Spartan Hero IP Core

Specification Document

Engineered by:

Raul Diaz

California State University, Long Beach
CECS 460: System on Chip Design

Table of Contents

1. Introduction

The *Hero* is a compact, capable, and fast system on chip optimized for the Spartan 3E FPGA. Implementing a fully functional UART core, the *Hero* SoC is designed to communicate with the embedded micron memory located on the Nexys2 board. The chip specification provided by this document demonstrates the mechanics and implementation of the *Hero*. The top level design of this chip instantiates a core design and TSI block

2. Applicable Documents

2.1 Applicable External Documents

- ***PicoBlaze 8-bit embedded processor***:
  - The PicoBlaze™ embedded microcontroller is an efficient, cost-effective embedded processor core for Spartan®-3, Virtex®-II, and Virtex-II Pro FPGAs. This user guide describes the capabilities, features, and benefits of PicoBlaze hardware design and how to effectively use the PicoBlaze instruction set and tools to create software applications.

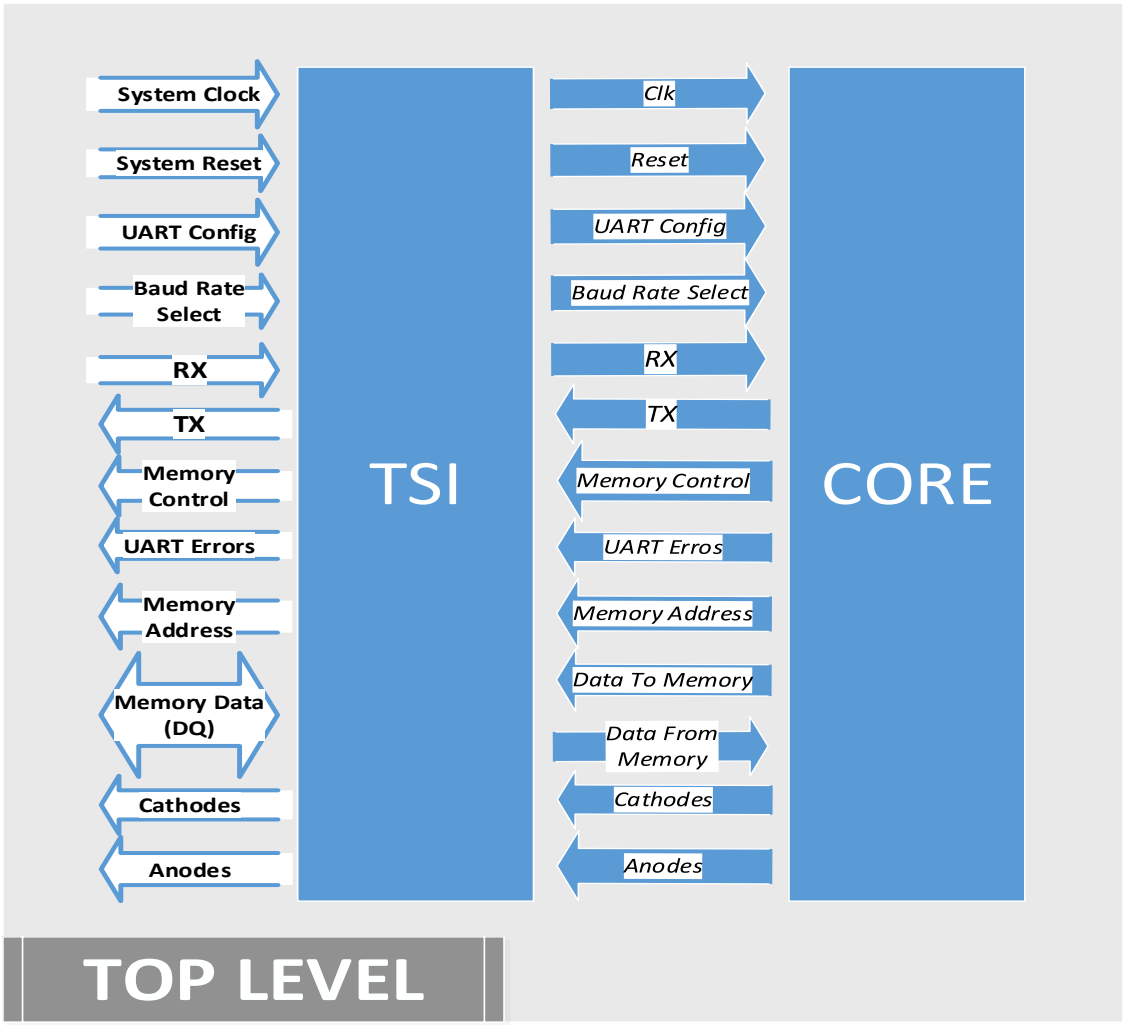
- ***Micron® CellularRAM™ (8M x 16)*** :
  - Micron® CellularRAM™ is a high-speed, CMOS pseudo-static random access memory developed for low-power, portable applications. The T45W8MW16BGX device has a 128Mb DRAM core, organized as 8 Meg x 16 bits. These devices include an industrystandard burst mode Flash interface that dramatically increases read/write bandwidth compared with other low-power SRAM or pseudo-SRAM offerings.

3. Top Level Design

3.1 Description

The Top level module implements the Core design and the TSI Block which are used to communicate with an exterior interface. For the purposes of this project, the micron memory located on the Nexys2 board will be used. It is the functionality of the TSI block to handle the I/O used by the core. The TSI block can be modified to handle various electrical characteristics which may be presented in a real world application. However for the sake of simplicity, the I/O ports of the top level design are assumed to handle the standard electrical inputs and distribute the standard electrical output. If one may choose to alter the electrical I/O, one may do so in the system's constraints file.

3.2 Block Diagram



3.3 Data Flow Description

Inputs to the system are received from the Nexys2 board and from an external device implementing an RS232 protocol. Inputs are processed in the TSI Block to ensure synchronization of the system before entering the Core Cell. The inputs from the Nexys2 control the UART configuration and the baud rate. (*Refer to the baud rate table for detailed selection*). The UART Configuration is as follows:

Dip Switch 2	Dip Switch 1	Dip Switch 0
8 bit transfer enable	Parity enable	Odd parity detection enable

The UART configuration is set upon reset and will not change as the system is running. The core handles the logic flow for the system and distributes processed data to the TSI to be outputted to the system.

3.4 I/O

Signal Name	Pin Assignment	Electrical Characteristic	Direction
SYS_CLK	B8	IOSTANDARD	INPUT
SYS_RST	H13	IOSTANDARD	INPUT
RX_IN	U6	IOSTANDARD	INPUT
UART_CONFIG[2]	K18	IOSTANDARD	INPUT
UART_CONFIG[1]	H18	IOSTANDARD	INPUT
UART_CONFIG[0]	G18	IOSTANDARD	INPUT
BAUD_SEL[3]	R17	IOSTANDARD	INPUT
BAUD_SEL[2]	N17	IOSTANDARD	INPUT
BAUD_SEL[1]	L13	IOSTANDARD	INPUT
BAUD_SEL[0]	L14	IOSTANDARD	INPUT
TX_OUT	P9	IOSTANDARD	OUTPUT
CE_	R6	IOSTANDARD	OUTPUT
OE_	T2	IOSTANDARD	OUTPUT
WE_	N7	IOSTANDARD	OUTPUT
ADV_	J4	IOSTANDARD	OUTPUT
CRE	P7	IOSTANDARD	OUTPUT
UB_	K4	IOSTANDARD	OUTPUT
LB_	K5	IOSTANDARD	OUTPUT
OCLKMEM	H5	IOSTANDARD	OUTPUT
UART_ERROR[2]	R4	IOSTANDARD	OUTPUT
UART_ERROR[1]	F4	IOSTANDARD	OUTPUT
UART_ERROR[0]	P15	IOSTANDARD	OUTPUT
MEM_ADDR [22:0]	J1 … K6*	IOSTANDARD	OUTPUT
DQ [15:0]	L1 … T1*	IOSTANDARD	INOUT

**Further Details can be found in Appendix Source Code*

3.5 Clocks

The system clock provided by the FPGA and is running at 50 MHz

3.6 Resets

The system reset is an active low assertion of the onboard button 3. The core implements an active high reset by instantiating the AISO module (Stabilizer) for a synchronous signal throughout the system.

3.7 Software

The software for the system was written in assembly and compiled using the Xilinx kcpsm3 assembler. The assembler produced a Verilog HDL file which was instantiated in the Core of the design. The program flow is to store received ASCII bytes from an exterior RS232 transmitter into memory. The program contained particular 'special keys' which would perform subroutines recognized by the user (i.e. constructive delete, new line feed, memory dump).

Interfacing with the *Hero* SoC can be done using a serial communication terminal. The source code for the memory.psm file can be found in the appendix of this document.

4. Externally Developed Blocks

4.1 Embedded PicoBlaze Processor

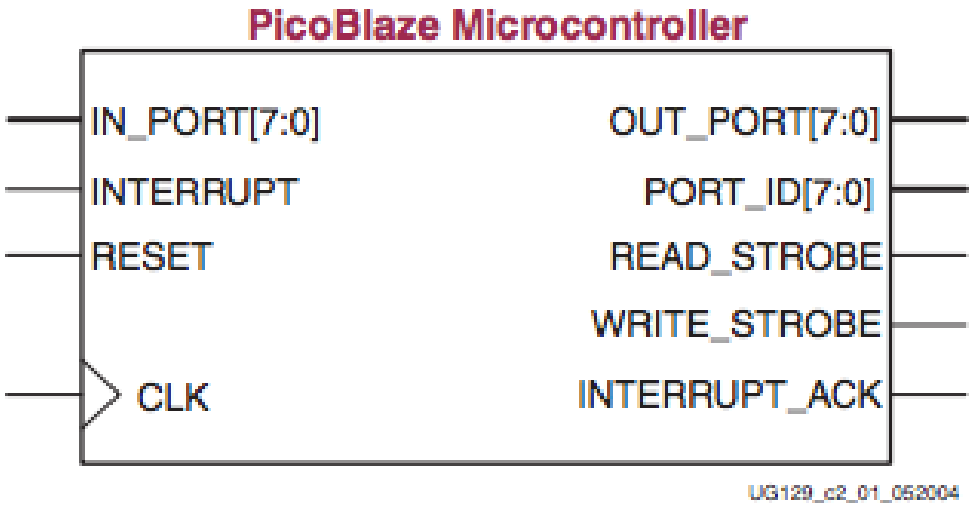
4.1.1 Description

The Picoblaze processor is responsible for dataflow between the UART module and the Memory interface Block (MIB). It is capable of reading status throughout the system and handles logic flow based on received signals. Control signals for the UART and MIB are managed through read and write strobes set by the Picoblaze along with their respective port ID, which is determined in the software flow.

The PicoBlaze processor was programmed in assembly and converted to Verilog HDL using the kcpsm3 assembler provided by Xilinx. In order to interface with PicoBlaze, a communication terminal would be required. The logic flow is as follows:

The program begins by displaying a welcome banner 'CSULB CECS 460' followed by the "~$" prompt on the next line, which emulates the Linux terminal prompt for a welcoming feeling. An external device implementing the RS232 protocol and proper configuration would be able to echo their transmitted ASCII character on the communication terminal. As ASCII characters are being received, the Picoblaze is using the memory interface block (MIB) to store each character into the micron memory located on the Nexys2. If an asterisk is received '*' the PicoBlaze performs a memory dump on the terminal, displaying a record of the received data.

4.1.2 Block Diagram



4.1.3 I/O

Pin Name	Wire	Direction	Description
Clk	Clk	INPUT	System Clock to PB
Reset	sRst	INPUT	Synchronous Reset
In_Port[7:0]	port_in_data	INPUT	Data to PicoBlaze
Out_Port[7:0]	port_out_data	OUTPUT	Data from PicoBlaze
Port_ID[7:0]	port_ID	OUTPUT	Up to 256 identifiable ports for input or output
Read_Strobe	rd_st	OUTPUT	Signals PB is currently reading Input
Write_Strobe	wr_st	OUTPUT	Signals PB is currently outputting

4.1.4 Memory Map

Port ID	Operation
0	Read UART Status
1	Read UART Data
11 (0x0B)	Write to Address Register 0
12 (0x0C)	Write to Address Register 1
13 (0x0D)	Write to Address Register 2
14 (0x0E)	Write to Data Register 0
15 (0x0F)	Write to Data Register 1
16 (0x10)	Read from Read Buffer Register 0
17 (0x11)	Read from Read Buffer Register 1
18 (0x12)	Perform Memory Read
19 (0x13)	Perform Memory Write
20 (0x14)	Read MIB Status

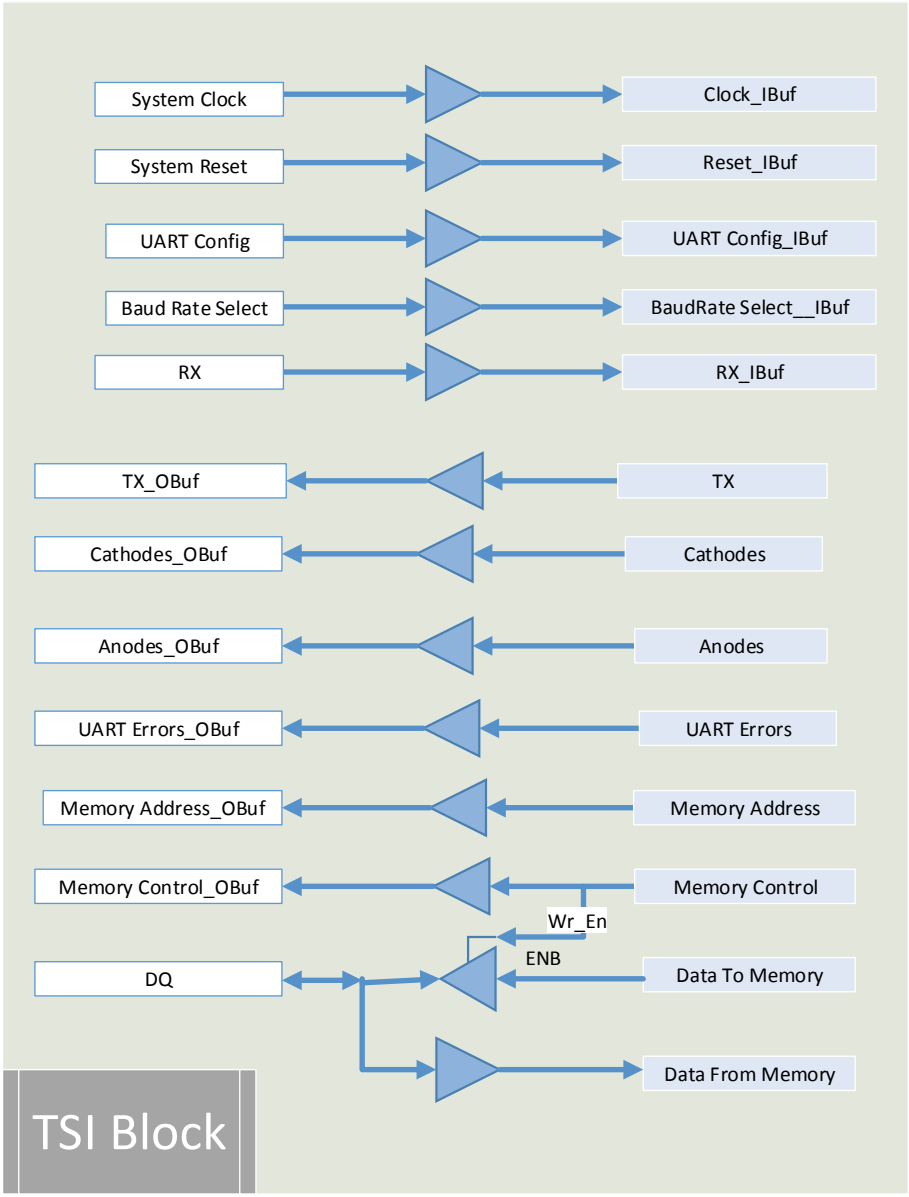
5. Internally Developed Blocks

5.1 Technology Specific Instance (TSI Block)

5.1.1 Description:

The technology specific instance block allows for a real world application to be applied to the SoC. By instantiating the I/O buffers provided by the Spartan 3E HDL library, the TSI ensures a synchronized design to any exterior application. The TSI also adds portability by providing flexible electrical to specification to the cores I/O ports. This can be done in the systems constrains file.

5.1.2 Block Diagram



5.1.3 I/O

I/O Name	Direction	Description
Sys_Clk	Input	System Clock
Sys_Rst	Input	System Reset
Tx_data	Input	Transmit Data from Core
Cs	Input	Chip Select from Core
{Rd_mem, wr_mem, adv_mib, cre_mib, upperbyte_en, lowbyte_en }	Input	Read Enable from Core
data_to_mem [15:0]	Input	Data to Memory From Core
addr_to_mem[22:0]	Input	Address to Memory From Core
UART_error[2:0]	Input	UART Error From Core
Anodes[3:0]	Input	Anodes From Core
Cathodes[6:0]	Input	Cathodes From Core
Rx_buf	Input	Received Data from exterior Device
UART_Confg_buf[2:0]	Input	UART Configuration from exterior Device
baud_sel_buf[3:0]	Input	Baud Rate Select from exterior device
CLKMEM	Input	Memory Clock from Core (Always Low)
Rx_in	Output	Received Data to Core
data_from_mem[15:0]	Output	Data from Memory To Core
UART_Config[2:0]	Output	UART Configuration To Core
baud_sel[3:0]	Output	Baud Rate Select To Core
Clk_buf	Output	Clock to Core
Rst_buf	Output	Reset To Core
Tx_buf	Output	Transmit Data To exterior device
{CE_,OE_, WE_, ADV_, CRE, UB_, LB_}	Output	Memory Configuration To Micron Memory
Addr_buf[22:0]	Output	Address to Micron Memory
UART_error_buf[2:0]	Output	UART Errors to Exterior Device
anodes_buf[3:0]	Output	Anodes to Exterior Device
cathodes_buf	Output	Cathodes to exterior device
OCLKMEM	Output	Clock to Micron Memory (Always Low)
DQ	InOut	Bidirectional Data Bus to and from Micron Memory

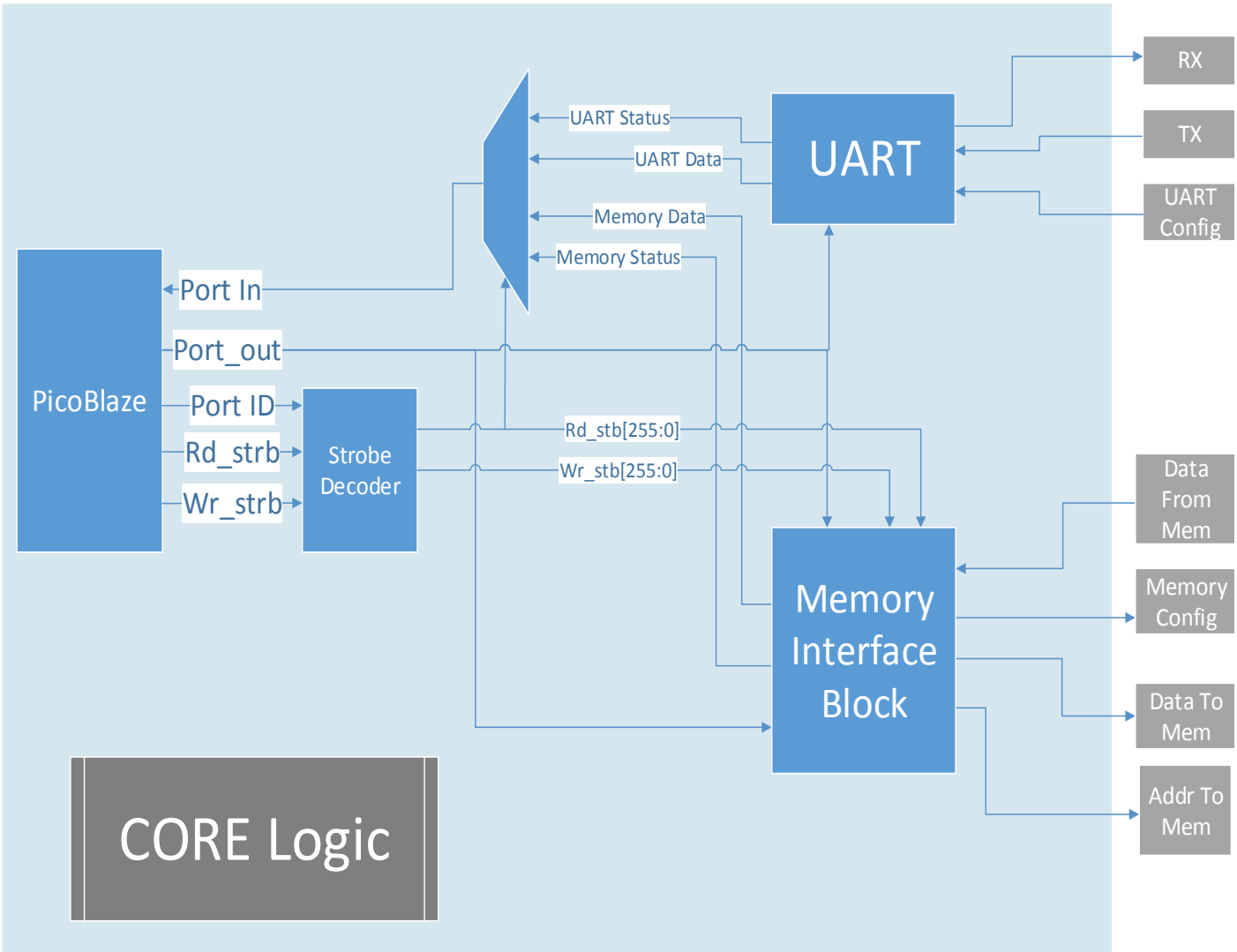
5.2 Core Logic Cell

5.2.1 Description:

The Core cell contains the logic flow of the SoC. By instantiating the embedded Picoblaze processer, UART, and the Memory Interface block, the Core is capable of providing system communication to an exterior device.

A 4:1 selection case statement multiplexes the data going into the PicoBlaze processor. The read strobes set within the decoder module determine the selection of the multiplexer, which are set by software logic flow within the PicoBlaze. The mux provides the PicoBlaze the option of reading received data from the UART, the status of the UART, the data read from the embedded micron memory or the status of the MIB module (Ready Flag).

5.2.2 Block Diagram:



5.2.3 I/O:

I/O Name	Direction	Description
Clk	Input	System Clock
Rst	Input	System Reset (Active Low)
parity_en	Input	Parity Enable
odd_en	Input	Odd Parity Detection
bit8_en	Input	8 bit Transfer Enable
Rx_in	Input	Received Bit
baud_sel[3:0]	Input	Baud Rate Selection
data_from_mem[15:0]	Input	Data Read From Memory
data_to_mem [15:0]	Output	Data to Write in Memory
addr_to_mem [22:0]	Output	Address Memory
Tx_data	Output	Transfer Data Bit
Cathodes[6:0] {a,b,c,d,e,f,g}	Output	Cathodes for 7-segment
Anodes[3:0] {a3,a2.a1.a0}	Output	Anodes for 7-segment
Flagreg[2:0] {frm_err,ov_err,p_err}	Output	Flag Register (Framing Error, Overflow Error, Parity Error)
{ cs, rd_mem ,wr_mem ,adv_mib, cre_mib ,upperbyte_en, lowbyte_en }	Output	Memory Configuration {CE_,OE_,WE_,ADV_,CRE_,UE,LE}

5.2.4 Register Map

Register Name	Width	Description
Flagreg	3 bit	Holds UART Error Flags
Port_in_data	8 bit	Data from mux to PicoBlaze
Rxdata / txdata	8 bit	Received & Transmitted data to 7-segment display.

5.2.5 Status Registers

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit1	Bit 0
UART Status	*<0>*	*Framing Error*	*Overflow Error*	*Parity Error*	*<0>*	*<0>*	*Tx Ready*	*Rx Ready*
MIB Status	*<0>*	*<0>*	*<0>*	*<0>*	*<0>*	*<0>*	*<0>*	*MIB Ready*

5.2.6 Internal Modules:

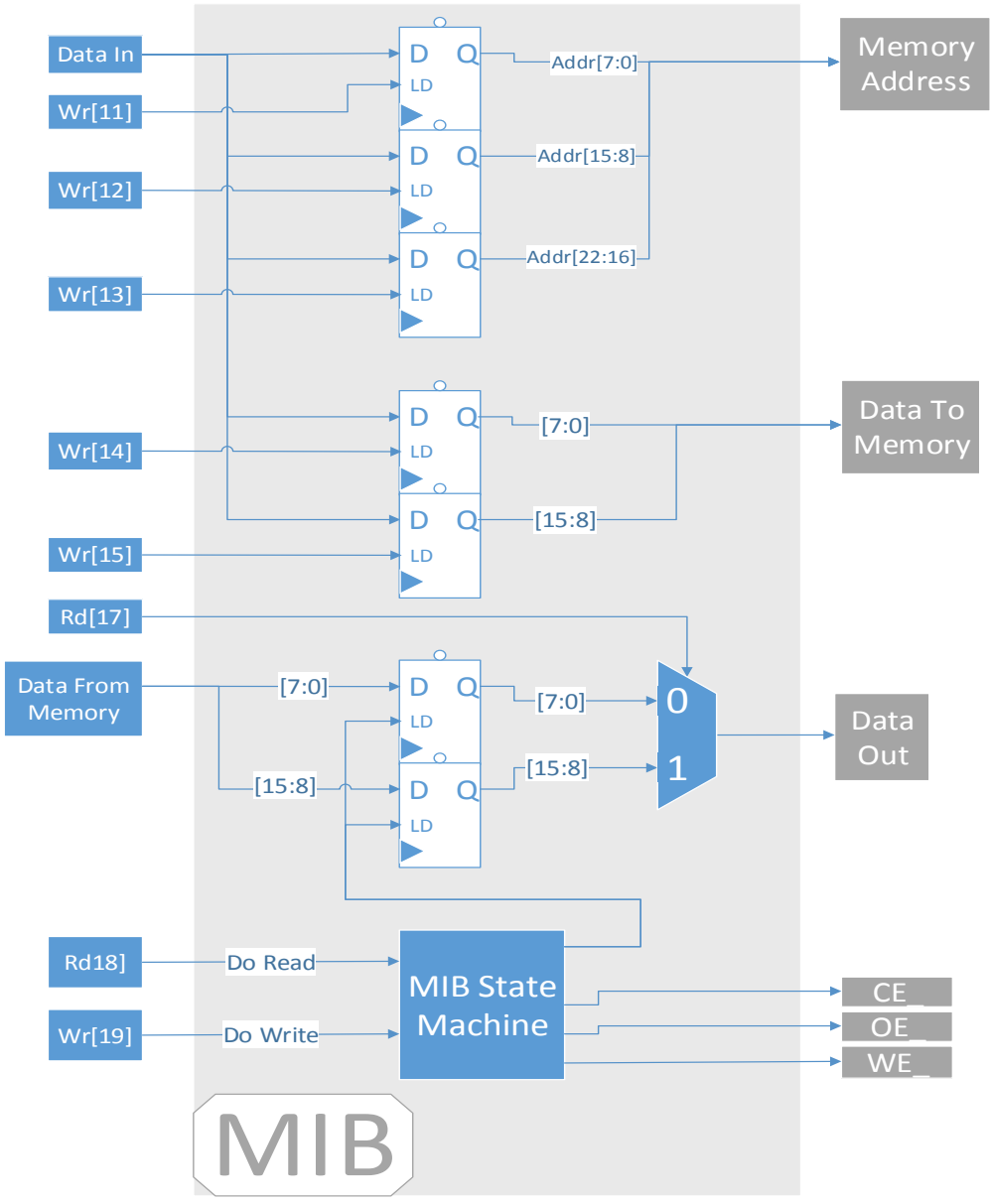
See sections 5.3 – 5.8

5.3 Memory Interface Block (MIB)

5.3.1 Description:

The Memory Interface Block provides the communication link between the PicoBlaze processor and the exterior memory interface. The design of the MIB is constructed around the MIB State Machine which provides the necessary load signals to the 2 read buffer register holding data from memory. The state machine also provides read and write enable (OE_ & WE_) signals to the memory chip based on the respective state the PicoBlaze may be in.

5.3.2 Block Diagram



5.3.3 I/O

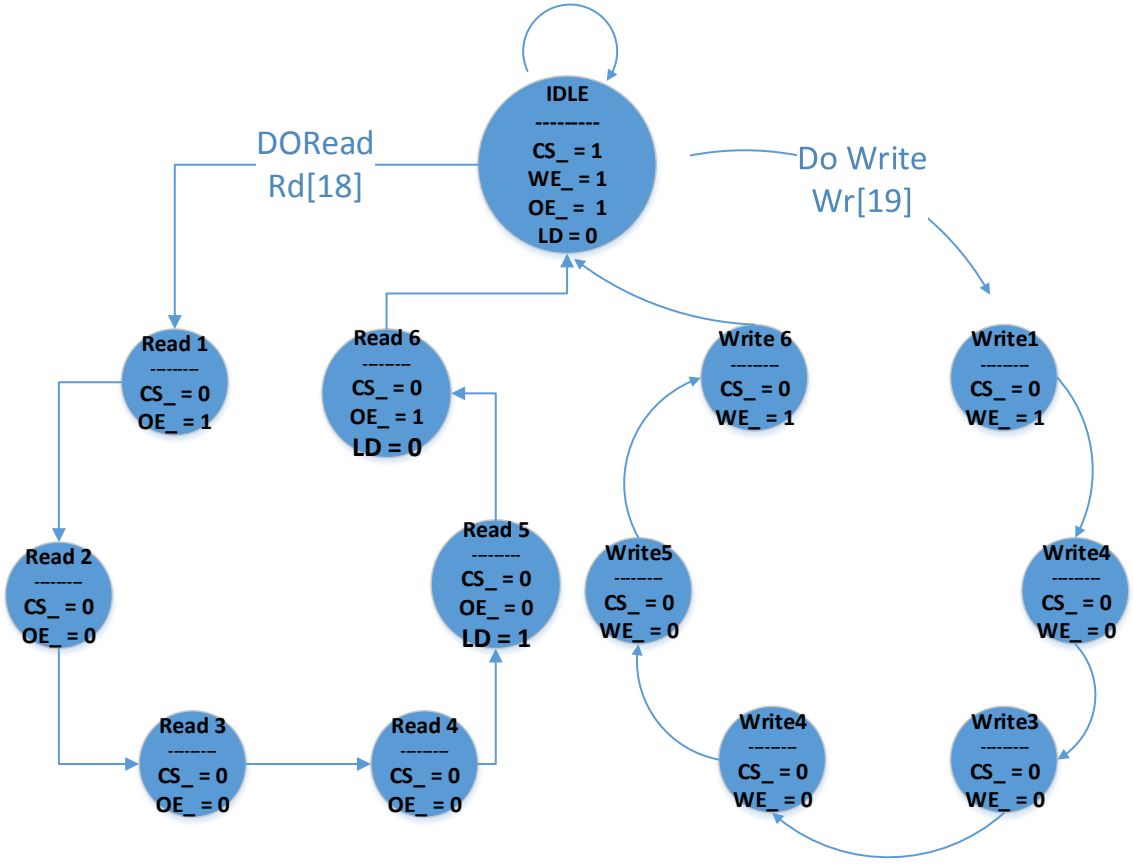
I/O Pin	Direction	Description
Clk	Input	System Clock
Rst	Input	System Reset (Active low)
datain[7:0]	Input	Data to be written to write buffer registers or address buffer registers based on respective wr_strb
data_from_mem [15:0]	Input	Data to be loaded into 2 8-bit read buffer registers
wr_strb[255:0]	Input	Used to signal which P.B port is being used to write
rd_strb[255:0]	Input	Used to signal which P.B port is being used to read
CE_	Output	Chip Select for micron memory
OE_	Output	Output Enable(Read En) for micron memory
WE_	Output	Write Enable for micron memory
ADV_	Output	Always Low (Unused)
CRE	Output	Always Low (Unused)
UB_	Output	Always Low (Unused)
LB_	Output	Always Low (Unused)
dataout[7:0]	Output	Data to be read from particular read buffer register based on respective rd_strb
status[7:0]	Output	Only LSB is used to signal MIB ready for next Read memory transaction
data_to_mem[15:0]	Output	Data outputted using 2 8-bit write register for memory write.
addr_to_mem[22:0]	Output	Data outputted using 3 8-bit address register for memory address.

5.3.4 MIB State Machine

5.3.4.1 Description

The MIB Finite State Machine is an essential component to MIB. It controls the Chip Select, Read, and Write enable strobes used to activate the micron memory chip. Upon reset, this FSM remains in idle waiting for rdstrobe[18] or wrstrobe[19] from the system. Based on received strobe, the FSM goes into an uninterrupted 6 state transition changing on each rising edge of the system clock. Details on communication can be found in the Micron Memory Datasheet.

5.3.4.2 State Machine Diagram



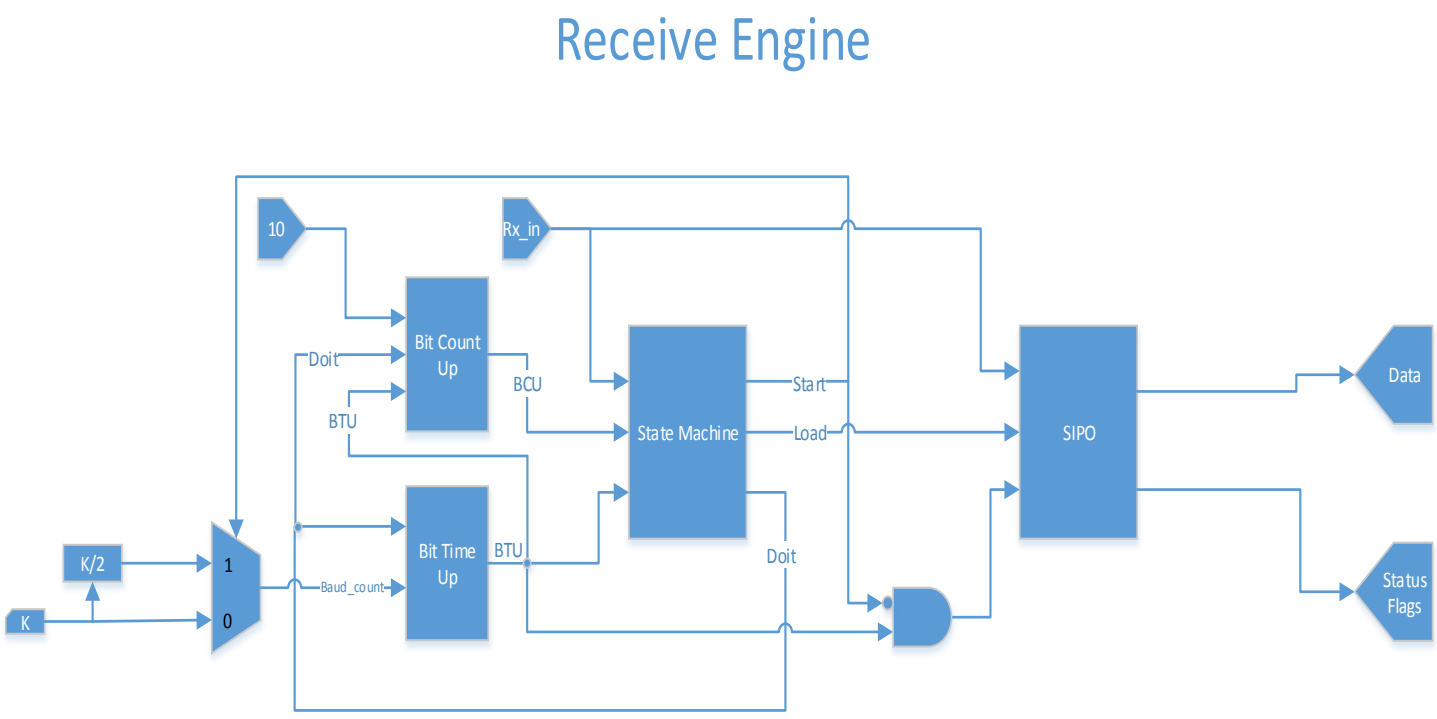
5.4 Receive Engine (Rx)

5.4.1 Description:

The Receive Engine is a synchronous communications module capable of receiving serial data using an RS232 protocol. It outputs an 8 bit data to an exterior device and is capable of producing 4 status signals Rx Ready, Parity Error, Overflow Error and a Framing Error.

There are 4 main instantiations involved: Bitcountup.v, Bit_time_counter.v, Recieve_SM, and the SIPO Shift Register. An 18 bit Baud Count value is also produced based on the start signal from the FSM. This signal will determine if the Baud Count is divided by 2, in order to shift the trigger mark to the midpoint of a serial transfer. A shift signal is also produced upon a low start signal from the FSM and a BTU signal to determine a shift in the SIPO.

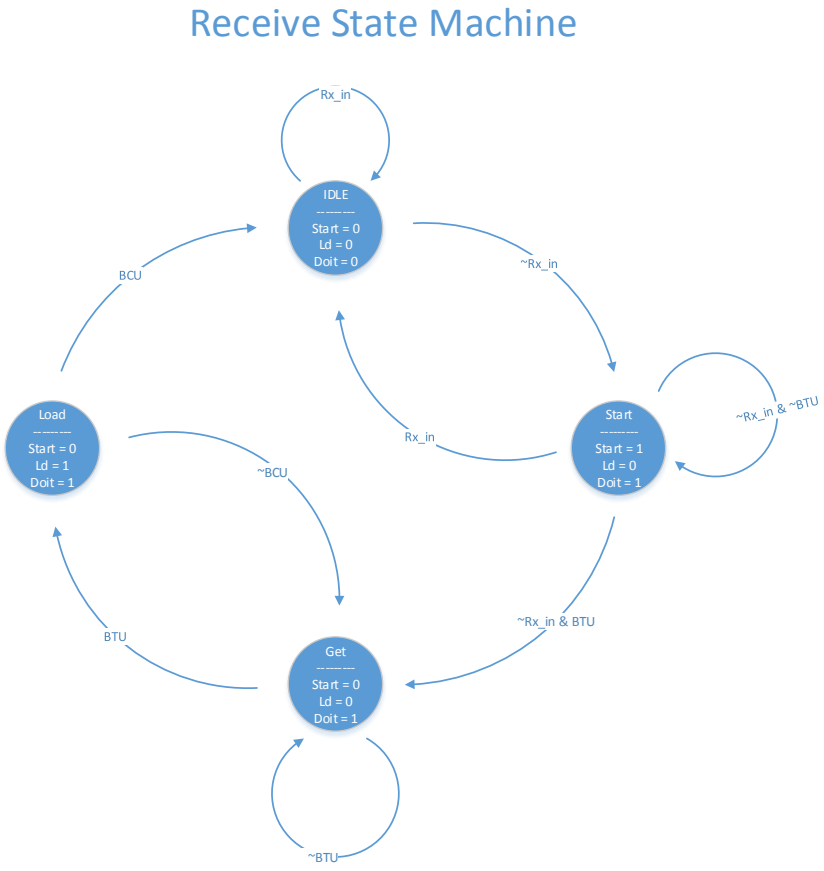
5.4.2 Block Diagram:



5.4.3 I/O:

I/O Pin	Direction	Description
Clk	Input	System Clock
Rst	Input	System Reset (Active low)
Rx_in	Input	Received Serial Bit
parity_en	Input	Parity Enable Setting
bit8_en	Input	8 bit Data Transfer Enable
odd_en	Input	Odd Parity Detection
rd_strb	Input	Begins Receive System Detection
Baud_val[17:0]	Input	18 bit Count Value for Baudrate timing
Rx_rdy	Output	Signals Receive Engine Ready
p_err	Output	Signals Parity Error Detected
ov_err	Output	Signals Overflow Error Detected
frm_err	Output	Signals Framing Error Detected
Rx_out[7:0]	Output	8 bit Received data

5.4.4 State Machine:

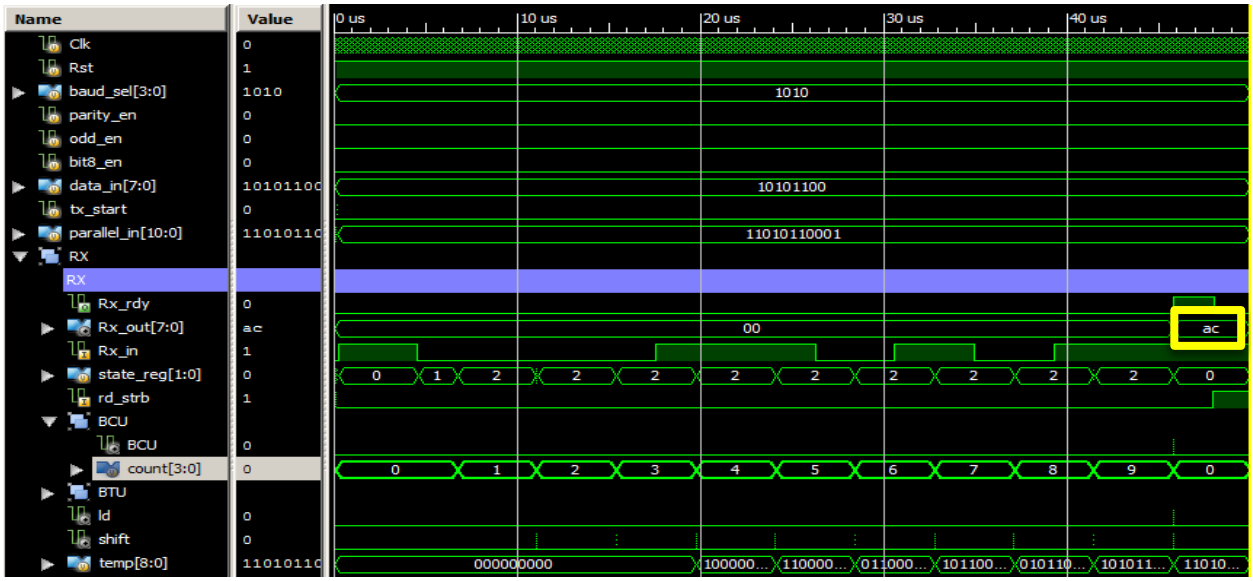


5.4.5 Verification:

Varification for the RX Engine was performed on the UART Top Level Design as shown:

**Note the Consistency of the Rx_in Wave form vs the predetermined data_in value*.

```
46    always
47    #10 Clk = ~Clk ;
48
49    initial begin
50       // Initialize Inputs
51       Clk = 0;
52       Rst = 0;
53       baud_sel = 4'hA;
54       parity_en = 0;
55       odd_en = 0;
56       bit8_en = 0;
57       data_in = 8'hAC;
58       tx_start = 0;
59
60       // Wait 100 ns for global reset to finish
61       #100;
62       @ (posedge Clk )
63          Rst = 1;
64
65       // Add stimulus here
66
67       #100;
68       @ (posedge Clk )
69          tx_start = 1;
70       #100;
71       @ (posedge Clk )
72          tx_start = 0;
73
74    end
75
76 endmodule
```



5.4.6 Internal Modules

5.4.6.1 Bit Count Up

The BitCountUp module determines the number of transferred bits during an operation. The done flag will signal when 'bitCount' bits have successfully transferred based on the BTU(bit time up) flag from the system. On Reset, count is set to bitCount in order to signal a done to the system.

5.4.6.2 Bit Time up Counter:

The Bit time counter module generates a single clock pulse to signal the end of a bit time transfer. A bit time for the system is determined by the baud_count wire, an 18 bit wide bus. The Baud rate table determines the count used in the procedural assignment logic.

5.4.6.3 Receive State Machine:

This synchronous State Machine provides the Receive Engine the necessary signals for receiving and processing particular bits based on the current state of the RS232 protocol transfer.

There are 4 states exercised at all times during execution:

0. The Idle state will wait for the start bit during an RS232 protocol, it ensures grounded signals from all counters used in the RX Engine. Upon a low signal from the Rx data input, the State will transition to the Start state for further processing.
1. At the Start State, the system is processing the start bit from an exterior device and will set the trigger for shifting and storing values at the middle of a bit transfer from the perspective of the exterior device. (I.e. sampling is performed at the middle of a bit transfer to ensure proper processing of a bit, this will reduce timing errors due to baud rates of other devices). The Start state also begins all counters used in the RX engine. When a BTU signal is received, the system will transition into the Get state.
2. The Get state provides the necessary signals used to store the incoming bits in the RX engine. It will transition into the load state on a high BTU signal.
3. At the Load state, the system is checking for the BCU signal which signals the maximum allowable bits for in transfer has occurred. At which point, the FSM will provide a load signal to the system to allow access to the received data. The state will then transition into the idle state and wait for the next start bit. However if a low BCU signal is read, the state will reseed to the get state for further processing.

5.4.6.4 SIPO Shift Register:

The synchronous 'Serial in Parallel out' Shift Register implemented in this design is used for the RX engine. This module also provides necessary status flags to the system for continuous processing. On a shift signal from the system the SIPO will shift new bits to the right. On a load & Rx Ready signal, the SIPO will update the data out register with the received bits stored in the temp register.

This SIPO module is constructed for the use of an RS232 protocol and provides error signals upon received data vs computed data such as a Framing Error, Parity Error and Overflow error. This module also provides logic for an Rx Ready signal which determines if the RX engine is ready to receive a sequence of information. Upon a load signal, the last bit (stop bit) is excluded from the *data out* register. Depending on the UART configuration, an optional 8th bit is loaded along with all other received bits.

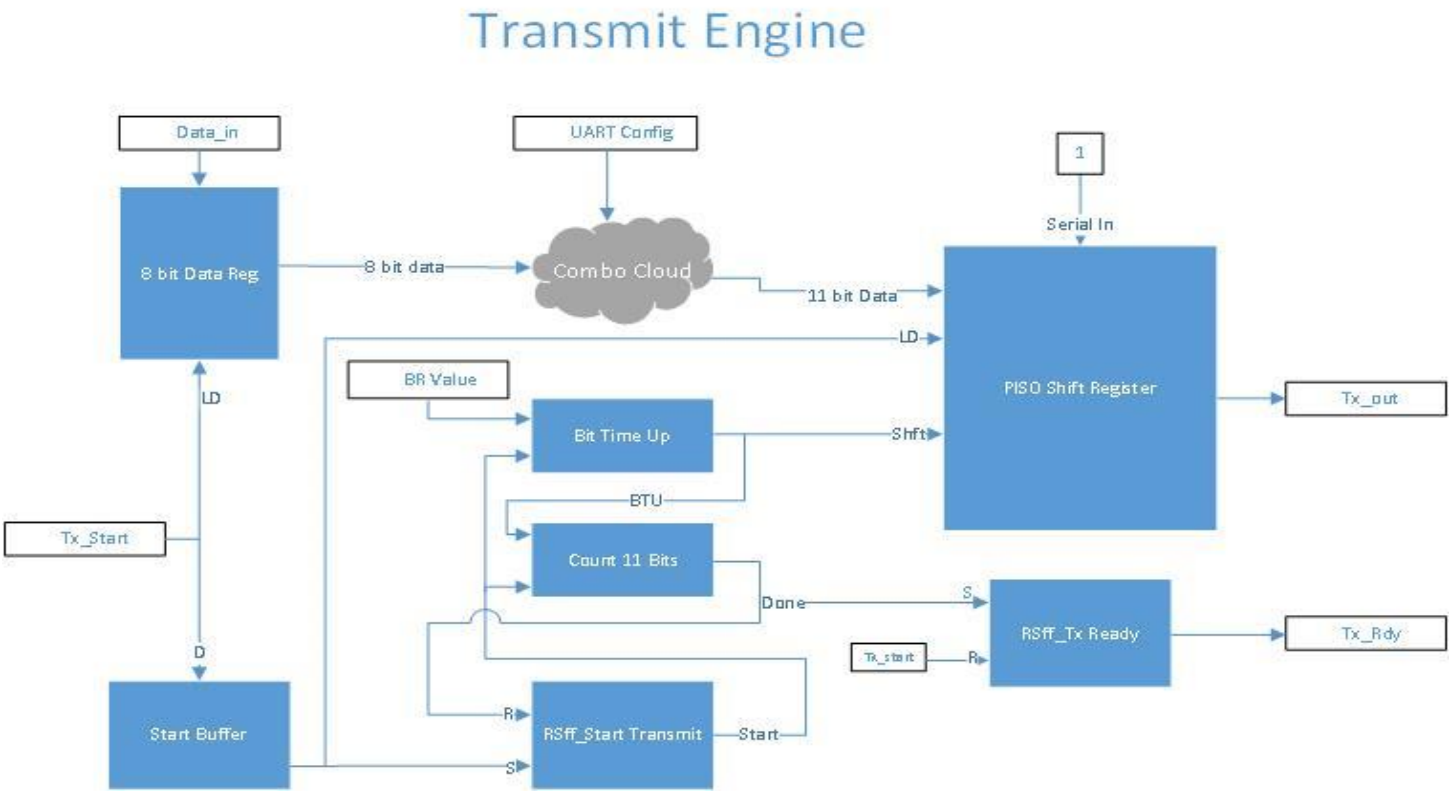
5.5 Transmit Engine (Tx):

5.5.1 Description:

The TxEngine module is an 8-bit synchronous loadable transmit engine. Operation is executed on a high signal from the tx_start port which triggers the load of an 8 bit register and signals 2 counters which handle baud rate timing and bit transfers for the system. The tx_out will transmit an 11 bit value using a 'Parallel In-Serial out' shift register module. The 11 bit value exercises an RS232 protocol which includes: 1 stop bit, 1 start bit, 8 bit data, and based on the UART Config register, a Parity bit. Trailing ones are followed after data has completed transmission.

The UART configuration is set one time upon reset and stored in a 3-bit register used in a procedural case statment. Bit8_en port will determine if an 8 bit value is transmitted, if the port is low by default the system will transmit 7 bits. parity_en port will determine if parity is detected on the input, and Odd_en port will check for an Odd parity or even parity if bit is set low. The baud rate timing detection is handled with an 18 bit counter value corresponding to a specific baud rate requested by the system. After the timing is complete on an 11 bit transfer, the tx_rdy port will output high to signal the Tx_engine is ready for another transfer.

5.5.2 Block Diagram



5.5.3 I/O

I/O Pin	Direction	Description
Clk	Input	System Clock
Rst	Input	System Reset (Active low)
parity_en	Input	Parity Enable Setting
bit8_en	Input	8 bit Data Transfer Enable
odd_en	Input	Odd Parity Detection
tx_start	Input	Begins Transmit System
data_in[7:0]	Input	8 bit Transmitted Data
Baud_val[17:0]	Input	18 bit Count Value for Baudrate timing
tx_out	Output	Serial Transmitted Data
tx_rdy	Output	Signals Ready for next Transmit Byte

5.5.4 Register Map

Register Name	Width	Description
pipo_buf	8 bits	Holds byte to be transmitted
ld_buf	1 bit	Gets set upon tx_start signal from input. Drives the load signal to the shift register
parallel_in	11 bits	Holds RS232 11 bit data transfer information

5.5.5 Verification

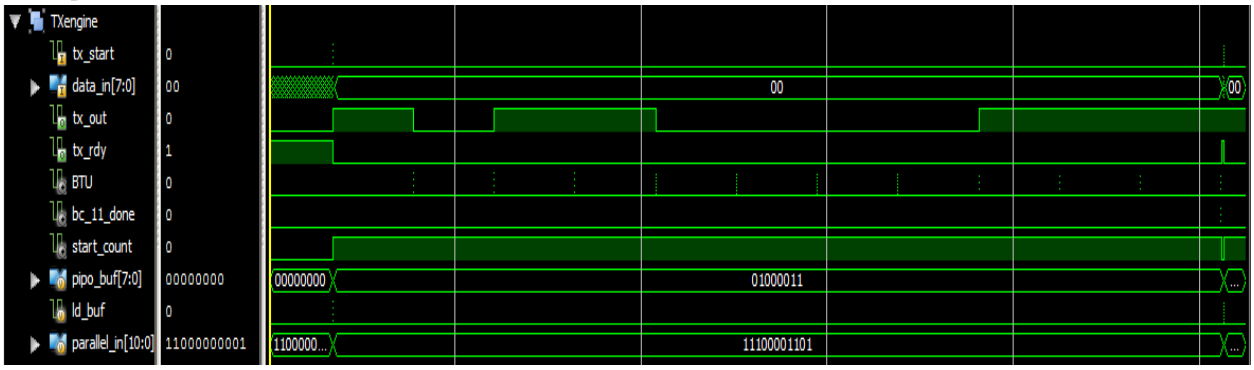
```
    // Instantiate the Unit Under Test (UUT)
    TopLevel uut (
        .Clk(Clk),
        .Rst(Rst),
        .baud_sel(baud_sel),
        .parity_en(parity_en),
        .odd_en(odd_en),
        .bit8_en(bit8_en),
        .Tx_data(Tx_data)
    );

    always
      #10 Clk = ~Clk;

    initial begin
        // Initialize Inputs
        Clk = 0;
        Rst = 1;
        baud_sel = 9;
        {parity_en, bit8_en, odd_en } = 3'b000;
        // Wait 100 ns for global reset to finish
        #100;
            @(negedge Clk)
                 Rst = 0;


    end

endmodule
```



5.5.6 Internal Modules

5.5.6.1 PISO Shift Register:

A Parallel IN Serial OUT synchronous shift register with active low reset used to transmit an 11 bit data bus from the system. Loads new data on a high load signal assertion. Starting with the LSB, data is shifted out on a shift enable signal, and the serial in data is shifted into the MSB data register.

5.5.6.2 11 Bit Counter:

A synchronous active low reset module, the count 11 bits module determines the number of transferred bits during an operation. The done flag will signal when 11 bits have successfully transferred based on the BTU (bit time up) flag from the system. On Reset, count is set to 11 in order to signal a done to the system.

5.5.6.3 Bit Time up Counter:

The Bit time counter module generates a single clock pulse to signal the end of a bit time transfer. A bit time for the system is determined by the baud_count wire, an 18 bit wide bus. The Baud rate table determines the count used in the procedural assignment logic.

5.5.6.4 RS FLOP

The RS Flop is synchronous module with active low reset used to configure the output Q based on 2 input signals.

The Transmit Engine instantiates the RS flop in 2 cases. In the Start transmission case, the flop would 'kick-start' the Transmit Engine by driving the load signal on the 8 bit data register. In the second case, the RS flop is used to drive the Tx ready signal to an external module to signal the next process.

5.6 Baudrate Decoder

5.6.1 Description

The Baudrae Value Decoder module will determine the magnitude of the count value used throughout the transmit engine. A 4 bit baude select value is recieved from the system and used to select its corresponding Count value as shown in the table below. The baud value is an 18 bit number and is set one time only on reset.

Baud Rate Table

Baud Rate Table			
Sel	Baud Rate	Bit Time = 1/BR	# CLKS
0	300	3.33E-03	166667
1	600	1.67E-03	83333
2	1200	833.3E-3	41667
3	2400	4.17E-04	20833
4	4800	2.08E-04	10417
5	9600	1.04E-04	5208
6	19200	5.21E-05	2604
7	38400	2.60E-05	1302
8	57600	1.74E-05	868
9	115200	8.68E-06	434
10	230400	4.34E-06	217
11	460800	2.17E-06	109
12	921600	1.09E-06	54

5.6.2 I/O:

I/O Pin	Direction	Description
Clk	Input	System Clock
Rst	Input	System Reset (Active low)
Baudsel[3:0]	Input	Holds Baudrate selection from dip switches
Baud_val[17:0]	Output	18 bit Count Value for Baudrate timing

5.7 Strobe Decoder:

5.7.1 Description

The decode module is used to interface with the Picoblaze processor's port id and rd/wr strobes. The decoder is used to select and activate particular modules used throughout this project. By decoding the port ID, the system can determine which module is currently INPUTTING or OUTPUTTING to the Picoblaze.

5.7.2 I/O

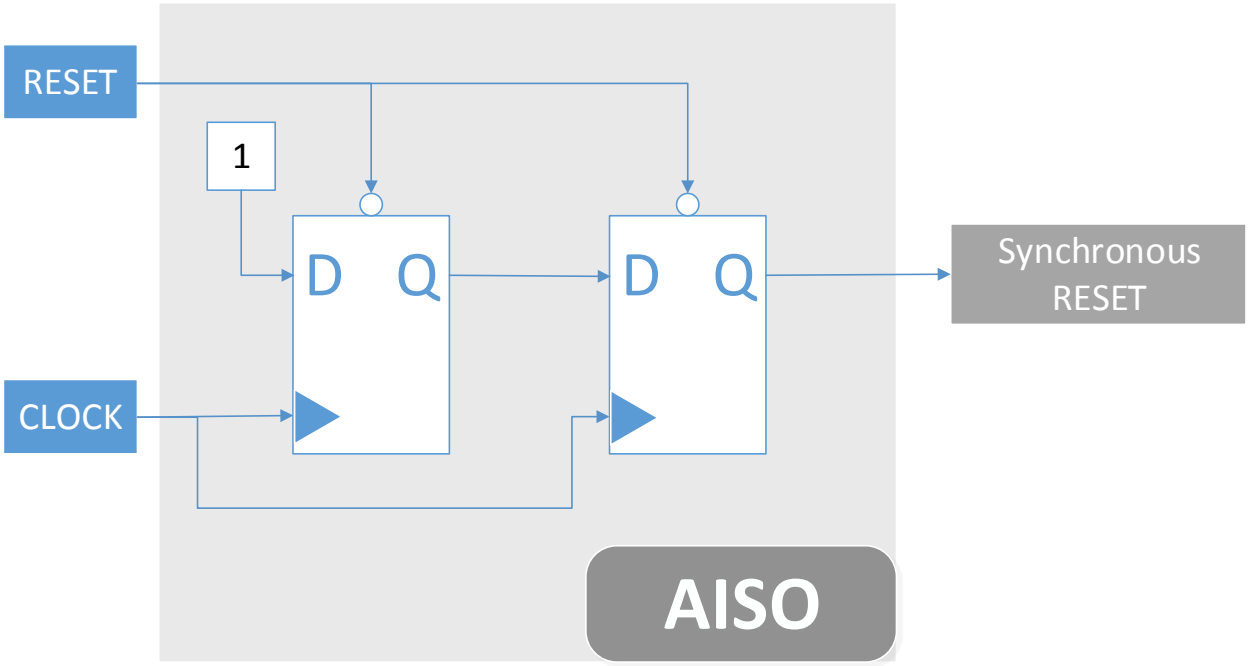
I/O Pin	Direction	Description
rd_in	Input	Read Signal from System
wr_in	Input	Write Signal from System
port_id [7:0]	Input	8 bit Identifiable port ID
rd_out[255:0]	Output	Holds corresponding Read bit from port id
wr_out[255:0]	Output	Holds corresponding write bit from port id

5.8 AISO (Asynchronous In Synchronous Out):

5.8.1 Description

The AISO module was used to turn an asynchronous signal into a synchronous signal. The synchronous signal would be used to create an active low reset for the circuit. This was achieved by the standard model demonstrated in a paper on Asynchronous & Synchronous Reset Design Techniques by Clifford E. Cummings, Don Mills, and Steve Golson which was presented in class. In short, the circuit would implement 2 flops which would remove metastability in the case of a setup time violation in the first ff due to a hardware mechanical bounce. The second flop would provide the synchronization of the flop after a state of metastability in the first flop.

5.8.2 Block Diagram



5.8.3 I/O

I/O Pin	Direction	Description
Clk	Input	System Clock
async_rst	Input	Asynchronous Reset from on board Button
sync_rst	Output	Synchronous Reset for System (Active Low)

6. Chip Level Verification / Test

Chip Level verification was done by observation testing of hardware through a host PC. Using Xilinx Adept software, the bitstream file was programmed to the FPGA using JTAG. Setting the according dip switches on the nexys2 board, baud rate values and UART configuration were tested under instructor supervision and were approved for submission.

Appendix

```
;**********************  CECS 460 Project 4: UART **********************
; ## Engineer: Raul Diaz
;## Course:   CECS460
;## Semester: Sp 15
;## Modified: 5/10/15
;;=============================================================================
;* File: memory.psm
;=============================================================================
;**********************   Constant Values  **********************
            CONSTANT dataBus        , 01
            CONSTANT statusReg      , 00
            CONSTANT wradrReg1      , 0B ; port 11
            CONSTANT wradrReg2      , 0C ; port 12
            CONSTANT wrdataReg0     , 0E ; port 14
            CONSTANT wrdataReg1     , 0F ; port 15
            CONSTANT rddataReg0     , 10 ; port 16
            CONSTANT rddataReg1     , 11 ; port 17
            CONSTANT memrd          , 12 ; port 18
            CONSTANT memwr          , 13 ; port 19
            CONSTANT rdMIBstat      , 14 ; port 20

            NAMEREG  s0             , scratchbyte   ; transmit/recieve byte register
            NAMEREG  s1             , checkstatus   ; Comparator register
            NAMEREG  s2             , linecounter   ; keeps track of characters on screen
            NAMEREG  sB             , addr0         ; memory pointer
            NAMEREG  sC             , addr1         ;keeps track of characters in memory
            NAMEREG  sD             , memCount      ; traverse through memory

;===============Initialize values and console output =============;
            LOAD     scratchbyte    , 00
            LOAD     checkstatus    , 00
            LOAD     linecounter    , 00
            LOAD     addr0          , 00
            LOAD     addr1          , 00
            LOAD     S4             , 00
            CALL     BANNER
;*****************************************************************
;                       Start of Algorithm
;*****************************************************************
START:
            COMPARE  linecounter    , 29
            JUMP     NZ             , RCVE
            CALL     NEWLINE

RCVE:
            CALL    RECEIVE
;BEGIN PROCESSING RECIEVED BYTE

            COMPARE scratchbyte     , 08    ;backspace
            JUMP    NZ              , NOTBS
            CALL    BACKSPACE
            JUMP    START
NOTBS:
```

```
            COMPARE scratchbyte     , 0D    ;CARRAGE RETURN
            JUMP    NZ              , NOTCR
            CALL    NEWLINE
            JUMP    START
NOTCR:
            COMPARE scratchbyte     ,2A     ;<*>
            JUMP    NZ              ,TRNSMIT
            JUMP    MEMDUMP
            JUMP    START
TRNSMIT:
            ADD     linecounter     ,01     ; incrmt linecounter
            CALL    TRANSMIT
            CALL    SAVEMEM
            JUMP    START
;================================;;================================;
SAVEMEM:
            OUTPUT  addr0           , wradrReg1     ; write ADDR reg 0
            OUTPUT  addr1           , wradrReg2     ; write ADDR reg 1
            OUTPUT  scratchbyte     , wrdataReg0    ; write DATA reg 0
            LOAD    scratchbyte     , 2A            ; <*>
            OUTPUT  scratchbyte     , wrdataReg1    ; write DATA reg 1
            OUTPUT  scratchbyte     , memwr     ; Perform memory write
            ADD     addr0           , 01        ; inc addr
            ADDCY   addr1           , 00
            RETURN
;================================;
MEMDUMP:
           COMPARE  addr0           , 00
           JUMP     Z               , ENDDUMP
           LOAD     memCount        , 00
STARTDUMP:
           OUTPUT   memCount        , wradrReg1
           INPUT    scratchbyte     , memrd         ; Perform memory rd
MIBRDY:
           INPUT    checkstatus     , rdMIBstat
           AND      checkstatus     , 01            ; MASK STATUS REGISTER FOR MIB Ready BIT
           JUMP     Z               , MIBRDY
           input    scratchbyte     , rddataReg0
           CALL     TRANSMIT
           ADD      memCount        , 01            ; INC MEMORY POINTER
           COMPARE  addr0           , memCount      ; IS MEMORY POINTER AT CURRENT ADDR
           JUMP     NZ              , STARTDUMP
           LOAD     addr0           , 00            ; RESET ADDRESS
ENDDUMP:
            JUMP    START
;================================; ;================================;
;=============== BACKSPACE Subroutine ==============;
BACKSPACE:
            COMPARE linecounter     , 00
            JUMP    Z               , COUNTISZERO
            LOAD    scratchbyte     , 08        ;BACKSPACE
            CALL    TRANSMIT
            LOAD    scratchbyte     , 20        ;SPACE
```

```
            CALL    TRANSMIT
            LOAD    scratchbyte     , 08        ;BACKSPACE
            CALL    TRANSMIT
            SUB     linecounter     , 01        ; SUBTRACT line counter
COUNTISZERO:
            RETURN
;================ Prompt Subroutine =============;
PROMPT:
            LOAD    scratchbyte     , 7E        ;~
            CALL    TRANSMIT
            LOAD    scratchbyte     , 24        ;$
            CALL    TRANSMIT
            RETURN
;================ NEWLINE Subroutine =============;
NEWLINE:
            LOAD    scratchbyte     , 0D        ;Line Feed
            CALL    TRANSMIT
            LOAD    scratchbyte     , 0A        ;Carage Return
            CALL    TRANSMIT
            CALL    PROMPT
            LOAD    linecounter     , 00            ; Reset line counter
            RETURN
;================ RECEIVE Subroutine =============;
RECEIVE:
            INPUT   checkstatus     , statusReg
            AND     checkstatus     , 01         ; MASK STATUS REGISTER FOR rX Ready BIT
            JUMP    Z               , RECEIVE
            INPUT   scratchbyte     , dataBus    ; READY VALUE FROM RX ENGINE
            RETURN

;================ Transmit Subroutine =============;
TRANSMIT:
            INPUT   checkstatus     , statusReg
            AND     checkstatus     , 02        ; Mask status register for tx ready bit
            JUMP    Z               , TRANSMIT
            OUTPUT  scratchbyte     , dataBus   ; Send value to Tx Engine
            RETURN
;================ BANNER Subroutine =============;
BANNER:
CSULBCECS460:
            LOAD    scratchbyte     , 43            ; C
            CALL    TRANSMIT
            LOAD    scratchbyte     , 53            ; S
            CALL    TRANSMIT
            LOAD    scratchbyte     , 55            ; U
            CALL    TRANSMIT
            LOAD    scratchbyte     , 4C            ; L
            CALL    TRANSMIT
            LOAD    scratchbyte     , 42            ; B
            CALL    TRANSMIT
            LOAD    scratchbyte     , 20            ; <space>
            CALL    TRANSMIT
            LOAD    scratchbyte     , 43            ; C
```

```
            CALL    TRANSMIT
            LOAD    scratchbyte     , 45            ; E
            CALL    TRANSMIT
            LOAD    scratchbyte     , 43            ; C
            CALL    TRANSMIT
            LOAD    scratchbyte     , 53            ; S
            CALL    TRANSMIT
            LOAD    scratchbyte     , 20            ; <space>
            CALL    TRANSMIT
            LOAD    scratchbyte     , 34            ; 4
            CALL    TRANSMIT
            LOAD    scratchbyte     , 36            ; 6
            CALL    TRANSMIT
            LOAD    scratchbyte     , 30            ; 0
            CALL    TRANSMIT
            CALL    NEWLINE
            RETURN
```

```
1   #================================================================================
2   #* File: TOPUCF.ucf
3   #* Description: System constraints file
4   ## Engineer: Raul Diaz
5   ## Course:   CECS460
6   ## Semester: Sp 15
7   ## Modified: 5/10/15
8   #================================================================================
9   NET "SYS_CLK"                 LOC = "B8"; //50Mhz Onboard Clk
10  //Buttons
11  NET "SYS_RST"                 LOC = "H13"; //BTN 3
12
13  // Switches
14  NET "UART_CONFIG[0]"          LOC = "G18"; //SW 0-odd_en
15  NET "UART_CONFIG[1]"          LOC = "H18"; //SW 1-parity_en
16  NET "UART_CONFIG[2]"          LOC = "K18"; //SW 2-bit8_en
17
18  NET "BAUD_SEL[3]"             LOC = "R17"; //SW 7
19  NET "BAUD_SEL[2]"             LOC = "N17"; //SW 6
20  NET "BAUD_SEL[1]"             LOC = "L13"; //SW 5
21  NET "BAUD_SEL[0]"             LOC = "L14"; //SW 4
22
23  //UART
24  NET "TX_OUT"                  LOC = "P9";  //tx out
25  NET "RX_IN"                   LOC = "U6";  //rx in
26
27  //LEDs
28  NET "UART_ERROR[2]"           LOC = "R4"; //LD7
29  NET "UART_ERROR[1]"           LOC = "F4"; //LD6
30  NET "UART_ERROR[0]"           LOC = "P15"; //LD5
31  #NET ""                  LOC = "E17"; //LD4
32  #NET ""                  LOC = "K14"; //LD3
33  #NET ""                  LOC = "K15"; //LD2
34  #NET ""                  LOC = "J15"; //LD1
35  #NET ""                  LOC = "J14"; //LD0
36
37  //ANODES
38  NET "ANODES[3]"              LOC = F15; //AN3
39  NET "ANODES[2]"              LOC = C18; //AN2
40  NET "ANODES[2]"              LOC = H17; //AN1
41  NET "ANODES[1]"              LOC = F17; //AN0
42  //Cathodes
43  NET "CATHODES[6]"            LOC = L18;  //a
44  NET "CATHODES[5]"            LOC = F18;  //b
45  NET "CATHODES[4]"            LOC = D17;  //c
46  NET "CATHODES[3]"            LOC = D16;  //d
47  NET "CATHODES[2]"            LOC = G14;  //e
48  NET "CATHODES[1]"            LOC = J17;  //f
49  NET "CATHODES[0]"            LOC = H14;  //g
50
51  //Memory Configuration
52  NET "CE_"                    LOC="R6";
53  NET "WE_"                    LOC="N7";
54  NET "OE_"                    LOC="T2";
55  NET "ADV_"                   LOC="J4";
56  NET "CRE"                    LOC ="P7";
57  NET "UB_"                    LOC="K4";
```

```
  58   NET "LB_"                   LOC="K5";
  59   NET "OCLKMEM"               LOC="H5";
  60   //Memory Address
  61   NET "MEM_ADDR[22]"          LOC="K6";
  62   NET "MEM_ADDR[21]"          LOC="D1";
  63   NET "MEM_ADDR[20]"          LOC="K3";
  64   NET "MEM_ADDR[19]"          LOC="D2";
  65   NET "MEM_ADDR[18]"          LOC="C1";
  66   NET "MEM_ADDR[17]"          LOC="C2";
  67   NET "MEM_ADDR[16]"          LOC="E2";
  68
  69   NET "MEM_ADDR[15]"          LOC="M5";
  70   NET "MEM_ADDR[14]"          LOC="E1";
  71   NET "MEM_ADDR[13]"          LOC="F2";
  72   NET "MEM_ADDR[12]"          LOC="G4";
  73   NET "MEM_ADDR[11]"          LOC="G5";
  74   NET "MEM_ADDR[10]"          LOC="G6";
  75   NET "MEM_ADDR[9]"           LOC="G3";
  76   NET "MEM_ADDR[8]"           LOC="F1";
  77
  78   NET "MEM_ADDR[7]"           LOC="H6";
  79   NET "MEM_ADDR[6]"           LOC="H3";
  80   NET "MEM_ADDR[5]"           LOC="J5";
  81   NET "MEM_ADDR[4]"           LOC="H2";
  82   NET "MEM_ADDR[3]"           LOC="H1";
  83   NET "MEM_ADDR[2]"           LOC="H4";
  84   NET "MEM_ADDR[1]"           LOC="J2";
  85   NET "MEM_ADDR[0]"           LOC="J1";
  86   //Memory Data
  87   NET "DQ[15]"                LOC="T1";
  88   NET "DQ[14]"                LOC="R3";
  89   NET "DQ[13]"                LOC="N4";
  90   NET "DQ[12]"                LOC="L2";
  91   NET "DQ[11]"                LOC="M6";
  92   NET "DQ[10]"                LOC="M3";
  93   NET "DQ[9]"                 LOC="L5";
  94   NET "DQ[8]"                 LOC="L3";
  95   NET "DQ[7]"                 LOC="R2";
  96   NET "DQ[6]"                 LOC="P2";
  97   NET "DQ[5]"                 LOC="P1";
  98   NET "DQ[4]"                 LOC="N5";
  99   NET "DQ[3]"                 LOC="M4";
 100   NET "DQ[2]"                 LOC="L6";
 101   NET "DQ[1]"                 LOC="L4";
 102   NET "DQ[0]"                 LOC="L1";
 103
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ## Engineer: Raul Diaz
4   ## Course:   CECS460
5   ## Semester: Sp 15
6   ## Modified: 5/10/15
7   ==================================================================================
8   * File: TopLevel.v
9   ==================================================================================
10  * The Top level module implements the Core design and the TSI Block which are used
11  * to communicate with an exterior interface. For the purposes of this project, the
12  * micron memory located on the Nexys2 board will be used. It is the functionality
13  * of the TSI block to handle the I/O used by the core. The TSI block can be
14  * modified to handle various electrical characteristics which may be presented in
15  * a real world application. However for the sake of simplicity, the I/O ports of
16  * the top level design is assumed to handle the standard electrical inputs and
17  * distribute the standard electrical output. If one may choose to alter the
18  * electrical I/O, one may do so in the constraints file
19  **********************************************************************************/
20  module TopLevel(    SYS_CLK, SYS_RST, RX_IN, UART_CONFIG, BAUD_SEL,
21                                TX_OUT, UART_ERROR,  MEM_ADDR, DQ,
22                                CE_, OE_, WE_, ADV_, CRE, UB_, LB_,OCLKMEM,
23                                ANODES, CATHODES,
24                );
25
26        input  SYS_CLK, SYS_RST;
27        input  RX_IN;
28        input  [2:0] UART_CONFIG;
29        input  [3:0] BAUD_SEL;
30        output TX_OUT, CE_, OE_, WE_, ADV_, CRE, UB_, LB_, OCLKMEM;
31        output [2:0] UART_ERROR;
32        output [3:0] ANODES;
33        output [6:0] CATHODES;
34        output [22:0]MEM_ADDR;
35        inout  [15:0] DQ;
36
37        /* Interconnects */
38        wire       Clk, Rst, parity_en, odd_en, bit8_en, Rx_in, Tx_data;
39        wire       cs, rd_mem, wr_mem, adv_mib, cre_mib,  upperbyte_en, lowbyte_en;
40        wire [3:0] baud_sel;
41        wire [15:0] data_to_mem, data_from_mem;
42        wire [22:0] addr_to_mem;
43
44        CoreLogic CORE(
45            .Clk(Clk),                      .Rst(Rst),
46            .baud_sel(baud_sel),            .parity_en(parity_en),
47            .odd_en(odd_en),                .bit8_en(bit8_en),
48            .Rx_in(Rx_in),
49            .data_from_mem(data_from_mem),
50
51            /*************** OUTPUTS ********************/
52            .Tx_data(Tx_data),          .flagreg(flagreg),
53            .data_to_mem(data_to_mem),  .addr_to_mem(addr_to_mem),
54            .cs(cs),                    .rd_mem(rd_mem),
55            .wr_mem(wr_mem),            .adv_mib(adv_mib),
56            .cre_mib(cre_mib),          .upperbyte_en(upperbyte_en),
57            .lowbyte_en(lowbyte_en),
```

```
58            .a3(a3),                    .a2(a2),
59            .a1(a1),                    .a0(a0),
60            .a(a), .b(b), .c(c), .d(d), .e(e), .f(f), .g(g)
61          );
62
63          TSI_Block TSI(
64           /* TSI SYSTEM INPUT Buffers */   /* INPUTS TO CORE LOGIC  */
65             .Sys_Clk(SYS_CLK),             .Clk_buf(Clk),
66             .Sys_Rst(SYS_RST),             .Rst_buf(Rst),
67             .Rx_buf(RX_IN),                .Rx_in(Rx_in),
68             .UART_Confg_buf(UART_CONFIG),  .UART_Config({bit8_en, parity_en, odd_en}),
69             .baud_sel_buf(BAUD_SEL),       .baud_sel(baud_sel),
70             .DQ(DQ),                       .data_from_mem(data_from_mem),
71
72           /* OUTPUTS FROM CORE LOGIC  */        /* TSI SYSTEM OUTPUT Buffers */
73             .data_to_mem(data_to_mem),
74             .Tx_data(Tx_data),                .Tx_buf(TX_OUT),
75             .cs(cs),                          .CE_(CE_),
76             .rd_mem(rd_mem),                  .OE_(OE_),
77             .wr_mem(wr_mem),                  .WE_(WE_),
78             .adv_mib(adv_mib),                .ADV_(ADV_),
79             .cre_mib(cre_mib),                .CRE(CRE),
80             .upperbyte_en(upperbyte_en),      .UB_(UB_),
81             .lowbyte_en(lowbyte_en),          .LB_(LB_),
82             .addr_to_mem(addr_to_mem),        .Addr_buf(MEM_ADDR),
83             .anodes({a3, a2, a1, a0}),        .anodes_buf(ANODES),
84             .cathodes({a, b, c, d, e, f, g}), .cathodes_buf(CATHODES),
85             .UART_error(flagreg),             .UART_error_buf(UART_ERROR),
86             .CLKMEM(1'b0),                    .OCLKMEM(OCLKMEM)
87         );
88
89   endmodule
90
```

```
1   `timescale 1ns / 1ps
2   /**********************************************************************************
3   ## Engineer: Raul Diaz
4   ## Course:   CECS460
5   ## Semester: Sp 15
6   ## Modified: 5/10/15
7   ==================================================================================
8   * File: TSI_Block.v
9   ==================================================================================
10  *    The technology specific instance block allows for a real world application to be
11  *  applied to the SoC. By instantiating the I/O buffers provided by the Spartan 3E HDL
12  *  library, the TSI ensures a synchronized design to any exterior application. The TSI
13  *  also adds portability by providing flexible electrical to specification to the
14  *  cores I/O ports. This can be done in the systems constrains file.
15  **********************************************************************************/
16  module TSI_Block(
17
18       /* CORE Portlist */
19         input  Sys_Clk, Sys_Rst, Tx_data,
20         input  cs, rd_mem, wr_mem, adv_mib, cre_mib, upperbyte_en, lowbyte_en,
21         input  [15:0] data_to_mem,
22         input  [22:0] addr_to_mem,
23         input  [2:0] UART_error,
24         input  [3:0] anodes,
25         input  [6:0] cathodes,
26
27         output  Rx_in,
28         output  [15:0] data_from_mem,
29         output  [2:0] UART_Config,
30         output  [3:0] baud_sel,
31
32         /* TSI Buffers */
33         output Clk_buf, Rst_buf,
34         output Tx_buf,
35         output  CE_,OE_, WE_, ADV_, CRE, UB_, LB_,
36         output  [22:0]Addr_buf,
37         output  [2:0] UART_error_buf,
38         output  [3:0] anodes_buf,
39         output  [6:0] cathodes_buf,
40         output  OCLKMEM ,
41         input  Rx_buf,
42         input  [2:0] UART_Confg_buf,
43         input  [3:0] baud_sel_buf,
44         input  CLKMEM,
45         inout  [15:0] DQ
46        );
47
48  /*################## CLOCK BUFFER ############################## */
49
50       IBUFG BUFG_CLK (
51           .O(Clk_buf),         // 1-bit output: Clock buffer output
52           .I(Sys_Clk)          // 1-bit input: Clock buffer input
53       );
54
55
56  /*################## INPUT Buffers ############################## */
57       IBUFG
58            /* Reset Handler */
```

```
 59             IBUFG_RST (
 60                  .I(Sys_Rst),   // Buffer input (connect directly to top-level port)
 61                  .O(Rst_buf)    // Buffer output
 62             ),
 63
 64             /* Recieve RX Handler */
 65             IBUFG_Rx (
 66                  .I(Rx_buf),
 67                  .O(Rx_in)
 68             );
 69             /* UART CONFIG Handler */
 70        IBUF IBUF_UARTCONFIG [6:0] (
 71                  .I({UART_Confg_buf,baud_sel_buf}),
 72                  .O({UART_Config, baud_sel})
 73             );
 74
 75   /*################# OUTPUT Buffers ############################## */
 76
 77             /* Address Handler */
 78        OBUF OBUF_ADDR[22:0]  (
 79             .O(Addr_buf[22:0]),    // Buffer output (connect directly to top-level port)
 80             .I(addr_to_mem[22:0])  // Buffer input
 81             ),
 82             /* Anodes/Cathodes Handler */
 83             OBUF_disp[10:0] (
 84                  .O({anodes_buf, cathodes_buf}),
 85                  .I({anodes, cathodes})
 86             ),
 87             /* UART ERROR Handler */
 88             OBUF_UARTErr[2:0] (
 89                  .O(UART_error_buf[2:0]),
 90                  .I(UART_error[2:0])
 91             ),
 92             /* Memory Control Handler */
 93             OBUF_MemCNTRL[6:0] (
 94                  .O({CE_,OE_, WE_, ADV_, CRE, UB_, LB_ }),
 95                  .I({cs, rd_mem, wr_mem, adv_mib, cre_mib, upperbyte_en, lowbyte_en})
 96             ),
 97             /* Tx Handler */
 98             OBUF_TX (
 99                  .O(Tx_buf),
100                  .I(Tx_data)
101             ),
102             /* MemClock Handler */
103             OBUF_CLKMEM(
104                   .O(OCLKMEM),
105                   .I(CLKMEM)
106             );
107
108   /*################# InOut Buffers ############################## */
109              /* Data Handler */
110        IOBUF IOBUF_DQ[15:0]  (
111                    .O(data_from_mem[15:0]),// Buffer output
112                    .IO(DQ[15:0]),          // Buffer inout port
113                    .I(data_to_mem[15:0]),  // Buffer input
114                    .T(wr_mem)              // 3-state enable input
115                  );
116
```

```
1   `timescale 1ns / 1ps
2   /**********************************************************************************
3   ## Engineer: Raul Diaz
4   ## Course:   CECS460
5   ## Semester: Sp 15
6   ## Modified: 5/10/15
7   ==================================================================================
8   * File: CoreLogic.v
9   ==================================================================================
10  *    The Core cell contains the logic flow of the SoC. By instantiating the embedded
11  * Picoblaze processer, UART, and the Memory Interface block, the Core is capable
12  * of providing system communication to an exterior device.
13  *    A 4:1 selection case statement multiplexes the data going into the PicoBlaze
14  * processor. The read strobes set within the decoder module determine the selection
15  * of the multiplexer, which are set by software logic flow within the PicoBlaze.
16  * The mux provides the PicoBlaze the option of reading received data from the UART,
17  * the status of the UART, the data read from the embedded micron memory or the
18  * status of the MIB module (Ready Flag).
19
20  **********************************************************************************/
21  module CoreLogic(Clk, Rst, baud_sel, parity_en, odd_en, bit8_en, Rx_in,
22                           Tx_data, a3, a2, a1, a0,
23                           a, b, c, d, e, f, g, flagreg,
24                           data_from_mem,
25                           data_to_mem, addr_to_mem,
26                           cs, rd_mem, wr_mem, adv_mib, cre_mib, upperbyte_en,
27                           lowbyte_en
28                           );
29
30          input  Clk, Rst;
31          input  parity_en, odd_en, bit8_en, Rx_in;
32          input  [3:0] baud_sel;
33
34          input  [15:0] data_from_mem;
35          output [15:0] data_to_mem;
36          output [22:0] addr_to_mem;
37
38          output cs, rd_mem, wr_mem, adv_mib, cre_mib, upperbyte_en, lowbyte_en;
39
40
41          output          Tx_data;
42          output          a3, a2, a1, a0;
43          output          a, b, c, d, e, f, g;
44          output reg [2:0]flagreg;
45
46          wire   p_err, ov_err, frm_err;
47          wire   Tx_rdy, wr_st, Rx_rdy, sRst, flag;
48          wire   [7:0]        port_ID, port_out_data, status, Rx_out;
49          wire   [7:0]        data_to_pb;
50          wire   [17:0]       baud_val;
51          wire   [255:0]      wr_decode, rd_decode;
52          wire   [7:0]        MIB_status;
53          reg    [7:0]       port_in_data;
54          reg    [7:0]        rxdata, txdata;
55
56        assign   status = {1'b0, frm_err,ov_err, p_err, 2'b0, Tx_rdy, Rx_rdy };
57        assign   flag = frm_err | ov_err | p_err;
```

```
 58
 59      /*****************************************
 60        Procedrual Combo block
 61      *****************************************/
 62
 63        always  @(*) begin
 64             /*******************************************
 65                  4:1 Mux Input to PicoBlaze Data Port
 66             *******************************************/
 67          casex({rd_decode[20], rd_decode[17], rd_decode[16], rd_decode[1], rd_decode[0]})
 68                  5'bxxxx1:      port_in_data = status;
 69                  5'bxxx1x:      port_in_data = Rx_out;
 70                  5'bxx1xx:      port_in_data = data_to_pb;
 71                  5'bx1xxx:      port_in_data = data_to_pb;
 72                  5'b1xxxx:      port_in_data = MIB_status;
 73                  default: port_in_data = port_in_data;
 74          endcase
 75        end
 76
 77        /*****************************************
 78                     Sequential block
 79        *****************************************/
 80        always @(posedge Clk, negedge sRst) begin
 81             if(!sRst) begin
 82                {txdata,rxdata,flagreg}<= 19'b0;
 83
 84             end
 85             /*******************************
 86                Rx buffer register
 87             *******************************/
 88             else
 89             if(rd_decode[1])     rxdata <= Rx_out;
 90             /*******************************
 91                Tx buffer register
 92             *******************************/
 93             else
 94             if(Tx_rdy && wr_decode[1])         txdata <= port_out_data;
 95             /*******************************
 96                Flag buffer register
 97             *******************************/
 98             else
 99             if(flag)
100                  flagreg <= {frm_err,ov_err,p_err};
101
102
103        end
104
105        //*********************************************************
106        // Memory Interface Block
107        //*********************************************************
108          Memory_Interface_Block  MIB(
109             .clk(Clk),
110             .rst(sRst),
111             .wr_strb(wr_decode),
112             .rd_strb(rd_decode),
113             .datain(port_out_data),
114             .data_from_mem(data_from_mem),
```

```
115             .data_to_mem(data_to_mem),
116             .addr_to_mem(addr_to_mem),
117             .status(MIB_status),
118             .CE_(cs),
119             .OE_(rd_mem),
120             .WE_(wr_mem),
121             .ADV_(adv_mib),
122             .CRE(cre_mib),
123             .UB_(upperbyte_en),
124             .LB_(lowbyte_en),
125             .dataout(data_to_pb)
126       );
127
128     //*********************************************************
129     // Recieve Engine
130     //*********************************************************
131       RxEngine RX(
132             .Clk(Clk),
133             .Rst(sRst),
134             .Rx_in(Rx_in),
135             .parity_en(parity_en),
136             .bit8_en(bit8_en),
137             .odd_en(odd_en),
138             .Baud_val(baud_val),
139             .rd_strb(rd_decode[0]),
140             .Rx_out(Rx_out),
141             .Rx_rdy(Rx_rdy),
142             .p_err(p_err),
143             .ov_err(ov_err),
144             .frm_err(frm_err)
145       );
146
147     //*********************************************************
148     // Transmit Engine
149     //*********************************************************
150      TxEngine TX(
151             .Clk(Clk),
152             .Rst(sRst),
153             .data_in(port_out_data),
154             .tx_start(wr_decode[1]),
155             .parity_en(parity_en),
156             .bit8_en(bit8_en),
157             .odd_en(odd_en),
158             .Baud_val(baud_val),
159             .tx_out(Tx_data),
160             .tx_rdy(Tx_rdy)
161        );
162
163     //*********************************************************
164     // Baudrate value Decoder
165     //*********************************************************
166      Baud_val_Decoder br_decode(
167             //.Clk(Clk),
168             //.Rst(sRst),
169             .Baudsel(baud_sel),
170             .Baud_val(baud_val)
171       );
```

```
172
173         //*********************************************************
174         // Decode Block Module
175         //*********************************************************
176         strobe_decode decode(
177                 .port_id(port_ID),
178                 .rd_in(rd_st),
179                 .wr_in(wr_st),
180                 .rd_out(rd_decode),
181                 .wr_out(wr_decode)
182           );
183         //*********************************************************
184         // Asynchronous In Synchronous Out Module
185         //*********************************************************
186         ASyncIn_SyncOut aiso(
187                 .Clk(Clk),
188              .async_rst(Rst),
189                 .sync_rst(sRst)
190         );
191         //*********************************************************
192         // PicoBlaze Processor
193         //*********************************************************
194         embedded_kcpsm3 ROM(
195                 .port_id(port_ID),
196                 .write_strobe(wr_st),
197                 .read_strobe(rd_st),
198                 .out_port(port_out_data),
199                 .in_port(port_in_data),
200                 .interrupt(1'b0),
201                 .interrupt_ack( ),
202                 .reset(~sRst),
203                 .clk(Clk)
204         );
205
206         //*********************************************************
207         // Display Controller Module
208         //*********************************************************
209         display_controller  disp_cont(
210                 .clk_50Mhz(Clk),
211                 .reset(~sRst),
212                 .bytesel_hi(txdata[7:4]),
213                 .bytesel_lo(txdata[3:0]),
214                 .d_hi(rxdata[7:4]),
215                 .d_lo(rxdata[3:0]),
216                 .a3(a3), .a2(a2), .a1(a1), .a0(a0),
217                 .a(a), .b(b), .c(c), .d(d), .e(e), .f(f), .g(g)
218               );
219     endmodule
220
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ## Engineer: Raul Diaz
4   ## Course:   CECS460
5   ## Semester: Sp 15
6   ## Modified: 5/10/15
7   ==================================================================================
8   * File: Memory_Interface_Block.v
9   ==================================================================================
10  *    The Memory Interface Block provides the communication link between
11  * the PicoBlaze processor and the exterior memory interface. The design of the
12  * MIB is constructed around the MIB State Machine which provides the necessary
13  * load signals to the 2 read buffer register holding data from memory. The state
14  * machine also provides read and write enable (OE_ & WE_) signals to the memory
15  * chip on respective states the PicoBlaze may be in.
16
17  **********************************************************************************/
18  module Memory_Interface_Block(clk, rst, wr_strb, rd_strb, data_from_mem,
19                                addr_to_mem, status, datain, data_to_mem,  dataout,
20                                CE_,OE_, WE_, ADV_, CRE, UB_, LB_
21
22                                );
23
24        input     clk, rst;
25        input     [7:0]   datain;
26        input     [15:0]  data_from_mem;
27        input     [255:0] wr_strb, rd_strb;
28        output    CE_,OE_, WE_, ADV_, CRE, UB_, LB_;
29        output    [7:0] dataout;
30        output    [7:0]  status;
31        output    [15:0] data_to_mem;
32        output    [22:0] addr_to_mem;
33
34        wire      ld;
35        wire      [7:0] rd_data0, rd_data1;
36
37        assign    dataout = rd_strb[17] ? rd_data1 : rd_data0;
38        assign    {ADV_, CRE, UB_, LB_} = 4'b0;      /* Always low */
39
40      /*****************************************************
41      *   RS Flop: Sets and Reseting MIB Ready Status
42      *****************************************************/
43        RSFlop Mem_RDY(
44          .Clk(clk),
45          .Rstb(rst),
46          .R(rd_strb[18]),
47          .S(ld),
48          .Q(status[0])
49        );
50
51      /*****************************************************
52      *      MIB State Machine
53      *****************************************************/
54        MIB_SM SM(
55          .Clk(clk),
56          .Rstb(rst),
57          .do_rd(rd_strb[18]),
```

```
  58          .do_wr(wr_strb[19]),
  59          .cs(CE_),
  60          .rd_mem(OE_),
  61          .wr_mem(WE_),
  62          .ld(ld)
  63         );
  64
  65      /*****************************************************
  66      *    Synch Clock; Active Low Rst: 8 bit Registers
  67      *****************************************************/
  68        /* Address Registers */
  69        LDRG_8bit
  70                  addr0(
  71                       .Clk(clk),
  72                       .Rstb(rst),
  73                       .D(datain),
  74                       .en(wr_strb[11]),
  75                       .Q(addr_to_mem[7:0])
  76                  ),
  77                  addr1(
  78                       .Clk(clk),
  79                       .Rstb(rst),
  80                       .D(datain),
  81                       .en(wr_strb[12]),
  82                       .Q(addr_to_mem[15:8])
  83                  ),
  84                  addr2(
  85                       .Clk(clk),
  86                       .Rstb(rst),
  87                       .D(datain),
  88                       .en(wr_strb[13]),
  89                       .Q(addr_to_mem[22:16])
  90                  ),
  91
  92        /* Write Buffer Register */
  93                  wrdata0(
  94                       .Clk(clk),
  95                       .Rstb(rst),
  96                       .D(datain),
  97                       .en(wr_strb[14]),
  98                       .Q(data_to_mem[7:0])
  99                  ),
 100                  wrdata1(
 101                       .Clk(clk),
 102                       .Rstb(rst),
 103                       .D(datain),
 104                       .en(wr_strb[15]),
 105                       .Q(data_to_mem[15:8])
 106                  ),
 107
 108        /* Read Buffer Register */
 109                  ReadBuff_reg1(
 110                       .Clk(clk),
 111                       .Rstb(rst),
 112                       .D(data_from_mem[15:8]),
 113                       .en(ld),
 114                       .Q(rd_data1)
```

```
115                     ),
116                     ReadBuff_reg0(
117                           .Clk(clk),
118                           .Rstb(rst),
119                           .D(data_from_mem[7:0]),
120                           .en(ld),
121                           .Q(rd_data0)
122                     );
123     endmodule
124
```

```
1   `timescale 1ns / 1ps
2   /**********************************************************************************
3   ## Engineer: Raul Diaz
4   ## Course:   CECS460
5   ## Semester: Sp 15
6   ## Modified: 5/10/15
7   ==================================================================================
8   * File: MIB_SM.v
9   ==================================================================================
10  *  The MIB Finite State Machine is an essential component to MIB. It controls the Chip
11  *  Select, Read, and Write enable strobes used to activate the micron memory chip.
12  *  Upon reset, this FSM remains in idle waiting for rdstrobe[18] or wrstrobe[19] from
13  *  the system. Based on received strobe, the FSM goes into an uninterrupted 6 state
14  *  transition changing on each rising edge of the system clock.  Details on
15  *  communication can be found in the Micron Memory Datasheet.
16
17  **********************************************************************************/
18  module MIB_SM(Clk, Rstb, do_rd, do_wr, cs, rd_mem, wr_mem, ld);
19       input      Clk, Rstb;
20       input      do_rd, do_wr;
21       output reg cs, rd_mem, wr_mem, ld;
22
23       reg ns_cs, ns_rd, ns_wr, ns_ld;
24       reg [3:0] state_reg, state_next;
25
26       /* Symbolic State Declorations */
27       localparam [3:0]
28            idle      = 4'h0,
29            wr_1      = 4'h1,   rd_1      = 4'h7,
30            wr_2      = 4'h2,   rd_2      = 4'h8,
31            wr_3      = 4'h3,   rd_3      = 4'h9,
32            wr_4      = 4'h4,   rd_4      = 4'hA,
33            wr_5      = 4'h5,   rd_5      = 4'hB,
34            wr_6      = 4'h6,   rd_6      = 4'hC;
35
36  /*****************  Sequential block  **********************/
37       always @(posedge Clk, negedge Rstb)
38            if(!Rstb) begin
39                 state_reg  <= idle;
40                 {cs, rd_mem, wr_mem, ld} <= 4'b1110;
41            end
42            else       begin
43                 state_reg <= state_next;
44                 {cs, rd_mem, wr_mem, ld} <= {ns_cs, ns_rd, ns_wr, ns_ld};
45            end
46
47  /*************Combo & Next State Logic block******************/
48       always @(*) begin
49            /* Default Assignments */
50            state_next = state_reg;
51            {ns_cs, ns_rd, ns_wr, ns_ld} = 4'b1110;
52
53            case(state_reg)
54                 idle:
55                 begin
56                      if(do_rd)
57                           {state_next, ns_cs} = {rd_1, 1'b0}; /* Enable Chip Sel */
```

```
58                      else
59                      if(do_wr)
60                           {state_next, ns_cs} = {wr_1, 1'b0}; /* Enable Chip Sel */
61
62                      else
63                           state_next = idle ;
64                 end
65   /************* Write Cycle *****************/
66                 wr_1:
67                 begin
68                      state_next = wr_2;
69
70                      /* Enable Chip Sel & Enable Memory Write */
71                      {ns_cs, ns_wr} = 2'b0;
72                 end
73                 wr_2 :
74                 begin
75                      state_next = wr_3;
76
77                      /* Enable Chip Sel & Enable Memory Write */
78                      {ns_cs, ns_wr} = 2'b0;
79                 end
80                 wr_3 :
81                 begin
82                      state_next = wr_4;
83
84                      /* Enable Chip Sel & Enable Memory Write */
85                      {ns_cs, ns_wr} = 2'b0;
86                 end
87                 wr_4:
88                 begin
89                      state_next = wr_5;
90
91                      /* Enable Chip Sel & Enable Memory Write */
92                      {ns_cs, ns_wr} = 2'b0;
93                 end
94                 wr_5:
95                 begin
96                      state_next = wr_6;
97
98                      /* Disable Memory Write */
99                      {ns_cs, ns_wr} = 2'b01;
100                end
101                wr_6: state_next = idle;
102
103  /************* Read Cycle *****************/
104                rd_1:
105                begin
106                     state_next = rd_2;
107                     /* Enable Chip Sel & Enable Memory Read */
108                     {ns_cs, ns_rd} = 2'b0;
109                end
110                rd_2 :
111                begin
112                     state_next = rd_3;
113                     /* Enable Chip Sel & Enable Memory Read */
114                     {ns_cs, ns_rd} = 2'b0;
```

```
115                     end
116                     rd_3:
117                     begin
118                          state_next = rd_4;
119                          /* Enable Chip Sel & Enable Memory Read */
120                          {ns_cs, ns_rd} = 2'b0;
121                     end
122                     rd_4:
123                     begin
124                          state_next = rd_5;
125                          /* Enable Chip Sel & Enable Memory Read */
126                          {ns_cs, ns_rd, ns_ld} = 3'b001;
127                     end
128                     rd_5:
129                     begin
130                          state_next = rd_6;
131                          /* Disable Read Write */
132                          {ns_cs, ns_rd} = 2'b01;
133                     end
134
135                     rd_6: state_next = idle;
136                endcase
137          end
138     endmodule
139
```

```
1   `timescale 1ns / 1ps
2   /////////////////////////////////////////////////////////////////////////////////
3   ================================================================================
4   * File: RxEngine.v
5   ================================================================================
6   * Description:
7   The Recieve Engine is a synchronous communications module capable of recieving
8   serial data using an RS232 protocol. It outputs an 8 bit data to an exterior device
9   and is capable of producing 4 status signals Rx Ready, Parity Error, Overflow Error
10  and a Framing Error.
11  There are 4 main instantiations involved: Bitcountup.v, Bit_time_counter.v, Recieve_SM,
12  and the SIPO Shift Register. An 18 bit Baud Count value is also produced based on the
13  start signal from the FSM. This signal will determine if the Baud Count is divided
14  by 2, in order to shift the trigger mark to the midpoint of a serial transfer. A shift
15  signal is also produced upon a low start signal from the FSM and a BTU signal to
16  determine a shift in the SIPO.
17
18  ********************************************************************************/
19  module RxEngine(Clk, Rst, Rx_in, parity_en, bit8_en, odd_en, Baud_val, rd_strb,
20                            Rx_out, Rx_rdy, p_err, ov_err, frm_err  );
21
22        input   Clk, Rst;
23        input   Rx_in, parity_en, bit8_en, odd_en, rd_strb;
24        input   [17:0] Baud_val;
25
26        output  Rx_rdy, p_err, ov_err, frm_err;
27        output  [7:0] Rx_out;
28
29        wire [17:0] baud_count;
30        wire start, BTU;
31
32  /*************************  Combo  Block  ********************************/
33        assign baud_count = start ? (Baud_val >> 1) : Baud_val;
34        assign shift = ~start & BTU;
35
36  /************************  Instantiation Block  ***************************/
37
38     /*****************************************
39               Bit Count up Module
40     *****************************************/
41       BitCountUp bitcounter(
42            .Clk(Clk),
43            .Rst(Rst),
44            .start(doit),
45            .BTU(BTU),
46            .bitCount(4'ha),
47            .done(BCU) );
48
49     /*****************************************
50                  Bit Time Counter Module
51     *****************************************/
52       Bit_time_counter bittimecount(
53            .Clk(Clk),
54            .Rst(Rst),
55            .start(doit),
56            .baud_count(baud_count),
57            .BTU(BTU) );
```

```
58        /*******************************************
59          Recieve State Machine
60       *******************************************/
61         Recieve_StateMachine SM(
62              .Clk(Clk),
63              .Rstb(Rst),
64              .Rx(Rx_in),
65              .BCU(BCU),
66              .BTU(BTU),
67              .start(start),
68              .ld(ld),
69              .doit(doit) );
70
71       /*******************************************
72                   Serial In Parallel Out Shift Reg
73        *******************************************/
74         Shiftreg_SIPO SIPO(
75              .Clk(Clk),
76              .Rst(Rst),
77              .ld(ld),
78              .Rx_in(Rx_in),
79              .shift(shift),
80              .parity_en(parity_en),
81              .bit8_en(bit8_en),
82              .odd_en(odd_en),
83              .rd_srtb(rd_strb),
84              .frm_err(frm_err),
85              .par_err(p_err),
86              .ov_err(ov_err),
87              .Rx_rdy(Rx_rdy),
88              .data_out(Rx_out) );
89
90    endmodule
91
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ================================================================================
4   * File: BitCountUp.v
5   ================================================================================
6   * The BitCountUp module determines the number of transfered bits during an
7   * operation. The done flag will signal when 'bitCount' bits have succesfully tranfered
8   * based on the BTU(bit time up) flag from the system.
9   * On Reset, count is set to bitCount in order to signal a done to the system.
10  *
11  *********************************************************************************/
12  module BitCountUp(Clk, Rst, start, BTU, bitCount, done);
13       input Clk, Rst;
14       input start, BTU;
15       input [3:0] bitCount;
16       output done;
17
18       reg [3:0] count;
19       assign done = (count == bitCount);
20
21       /*********************Sequential Block**********************/
22       always @(posedge Clk, negedge Rst)
23            if(!Rst)          count <= bitCount;     else
24            if(!start)        count <= 4'b0;         else
25            if(!BTU)          count <= count;        else
26                              count <= count +1;
27
28  endmodule
29
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ==================================================================================
4   * File: Bit_time_counter.v
5   ==================================================================================
6   * The Bit time counter module generates a single clock pulse to signal the end of
7   * a bit time transfer. A bit time for the system is determined by the baud_count
8   * wire, an 18 bit wide bus. The following table determines signal handling logic
9   *
10  *
11  *********************************************************************************/
12  module Bit_time_counter(Clk, Rst, start, baud_count, BTU);
13       input Clk, Rst;
14       input start;
15       input [17:0] baud_count;
16       output BTU;
17
18       reg [17:0] count;
19       assign BTU = (count == (baud_count - 1));
20
21       /*********************Sequential Block**********************/
22       always @(posedge Clk, negedge Rst)
23            if(!Rst)          count <= 18'b0;         else
24            if(start & !BTU) count <= count + 1;     else
25                             count <= 18'b0;
26
27  endmodule
28
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ================================================================================
4   * File: Shiftreg_SIPO.v
5   ================================================================================
6   Description:
7   The sychronous 'Serial in Parallel out' Shift Register implemented in this design
8   is used for the RX engine. This module also provides nessesary status flags to the
9   system for continous processing. On a shift signal from the system the SIPO will shift
10  new bits to the right. On a load & RxReady signal, the SIPO will update the data out
11  register with the recieved bits stored in the temp register.
12
13  This SIPO module is constructed for the use of an RS232 protocol and provides error
14  signals upon received data vs computed data such as a Framming Error, Parity Error
15  and Overflow error. This module also provides logic for an RxReady signal which
16  determines if the RX engine is ready too recieve a sequence of information.
17  Upon a load signal, the last bit(stop bit) is excluded from the data_out register.
18  Depending on the UART configuration, an optional 8th bit is loaded along with all
19  other recieved bits.
20  **********************************************************************************/
21  module Shiftreg_SIPO(Clk, Rst, ld, Rx_in, shift, parity_en, bit8_en, odd_en, rd_srtb,
22                       frm_err, par_err, ov_err, Rx_rdy, data_out);
23
24     input  Clk, Rst;
25     input  ld, Rx_in, shift, parity_en, bit8_en, odd_en, rd_srtb;
26
27     output  frm_err, par_err, ov_err, Rx_rdy;
28     output reg [7:0] data_out;
29
30     reg [8:0] temp;
31     wire bit7, recived_parity, computed_parity ;
32     wire sP_err, sOv_err, sRx_rdy, sFrm_err, bit7xortemp;
33
34  /*************Combinational Block*****************/
35     /* Bit 7 assignment */
36     assign bit7 = bit8_en ? 1'b0 : temp[7];
37     /*xor on 7 bits of shifted value and bit7 */
38     assign bit7xortemp = temp[6:0] ^ bit7;
39
40     /* Compute parity from recived data*/
41     assign computed_parity = ~odd_en ? bit7xortemp : ~bit7xortemp ;
42     /* Locate recived parity */
43     assign recived_parity = bit8_en ? temp[7] : temp[8];
44
45     /* Set signal for p_err register & ov_err register*/
46     assign sP_err = ld & parity_en & ~Rx_rdy & (computed_parity ^ recived_parity);
47     assign sOv_err = Rx_rdy & ld;
48
49     /* set signal for Rx_rdy register & frm_err register*/
50     assign sRx_rdy = ~Rx_rdy & ld;
51     assign sFrm_err = ~Rx_rdy & ld  & sP_err & temp[8]; //Stopbit && parity error
52  /*************Instantiation Block*****************/
53     RSFlop   perror(
54                 .Clk(Clk),
55                 .Rstb(Rst),
56                 .R(rd_srtb),
57                 .S(sP_err),
```

```
58                .Q(par_err)) ;
59
60      RSFlop   overror(
61                .Clk(Clk),
62                .Rstb(Rst),
63                .R(rd_srtb),
64                .S(sOv_err),
65                .Q(ov_err)) ;
66
67      RSFlop   frmerror(
68                .Clk(Clk),
69                .Rstb(Rst),
70                .R(rd_srtb),
71                .S(sFrm_err),
72                .Q(frm_err)) ;
73
74      RSFlop   rxRdy(
75                .Clk(Clk),
76                .Rstb(Rst),
77                .R(rd_srtb),
78                .S(sRx_rdy),
79                .Q(Rx_rdy)) ;
80
81   /*************Sequential block*****************/
82      always @(posedge Clk, negedge Rst) begin
83         if(!Rst) {temp, data_out} <= 17'b0;                 else
84         if(ld && ~Rx_rdy) data_out <= { bit7, temp[6:0] };  else
85         if(shift) temp <= {Rx_in, temp[8:1]};
86
87
88      end
89
90   endmodule
91
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ================================================================================
4   * File: Recieve_StateMachine.v
5   ================================================================================
6   Description:
7   This sychronous State Machine provides the Recieve Engine the nessary signals
8   for recieving and processing particular bits based on the current state of
9   the RS232 protocol transfer.
10
11  There are 4 states exercised at all times during execution:
12  The Idle state will wait for the start bit during an RS232 protocol, it ensures
13  grounded signals from all counters used in the RX Engine. Upon a low signal from
14  the Rx data input, the State will transition to the Start state for further
15  processing.
16
17  At the Start State, the system is processing the start bit from an exterior device
18  and will set the trigger for shifting and storing values at the middle of a bit
19  transfer from the percpective of the exterior device. (i.e sampling is performed
20  at the middle of a bit transfer to ensure proper processing of a bit, this will
21  reduce timing errors due too baudrates of other devices). The Start state also
22  begins all counters used in the RX engine. When a BTU signal is recieved, the system
23  will transistion into the Get state.
24
25  The Get state provides the nessasary signals used to store the incomming bits in the
26  RX engine. It will transition into the load state on a high BTU signal.
27
28  At the Load state, the system is checking for the BCU signal which signals the maximum
29  allowable bits for in transfer has occured. At which point, the FSM will provide a
30  load signal to the system to allow access to the recieved data.The state will then
31  transistion into the Idle state and wait for the next start bit.However if a low
32  BCU signal is read, the state will reseed to the get state for further processing.
33  *********************************************************************************/
34  module Recieve_StateMachine(Clk, Rstb, Rx, BCU, BTU,
35                              start, ld, doit);
36     input Clk, Rstb;
37     input Rx, BCU, BTU;
38
39     output reg start, ld, doit;
40
41     reg [1:0] state_reg, state_next;
42     reg nxt_start, nxt_ld, nxt_doit;
43
44     /* Symbolic State Declorations */
45     localparam [1:0]
46        idle     = 2'b00,
47        startbit = 2'b01,
48        get      = 2'b10,
49        load     = 2'b11;
50
51  /*****************  Sequential block  **********************/
52     always @(posedge Clk, negedge Rstb)
53        if(!Rstb)   begin
54           state_reg <= idle;
55           start     <= 1'b0;
56           ld        <= 1'b0;
57           doit      <= 1'b0;
```

```
58          end
59          else        begin
60             state_reg <= state_next;
61             start     <= nxt_start ;
62             ld        <= nxt_ld    ;
63             doit      <= nxt_doit ;
64          end
65
66    /*************Combo & Next State Logic block******************/
67
68    /******************  FSM Content  ****************************
69    Idle      : Waits for start bit
70    Startbit  : Sets trigger point, and begins counters to system
71    Get       : Recieves bits from exterior device
72    Load      : Allows access system to read recieved data
73    ***************************************************************/
74
75       always @(*) begin
76       /* Default Assignments */
77          state_next = state_reg;
78          nxt_start = start ;
79          nxt_ld    = ld    ;
80          nxt_doit  = doit ;
81
82          case(state_reg)
83             idle :
84                begin
85                   if(!Rx)  begin
86                      state_next = startbit;
87                      nxt_start   = 1'b1;
88                      nxt_ld      = 1'b0;
89                      nxt_doit    = 1'b1;
90                   end
91                   else  begin
92                      state_next = idle;
93                      nxt_start   = 1'b0;
94                      nxt_ld      = 1'b0;
95                      nxt_doit    = 1'b0;
96                   end
97                end
98
99             startbit :
100               begin
101
102                  if(~Rx && BTU) begin
103                     state_next = get;
104                     nxt_start   = 1'b0;
105                     nxt_ld      = 1'b0;
106                     nxt_doit    = 1'b1;
107                  end
108                  else
109                  if(Rx)   begin
110                     state_next = idle;
111                     nxt_start   = 1'b0;
112                     nxt_ld      = 1'b0;
113                     nxt_doit    = 1'b0;
114                  end
```

```
115                end
116
117             get:
118                begin
119                   if(BTU) begin
120                      state_next = load;
121                      nxt_start   = 1'b0;
122                      nxt_doit    = 1'b1;
123                      nxt_ld      = 1'b0;
124                   end
125                end
126
127             load:
128                begin
129                   if(BCU) begin
130                      state_next = idle;
131                      nxt_start   = 1'b0;
132                      nxt_ld      = 1'b1;
133                      nxt_doit    = 1'b0;
134
135                   end
136                   else  begin
137                      state_next = get;
138                      nxt_start   = 1'b0;
139                      nxt_ld      = 1'b0;
140                      nxt_doit    = 1'b1;
141                   end
142                end
143
144             default:
145                begin
146                   state_next = idle;
147                   nxt_start   = 1'b0;
148                   nxt_ld      = 1'b0;
149                   nxt_doit    = 1'b0;
150                end
151          endcase
152          /************  END FSM  ****************/
153       end
154
155    endmodule
156
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ==================================================================================
4   * File: TxEngine.v
5   ==================================================================================
6      The TxEngine module is an 8 bit sychronous loadable transmit engine. Operation is
7   executed on a high signal fron the tx_start port which triggers the load of an
8   8 bit register and signals 2 counters which handle baud rate timing and bit
9   transfers for the system. The tx_out will transmit an 11 bit value using a
10  'Parallel In- Serial out' shift register module. The 11 bit value exercises an
11  RS232 protocol which includes: 1 stop bit, 1 start bit, 8 bit data , and based on
12  the UART Config register, a Parity bit. Trailing ones, are followed after data has
13  completed transmission.
14     The UART configeration is set one time upon reset and stored in a 3 bit register
15  used in a procedural case statment. bit8_en port will determine if an 8 bit value
16  is transmitted, if the port is low by default the system will transmit 7 bits.
17  parity_en port will determine if parity is detected on the input, and Odd_en
18  port will check for an Odd parity or even parity if bit is set low.
19  The baudrate timing detection is handled with an 18 bit counter value
20  corresponding to a specific baudrate requested by the system.
21  high to signal the Tx_engine is ready for another transfer.
22  After the timming is complete on an 11 bit transfer, the tx_rdy port will output
23  *********************************************************************************/
24  module TxEngine(Clk, Rst, data_in, tx_start, parity_en, bit8_en,
25                  odd_en, Baud_val, tx_out, tx_rdy);
26
27       input Clk, Rst;
28       input bit8_en, parity_en, odd_en, tx_start;
29       input [7:0]  data_in;
30       input [17:0] Baud_val;
31
32       output tx_out, tx_rdy;
33
34       /* PIPO Buffer register */
35       reg [7:0] pipo_buf;
36       /*Decode load buffer */
37       reg ld_buf;
38       /* PISO Data bus*/
39       reg [10:0] parallel_in;
40       /* Uart Config */
41       reg [2:0] Uart_config;
42       /* Data signals */
43       wire A, B, C, D,BTU, bc_11_done;
44
45        /*****************************************
46                     Continous  Assignments
47        *****************************************/
48       assign A = ^pipo_buf[6:0];
49       assign B = ~^pipo_buf[6:0];
50       assign C = ^pipo_buf[7:0];
51       assign D = ~^pipo_buf[7:0];
52
53       /*****************************************
54                     Procedual Combo block
55       *****************************************/
56       always @(*)  begin
57            case(Uart_config)
58                 /*7N1*/  0: parallel_in = { 1'b1, 1'b1, pipo_buf[6:0], 1'b0, 1'b1 };
```

```
59                  /*7N1*/  1: parallel_in = { 1'b1, 1'b1, pipo_buf[6:0], 1'b0, 1'b1 };
60                  /*7E1*/  2: parallel_in = { 1'b1, A   , pipo_buf[6:0], 1'b0, 1'b1 };
61                  /*7O1*/  3: parallel_in = { 1'b1, B   , pipo_buf[6:0], 1'b0, 1'b1 };
62                  /*8N1*/  4: parallel_in = { 1'b1, pipo_buf[7:0], 1'b0, 1'b1 };
63                  /*8N1*/  5: parallel_in = { 1'b1, pipo_buf[7:0], 1'b0, 1'b1 };
64                  /*8E1*/  6: parallel_in = { C   , pipo_buf[7:0], 1'b0, 1'b1 };
65                  /*8O1*/  7: parallel_in = { D   , pipo_buf[7:0], 1'b0, 1'b1 };
66                  /*7N1*/  default:parallel_in = { 1'b1,1'b1,pipo_buf[6:0],1'b0,1'b1};
67              endcase
68
69          end
70
71          /*******************************************
72                      Parallel In Serial Out Shift Reg
73           *******************************************/
74          Shiftreg_PISO piso(
75            .Clk(Clk),
76            .Rst(Rst),
77            .ld(ld_buf),
78            .sh_en(BTU),
79            .data_in(parallel_in),
80            .ser_in(1'b1),
81            .ser_out(tx_out) );
82
83          /*******************************************
84                       11 bit Counter Module
85           *******************************************/
86          count_11_bits count11(
87            .Clk(Clk),
88            .Rst(Rst),
89            .start(start_count),
90            .BTU(BTU),
91            .done(bc_11_done) );
92
93          /*******************************************
94                       Bit Time Counter Module
95           *******************************************/
96          Bit_time_counter btc(
97            .Clk(Clk),
98            .Rst(Rst),
99            .start(start_count),
100           .baud_count(Baud_val),
101           .BTU(BTU) );
102
103         /*******************************************
104                      RS Start Transmit module
105          *******************************************/
106         RSFlop start_transmit_ff(
107           .Clk(Clk),
108           .Rstb(Rst),
109           .R(bc_11_done),
110           .S(ld_buf),
111           .Q(start_count) );
112         /*******************************************
113                      RS TX Ready module
114          *******************************************/
115          RSFlop tx_ready(
116           .Clk(Clk),
```

```
117         .Rstb(Rst),
118         .R(tx_start),
119         .S(bc_11_done),
120         .Q(tx_rdy) );
121       /*******************************************
122                   Sequential block
123        *******************************************/
124       always @(posedge Clk, negedge Rst) begin
125             /*******************************
126                One shot Uart Configuration
127             *******************************/
128             if(!Rst)
129                  Uart_config <= {bit8_en, parity_en, odd_en};
130
131             /*******************************
132                Load Enable buffer register
133             *******************************/
134
135             if(!Rst)        ld_buf <= 0;            else
136                             ld_buf <= tx_start;
137
138             /*******************************
139                PIPO buffer register
140             *******************************/
141
142             if(!Rst)        pipo_buf <= 8'b0;        else
143             if(tx_start)    pipo_buf <= data_in;     else
144                             pipo_buf <= pipo_buf;
145
146       end
147   endmodule
148
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ================================================================================
4   * File: Shiftreg_Piso.v
5   ================================================================================
6   * A Parallel IN Serial OUT shift register used to transmit an 11bit data bus from
7   * from the system. Active on every rising edge of the Clk or falling edge of a
8   * Rst bit, data is cleared on an active low reset. Loads new data on a ld assertion
9   * and on a shift enable signal, data is shifted to the right as the serial in signal
10  * is shifted into the MSB.
11  *
12  *********************************************************************************/
13  module Shiftreg_PISO(Clk, Rst, ld, sh_en, data_in, ser_in, ser_out );
14       input Clk, Rst;
15       input ld, sh_en, ser_in;
16       input [10:0] data_in ;
17       output ser_out;
18
19       reg [10:0] transmit_data;
20
21  /*************Continous assignment*****************/
22       assign ser_out = transmit_data[0];
23
24  /*************Sequential block*****************/
25       always @(posedge Clk, negedge Rst)
26            if(!Rst)  transmit_data <= 10'b0;   else
27            if(ld)    transmit_data <= data_in; else
28            if(sh_en) transmit_data <= { ser_in, transmit_data[10:1] } ;
29
30  endmodule
31
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ================================================================================
4   * File: count_11_bits.v
5   ================================================================================
6   * The count 11 bits module determines the number of transfered bits during an
7   * operation. The done flag will signal when 11 bits have succesfully tranfered based
8   * on the BTU(bit time up) flag from the system.
9   * On Reset, count is set to 11 in order to signal a done to the system.
10  *
11  *********************************************************************************/
12  module count_11_bits(Clk, Rst, start, BTU, done);
13       input Clk, Rst;
14       input start, BTU;
15       output done;
16
17       reg [3:0] count;
18       assign done = (count == 4'hB);
19
20       /**********************Sequential Block**********************/
21       always @(posedge Clk, negedge Rst)
22            if(!Rst)          count <= 4'hB;         else
23            if(!start)        count <= 4'b0;         else
24            if(!BTU)          count <= count;         else
25                              count <= count +1;
26
27  endmodule
28
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ==================================================================================
4   * File: Bit_time_counter.v
5   ==================================================================================
6   * The Bit time counter module generates a single clock pulse to signal the end of
7   * a bit time transfer. A bit time for the system is determined by the baud_count
8   * wire, an 18 bit wide bus. The following table determines signal handling logic
9   *
10  *
11  *********************************************************************************/
12  module Bit_time_counter(Clk, Rst, start, baud_count, BTU);
13       input Clk, Rst;
14       input start;
15       input [17:0] baud_count;
16       output BTU;
17
18       reg [17:0] count;
19       assign BTU = (count == (baud_count - 1));
20
21       /*********************Sequential Block**********************/
22       always @(posedge Clk, negedge Rst)
23            if(!Rst)          count <= 18'b0;         else
24            if(start & !BTU) count <= count + 1;     else
25                             count <= 18'b0;
26
27  endmodule
28
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ================================================================================
4   * File: Baud_val_Decoder.v
5   ================================================================================
6      The Baudrae Value Decoder module will determine the magnitude of the count value
7   used throughout the transmit engine. A 4 bit baude select value is recieved from
8   the system and used to select its corresponding Count value as shown in the table
9   below. The baud value is an 18 bit number and is set one time only on reset.
10
11  *********************************************************************************
12                        Sel  BR             BT     # CLKS
13                       ==================================
14                        0    300         3.33E-03  166667
15                        1    600         1.67E-03  83333
16                        2    1200        8.33E-04  41667
17                        3    2400        4.17E-04  20833
18                        4    4800        2.08E-04  10417
19                        5    9600        1.04E-04  5208
20                        6    19200       5.21E-05  2604
21                        7    38400       2.60E-05  1302
22                        8    76800       1.30E-05  651
23                        9    153600      6.51E-06  326
24                        10   307200      3.26E-06  163
25                        11   614400      1.63E-06  81
26                        12   1228800     8.14E-07  41
27
28  *********************************************************************************/
29  module Baud_val_Decoder(Clk, Rst, Baudsel, Baud_val);
30    input      Clk, Rst;
31    input      [3:0] Baudsel;
32    output reg [17:0] Baud_val;
33    /*****************************************
34               Decode Sequential block
35    *****************************************/
36    always @(posedge Clk, negedge Rst)
37       if(!Rst)
38            case(Baudsel)
39                 0:  Baud_val = 166667;
40                 1:  Baud_val = 83333;
41                 2:  Baud_val = 41667;
42                 3:  Baud_val = 20833;
43                 4:  Baud_val = 10417;
44                 5:  Baud_val = 5208;
45                 6:  Baud_val = 2604;
46                 7:  Baud_val = 1302;
47                 8:  Baud_val = 868;
48                 9:  Baud_val = 434;
49                 10: Baud_val = 217;
50                 11: Baud_val = 109;
51                 12: Baud_val = 54;
52                 default: Baud_val = 166667;
53            endcase
54  endmodule
55
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   ==================================================================================
4   * File: strobe_decode.v
5   ==================================================================================
6   *
7   *  The decode module is used to interface with the picoblaze processor's port id and
8   *  rd/wr strobes. The decoder is used to select and activate particular modules
9   *  used throughout this project. By decoding the port ID, the system can determine
10  *  which module is currenty INPUTTING or OUTPUTTING to the picoblaze.
11  *
12  *********************************************************************************/
13  module strobe_decode(port_id, rd_in, wr_in, rd_out, wr_out);
14
15     input      rd_in, wr_in ;
16     input      [7:0] port_id;
17     output reg [255:0] rd_out, wr_out;
18    /*****************************************
19                  Procedual Combo block
20    *****************************************/
21     always @(*)    begin
22     /* Clear all previous values */
23        rd_out = 8'b0;
24        wr_out = 8'b0;
25     /* Assign current values */
26        rd_out[port_id] = rd_in ;
27        wr_out[port_id] = wr_in;
28        end
29
30  endmodule
31
```

```
1   `timescale 1ns / 1ps
2   /*********************************************************************************
3   * File: AISO.v
4   * The AISO module was taken from a paper on Asynchronous & Synchronous Reset
5     Design Techniques by Clifford E. Cummings, Don Mills, and Steve Golson.
6     This AISO has small modifications to the original paper in order to meet
7     specifications presented in the outline of this project.
8
9   * Async IN Sync OUT. Used to synchronize a hardware reset for the system.
10  * This module uses 2 flip-flop registers to remove metastability which may occur
11  * if the setup time for the first ff is violated from a hardware mechanical debounce.
12  * Ultimitely this module will synchronize the system with an active low reset.
13  *     i.e If a mech. reset is asserted, the output will traverse a 0 through ff1, and
14  *     on the next clk, output a 0.
15  *     else the output remains 0.
16  *********************************************************************************/
17  module ASyncIn_SyncOut(Clk, async_rst, sync_rst);
18     input       Clk, async_rst;
19     output reg  sync_rst;
20     reg         ff1;
21
22  /*************Conditional block*****************/
23     always @(posedge Clk, negedge async_rst)
24        if(async_rst == 1'b0) ff1 <= 1'b1;
25        else                  ff1 <= 1'b0;
26
27     always @(posedge Clk)
28        sync_rst <= ff1;
29
30  endmodule
31
```