# 460 UART

# Chip Specifications

**Brandon Torres**
**5/7/2015**

**Contents**

1. Introduction ..... 4
2. Applicable External Documents ..... 4
  2.1 Requirements ..... 4
  2.2 Specifications ..... 4
  2.3 Applicable External Document ..... 5
    2.3.1 RS232 ..... 5
    2.3.2 PicoBlaze (KCPSM3) ..... 5
3. Top Level Design ..... 6
  3.1 Description ..... 6
  3.2 Block Diagram ..... 6
  3.3 Data Flow Description ..... 7
  3.4 Logic Block Description ..... 7
    3.4.1 Receive ..... 7
    3.4.2 Transmit ..... 7
    3.4.3 KCPSM3 ..... 7
    3.4.4 8-to-256 Decoder ..... 8
    3.4.5 AISO (Asynchronous In Synchronous Out) ..... 8
  3.5 Clock ..... 8
  3.6 Reset ..... 8
  3.7 Chip Mode Definition ..... 8
4. Externally Developed Blocks ..... 9
  4.1 KCPSM3 (PicoBlaze Processor) ..... 9
    4.1.1 Description ..... 9
    4.1.2 Block Diagram ..... 9
    4.1.3 I/O ..... 9
    4.1.4 Instruction Set ..... 10
    4.1.5 Program ..... 10
5. Internally Developed Blocks ..... 12
  5.1 Baud Decoder ..... 12
    5.1.1 Description ..... 12
    5.1.2 Block Diagram ..... 12
    5.1.3 I/O ..... 12

5.1.4 Verification .......... 13
6. Chip Level Verification .......... 13
APPENDIX: Source Code .......... 14
Lab4_Top.v .......... 15
AISO.v .......... 17
Baud_Decoder.v .......... 18
Baud_Counter.v .......... 19
Bit_Counter.v .......... 20
Baud_Delay.v .......... 21
PISO.v .......... 22
User Constraints File .......... 23

## 1. Introduction

This document defines the specification for the UART chip (FPGA) which allows serial communication using the RS232 protocol.

## 2. Applicable External Documents

### 2.1 Requirements

The chip is required to communicate to other devices using the UART standard. The hardware should display a prompt on the start of each new line. It should also be capable of deleting a transmitted character. The chip should be able to operate with even/odd parity, 7/8 data bits, and at multiple baud rates.

### 2.2 Specifications

1. A prompt must be displayed on startup.

2. Receive data at a baud rate that is >= 300 and <= 921600 depending on the desired user speed.

3. The chip must be able to handle data that is either 7 or 8 bits depending on desired user settings.

4. The chip must be able to determine if there is a parity error if parity is enabled by the user.

5. The receive data must then be transmitted at the same baud rate that it is received, at the appropriate amount of data bits, and with a parity bit if enabled.

6. The backspace key must be a destructive delete. This means that the character that is located behind the cursor must be removed and the next character must be displayed in the position of the deleted character.

7. A carriage return or line feed input must jump to the next line and reset the cursor position. The prompt must also be displayed at the start of each new line.

8. The backspace key must not delete the prompt.

## 2.3 Applicable External Document

### 2.3.1 RS232

RS-232 is a standard for serial communication transmission of data. It formally defines the signals connecting between a DTE (data terminal equipment) such as a computer terminal, and a DCE (data circuit-terminating equipment, originally defined as data communication equipment), such as a modem. The RS-232 standard is commonly used in computer serial ports. The standard defines the electrical characteristics and timing of signals, the meaning of signals, and the physical size and pinout of connectors.

### 2.3.2 PicoBlaze (KCPSM3)

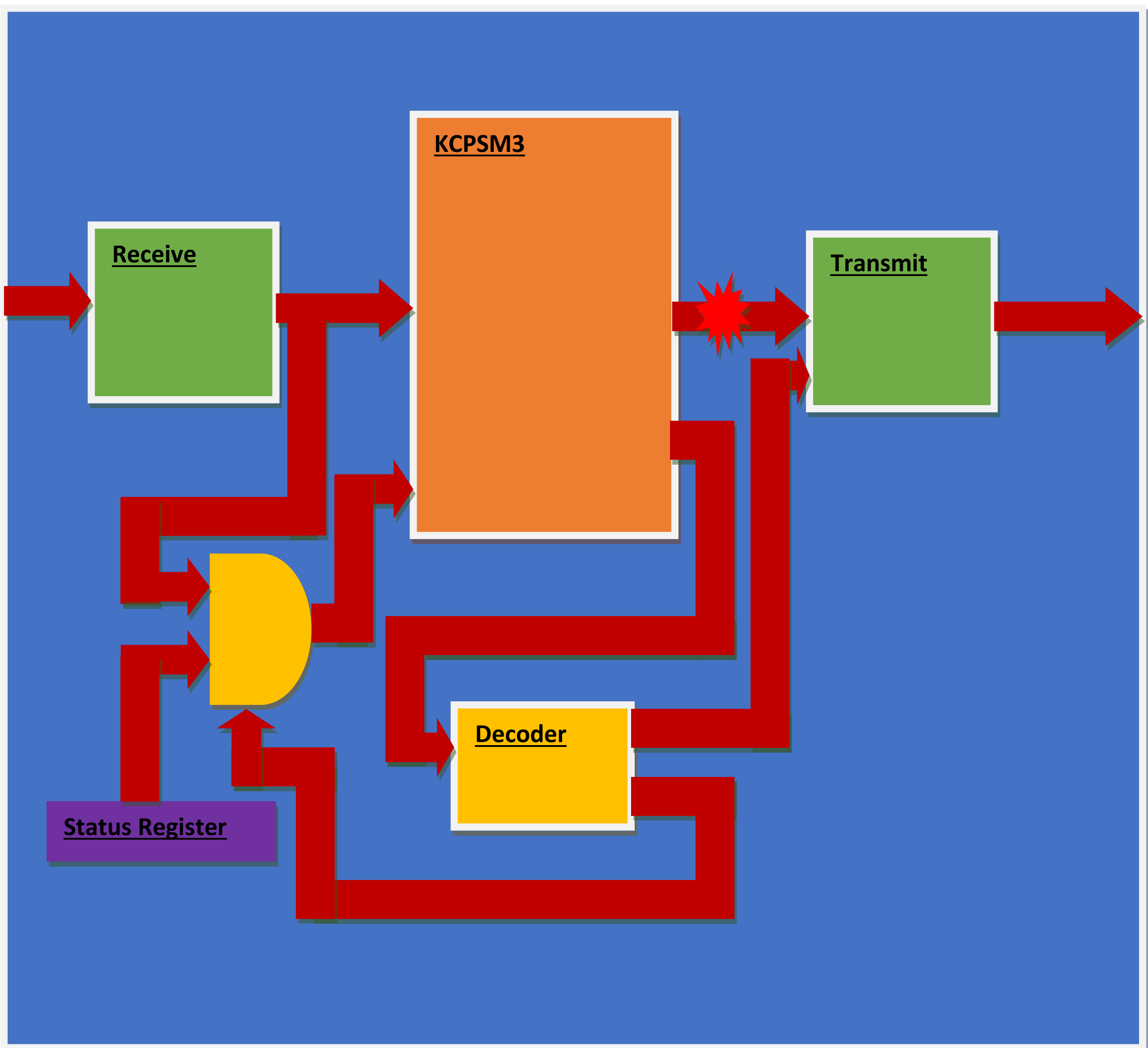
KCPSM3 is a very simple 8-bit microcontroller primarily for the Spartan-3 devices but also suitable for use in Virtex-II and Virtex-IIPRO devices. Although it could be used for processing of data, it is most likely to be employed in applications requiring a complex, but non-time critical state machine. Hence it has the name of '(K)constant Coded Programmable State Machine'. This revised version of popular KCPSM macro has still been developed with one dominant factor being held above all others - Size! The result is a microcontroller which occupies just 96 Spartan-3 Slices which is just 5% of the XC3S200 device and less than 0.3% of the XC3S5000 device. Together with this small amount of logic, a single block RAM is used to form a ROM store for a program of up to 1024 instructions. Even with such size constraints, the performance is respectable at approximately 43 to 66 MIPS depending on device type and speed grade.

## 3. Top Level Design

### 3.1 Description

The transmit and the receive portion of the UART function independently of each other. The receive is responsible for identifying when there is an incoming transmission. It will then capture the data bits and send them to the PicoBlaze once transmission is complete. The PicoBlaze will then the appropriate data to be output to the transmit engine. This will usually just be the received data. If a special character is entered such as a backspace then the PicoBlaze will then take different action depending on the character. Once the transmit has received the data it will then send the data to the UART.

### 3.2 Block Diagram



### 3.3 Data Flow Description

When the chip has detected an incoming transmission it will trigger the receive engine to begin sampling data. The data sampled will then be sampled every bit time for 11 bits. This sampling will occur at the halfway point of the bit time of the data that is being received. Once the receive engine has received 11 bits it will stop sampling. It will send an interrupt to the KCPSM3. The KCPSM3 will then read the data from the receive engine. It will then attempt to send the data to the receive engine. The KCPSM3 will check the status register to check if the transmit engine is ready. If it is not then the KCPSM3 will continue to poll the status register until the transmit engine is ready. Once ready the data that the KCPSM3 read from the receive engine will be sent to the transmit engine for transmission. The transmit engine will then begin to transmit the data one bit at a time every bit time. This will continue until all 11 data bits are transmitted. Once the transmission has been completed the chip will be idle until an incoming transmission is detected again.

### 3.4 Logic Block Description

#### 3.4.1 Receive

The receive engine has the following inputs: Clock, Reset, BaudVal, SerialDataIn. It has the following outputs: ReceiveReady, DataOut. When the start bit is detected by the receive engine it will start its baud decoder in half a bit time. This will allow the receive engine to sample in the middle of the bit time of the data that is being received. This will minimize the chance of sampling at an incorrect bit. If reset is pressed all internal registers of the receive engine are set to 0.

#### 3.4.2 Transmit

The transmit engine has the following input: Clock, Reset, BaudVal, DataIn. It has the following outputs: TransmitRdy, SerialDataOut. Once the transmit engine has received data via DataIn TransmitRdy will go low and it will begin to output the data one bit at a time until all 11 bits have been transmitted. Once all the bits have been transmitted then TransmitRdy will go high. If reset is pressed all internal registers of the transmit engine are set to 0.

#### 3.4.3 KCPSM3

When initialized the processor will output a prompt to the transmit engine. Once completed it will sit idle until the interrupt is triggered. Once the interrupt is triggered it will read port 2 and store that data in a register. The processor will check this data against predefined special characters. If the data that was received was a special character then processor will handle that character appropriately. Otherwise the processor will then check the TransmitRdy flag. If TransmitRdy is high then the data that was stored previously will be sent to the transmit engine. If it is not the processor will wait until the TransmitRdy flag is high. If reset is pressed then the program counter is set to 0.

### 3.4.4 8-to-256 Decoder

The decoder has the following inputs: PortId and EventStrobe. It has 1 output: portStrobe. The decoder takes in the PortId and EventStrobe from the KCPSM3. When EventStrobe is high it will enable to appropriate portStrobe according to decoder logic. This will then either drive the transmit engine or select an input for the KCPSM3.

### 3.4.5 AISO (Asynchronous In Synchronous Out)

This module is used to prevent an issue in which flops in the chip could go metastable. This can be caused by an asynchronous reset signal. The AISO fixes this issue by connecting the output of one flop to the input of another flop. The output of the second flop is then used as the reset signal.

### 3.5 Clock

The clock is provided by the FPGA. The Spartan 3E has a clock frequency of 50 MHz. The clock period is 20ns.

### 3.6 Reset

All resets in the design are low active. The asynchronous reset signal is fed into the AISO. The AISO then provides a synchronous reset signal to the chip.

### 3.7 Chip Mode Definition

The design features the ability for 12 different baud rates, 7/8 data bits, and even/odd parity. The configuration may only be changed on reset. The pin configuration is shown on the tables below.

| Mode Configuration | 7/8 | PE | O/E |
|---|---|---|---|
| 7N1 | 0 | 0 | 0 |
| 7N1 | 0 | 0 | 1 |
| 7E1 | 0 | 1 | 0 |
| 7O1 | 0 | 1 | 1 |
| 8N1 | 1 | 0 | 0 |
| 8N1 | 1 | 0 | 1 |
| 8E1 | 1 | 1 | 0 |
| 8O1 | 1 | 1 | 1 |

| Sel | Baud Rate | Bit Time(sec) | # of Clocks |
|---|---|---|---|
| 0 | 300 | 0.003333333 | 166666 |
| 1 | 600 | 0.001666667 | 83332 |
| 2 | 1200 | 0.000833333 | 41666 |
| 3 | 2400 | 0.000416667 | 20832 |
| 4 | 4800 | 0.000208333 | 10416 |
| 5 | 9600 | 0.000104167 | 5207 |
| 6 | 19200 | 5.20833E-05 | 2603 |
| 7 | 28400 | 3.52113E-05 | 1760 |
| 8 | 57600 | 1.73611E-05 | 867 |
| 9 | 115200 | 8.68056E-06 | 433 |
| 10 | 230400 | 4.34028E-06 | 216 |
| 11 | 460800 | 2.17014E-06 | 108 |
| 12 | 921600 | 1.08507E-06 | 53 |

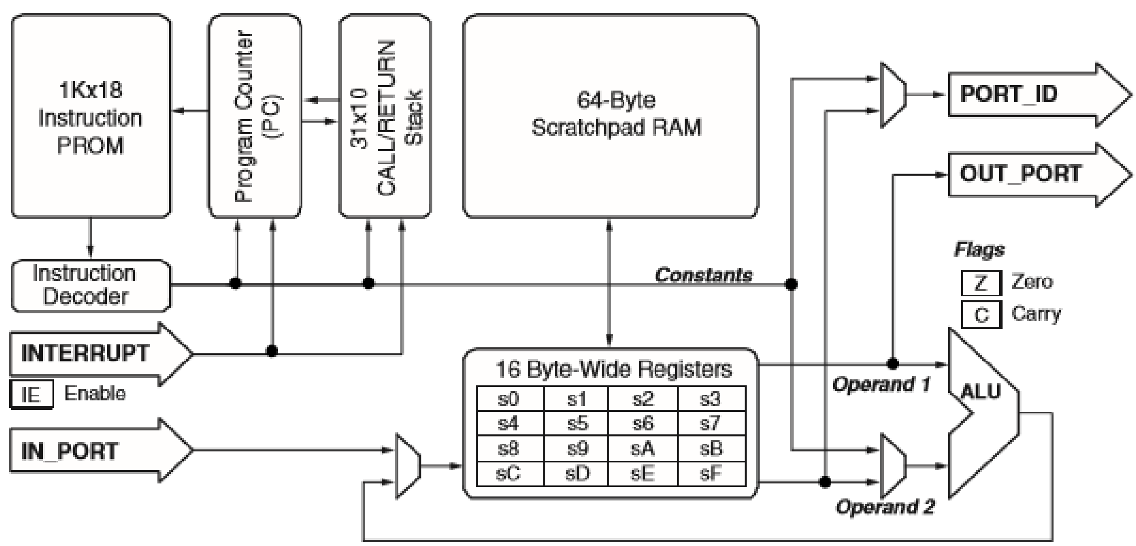
# 4. Externally Developed Blocks

## 4.1 KCPSM3 (PicoBlaze Processor)

### 4.1.1 Description

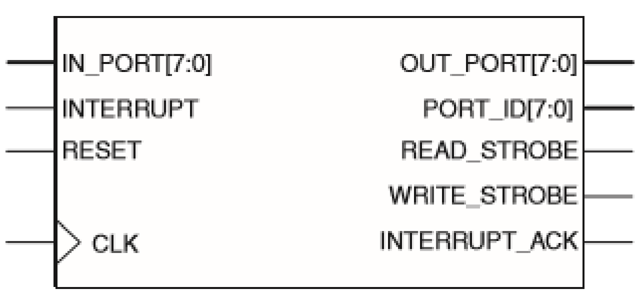
The PicoBlaze is responsible for handling the data after the receive engine has finished receiving it. Once the PicoBlaze receive the data it is responsible for it will then determine whether it needs to echo the data or take special action if a special character is entered. Currently the only special characters that are recognized are “Backspace”, “Carriage Return”, and “Line Feed”. More could be implemented at a later time. Once the character has been determined then the appropriate data is sent to the transmit engine. The PicoBlaze features various instructions and registers. Only very limited amounts are used for the purposes of the UART.

### 4.1.2 Block Diagram




### 4.1.3 I/O

The PicoBlaze interfaces with other modules in the following manner.



### 4.1.4 Instruction Set

'X' and 'Y' refer to the definition of the storage registers 's' in the range 0 to F.
'kk' represents a constant value in the range 00 to FF.
'aaa' represents an address in the range 000 to 3FF.
'pp' represents a port address in the range 00 to FF.
'ss' represents an internal storage address in the range 00 to 3F.

**Program Control Group**

```
JUMP aaa
JUMP Z,aaa
JUMP NZ,aaa
JUMP C,aaa
JUMP NC,aaa

CALL aaa
CALL Z,aaa
CALL NZ,aaa
CALL C,aaa
CALL NC,aaa

RETURN
RETURN Z
RETURN NZ
RETURN C
RETURN NC
```

Note that call and return supports up to a stack depth of 31.

**Arithmetic Group**

```
ADD sX,kk
ADDCY sX,kk
SUB sX,kk
SUBCY sX,kk
COMPARE sX,kk

ADD sX,sY
ADDCY sX,sY
SUB sX,sY
SUBCY sX,sY
COMPARE sX,sY
```

**Interrupt Group**

```
RETURNI ENABLE
RETURNI DISABLE

ENABLE INTERRUPT
DISABLE INTERRUPT
```

**Logical Group**

```
LOAD sX,kk
AND sX,kk
OR sX,kk
XOR sX,kk
TEST sX,kk

LOAD sX,sY
AND sX,sY
OR sX,sY
XOR sX,sY
TEST sX,sY
```

**Storage Group**

```
STORE sX,ss
STORE sX,(sY)
FETCH sX,ss
FETCH sX,(sY)
```

**Shift and Rotate Group**

```
SR0 sX
SR1 sX
SRX sX
SRA sX
RR sX

SL0 sX
SL1 sX
SLX sX
SLA sX
RL sX
```

**Input/Output Group**

```
INPUT sX,pp
INPUT sX,(sY)
OUTPUT sX,pp
OUTPUT sX,(sY)
```

### 4.1.5 Program

The PicoBlaze is running the following program for operation of the UART:

```
        NAMEREG sA, transmit_data
        NAMEREG sB, status
        NAMEREG sC, counter
         ;
start:          LOAD counter,00                          ;reset input counter
                LOAD transmit_data,00
                ENABLE INTERRUPT
                OUTPUT transmit_data,01
                CALL NEWLINE

loop:           ADD s1,0
                JUMP loop

NEWLINE:        LOAD transmit_data,0A                    ;(Line Feed)
                CALL TRANSMIT

                LOAD transmit_data,0D                    ;(Carriage Return)
                CALL TRANSMIT

                 LOAD transmit_data,34                   ;4
                CALL TRANSMIT

                LOAD transmit_data,36                    ;6
                CALL TRANSMIT

                LOAD transmit_data,30                    ;0
                CALL TRANSMIT
```

```
                LOAD transmit_data,3E                   ;>
                CALL TRANSMIT

                LOAD transmit_data,20                   ;(SPACE)
                CALL TRANSMIT

                LOAD transmit_data,00                   ;(NULL)

                LOAD counter,00
                RETURN


                ADDRESS 280
TRANSMIT:       INPUT status,00
                AND   status,02
                 COMPARE status,02
                JUMP NZ,TRANSMIT
                OUTPUT transmit_data,01
                LOAD transmit_data,00
                RETURN



                ADDRESS 2B0
int_routine:    INPUT transmit_data,02
                COMPARE transmit_data,08
                JUMP NZ,NOT_BACKSPACE
                COMPARE counter,00
                JUMP Z , NULL_CHAR
                CALL TRANSMIT
                LOAD transmit_data,20
                CALL TRANSMIT
                LOAD transmit_data,08
                CALL TRANSMIT
                SUB counter,01
NOT_BACKSPACE:  COMPARE transmit_data,0A
                CALL Z,NEWLINE
                COMPARE transmit_data,0D
                CALL Z,NEWLINE
                COMPARE transmit_data,00
                JUMP Z,NULL_CHAR
                CALL TRANSMIT
                ADD counter,01
NULL_CHAR:       RETURNI ENABLE

                 ADDRESS 3FF                                 ;set interrupt vector
                 JUMP int_routine
```

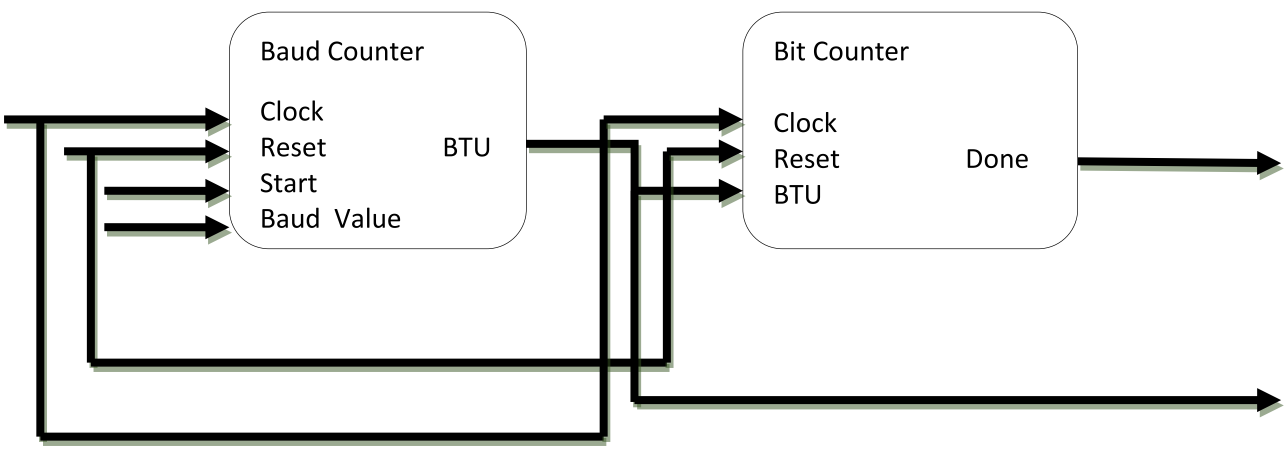
## 5. Internally Developed Blocks

### 5.1 Baud Decoder

#### 5.1.1 Description

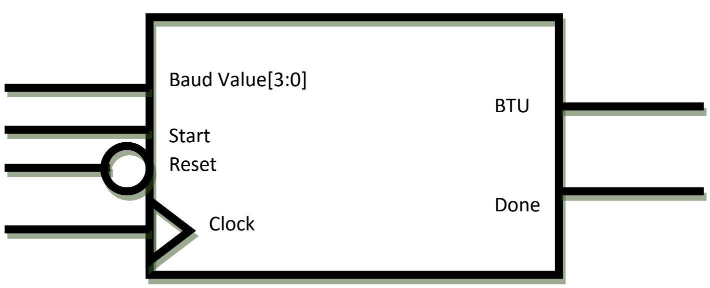
The baud decoder is used so the modules that interact with UART communication are able to identify when a bit time has elapsed and to identify when 11 bit times have elapsed. The baud decoder consists of two modules. A baud counter that determines when a bit time has elapsed and a bit counter that determines when 11 bits have passed.

#### 5.1.2 Block Diagram

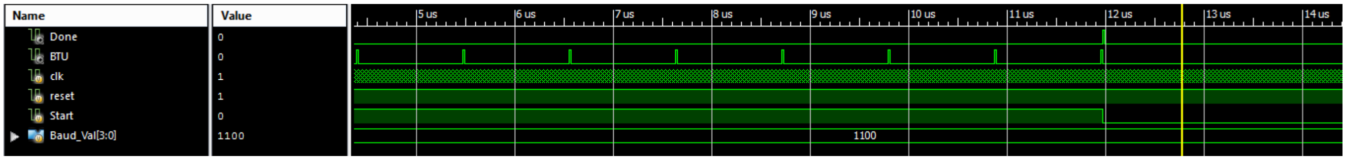



#### 5.1.3 I/O

The baud decoder interface with other modules using the following inputs and outputs:



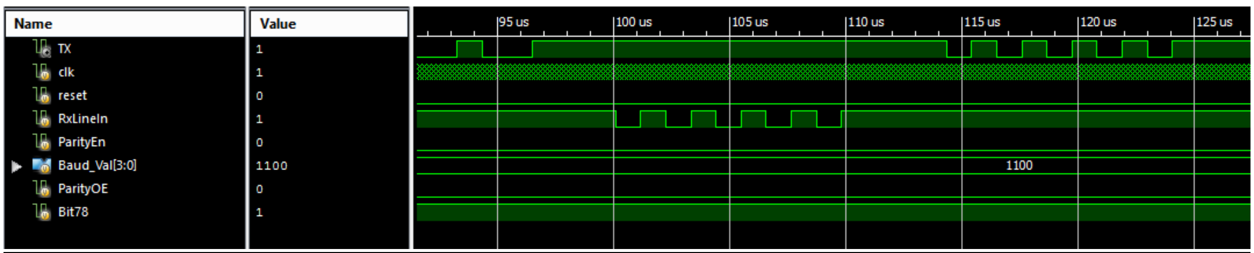
### 5.1.4 Verification




Here it can be seen that the in mode 12 the baud decoder will issue a BTU every 1.08us. After 11 BTU the decoder will signal that it is done and will wait until start is high again.

## 6. Chip Level Verification

The test shows the chip receiving data in the RxLineIn. Once the data transfer has completed the TX can be seen echoing the data that was received in the RxLineIn.



# APPENDIX: Source Code

**Lab4_Top.v**

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////////////////////
//      Author:        Brandon Torres
//      Email:         Brandon0torres@gmail.com
//      Filename:      Lab4_top.v
//
//      Notes:         Lab 4 contains both the ability to receive and to transmit.
//                     The receive data is sent back to the transmit to be echoed.
/////////////////////////////////////////////////////////////////////////////////
module Lab4_top(clk,reset,RxLineIn,ParityEn,Baud_Val,ParityOE,Bit78,TX);
        input clk,reset,RxLineIn,ParityEn,ParityOE,Bit78;
        input[3:0] Baud_Val;
        output TX,overflow,parityErrorOut;

        wire sync_reset,Done,Start,TXRDY,readStrobe,writeStrobe,BTU,A,B,C,D,int_ack,
                RxDone,RxStart,RxStartDelay,RxBTU,RxRdy,receiveParity,parityValue,parityError;
        wire[255:0] portStrobe;
        wire[7:0] portID,picoOut,holdOut;
        wire[10:0]transmitData,RxData;
        wire[7:0] status;
        reg [20:0] count_val;
        reg [7:0] PicoIn;
        reg [2:0] mode;
        reg delayReg;


        AISO sync(.clk(clk),.async_in(reset),.sync_out(sync_reset));
        dec8to255 portad(.in(portID),.en(eventStrobe),.out(portStrobe));

        embedded_kcpsm3 proc(
                              .port_id(portID),
                              .write_strobe(writeStrobe),
                              .read_strobe(readStrobe),
                              .out_port(picoOut),
                              .in_port(PicoIn),
                              .interrupt(RxRdy),
                              .interrupt_ack(int_ack),
                              .reset(~sync_reset),
                              .clk(clk));



///////////////////////////////////////////////////////////////////////////////////////////////
//Tx Engine
///////////////////////////////////////////////////////////////////////////////////////////////
//Baud decoder for tx engine
Baud_Decoder
baud(.clk(clk),.reset(sync_reset),.Start(Start),.Done(Done),.Baud_Val(Baud_Val),.BTU(BTU));
reg8 pico_hold(.clk(clk),.reset(sync_reset),.ld(portStrobe[1]),.Din(picoOut),.Dout(holdOut));
PISO11transmit(.clk(clk),.reset(sync_reset),.shift(BTU),.ld(delayReg),.Din(transmitData),
               .SDI(1'b1),.SDO(TX),.Data());

// Signals start of transmit
RSFlop TransmitStart(.clk(clk),.reset(sync_reset),.R(Done),.S(delayReg),.Out(Start));
// Signals that Tx engine is ready to transmit
RSFlop TXReady(.clk(clk),.reset(sync_reset),.R(writeStrobe),.S(Done),.Out(TXRDY));

//Delay register that sends ld command to Baud decoder and PISO11
always @(posedge clk, negedge sync_reset)
        if(!sync_reset)
                delayReg<=1'b0;
        else
                delayReg<=portStrobe[1];
///////////////////////////////////////////////////////////////////////////////////////////////
//End Tx Engine
///////////////////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////////////////////
//Rx Engine
////////////////////////////////////////////////////////////////////////////////////////////////
Baud_Delay
Delay(.clk(clk),.reset(sync_reset),.Start(RxStart),.Baud_Val(Baud_Val),.BTU(RxStartDelay));

Baud_Decoder
Receivebaud(.clk(clk),.reset(sync_reset),.Start(RxStartDelay),.Done(RxDone),.Baud_Val(Baud_Val),
            .BTU(RxBTU)); //Baud decoder for rx engine

PISO11Receive(.clk(clk),.reset(sync_reset),.shift(RxBTU),.ld(1'b0),.Din(11'b0),.SDI(RxLineIn),
              .SDO(),.Data(RxData));

// Signals the start of receive
RSFlop ReceiveStart(.clk(clk),.reset(sync_reset),.R(RxDone),.S(~RxLineIn),.Out(RxStart));
// Signals the start of receive
RSFlop ReceiveRdy(.clk(clk),.reset(sync_reset),.R(int_ack),.S(RxDone),.Out(RxRdy));

////////////////////////////////////////////////////////////////////////////////////////////////
//End Rx Engine
////////////////////////////////////////////////////////////////////////////////////////////////

        //PicoBlaze input selector
        always @(portStrobe)
                        case(portStrobe)
                                256'b01: PicoIn<=status;
                                256'b100: PicoIn<=RxData[7:0];
                                default: PicoIn<=PicoIn;
                        endcase

        //Sets the bits according to parity and 7/8
        always @(posedge clk,negedge sync_reset)
                if(!sync_reset)
                        case({Bit78,ParityEn,ParityOE})
                                3'b000: mode<=2'b11;                //7N1
                                3'b001: mode<=2'b11;                //7N1
                                3'b010: mode<={1'b1,A};             //7E1
                                3'b011: mode<={1'b1,B};             //7O1
                                3'b100: mode<={1'b1,holdOut[7]};    //8N1
                                3'b101: mode<={1'b1,holdOut[7]};    //8N1
                                3'b110: mode<={C,holdOut[7]};       //8E1
                                3'b111: mode<={D,holdOut[7]};       //8O1
                                default:mode<={1'b1,holdOut[7]};
                        endcase
                else
                        mode<=mode;
        //PicoBlaze want to read or right
        assign eventStrobe = readStrobe | writeStrobe;

        //Sets data to be sent to PISO11
        assign transmitData= {mode,holdOut[6:0],2'b01};
        //Error
        assign overflow= (RxRdy&RxStart) ? 1'b1 : 1'b0;
        assign receiveParity= Bit78 ? ^RxData[7:0] : ^RxData[6:0];
        assign ParityValue=Bit78 ? RxData[8] : RxData[7];
        assign parityError= ParityOE ?~(receiveParity^ParityValue) : (receiveParity^ParityValue);
        assign ParityErrorOut= ParityEn ? parityError : 1'b0;
        //Status
        assign status={6'b0,TXRDY,1'b0};

        //Parity assignments
   assign A=^holdOut[6:0];  //Even parity 7-bit
        assign B=~^holdOut[6:0]; //Odd  parity 7-bit
        assign C=^holdOut[7:0];  //Even parity 8-bit
        assign D=~^holdOut[7:0]; //Odd  parity 8-bit
endmodule
```

**AISO.v**

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//      Author:         Brandon Torres
//      Email:          Brandon0torres@gmail.com
//      Filename:       AISO.v
//
//      Notes:          The module has two flip flops that are connected to the same clock
//                      as the rest of the modules. This module will take an asynchronous
//                      signal in and then output a synchronous signal.
//////////////////////////////////////////////////////////////////////////////////
module AISO(clk,async_in,sync_out);
        input clk,async_in;
        output wire sync_out;
        reg inreg,outreg;

        always @(posedge clk, posedge async_in)begin
                if(async_in)
                        inreg<=0;
                else
                        inreg<=1;

        end

        always @(posedge clk)
                outreg<=inreg;

        assign sync_out=outreg;


endmodule
```

**Baud_Decoder.v**

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//      Author:        Brandon Torres
//      Email:         Brandon0torres@gmail.com
//      Filename:      Baud_Decoder.v
//
//      Notes:         The module interconnects Baud_Counter and Bit_Counter. Bit_Counter
//                             counts every BTU and sets Done once 11 BTU have occurred.
////////////////////////////////////////////////////////////////////////////////
module Baud_Decoder(clk,reset,Start,Done,Baud_Val,BTU);
        input clk,reset,Start;
        input[3:0] Baud_Val;
        output wire Done,BTU;

        Baud_Counter bc(.clk(clk),.reset(reset),.Start(Start),.Baud_Val(Baud_Val),
                        .BTU(BTU));

        Bit_Counter bitc(.clk(clk),.reset(reset),.BTU(BTU),.Done(Done));


Endmodule
```

## Baud_Counter.v

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////////////////////
//      Author:         Brandon Torres
//      Email:          Brandon0torres@gmail.com
//      Filename:       Baud_Counter.v
//
//      Notes:          The Baud rate is only set on reset and cannot be changed during run.
//                      The module sets the Baud rate for the UART depending on the user
//                      input. Once the Baud time has passed then BTU is set for one clock.
/////////////////////////////////////////////////////////////////////////////////
module Baud_Counter(clk,reset,Start,Baud_Val,BTU);
         input clk,reset,Start;
         input [3:0] Baud_Val;
         output wire BTU;

         reg [17:0] count_val;
         reg [17:0] q_reg;
         wire[17:0] q_next;

         always @(posedge clk, negedge reset)
               if(!reset)begin
                      q_reg<=0;
                       case(Baud_Val)
                             4'b0000: count_val<=166666; //300 Baud
                             4'b0001: count_val<=83332;  //600 Baud
                             4'b0010: count_val<=41666;  //1200 Baud
                             4'b0011: count_val<=20832;  //2400 Baud
                             4'b0100: count_val<=10416;  //4800 Baud
                             4'b0101: count_val<=5207;   //9600 Baud
                             4'b0110: count_val<=2603;   //19200 Baud
                             4'b0111: count_val<=1760;   //28400 Baud
                             4'b1000: count_val<=867;    //57600 Baud
                             4'b1001: count_val<=433;    //115200 Baud
                             4'b1010: count_val<=216;    //230400 Baud
                             4'b1011: count_val<=108;    //460800 Baud
                             4'b1100: count_val<=53;     //921600 Baud
                             default: count_val<=166666; //Default is 300 Baud
                      endcase
               end
               else if(BTU)
                      q_reg<=0;
               else if(Start)
                      q_reg<=q_next;
               else
                      q_reg<=0;

        assign q_next=q_reg+1;

        //set tick when after each BTU
        assign BTU= (q_reg==count_val) ? 1'b1 : 1'b0;


endmodule
```

## Bit_Counter.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//      Author:        Brandon Torres
//      Email:         Brandon0torres@gmail.com
//      Filename:      Bit_Counter.v
//
//      Notes:         The module is a simple counter that increments every time it
//                     receives BTU. Once eleven BTU have occurred then the Done signal is
//                     set.
////////////////////////////////////////////////////////////////////////////////
module Bit_Counter(clk,reset,BTU,Done);
        input clk,reset,BTU;
        output wire Done;
        reg [3:0] q_reg;

        always @(posedge clk, negedge reset)
                if(!reset)
                        q_reg<=4'b0;
                else if(BTU)
                        q_reg<=q_reg+1;
                else if(q_reg==11)
                        q_reg<=4'b0;
                else
                        q_reg<=q_reg;
        assign Done=(q_reg==11) ? 1'b1 : 1'b0;

endmodule
```

## Baud_Delay.v

```
////////////////////////////////////////////////////////////////////////////////
//      Author:        Brandon Torres
//      Email:         Brandon0torres@gmail.com
//      Filename:      Baud_Delay.v
//
//      Notes:         The Baud rate is only set on reset and cannot be changed during run.
//                     The module sets the Baud rate for the UART depending on the user
//                     input. Once the Baud time has passed then BTU is set for one clock.
////////////////////////////////////////////////////////////////////////////////
module Baud_Delay(clk,reset,Start,Baud_Val,BTU);
        input clk,reset,Start;
        input [3:0] Baud_Val;
        output wire BTU;

        reg [17:0] count_val;
        reg [17:0] q_reg;
        wire[17:0] q_next;

        always @(posedge clk, negedge reset)
              if(!reset)begin
                     q_reg<=0;
                      case(Baud_Val)
                            4'b0000: count_val<=166666/2; //300 Baud
                            4'b0001: count_val<=83332/2;  //600 Baud
                            4'b0010: count_val<=41666/2;  //1200 Baud
                            4'b0011: count_val<=20832/2;  //2400 Baud
                            4'b0100: count_val<=10416/2;  //4800 Baud
                            4'b0101: count_val<=5207/2;   //9600 Baud
                            4'b0110: count_val<=2603/2;   //19200 Baud
                            4'b0111: count_val<=1760/2;   //28400 Baud
                            4'b1000: count_val<=867/2;    //57600 Baud
                            4'b1001: count_val<=433/2;    //115200 Baud
                            4'b1010: count_val<=216/2;    //230400 Baud
                            4'b1011: count_val<=108/2;    //460800 Baud
                            4'b1100: count_val<=53/2;     //921600 Baud
                            default: count_val<=166666/2; //Default is 300 Baud
                     endcase
              end
              else if(!Start)
                     q_reg<=0;
              else if(BTU)
                     q_reg<=q_reg;
              else
                     q_reg<=q_next;

       assign q_next=q_reg+1;
       //set tick when after each BTU
       assign BTU= (q_reg==count_val) ? 1'b1 : 1'b0;


endmodule
```

**PISO.v**

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////////////////////
//      Author:         Brandon Torres
//      Email:          Brandon0torres@gmail.com
//      Filename:       PISO11.v
//
//      Notes:          The register is an eleven bit register that can be loaded with
//                      serial in and parallel in. The LSB of the data is the output of the
//                      register.
/////////////////////////////////////////////////////////////////////////////////
 module PISO11(clk,reset,shift,ld,Din,SDI,SDO,Data);
        input clk, reset, shift, ld, SDI;
        input [10:0] Din;
        output wire SDO;
        output reg [10:0] Data;

        //Register that holds 11bits and shifts left in SDI when shift is enabled
        always @(posedge clk, negedge reset)
                if (!reset)
                        Data <= 10'b1111111111;
                else if(ld)
                        Data <= Din;
                else if(shift)
                        Data <= {SDI,Data[10:1]};
                else
                        Data <= Data;

        assign SDO = Data[0];



endmodule
```

## User Constraints File

```
# Pinouts for Nexys2 Board  (Spartan3-500E)

# Start I/O Pin Assignments
NET "clk"              LOC = "b8"   ;

# Slide Switches
NET "Bit78"             LOC = "g18"  ;
NET "ParityEn"          LOC = "h18"  ;
NET "ParityOE"          LOC = "k18"  ;
NET "Baud_Val[0]"       LOC = "l14"  ;
NET "Baud_Val[1]"       LOC = "l13"  ;
NET "Baud_Val[2]"       LOC = "n17"  ;
NET "Baud_Val[3]"       LOC = "r17"  ;

# LEDs
NET "overflow"         LOC = "j14"    ;
NET "parityErrorOut"    LOC = "j15"    ;
# Push Buttons
NET "reset"             LOC = "h13"  ;

# RS-232
NET "TX"                 LOC = "p9"   ;
NET "RX"                 LOC = "p9"   ;
```