\_cts\_n (Clear to Send) and o\_rts\_n (Ready to Send) signals to manage data flow between devices. If set to 0, hardware flow control is disabled, and the UART operates without using these signals. |