Gisselquist
Technology, LLC

WBUART32 SPECIFICATION

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

August 26, 2016

Revision History

Rev.	Date	Author	Description
0.1	8/26/2016	D. Gisselquist	Initial Draft Specification

Contents

Figures

Figure Page

Tables

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 a very useful capability 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.

2.

Architecture

The HDL portion of the core itself consists of three files: `rxuart.v`, `txuart.v`, and `wbuart-insert.v`. These are, respectively, the receive UART code, the transmit UART code, and an example of how the receiver and transmitter may be connected to a Wishbone bus.

Each of the core files, `rxuart.v` and `txuart.v`, are fully capable. They each accept a 29–bit setup value specifying baud rate, the number of bits per byte (between 5 and 8), whether or not parity is used, 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,

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

where $f_{\rm SYS}$ is the system clock frequency, $f_{\rm BAUD}$ 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 $f_{\rm BAUD}$ approaches $f_{\rm SYS}$, 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 as soon as the strobe is asserted and the busy line is not. 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 port 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. Also, while the strobe is high, the `o_break` line will indicate whether the receiver is in a "break" state, `o_frame_err` will indicate whether or not there was a framing error (i.e., no stop bit), and `o_parity_err` will indicate wheher or not the parity matched.

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.

An 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 nine will place the transmitter into a "break" condition, only 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.

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. Once connected, using this 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.

3.

Operation

To use the core, a couple of steps are required. First, wire it up. The `wbuart-insert.v` file should provide a good example of how to wire it up. Second, set the UART configuration register. This is ideally set in an initial statement within the code somewhere, but can easily be set elsewhere by writing to this register from the bus.

From a simulation standpoint, it will also need to be wired up. Somewhere, internal to the top–level Verilator C++ simulation file, you'll want to have a line similar to,

```
if (!clk)
        tb->i_rx =  uartsim(tb->o_uart, setup);
```

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 no bits but 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, the `rx_int` line will be set, 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. Writing to the port while 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 core really only has one register associated with it, which is the setup register. The format of this register is important, although not necessarily trivial or obvious. We'll cover two other registers here, though, associated with the example wishbone connections from `wbuart-insert.v`. All three of these registers are shown in Tbl. 4.1.

Since the connections presented are only examples, they are listed without addresses, as their wishbone bus connectivity will be determined once they are connected.

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), only the number of clocks per baud interval needs to be set. The top two bits are unused, making this a 30–bit number. The other fields are: 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 clear for a single stop bit. P determines whether or not a parity bit exists (1 for parity, 0 for none), while F determines whether or not the parity is fixed. Tbl. **??** lists out the various values possible here.

4.2 RX_DATA Register

Fig. 4.2 breaks out the various bit fields of the receive data register used in the `wbuart-insert.v` example of connecting it to a bus. 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 were detected. These are valid like the data word: when the strobe line is set. 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.

Name	Address	Width	Access	Description
SETUP		30	R/W	UART configuration/setup register.
RX_DATA		12	R(/W)	Read data, reads from the UART.
TX_DATA		12	(R/)W	Transmit data: writes send out the UART.

Table 4.1: UART Registers

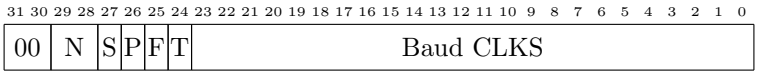

Figure 4.1: SETUP Register fields

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

4.3 TX_DATA Register

Fig. 4.3 breaks out the various bit fields of the transmit data register used in `wbuart-insert.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 they don't fit in the receive data register since they would violate the error condition detector. They're thrown in here for whatever useful purpose one might find. The *B* field, when set, sends a break condition down the wire. Writing to the TXDATA register, clearing the *B* field, will clear the transmitter from the break condition without transmitting anything. The *S* field is similar to the RXDATA strobe register. It will be true whenever the transmitter is busy or a byte is waiting for it. It will be clear only when the transmitter is idle.

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

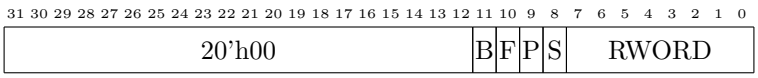

Figure 4.2: RXDATA Register fields

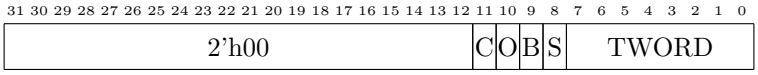

Figure 4.3: TXDATA Register fields

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	Signal Name / Wishbone Equivalent i_wb_clk / CLK_I i_wb_cyc / CYC_I i_wb_stb / STB_I i_wb_we / WE_I i_wb_addr / ADR_I i_wb_data / DAT_I o_wb_ack / ACK_O o_wb_stall / STALL_O o_wb_data / DAT_O

Table 6.1: Wishbone Datasheet

and so it is included here. It references the connections 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.

What this table doesn't show is that all accesses to the port take a single clock. That is, if the `i_wb_stb` line is high on one clock, the `i_wb_ack` line will be high the next. 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_rx` line to receive, and the `o_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_rx =  uartsim(tb->o_uart, setup);
```

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, and Tbl. 7.2, detailing the I/O ports of the UART transmitter.

The "ports" associated with the `wbuart-insert.v` example may be inferred from the wishbone data sheet.

Port	Width	Direction	Description
i_clk	1	Input	The system clock
i_reset	1	Input	A positive, synchronous reset
i_setup	30	Input	The 30–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	30	Input	The 30–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)
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