**Defining the UART**

The use of hardware description languages (HDLs) is becoming increasingly common for designing and verifying FPGA designs. Behavior level description not only increases design productivity, but also provides unique advantages for design verification. The most dominant HDLs today are Verilog and VHDL. This application note illustrates the use of Verilog in the design and verification of a digital UART (**U**niversal **A**synchronous **R**eceiver & **T**ransmitter).

The UART consists of two independent HDL modules. One module implements the transmitter, while the other module implements the receiver. The transmitter and receiver modules can be combined at the top level of the design, for any combination of transmitter and receiver channels required. Data can be written to the transmitter and read out from the receiver, all through a single 8 bit bi-directional CPU interface. Address mapping for the transmitter and receiver channels can easily be built into the interface at the top level of the design. Both modules share a common master clock called mclkx16. Within each module, mclkx16 is divided down to independent baud rate clocks.
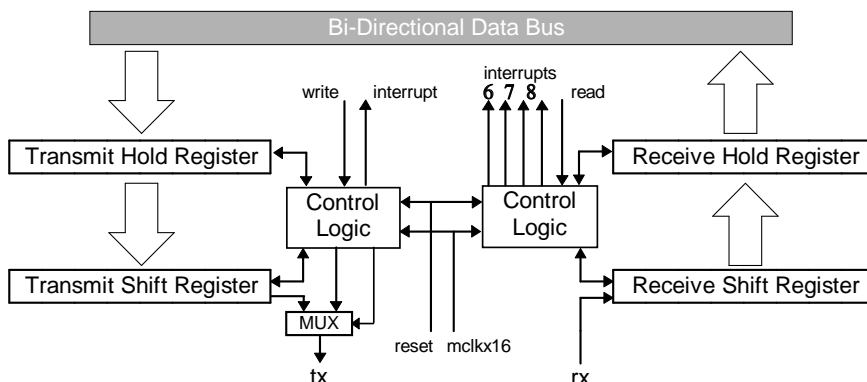
FIGURE 1
Digital UART

QUICKLOGIC

8
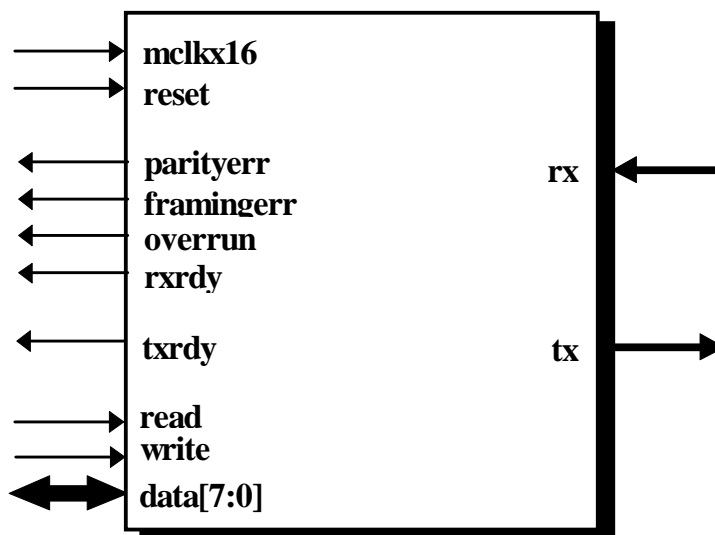
Features of the UART	Supported
Full duplex operation	✓
Standard UART data format	✓
Even or Odd parity mode	✓
Parity error check	✓
Framing error check	✓
Overrun error check	✓
Receive data ready interrupt	✓
Transmit data ready interrupt	✓

UART functional overview

A basic overview of the UART is shown below. At the left hand side is shown the "transmit hold register", "transmit shift register" and the transmitter "control logic" blocks, all contained within the transmitter module called "txmit". At the right hand side is shown the "receive shift register", "receive hold register" and the receiver "control logic" blocks, all contained within the receiver module called "rxcver". The two modules have separate inputs and outputs for most of their control lines, only the bi-directional data bus, master clock and reset lines are shared by both modules. Smaller functions, such as combinatorial logic synthesized from HDLs, are mapped in parallel into independent fragments, providing high gate utilization without sacrificing performance. Related and unrelated functions can be packed into the same logic cell, increasing effective density and gate utilization.

FIGURE 1 UART block diagram

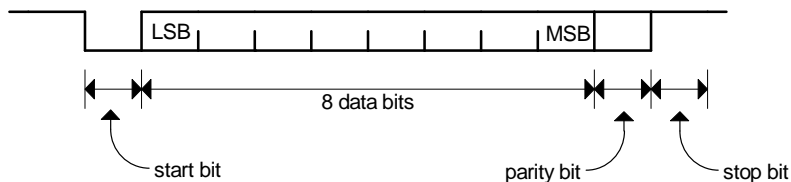| SYMBOL	TYPE	DESCRIPTION
mclkx16	input	Master input clock used for internal baud rate generation.
reset	input	Master reset input signal.
parityerr	output	Indicates whether or not a parity error was detected during receive of a data frame. Encoding can be based on Even or Odd parity mode.
framingerr	output	Indicates if the serial data format sent to the rx input did not match the proper UART data format shown in Figure 2.
overrun	output	Indicates when new data are sent to the receiver, but data previously received is still held internally by the receiver, and has not yet been read out.
rxrdy	output	Indicates that new data has been received, and are ready to be read out.
txrdy	output	Indicates that new data can be written to the transmitter.
read	input	Active low read strobe signal, used for reading out data from the receiver.
write	input	Active low write strobe signal, used for writing data to the transmitter.
data[7:0]	inout	Bi-directional data bus. Data to be transmitted, or data that has been received, are transferred through this data bus.
tx	output	Transmitter serial output. tx will be held high during reset, or when no transmissions are taking place.
rx	input	Receiver serial input. rx should be held high (pulled-up), when no transmissions are taking place.

**TABLE 1
I/O Functions
of the UART**

Figure 3 shows the UART serial data format. Serial data are contained within frames of 8 data bits, as well as coded information bits. Between successive transmissions, the transmission line is held high. A transmission is initialized by a leading low start bit. Next to the leading low start bit comes 8 bits of data information, beginning with the LSB and ending with the MSB. After the 8 data bits comes the parity bit, representing the parity result of the 8 data bits. The parity bit can be set true based on even parity or odd parity mode. Next to the parity bit comes a trailing high stop bit indicating the end of a data frame.

**The UART standard
data format**

8

**Application
Notes**

QUICKLOGIC

**FIGURE 3
Digital UART
data format**

**UART timing
diagrams**

Figure 4 below shows how data written to the "transmit hold register" gets loaded into the "transmit shift register", and at the rising edge of the baud rate clock, shifted to the tx output.

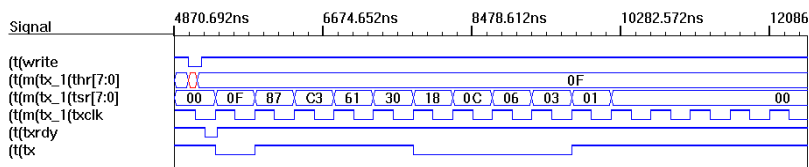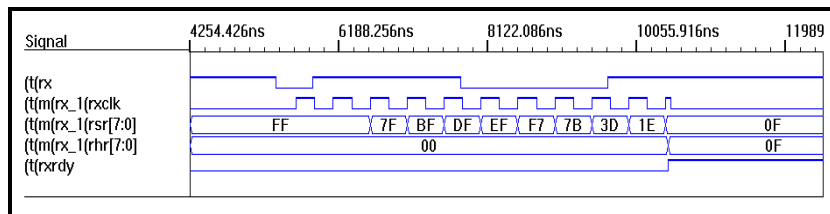**FIGURE 4
Transmitter Timing**

Figure 5 below shows how data gets shifted from the rx input to the "receive shift register", and afterwards loaded into the "receive hold register". Finally the receiver raises the"rxrdy" flag.

**FIGURE 5
Receiver Timing**

**The Transmitter
module**

The master clock called mclkx16 is divided down to the proper baud rate (equal to mclkx16/16) and is then called txclk. Data written in parallel format to the module is latched internally, and shifted in serial format to the tx output at the frequency of the baud rate clock. Data shifted to the tx output follows the UART data format shown in Figure 3.

SYMBOL	TYPE	DESCRIPTION
mclkx16	input	Master input clock used for internal baud rate generation.
reset	input	Master reset input signal.
write	input	Active low write strobe signal, used for writing data to the transmitter.
data[7:0]	input	Data to be transmitted is written to the transmitter through the data bus. At the rising edge of the write strobe, the contents of the data bus are latched into an internal "transmit hold register".
tx	output	Serial data output. Serial data frames are transmitted via this pin. The tx output will be held high during reset, or when no transmissions are taking place.
txrdy	output	Indicates that the data latched into the "transmit hold register" now have been successfully loaded into the "transmit shift register", and the "transmit hold register" again is ready for new data to be written.

TABLE 2
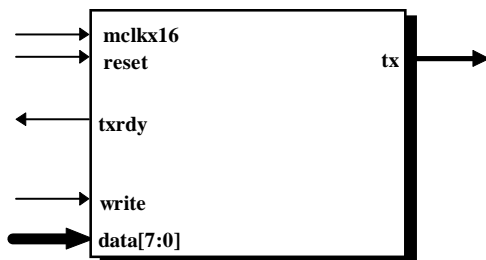**I/O Functions of the transmitter module**
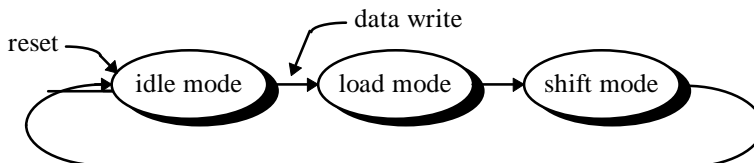

FIGURE 6
Transmitter module

8

Application
Notes

QUICKLOGIC

SYMBOL	TYPE	INTERNAL FLAG & SIGNAL DESCRIPTION
thr	reg[7:0]	8 bit "transmit hold register" used to hold the contents of the data bus, when new data is written to the module.
tsr	reg[7:0]	8 bit "transmit shift register", used for shifting out the contents of thr.
paritymode	wire	The value of this bit indicates the parity mode of the transmitter. Initialized to 1 => Odd parity mode. Initialized to 0 => Even parity mode.
txparity	reg	Signal that holds the parity result of the tsr.
tag1, tag2	reg	Tag bits used for decoding the state of the transmission.
txclk	reg	Baud rate clock used to shift the contents of the tsr register to the tx output.
txdone	wire	txdone is true when a transmit sequence has been completed.
txdatardy	reg	txdatardy is true when "transmit hold register" holds new data that is ready to be transmitted.
paritycycle	wire	Indicates the state of the transmission, when the tx output gets the value of txparity.
cnt	reg[3:0]	Count register used for dividing the clkx16 down to txclk.

Behavioral description
of the transmitter

The transmitter waits for new data to be written to the module. When new data is written a transmit sequence is initialized. Data that was written in parallel to the module gets transmitted as serial data frames at the tx output. When no transmit sequences are in place, the tx output is held high.

FIGURE 7
Sequence of
transmitter

MODE	DESCRIPTION
idle	Active high reset forces the transmitter into idle mode, and the tx output is held high. The transmitter is waiting for new data to be written to thr. If txdone and txdatardy both are true, the transmitter enters the load mode.
load	The contents of the "transmit hold register" are loaded into the "transmit shift register". Tag bits are concatenated to the "transmit shift register" for monitoring the sequence of transmission. The low start bit is asserted to the tx output. The signal txrdy indicates that the "transmit shift register" has been successfully loaded, and the "transmit hold register" again is ready for new data to be written.
shift	Data is transmitted from tsr to the tx output at the rising edge of txclk. Generation of the parity bit value takes place during the transmission. During shifting of data, the "transmit shift register" is "zero filled". Decoding of tsr and the tag bits determines the state of transmission: • "shifting data" => tx gets tsr[0] • "parity cycle" => tx gets txparity • "txdone" => tx gets stop bit

**TABLE 4
Behavioral modes
of transmitter
module**

Creating logic in Verilog starts with the module port declaration. This section defines signals that are ported to and from the module. No directions are specified at this point.

**Implementation
of the
transmitter**

```
module txmit (mclkx16, write, reset, tx, txrdy, data);
```

Next to port definitions comes port directions. Directions are specified as input, output or inout (bi-directional), and can be referred to in Table 1. Next to the specification of port directions comes the declaration of internal signals. Internal signals in Verilog are declared as "wire" or "reg" data types. Signals of the "wire" type are used for continuous assignments, also called combinatorial statements. Signals of the "reg" type are used for assignments within the Verilog "always" block, often use for sequential logic assignments, but not necessarily. For further explanation see a Verilog reference book. Data types of the internal signals of the module can be found in Table 3.

**FIGURE 8
Transmitter module
declaration**

8

**Application
Notes**

QUICKLOGIC

We have now covered all necessary declarations, and are ready to look at the actual implementation. Using a hardware description language allows us to describe the function of the transmitter in a more behavioral manner, rather than focus on its actual implementation at the gate level

In software programming languages, functions and procedures break larger programs into more readable, manageable and maintainable pieces. The Verilog language provides functions and tasks as constructs, analogous to software functions and procedures. A Verilog function and task are used as the equivalent to multiple lines of Verilog code, where certain inputs or signals affect certain outputs or variables. The use of functions and tasks usually takes place where multiple lines of code are repeatedly used in a design, and hence makes the design easier to read and maintain. A Verilog function can have multiple inputs, but always only one output, while the Verilog task can have both multiple inputs and multiple outputs and even in some cases, neither. Below is shown the Verilog task that includes all necessary sequential statements to describe the transmitter in the "shift" mode.

FIGURE 9
Transmitter Shift
Mode

```
task shift_data;
   begin
      tsr     <= tsr >> 1;           // Right shift tsr by one.
      tsr[7]  <= tag1;               // Set tsr[7] = tag1.
      tag1    <= tag2;               // Set tag1 = tag2.
      tag2    <= 1'b0;               // Set tag2 = 0.
      txparity  <= txparity ^ tsr[0];  // Generate parity.
   end
endtask
```

Here we see the two tag bits called tag1 and tag2 concatenated to the transmit shift register. Similar tasks were created to describe the transmitter in "idle" and "load" modes. By using these Verilog tasks, we can now create a very "easy to read" behavioral model of the whole transmit process.

If txdone and txdatardy both are true, the transmitter enters the load mode. After the load mode, the transmitter enters the shift mode. At the rising edge of the baud rate clock, the contents of tsr are shifted to the tx output. Parity generation takes place during shifting of the tsr, as shown in Figure 10.

```
        always @(posedge txclk or posedge reset)
        if (reset)
                idle_reset;                       // Reset internal bits of transmitter.
        else
                begin
                if (txdone && txdatardy)
                        load_data;                // Load new data to tsr.
                else
                        begin
                        shift_data;               // Shift contents of tsr to tx output.

                        // Shift out data or parity bit or stop/idle bit.
                        if (txdone )
                                tx <= 1'b1;       // Output stop/idle bit.
                        else if (paritycycle)
                                tx <= txparity;   // Output parity bit.
                        else
                                tx <= tsr[0];     //Shift out data bit.
                        end
                end
```

It's important to note, that the tsr is "zero filled" during transmission. The combination of the two trailing tag bits and the zero filled tsr indicates the different states during shifting. Paritycycle is high on the cycle next to the last cycle, i.e., when tsr[1] gets tag2.

```
assign paritycycle = tsr[1] && !(tag2 || tag1 || tsr[7] || tsr[6] || tsr[5] || tsr[4] || tsr[3] || tsr[2]);
```

Txdone is high when the shifting is complete, i.e., when tx gets tag2.

```
assign txdone = !(tag2 || tag1 || tsr[7] || tsr[6] || tsr[5] || tsr[4] || tsr[3] || tsr[2] || tsr[1] || tsr[0]);
```

Based on the different states during the transmission sequence, the "data bits", "parity bit" or "stop bit" are multiplexed to the tx output.

The contents of the data bus are latched into thr at the rising edge of write. At the next rising edge of txclk, the contents of thr are loaded into tsr, the active low start bit is asserted to tx, and the txrdy flag indicates that thr again is ready for new data to be written. At each rising edge of txclk, the contents of tsr are shifted to tx. Parity generation takes place during the shifting of data. Paritycycle is high one cycle next to the last cycle, and tx gets the parity result. The internal txdone is high when the shifting is complete, and the active high stopbit is asserted to tx.

**Simulation of a
transmit sequence**

8

**Application
Notes**

QUICKLOGIC

FIGURE 11
Transmit sequence
using a 2mhz
Baud rate
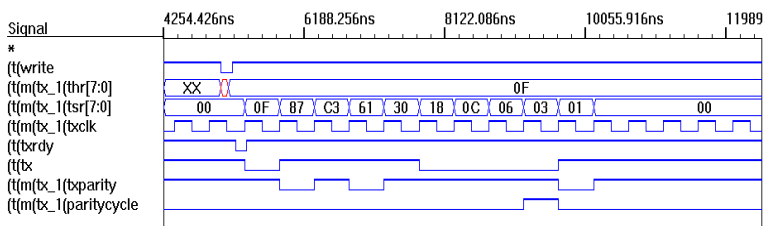
The Receiver
module

Further details on the implementation are available in the design source file available from QuickLogic.

The master clock mclkx16 is divided down to the proper baud rate clock called rxclk, and equals to mclkx16/16. Serial data to be received at the rx input of the module, must follow the UART data format shown in fig. 3. Data received in serial format can be read out in parallel format, through the 8 bit data bus.
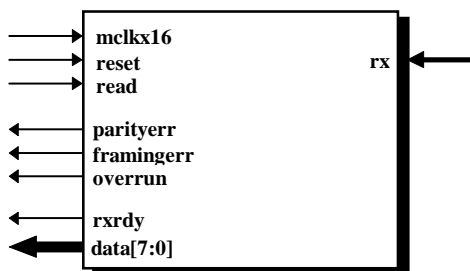
FIGURE 12
Receiver module

TABLE 5
I/O Signals of the
receiver module

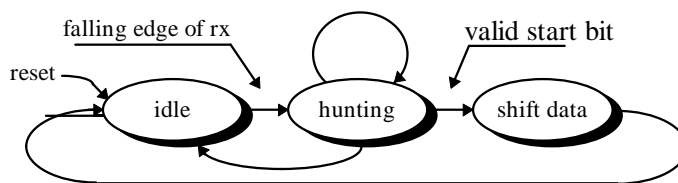| SYMBOL	TYPE	I/O DESCRIPTION
mclkx16	input	Master input clock used for internal baud rate generation.
reset	input	Master reset input signal.
read	input	Active low read strobe signal, used for reading out data from the receiver.
data[7:0]	output	Data that are received can be read out in parallel format through the data bus. At the falling edge of the read strobe, the data are applied to the data bus.
rx	input	Receives serial data. rx should be held high (pulled-up), when no transmission are taking place.
rxrdy	output	Indicates that new data has been received, and is ready to be read out.
parityerr	output	Indicates whether or not a parity error has occurred. Encoding can be based on Even or Odd parity.
framingerr	output	Indicates if serial data sent to the rx input did not match the proper UART data format shown in Figure 3.
overrun	output	Indicates when new data are sent to the receiver, data previously received is still held internally by the receiver, and has not yet been read out.

SYMBOL	TYPE	INTERNAL FLAG & SIGNAL DESCRIPTION
rhr	reg[7:0]	8 bit "receive hold register" used to hold the contents of the data that has been received at the rx input.
rsr	reg[7:0]	8 bit "receive shift register", used for shifting in the data at the rx input.
paritymode	wire	The value of this bit indicates the parity mode of the receiver. Set to 1 => Odd parity mode. Set to 0 => Even parity mode.
rxparity	reg	Signal that holds the parity result of the rsr.
paritygen	reg	paritygen is pre-loaded with paritymode. After receiving a data frame, paritygen is true if a parity error has occurred.
rxclk	reg	Baud rate clock signal used for shifting in data.
rxstop	reg	Holds the stop bit of a data frame received at the rx input.
idle	reg	Receiver status bit.
hunt	reg	Receiver status bit.
rxdatardy	reg	Indicates that new data has been received, and is ready to be read out.
rxcnt	reg[3:0]	Count register used for dividing the mclkx16 down to rxclk.

TABLE 6
Internal signals of the receiver module

Between successive transmissions, the transmission line is held high, according to standard UART behavior. The receiver waits in "idle" mode for the rx input to go low. At the falling edge of rx the receiver enters "hunt" mode, searching for a valid start bit of a new data frame to be received. If a valid start bit is detected, the receiver enter "shift data" mode. If an invalid start bit is detected, the receiver returns to "idle" mode. During reception of a data frame, various parity and error checks are performed. When a complete data frame has been received the receiver returns to idle mode.

Behavioral description of the receiver

The basic operation of the receiver is shown below:

FIGURE 13
Sequence of receiver

falling edge of rx   valid start bit

reset

idle   hunting   shift data

8
Application Notes

QUICKLOGIC

TABLE 7
Behavioral modes
of receiver module

MODE	DESCRIPTION
idle	Active high reset forces the receiver into idle mode. The receiver is waiting for a falling edge of the rx input, to indicate the start bit of new data frame to be received. Generation of the internal "baud rate clock" rxclk is disabled, to ensure that invalid data is not received. All bits of the "receive shift register" are filled with all "1s"
hunting	At the falling edge of the rx input, the receiver starts hunting/searching for a valid start bit. A valid start bit occurs with eight clock cycles of mclkx16 with rx staying low. This ensures correct sampling of the start bit and following data bits, at their respective center points, and prevents noise spikes at the transmission line to be interpreted as a valid start bit. If a valid start bit is found, the receiver enables generation of rxclk, now synchronized to the center point of the start bit.
shift data	At the rising edge of rxclk, data are shifted from the rx input to the "receive shift register". When a leading low "start" bit reaches rsr[0], the next rxclk will preset the receiver into idle mode again, thereby disabling generation of rxclk. When the receiver is done shifting data and returns to idle mode, the contents of the "receive shift register" is loaded into the "receive hold register". The output flag "rxrdy" indicates that new data in "receive hold register" are ready to be read out.

The frequency of rxclk is equal to mclkx16/16, and the first rising edge of the rxclk will always occur at the center point of the start bit. Figure 14 below shows how generation of the baud rate clock rxclk is synchronized to the center points of the start bit and the following data bits.

FIGURE 14
Synchronizing
rxclk to the center
point of start bit
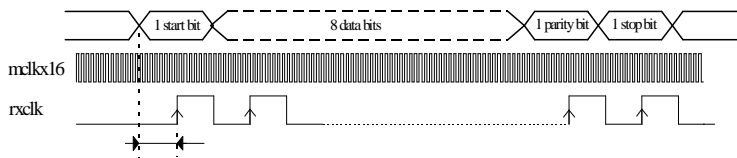
In order to create an easy to read and easy to maintain behavioral model of the receiver, two Verilog tasks are written to describe the different modes of the receiver. The Verilog task called "idle_reset" holds all necessary sequential statements to describe the receiver at reset condition, and when the receiver is in its idle mode.

Implementation of the receiver module

```
task idle_reset;
    begin
    rsr        <= 8'b11111111;   // All 1's ensure that idle stays low during data shifting.
    rxparity   <= 1'b1;          // Preset to high to ensure idle = 0 during data shifting.
    paritygen  <= 1'b1;          // Preset to 1 => odd parity mode, 0 => even parity mode.
    rxstop     <= 1'b0;          // Forces idle = 1, when rsr[0] gets rxstop bit.
    end
endtask
```

**FIGURE 15
Receiver at
reset condition
and idle mode**

When the receiver is not in its reset condition, and not in idle mode, the receiver samples data at the rx input, shifts the data to the "receive shift register", and generates parity based on the incoming data. The Verilog task called "shift_data" holds all necessary sequential statements to describe all the above actions.

```
task shift_data;
    begin
    rsr        <= rsr >> 1;              // Right shift receive shift register.
    rsr[7]     <= rxparity;             // Load rsr[7] with rxparity.
    rxparity   <= rxstop;              // Load rxparity with rxstop.
    rxstop     <= rx;                  // At 1'st shift rxstop gets low "start bit".
    paritygen  <= paritygen ^ rxstop;  // Generate parity as data are shifted.
    end
endtask
```

**FIGURE 16
Receiver
shifting data**

Using the two Verilog tasks described above, we are now able to create the behavioral level description of the receiver in its reset condition, idle mode or when shifting in data.

All of these actions are synchronous to the baud rate clock called rxclk, and the implementation is shown below.

```
always @(posedge rxclk or posedge reset)
if (reset)
    idle_reset;              // Reset internal bits.
else
    begin
    if (idle)
            idle_reset;      // Reset internal bits.
    else
            shift_data;      // Shift data and generate parity.
    end
```

**FIGURE 17
Receive process**

8

QUICK LOGIC

A complete data frame has been received when the leading *low* start bit reaches rsr[0], and the receiver returns to idle mode again at the next rising edge of rxclk.

At return to idle mode the receiver raises the "receive data ready" interrupt to indicate that the new data received now can be read out in parallel format. Error flags are updated as well upon return to idle mode, and cleared when data are read out of the receiver. At the falling edge of read, the contents of the rhr are latched to the data bus. In Table 8 shown below are the various error checks supported by the receiver.

**TABLE 8
Error checks
supported by**

ERROR TYPE	DESCRIPTION
parity check	Parity generation takes place when data are received. A parity generator register called paritygen was pre-loaded in idle mode (1 = odd, 0 = even). After a complete receive session, the value of paritygen will indicate whether or not a parity error has occurred.
frame error	If the stop bit of the data frame received is not equal to 1, it's considered to be a frame error. The data frame did not match the standard UART data format as described in Figure 2.
overrun	When new data has be received into the rsr, but rhr holds previously received data that hasn't been read out yet, it's considered an overrun error. The contents of the rsr are not loaded into the rhr.

**Simulation of a
receive sequence
receiver**

Between successive transmissions, the transmission line is held high. At the falling edge of rx input, the internal rxcnt starts counting up, synchronous to mclkx16. If the rx input stays low for 8 cycles of mclkx16, the internal status bit idle is reset, and thereby enables generation of rxclk. Rxclk is now synchronized to the center point of the low start bit. At the rising edge of rxclk, data are shifted from the rx input to rsr. When the leading low start bit reaches rsr[0], the next rising edge of rxclk forces idle high again, and there by disable generation of rxclk. During a receive sequence, exactly 11 cycles of rxclk are generated, in order to sample a total of 1 leading low start bit, 8 data bits, 1 parity bit and 1 trailing high stop bit. At return to idle mode, the contents of rsr are loaded into rhr and the status flags are updated. The flag "rxrdy" now indicates that the contents of rhr can be read out. At the falling edge of read, the contents of rhr are applied to the data bus.
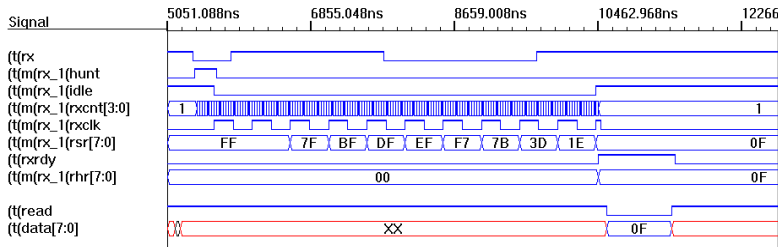
**FIGURE 18
Simulation of a
receive sequence
using a 2MHz
baudrate**

We have now studied how HDLs can be used for the behavioral level design implementation of a digital UART. While HDLs make the design implementation easier to read and hopefully to understand as well, they also provide the ability to easily describe dependencies between various processes that usually occur in such complex event-driven systems as the UART. This ability to describe dependency between various processes is extremely valuable for simulation purposes as we will see very soon.

Simulation stimulus in Verilog HDL is called a "test fixture". A test-fixture is a Verilog module that holds all lines of HDL code necessary to generate the simulation stimulus, while it at the same time port maps these signals to the design that is to be simulated.

The port mapping is done by hierarchical module instantiation of the UART top level module into the test-fixture, as shown below.

```
uart m (.mclkx16(mclkx16), .reset(reset), .read(read), .write(write),
        .data(data), .rx(rx), .tx(tx), .rxrdy(rxrdy), .txrdy(txrdy),
        .parityerr(parityerr), .framingerr(framingerr), .overrun(overrun));
```

**Using Hardware
Description
Language for
Simulation**

**FIGURE 19
Top level UART
module
instantiation**

This allows simulation stimulus to be applied to the inputs of the design, while monitoring the outputs of the design. Input stimulus can be made conditionally to the response on the outputs. Figure 19 shows how the test-fixture port maps to the top level of the UART.

8

Application Notes

QUICKLOGIC

**FIGURE 20
UART model and
test-fixture**
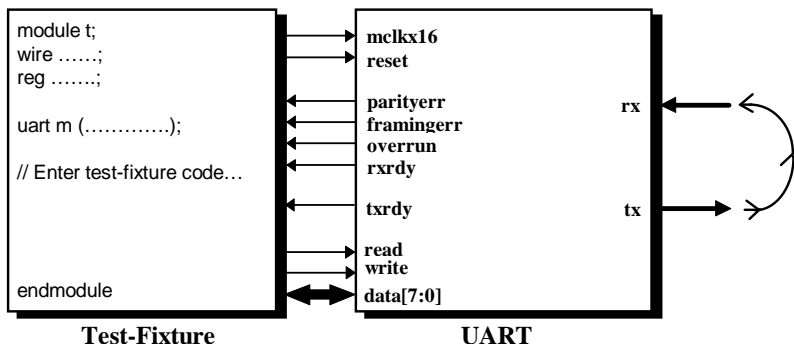
| Test-Fixture | UART |

Within the test-fixture the tx output of the transmitter module is looped back to the rx input of the receiver module. This allows the transmitter module to be used as a test signal generator for the receiver module. Data can be written in parallel format to the transmitter module and looped back in serial format to the rx input of the receiver module, and data received can be read out in parallel format from the receiver module.

In order to automate the testing of the UART as much as possible, three independent Verilog tasks were written as follows:

The Verilog task "**write_to_transmitter**" holds all necessary statements required to generate a single parallel data write sequence to the transmitter module. Data that is written to the transmitter upon execution of the "write_to_transmitter" task, gets latched internally to the test-fixture for later analysis.

The Verilog task "**read_out_receiver**" holds all necessary statements required to generate a single parallel data read out sequence from the receiver module. Data that is read out of the receiver upon execution of the "read_out_receiver" task gets latched internally to the test-fixture for later analysis.

The Verilog task "**compare_data**" holds all necessary statements required to compare the previous data written to the transmitter module, to the corresponding and most recent data received and read out from the receiver module. If any discrepancy occurs, the "compare_data" task flags an error by writing out the data values that were written to the transmitter module, as well as the corresponding data values that were received by and read out from the receiver module. The simulation is immediately stopped by the "compare_data" task if any discrepancy occurs.

Besides the three above-mentioned Verilog tasks, the test-fixture holds the statements to generate mclkx16, the master reset signals, and the "tx to rx" loop back feature. These statements are considered trivial, and will not be illustrated here, but can be referred to within the test-fixture itself.

The core of the test-fixture is a behavioral level "for loop" that executes the three above-mentioned Verilog tasks in order to write all possible data combinations to the transmitter and verify that same data gets properly received by the receiver. The "for loop" is shown below in Figure 21.

```
// Write every possible combinations to the transmitter.
        for (i = 8'h0; i  <= 8'hff; i = i + 1) begin
                    write_to_transmitter(i);        // Write new data to transmitter.
                    wait(rxrdy);                    // Wait for rxrdy.
                    read_out_receiver;              // Read out data from receiver.
                    compare_data;                   // Compare "data send" to "data received".
                    end
```

FIGURE 21
Core sequence
of test-fixture

The above shown "for loop" uses the Verilog "wait" statement. The "wait" statement is a concurrent process statement that waits for the conditional expression to become true. Hence in this case, the "for loop" waits for the receiver to complete any current receive sequence indicated by the "rxrdy" flag going high. Conceptually, the execution of the "for loop" stops until "rxrdy" goes high. When "rxrdy" goes high, the for loop immediately executes the "read_out_receiver" task, followed by the "compare_data" task.

According to the UART data format shown in Figure 3, the test-fixture has to process 256 different data combination to the UART in order to test all possible data combinations. When the "for loop" has processed all data combinations without any error flags from the "compare_data" task, the test-fixture finally completes and stops.
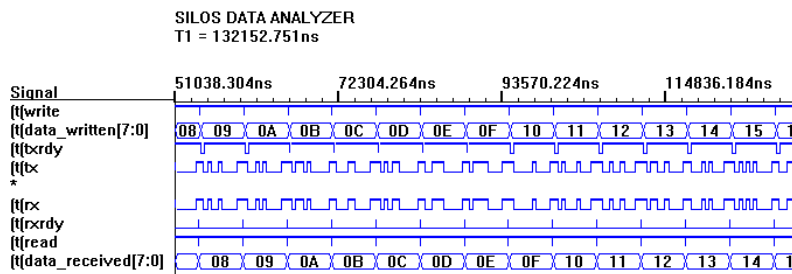

FIGURE 22
Top level UART
simulation
sequence

8

Application Notes

QUICKLOGIC

Figure 22 shows a simulation sequence using a 2MHz baud rate frequency, and illustrates the execution of the "for loop" from Figure 21.

1.   Data is written to the transmitter upon execution of the "write_to_transmitter" task.
2.   The data written is automatically latched into "data_written[7:0]" at the rising edge of the write strobe.
3.   The txrdy flag indicates when the transmitter is ready for new data to be written.
4.   At the selected baud rate the data written in parallel format to the transmitter now gets transmitted in serial format through the tx output.
5.   The tx output is fed back to the rx input of the receiver.
6.   The rxrdy flag indicates when new data have been received.
7.   At the rising edge of the rxrdy flag, the "for loop" executes the "read_out_receiver" task. Data received is automatically latched into "data_received[7:0]" at the falling edge of the read strobe.
8.   Data_received[7:0] gets compared to data_written[7:0] by the "compare_data" Verilog task.

Different flavors of the test-fixture

We have now briefly discussed the UART test-fixture, as well as various simulation aspects of the Verilog language. The UART test-fixture described above uses generic read and write signal sequences chosen by the author, but for more specific system level integration and simulation of the UART, the "write_to_transmitter" task as well as the "read_out _receiver" task can easily be modified to reflect any given CPU's write and read cycles.

Should any further simulation features than already explained within this application note be required, the test-fixture can easily be modified to add these.

Silicon for synthesis

While HDLs as design implementation method offers several advantages over traditional FPGA design entry approaches such as schematic capture, it requires great flexibility as well as high performance by the target devices for the synthesis flow. The synthesis flow for the UART has been targeted to two flexible and high performance FPGA architectures available from QuickLogic, called the pASIC 1 and the pASIC 2 families. The results after synthesis and place and route is shown below in Table 9.

The UART design and simulation files were entered using the HDL sensitive editor called TurboWriter from Saros Technologies, and the HDL source files for the design were synthesized using the fast and efficient Synplify-Lite synthesis tool from Synplicity. After synthesis, the design was placed & routed using the SpDE place & ROUTE tools from QuickLogic. After Place & Route, the UART design was simulated using back-annotated Verilog post-layout timing models. The fast Verilog simulator called Silos III from Simucad was used for the post-layout simulation. All of these tools are available within the Quick*Works* tool suite from QuickLogic.
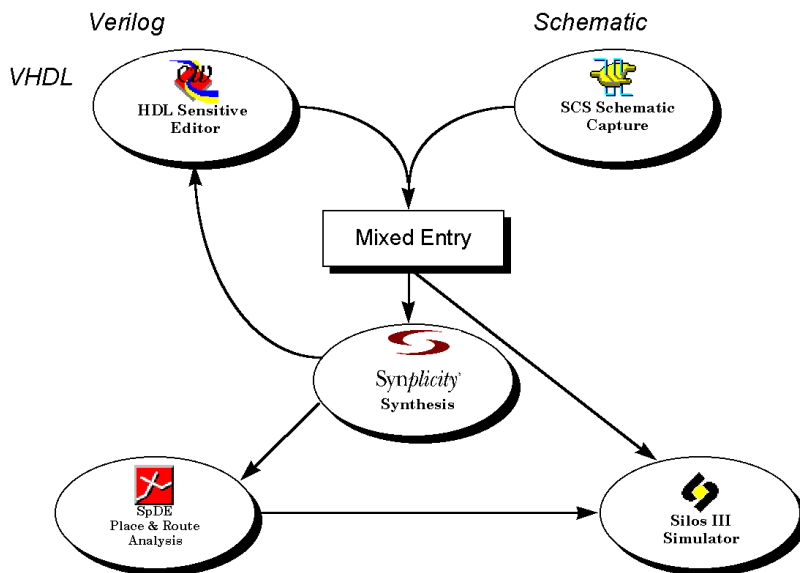

FIGURE 23
**Quick*Works*
Tool suite**

	Function	PASIC 1*	PASIC 2**
Utilization	Transmitter	27 logic cells	20 logic cells
	Receiver	38 logic cells	34 logic cells
	Full Duplex UART	66 logic cells	54 logic cells
Baudrate	Transmitter	0 - 5MHz	0 - 5MHz
	Receiver	0 - 5MHz	0 - 5MHz
	Full Duplex UART	0 - 5MHz	0 - 5MHz

TABLE 9
**UART utilization
and performance**

*)   *Device used was QL12x16B-2PL68C*
**) *Device used was QL2007-2PL84C*

All numbers shown above represent worse-case numbers within commercial temperature range, Automatic Place & Route.

8

**Application
Notes**

QUICKLOGIC