WBUART32 SPECIFICATION

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

February 20, 2017

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 MERCHANTIBILITY 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 along with this program. If not, see `http://www.gnu.org/licenses/` for a copy.

Revision History

Rev.	Date	Author	Description
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

Contents

Page

1 Introduction 1

2 Architecture 2

3 Operation 4

4 Registers 6
4.1 Setup Register 6
4.2 FIFO Register 7
4.3 RX_DATA Register 8
4.4 TX_DATA Register 8

5 Clocks 10

6 Wishbone Datasheet 11

7 I/O Ports 13

Figures

Figure Page

4.1. SETUP Register fields . . . 7
4.2. RXDATA Register fields . . . 7
4.3. RXDATA Register fields . . . 8
4.4. TXDATA Register fields . . . 8

Tables

Table Page

4.1. UARTRegisters . . . 6
4.2. Parity setup . . . 7

5.1. Clock Requirements . . . 10

6.1. Wishbone Datasheet . . . 11

7.1. RXUART port list . . . 14
7.2. TXUART port list . . . 14
7.3. WBUART port list . . . 15

Preface

It may be that building a UART is a mandatory coming of age task for any HDL designer. The task is simple, easy, and there's not all that much to it. This project comes out of some of my first experiences with Verilog.

Since then, it has been augmented with quite a few useful capabilities for simulating a UART connection when using Verilator. It is this, perhaps unusual, addition to the core set that makes this core worth taking note of.

I hope you find it useful.

Dan Gisselquist, Ph.D.

1. Introduction

The Universal Asynchronous Serial Transport, or UART, has become quite the 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 particular UART implementation offers something many of these other examples do not: a Verilator simulation capability. This will allow 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 yes it can truly run Linux–all within Verilator.

As a final addition, there are four files in the test bench section which can be used as top–level design files to prove whether or not the serial port on a given circuit board works.

2. Architecture

The HDL portion of the core itself consists of four basic files: `rxuart.v`, `txuart.v`, `ufifo.v` and `wbuart.v`. These are, respectively, the receive UART code, the transmit UART code, a fairly generic FIFO, and a fully wishbone compliant UART peripheral. This latter file demonstrates one example of 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 into a module implementing a simpler wishbone interface without the FIFO.

Each of the core files, `rxuart.v` and `txuart.v`, are fully capable. They 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 or not 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 Chap.4.

A further note on the `rxuart.v` module is in order. This module double latches the input, in the proper two buffer fashion to avoid problems with metastability. Then, upon the detection of the start bit (i.e. a high to low transition), the port waits a half of a baud, and then starts its baud clock so as 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 still lie within the associated bit period. This leaves us with the criteria that,

$$\left|(N+2)\left(\frac{f_{\mathrm{SYS}}}{f_{\mathrm{BAUD}}}-\mathtt{CKS}\right)\right| \quad < \quad \frac{f_{\mathrm{SYS}}}{2f_{\mathrm{BAUD}}},. \tag{2.1}$$

where f_{SYS} is the system clock frequency, f_{BAUD} 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. What this means is that, for transmission rates where f_{BAUD} approaches f_{SYS}, the number of data rates that can actually 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 was 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 as opposed to inputs. When the strobe is high, the data is valid. It will only be high for 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 whether or not there was a framing error (i.e., no stop bit), and `o_parity_err` will indicate whether or not 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. In a similar fashion, the `rx_stb` line, or the bus equivalent of `rx_ready`, may be used for receive interrupt lines–although it will need to be latched as both `wbuart.v` and `wbuart-insert.v` demonstrate.

An simple example of how to put this configuration together is found in `wbuart-insert.v`. In this example given, 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. In a similar fashion, the `tx_data` register can be written to with a byte in order 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 be used to determine if the transmitter is busy (via polling), whether it is currently in a break condition, or even what bit is currently 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, as well as 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 be used to 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, there are a series of example files found in the bench/verilog directory. `helloworld.v` presents an example of a simple UART transmitter sending the "Hello, World \r\n" message over and over again. 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. This tests both `txuart.v` and `rxuart.v`. A third test file, `linetest.v`, works by waiting for a line of data to be received, after which it parrots that line back to the terminal. A fourth test file, `speechfifo.v` tests both the wishbone interface as well as the FIFO, by filling the UART, 10 samples at a time, with text from Abraham Lincoln's Gettysburg address. All three of these example files may be used as stand-alone top–level design files to verify your own UART hardware functionality.

3. Operation

To use the core, a couple of steps are required. First, wire it up. This includes wiring the i_uart and o_uart ports, as well as any i_rts and/or o_cts 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 inwbuart-insert.v. Alternatively, the wbuart.v file may be connected to a straight 32–bit wishbone bus. Second, set the UART configuration register. This is ideally set by setting the INITIAL_SETUP parameter of rxuart, txuart or even wbuart Alternatively, you can write to the setup register at a later time, as is done within the speechfifo.v bench test.

From a simulation standpoint, it will also need to be "wired" up inside your C++ main Verilator file. Somewhere, internal to the top–level Verilator C++ simulation file, you'll want to have some setup lines similar to,

```
#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 shouldn't be needed for normal usages of these techniques, but you may find it helpful to know should you examine this code or should you wish to build your own test file that proves its own 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. This is an indication that the

byte is valid. Alternatively, you could wait until the an interrupt line is set and then 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 then transmit. 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 it 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.

4.

Registers

The `wbuart` core supports four registers, shown in Tbl. 4.1. We'll cover the format of all of these registers here, as they are defined by `wbuart.v`.

4.1 Setup Register

The setup register is perhaps the most critical of all the registers. This is shown in Fig.4.1. It is designed so that, for any 8N1 protocol (eight data bits, no parity, one stop bit, hardware flow control on), all of 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 fields are: H which, when set, turns off any hardware flow control. N sets the number of bits per word. A value of zero corresponds to 8–bit words, a value of one to seven bit words, and so forth up to a value of 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 or not a parity bit is used (1 for parity, 0 for no parity), while F determines whether or not the parity is fixed. Tbl. 4.2 lists out the various values possible here.

The final portion of this register is the baud `CLKS`. This is the number of ticks of your sytem clock per baud interval,

$$\texttt{CLKS} = \frac{f_{\text{SYS}}}{f_{\text{BAUD}}}.$$

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 would set `CLKS` to

$$\texttt{CLKS}_{\texttt{Example}} = \frac{100 \cdot 10^6}{115200} \frac{\text{Clocks per Second}}{\text{Baud Intervals per Second}} \approx 868 \text{ Clocks per Baud Interval}$$

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, reads from the UART.
`TX_DATA`	2'b11	15	(R/)W	Transmit data: writes send out the UART.

Table 4.1: UART Registers

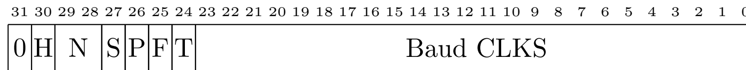

Figure 4.1: SETUP Register fields

P	F	T	Setting
1	0	0	Odd parity
1	0	1	Even parity
1	1	0	Parity bit is a Space (1'b0)
1	1	1	Parity bit is a Mark (1'b1)
0			No parity

Table 4.2: Parity setup

Changes to this setup register will take place in the transmitter as soon as the transmitter 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 also reset the receiver and receive FIFO together. Once reset, the receiver will insist on a minimum of sixteen idle baud intervals before receiving the next byte.

4.2 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-bits, as shown in Fig. 4.2. We'll 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 the transmitter if there is at least one open space in the FIFO, and true in the receiver if there is at least one value needing to be read.

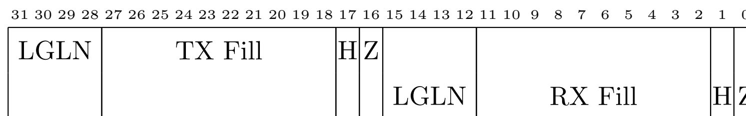

Figure 4.2: RXDATA Register fields

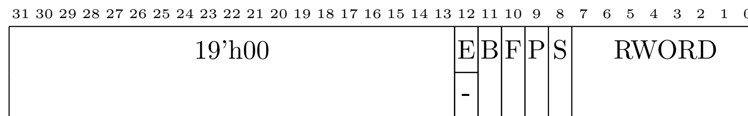

Figure 4.3: RXDATA Register fields

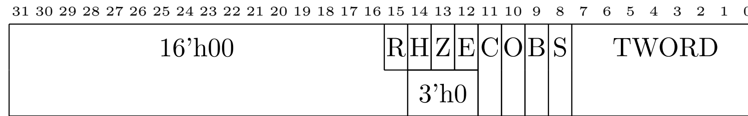

Figure 4.4: TXDATA Register fields

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

Writes to this FIFO status register are quietly ignored.

4.3 RX_DATA Register

Fig. 4.3 breaks out the various bit fields of the receive data register used in `wbuart.v`. In particular, the *B* field indicates that the receive 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 rather are cleared by writing to 1's to them. The *S* field will be false when the `RWORD` is valid. Hence, if `(RWORD & 0x0ff)` is zero there is a word ready to be received without error.

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.

4.4 TX_DATA Register

Fig. 4.4 breaks out the various bit fields of the transmit data register used in `wbuart.v`. The *C* field indicates whether or not the receive data line is high or low, the *O* field indicates the same for the transmit line. These aren't particularly useful or valuable, but the *C* bit doesn't 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, writes 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 similar to the RXDATA strobe register. It is a read–only bit that will be true any time 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 received ready-to-send (RTS) value.

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, both clearing any error indication in the FIFO as well as clearing the FIFO itself.

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

5. Clocks

The UART has been tested with a clock as fast as 200 MHz (Tbl. 5.1). It should be able to use slower clocks, but only subject to the ability to properly set the baud rate as shown in Eqn. (2.1) on Page 2.

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

Name	Source	Rates (MHz)		Description
		Max	Min	
`i_clk`	(System)	200 MHz		System clock

Table 5.1: Clock Requirements

6. Wishbone Datasheet

Tbl. 6.1 is required by the wishbone specification in order to declare the core as wishbone compliant,

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`

Table 6.1: Wishbone Datasheet

and so it is included here. It references the connections used in `wbuart.v` as well as those exemplified by `wbuart-insert.v`. The big 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.

What this table doesn't show is that all accesses to the port take a single clock for `wbuart-insert.v`, or two clocks for `wbuart.v`. That is, 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 particular wishbone implementation assumes that if `i_wb_stb`, then `i_wb_cyc` will be high as well. Hence it only checks whether or not `i_wb_stb` is true to determine if a transaction

has taken place. If your bus does not meet this requirement, you'll need to AND `i_wb_stb` with `i_wb_cyc` before using the core.

7. I/O Ports

In it's simplest form, the UART offers simply two I/O ports: the i_uart_rx line to receive, and the o_uart_tx line to transmit. These lines need to be brought to the 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->o_uart_tx);
```

For those interested in hardware flow control, the core also offers an i_rts input to control the flow out of our transmitter, and an o_cts output when the receiver is full.

A more detailed discussion of the connections associated with these modules can begin with Tbl. 7.1, detailing the I/O ports of the UART receiver, Tbl. 7.2, detailing the I/O ports of the UART transmitter, and Tbl. 7.3 detailing the non–wishbone I/O ports of the wishbone controller.

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	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, 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 i_uart

Table 7.1: RXUART port list

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 to one when you wish to transmit data, clear once it has been accepted
i_data	8	Input	The data to be transmitted, ignored unless (i_wr)&&(!o_busy)
i_rts	1	Input	A hardware flow control wire, true if the receiver is ready to receive
o_uart	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 7.2: TXUART port list

Port	W	Direction	Description
i_uart_rx	1	Input	The receive wire coming from the external port
o_uart_tx	1	Output	The transmit wire to be connected to the external port
i_rts	1	Input	The hardware flow control ready-to-send (i.e. receive) input for the transmitter
o_cts	1	Output	The hardware flow control clear-to-send output
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

Table 7.3: WBUART port list