

uartlCE40 — Asynchronous receiver/transmitter for iCE40 FPGAs

Baard Nossum

July 16, 2016

Contents

Contents	i
1 Introduction	1
1.1 Features	1
1.2 Project scope and documentation	1
2 Top level	2
3 Transmit module	3
3.1 High level implementation	3
3.2 Low level implementation	4
4 Receive module	7
4.1 High level implementation	7
4.2 Low level implementation	9
5 The dividers	11
5.1 High level implementation	11
5.2 Low level implementation	12

6	All together	14
6.1	High level implementation	14
6.2	Low level implementation	14
7	Testbench	15
7.1	Use of the test bench	17
8	ICEstick	19
9	Conclusion	23

Introduction

1.1 Features

- Specifically written for iCE40
- Easy to use
- Small footprint, 32 (default) or 34 logicCells
- Format is hardcoded: 8 data bits, no parity, one stop bit (8N1)
- Samples RX pin 8 (default) or 16 times per bit period

A note on nomenclature

In practical terms, the `usart`¹ is nearly always used asynchronously, with 8 data bits. There are still applications that relies on 1 or 2 stop bits, and odd or even parity, but a large majority of applications uses 1 stop bit, no parity. The module described in this paper should perhaps be called an `art` (asynchronous receiver/transmitter), but I bow to conventions and call the module an “uart”.

1.2 Project scope and documentation

Two verilog source code files, `uartICE40` and `uartICE40hl` are the result of this project. The rest of the files are just there to verify that those two files are correct. The files `uartICE40` and `uartICE40hl` should be functionally identical. While `uartICE40` is written using primitives of the iCE40 architecture, the corresponding “high-level” `uartICE40hl` is synthesized from standard Verilog. Motivation for two different implementations is simply that it is cumbersome to understand a low level implementation that relies on explicitly instantiated LUTs. FPGA projects often have many dependencies, and files. Users of the two files above will need to move the files to relevant locations,

Documentation and code is generated using `noweb`.² It is a paradox that documentation of such a small and easy module extends to 23 pages.

¹`usart`: Universal, synchronous/asynchronous receiver transmitter. This implies that it can be used synchronous with an external clock. “universal” denotes that one can select the number of data bits and stop bits, and whether (odd or even) parity is to be used or not.

²For those unfamiliar with `noweb` - it is a tool that allow writing of documentation and of code in the same textfile. A definite advantage is that the code that is documented is indeed the code that is generated.

Chapter 2

Top level

Regardless of whether the implementation is high or low level, `uartICE40` has the following interface:

2a `<module head 2a>`≡ (14)

```
module uartICE40
  # (parameter SUBDIV16 = 0, // Examine rx line 16 or 8 times per bit
    ADJUSTSAMPLEPOINT=0 // See documentation
  ) (
    input      clk, // System clock
    input      bitxce, // High 1 clock cycle 8 or 16 times per bit
    input      load, // Time to transmit a byte. Load transmit buffer
    input [7:0] d, // Byte to load into transmit buffer
    input      rxpin, // Connect to receive pin of uart
    output     txpin, // Connect to INVERTED transmit pin of uart
    output     txbusy, // Status of transmit. When high do not load
    output     bytercvd, //Status receive. True 1 clock cycle only
    output [7:0] q // Received byte from serial receive/byte buffer
  );
```

`uartICE40` top module instantiates three submodules. It does not contain any code itself. To limit typing, I use the verilog mode of emacs. Verilog source files are expanded so that the source code is usable to those programmers that do not use emacs.

2b `<module body 2b>`≡ (14)

```
/*AUTOWIRE*/
uarttx_m uarttx_i (*AUTOINST*);
uartrx_m uartrx_i (*AUTOINST*);
rxtxdiv_m #( .ADJUSTSAMPLEPOINT(ADJUSTSAMPLEPOINT),
              .SUBDIV16(SUBDIV16))
rxtxdiv_i
(*AUTOINST*);
endmodule
```

Transmit module

A uart transmitter is very simple. Basically we load a shift register with the byte to transfer, and shift the byte out one and one bit. Before shifting out the first (lsb) data bit, we must shift out the start bit. After the byte has been shifted out, we must shift out the stop bit. A status output tells if the transmit module is busy.

3.1 High level implementation

The interface to the transmit module has the system clock as input. When data is to be loaded, it is given to this module on the 8-bit `d` input, qualified by `load`. This module should shift each `txce` clock cycle. `txce` is logically or'ed with `load`. The shift register is 10 bits: The transmit shift register only changes when we load data, or when `txce` is active. When this happens, we do the following:

```

3  <hl uarttx module 3>≡ (14a)
    module uarttx_m
      (input      clk,load,load0Rtxce,
       input [7:0] d,
       output reg txpin, txbusy
      );
      reg [9:0] a;
      always @(posedge clk)
        if ( load0Rtxce ) begin
          a[9:0] <= load ? {1'b0,d,1'b0} : {1'b0,a[9:1]};
          txbusy <= load | |a[9:1];
          txpin  <= (load & txpin) | (!load & |a[9:1] & a[0]);
        end
    endmodule

```

- iCE40 FPGAs starts up with all flip-flops (FF) cleared. At power-up, I want the uart to be inactive. The easiest way to do this is probably to *invert* the pin output. But then we must compensate by inverting the data byte during load.
- If the shift register is not completely empty, or if we are loading the shift register, the transmit module is busy.
- Loading of the shift register is asynchronous to shifting, a FF is used to synchronize the shift register to `txce`. This give the `txpin` output FF.

Synplify or yosys is not able to understand that `txbusy` can be found out of the carry chain. This module is written this way to better explain the low level implementation in the next section.

3.2 Low level implementation

The transmit part in 12 LogicCells. txpin is to be connected to a pad with *inverted* output. This way uart transmit will go to inactive during power-up. The main idea here is to have a 10-bit shift register. When all bits are shifted out, a fact we find from the carry chain, transmission is done. We also need a FF to synchronize "load" with "txce".

In addition we need a FF to record that the shift register is busy. The shift register is busy from the clock cycle after it is loaded, until start of transmit of the stop bit. For continuous transfer, a microcontroller is expected to hook up ~txbusy as an interrupt source. A new byte must then be output in 7/8 bit times. An example: At 12 MHz clock, 115200 bps, a new byte must be written in at most 91 clock cycles to saturate the transmit path. The layout of the low-level implementation is shown in figure 3.1. In an ideal world, the high level description would map into this use of resources, but since the world is less than ideal, we must write it ourselves.

The module head is very similar to the high level implementation. When it comes to variables, a few more are needed:

```

4a  <ll uarttx module head 4a>≡ (14b)
    module uarttx_m
    (
        input      clk,load,load0Rtxce,
        input [7:0] d,
        output      txpin,
        output      txbusy
    );
    genvar      i;
    wire         c_txbusy,c_pp;
    wire [9:0]    c_a,a;
    wire [10:1]    cy;

```

The stop bit is set when we load, and cleared when we shift. We implement $ff \leftarrow load \mid ff \sim txce$

```

4b  <ll uarttx module code 4b>≡ (14b) 4c▷
    SB_LUT4 #(.LUT_INIT(16'haaaa))
    ff_i(.O(c_a[9]), .I3(1'b0), .I2(1'b0), .I1(1'b0), .I0(load));
    SB_DFFE ff_r(.Q(a[9]), .C(clk), .E(load0Rtxce), .D(c_a[9]));

```

A generate statement takes care of the shift register with parallel load. Zerodetection takes place in the carry chain:

```

4c  <ll uarttx module code 4b>+≡ (14b) <4b 6a>
    generate
        for ( i = 0; i < 9; i = i + 1 ) begin : blk
            if ( i == 0 ) begin
                SB_LUT4 #(.LUT_INIT(16'h55cc))
                shcmb( .O(c_a[i]), .I3(load), .I2(1'b1), .I1(a[i+1]), .I0(1'b0));
                SB_CARRY shcy(.CO(cy[i+1]), .CI(1'b0), .I1(1'b1), .I0(a[i+1]));
            end else begin
                SB_LUT4 #(.LUT_INIT(16'h55cc))
                shcmb( .O(c_a[i]), .I3(load), .I2(1'b1), .I1(a[i+1]), .I0(d[i-1]));
                SB_CARRY shcy(.CO(cy[i+1]), .CI(cy[i]), .I1(1'b1), .I0(a[i+1]));
            end
            SB_DFFE r( .Q(a[i]), .C(clk), .E(load0Rtxce), .D(c_a[i]));
        end
    endgenerate

```

Figure 3.1: The transmit module in it's low-level version

Transmit is busy from the cycle after load, until the cycle where transmission of the stop bit starts. An implication is that the unit feeding the transmitter has a rather short window to write a new byte if continuous transmission is desired. $txbusy \leftarrow load \mid (\sim load \ \& \ a[9:1]) \mid (\sim load \& \sim txce \& txbusy)$ is implemented. Note that carry is transported unchanged across this LUT.

6a $\langle ll \text{ uarttx module code 4b} \rangle + \equiv$ (14b) $\triangleleft 4c \ 6b \triangleright$

```

SB_LUT4 #(.LUT_INIT(16'hffaa))
txbusy_i( .O(c_txbusy), .I3(cy[9]), .I2(1'b1), .I1(1'b0), .I0(load));
SB_CARRY msbcy( .CO(cy[10]), .CI(cy[9]), .I1(1'b1), .I0(1'b0));
SB_DFFE txbusy_r( .Q(txbusy), .C(clk), .E(load0Rtxce), .D(c_txbusy));

```

Finally we have the synchronisation stage. $pp \leftarrow load * pp \mid \sim load * txce \& cy_{10} * a_0 \mid \sim load \& \sim txce \& pp$ is implemented.

6b $\langle ll \text{ uarttx module code 4b} \rangle + \equiv$ (14b) $\triangleleft 6a$

```

SB_LUT4 #(.LUT_INIT(16'hb888))
pp_i( .O(c_pp), .I3(cy[10]), .I2(a[0]), .I1(load), .I0(txpin));
SB_DFFE pp_r( .Q(txpin), .C(clk), .E(load0Rtxce), .D(c_pp) );
endmodule

```

Receive module

4.1 High level implementation

The receive module consists of a receive state machine and a receive shift register. See page 8 for the receive state machine.

```
7  <hl uartrx module 7>≡ (14a)
    module uartrx_m
        # (parameter HUNT = 2'b00, GRCE = 2'b01, ARMD = 2'b10, RECV = 2'b11 )
        (
            input          clk,rxce,rxpin,
            output          bytercvd,
            output [1:0]    rxst,
            output reg [7:0] q
        );
        uartrxsm_m #(.HUNT(HUNT), .GRCE(GRCE), .ARMD(ARMD), .RECV(RECV))
        rxsm(// Inputs
            .lastbit( q[0] ),
            /*AUTOINST*/);
        always @(posedge clk)
            if ( rxce )
                q <= (rxst == ARMD) ? 8'h80 : (rxst == RECV) ? {rxpin,q[7:1]} : q;
    endmodule
```

Each rxce, if the state machine is in state ARMD, the shift register is initiated to 8'h80. If we are receiving, the shift register is shifted. Otherwise the shift register is held.

By initiating the shift register to 8'h80, we know a complete byte is received when a 1'b1 is shifted out of the shift register, so we need no counter for the number of received bits.

High level receive state machine

The states of the state machine is encoded as parameters to avoid polluting the name space. When rxce is active (8 or 16 times per bit), the state machine may change state.

```

8  <hl uartrx state machine module 8>≡ (14a)
    module uartrxsm_m
        # ( parameter HUNT = 2'b00, GRCE = 2'b01, ARMD = 2'b10, RECV = 2'b11 )
        (input          clk,rxce,rxpin,lastbit,
         output         bytercvd,
         output reg [1:0] rxst
        );
        reg [1:0]      nxt;
        always @(*AS*/) begin
            casez ( {rxst,rxpin,lastbit,rxce} )
                {HUNT,3'b1??} : nxt = HUNT;
                {HUNT,3'b0?0} : nxt = HUNT;
                {HUNT,3'b0?1} : nxt = ARMD;
                {ARMD,3'b??0} : nxt = ARMD;
                {ARMD,3'b0?1} : nxt = RECV;
                {ARMD,3'b1?1} : nxt = HUNT; // False start bit.
                {RECV,3'b??0} : nxt = RECV;
                {RECV,3'b?01} : nxt = RECV;
                {RECV,3'b?11} : nxt = GRCE;
                {GRCE,3'b??0} : nxt = GRCE;
                {GRCE,3'b0?1} : nxt = HUNT; // Stop bit wrong, reject byte.
                {GRCE,3'b1?1} : nxt = HUNT; // Byte received
            endcase
        end
        always @(posedge clk)
            rxst <= nxt;
        assign bytercvd = (rxst == GRCE) && rxpin && rxce;
    endmodule

```

When hunting for a start bit (HUNT), if the rxpin is low, the state machine transitions into the armed state (ARMD). After 1/2 bit time, rxpin is resampled. If it is low, we conclude a start bit has been seen. If the input is noisy, this is likely to be a bad implementation, but easily corrected by applying a digital low-pass filter on the rxpin.

Note that after the last data bit has been sampled, the state machine goes to GRCE. The bitclock divider unit know when the receive state machine is in GRCE, and if it sees a high data bit, it will (prematurely) give a rxce, this will then lead to a quicker entry to the HUNT state. An advantage is slightly more tolerant limits on a (too fast) receive data stream, a disadvantage is that the rxpin is used in two consecutive clock cycles, hence in case of noisy input we may lose bytes. There is perhaps room for improvement here.


```

        SB_LUT4 #(.LUT_INIT(16'h0080))
        bytercvd_i( .0(bytercvd), .I3(rxst[1]), .I2(rxst[0]), .I1(rxpin), .I0(rxce));
    endmodule

```

The receive state machine is assembled with the shift register, implemented in a generate loop. Total size 11 logicCells.

```

10  <ll uartrx module 10>≡ (14b)
    module uartrx_m
    (
        input      clk,rxce,rxpin,
        output     bytercvd,
        output [1:0] rxst,
        output [7:0] q
    );
    genvar        i;
    wire [7:0]    c_sh;

    uartrxsm_m rxsm(// Inputs
                    .lastbit( q[0] ),
                    /*AUTOINST*/);

    generate
        for ( i = 0; i < 8; i = i + 1 ) begin : blk
            localparam a = i == 7 ? 16'hbfb0 : 16'h8f80;
            SB_LUT4 #(.LUT_INIT(a))
            sh( .0(c_sh[i]), .I3(q[i]), .I2(rxst[1]), .I1(rxst[0]),
                .I0(i==7 ? rxpin:q[i+1]));
            SB_DFFE shreg( .Q(q[i]), .C(clk), .E(rxce), .D(c_sh[i]) );
        end
    endgenerate
    endmodule

```

The dividers

An important input to the uart is the `bitxce` clock. It is used to increment a free-running 3 or 4 bit counter destined for the transmit module. It is also used to increment a 3 or 4 bit resettable counter destined for the receive module. The receive module also need a reset signal, it is placed in this module to save a LUT in the low level implementation.

When the transmission rate of the uart is close to $\text{clk}/8$ (or $\text{clk}/16$), we reset the `rxce` counter to 5 (or 9) rather than to 4 (or 8). This fine-adjustment places sampling of bits closer to the middle of the window in these cases.

5.1 High level implementation

```

11  <hl uart counters module 11>≡ (14a)
    module rtxdiv_m
        # (parameter HUNT = 2'b00, GRCE = 2'b01, ARMD = 2'b10, RECV = 2'b11,
            SUBDIV16 = 0, ADJUSTSAMPLEPOINT = 0
        )
        (input      clk,bitxce,load,rxpin,
         input [1:0] rxst,
         output     loadORtxce,rst4,
         output reg  rxce
        );
        localparam rstval = SUBDIV16 ? (ADJUSTSAMPLEPOINT ? 4'b1001 : 4'b1000) :
            (ADJUSTSAMPLEPOINT ? 3'b101 : 3'b100);
        reg [2+SUBDIV16:0] txcnt,rxcnt;
        always @(posedge clk) begin
            if ( bitxce ) begin
                txcnt <= txcnt + 1;
                rxcnt <= rst4 ? rstval : (rxcnt+1);
            end
            rxce <= (((rxst == ARMD) | (rxst == RECV)) & (&rxcnt & bitxce) )
                | ((rxst == HUNT | rxst == GRCE) & rxpin);
        end
        assign loadORtxce = (&txcnt & bitxce) | load;
        assign rst4 = rxst == HUNT;
    endmodule

```

5.2 Low level implementation

A sketch of the implementation (in 9 or 11 logicCells) is seen in figure 5.1.

```

12  <ll uart counters module 12>≡ (14b)
    module rtxdiv_m
        #( parameter ADJUSTSAMPLEPOINT = 0, SUBDIV16 = 0)
        (input      clk,bitxce,load,rxpin,
         input [1:0] rxst,
         output      load0Rtxce,rxce,rst4
        );
        localparam rstval_lsb = ADJUSTSAMPLEPOINT ? 16'haffa : 16'h0550;
        localparam LOOPLIM = SUBDIV16 ? 4 : 3;
        wire [LOOPLIM+1:0] cy,rxcy;
        wire                c_rxce;
        wire [LOOPLIM-1:0] c_txcnt,txcnt,c_rxcnt,rxcnt;
        genvar              j;

        assign cy[0] = 1'b0;
        generate
            for ( j = 0; j < LOOPLIM; j = j + 1 ) begin : blk0
                SB_LUT4 #(.LUT_INIT(16'hc33c)) i_txcnt1(.0(c_txcnt[j]),
                    .I3(cy[j]), .I2(txcnt[j]), .I1(j==0 ? bitxce:1'b0), .I0(1'b0));
                SB_CARRY i_cy1(.CO(cy[j+1]),
                    .CI(cy[j]), .I1(txcnt[j]), .I0(j==0 ? bitxce:1'b0));
                SB_DFF reg1( .Q(txcnt[j]), .C(clk), .D(c_txcnt[j]));
                if ( j == LOOPLIM-1 ) begin
                    SB_LUT4 #(.LUT_INIT(16'hfaaa))
                        i_txcnt3(.0(load0Rtxce),
                            .I3(cy[j+1]),.I2(bitxce ), .I1(bitxce),.I0(load));
                    SB_CARRY i_cy3(.CO(rxcy[0]),
                        .CI(cy[j+1]),.I1(bitxce ), .I0(bitxce));
                end
            end
        endgenerate
        generate
            for ( j = 0; j < LOOPLIM; j = j + 1 ) begin : blk1
                if ( j != LOOPLIM-1 ) begin
                    SB_LUT4 #(.LUT_INIT(j == 0 ? rstval_lsb : 16'h0550)) i_rxcnt0
                        (.0(c_rxcnt[j]), .I3(rxcy[j]), .I2(rxcnt[j]),.I1(1'b0),.I0(rst4));
                    SB_CARRY i_cy4(.CO(rxcy[j+1]),.CI(rxcy[j]),.I1(rxcnt[j]),.I0(1'b0));
                end else begin
                    SB_LUT4 #(.LUT_INIT(j == (LOOPLIM-1) ? 16'hcffc:16'h0550)) i_rxcntl
                        (.0(c_rxcnt[j]), .I3(rxcy[j]), .I2(rxcnt[j]),.I1(rst4),.I0(1'b0));
                    SB_CARRY i_cy4(.CO(rxcy[j+1]),.CI(rxcy[j]),.I1(rxcnt[j]),.I0(rst4));
                end
                SB_DFF reg4( .Q(rxcnt[j]), .C(clk), .D(c_rxcnt[j]));
                if ( j == LOOPLIM-1 ) begin
                    SB_LUT4 #(.LUT_INIT(16'h0055)) i_rst
                        (.0(rst4), .I3(rxst[1]), .I2(1'b0),.I1(bitxce), .I0(rxst[0]));
                    SB_CARRY i_andcy
                        (.CO(rxcy[j+2]),.CI(rxcy[j+1]),.I1(1'b0),.I0(bitxce));
                    SB_LUT4 #(.LUT_INIT(16'hfc30)) i_rxce
                        (.0(c_rxce), .I3(rxcy[j+2]),.I2(rxpin),.I1(rxst[1]),.I0(rxst[0]));
                    SB_DFF regrxce( .Q(rxce), .C(clk), .D(c_rxce));
                end
            end
        endgenerate
    endmodule

```

```

rxstate[0] --| I0 |     
rxstate[1] --| I1 | -| | - rxce = ((ARM0 | RECV) & rxcy & bitxce) |
rxpin -----| I2 | >_ | (GRCE | HUNT) & rxpin
+-----| I3 | Note, rxstate[0] not really needed. Remove
|
| rxcy & bitxce "(receive count overflow, or rst4) & bitxce"
/cy\
rxst[0](((---- I0 rst4 = rxst == 2'b0
bitxce ---+((---- I1 Note that carry is not entered into
0 ----(+--- I2 this LUT, but it could be, can move
rxst[1]-(--- I3 rxst[1] to I2.
|
| /cy\ rxcnt is an up-counter
0 -(((---- I0
rst4 --+((---- I1 ~rst4&(rxcnt2^cy) ---| |-- rxcnt2
rxcnt2 --(+---- I2 | rst4 >_ |
+----- I3
|
| /cy\
rst4-(((---- I0
0 --+((---- I1 ~rst4&(rxcnt1^cy) ---| |-- rxcnt1
rxcnt1 --(+---- I2 >_ |
+----- I3
|
| /cy\
rst4-(((---- I0
0 --+((---- I1 ~rst4&(rxcnt0^bitxce) ---| |-- rxcnt0
rxcnt0 --(+---- I2 >_ |
+----- I3
| bitxce
| /cy\
load -(((---- I0
bitxce --+((---- I1
bitxce --(+---- I2 (cy&bitxce) | load = load0Rtxce
+----- I3
|
| /cy\ txcnt is a up-counter
0 -(((---- I0
0 --+((---- I1 (bitxce^txcnt2^cy)---| |-- txcnt2
txcnt2 --(+---- I2 >_ |
+----- I3
|
| /cy\
0 -(((---- I0
0 --+((---- I1 (bitxce^txcnt1^cy)---| |-- txcnt1
txcnt1 --(+---- I2 >_ |
+----- I3
|
| /cy\
0 -(((---- I0
bitxce --+((---- I1 (bitxce^txcnt0^0) ---| |-- txcnt0
txcnt0 --(+---- I2 >_ |
+----- I3
|
gnd

```

Figure 5.1: Approximative sketch of counters for uart control

All together

6.1 High level implementation

```
14a <../src/uartICE40hl.v 14a>≡
    /* High level version of a small simple asynchronous transmitter/receiver
       For documentation see the wiki pages. */
    <module head 2a>
    <module body 2b>
    <hl uarttx module 3>
    <hl uartrx state machine module 8>
    <hl uartrx module 7>
    <hl uart counters module 11>
```

6.2 Low level implementation

Total size is $12+11+9 = 32$ logicCells when we oversample 8 times, and $12+11+11 = 34$ logicCells when we oversample 16 times.

```
14b <../src/uartICE40.v 14b>≡
    /* A small simple asynchronous transmitter/receiver
       For documentation see the wiki pages. */
    <module head 2a>
    <module body 2b>
    <ll uarttx module head 4a>
    <ll uarttx module code 4b>
    <ll uartrx state machine module 9>
    <ll uartrx module 10>
    <ll uart counters module 12>
```

Testbench

A previous implementation of the uart generated the x8 (or x16) bit clock internal to the module. This complicated simulation. Now that the important bitxce input is external to the uart, testing is simplified.

```
15a <../src/tbuartICE40.v 15a>≡ 15b▷
module tst;
  reg [31:0] cyclecounter,simtocy,tx_cyclecounter;
  reg      load,bytercvd_dly1;
  wire     rxpin;
  reg [7:0] d;
  reg      seenB;
  reg      base_clk;
  reg      rx_clk;
  reg      tx_clk;
  reg [2:0] bitxce_tx_cnt;
  reg [2:0] bitxce_rx_cnt;
  localparam char1 = 8'hc1, char2 = 8'h4e;
  localparam SIMTOCY = (1+'SUBDIV16)*4000,RXCLKSTART = 100;
  localparam subdiv16 = 'SUBDIV16; // From makefile
```

We want to instantiate two uarts, one for transmit and one for receive. We then send a character 8'hc1, followed by 8'h4e. Lets start by defining a base clock, and do some initiation:

```
15b <../src/tbuartICE40.v 15a>+≡ <15a 16a>
/*AUTOWIRE*/
always # 20 base_clk = ~base_clk;

initial begin
  $dumpfile('TSTFILE');//"obj/tst.lxt"
  $dumpvars(0,tst);
  d <= 0;      tx_clk <= 0;    simtocy = SIMTOCY;  bitxce_rx_cnt <= 0;
  load <= 0;   rx_clk <= 0;    cyclecounter <= 0;  tx_cyclecounter <= 0;
  seenB <= 0;  base_clk <= 0;  bitxce_tx_cnt <= 0;
end
```

We should have a result in not too many cycles:

```
16a <../src/tbuartICE40.v 15a>+≡ <15b 16b>
    always @(posedge base_clk ) begin
        cyclecounter <= cyclecounter+1;
        if ( cyclecounter > SIMTOCY ) begin
            if ( simtocy == SIMTOCY )
                $display( "Simulation went off the rails" );
            $finish;
        end
        tx_clk <= ~tx_clk;
        if ( cyclecounter > RXCLKSTART )
            rx_clk <= ~rx_clk;
    end
```

Feed the transmitting uart

```
16b <../src/tbuartICE40.v 15a>+≡ <16a 16c>
    always @(posedge tx_clk) begin
        tx_cyclecounter <= tx_cyclecounter + 1;
        load <= ( tx_cyclecounter == 100 ||
            tx_cyclecounter == 100 + 8*8*10*(1+'SUBDIV16) )
            ? 1'b1 : 1'b0;
        if ( tx_cyclecounter == 99 ) begin
            d <= char1;
        end else if ( tx_cyclecounter == 150 ) begin
            d <= char2;
        end
    end
```

Examine if we have received something:

```
16c <../src/tbuartICE40.v 15a>+≡ <16b 17a>
    always @(posedge rx_clk) begin
        bytercvd_dly1 <= bytercvd;
        if ( bytercvd_dly1 ) begin
            if ( seenB ) begin
                if ( q != char2 ) begin
                    $display( "Something wrong2" );
                    simtocy <= cyclecounter+400;
                end else begin
                    $display( "Success" );
                    simtocy <= cyclecounter+400;
                end
            end else begin
                if ( q != char1 ) begin
                    $display( "Something is wrong" );
                    simtocy <= cyclecounter+400;
                end else begin
                    //$display("HERE");
                    seenB <= 1;
                end
            end
        end
    end
```

For the transmitter:

```

17a <../src/tbuartICE40.v 15a>+≡ <16c 17b>
    wire dummy_txpin, dummy_txbusy, dummy_bytercvd;
    wire bitxce_rx, bitxce_tx, dummy_rxpin;
    wire [7:0] dummy_q;
    localparam adjsamplept = 'BITLAX;

    assign dummy_rxpin = 0;
    uartICE40
    #( .SUBDIV16(subdiv16), .ADJUSTSAMPLEPOINT(adjsamplept))
    dut_tx
    (// Outputs
      .bytercvd(dummy_bytercvd),
      .q (dummy_q[7:0]),
    // Inputs
      .rxpin (dummy_rxpin),
      .clk (tx_clk),
      .bitxce (bitxce_tx),
    /*AUTOINST*/);

```

For the receiver:

```

17b <../src/tbuartICE40.v 15a>+≡ <17a 17c>
    uartICE40
    #( .SUBDIV16(subdiv16), .ADJUSTSAMPLEPOINT(adjsamplept))
    dut_rx
    (// Outputs
      .txpin( dummy_txpin ),
      .txbusy( dummy_txbusy ),
    // Inputs
      .clk (rx_clk ),
      .bitxce(bitxce_rx),
      .load( 1'b0 ),
      .d (0),
    /*AUTOINST*/);

```

The testbench uses a bit-serial loopback. Pads not simulated, so txpin inverted here.

```

17c <../src/tbuartICE40.v 15a>+≡ <17b 17d>
    assign rxpin = ~txpin;

```

Provide the 8 or 16 times bitrate clocks:

```

17d <../src/tbuartICE40.v 15a>+≡ <17c>
    always @(posedge tx_clk)
        bitxce_tx_cnt <= bitxce_tx_cnt + 1;
    always @(posedge rx_clk)
        bitxce_rx_cnt <= bitxce_rx_cnt + 1;
    assign bitxce_tx = bitxce_tx_cnt == 0 || 'BITLAX;
    assign bitxce_rx = bitxce_rx_cnt == 0 || 'BITLAX;
endmodule

```

7.1 Use of the test bench

I tested the module using the excellent iverilog/gtkwave combination, with a scruffy home-grown simulation library. No attempt is made to provide these elements to other readers. I recommend any users of uartICE40 to do their own testing, in their own environment. The testbench above could be used as a starting point.

Tested

8 variants are tested:

High-level code	8-times oversampling	bitxce active each cycle
High-level code	8-times oversampling	bitxce active one of 8 cycles
High-level code	16-times oversampling	bitxce active each cycle
High-level code	16-times oversampling	bitxce active one of 8 cycles
Low-level code	8-times oversampling	bitxce active each cycle
Low-level code	8-times oversampling	bitxce active one of 8 cycles
Low-level code	16-times oversampling	bitxce active each cycle
Low-level code	16-times oversampling	bitxce active one of 8 cycles

Chapter 8

ICEstick

A top level module is needed for synthesis.

```
19 <./src/icestickuart.v 19>≡
/* Top level that just instantiates a UART in loopback mode in an icestick.
 * Assumptions: 12M clock. 115200 bps. 8N1 format.
 * Note: Needs retesting on hardware after code reorganization.
 */
/*
 * LogicCells:
 * 38 for uart proper
 * 1 for metastability removal rxpin
 * 1 for generation of constant 1'b1.
 * ---
 * 40 logicCells in total
 */
/*
PIO3_08
[x] --| |~ rxpinmeta1 -| |~| |~ rxpin -> UART
      >_|          |~| >_|

UART - txpin ->| |~|>o-[x] PIO03_07
              >_|

*/

module top
( inout PIO3_08,PIO3_07,PIO1_14,PIO1_02,GBIN6
);
wire [7:0] d;
wire      clk,cte1,rxpinmeta1,c_rxpinmeta1,rxpin;
reg [3:0] bitxcecmt;
/*AUTOWIRE*/

// One LUT consumed to get a constant 1.
// May get constant 1 from an unbonded pad instead.
assign cte1 = 1'b1;

// Clock pin
SB_GB_IO clockpin
( .PACKAGE_PIN(GBIN6),
```

```

        .GLOBAL_BUFFER_OUTPUT(clk));

// Transmit pin
SB_IO #( .PIN_TYPE(6'b011111)) // OUTPUT_REGISTERED_INVERTED/INPUT_LATCH
IO_tx
( .PACKAGE_PIN(PI03_07),
  .OUTPUT_CLK(clk),
  .D_OUT_0(txpin) );

// txbusy to LED0
SB_IO #( .PIN_TYPE(6'b010111)) // OUTPUT_REGISTERED/INPUT_LATCH
IO_txbusy
( .PACKAGE_PIN(PI01_14),
  .OUTPUT_CLK(clk),
  .D_OUT_0(txbusy) );

// bitxce to J2 pin 1 for debugging
SB_IO #( .PIN_TYPE(6'b010111)) // OUTPUT_REGISTERED/INPUT_LATCH
IO_bitxce
( .PACKAGE_PIN(PI01_02),
  .OUTPUT_CLK(clk),
  .D_OUT_0(bitxce) );

// Receive pin
SB_IO #( .PIN_TYPE(6'b000000)) // NO_OUTPUT/INPUT_REGISTERED
IO_rx
( .PACKAGE_PIN(PI03_08),
  .INPUT_CLK(clk),
  .D_IN_0(rxpinmeta1) );
// Metastability. I explicitly instantiate a LUT,
SB_LUT4 #( .LUT_INIT(16'haaaa))
cmb( .O(c_rxpinmeta1), .I3(1'b0), .I2(1'b0), .I1(1'b0), .I0(rxpinmeta1));
SB_DFF metareg( .Q(rxpin), .C(clk), .D(c_rxpinmeta1));

// Prescaler : 12000000/(115200*8) = 13.02, so make a counter
// 4 5 6 7 8 9 a b c d e f 10
always @(posedge clk)
    bitxcecnt <= bitxcecnt[3] ? 4'h4 : bitxcecnt+4'h1;
assign bitxce = bitxcecnt[3];
// The module proper
uartICE40 uart
(*AUTOINST*);

// Connect the uart in loopback:
assign load = bytercvd;
assign d = q;
endmodule

// Local Variables:
// verilog-library-directories:(" " ../fromrefdesign" )
// verilog-library-files:(" " ../PROJ/ice_simlib/ice_simlib.v" "uart.v" )
// verilog-library-extensions:(".v" )
// End:

```

Compilation results – Synplify, low level

When using the low level implementation, the following excerpt from the “placer.log” file show the size:

Final Design Statistics

Number of LUTs	:	38
Number of DFFs	:	34
Number of Carrys	:	18
Number of RAMs	:	0
Number of ROMs	:	0
Number of IOs	:	4
Number of GBIOs	:	1
Number of GBs	:	0
Number of WarmBoot	:	0
Number of PLLs	:	0

Device Utilization Summary

LogicCells	:	39/1280
PLBs	:	9/160
BRAMs	:	0/16
IOs and GBIOs	:	5/96
PLLs	:	0/1

This was when compiling with Synplify. Design resource usage is theoretically:

32	Uart proper
4	Predivider
1	Removal of metastability from rxpin
1	Generation of constant 1'b1
<hr/>	
38	Total

This matches the real result.

Compilation results – Synplify, high level

The same exercise for the high-level implementation:

Design Statistics after Packing

Number of LUTs	:	46
Number of DFFs	:	34
Number of Carrys	:	0

Device Utilization Summary after Packing

Sequential LogicCells		
LUT and DFF	:	34
LUT, DFF and CARRY	:	0
Combinational LogicCells		
Only LUT	:	12
CARRY Only	:	0
LUT with CARRY	:	0
LogicCells	:	46/1280
PLBs	:	7/160
BRAMs	:	0/16
IOs and GBIOs	:	5/96
PLLs	:	0/1

I2088: Phase 3, elapsed time : 0.2 (sec)

Phase 4

I2088: Phase 4, elapsed time : 0.1 (sec)

Phase 5

I2088: Phase 5, elapsed time : 0.0 (sec)

Phase 6

I2088: Phase 6, elapsed time : 12.9 (sec)

Final Design Statistics

Number of LUTs	:	46
Number of DFFs	:	34
Number of Carrys	:	0
Number of RAMs	:	0
Number of ROMs	:	0
Number of IOs	:	4
Number of GBIOs	:	1
Number of GBs	:	0
Number of WarmBoot	:	0
Number of PLLs	:	0

Device Utilization Summary

LogicCells	:	46/1280
PLBs	:	8/160
BRAMs	:	0/16
IOs and GBIOs	:	5/96
PLLs	:	0/1

Note that no carry-chain resources are used above. The high-level implementation use LUTs to find out if the transmit buffer is empty.

Conclusion

The low level implementation uses 32 logicCells, while the high level implementation uses 39 logicCells. This is a lot of effort (and near unreadable code) for a meagre saving of 7 logicCells. If I had been able to write a good high-level module from the very start, a lot of effort had been saved. However, the high-level module was made *after* the low-level module, and reflects the design choices of the low-level module. A conclusion must be: Think low-level, code high-level.

And this ends the documentation of a simulated and tested small asynchronous receiver/transmitter specifically written for iCE40 FPGAs.