**Project 4 - Full UART Engine**

Programmer: Victor Espinoza

SID: 010657450

CECS 460, Section 1; Tu/Th 9:30 - 11:45 A.M.

Lab Section 2

Due: Thursday, November 19, 2015

Table of Contents

[1. Introduction: 4](#_Toc435643092)

[1.1.1 Requirements 4](#_Toc435643093)

[2. Applicable Documents: 4](#_Toc435643094)

[2.1.1 Applicable External Documents 4](#_Toc435643095)

[2.2.2 Applicable Internal Documents 4](#_Toc435643096)

[3. Requirements: 4](#_Toc435643097)

[3.1.1 Performance Requirements 4](#_Toc435643098)

[3.1.2 Interface Requirements 4](#_Toc435643099)

[3.1.3 Partitioning Requirements 5](#_Toc435643100)

[3.1.4 Physical Requirements 5](#_Toc435643101)

[3.1.5 Power Requirements 5](#_Toc435643102)

[3.1.6 Environmental Requirements 5](#_Toc435643103)

[4. Top Level Design: 5](#_Toc435643104)

[4.1.1 Top Level / Data Flow Description 5](#_Toc435643105)

[4.1.2 Block Diagram 6](#_Toc435643106)

[4.1.3 I/O 8](#_Toc435643107)

[4.1.4 Clocks 8](#_Toc435643108)

[4.1.5 Resets 8](#_Toc435643109)

[4.1.6 Software 8](#_Toc435643110)

[5. Externally Developed Blocks: 8](#_Toc435643111)

[5.1.1 Description 8](#_Toc435643112)

[5.1.2 Block Diagram 9](#_Toc435643113)

[5.1.3 I/O 10](#_Toc435643114)

[5.1.4 Register Map 11](#_Toc435643115)

[6. Internally Developed Blocks: 11](#_Toc435643116)

[6.1.1 UART Engine Description 11](#_Toc435643117)

[6.1.2 UART Transmit Description 12](#_Toc435643118)

[6.1.3 UART Receive Description 13](#_Toc435643119)

[6.1.4 PicoBlaze Assembly 14](#_Toc435643120)

[6.1.5 PicoBlaze Assembly Description 26](#_Toc435643121)

[6.1.6 Block Diagram 27](#_Toc435643122)

[6.1.7 I/O 29](#_Toc435643123)

[6.1.8 Register Map 29](#_Toc435643124)

[6.1.9 Transmit Engine Verification 29](#_Toc435643125)

[6.1.10 Receive Engine Verification 30](#_Toc435643126)

[7. Chip Level Verification: 30](#_Toc435643127)

[8. Chip Level Test: 30](#_Toc435643128)

# 1. Introduction:

The SOC is using a Universal Asynchronous Receiver Transmitter (UART) interface to receive and transmit data. The user sets up an agreed upon connection with the SOC and then opens up a terminal window. Upon reset, a banner is displayed and so is a command prompt (\*>). The user can then enter a character on the keyboard and its data is sent to the PicoBlaze (an 8-bit microcontroller) to process the data. If the character is a carriage return or a line feed character, then the PicoBlaze will have the UART Engine transmit a new line command and re-issue the prompt to the terminal window. If the character is an asterisk (\*), then the PicoBlaze will have the UART Engine transmit the hometown of the developer (Porterville, CA) to the terminal window. Any other valid ASCII character that the user enters on the keyboard will be echoed to the terminal. The PicoBlaze is also able to process the status of the UART engine and informs the user whenever an overflow error or a parity error occurs within the UART communication. The status flags are cleared every time that the PicoBlaze reads the status of the UART engine.

## 1.1.1 Requirements

1. The software should service the RX engine. Coming out of reset the PicoBlaze should transmit a prompt to the screen. Whenever a byte is successfully received by the RX engine the PicoBlaze will read the byte.

2. If it is a back space (0x08) there should be a destructive delete (as long as there are characters to delete).

3. When the PicoBlaze receives a CR or LF it should respond with a newline and a prompt.

4. An asterisk ‘\*’ should cause the UART to respond with your home town.

5. Every other printable ASCII character should be echoed.

# 2. Applicable Documents:

## 2.1.1 Applicable External Documents

[PicoBlaze 8-Bit Embedded Microcontroller User's Guide](http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf)

## 2.2.2 Applicable Internal Documents

There are no applicable internal documents at this time.

# 3. Requirements:

## 3.1.1 Performance Requirements

The System On Chip (SOC) will be communicate using a Universal Asynchronous Receiver Transmitter (UART) interface. It will be able to communicate at the following baud rates: 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, and 460800.

## 3.1.2 Interface Requirements

The SOC will be using a Universal Asynchronous Receiver Transmitter (UART) interface.

## 3.1.3 Partitioning Requirements

After testing the design, I concluded that the baud rate of 921600 yielded inconsistent results in transmitting and receiving data using a UART communication scheme, so I decided to exclude this baud rate from the available baud rates that my SOC can communicate at.

## 3.1.4 Physical Requirements

This section should define any limitations placed on the design with respect to gate count, pin count, etc.

## 3.1.5 Power Requirements

A 3.3V power supply will drive the SOC.

## 3.1.6 Environmental Requirements

The SOC will be able to operate between 0**°** C – 70**°** C.

# 4. Top Level Design:

## 4.1.1 Top Level / Data Flow Description

The project consists of six different blocks. The first block in the Block Diagram is the Asynchronous In, Synchronous Out Synchronization Circuit. This makes it so that the reset button on the Nexys 2 Board (which is high-active) generates a low signal every time it is activated. This is because all of the reset logic in the system (excluding the external PicoBlaze microcontroller is low-active). The output of my AISO (rstsb) goes to all of the other blocks in the design. Because the PicoBlaze has a high-active reset, I needed to invert the rstsb output before I connected the rstsb signal to the PicoBlaze.

The second block in the design is the PicoBlaze. I use the PicoBlaze to process all of the data coming in from the Universal Asynchronous Receiver Transmitter interface terminal. I also use the PicoBlaze to determine what data is going to be transmitted to the terminal. The port\_id, write\_strobe, and read\_strobe outputs go into the third block, which is a decoder that I use to determine what data to write to the terminal and what data to read from the UART interface.

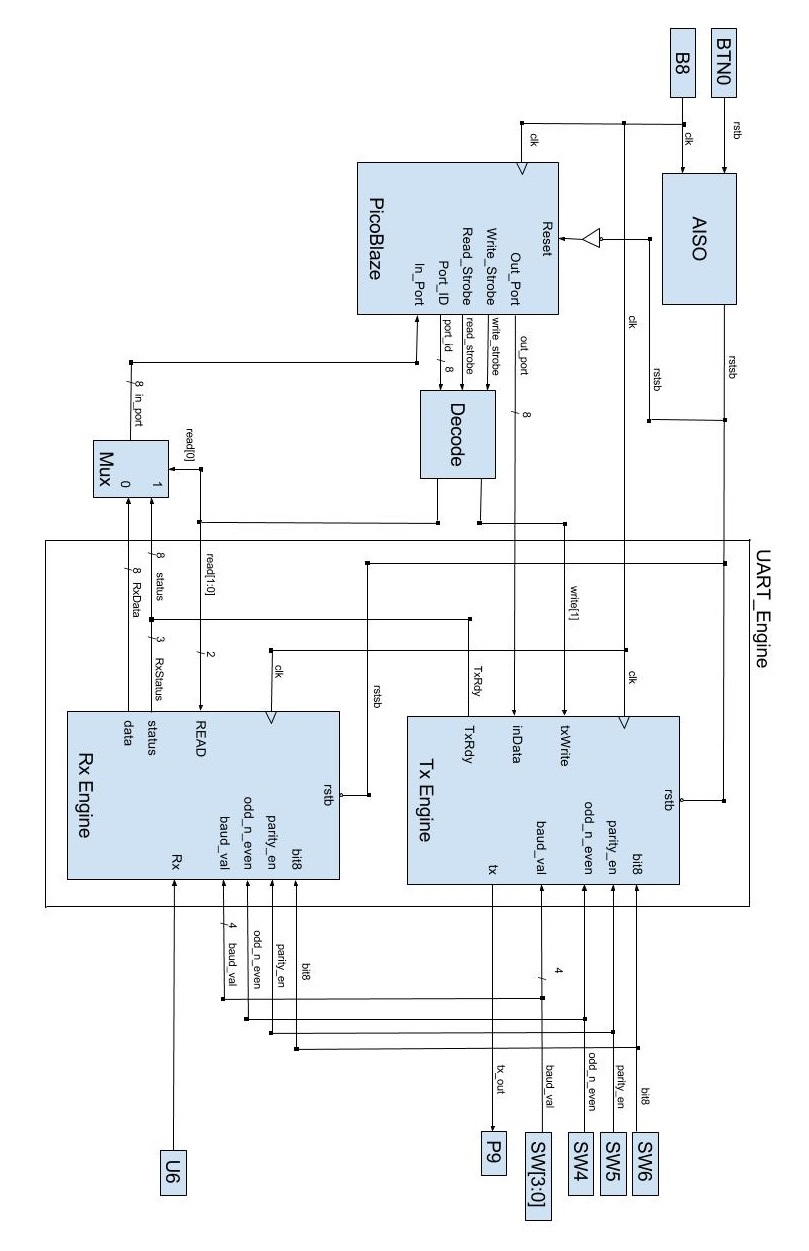
The fourth block consists of the transmit engine, where I format the data bits in the correct sequence and determine the parity, number of bits, and baud rate at which to transmit the data. I then shift load this data into a shift register and shift out the data bits individually until all of the bits have been transmitted to the terminal. The transmit engine also outputs a TxRdy flag to let the PicoBlaze know when it is ready to transmit another byte.

The fifth block consists of the receive engine. This block receives data from the terminal by determining the parity, number of bits, and baud rate at which the data is being received. The data bits are shifted into a shift register. Once all data bits have been shifted in, the receive engine then loads the received byte into a register and calculates the parity error. The receive engine also continually updates the overflow flag to make sure that an overflow does not occur (when the receive engine starts receiving another byte before it finishes receiving the previous byte). The receive engine outputs both the received byte of data and its status flags (Overflow, Parity Error, and RxRdy). The received byte of data is then sent over to the PicoBlaze so it can process it and figure out what to transmit to the terminal.

Both the receive engine status flags and the transmit engine status flags are concatenated together and they then go into the sixth and final block, which is a multiplexer. The multiplexer decides what 8 bits of data to send to the in\_port of the PicoBlaze based on whether read[0] is set high. If it is, then the in\_port of the PicoBlaze gets the status flags {2'b0, Overflow, ParityError, 2'b0, TxRdy, RxRdy}, otherwise it gets the received byte data from the receive engine. That sums up the top-level design of my project.

## 4.1.2 Block Diagram

-On Next Page:



## 4.1.3 I/O

* (I) bit8 (1: 8 bits, 0: 7 bits)

Pin Assignment: N17 (SW 6)

* (I) parity\_en (1: parity enabled, 0: no parity)

Pin Assignment: L13 (SW5)

* (I) odd\_n\_even: (1: odd parity, 0: even parity)

Pin Assignment: L14 (SW 4)

* (I) baud\_val: (4 bits wide / combinations determine the baud rate at which the UART engine will run at)

Pin Assignment: K17 (SW 3), K18 (SW 2), H18 (SW 1), G18 (SW 0),

* (I) txWrite: High active write enable
* (O) Txrdy: High active signal telling the PicoBlaze that it may write another byte
* (I) inData[7:0]: data from processor to be transmitted
* (O) tx: Data bit being shifted out from Transmit Engine.
* (I) READ[1:0]: High active READ enable
* (O) status[2:0]: Status bits of the Receive Engine (OVERFLOW, PARITY\_ERROR, RxRdy)
* (O) data[7:0]: data received from Receive Engine that should be passed to the PicoBlaze so that it can process the data.
* (I) Rx: Data bit being shifted into the Receive Engine.

## 4.1.4 Clocks

All blocks in this design use one clock that runs at a frequency of 50MHz. This means that the clock has a period of 20 nanoseconds (1/50MHz).

## 4.1.5 Resets

All original intellectual property (IP) developed for this design use an asynchronous, LOW active reset. The external PicoBlaze microcontroller in our design, on the other hand, uses an asynchronous HIGH active reset (meaning that we must feed an inverted reset value to the PicoBlaze).

## 4.1.6 Software

For the sake of readability, the software for this design is included at the end of the report.

# 5. Externally Developed Blocks:

## 5.1.1 Description

\*NOTE: The following description, block diagram, I/O, and Register Maps were taken directly from the PicoBlaze Users Guide, which can be accessed [here](http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf).

The PicoBlaze™ microcontroller is a compact, capable, and cost-effective fully embedded 8-bit RISC microcontroller core optimized for the Xilinx FPGA families. The KCPSM3 version described in this user guide occupies just 96 FPGA slices in a Spartan®-3 Generation FPGA which is only 12.5% of an XC3S50 device and a miniscule 0.3% of an XC3S5000 device. In typical implementations, a single FPGA block RAM stores up to 1024 program instructions, which are automatically loaded during FPGA configuration. Even with such resource efficiency, the PicoBlaze microcontroller performs a respectable 44 to 100 million instructions per second (MIPS) depending on the target FPGA family and speed grade.

The KCPSM6 version of the PicoBlaze microcontroller is optimized for the Spartan-6, Virtex®-6, and 7 Series FPGAs and exploits the progress in technology to provide an even

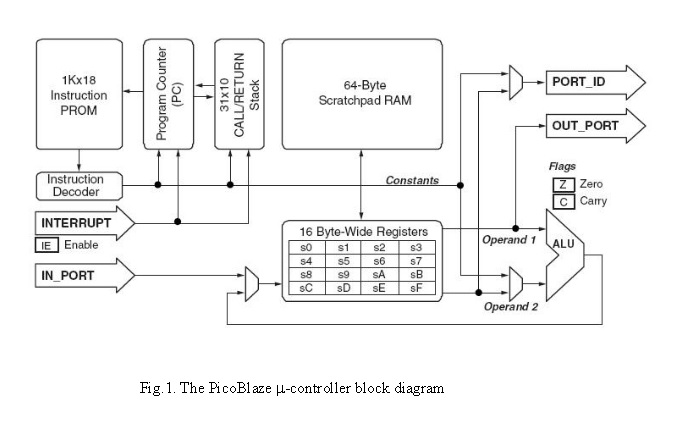
higher degree of efficiency. The KCPSM6 microcontroller is a superset of KCPSM3 and is incredibly small occupying only 26 Slices.

The PicoBlaze microcontroller core is totally embedded within the target FPGA and requires no external resources. The PicoBlaze microcontroller is extremely flexible. The basic functionality is easily extended and enhanced by connecting additional FPGA logic to the microcontroller ’s input and output ports.

The PicoBlaze microcontroller provides abundant, flexible I/O at much lower cost than off-the-shelf controllers. Similarly, the PicoBlaze peripheral set can be customized to meet the specific features, function, and cost requirements of the target application. Because the PicoBlaze microcontroller is delivered as synthesizable VHDL source code, the core is future-proof and can be migrated to future FPGA architectures, effectively eliminating product obsolescence fears. Being integrated within the FPGA, the PicoBlaze microcontroller reduces board space, design cost, and inventory.The PicoBlaze FPC is supported by a suite of development tools including an assembler, a graphical integrated development environment (IDE), a graphical instruction set simulator, and VHDL source code and simulation models. Similarly, the PicoBlaze microcontroller is also supported in the Xilinx System Generator development environment.

## 5.1.2 Block Diagram

-On Next Page:



## 5.1.3 I/O

The Input/Output ports extend the PicoBlaze microcontroller’s capabilities and allow the microcontroller to connect to a custom peripheral set or to other FPGA logic. The PicoBlaze microcontroller supports up to 256 input ports and 256 output ports or a combination of input/output ports. The PORT\_ID output provides the port address. During an INPUT operation, the PicoBlaze microcontroller reads data from the IN\_PORT port to a specified register, sX. During an OUTPUT operation, the PicoBlaze microcontroller writes the contents of a specified register, sX, to the OUT\_PORT port.

Interface Signals:

* IN\_PORT[7:0] Input - Input Data Port: Present valid input data on this port during an INPUT instruction. The data is captured on the rising edge of CLK.
* INTERRUPT Input - Interrupt Input: If the INTERRUPT\_ENABLE flag is set by the application code, generate an INTERRUPT Event by asserting this input High for at least two CLK cycles. If the INTERRUPT\_ENABLE flag is cleared, this input is ignored.
* RESET Input - Reset Input: To reset the PicoBlaze microcontroller and to generate a RESET Event, assert this input High for at least one CLK cycle. A Reset Event is automatically generated immediately following FPGA configuration.
* CLK Input - Clock Input: The frequency may range from DC to the maximum operating frequency reported by the Xilinx ISE development software. All PicoBlaze synchronous elements are clocked from the rising clock edge. There are no clock duty-cycle requirements beyond the minimum pulse width requirements of the FPGA.
* OUT\_PORT[7:0] Output - Output Data Port: Output data appears on this port for two CLK cycles during an OUTPUT instruction. Capture output data within the FPGA at the rising CLK edge when WRITE\_STROBE is High.
* PORT\_ID[7:0] Output - Port Address: The I/O port address appears on this port for two CLK cycles during an INPUT or OUTPUT instruction.
* READ\_STROBE Output - Read Strobe: When asserted High, this signal indicates that input data on the IN\_PORT[7:0] port was captured to the specified data register during an INPUT instruction. This signal is asserted on the second CLK cycle of the two-cycle INPUT instruction. This signal is typically used to acknowledge read operations from FIFOs.
* WRITE\_STROBE Output - Write Strobe: When asserted High, this signal validates the output data on the OUT\_PORT[7:0] port during an OUTPUT instruction. This signal is asserted on the second CLK cycle of the two-cycle OUTPUT instruction. Capture output data within the FPGA on the rising CLK edge when WRITE\_STROBE is High.
* INTERRUPT\_ACK Output - Interrupt Acknowledge: When asserted High, this signal acknowledges that an INTERRUPT Event occurred. This signal is asserted during the second CLK cycle of the two-cycle INTERRUPT Event. This signal is optionally used to clear the source of the INTERRUPT input.

## 5.1.4 Register Map

The PicoBlaze microcontroller includes 16 byte-wide general-purpose registers, designated as registers s0 through sF. For better program clarity, registers can be renamed using an assembler directive. All register operations are completely interchangeable; no registers are reserved for special tasks or have priority over any other register. There is no dedicated accumulator; each result is computed in a specified register.

The PicoBlaze microcontroller provides an internal general-purpose 64-byte scratchpad

RAM, directly or indirectly addressable from the register file using the STORE and FETCH instructions. The STORE instruction writes the contents of any of the 16 registers to any of the 64 RAM locations. The complementary FETCH instruction reads any of the 64 memory locations into any of the 16 registers. This allows a much greater number of variables to be held within the boundary of the processor and tends to reserve all of the I/O space for real inputs and output signals. The six-bit scratchpad RAM address is specified either directly (ss) with an immediate constant, or indirectly using the contents of any of the 16 registers (sY). Only the lower six bits of the address are used; the address should not exceed the 00 - 3F range of the available memory.

# 6. Internally Developed Blocks:

## 6.1.1 UART Engine Description

The SOC is using a Universal Asynchronous Receiver Transmitter (UART) interface to receive and transmit data. The UART Engine consists of both a Transmit Engine and a Receive Engine. The Transmit Engine is in charge of transmitting characters to the terminal and the Receive Engine is in charge of receiving the data that is coming from the terminal. The user sets up an agreed upon connection with the SOC and then opens up a terminal window. The UART Engine is connected to the 8-bit PicoBlaze Microcontroller, which is used to process and control the data going to and coming from the Transmit Engine. Upon reset, the PicoBlaze initiates the transmission of the banner by setting the appropriate character data and waiting until the Transmit engine is ready to transmit the next byte. It does this by reading in the status flags and monitoring the TxRdy output from the Transmit Engine. Upon reset, I have it so that my Transmit Engine is automatically ready to transmit a byte of data. Once the banner is transmitted, the PicoBlaze then transmits a prompt (\*>) to the terminal and waits for the user to enter a character onto the terminal. Once a character is entered, the Receive Engine then shifts in the character data one byte at a time. Once the byte is successfully shifted in, I then send this data over to the PicoBlaze so it can process the character received. It then determines what action to take based on what character the PicoBlaze received. If the character is a carriage return or a line feed character, then the PicoBlaze will transmit a new line command and re-issue the prompt to the terminal window. If the character is a carriage return or a line feed character, then the PicoBlaze will have the UART Engine transmit a new line command and re-issue the prompt to the terminal window. If the character is an asterisk (\*), then the PicoBlaze will have the UART Engine transmit the hometown of the developer (Porterville, CA) to the terminal window. Any other valid ASCII character that the user enters on the keyboard will be echoed to the terminal. The PicoBlaze is also able to process the status of the UART engine and informs the user whenever an overflow error or a parity error occurs within the UART communication. The status flags are cleared every time that the PicoBlaze reads the status of the UART engine.

I also added my Seven-Segment display to this lab so that I can display my character counter value on the upper 2 Seven-Segment displays and the data being received by the Receive Engine in the lower 2 Seven-Segment displays. This helped me out tremendously in verifying that I was receiving the proper character data.

## 6.1.2 UART Transmit Description

The Transmit Engine module contains all of the logic needed to transmit data via universal asynchronous receiver/transmitter (UART) communication. In order to create a stable UART connection, I needed to make sure that I could run my program at different baud rates and different data formats (7N1, 7O1, 7E1, 8N1, 8O1, 8E1). I also needed to make sure that both my program and the terminal software were running at the same baud-rate and data format.

In the Transmit Engine, I am shifting out the data in 12-bit data packets. I use a shift register to determine what data gets shifted out. I am able to load in the data that is going to be shifted out by delaying the write[1] strobe from the PicoBlaze by 1 clock cycle (this is because the data on the out\_port of the PicoBlaze is not available until the next clock cycle). If my write1Delay variable is set high, I then load my shift register with the out\_port data of the PicoBlaze, which I stored using a flop within my Transmit Engine. I then update the remaining values in the shift register to make sure that I have accounted for the right parity, stop, and start bits. This is done using the bit8, parity\_en, and odd\_n\_even inputs and decoding their values so that I know what values to put in the b8, b7 bits of the shift register. Once all of the data in the shift register is accurate, I then start shifting out the bits 1 bit at a time. I fill the most significant bit of my shift register with a 1 (which is equivalent to filling up the register with a bunch of stop bits).

I keep track of when I am done shifting out all of the bits in my shift register by using a bit counter. In order to know when a bit is done being shifted out, I also needed to use a baud-rate counter. I first decoded the baud rate by reading in the data from my four baud\_rate switches. The next step was to determine what baud-rate I was running at based on those four switches. After determining the baud rate, I then needed to determine the amount of ticks I would need to count in order to achieve the specified baud-rate. I did this by using the formula baudCountNum = (1/baud rate) / (1/50MHz). This gave me the terminal count number that I needed to reach in order to run at the desired baud rate. I then assigned my baudCountNum variable to this terminal count value. Once that value was reached, I then knew that I had finished transmitting one bit via UART communication. Now the next step was to increment my Bit Counter. I repeated this process until my Bit Counter reached 11, meaning that I had finished transmitting all 12 bits of data in the data packet. For this project, I am always going to end up transmitting 12 bits of data per data packet. That is why my bit time counter limit is always 11 (0 to (12-1)). Once all 12 bits of data have been shifted out, I then set my Done variable to 1, which also sets my TxRdy variable to a 1. I then output this TxRdy variable, along with the other status variables to the PicoBlaze. These status variables are concatenated and represent the status register of the Transmit Engine. The TxRdy bit lets the PicoBlaze know that the Transmit Engine is ready to transmit another byte, at which point the PicoBlaze will output the next character being transmitted in the sequence to the Transmit Engine.

## 6.1.3 UART Receive Description

The Receive Engine module contains all of the logic needed to receive data via universal asynchronous receiver/transmitter (UART) communication. In order to create a stable UART connection, I needed to make sure that I could run my program at different baud rates and different data formats (7N1, 7O1, 7E1, 8N1, 8O1, 8E1). I also needed to make sure that both my program and the terminal software were running at the same baud-rate and data format.

My receive engine module contains all of the logic needed to receive information from a computer via universal asynchronous receiver/transmitter (UART) communication. In order to create a stable UART connection, I needed to make sure that I could run my program at different baud rates and different data formats (7N1, 7O1, 7E1, 8N1, 8O1, 8E1). For this program, I am shifting in the data from the receive engine 1-bit at a time. I use a shift register to store the receive engine data bits. Each data bit gets shifted in one bit at a time. For the receive engine we have a state diagram that consists of four different states: IDLE, S1, S2, and S3. The state machine starts off in the IDLE state at reset. In this state, it waits for a start bit (which is agreed to be a zero). We then move to the S1 state, where the state machine waits for half a bit-time. If the Rx input changes to a 1 during this half bit-time interval, I then move back to the IDLE state where we reset the bit-time interval and wait to receive a start bit value (0). If the Rx input stays the same (0) for the entirety of the half-bit time, this means that we actually received a start bit and I then move on to the next state, S2. I now know that we need to start collecting the data bits, so in S2, I wait for an entire bit time. After the bit time is achieved, I am know in the middle of the first data bit. I then move on to state S3, where I shift the data bit value into my shift register. I then check to see if all of the data bits have been shifted in to my shift register by checking the status of my bcu variable. This variable is set high for one clock cycle whenever all of the data bits have been received. If the bcu variable is set high, then I go back to the idle state where I wait until another bit is received. If the variable is not set high, then I go back to S2 and wait for another bit time so that I can shift in the next bit into the receive engine. Once all of the data bits have been shifted into the receive engine, I then send these data bits to the PicoBlaze so that it can process the received character and determine what to do with that data. I also send the status bits of the Rx Engine (OVERFLOW, PARITY\_ERR, and RxRdy) so that the PicoBlaze can inform the user if there were any errors in receiving the data. The OVERFLOW and PARITY\_ERR flags are cleared every time they are read from the Receive Engine.

The core buad rate and bit time counter logic are identical to that described in the transmit engine. There are a few minor differences, however. For one, I am using the DOIT variable generated in my state machine instead of the start bit generated in the Transmit Engine. In my baud rate counter logic, I also added in a multiplexer where I decide what value to load into the finalBaudCountNum variable. If my START variable is set high, then I know that I am waiting for the start bit of the received data. I then shift my baudCountNum to the right by 1. This divides the value by 2, which means that I am waiting for half of a bit time instead of the whole bit time. If the START variable is set low, then I know that I am waiting for the whole bit time and I keep my baudCountNum the same. Finally, I had to delay the done variable by one clock cycle in my design. Before I did this, my done bit was assigned when bcu was set high. This means that the done bit was automatically set high after the final bit time was achieved. This however, did not mean that the Receive Engine was finished yet because it still needed to load the final bit into the shift register. As a result, I was receiving overflow errors when trying to receive bytes in succession of each other because RxRdy was set high before the Rx Engine was finished receiving the data. In order to fix this, I delayed the done bit by putting it through a flop. This delayed the done bit for one clock cycle (which was enough time for the receive engine to load in the final data bit to the shift register). This fixed my overflow problem and allowed me to ensure that I was actually finished receiving the whole byte before I asserted the RxRdy bit.

The RxRdy bit is set high whenever it is ready to receive another byte. While the Receive Engine is receiving a byte, the RxRdy bit will be set low for the duration of the receive. When the Receive Engine is done receiving the byte, it then sends it to the PicoBlaze and then asserts the RxRdy flag so that it can notify the PicoBlaze that it is ready to receive another byte of data.

## 6.1.4 PicoBlaze Assembly

In order to get the PicoBlaze to successfully receive and transmit data via my UART Engine, I had to create my own assembly code where I process the received data and send the appropriate data to the Transmit Engine after the TxRdy bit from the status register was set high. Here is the assembly code that I wrote for this project:

;UART Full

; ;================================================================

; data constants

;================================================================

;selected ASCII codes

CONSTANT ASCII\_CR , 0D ; carriage return <CR>

CONSTANT ASCII\_LF , 0A ; line feed <LF>

CONSTANT ASCII\_Space , 20 ; Space

CONSTANT ASCII\_Asterisk , 2A ; \*(Asterisk) character

CONSTANT ASCII\_Greater\_Than, 3E ; >(Greater Than) character

CONSTANT ASCII\_bslash , 2F ; /(Backslash) character

CONSTANT ASCII\_colon , 3A ; :(colon) character

CONSTANT ASCII\_comma , 2C ; ,(comma) character

CONSTANT ASCII\_dash , 2D ; -(dash) character

CONSTANT ASCII\_exclamation , 21 ; !(exclamation point) character

CONSTANT ASCII\_backspace , 08 ; BS(backspace) character

CONSTANT ASCII\_delete , 7F ; DEL(descructive delete) character

CONSTANT ASCII\_C\_U , 43 ; Uppercase C

CONSTANT ASCII\_E\_U , 45 ; Uppercase E

CONSTANT ASCII\_S\_U , 53 ; Uppercase S

CONSTANT ASCII\_4 , 34 ; number 4

CONSTANT ASCII\_6 , 36 ; number 6

CONSTANT ASCII\_0 , 30 ; number 0

CONSTANT ASCII\_V\_U , 56 ; Uppercase V

CONSTANT ASCII\_i , 69 ; Lowercase i

CONSTANT ASCII\_c , 63 ; Lowercase c

CONSTANT ASCII\_t , 74 ; Lowercase t

CONSTANT ASCII\_o , 6F ; Lowercase o

CONSTANT ASCII\_r , 72 ; Lowercase r

CONSTANT ASCII\_s , 73 ; Lowercase s

CONSTANT ASCII\_p , 70 ; Lowercase p

CONSTANT ASCII\_n , 6E ; Lowercase n

CONSTANT ASCII\_z , 7A ; Lowercase z

CONSTANT ASCII\_a , 61 ; Lowercase a

CONSTANT ASCII\_F\_U , 46 ; Uppercase F

CONSTANT ASCII\_u , 75 ; Lowercase u

CONSTANT ASCII\_l , 6C ; Lowercase l

CONSTANT ASCII\_U\_U , 55 ; Uppercase U

CONSTANT ASCII\_A\_U , 41 ; Uppercase A

CONSTANT ASCII\_R\_U , 52 ; Uppercase R

CONSTANT ASCII\_T\_U , 54 ; Uppercase T

CONSTANT ASCII\_D\_U , 44 ; Uppercase D

CONSTANT ASCII\_e , 65 ; Lowercase e

CONSTANT ASCII\_1 , 31 ; number 1

CONSTANT ASCII\_9 , 39 ; number 9

CONSTANT ASCII\_5 , 35 ; number 5

CONSTANT ASCII\_H\_U , 48 ; Uppercase H

CONSTANT ASCII\_m , 6D ; Lowercase m

CONSTANT ASCII\_w , 77 ; Lowercase w

CONSTANT ASCII\_P\_U , 50 ; Uppercase P

CONSTANT ASCII\_v , 76 ; Lowercase v

CONSTANT ASCII\_I\_U , 49 ; Uppercase I

CONSTANT ASCII\_Y\_U , 59 ; Uppercase Y

CONSTANT ASCII\_O\_U , 4F ; Uppercase O

CONSTANT ASCII\_L\_U , 4C ; Uppercase L

CONSTANT ASCII\_W\_U , 57 ; Uppercase W

;================================================================

; port aliases

;================================================================

;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_input port definitions\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

CONSTANT rd\_flag\_port, 00 ;status of transmit engine

CONSTANT rx\_data\_port, 01 ;received data from receive engine

NAMEREG s0, tx\_data ;data to be transmitted by uart

NAMEREG s2, rx\_data ;data to be received by uart

NAMEREG s3, char\_counter ;keeps track of how many characters have been

;transmitted on the current line (for destructive

;delete)

;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_output port definitions\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

CONSTANT uart\_tx\_port, 01 ;outputs to register 1 (Write\_Strobe[1])

;================================================================

; Main Program

;================================================================

main\_program:

load char\_counter, 00 ;zero out the character counter

call display\_banner ;Display the banner at the beginning of the program

infinite\_loop:

call proc\_uart ;receive uart characters

JUMP infinite\_loop

;================================================================

; routine : check\_status\_flags

; function : check the status flags of the UART engine and inform the user of

; any errors that have occurred (parity, overflow)

; Input Register : s1 - read port flags

; Temp Register : s5 - check for parity/overflow error

; Output Register : tx\_data (s0) - data to be transmitted by uart

;================================================================

check\_status\_flags:

load s5, s1 ;copy status register value into s5

and s5, 20 ;isolate overflow bit

sub s5, 20 ;check to see if Overflow bit is set high

jump nz, check\_parity\_errror ;if it isn't, check the parity error flag

call display\_overflow\_error ;display overflow error

jump done\_error\_checking ;jump to the end of the error checking

check\_parity\_errror :

load s5, s1 ;copy status register value into s5

and s5, 10 ;isolate parity error bit

sub s5, 10 ;check to see if Parity Error bit is set high

jump nz, done\_error\_checking ;if it isn't, jump to the end of the error checking

call display\_pairty\_error ;display overflow error

done\_error\_checking:

return

;================================================================

; routine : tx\_one\_byte

; function : Wait until uart TxRdy bit is set, which signifies that the UART is

; ready to transmit another byte. Then transmit another byte to

; the UART.

; Input Register : s1 - read port flags

; Output Register : tx\_data (s0) - data to be transmitted by uart

;================================================================

tx\_one\_byte :

input s1, rd\_flag\_port ;read in status of the Transmit Engine

call check\_status\_flags ;check for overflow/parity errors

and s1, 02 ;isolate TxRdy bit

sub s1, 02 ; check to see if TxRdy bit is set high

jump nz, tx\_one\_byte ;if it isn't, keep on waiting until it is set high

output tx\_data, uart\_tx\_port ;If it is set high, then transmit byte to uart

compare tx\_data, ASCII\_delete ;check to see if data is a destructive delete

jump nz, increment\_counter

sub char\_counter, 01 ;decrement character counter

jump done\_updating\_counter

increment\_counter:

add char\_counter, 01 ;increment character counter

done\_updating\_counter:

output char\_counter, 05 ;output character counter value

return

;================================================================

; routine: proc\_uart

; function : receive UART input character and process it:

; CR - transmit a new line (CR/LF) and prompt

; LF - transmit a new line (CR/LF) and prompt

; \* - Display the hometown of user

; other - echo character

; Input Register : rx\_data(s2) - received data,

; s1 - read port flags,

; Output Register : tx\_data (s0) - data to be transmitted by uart

;================================================================

proc\_uart:

input rx\_data , rx\_data\_port ;receive data from terminal

receive\_byte:

input s1, rd\_flag\_port ;read in status of the Receive Engine

call check\_status\_flags ;check for overflow/parity errors

and s1, 01 ;isolate RxRdy bit

sub s1, 01 ; check to see if RxRdy bit is set high

jump nz, receive\_byte ;if it isn't, keep on waiting until it is set high

input rx\_data , 03 ;store received data in register

compare rx\_data, ASCII\_CR ;see if carriage return was received

jump nz, compare\_lf ;if not, check for line feed

call new\_prompt ;trigger the newline prompt

jump uart\_receive\_byte\_done

compare\_lf:

compare rx\_data, ASCII\_LF ;see if line feed was received

jump nz, compare\_asterisk ;if not, check for asterisk

call new\_prompt ;trigger the newline prompt

jump uart\_receive\_byte\_done

compare\_asterisk:

compare rx\_data, ASCII\_Asterisk ;see if asterisk was received

jump nz, compare\_backspace ;if not, check for backspace

call transmit\_hometown ;transmit hometown

jump uart\_receive\_byte\_done

compare\_backspace:

compare rx\_data, ASCII\_backspace ;see if backspace was received

jump nz, compare\_delete ;if not, check for destructive delete

compare char\_counter, 04 ;check to see if there are any bits to delete

jump z, uart\_receive\_byte\_done ;if there are no characters available to delete, do nothing

load rx\_data, ASCII\_delete ;load with a destructive delete if backspace pressed

jump transmit\_received\_char ;delete the character

compare\_delete:

compare rx\_data, ASCII\_delete ;see if delete was received

jump nz, echo\_character ;if not, echo character

compare char\_counter, 04 ;check to see if there are any bits to delete

jump z, uart\_receive\_byte\_done ;if there are no characters available to delete, do nothing

jump transmit\_received\_char ;transmit the destructive delete

echo\_character:

compare char\_counter, AA ;check to see if counter limit has been reached

jump nz, transmit\_received\_char ;if it isn't, transmit the character

call new\_prompt ;otherwise start a new line

transmit\_received\_char:

load tx\_data , rx\_data

call tx\_one\_byte ; echo received character

uart\_receive\_byte\_done :

return

;================================================================

; routine: display\_banner

; function : Transmits the beginning banner onto the terminal screen:

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; \* CECS 460

; \* Victor Espinoza

; \* Full UART

; \* Due: 11/19/15

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;================================================================

display\_banner:

call transmit\_top\_of\_banner ;transmit asterisks

call transmit\_class ;transmit class name

call transmit\_student\_name ;transmit my name

call transmit\_project\_name ;transmit project name

call transmit\_due\_date ;transmit due date

call transmit\_top\_of\_banner ;transmit asterisks

load tx\_data , ASCII\_CR

call tx\_one\_byte ; transmit CR (Carriage Return)

load tx\_data , ASCII\_LF

call tx\_one\_byte ; transmit LF (Line Feed)

call new\_prompt ;transmit prompt

return

;================================================================

; routine: new\_prompt

; function : Transmits the new prompt (\*>) whenever a CR or LF chacacter

; are received or when the terminal issues a new line command.

;================================================================

new\_prompt:

load char\_counter, 00 ;zero out the character counter

load tx\_data , ASCII\_CR

call tx\_one\_byte ; transmit CR (Carriage Return)

load tx\_data , ASCII\_LF

call tx\_one\_byte ; transmit LF (Line Feed)

load tx\_data , ASCII\_Asterisk

call tx\_one\_byte ; transmit \* (Asterisk)

load tx\_data , ASCII\_Greater\_Than

call tx\_one\_byte ; transmit >(Greater Than)

return

;================================================================

; routine: transmit\_top\_of\_banner

; function : Transmits the top part of the banner (30 asterisks)

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;================================================================

transmit\_top\_of\_banner:

load tx\_data , ASCII\_CR

call tx\_one\_byte ; transmit CR (Carriage Return)

load tx\_data , ASCII\_LF

call tx\_one\_byte ; transmit LF (Line Feed)

load sF, 1D ;

asterisk\_loop:

compare sF, 00 ; check to see if loop is done transmitting

jump z, done\_transmitting\_asterisks ; if it is, jump to end of loop

load tx\_data , ASCII\_Asterisk

call tx\_one\_byte ; transmit \* (Asterisk)

sub sF, 01 ; decrement loop counter

JUMP asterisk\_loop

done\_transmitting\_asterisks:

return

;================================================================

; routine: transmit\_class

; function : Transmits the class name (CECS 460)

;================================================================

transmit\_class:

load tx\_data , ASCII\_CR

call tx\_one\_byte ; transmit CR (Carriage Return)

load tx\_data , ASCII\_LF

call tx\_one\_byte ; transmit LF (Line Feed)

load tx\_data , ASCII\_Asterisk

call tx\_one\_byte ; transmit \* (Asterisk)

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_C\_U

call tx\_one\_byte ; transmit C

load tx\_data , ASCII\_E\_U

call tx\_one\_byte ; transmit E

load tx\_data , ASCII\_C\_U

call tx\_one\_byte ; transmit C

load tx\_data , ASCII\_S\_U

call tx\_one\_byte ; transmit S

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_4

call tx\_one\_byte ; transmit 4

load tx\_data , ASCII\_6

call tx\_one\_byte ; transmit 6

load tx\_data , ASCII\_0

call tx\_one\_byte ; transmit 0

return

;================================================================

; routine: transmit\_student\_name

; function : Transmits the student name (Victor Espinoza)

;================================================================

transmit\_student\_name:

load tx\_data , ASCII\_CR

call tx\_one\_byte ; transmit CR (Carriage Return)

load tx\_data , ASCII\_LF

call tx\_one\_byte ; transmit LF (Line Feed)

load tx\_data , ASCII\_Asterisk

call tx\_one\_byte ; transmit \* (Asterisk)

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_V\_U

call tx\_one\_byte ; transmit V

load tx\_data , ASCII\_i

call tx\_one\_byte ; transmit i

load tx\_data , ASCII\_c

call tx\_one\_byte ; transmit c

load tx\_data , ASCII\_t

call tx\_one\_byte ; transmit t

load tx\_data , ASCII\_o

call tx\_one\_byte ; transmit o

load tx\_data , ASCII\_r

call tx\_one\_byte ; transmit r

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_E\_U

call tx\_one\_byte ; transmit E

load tx\_data , ASCII\_s

call tx\_one\_byte ; transmit s

load tx\_data , ASCII\_p

call tx\_one\_byte ; transmit p

load tx\_data , ASCII\_i

call tx\_one\_byte ; transmit i

load tx\_data , ASCII\_n

call tx\_one\_byte ; transmit n

load tx\_data , ASCII\_o

call tx\_one\_byte ; transmit o

load tx\_data , ASCII\_z

call tx\_one\_byte ; transmit z

load tx\_data , ASCII\_a

call tx\_one\_byte ; transmit a

return

;================================================================

; routine: transmit\_project\_name

; function : Transmits the project name (Full UART)

;================================================================

transmit\_project\_name:

load tx\_data , ASCII\_CR

call tx\_one\_byte ; transmit CR (Carriage Return)

load tx\_data , ASCII\_LF

call tx\_one\_byte ; transmit LF (Line Feed)

load tx\_data , ASCII\_Asterisk

call tx\_one\_byte ; transmit \* (Asterisk)

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_F\_U

call tx\_one\_byte ; transmit F

load tx\_data , ASCII\_u

call tx\_one\_byte ; transmit u

load tx\_data , ASCII\_l

call tx\_one\_byte ; transmit l

load tx\_data , ASCII\_l

call tx\_one\_byte ; transmit l

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_U\_U

call tx\_one\_byte ; transmit U

load tx\_data , ASCII\_A\_U

call tx\_one\_byte ; transmit A

load tx\_data , ASCII\_R\_U

call tx\_one\_byte ; transmit R

load tx\_data , ASCII\_T\_U

call tx\_one\_byte ; transmit T

return

;================================================================

; routine: transmit\_due\_date

; function : Transmits the project due date (Due: 11/19/15)

;================================================================

transmit\_due\_date:

load tx\_data , ASCII\_CR

call tx\_one\_byte ; transmit CR (Carriage Return)

load tx\_data , ASCII\_LF

call tx\_one\_byte ; transmit LF (Line Feed)

load tx\_data , ASCII\_Asterisk

call tx\_one\_byte ; transmit \* (Asterisk)

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_D\_U

call tx\_one\_byte ; transmit D

load tx\_data , ASCII\_u

call tx\_one\_byte ; transmit u

load tx\_data , ASCII\_e

call tx\_one\_byte ; transmit e

load tx\_data , ASCII\_colon

call tx\_one\_byte ; transmit : (Colon)

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_1

call tx\_one\_byte ; transmit 1

load tx\_data , ASCII\_1

call tx\_one\_byte ; transmit 1

load tx\_data , ASCII\_bslash

call tx\_one\_byte ; transmit / (Backslash)

load tx\_data , ASCII\_l

call tx\_one\_byte ; transmit l

load tx\_data , ASCII\_9

call tx\_one\_byte ; transmit 9

load tx\_data , ASCII\_bslash

call tx\_one\_byte ; transmit / (Backslash)

load tx\_data , ASCII\_l

call tx\_one\_byte ; transmit l

load tx\_data , ASCII\_5

call tx\_one\_byte ; transmit 5

return

;================================================================

; routine: transmit\_hometown

; function : Transmits the student's hometown (Hometown - Porterville, CA)

;================================================================

transmit\_hometown:

call new\_prompt ; transmit prompt

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_H\_U

call tx\_one\_byte ; transmit H

load tx\_data , ASCII\_o

call tx\_one\_byte ; transmit o

load tx\_data , ASCII\_m

call tx\_one\_byte ; transmit m

load tx\_data , ASCII\_e

call tx\_one\_byte ; transmit e

load tx\_data , ASCII\_t

call tx\_one\_byte ; transmit t

load tx\_data , ASCII\_o

call tx\_one\_byte ; transmit o

load tx\_data , ASCII\_w

call tx\_one\_byte ; transmit w

load tx\_data , ASCII\_n

call tx\_one\_byte ; transmit n

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_dash

call tx\_one\_byte ; transmit dash

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_P\_U

call tx\_one\_byte ; transmit P

load tx\_data , ASCII\_o

call tx\_one\_byte ; transmit o

load tx\_data , ASCII\_r

call tx\_one\_byte ; transmit r

load tx\_data , ASCII\_t

call tx\_one\_byte ; transmit t

load tx\_data , ASCII\_e

call tx\_one\_byte ; transmit e

load tx\_data , ASCII\_r

call tx\_one\_byte ; transmit r

load tx\_data , ASCII\_v

call tx\_one\_byte ; transmit v

load tx\_data , ASCII\_i

call tx\_one\_byte ; transmit i

load tx\_data , ASCII\_l

call tx\_one\_byte ; transmit l

load tx\_data , ASCII\_l

call tx\_one\_byte ; transmit l

load tx\_data , ASCII\_e

call tx\_one\_byte ; transmit e

load tx\_data , ASCII\_comma

call tx\_one\_byte ; transmit comma

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_C\_U

call tx\_one\_byte ; transmit C

load tx\_data , ASCII\_A\_U

call tx\_one\_byte ; transmit A

return

;================================================================

; routine: display\_pairty\_error

; function : Informs user of parity error (PARITY ERROR!)

;================================================================

display\_pairty\_error:

load tx\_data , ASCII\_P\_U

call tx\_one\_byte ; transmit P

load tx\_data , ASCII\_A\_U

call tx\_one\_byte ; transmit A

load tx\_data , ASCII\_R\_U

call tx\_one\_byte ; transmit R

load tx\_data , ASCII\_I\_U

call tx\_one\_byte ; transmit I

load tx\_data , ASCII\_T\_U

call tx\_one\_byte ; transmit T

load tx\_data , ASCII\_Y\_U

call tx\_one\_byte ; transmit Y

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_E\_U

call tx\_one\_byte ; transmit E

load tx\_data , ASCII\_R\_U

call tx\_one\_byte ; transmit R

load tx\_data , ASCII\_R\_U

call tx\_one\_byte ; transmit R

load tx\_data , ASCII\_O\_U

call tx\_one\_byte ; transmit O

load tx\_data , ASCII\_R\_U

call tx\_one\_byte ; transmit R

load tx\_data , ASCII\_exclamation

call tx\_one\_byte ; transmit exclamation

call new\_prompt ; transmit prompt

return

;================================================================

; routine: display\_overflow\_error

; function : Informs user of overflow error (OVERFLOW ERROR!)

;================================================================

display\_overflow\_error:

load tx\_data , ASCII\_O\_U

call tx\_one\_byte ; transmit O

load tx\_data , ASCII\_V\_U

call tx\_one\_byte ; transmit V

load tx\_data , ASCII\_E\_U

call tx\_one\_byte ; transmit E

load tx\_data , ASCII\_R\_U

call tx\_one\_byte ; transmit R

load tx\_data , ASCII\_F\_U

call tx\_one\_byte ; transmit F

load tx\_data , ASCII\_L\_U

call tx\_one\_byte ; transmit L

load tx\_data , ASCII\_O\_U

call tx\_one\_byte ; transmit O

load tx\_data , ASCII\_W\_U

call tx\_one\_byte ; transmit W

load tx\_data , ASCII\_Space

call tx\_one\_byte ; transmit space

load tx\_data , ASCII\_E\_U

call tx\_one\_byte ; transmit E

load tx\_data , ASCII\_R\_U

call tx\_one\_byte ; transmit R

load tx\_data , ASCII\_R\_U

call tx\_one\_byte ; transmit R

load tx\_data , ASCII\_O\_U

call tx\_one\_byte ; transmit O

load tx\_data , ASCII\_R\_U

call tx\_one\_byte ; transmit R

load tx\_data , ASCII\_exclamation

call tx\_one\_byte ; transmit exclamation

call new\_prompt ; transmit prompt

return

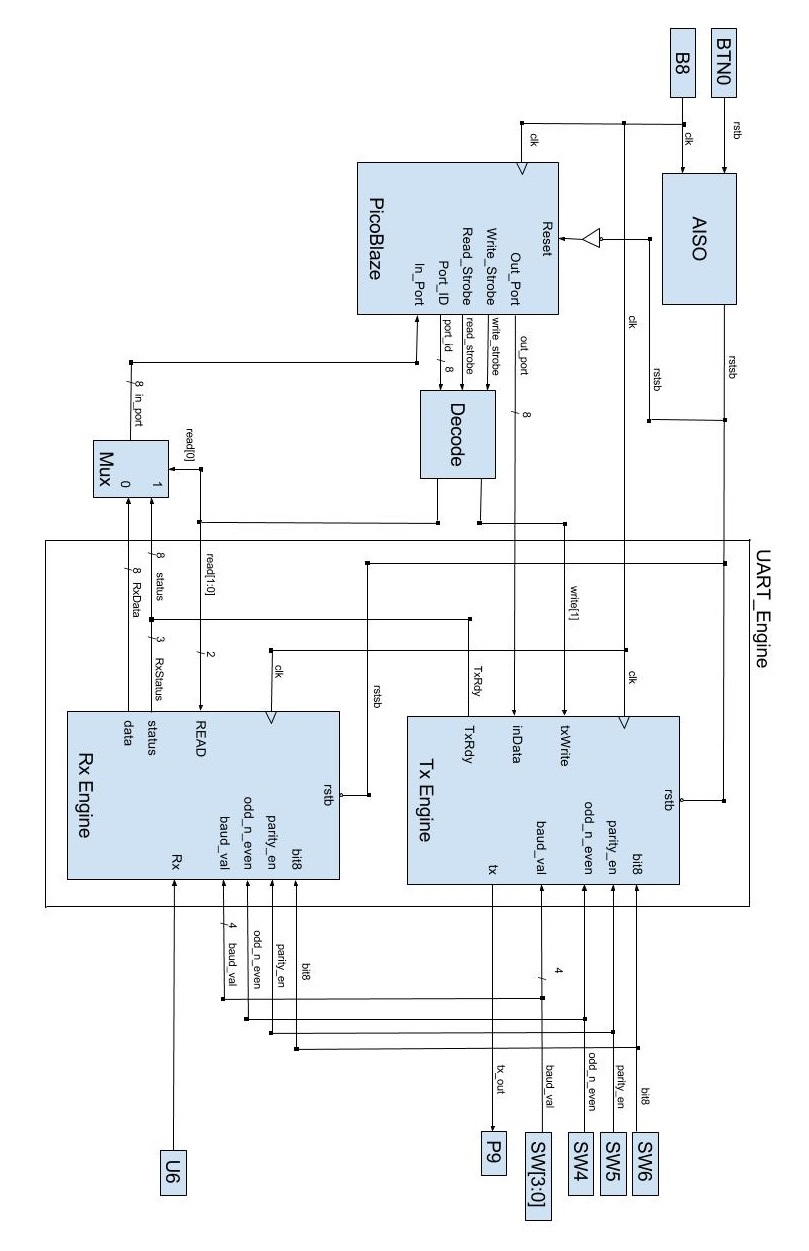
## 6.1.5 PicoBlaze Assembly Description

In my assembly code, I first define all of the constant values for the different ascii characters that I am going to be transmitting to the terminal. I then transmit my banner which displays the project name, my name, and the due date for the project. I output all of these characters to the PicoBlaze's outport by repeatedly calling the tx\_one\_byte task. The tx\_one\_byte task is a fairly simple task that outputs the appropriate register value for the ASCII character to the PicoBlaze's output. Before I output the value, however, I have to make sure that the TxRdy bit is set high within the Transmit Engine, which signifies that the engine is ready to transmit the next byte. I achieve this by sending the status register of the Transmit Engine (which has the TxRdy bit as the second bit in the register) to the in\_port of the PicoBlaze. I use the input s1, rd\_flag\_port instruction to store the value of the status register into the internal PicoBlaze register s1. I then call another task called check\_status\_flags which checks the OVERFLOW and PARITY\_ERR flags set in the Receive Engine and transmits the appropriate message to the terminal if these flags are set high (OVERFLOW ERROR and PARITY ERROR, respectively). After returning from the check\_status\_flags routine, I then subtract 02 from the s1 value in order to check to see if the TxRdy bit is set high. If the TxRdy bit is not set high, that means that the Transmit Engine is not ready to transmit the current data yet, so I keep on looping back to the beginning of my tx\_one\_byte task, where I continually read in the status register values and wait until the TxRdy bit is set high. When TxRdy is high, I then output the appropriate character value to the PicoBlaze's out\_port using the Output instruction. I output the character value to the port\_id 01 because I want to use the write[1] write\_strobe for my transmitting signal.

Once I am done printing the banner, I then transmit my prompt(\*>) and wait for the user to send a character to the Receive Engine. I monitor this by constantly reading in the status flags and seeing when RxRdy is set high (which indicates that the Receive Engine has finished receiving a byte of data and is ready to receive another byte. I then compare this received data to my ASCII line feed, carriage return, asterisk, and backspace / delete characters. If the received character happens to be one of these characters, then I perform the associated action that corresponds to that character (go to the next line and re-display the prompt, display my home town, and delete the previously transmitted character (if one is available). Otherwise I just echo the received character to the terminal window. I have a counter value that I increment every time a new character is transmitted onto the terminal screen and I decrement this counter whenever I delete a character. I have it so that the terminal window can store up to 170 (0xAA) characters on a line. If the user tries to enter a character that surpasses this limit, then I then issue a new line and prompt and then transmit the character as the first character that shows up after the new prompt on the next line. That is the gist of how my Assembly code works. For a more detailed discussion refer to the comments in the actual assembly code itself.

## 6.1.6 Block Diagram

-On Next Page



## 6.1.7 I/O

* (I) bit8 (1: 8 bits, 0: 7 bits)

Pin Assignment: N17 (SW 6)

* (I) parity\_en (1: parity enabled, 0: no parity)

Pin Assignment: L13 (SW5)

* (I) odd\_n\_even: (1: odd parity, 0: even parity)

Pin Assignment: L14 (SW 4)

* (I) baud\_val: (4 bits wide / combinations determine the baud rate at which the UART engine will run at)

Pin Assignment: K17 (SW 3), K18 (SW 2), H18 (SW 1), G18 (SW 0),

* (I) txWrite: High active write enable
* (O) Txrdy: High active signal telling the PicoBlaze that it may write another byte
* (I) inData[7:0]: data from processor to be transmitted
* (O) tx: Data bit being shifted out from Transmit Engine.
* (I) READ[1:0]: High active READ enable
* (O) status[2:0]: Status bits of the Receive Engine (OVERFLOW, PARITY\_ERROR, RxRdy)
* (O) data[7:0]: data received from Receive Engine that should be passed to the PicoBlaze so that it can process the data.
* (I) Rx: Data bit being shifted into the Receive Engine.
* (I) clk: Clock signal that drives the system
* (I) rstb: Reset signal that is disbursed to the entire system.

## 6.1.8 Register Map

PicoBlaze Registers used:

s0 - (tx\_data) : holds the 8-bit data to be transmitted by the UART Engine

s2 - (rx\_data) : holds the 8-bit data to be received by the UART Engine

s1 : holds the 8-bit status flag data of the UART Engine

s3 - (char\_counter) : holds the 8-bit counter data that keeps track of how many characters have been transmitted on the current line (for destructive delete command)

## 6.1.9 Transmit Engine Verification

For this Transmit Engine verification I created a test bench module for both my top\_level and my transmit\_engine files. For the top\_level test bench, all I did was initialize my inputs and set my baud-rate / data format to be 9600 / 8O1. I then ran the simulation and used the waveforms to verify that my Transmit Engine was storing/shifting the correct values out and that the PicoBlaze was outputting the character values at the appropriate times. The Transmit Engine test bench was a lot more involved than my top\_level test bench. In the Transmit Engine test bench, I made sure to test out a variety of data values and I also made sure that I was able to successfully run at all of the different baud-rates specified for my project. I also made sure that each data format (7N1, 7O1, 7E1, 8N1, 8O1m 8E1) worked correctly as well. I verified this by making sure that each 12-bit packet was transmitted in the correct order and by making sure that the appropriate values were being loaded into my shift register I also compared the desired shift register values to the actual shift register values and outputted the result to the console in order to help me debug my shift register and make sure that it was working properly. For a more detailed version of each test bench and its verification, refer to the header section of each test bench module within the source code. It is there that I go into further detail about how I verified the completeness of my design.

## 6.1.10 Receive Engine Verification

I verified that my Receive Engine was working correctly by using a simple test bench. In the Receive Engine test bench, I initialized my UART format to transfer at a baud rate of 460800 and have a format of 8O1. I then made sure that the receive engine correctly received the data of 0xAE. Since I extensively tested the Transmit Engine to make sure that I was able to successfully run at all of the different baud-rates and data formats (7N1, 7O1, 7E1, 8N1, 8O1m 8E1) for my project, I was convinced that the Receive Engine would yield the same results because the baud rate and format logic is almost identical between both engines. This is why I came to the conclusion that I did not need to test every possible baud-rate and data format combination for my Receive Engine. After I received the start bit from the receive data, I then waited for the bit time up to occur before I changed the Rx input value to the Receive Engine. This was so that I wasn't disturbing the Receive Engine while it was waiting for a bit time to begin collecting data. After a bit was shifted into the shift register, I then changed the Rx input in a way that would yield me receiving a byte of 0xAE. After all of the data bits were shifted into the shift register, I then loaded this data into a flop and sent the data to the PicoBlaze. As expected, the end result was that 0xAE was loaded into the flop and outputted to the PicoBlaze decoder (which selects what data is going to the In\_Port of the PicoBlaze) I also made sure that my Parity Error logic was correctly detecting a parity error by assigning the wrong parity to my byte of data. The 8O1 format should have an odd parity value of 0, but I gave the parity value a value of 1. As expected, The PARITY\_ERR status flag was set high and sent to the PicoBlaze decoder. I also made sure that all of my signals were changing correctly and that there were no surprises between the different states in my Receive Engine State Machine. This sums up what I did to verify the correctness of my Receive Engine. Note that I did not make a top level test bench for this project due to the sheer amount of transmits that I have to make for displaying the banner (it is a lot of characters!). Also, it is difficult to simulate the PicoBlaze receiving data because there is no terminal to display data through. This sums up how I verified the correctness of my Receive Engine.

# 7. Chip Level Verification:

For the Chip Level Verification, I verified the chip via simulation by using a series of test benches to verify the correctness of my UART interface. The discussion of what I did to verify my design can be found in sections 6.1.9 and 6.1.10.

# 8. Chip Level Test:

For the Chip Level test, I downloaded the project to a Nexys 2 board and confirmed the correctness of my design. I did this by opening up a terminal window and then making sure that all of the UART communication data was similar. I then pushed reset on the Nexys 2 board and this initialized the system. I then observed the banner being displayed on the terminal window and the command prompt being displayed as well. I then entered different characters on my keyboard and observed them being echoed onto the terminal (or performing the appropriate actions based on the chip requirements). After verifying the different character inputs and making sure that they accurately processed by the PicoBlaze, I then concluded that everything worked exactly as it was expected to.

Code:

Top Level:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 3 - Transmit Engine plus PicoBlaze

// Course: CECS 460

//

// Create Date: 23:38:35 11/09/2015

//

// Module Name: proj4\_UART\_Engine

// File Name: proj4\_UART\_Engine.v

//

// Description: This top\_level module basically ties in all of the other

// modules that I made and connects them together. It takes the

// primary inputs of clk, rstb, bit8, parity\_en, odd\_n\_even,

// baud\_val[3:0], and Rx. This project has one output called tx.

// This tx output goes to P9, which is a port that connects to

// the RS-232 serial connector that connects to a UART terminal

// on a computer via a serial cable. The clk input comes from the

// clock on the Nexys 2 board, while the rstb input is a push

// button. The bit8, parity\_en, odd\_n\_even, and baud\_val[3:0]

// inputs are all switches on the Nexys 2 board. The rstb input

// is synchronized and distributed to all of the other modules in

// this project. This synchronized rstb value (rstbs) is also

// negated and then used as the reset input to the PicoBlaze

// processor. Inside of the PicoBlaze, I display my banner and

// my newline prompt and I then wait for a byte of data to be

// received by checking the RxRdy bit in the status register.

// Once the RxRdy bit is set, I then know that the Receive Engine

// has received a byte and is ready to receive another byte. I

// then load the received byte into a flop and send that data to

// a decoder (which uses read[0] to determine whether I am

// reading in the status of the UART engine or reading in the

// received data. I then output the received byte of data to the

// PicoBlaze. On the second clk cycle of the output instruction,

// I decode the Write\_Strobe value by using the Write\_Strobe and

// the Port\_Id port of the picoblaze. I then put these values

// into my write[255:0] register (because there are 2^8 registers

// that can be written to the PicoBlaze: 00 - FF). For my design,

// I output my values to the Port\_ID of 01, so I needed to pass

// my write[1] value into my transmit engine to let it know

// that it should load in the appropriate data bits (Out\_Port)

// into the shift register. I use the same logic flow for the

// Read\_Strobe (I have a read[255:0] register and I use read[0]

// and read[1] in my design. read[0] is used to read in the

// status flags to the PicoBlaze while read[1] is used to

// notify the Receive Engine that it is going to receive the

// data coming from the terminal window. The baud\_val[3:0] bits

// determine the baud rate at which the Program is going to be

// receiving and transmitting characters. The bit8, parity\_en and

// odd\_n\_even inputs determine the format of the data bits being

// received and transmitted (7N1, 7E1, 7O1, 8N1, 8O1, and 8E1).

// All of these inputs (minus the Rx input) go into my Transmit

// Engine where I update my shift register and baud

// counter accordingly. These inputs (including the Rx input)

// also go into my Receive Engine where I shift in the data one

// bit at a time. Once I establish the desired baud rate

// and data format, I then open a terminal window under the same

// baud rate and data format and then I am ready to start

// receiving/transmitting characters to the UART terminal. In

// the PicoBlaze I display my banner and then I transmit my

// prompt (\*>). I then wait to receive a character from the

// terminal, where I then process the data and either

// issue a newline command and a new prompt (carriage return

// or newline characters), display my hometown (asterisk

// character), delete the previous character (delete/

// backspace character), or echo the character. I also added my

// Seven-Segment display to this lab so that I can display

// my character counter value on the upper 2 Seven-Segment

// displays and the data being received by the Receive Engine

// in the lower 2 Seven-Segment displays. This helped me out

// tremendously in verifying that I was receiving the proper

// character data.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module proj4\_UART\_Engine(clk, rstb, bit8, parity\_en, odd\_n\_even, baud\_val, Rx, tx,

a, b, c, d, e, f, g, a3, a2, a1, a0, LED);

//Input and Output Declarations

input clk, rstb, bit8, parity\_en, odd\_n\_even, Rx;

input [3:0] baud\_val;

output tx, a, b, c, d, e, f, g, a3, a2, a1, a0;

wire tx, a, b, c, d, e, f, g, a3, a2, a1, a0;

output [7:0] LED;

wire [7:0] LED;

//Local Declarations

//wire rstsb, TxSet, TxRst, Done, sSet, sRst, ld, btu, shift;

wire rstsb, tx\_out, TxRdy;

wire [7:0] in\_port, RxData;

wire [2:0] RxStatus;

wire interrupt, read\_strobe, write\_strobe, interrupt\_ack;

wire [7:0] out\_port;

wire [7:0] port\_id;

reg [255:0] write, read;

reg [7:0] charCounter;

assign LED = {1'b0, bit8, parity\_en, odd\_n\_even, baud\_val[3:0]};

assign tx = tx\_out;

//synch\_reset module instantiation

//module synch\_reset(clk, rstb, rstsb);

synch\_reset Synchronizer\_Circuit(

.clk(clk),

.rstb(rstb),

.rstsb(rstsb)

);

//instantiate the RxStatusblaze

embedded\_kcpsm3 ekcp3(

.port\_id(port\_id),

.write\_strobe(write\_strobe),

.read\_strobe(read\_strobe),

.out\_port(out\_port),

.in\_port(in\_port),

.interrupt(interrupt),

.interrupt\_ack(interrupt\_ack),

.reset(!rstsb),

.clk(clk)

);

assign interrupt = 0;

//write strobe and read strobe decode

always@(\*)begin

write = 0;

write[port\_id] = write\_strobe;

read = 0;

read[port\_id] = read\_strobe;

end

//transmit\_engine module instantiation

//module transmit\_engine(clk, rstb, bit8, parity\_en, odd\_n\_even, baud\_val,

//txWrite, inData, tx, TxRdy);

transmit\_engine transmit(

.clk(clk),

.rstb(rstsb),

.bit8(bit8),

.parity\_en(parity\_en),

.odd\_n\_even(odd\_n\_even),

.baud\_val(baud\_val),

.txWrite(write[1]),

.inData(out\_port),

.tx(tx\_out),

.TxRdy(TxRdy)

);

//module receive\_engine(clk, rstb, bit8, parity\_en, odd\_n\_even, baud\_val, READ,

// Rx, status, data);

receive\_engine receive(

.clk(clk),

.rstb(rstsb),

.bit8(bit8),

.parity\_en(parity\_en),

.odd\_n\_even(odd\_n\_even),

.baud\_val(baud\_val),

.READ(read[1:0]),

.Rx(Rx),

.status(RxStatus),

.data(RxData)

);

always@(posedge clk, negedge rstsb)

if(!rstsb)

charCounter <= 8'b00;

else if (write[5])

charCounter <= out\_port;

else

charCounter <= charCounter;

//Display Controller instantiation

//module display\_controller(clk, rstb, annode3, annode2, annode1, annode0,

// a3, a2, a1, a0, a, b, c, d, e, f, g);

display\_controller Display\_Controller(

.clk(clk),

.rstb(rstsb),

.annode3(charCounter[7:4]),

.annode2(charCounter[3:0]),

.annode1(RxData[7:4]),

.annode0(RxData[3:0]),

.a3(a3),

.a2(a2),

.a1(a1),

.a0(a0),

.a(a),

.b(b),

.c(c),

.d(d),

.e(e),

.f(f),

.g(g)

);

assign in\_port[7:0] = (read[0]) ? {2'b0, RxStatus[2:1], 2'b0, TxRdy, RxStatus[0]} : RxData;

endmodule

Synchronizer Circuit:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 3 - Transmit Engine plus PicoBlaze

// Course: CECS 460

// Create Date: 19:08:36 04/09/2015

//

// Module Name: synch\_reset

// File Name: synch\_reset.v

//

// Description: This is the Syncronization Circuit module that has an

// asynchronous input and a synchronous output. The purpose of this

// module is to take the asyncronous reset input and make it low-

// leveled active when pressed. This means that the reset value

// should be zero whenever the button is activated. When the reset

// button is pressed on the Nexys 2 board, its value comes into

// this module as the rstb input. However, in this module there are

// flip-flops that are designed with asynchronous (low active)

// resets. Also in this module, there is a synchronous reset that

// will neglect the asynchronous rstb input and force the 2

// flip-flops to zero at the posedge of the clock. This is how the

// rstb input turns from an input of 1 to an output of 0 and

// vice-versa. In other words, this is how the asynchronous rstb

// input turns into the synchronous rstsb output.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module synch\_reset(clk, rstb, rstsb);

//Input and Output declarations

input clk, rstb;

output wire rstsb;

//local variables / flip-flop declarations

reg rst\_reg\_delay;

reg rst\_synch\_out;

assign rstsb = rst\_synch\_out;

always@(posedge clk, posedge rstb)begin

if(rstb)begin //force rstb to be low when activated instead of being high

rst\_reg\_delay <= 1'b0;

rst\_synch\_out <= 1'b0;

end

else begin //Synchronize the output

rst\_reg\_delay <= 1'b1;

rst\_synch\_out <= rst\_reg\_delay;

end

end

endmodule

PicoBlaze Processor:

////////////////////////////////////////////////////////////////////////////////

// Copyright (c) 2004 Xilinx, Inc.

// All Rights Reserved

////////////////////////////////////////////////////////////////////////////////

// \_\_\_\_ \_\_\_\_

// / /\/ /

// /\_\_\_/ \ / Vendor: Xilinx

// \ \ \/ Version: 1.01

// \ \ Filename: embedded\_kcpsm3.v

// / / Date Last Modified: 08/04/2004

// /\_\_\_/ /\ Date Created: 06/03/2003

// \ \ / \

// \\_\_\_\/\\_\_\_\

//

//Device: Xilinx

//Purpose:

// This file instantiates the KCPSM3 processor macro and connects the

// program ROM.

//Reference:

// None

//Revision History:

// Rev 1.00 - kc - Start of design entry in VHDL, 06/03/2003.

// Rev 1.01 - sus - Converted to verilog, 08/04/2004.

////////////////////////////////////////////////////////////////////////////////

// NOTE: The name of the program ROM will probably need to be changed to

// reflect the name of the program (PSM) file applied to the assembler.

////////////////////////////////////////////////////////////////////////////////

// Contact: e-mail picoblaze@xilinx.com

//////////////////////////////////////////////////////////////////////////////////

//

// Disclaimer:

// LIMITED WARRANTY AND DISCLAIMER. These designs are

// provided to you "as is". Xilinx and its licensors make and you

// receive no warranties or conditions, express, implied,

// statutory or otherwise, and Xilinx specifically disclaims any

// implied warranties of merchantability, non-infringement, or

// fitness for a particular purpose. Xilinx does not warrant that

// the functions contained in these designs will meet your

// requirements, or that the operation of these designs will be

// uninterrupted or error free, or that defects in the Designs

// will be corrected. Furthermore, Xilinx does not warrant or

// make any representations regarding use or the results of the

// use of the designs in terms of correctness, accuracy,

// reliability, or otherwise.

//

// LIMITATION OF LIABILITY. In no event will Xilinx or its

// licensors be liable for any loss of data, lost profits, cost

// or procurement of substitute goods or services, or for any

// special, incidental, consequential, or indirect damages

// arising from the use or operation of the designs or

// accompanying documentation, however caused and on any theory

// of liability. This limitation will apply even if Xilinx

// has been advised of the possibility of such damage. This

// limitation shall apply not-withstanding the failure of the

// essential purpose of any limited remedies herein.

//////////////////////////////////////////////////////////////////////////////////

module embedded\_kcpsm3(

port\_id,

write\_strobe,

read\_strobe,

out\_port,

in\_port,

interrupt,

interrupt\_ack,

reset,

clk);

output[7:0] port\_id;

output write\_strobe;

output read\_strobe;

output[7:0] out\_port;

input[7:0] in\_port;

input interrupt;

output interrupt\_ack;

input reset;

input clk;

wire [7:0] port\_id;

wire write\_strobe;

wire read\_strobe;

wire [7:0] out\_port;

wire [7:0] in\_port;

wire interrupt;

wire interrupt\_ack;

wire reset;

wire clk;

wire [9:0] address;

wire [17:0] instruction;

//----------------------------------------------------------------------------------

//

// declaration of KCPSM3

//

//

// declaration of program ROM

//

//----------------------------------------------------------------------------------

//

// Start of test circuit description

//

kcpsm3 processor

( .address(address),

.instruction(instruction),

.port\_id(port\_id),

.write\_strobe(write\_strobe),

.out\_port(out\_port),

.read\_strobe(read\_strobe),

.in\_port(in\_port),

.interrupt(interrupt),

.interrupt\_ack(interrupt\_ack),

.reset(reset),

.clk(clk));

// uart\_rx program

// ( .address(address),

// .instruction(instruction),

// .clk(clk));

uart\_ful program

( .address(address),

.instruction(instruction),

.clk(clk));

endmodule

//----------------------------------------------------------------------------------

//

// END OF FILE EMBEDDED\_KCPSM3.V

//

//----------------------------------------------------------------------------------

Transmit Engine:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 3 - Transmit Engine plus PicoBlaze

// Course: CECS 460

//

// Create Date: 11:37:29 10/15/2015

//

// Module Name: transmit\_engine

// File Name: transmit\_engine.v

//

// Description: My transmit engine module contains all of the logic needed to

// communicate with a computer via universal asynchronous

// receiver/transmitter (UART) communication. In order to create

// a stable UART connection, I needed to make sure that I could

// run my program at different baud rates and different data

// formats (7N1, 7O1, 7E1, 8N1, 8O1, 8E1). For this program, I

// am shifting out the data in 12-bit data packets. I use a

// shift register to determine what data gets shifted out. I am

// able to load in the data that is going to be shifted out

// by delaying the write[1] strobe from the PicoBlaze by 1 clock

// cycle (this is because the data on the out\_port of the

// PicoBlaze is not available until the next clock cycle). If

// my write1Delay variable is set high, I then load my shift

// register with the out\_port data of the PicoBlaze. I then

// update the remaining values in the shift register to make

// sure that I have accounted for the right parity, stop, and

// start bits. This is done using the bit8, parity\_en, and

// odd\_n\_even inputs. Once all of the data in the shift register

// is accurate, I then start shifting out the bits 1 bit at a

// time. I fill the most significant bit of my shift register

// with a 1 (which is equivalent to filling up the register

// with a bunch of stop bits). I keep track of when I am done

// shifting out all of the bits in my shift register by using

// a bit counter. In order to know when a bit is done being

// shifted out, I also needed to use a baud-rate counter.

// I first decoded the baud rate by reading in the data on my

// four baud\_rate switches. The next step was to determine what

// baud-rate I was running at based on those four switches. After

// determining the baud rate, I then needed to determine the

// amount of ticks I would need to achieve that specified baud-rate.

// I did this by using the formula baudCountNum = (1/baud rate) /

// (1/50MHz). This gave me the terminal count number that I needed

// to achieve the desired baud rate. I then assigned my baudCountNum

// variable to this terminal count. Once that value was reached, I

// then knew that I had finished transmitting one byte via UART

// communication. Now the next step was to increment my Bit Counter.

// I repeated this process until my Bit Counter reached 11, meaning

// that I had finished transmitting all 12 bits of data in the

// data packet. For this project, we are always going to end up

// transmitting 12 bits of data per data packet. That is why my

// bit time counter limit is always 11 (0 - (12-1)). Once all 12

// bits of data have been shifted out, I then set my Done variable

// to 1, which also sets my TxRdy variable to a 1. I then output

// this TxRdy variable, along with the other status variables

// to the PicoBlaze. These status variables are concatenated and

// represent the status register of the Transmit Engine. The

// TxRdy bit lets the PicoBlaze know that the Transmit Engine is

// ready to transmit another byte, at which point the PicoBlaze

// will output the next character being transmitted in the

// sequence to the Transmit Engine.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module transmit\_engine(clk, rstb, bit8, parity\_en, odd\_n\_even, baud\_val, txWrite,

inData, tx, TxRdy);

//Input and Output Declarations

input clk, rstb, bit8, parity\_en, odd\_n\_even, txWrite;

input [3:0] baud\_val;

input [7:0] inData;

output tx;

output reg TxRdy;

//output transferLimit;

//Local Declarations

wire TxSet, TxRst, Done, sSet, sRst, ld, btu, shift;

reg [7:0] data;

//reg [6:0] transferLimitCounter;

reg [17:0] baudCountNum, baudCount, nBaudCount;

reg [3:0] bitCount, nBitCount;

reg start, write1Delay;

reg [11:0] sreg\_data;

reg b7, b8;

assign TxSet = Done;

assign TxRst = txWrite;

//R-S Flop that controlls the TxRdy input

//for the UART

always@(posedge clk, negedge rstb)

if(!rstb)

TxRdy <= 1'b1;

else if(TxSet)

TxRdy <= 1'b1;

else if(TxRst)

TxRdy <= 1'b0;

else

TxRdy <= TxRdy;

//assign pico\_data[7:0] = {6'b0, TxRdy, 1'b0};

assign sSet = txWrite;

assign sRst = Done;

//R-S Flop that controlls the start input

//for the UART

always@(posedge clk, negedge rstb)

if(!rstb)

start <= 1'b0;

else if(sSet)

start <= 1'b1;

else if(sRst)

start <= 1'b0;

else

start <= start;

assign ld = txWrite;

//register that holds the outport data

always@(posedge clk, negedge rstb)

if(!rstb)

data <= 8'b00;

else if (ld)

data <= inData;

else

data <= data;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Shift register logic

//register that delays the write strobe

//for one clock cycle

always@(posedge clk, negedge rstb)

if(!rstb)

write1Delay <= 1'b0;

else

write1Delay <= txWrite;

//logic used to determine b7 and b8 of the shift register (value changes

//depending on what bit8, parity\_en, and odd\_n\_even is).

always @(\*)

case ({bit8,parity\_en,odd\_n\_even})

3'b000: {b8,b7} = 2'b11; //7N1

3'b001: {b8,b7} = 2'b11; //7N1

3'b010: {b8,b7} = {1'b1, ^data[6:0]}; //7E1

3'b011: {b8,b7} = {1'b1, ~(^data[6:0])}; //7O1

3'b100: {b8,b7} = {1'b1, data[7]}; //8N1

3'b101: {b8,b7} = {1'b1, data[7]}; //8N1

3'b110: {b8,b7} = {(^data[7:0]), data[7]}; //8E1

3'b111: {b8,b7} = {~(^data[7:0]), data[7]}; //8O1

default: {b8,b7} = 2'b00; //no bits

endcase

//update shift register value uf write1Delay is set high

always @( posedge clk, negedge rstb)

if (!rstb)

sreg\_data <=12'hFFF; //load with all 1's

else if (write1Delay)

sreg\_data <= {1'b1,b8,b7,data[6:0], 1'b0, 1'b1};//load data into sreg

else if (shift)begin

sreg\_data <= sreg\_data >> 1; //shift data right by 1

sreg\_data [11] <= 1; //fill msb of shift register with a 1

end

assign tx = sreg\_data[0];

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//BAUD TIME COUNTER LOGIC:

//baud rate count decoder:

//values are derived using (1/baud rate) / (1/50MHz)

always@(\*)

case(baud\_val)

4'b0000: baudCountNum = 166667 -1; //300 BAUD Rate

4'b0001: baudCountNum = 41667 - 1; //1200 BAUD Rate

4'b0010: baudCountNum = 20833 - 1; //2400 BAUD Rate

4'b0011: baudCountNum = 10417 - 1; //4800 BAUD Rate

4'b0100: baudCountNum = 5208 - 1; //9600 BAUD Rate

4'b0101: baudCountNum = 2604 - 1; //19200 BAUD Rate

4'b0110: baudCountNum = 1302 - 1; //38400 BAUD Rate

4'b0111: baudCountNum = 868 - 1; //57600 BAUD Rate

4'b1000: baudCountNum = 434 - 1; //115200 BAUD Rate

4'b1001: baudCountNum = 217 - 1; //230400 BAUD Rate

4'b1010: baudCountNum = 109 - 1; //460800 BAUD Rate

4'b1011: baudCountNum = 54 - 1; //921600 BAUD Rate

default: baudCountNum = 166667 - 1; //300 BAUD Rate

endcase

//assign btu output tick and shift wire

assign btu = (baudCount == baudCountNum) ? 1'b1 : 1'b0;

assign shift = btu;

//Determine next state of baudCount

always@(\*)

case({btu, start})

2'b00 : nBaudCount = 18'b00;

2'b01 : nBaudCount = baudCount + 1;

2'b10 : nBaudCount = 18'b00;

2'b11 : nBaudCount = 18'b00;

default : nBaudCount = 18'b00;

endcase

//update baudCount accordingly

always @(posedge clk, negedge rstb)

if(!rstb) //check reset bit

baudCount <= 18'b0; //reset counter

else

baudCount <= nBaudCount; //update baudCount

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//BIT COUNTER LOGIC:

//For this lab we are always going to be transmitting 12 bits of data at a

//time. Once our counter has established that all 12 bits have been

//transmitted, we then set the Done variable high.

//assign Done output tick

assign Done = (bitCount == 11) ? 1'b1 : 1'b0;

//Determine next state of bitCount

always@(\*)

case({btu, start})

2'b00 : nBitCount = 4'b00;

2'b01 : nBitCount = (Done) ? 4'b00 : bitCount;

2'b10 : nBitCount = 4'b00;

2'b11 : nBitCount = bitCount + 1;

default : nBitCount = 4'b00;

endcase

//update bitCount accordingly

always @(posedge clk, negedge rstb)

if(!rstb) //check reset bit

bitCount <= 4'b0; //reset counter

else

bitCount <= nBitCount; //update bitCount

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

endmodule

Receive Engine:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 4 - Full UART Engine

// Course: CECS 460

//

// Create Date: 11:37:29 10/15/2015

//

// Module Name: receive\_engine

// File Name: receive\_engine.v

//

// Description: My receive engine module contains all of the logic needed to

// receive information from a computer via universal asynchronous

// receiver/transmitter (UART) communication. In order to create

// a stable UART connection, I needed to make sure that I could

// run my program at different baud rates and different data

// formats (7N1, 7O1, 7E1, 8N1, 8O1, 8E1). For this program, I

// am shifting in the data from the receive engine 1-bit at a time.

// I use a shift register to store the receive engine data bits.

// Each data bit gets shifted in one bit at a time. For the receive

// engine we have a state diagram that consists of four different

// states: IDLE, S1, S2, and S3. The state machine starts off in

// the IDLE state at reset. In this state, it waits for a start bit

// (which is agreed to be a zero). We then move to the S1 state,

// where the state machine waits for half a bit-time. If the Rx

// input changes to a 1 during this half bit-time interval, I then

// move back to the IDLE state where we reset the bit-time interval

// and wait to receive a start bit value (0). If the Rx input

// stays the same (0) for the entirety of the half-bit time, this

// means that we actually received a start bit and I then move on to

// the next state, S2. I now know that we need to start collecting

// the data bits, so in S2, I wait wait for an entire bit time.

// After the bit time is achieved, I am know in the middle of the

// first data bit. I then move on to state S3, where I shift the

// data bit value into my shift register. I then check to see if

// all of the data bits have been shifted in to my shift register

// by checking the status of my bcu variable. This variable is set

// high for one clock cycle whenever all of the data bits have

// been received. If the bcu variable is set high, then I go back

// to the idle state where I wait until another bit is received.

// If the variable is not set high, then I go back to S2 and wait

// for another bit time so that I can shift in the next bit into

// the receive engine. Once all of the data bits have been

// shifted into the receive engine, I then send these data bits

// to the PicoBlaze so that it can process the received character

// and determine what to do with that data.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module receive\_engine(clk, rstb, bit8, parity\_en, odd\_n\_even, baud\_val, READ,

Rx, status, data);

//Input and Output Declarations

input clk, rstb, bit8, parity\_en, odd\_n\_even, Rx;

input [1:0] READ;

input [3:0] baud\_val;

output [7:0] data;

output [2:0] status;

reg [7:0] data;

wire [2:0] status;

//Local Declarations

//symbolic state declaration

localparam [2:0]

idle = 2'b00,

state\_1 = 2'b01,

state\_2 = 2'b10,

state\_3 = 2'b11;

wire RxSet, RxRst, sSet, sRst, SEVEN, btu, bcu, done;//shift;

wire parSet, parRst;

reg START, DOIT, LOAD, nSTART, nDOIT, nLOAD;

wire [17:0] finalBaudCountNum;

reg [17:0] baudCountNum, baudCount, nBaudCount;

reg [3:0] bitCount, nBitCount;

reg [1:0] present\_state, next\_state;

reg RxRdy, OVERFLOW, doneDelay;

reg [8:0] sreg\_data;

wire b7, receivedParity, computedParity, generatedParity;

wire combinedParities, parityResult;

reg PARITY\_ERR;

assign RxSet = doneDelay & !RxRdy;

assign RxRst = READ[1];

//R-S Flop that controlls the RxRdy input

//for the UART

always@(posedge clk, negedge rstb)

if(!rstb)

RxRdy <= 1'b1;

else if(RxSet)

RxRdy <= 1'b1;

else if(RxRst)

RxRdy <= 1'b0;

else

RxRdy <= RxRdy;

assign sSet = LOAD & RxRdy;

assign sRst = READ[0];

//R-S Flop that controlls the overflow output

//for the UART

always@(posedge clk, negedge rstb)

if(!rstb)

OVERFLOW <= 1'b0;

else if(sSet)

OVERFLOW <= 1'b1;

else if(sRst)

OVERFLOW <= 1'b0;

else

OVERFLOW <= OVERFLOW;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Finite State Machine for RX system

//For the receive engine we have a state diagram that consists of four different

//states: IDLE, S1, S2, and S3. The state machine starts off in

//the IDLE state at reset. In this state, it waits for a start bit

//(which is agreed to be a zero). We then move to the S1 state,

//where the state machine waits for half a bit-time. If the Rx

//input changes to a 1 during this half bit-time interval, I then

//move back to the IDLE state where we reset the bit-time interval

//and wait to receive a start bit value (0). If the Rx input

//stays the same (0) for the entirety of the half-bit time, this

//means that we actually received a start bit and I then move on to

//the next state, S2. I now know that we need to start collecting

//the data bits, so in S2, I wait wait for an entire bit time.

//After the bit time is achieved, I am know in the middle of the

//first data bit. I then move on to state S3, where I shift the

//data bit value into my shift register. I then check to see if

//all of the data bits have been shifted in to my shift register

//by checking the status of my bcu variable. This variable is set

//high for one clock cycle whenever all of the data bits have

//been received. If the bcu variable is set high, then I go back

//to the idle state where I wait until another bit is received.

//If the variable is not set high, then I go back to S2 and wait

//for another bit time so that I can shift in the next bit into

//the receive engine. Once all of the data bits have been

//shifted into the receive engine, I then send these data bits

//to the PicoBlaze so that it can process the received character

//and determine what to do with that data.

always@(posedge clk, negedge rstb)begin

if(!rstb)begin

present\_state <= idle;

{START, DOIT, LOAD} <= 3'b000; //Reset outputs

end

else begin

present\_state <= next\_state; //update the present state

{START, DOIT, LOAD} <= {nSTART, nDOIT, nLOAD}; //update present outputs

end

end

always@(\*)begin

next\_state = present\_state; //default state: the same

{nSTART, nDOIT, nLOAD} = 3'b000; //wait for start bit

case(present\_state)

idle : begin

{nSTART, nDOIT, nLOAD} = 3'b000; //wait for start bit

next\_state = (Rx) ? idle : state\_1;

end

state\_1 : begin

{nSTART, nDOIT, nLOAD} = 3'b110; //wait for half bit-time

next\_state = (Rx) ? idle : ((btu) ? state\_2 : state\_1 );

end

state\_2 : begin

{nSTART, nDOIT, nLOAD} = 3'b010; //wait for full bit-time

next\_state = (btu) ? state\_3 : state\_2;

end

state\_3 : begin

{nSTART, nDOIT, nLOAD} = 3'b011; //load bit into shift register

next\_state = (bcu) ? idle : state\_2;

end

default : begin

{nSTART, nDOIT, nLOAD} = 3'b000; //wait for start bit

next\_state = idle;

end

endcase

end

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//update shift register value until it contains all of the receive data

always @(posedge clk, negedge rstb)

if (!rstb)

sreg\_data <=9'hFFF; //load with all 1's

else if (!START & !DOIT & !LOAD)

sreg\_data <=9'hFFF; //load with all 1's

else if (LOAD & !START)begin

sreg\_data <= sreg\_data >> 1; //shift data right by 1

sreg\_data [8] <= Rx; //fill msb of shift register with Rx input

end

assign SEVEN = !bit8;

//generate the parity error result

assign b7 = (SEVEN) ? 1'b0 : sreg\_data[7];

assign receivedParity = (SEVEN) ? sreg\_data[7] : sreg\_data[8];

assign computedParity = ^{b7,sreg\_data[6:0]};

assign generatedParity = (odd\_n\_even) ? ~computedParity : computedParity;

assign combinedParities = ^{receivedParity, generatedParity};

assign parityResult = doneDelay & parity\_en & combinedParities;

assign parSet = parityResult;

assign parRst = READ[0];

//R-S Flop that controlls the Parity Error output for the UART

always@(posedge clk, negedge rstb)

if(!rstb)

PARITY\_ERR <= 1'b0;

else if(parSet)

PARITY\_ERR <= 1'b1;

else if(parRst)

PARITY\_ERR <= 1'b0;

else

PARITY\_ERR <= PARITY\_ERR;

//data to picoblaze

always @(posedge clk, negedge rstb)

if(!rstb)

data <= 8'b0;

else if (doneDelay)

data <= {b7, sreg\_data[6:0]};

//status to picoblaze

assign status = {OVERFLOW, PARITY\_ERR, RxRdy};

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//BAUD TIME COUNTER LOGIC:

//baud rate count decoder:

//values are derived using (1/baud rate) / (1/50MHz)

always@(\*)

case(baud\_val)

4'b0000: baudCountNum = 166667 -1; //300 BAUD Rate

4'b0001: baudCountNum = 41667 - 1; //1200 BAUD Rate

4'b0010: baudCountNum = 20833 - 1; //2400 BAUD Rate

4'b0011: baudCountNum = 10417 - 1; //4800 BAUD Rate

4'b0100: baudCountNum = 5208 - 1; //9600 BAUD Rate

4'b0101: baudCountNum = 2604 - 1; //19200 BAUD Rate

4'b0110: baudCountNum = 1302 - 1; //38400 BAUD Rate

4'b0111: baudCountNum = 868 - 1; //57600 BAUD Rate

4'b1000: baudCountNum = 434 - 1; //115200 BAUD Rate

4'b1001: baudCountNum = 217 - 1; //230400 BAUD Rate

4'b1010: baudCountNum = 109 - 1; //460800 BAUD Rate

4'b1011: baudCountNum = 54 - 1; //921600 BAUD Rate

default: baudCountNum = 166667 - 1; //300 BAUD Rate

endcase

//The START signal determines whether I am counting for a full bit time

//or half of a bit time. When START is high, I assign the count value to

//half of the original value (by shifting it to the right once). This

//means that I am looking for the start bit of the received data (0)

//and as a result I am only waiting for half of a bit-time. When START

//is low, I assign te count value to the original value (meaning that I

//am waiting for a full bit-time).

assign finalBaudCountNum = (START) ? (baudCountNum >> 1) : baudCountNum;

//assign btu output tick and shift wire

assign btu = (baudCount == finalBaudCountNum) ? 1'b1 : 1'b0;

//assign shift = btu;

//Determine next state of baudCount

always@(\*)

case({btu, DOIT})

2'b00 : nBaudCount = 18'b00;

2'b01 : nBaudCount = baudCount + 1;

2'b10 : nBaudCount = 18'b00;

2'b11 : nBaudCount = 18'b00;

default : nBaudCount = 18'b00;

endcase

//update baudCount accordingly

always @(posedge clk, negedge rstb)

if(!rstb) //check reset bit

baudCount <= 18'b0; //reset counter

else

baudCount <= nBaudCount; //update baudCount

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//BIT COUNTER LOGIC:

//For this lab we are always going to be transmitting 12 bits of data at a

//time. Once our counter has established that all 12 bits have been

//transmitted, we then set the Done variable high.

//assign Done output tick

assign bcu = (bitCount == 11) ? 1'b1 : 1'b0;

assign done = bcu;

//Determine next state of bitCount

always@(\*)

case({btu, DOIT})

2'b00 : nBitCount = 4'b00;

2'b01 : nBitCount = (done) ? 4'b00 : bitCount;

2'b10 : nBitCount = 4'b00;

2'b11 : nBitCount = bitCount + 1;

default : nBitCount = 4'b00;

endcase

//update bitCount accordingly

always @(posedge clk, negedge rstb)

if(!rstb) //check reset bit

bitCount <= 4'b0; //reset counter

else

bitCount <= nBitCount; //update bitCount

//done delay (allows time to load the bit into the shift register)

always@(posedge clk, negedge rstb)

if(!rstb) //check reset bit

doneDelay <= 1'b0;

else

doneDelay <= done;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

endmodule

Display Controller

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 2 - PicoBlaze Integration with Interrupts

// Course: CECS 460

// Create Date: 13:19:08 04/18/2015

//

// Module Name: top\_level

// File Name: top\_level.v

//

// Description: This Display Controller module controls the 7-Segment display

// and everything that it needs in order to correctly display

// the appropriate values. This module was created in Professor

// Allison's CECS 301 class. Each section of this module is

// described within the module itself using comments.

//

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module display\_controller(clk, rstb, annode3, annode2, annode1, annode0,

a3, a2, a1, a0, a, b, c, d, e, f, g);

//Input and Output Declarations

input clk, rstb;

input [3:0] annode3, annode2, annode1, annode0;

//annode and cathode variables for the 7-Segment Display

output reg a, b, c, d, e, f, g;

output wire a3, a2, a1, a0;

//Local Variables

reg [3:0] ad\_out;

wire [1:0] seg\_sel;

reg clk\_250;

//counter variable

reg [15:0] count = 0;

//initialize LED clock:

//Divide 50MHz on-board clock to 250Hz. This is done so that

//it appears to the human observer that all four anode signals

//are on even when in reality they are not. This is done by

//driving the cathode patterns at least once every 16ms,

//which comes out to 4ms per annode/cathode pattern.

always @(posedge clk, negedge rstb)

//if rstb then rstb count and clock

if(!rstb) begin

count <= 0;

clk\_250 <= 0;

end

else begin

//increment count

count <= (count + 1);

//set output clock to opposite if count is greater than or equal to 25000

if(count >= 25000) begin

count <= 0;

clk\_250 <= ~clk\_250;

end

end

//initialize LED controller module:

//module led\_controller(clk, rstb, a3, a2, a1, a0, seg\_sel);

led\_controller LED\_contr(

.clk(clk\_250),

.rstb(rstb),

.a3(a3),

.a2(a2),

.a1(a1),

.a0(a0),

.seg\_sel(seg\_sel)

);

//initialize address mux:

//This multiplexer is used to select what data input is to

//be assigned to the ad\_out output and driven to the seven segment

//display. The value of the ad\_out is chosen based on the seg\_sel

//input that was generated by our led\_controller module.

always @(seg\_sel, annode3, annode2, annode1, annode0)

case (seg\_sel)

2'b00 : ad\_out = annode0; //annode 0

2'b01 : ad\_out = annode1; //annode 1

2'b10 : ad\_out = annode2; //annode 2

2'b11 : ad\_out = annode3; //annode 3

default: ad\_out = 4'bx; //default output

endcase

//initialize hex to seven segment display

//This hex\_to\_7segment module basicaly takes in a four bit input

//and depending on what that input is, it then transfers that

//information by turning on the appropriate segments (a-g) on the

//seven segment display in order to represent that four bit input.

//This was achieved by using case statements for all of the

//appropriate 4-bit input options and assigning the according

//values for a-g for that 4-bit input. Depending on the 4-bit

//value, the appropriate segments are turned on in order to

//display that value.

always@(ad\_out)

case (ad\_out)

4'b0000: {a, b, c, d, e, f, g} = 7'b0000001; //Display 0

4'b0001: {a, b, c, d, e, f, g} = 7'b1001111; //Display 1

4'b0010: {a, b, c, d, e, f, g} = 7'b0010010; //Display 2

4'b0011: {a, b, c, d, e, f, g} = 7'b0000110; //Display 3

4'b0100: {a, b, c, d, e, f, g} = 7'b1001100; //Display 4

4'b0101: {a, b, c, d, e, f, g} = 7'b0100100; //Display 5

4'b0110: {a, b, c, d, e, f, g} = 7'b0100000; //Display 6

4'b0111: {a, b, c, d, e, f, g} = 7'b0001111; //Display 7

4'b1000: {a, b, c, d, e, f, g} = 7'b0000000; //Display 8

4'b1001: {a, b, c, d, e, f, g} = 7'b0001100; //Display 9

4'b1010: {a, b, c, d, e, f, g} = 7'b0001000; //Display A

4'b1011: {a, b, c, d, e, f, g} = 7'b1100000; //Display B

4'b1100: {a, b, c, d, e, f, g} = 7'b0110001; //Display C

4'b1101: {a, b, c, d, e, f, g} = 7'b1000010; //Display D

4'b1110: {a, b, c, d, e, f, g} = 7'b0110000; //Display E

4'b1111: {a, b, c, d, e, f, g} = 7'b0111000; //Display F

default: {a, b, c, d, e, f, g} = 7'b1111111; //Display Nothing

endcase

endmodule

LED Controller:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 2 - PicoBlaze Integration with Interrupts

// Course: CECS 460

// Create Date: 20:10:05 10/12/2013

//

// Module Name: led\_controller

// File Name: led\_controller.v

//

// Description: This is an LED controller based on Professor Allison's CECS

// 301 class. The purpose of this module is to choose which

// anodes on the 7-segment displays are asserted and when they

// should be asserted. This is determined according to the LED

// clock within the Display Controller module.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module led\_controller(clk, rstb, a3, a2, a1, a0, seg\_sel);

//Input and Output Declarations

input clk, rstb;

output reg a3, a2, a1, a0;

output reg [1:0] seg\_sel;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// state register and next\_state variables

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reg [1:0] present\_state;

reg [1:0] next\_state;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Next State Combinational Logic

//(next state values can change anytime but will only be "clocked" below)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

always @(present\_state)

case (present\_state)

//next state combinations for present states

2'b00 : next\_state = 2'b01;

2'b01 : next\_state = 2'b10;

2'b10 : next\_state = 2'b11;

2'b11 : next\_state = 2'b00;

//next state combinations for all other present states

default: next\_state = 2'b00;

endcase

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// State Register Logic (Sequential Logic)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

always @(posedge clk, negedge rstb)

if (!rstb)

//Got a rstb, so set state register to all 0's

present\_state <= 2'b00;

else

//Got a posedge, so update state register with next state

present\_state <= next\_state;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Output Combinational Logic and Segment Select Logic

//(output values can change only when a present state changes)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

always @(present\_state)

case (present\_state)

//output combinations for present states

2'b00 : {seg\_sel, a3, a2, a1, a0} = 6'b00\_1110; //a[0] seg\_sel asserted

2'b01 : {seg\_sel, a3, a2, a1, a0} = 6'b01\_1101; //a[1] seg\_sel asserted

2'b10 : {seg\_sel, a3, a2, a1, a0} = 6'b10\_1011; //a[2] seg\_sel asserted

2'b11 : {seg\_sel, a3, a2, a1, a0} = 6'b11\_0111; //a[3] asserted

//output combinations for all other present states

default : {seg\_sel, a3, a2, a1, a0} = 6'b00\_1110;

endcase

endmodule

UCF File:

//Set the appropriate step and rstb

//clk input

NET "clk" LOC="B8";

//button inputs

NET "rstb" LOC="B18";

NET "tx" LOC="P9";

NET "Rx" LOC="U6";

//NET "rx" LOC="V6";

#NET "parity\_en" LOC="N17";

#NET "odd\_n\_even" LOC="L13";

#NET "bit8" LOC="L14";

NET "bit8" LOC="N17";

NET "parity\_en" LOC="L13";

NET "odd\_n\_even" LOC="L14";

NET "baud\_val[3]" LOC="K17";

NET "baud\_val[2]" LOC="K18";

NET "baud\_val[1]" LOC="H18";

NET "baud\_val[0]" LOC="G18";

//annode and cathode info for the 7

//segment display

NET "a3" LOC="F15"| IOSTANDARD = LVCMOS33;

NET "a2" LOC="C18"| IOSTANDARD = LVCMOS33;

NET "a1" LOC="H17"| IOSTANDARD = LVCMOS33;

NET "a0" LOC="F17"| IOSTANDARD = LVCMOS33;

NET "a" LOC= "L18"| IOSTANDARD = LVCMOS33;

NET "b" LOC= "F18"| IOSTANDARD = LVCMOS33;

NET "c" LOC= "D17"| IOSTANDARD = LVCMOS33;

NET "d" LOC= "D16"| IOSTANDARD = LVCMOS33;

NET "e" LOC= "G14"| IOSTANDARD = LVCMOS33;

NET "f" LOC= "J17"| IOSTANDARD = LVCMOS33;

NET "g" LOC= "H14"| IOSTANDARD = LVCMOS33;

NET "LED[7]" LOC= "R4"| IOSTANDARD = LVCMOS33;

NET "LED[6]" LOC= "F4"| IOSTANDARD = LVCMOS33;

NET "LED[5]" LOC= "P15"| IOSTANDARD = LVCMOS33;

NET "LED[4]" LOC= "E17"| IOSTANDARD = LVCMOS33;

NET "LED[3]" LOC= "K14"| IOSTANDARD = LVCMOS33;

NET "LED[2]" LOC= "K15"| IOSTANDARD = LVCMOS33;

NET "LED[1]" LOC= "J15"| IOSTANDARD = LVCMOS33;

NET "LED[0]" LOC= "J14"| IOSTANDARD = LVCMOS33;

Receive Engine Test Bench:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 4 - Full UART Engine

// Course: CECS 460

//

// Create Date: 18:56:08 11/13/2015

//

// Module Name: receive\_engine\_tb

// File Name: receive\_engine\_tb.v

//

// Description: I verified that my Receive Engine was working correctly by using

// a simple test bench. In the Receive Engine test bench, I

// initialized my UART format to transfer at a baud rate of 300

// and have a format of 8O1. I then made sure that the receive

// engine correctly received the data of 0xAE. Since I extensively

// tested the Transmit Engine to make sure that I was able to

// successfully run at all of the different baud-rates and data

// formats (7N1, 7O1, 7E1, 8N1, 8O1m 8E1) for my project, I was

// convinced that the Receive Engine would yield the same results

// because the baud rate and format logic is almost identical

// between both engines. This is why I came to the conclusion that

// I did not need to test every possible baud-rate and data format

// combination for my Receive Engine. After I received the start bit

// from the receive data, I then waited for the bit time up to occur

// before I changed the Rx input value to the Receive Engine. This

// was so that I wasn't disturbing the Receive Engine while it was

// waiting for a bit time to begin collecting data. After a bit was

// shifted into the shift register, I then changed the Rx input in a

// way that would yield me receiving a byte of 0xAE. After all of the

// data bits were shifted into the shift register, I then loaded

// this data into a flop and sent the data to the PicoBlaze. As

// expected, the end result was that 0xAE was loaded into the flop

// and outputted to the PicoBlaze decoder (which selects what data

// is going to the In\_Port of the PicoBlaze) I also made sure that

// my Parity Error logic was correctly detecting a parity error by

// assigning the wrong parity to my byte of data. The 8O1 format

// should have an odd parity value of 0, but I gave the parity value

// a value of 1. As expected, The PARITY\_ERR status flag was set high

// and sent to the PicoBlaze decoder. I also made sure that all of my

// signals were changing correctly and that there were no surprises

// between the different states in my Receive Engine State Machine.

// This sums up what I did to verify the correctness of my Receive

// Engine. Note that I did not make a top level test bench for this

// project due to the sheer amount of transmits that I have to make

// for displaying the banner (it is a lot of characters!). Also, it

// is difficult to simulate the PicoBlaze receiving data because

// there is no terminal to display data through. This sums up how I

// verified the correctness of my Receive Engine.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module receive\_engine\_tb;

//Inputs

reg clk;

reg rstb;

reg bit8;

reg parity\_en;

reg odd\_n\_even;

reg [3:0] baud\_val;

reg [1:0] READ;

reg Rx;

wire [2:0] status;

//Outputs

wire [7:0] data;

//Instantiate the Unit Under Test (UUT)

//receive\_engine(clk, rstb, bit8, parity\_en, odd\_n\_even, baud\_val, READ,

// Rx, status, data);

receive\_engine uut(

.clk(clk),

.rstb(rstb),

.bit8(bit8),

.parity\_en(parity\_en),

.odd\_n\_even(odd\_n\_even),

.baud\_val(baud\_val),

.READ(READ),

.Rx(Rx),

.status(status),

.data(data)

);

//vary the clk signal every 10ns to mimick a

//period of 20ns (which is the period of our boards)

always #10 clk = ~clk;

//always #20 tx\_Write = ~tx\_Write;

initial begin

//Initialize Inputs

clk = 0;

rstb = 0; //low active reset

//8O1 (460800 Baud) Transmitting 0xAE = 1010\_1110

bit8 = 1; //8 bits of data

parity\_en = 1; //parity enabled

odd\_n\_even = 1; //odd parity

baud\_val = 4'hA; //baud = 460800

Rx = 1'b0; //data to be transmitted

READ = 2'b00;

//Wait 100 ns for global reset to finish

#100 @(posedge clk) rstb = 1; // have reset become unactive.

READ = 2'b10;

#20 @(posedge clk)

READ = 2'b00;

//Receive 0xAE

wait (uut.btu == 1);

wait (uut.btu == 0);

Rx = 1'b1;

//d0

wait (uut.btu == 1);

wait (uut.btu == 0);

Rx = 1'b0;

//d1

wait (uut.btu == 1);

wait (uut.btu == 0);

Rx = 1'b1;

//d2

wait (uut.btu == 1);

wait (uut.btu == 0);

Rx = 1'b1;

//d3

wait (uut.btu == 1);

wait (uut.btu == 0);

Rx = 1'b1;

//d4

wait (uut.btu == 1);

wait (uut.btu == 0);

Rx = 1'b0;

//d5

wait (uut.btu == 1);

wait (uut.btu == 0);

Rx = 1'b1;

//d6

wait (uut.btu == 1);

wait (uut.btu == 0);

Rx = 1'b0;

//d7 / Parity / Stop

wait (uut.btu == 1);

wait (uut.btu == 0);

Rx = 1'b1;

wait (uut.done == 1); //wait until byte is done being received.

#200

$stop;

end

endmodule

Transmit Engine Test Bench:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 3 - Transmit Engine plus PicoBlaze

// Course: CECS 460

//

// Create Date: 14:41:51 10/15/2015

//

// Module Name: transmit\_engine\_tb

// File Name: transmit\_engine\_tb.v

//

// Description: For this test bench I first initialized my Transmit Engine

// to be running at a 300 Baud Rate and at a 7N1 format. I then

// made sure that the correct value was getting loaded into my

// shift register and outputted the result of this comparison

// to the console. I then waited until the Done bit was set high,

// which signified that I was finished transmitting the 12 bits

// of data. I then loaded up the next byte of data, changed the

// baud rate to 1200, kept the 7N1 format, and repeated the

// process mentioned above. I made sure to transmit different

// combinations of data and to test out every format and

// baud rate available in my design. Once I verified that all

// of the data was being loaded and shifted out correctly at

// each of the different baud rates and data formats, I then

// concluded that my transmit engine was working correctly.

//

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module transmit\_engine\_tb;

//Inputs

reg clk;

reg rstb;

reg bit8;

reg parity\_en;

reg odd\_n\_even;

reg [3:0] baud\_val;

reg tx\_Write;

reg [7:0] in\_data;

//Outputs

wire tx; //transferLimitReached;

wire TxRdy;

//Instantiate the Unit Under Test (UUT)

//module transmit\_engine(clk, rstb, bit8, parity\_en, odd\_n\_even, baud\_val, txWrite,

//inData, tx, TxRdy);

transmit\_engine uut(

.clk(clk),

.rstb(rstb),

.bit8(bit8),

.parity\_en(parity\_en),

.odd\_n\_even(odd\_n\_even),

.baud\_val(baud\_val),

.txWrite(tx\_Write),

.inData(in\_data),

.tx(tx),

.TxRdy(TxRdy)

);

//vary the clk signal every 10ns to mimick a

//period of 20ns (which is the period of our boards)

always #10 clk = ~clk;

//always #20 tx\_Write = ~tx\_Write;

initial begin

//Initialize Inputs

clk = 0;

rstb = 0; //low active reset

//7N1 (300 Baud) Transmitting 0x65 = 110\_0101

bit8 = 0; //7 bits of data

parity\_en = 0; //parity disabled

odd\_n\_even = 0; //no parity

baud\_val = 4'h0; //baud = 300

in\_data = 8'h65; //data to be transmitted

//Wait 100 ns for global reset to finish

#100 @(posedge clk) rstb = 1; // have reset become unactive.

tx\_Write = 1; //assert tx\_Write after reset.

#20 @(posedge clk) tx\_Write = 0;

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'hf95) //7N1 tx -> 0x65 = f95

$display("Incorrect Data! Expected: 0xf95 , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0xf95 , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0); //wait until byte is done being transmitted.

//7N1 (1200 Baud) Transmitting 0x54 = 101\_0100

bit8 = 0; //7 bits of data

parity\_en = 0; //parity disabled

odd\_n\_even = 1; //no parity

baud\_val = 4'h1; //baud = 1200

in\_data = 8'h54; //data to be transmitted

tx\_Write = 1; //assert tx\_Write

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'hf51) //7N1 tx -> 0x54 = f51

$display("Incorrect Data! Expected: 0xf51 , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0xf51 , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0); //wait until byte is done being transmitted.

//7E1 (2400 Baud) Transmitting 0x72 = 111\_0010

bit8 = 0; //7 bits of data

parity\_en = 1; //parity enabled

odd\_n\_even = 0; //even parity

tx\_Write = 1; //assert tx\_Write

baud\_val = 4'h2; //baud = 2400

in\_data = 8'h72; //data to be transmitted

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'hdc9) //7E1 tx -> 0x72 = dc9

$display("Incorrect Data! Expected: 0xdc9 , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0xdc9 , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0); //wait until byte is done being transmitted.

//7O1 (4800 Baud) Transmitting 0x4F = 100\_1111

bit8 = 0; //7 bits of data

parity\_en = 0; //parity disabled

odd\_n\_even = 1; //no parity

tx\_Write = 1; //assert tx\_Write

baud\_val = 4'h3; //baud = 4800

in\_data = 8'h4F; //data to be transmitted

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'hf3d) //7O1 tx -> 0x4f = f3d

$display("Incorrect Data! Expected: 0xf3d , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0xf3d , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0);

//8N1 (9600 Baud) Transmitting 0x4F = 0100\_1111

bit8 = 1; //8 bits of data

parity\_en = 0; //parity enabled

odd\_n\_even = 0; //no parity

tx\_Write = 1; //assert tx\_Write

baud\_val = 4'h4; //baud = 9600

in\_data = 8'h4F; //data to be transmitted

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'hd3d) //8N1 tx -> 0x4f = d3d

$display("Incorrect Data! Expected: 0xd3d , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0xd3d , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0);

//8N1 (19200 Baud) Transmitting 0xAE = 1010\_1110

bit8 = 1; //8 bits of data

parity\_en = 0; //parity enabled

odd\_n\_even = 1; //no parity

tx\_Write = 1; //assert tx\_Write

baud\_val = 4'h5; //baud = 19200

in\_data = 8'hAE; //data to be transmitted

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'heb9) //8N1 tx -> 0xAE = eb9

$display("Incorrect Data! Expected: 0xeb9 , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0xeb9 , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0);

//8E1 (38400 Baud) Transmitting 0xBD = 1011\_1101

bit8 = 1; //8 bits of data

parity\_en = 1; //parity enabled

odd\_n\_even = 0; //even parity

tx\_Write = 1; //assert tx\_Write

baud\_val = 4'h6; //baud = 38400

in\_data = 8'hBD; //data to be transmitted

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'haf5) //8E1 tx -> 0xBD = af5

$display("Incorrect Data! Expected: 0xaf5 , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0xaf5 , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0);

//8O1 (57600 Baud) Transmitting 0x4F = 0000\_1000

bit8 = 1; //8 bits of data

parity\_en = 1; //parity enabled

odd\_n\_even = 1; //odd parity

tx\_Write = 1; //assert tx\_Write

baud\_val = 4'h7; //baud = 57600

in\_data = 8'h08; //data to be transmitted

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'h821) //8O1 tx -> 0x08 = 821

$display("Incorrect Data! Expected: 0x821 , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0x821 , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0);

//8O1 (115200 Baud) Transmitting 0x4F = 0100\_1111

bit8 = 1; //8 bits of data

parity\_en = 1; //parity enabled

odd\_n\_even = 1; //odd parity

tx\_Write = 1; //assert tx\_Write

baud\_val = 4'h8; //baud = 115200

in\_data = 8'h4F; //data to be transmitted

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'h93d) //8O1 tx -> 0x4F = 93d

$display("Incorrect Data! Expected: 0x93d , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0x93d , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0);

//8O1 (230400 Baud) Transmitting 0x00 = 0000\_0000

bit8 = 1; //8 bits of data

parity\_en = 1; //parity enabled

odd\_n\_even = 1; //odd parity

tx\_Write = 1; //assert tx\_Write

baud\_val = 4'h9; //baud = 230400

in\_data = 8'h00; //data to be transmitted

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'hc01) //8O1 tx -> 0x00 = c01

$display("Incorrect Data! Expected: 0xc01 , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0xc01 , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0);

//8O1 (460800 Baud) Transmitting 0x55 = 0101\_0101

bit8 = 1; //8 bits of data

parity\_en = 1; //parity enabled

odd\_n\_even = 1; //odd parity

tx\_Write = 1; //assert tx\_Write

baud\_val = 4'hA; //baud = 460800

in\_data = 8'h55; //data to be transmitted

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'hd55) //8O1 tx -> 0x55 = d55

$display("Incorrect Data! Expected: 0xd55 , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0xd55 , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0);

//8O1 (921600 Baud) Transmitting 0x4F = 0010\_0000

bit8 = 1; //8 bits of data

parity\_en = 1; //parity enabled

odd\_n\_even = 1; //odd parity

tx\_Write = 1; //assert tx\_Write

baud\_val = 4'hB; //baud = 921600

in\_data = 8'h20; //data to be transmitted

#20 @(posedge clk) tx\_Write = 0; //deassert tx\_Write

@(posedge clk) if(uut.write1Delay == 1)begin

#20; //wait for next clock cycle (when data is updated)

if (uut.sreg\_data != 12'h881) //8O1 tx -> 0x20 = 881

$display("Incorrect Data! Expected: 0x881 , Actual: 0x%h",

uut.sreg\_data[11:0]);

else

$display("Correct Data! Expected: 0x881 , Actual: 0x%h",

uut.sreg\_data[11:0]);

end

wait (uut.Done == 1); //wait until byte is done being transmitted.

wait (uut.TxRdy == 1); //wait until transmit engine is ready

//to receive another byte.

wait (uut.Done == 0);

$stop;

end

endmodule