TRAMELBLAZE SOPC-F2018

Chip Design Specification

Jacob Nguyen

Version 1.0

CECS 460

12/10/2018

TABLE OF CONTENTS

I. INTRODUCTION

Both microcontrollers and FPGAs are excellent at implementing practically any kind of digital logic. However, both come with their own advantages in performance and use of resources.

Microcontrollers are adept at performing control functions and applications (state machines), especially when requirements are constantly changing. The FPGA resources required to implement the microcontroller remain relatively constant, even when requirements grow in complexity, making it a suitable cost-efficient option. Programming a microcontroller using assembly code is typically easier than creating similar structures in FPGA logic.

While microcontrollers continue to be versatile, they are often limited by their performance. Microcontroller instructions execute sequentially and as application complexity grows, so does the number of instructions required to implement it, thus decreasing system performance. In contrast, FPGAs are more flexible in performance and execution. Not only can FPGAs perform algorithms sequentially but also in parallel. Parallel implementation is undoubtedly faster but unlike microcontrollers, growing complexity requires the consumption of more FPGA resources.

The chip design to be discussed provides an embedded microcontroller that can be used for a variety of target FPGA families. Its specialty is being capable of providing the low cost of using a microcontroller and the high performance of FPGAs. For example, microcontrollers can implement control functions and state machines while FPGAs can perform data path related tasks.

The chip design uses an embedded microcontroller dubbed the TramelBlaze. The TramelBlaze microcontroller is a fully embedded 16-bit CISC microcontroller based on the PicoBlaze™ architecture optimized for Xilinx FPGA families including Spartan-3E, Spartan 6, and Artix 7. The TramelBlaze is capable of being fully embedded into a target FPGA and provides extreme flexibility in interfacing I/O and digital logic in an FPGA. Because the TramelBlaze is delivered as synthesizable Verilog source code, the core logic can also be implemented into future FPGA architectures, eliminating the chance of it becoming obsolete.

Why use the TramelBlaze?

	TramelBlaze Microcontroller	FPGA Logic
Strengths	• Easy to program, excellent for control and state machine applications • Resource requirements remain relatively constant with increasing complexity • Reuses logic resources, excellent for low-performance functions	• Significantly higher performance • Excellent at parallel operations • Sequential vs. parallel implementation trade-offs optimize performance or cost • Fast response to multiple, simultaneous inputs
Weaknesses	• Executes sequentially • Performance decreases with increasing complexity • Program memory requirements increase with increasing complexity • Slower response to simultaneous inputs	• Control and state machines more difficult to program • Logic resources grow with increasing complexity

Again, having an embedded microcontroller like the TramelBlaze within a FPGA provides the strengths of both the TramelBlaze and FPGA logic.

II. DOCUMENTS

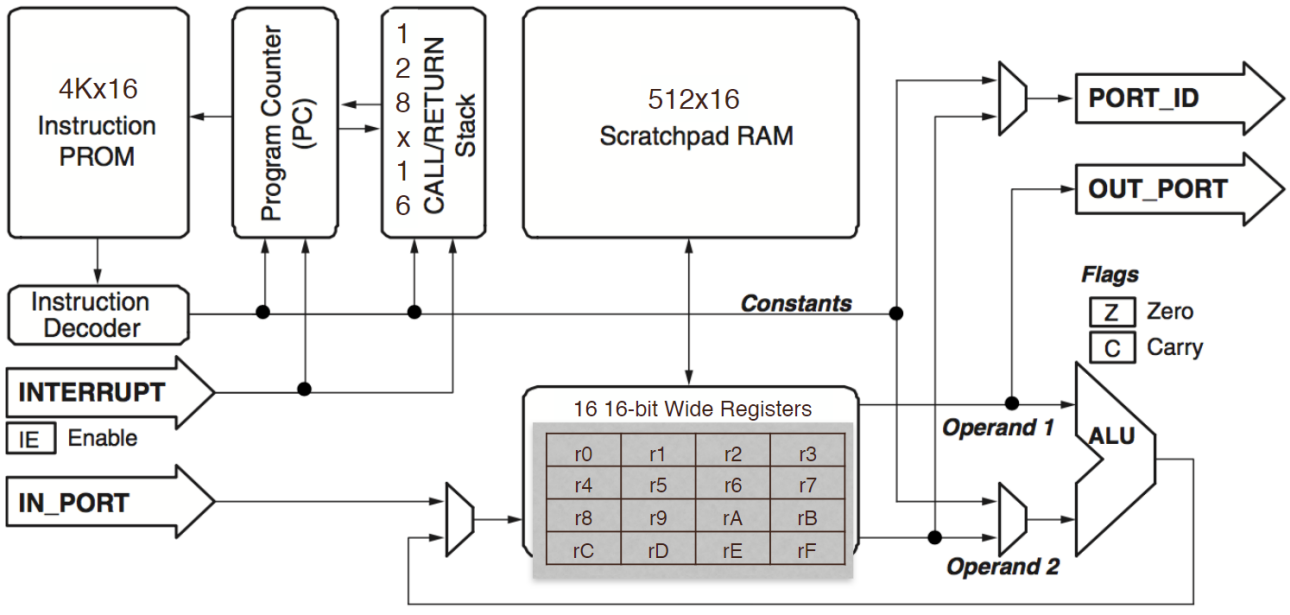
A. Applicable External documents

"TramelBlaze.pdf", "PicoBlaze™ User Guide"

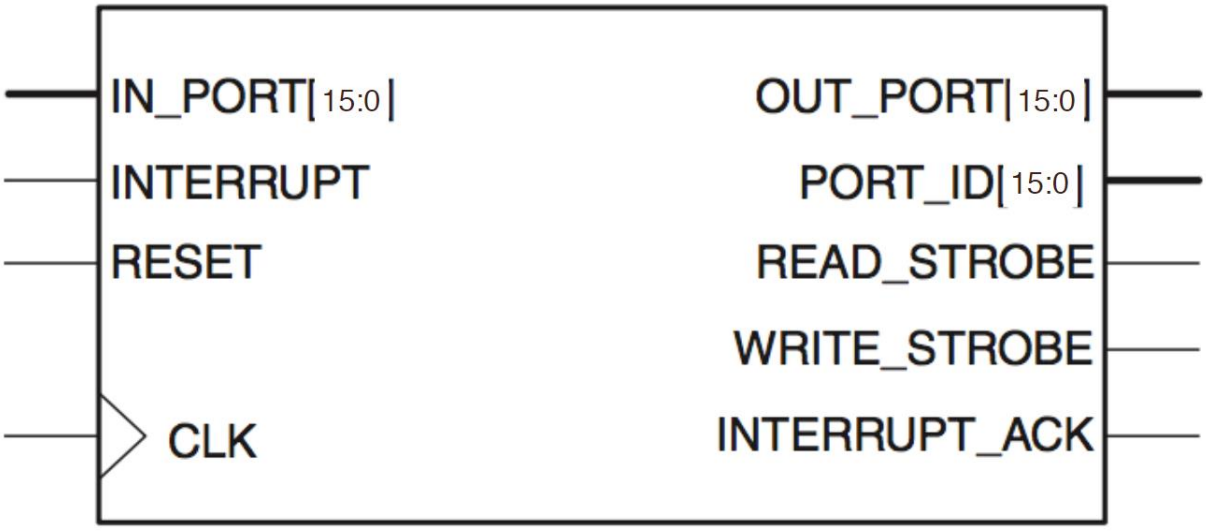
The TramelBlaze is essentially a 16-bit emulator to the PicoBlaze™. It was designed for the purpose of supporting later families of the Xilinx FPGA boards such as Artix 7 (PicoBlaze™ supports Spartan 3 and Spartan 6). It is delivered as Verilog source code and comes with a Python assembler for developing and implementing software for the TramelBlaze. The TramelBlaze supports the following features:

- 16 16-bit registers, r0 through rf **OR** s0 through sf interchangeably
- 4K Instruction ROM, stores one or two-word instructions, may be built for simulation and synthesis
- 16-bit Arithmetic Logic Unit (ALU) with CARRY and ZERO flags
- 512x16 Scratchpad RAM with FETCH and STORE capabilities
- 65536 input and 65536 output ports, allowing it to interface TramelBlaze registers to surrounding FPGA logic
- Program Counter (PC) to iterate through instructions, to be incremented upon fetching each instruction
- Single-interrupt servicing protocol, requires INTERRUPT_ENABLE flag to be set

TramelBlaze Arhitecture (16-bit emulator of 8-bit PicoBlaze™)



TramelBlaze Top Level Block Diagram



TramelBlaze Interface Signal Descriptions

Signal	Direction	Description
IN_PORT[15:0]	Input	**Input Data Port**: Present valid input data on this port during an INPUT instruction. The data is captured on the rising edge of CLK.
INTERRUPT	Input	**Interrupt Input**: If the INTERRUPT_ENABLE flag is set by the application code, generate an INTERRUPT Event by asserting this input High for at least two CLK cycles. If the INTERRUPT_ENABLE flag is cleared, this input is ignored.
RESET	Input	**Reset Input**: To reset the TramelBlaze microcontroller and to generate a RESET Event, assert this input High for at least one CLK cycle. A Reset Event is automatically generated immediately following FPGA configuration.
CLK	Input	**Clock Input**: Clock signal is generated via FPGA constraint file. All synchronous elements are clocked from the rising edge of the clock.
OUT_PORT[15:0]	Output	**Output Data Port**: Output data appears on this port for two CLK cycles during an OUTPUT instruction. Capture output data within the FPGA at the rising CLK edge when WRITE_STROBE is High.

Signal	Direction	Description
READ_STROBE	Output	**Read Strobe**: When asserted High, this signal indicates that input data on the IN_PORT[7:0] port was captured to the specified data register during an INPUT instruction. This signal is asserted on the second CLK cycle of the two-cycle INPUT instruction. This signal is typically used to acknowledge read operations from FIFOs.
WRITE_STROBE	Output	**Write Strobe**: When asserted High, this signal validates the output data on the OUT_PORT[7:0] port during an OUTPUT instruction. This signal is asserted on the second CLK cycle of the two-cycle OUTPUT instruction. Capture output data within the FPGA on the rising CLK edge when WRITE_STROBE is High.
INTERRUPT_ACK	Output	**Interrupt Acknowledge**: When asserted High, this signal acknowledges that an INTERRUPT Event occurred. This signal is asserted during the second CLK cycle of the two-cycle INTERRUPT Event. This signal is optionally used to clear the source of the INTERRUPT input.

TramelBlaze Top Level Instantiation (with Full UART implementation)

```
tramelblaze_top tbt(
    .CLK(clk_TSI),
    .RESET(reset_s),
    .IN_PORT({8'b0, UART_DATA}),
    .INTERRUPT(intr),
    .OUT_PORT(OUT_PORT),
    .PORT_ID(PORT_ID),
    .READ_STROBE(READ_STROBE),
    .WRITE_STROBE(WRITE_STROBE),
    .INTERRUPT_ACK(inta)
);
```

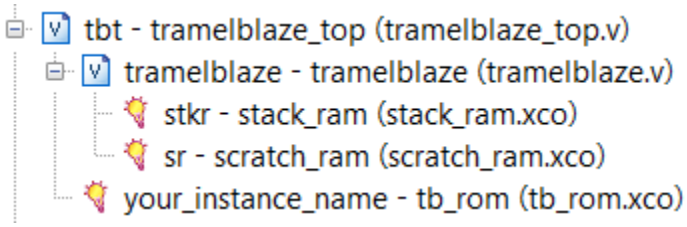
TramelBlaze Instantiation within TramelBlaze_top

Both processor and instruction memory are instantiated here.

```
tramelblaze tramelblaze
      (
      .CLK(CLK),
      .RESET(RESET),
      .IN_PORT(IN_PORT),
      .INTERRUPT(INTERRUPT),

      .OUT_PORT(OUT_PORT),
      .PORT_ID(PORT_ID),
      .READ_STROBE(READ_STROBE),
      .WRITE_STROBE(WRITE_STROBE),
      .INTERRUPT_ACK(INTERRUPT_ACK),

      .ADDRESS(ADDRESS),
      .INSTRUCTION(INSTRUCTION)
      );

tb_rom your_instance_name
      (
      .clka(CLK),                    // input clka
      .addra(ADDRESS),               // input [11 : 0] addra
      .douta(INSTRUCTION)            // output [15 : 0] douta
      );
```

Hierarchy for TramelBlaze instantiation

Required files include all three memory blocks: stack RAM, scratch RAM, executable instruction ROM.



TramelBlaze Instruction Set

Instruction	Description	Function	ZERO	CARRY
ADD sX, kk	Add register sX with literal kk	sX <- sX + kk	?	?
ADD sX, sY	Add register sX with register sY	sX <- sX + sY	?	?
ADDCY sX, kk (ADDC)	Add register sX with literal kk with CARRY bit	sX <- sX + kk + CARRY	?	?
ADDCY sX, sY (ADDC)	Add register sX with register sY with CARRY bit	sX <- sX + sY + CARRY	?	?
AND sX, kk	Bitwise AND register sX with literal kk	sX <- sX AND kk	?	0
AND sX, sY	Bitwise AND register sX with register sY	sX <- sX AND sY	?	0
CALL aaa	Unconditionally call subroutine at aaa	TOS <- PC PC <- aaa	-	-
CALL C aaa	If CARRY flag set, call subroutine at aaa	If CARRY=1, {TOS <- PC, PC <- aaa}	-	-
CALL NC aaa	If CARRY flag not set, call subroutine at aaa	If CARRY=0, {TOS <- PC, PC <- aaa}	-	-
CALL NZ aaa	If ZERO flag not set, call subroutine at aaa	If ZERO=0, {TOS <- PC, PC <- aaa}	-	-
CALL Z aaa	If ZERO flag set, call subroutine at aaa	If ZERO=1, {TOS <- PC, PC <- aaa}	-	-
COMPARE sX, kk (COMP)	Compare register sX with literal kk. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If sX=kk, ZERO <- 1 If sX<kk, CARRY <- 1	?	?
COMPARE sX, sY (COMP)	Compare register sX with register sY. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If sX=sY, ZERO <- 1 If sX<sY, CARRY <- 1	?	?
DISABLE INTERRUPT (DINT)	Disable interrupt input	INTERRUPT_ENABLE <- 0	-	-
ENABLE INTERRUPT (EINT)	Enable interrupt input	INTERRUPT_ENABLE <- 1	-	-
Interrupt Event	Asynchronous interrupt input. Preserve flags and PC. Clear INTERRUPT_ENABLE flag. Jump to interrupt vector at address 3FF.	Preserved ZERO <- ZERO Preserved CARRY <- CARRY INTERRUPT_ENABLE <- 0 TOS <- PC PC <- 3FF	-	-

Instruction	Description	Function	ZERO	CARRY
FETCH sx, (sY) (FETCH sX, sY)	Read scratchpad RAM location pointed to by register sY into register sX	sX <- RAM[(sY)]	-	-
FETCH sX, ss	Read scratchpad RAM location ss into register sX	sX <- RAM[ss]	-	-
INPUT sx, (sY) (INPUT sX, sY)	Read value on input port location pointed to by register sY into register sX	PORT_ID <- sY sX <- IN_PORT	-	-
INPUT sX, pp (INPUT sX, pp)	Read value on input port location pp into register sX	PORT_ID <- pp sX <- IN_PORT	-	-
JUMP aaa	Unconditionally jump to aaa	PC <- aaa	-	-
JUMP C aaa	If CARRY flag set, jump to aaa	If CARRY=1, PC <- aaa	-	-
JUMP NC aaa	If CARRY flag not set, jump to aaa	If CARRY=0, PC <- aaa	-	-
JUMP NZ aaa	If ZERO flag not set, jump to aaa	If ZERO=0, PC <- aaa	-	-
JUMP Z aaa	If ZERO flag set, jump to aaa	If ZERO=1, PC <- aaa	-	-
LOAD sX, kk	Load register sX with literal kk	sX <- kk	-	-
LOAD sX, sY	Load register sX with register sY	sX <- sY	-	-
OR sX, kk	Bitwise OR register sX with literal kk	sX <- sX OR kk	?	0
OR sX, sY	Bitwise OR register sX with register sY	sX <- sX OR sY	?	0
OUTPUT sX, (sY) (OUTPUT sX, sY)	Write register sX to output port location pointed to by register sY	PORT_ID <- sY OUT_PORT <- sX	-	-
OUTPUT sX, pp (OUTPUT sX, pp)	Write register sX to output port location pp	PORT_ID <- pp OUT_PORT <- sX	-	-
RETURN	Unconditionally return from subroutine	PC <- TOS+1	-	-
RETURN C	If CARRY flag set, return from subroutine	If CARRY=1, PC <- TOS+1	-	-
RETURN NC	If CARRY flag not set, return from subroutine	If CARRY=0, PC <- TOS+1	-	-
RETURN NZ	If ZERO flag not set, return from subroutine	If ZERO=0, PC <- TOS+1	-	-
RETURN Z	If ZERO flag set, return from subroutine	If ZERO=1, PC <- TOS+1	-	-
RETURNI DISABLE (RETI DISABLE)	Return from interrupt service routine. Interrupt remains disabled.	PC <- TOS ZERO <- Preserved ZERO CARRY <- Preserved CARRY INTERRUPT_ENABLE <- 0	?	?
RETURNI ENABLE (RETI ENABLE)	Return from interrupt service routine. Re-enable interrupt.	PC <- TOS ZERO <- Preserved ZERO CARRY <- Preserved CARRY INTERRUPT_ENABLE <- 1	?	?
RL sX	Rotate register sX left	sX <- {sX[6:0],sX[7]} CARRY <- sX[7]	?	?
RR sX	Rotate register sX right	sX <- {sX[0],sX[7:1]} CARRY <- sX[0]	?	?
SL0 sX	Shift register sX left, zero fill	sX <- {sX[6:0],0} CARRY <- sX[7]	0	?
SL1 sX	Shift register sX left, one fill	sX <- {sX[6:0],1} CARRY <- sX[7]	?	?

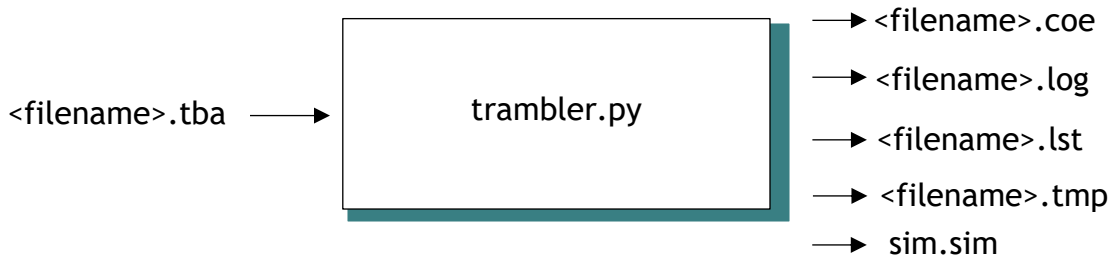
Instruction	Description	Function	ZERO	CARRY
SLA sX	Shift register sX left through all bits, including CARRY	sX <- {sX[6:0],CARRY} CARRY <- sX[7]	?	?
SLX sX	Shift register sX left. Bit sX[0] is unaffected.	sX <- {sX[6:0],sX[0]} CARRY <- sX[7]	?	?
SR0 sX	Shift register sX right, zero fill	sX <- {0,sX[7:1]} CARRY <- sX[0]	?	?
SR1 sX	Shift register sX right, one fill	sX <- {1,sX[7:1]} CARRY <- sX[0]	0	?
SRA sX	Shift register sX right through all bits, including CARRY	sX <- {CARRY,sX[7:1]} CARRY <- sX[0]	?	?
SRX sX	Arithmetic shift register sX right. Sign extend sX. Bit sX[7] Is unaffected.	sX <- {sX[7],sX[7:1]} CARRY <- sX[0]	?	?
STORE sX, (sY) (STORE sX, sY)	Write register sX to scratchpad RAM location pointed to by register sY	RAM[(sY)] <- sX	-	-
STORE sX, ss	Write register sX to scratchpad RAM location ss	RAM[ss] <- sX	-	-
SUB sX, kk	Subtract literal kk from register sX	sX <- sX – kk	?	?
SUB sX, sY	Subtract register sY from register sX	sX <- sX – sY	?	?
SUBCY sX, kk	Subtract literal kk from register sX with CARRY (borrow)	sX <- sX – kk - CARRY	?	?
SUBCY sX, sY	Subtract register sY from register sX with CARRY (borrow)	sX <- sX – sY - CARRY	?	?
TEST sX, kk	Test bits in register sX against literal kk. Update CARRY and ZERO flags. Registers are unaffected.	If (sX AND kk) = 0, ZERO <- 1 CARRY <- odd parity of (sX AND kk)	?	?
TEST sX, sY	Test bits in register sX against register sX. Update CARRY and ZERO flags. Registers are unaffected.	If (sX AND sY) = 0, ZERO <- 1 CARRY <- odd parity of (sX AND kk)	?	?
XOR sX, kk	Bitwise XOR register sX with literal kk	sX <- sX XOR kk	?	0
XOR sX, sY	Bitwise XOR register sX with register sY	sX <- sX XOR sY	?	0

Key:

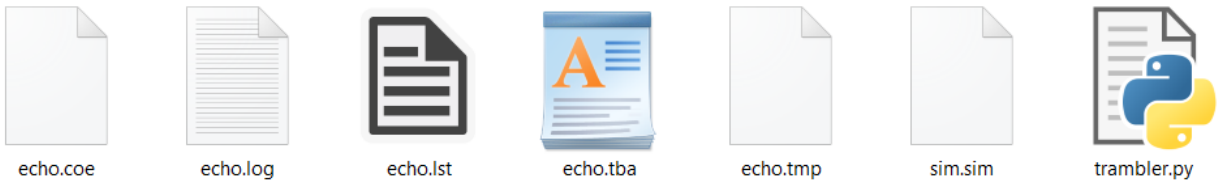
sX =	One of 16 possible register locations ranging from s0 through sF or specified as a literal
sY =	One of 16 possible register locations ranging from s0 through sF or specified as a literal
aaa =	10-bit address, specified either as a literal or a three-digit hexadecimal value ranging from 000 to 3FF or a labeled location
kk =	8-bit immediate constant, specified either as a literal or a two-digit hexadecimal value ranging from 00 to FF or specified as a literal
pp =	8-bit port address, specified either as a literal or a two-digit hexadecimal value ranging from 00 to FF or specified as a literal
ss =	6-bit scratchpad RAM address, specified either as a literal or a two-digit hexadecimal value ranging from 00 to 3F or specified as a literal
RAM[n] =	Contents of scratchpad RAM at location n
TOS =	Value stored at Top Of Stack

Programming the TramelBlaze

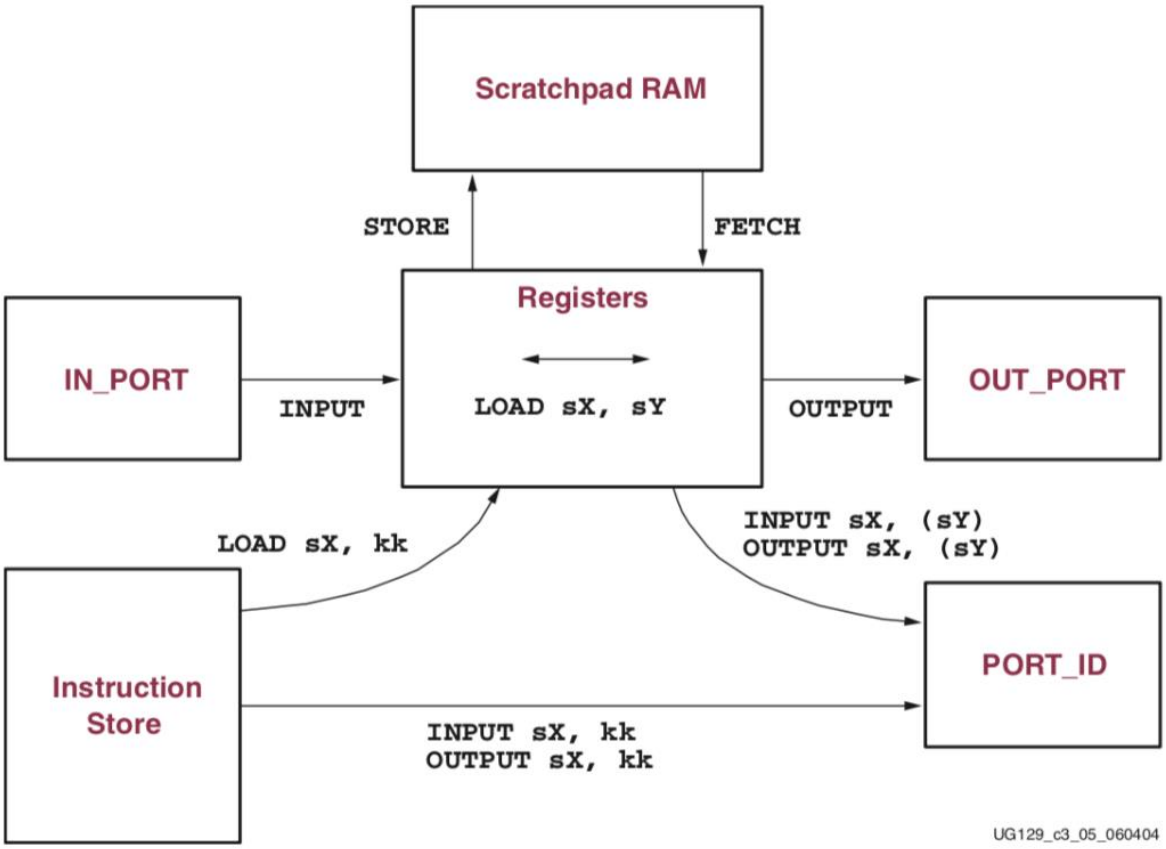
In order to utilize the TramelBlaze instruction set, the TramelBlaze also comes with a Python assembler (trambler.py) to develop and implement software. The software is in the form of assembly code. This assembly code is written in a text file (.tba) and uses TramelBlaze syntax. Programs are best written on any kind of Notepad or Wordpad tool that is available on most Windows computers. When assembled, the code is formatted into a .coe file, a format specifically for initializing the instruction ROM in a Verilog design flow. The assembler also produces a log file, stating whether assembly is successful or that errors have occurred.



Example Directory After Successful Assembly



Data Movement Instructions Within the TramelBlaze



B. Applicable Internal Documents

N/A

III. REQUIREMENTS

With knowledge on the TramelBlaze microcontroller, the requirements in “Project 3 F18.pdf” states that the microcontroller is to be instantiated in a SOPC design to implement a full UART. The design is developed within the Xilinx development software using Verilog HDL and programmed to target FPGA board. The design should also include Technology-Specific Instantiation (TSI) for the target board family.

A full UART consists of both a transmit and receive engine as well as different UART configurations set by the user such as baud rate, eighth bit enable, parity enable, and odd or even parity. The UART allows for both engines to share the same configurations and supports full duplex communication. Both engines have one-bit “ready” signals: TX_RDY for transmit, RX_RDY for receive. TX_RDY is asserted whenever the transmit engine is ready to transmit data, and RX_RDY is asserted when it is finished receiving a full byte of data. When either of the signals are asserted, the UART module is responsible for sending an interrupt signal to the TramelBlaze to allow it to begin executing an Interrupt Service Routine (ISR) programmed through assembly code.

The module also includes supplemental logic such as a baud rate decoder to calculate regular bit time and half bit time, a mux selecting between UART data and UART status register, and TX_RDY and RX_RDY being fed through Positive Edge Detect (PED) circuits to generate an interrupt signal to the TramelBlaze.

The required software allows the design to interface with a serial terminal. The software requirements are as follows:

- Walk down a “one” through the LED’s in the main loop
- Display a banner when starting out of reset and provide a prompt
- When a key is pressed on terminal, a byte is sent to the receive engine via the RX signal and causes an interrupt; UART status register is read to determine source of interrupt (TX_RDY or RX_RDY)
- Characters received are processed as follows:
    - A <CR> should send the cursor to the start of the next line and refresh the prompt
    - A <BS> should erase the character in front of the cursor – prompt must not be erased
    - An “*” will result in outputting of the home town followed by a new line and prompt
    - An “@” will result in the outputting of the number of characters received followed by a new line and prompt
    - All other characters are echoed on the display up to the 40th character – when the 40th character has been echoed, then a new line and prompt is issued

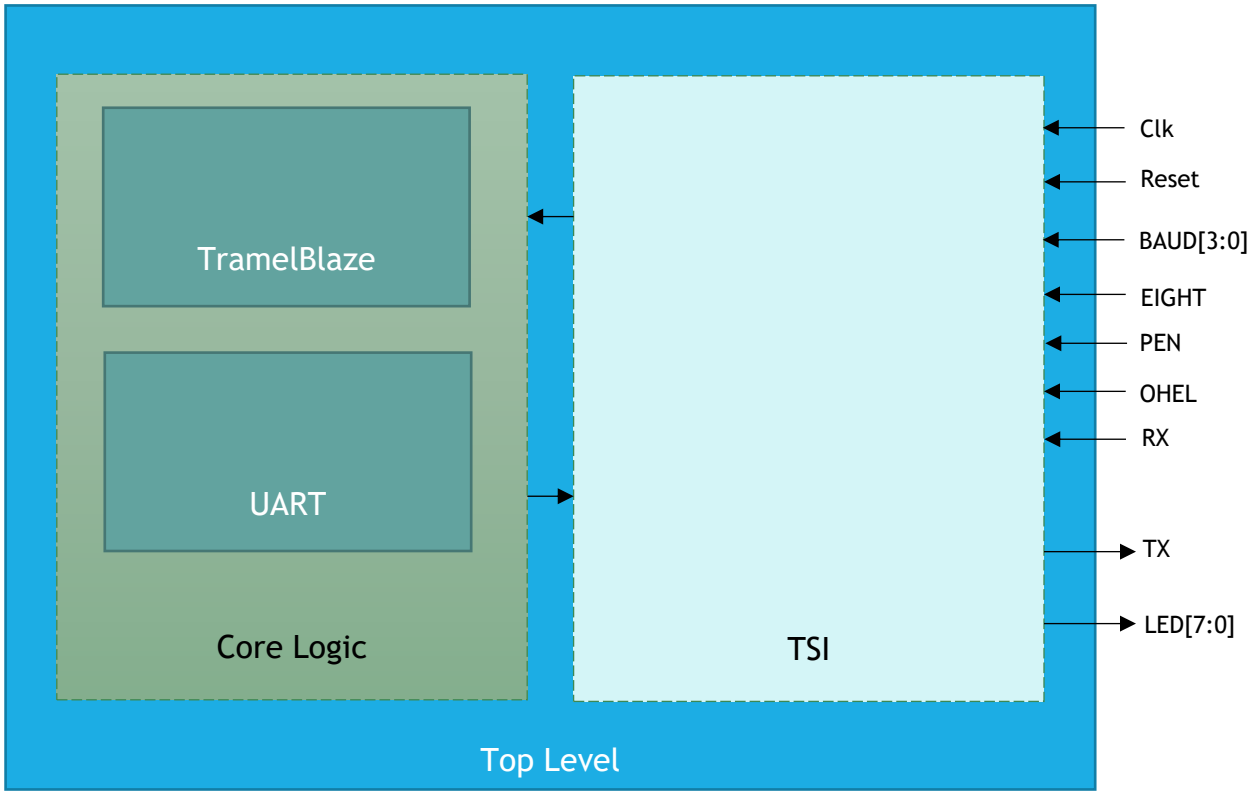
IV. TOP LEVEL DESIGN

A. Description

The top level module of the design instantiates both TSI and the core logic. The TSI ensures each I/O have a particularly selected device to meet the electrical and timing requirements of the external interface. The core logic contains the main design for the TramelBlaze embedded microcontroller and full UART implementation. The design is optimized for the Digilent Nexys 3 FPGA board of the Spartan 6 family.

I/O signal names are defined within the core logic and are assigned within the User Constraints File (.ucf or UCF) for target FPGA. Users can then interface with the design using the programable features of the FPGA (buttons, switches, etc.). In the case of the UART, users can toggle between different baud rates and set configurations such as eighth bit, parity enable, and odd/even parity.

B. Block Diagram



C. Data Flow Description

Any I/O coming into or from the core logic within the top level module must pass through the TSI block. The TSI block instantiates I/O buffers of a target library (Spartan 6) for each and every I/O to ensure the I/O meets the electrical and timing requirements of a particular the external interface.

The clock signal passes through the TSI as a global buffer (BUFG) component. This is to ensure as little clock skew as possible to hardware components that are physically farther away on the board.

Inputs to the design pass through as an input buffer (IBUF) component. This includes reset, BAUD[3:0], EIGHT, PEN, OHEL, and the Rx signal. These signals are then outputted from the TSI block to the core logic.

Outputs to the design pass through as an output buffer (OBUF) component. This includes LED[7:0] and the Tx signal. These are both outputs from the core logic to the TSI block.

The Rx input and Tx output are designed to serially interface with a terminal. Any data coming in or out is processed by the core logic and communicated to the terminal.

D. I/O

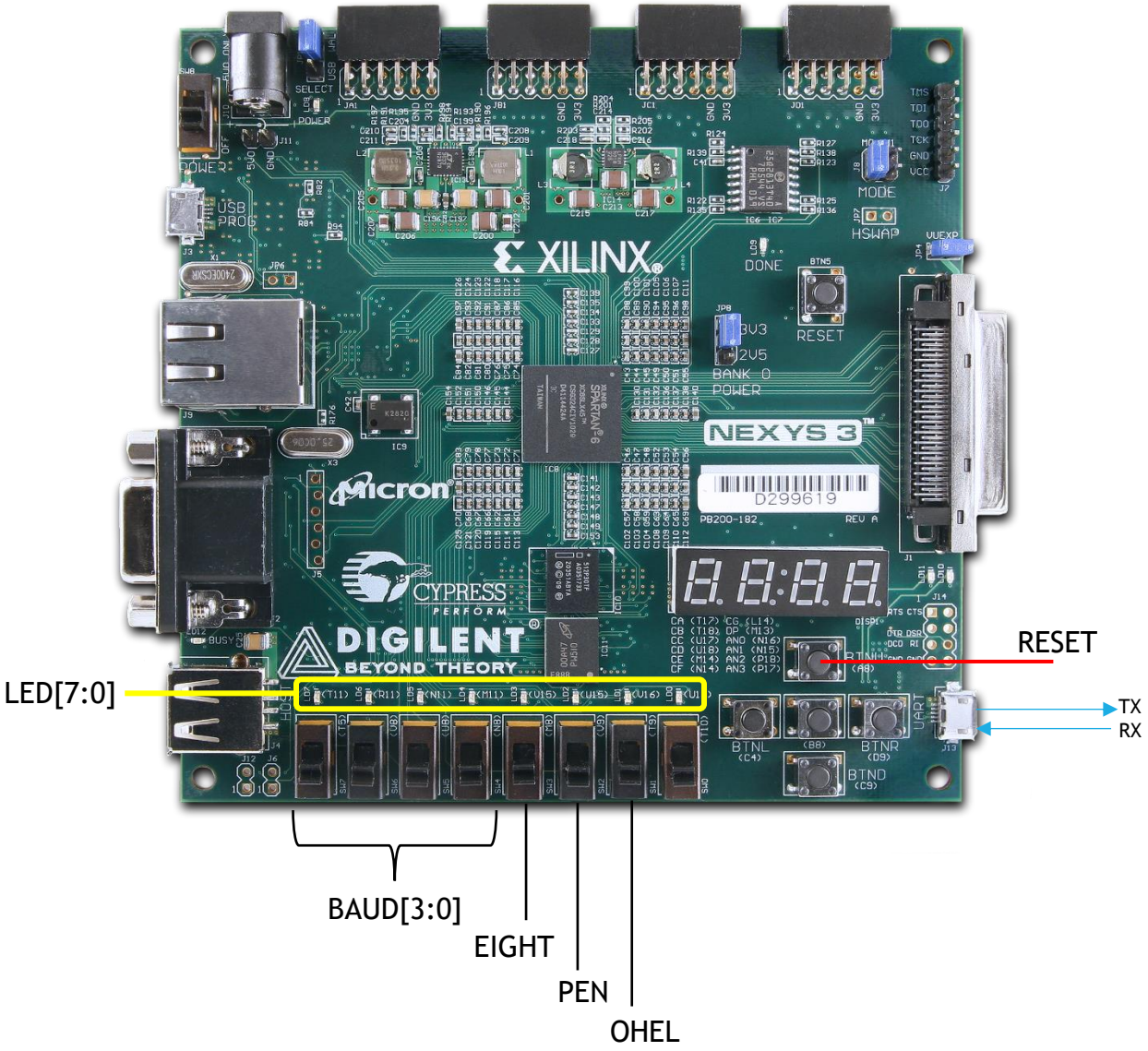
1. Signal Names

Signal	Direction	Description
Clk	Input	100MHz clock frequency
BAUD[3:0]	Input	Selectable Baud Rates with speeds (bits/sec): 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600
EIGHT	Input	Transmit/Receive engine process eight bits (seven if not asserted)
PEN	Input	Enable Parity Bit
OHEL	Input	1 for Odd Parity, 0 for Even Parity (only applicable if PEN = 1)
Reset	Input	System reset; resets design back to its initial state
RX	Input	RX input to UART
TX	Output	TX output from UART
LED[7:0]	Output	FPGA LEDs

2. Pin Assignments

Signal	Nexys 3 Pin	Output
Clk	V10	100 MHz Clock
BAUD[3]	T5	SW[7]
BAUD[2]	V8	SW[6]
BAUD[1]	U8	SW[5]
BAUD[0]	N8	SW[4]
EIGHT	M8	SW[3]
PEN	V9	SW[2]
OHEL	T9	SW[1]
RX	N17	Rx (Micro-USB)
Reset	A8	Button Up
TX	N18	Tx (Micro-USB)
LED[7]	T11	LED[7]
LED[6]	R11	LED[6]
LED[5]	N11	LED[5]
LED[4]	M11	LED[4]
LED[3]	V15	LED[3]
LED[2]	U15	LED[2]
LED[1]	V16	LED[1]
LED[0]	U16	LED[0]

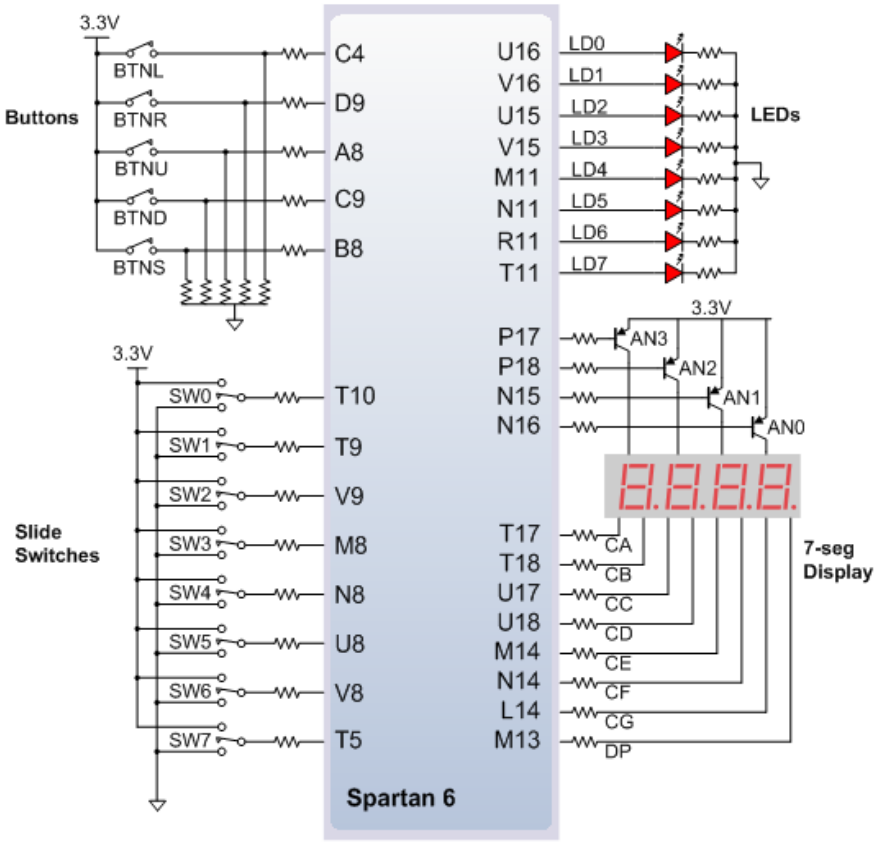
Nexys 3 Pin Assignment



3. Electrical Characteristics

The SOPC design makes use of eight individual LEDs, seven slide switches, and one push button. All I/O is to be powered by a 3.3V power source. Buttons and switches are connected to the FPGA via series resistors to prevent damage from an accidental short circuit. The pushbuttons are “momentary” switches that normally generate a low output when they are at rest, and a high output only when they are pressed. Slide switches generate constant high or low inputs depending on their position. LEDs are anode-connected to the FPGA via 390-ohm resistors, so they will turn on when a logic high voltage is applied to their respective I/O pin.

Basic I/O Circuit of the Nexys 3



E. Clocks

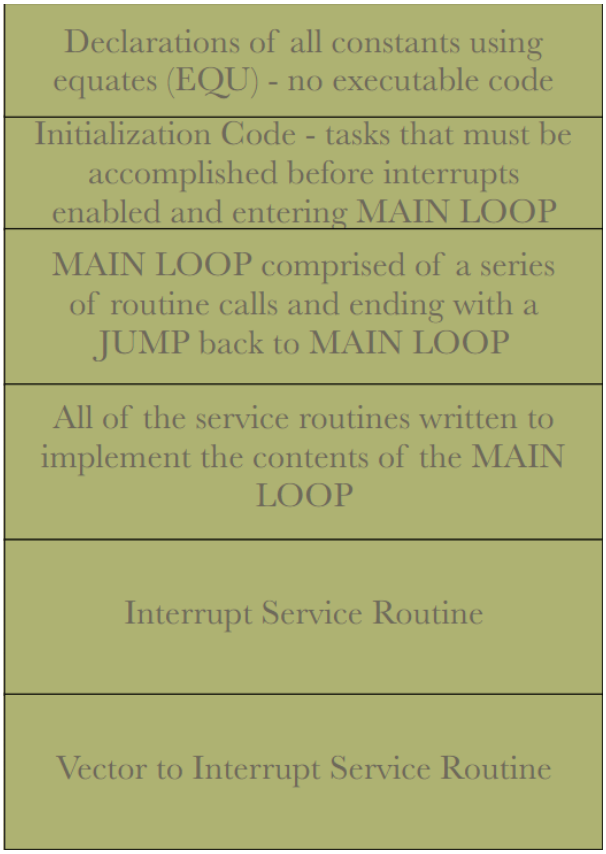
The specified system clock for the SOPC design is 100 MHz via the fixed-frequency oscillator of the Nexys 3 FPGA board. The same frequency is also applied to the test fixture for verification purposes and is defined in the UCF file for when a bit file is to be generated for testing. All synchronous elements such as registers and memory blocks are clocked on the rising clock edge.

F. Resets

In the event of a reset, the SOPC design responds by clearing and restarting various hardware elements and forcing the TramelBlaze to be in its initial state. The design also automatically reset immediately after the FPGA configuration process completes. The Program Counter (PC) is reset to address 0, flags are cleared, interrupts are disabled, and the CALL/RETURN stack is reset. General-purpose registers and scratchpad RAM are unaffected by reset.

G. Software

Writing the software assembly code for the TramelBlaze employs proficient practice of code organization and "top-down design." When developing the software, it is vitally important to divide the code into distinguishable sections that each serve a purpose to meeting the requirements. Using assembler directives and address labels helps clarify the method in which the software is implemented (i.e. labeled registers and subroutines). For TramelBlaze software, it is recommended that the code is broken up into six portions:



Visualized Code Organization

1. *Declaration* – define all constants and string substitutions
2. *Initialization* – all the setup required before enabling interrupts
3. *Main Loop* - routine where the processor spends most of its time. Cycling is achieved by JUMP at the end of the loop
4. *Subroutines to the Main Loop*
5. *ISR* – Interrupt Service Routine
6. *Vector to the ISR*

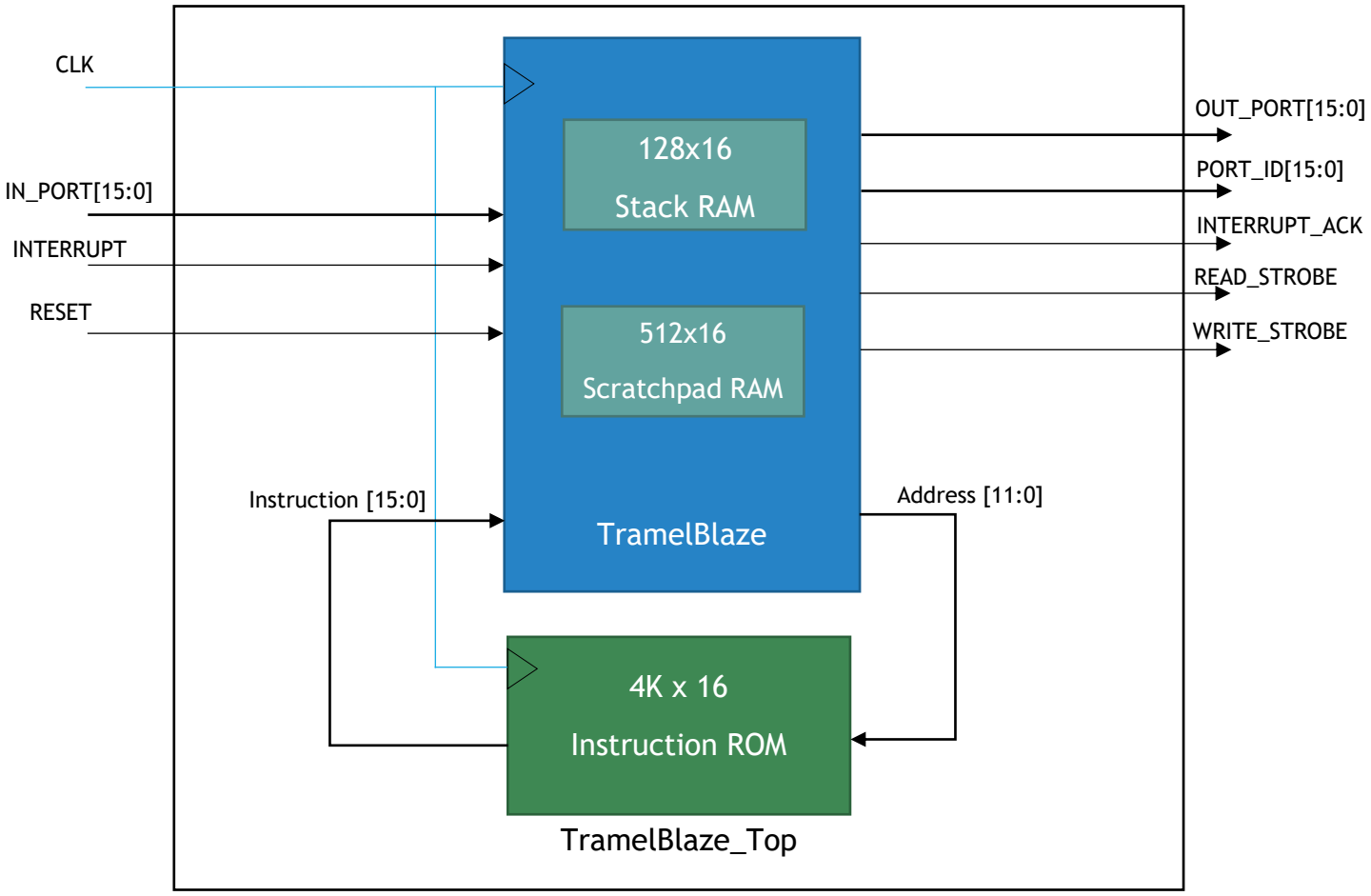
V. EXTERNALLY DEVELOPED BLOCKS

A. TramelBlaze

Description

The SOPC design incorporates a 16-bit embedded microcontroller, the TramelBlaze, based upon the 8-bit PicoBlaze™ architecture. It was developed by a third-party source for the purpose of demonstrating the concepts learned in a college course about System-On-Chip design. It has been designed to be a 16-bit PicoBlaze™ emulator with larger memory blocks. While the architecture of the microcontroller has been explained when referencing the necessary external documents associated with the design (pg. 5), it is still important to mention that it is responsible for executing the software implementation of the full UART.

Block Diagram



I/O

The Input/Output ports extend the TramelBlaze microcontroller's capabilities and allow the microcontroller to connect to a custom peripheral set or to other FPGA logic. The TramelBlaze microcontroller supports up to 65536 input ports and 65536 output ports or a combination of input/output ports. The PORT_ID output provides the port address. During an INPUT operation, the TramelBlaze microcontroller reads data from the IN_PORT port to a specified register, sX. During an OUTPUT operation, the TramelBlaze microcontroller writes the contents of a specified register, sX, to the OUT_PORT port.

Register Map

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

Register	Bit Size	Description
s0	16	Checks Rx Status
s1	16	Echo register
s2	16	Subtrahend for binary to ascii conversion
s3	16	Scratchpad RAM address
s4	16	LED register
s5	16	Scratchpad read/write register
s6	16	LED counter 1
s7	16	LED counter 2
s8	16	Character counter in terminal
s9	16	Character counter register and flag
sa	16	Banner flag
sb	16	Prompt flag
sc	16	Prompt has started flag
sd	16	Hometown flag
se	16	Digit for binary to ascii conversion
sf	16	Backspace flag

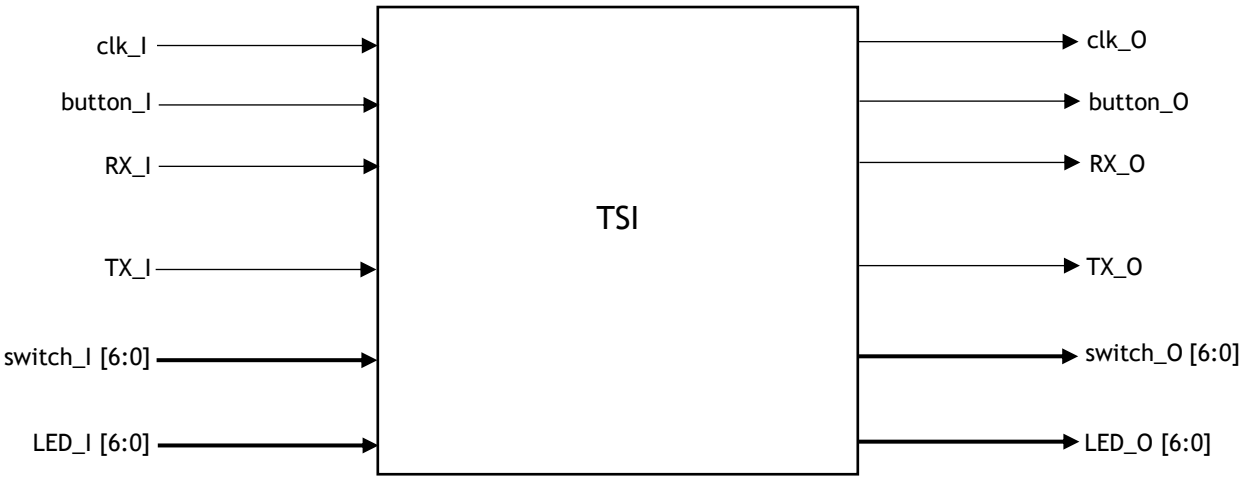
VI. INTERNALLY DEVELOPED BLOCKS

A. Technology-Specific Instantiation (TSI)

Description

The TSI block instantiates all I/O as buffer components specifically for the Spartan 6 FPGA family. The TSI block ensures the I/O are within the electrical and timing requirements of its external interface. Clock signals that pass through the TSI have their clock skew reduced for hardware that is physically farther away with the use of a global buffer. Inputs and Outputs are instantiated with input buffers (IBUF) and output buffers (OBUF) respectively.

Block Diagram



Inputs/Outputs

Signal Name	Direction	Description
clk_I	Input	100 MHz oscillator
button_I	Input	Reset button
RX_I	Input	Rx signal
TX_I	Input	Tx signal from UART
switch_I[6:0]	Input	Baud Rate, EIGHT, PEN, OHEL configuration
LED_I[7:0]	Input	LED data from core logic
clk_O	Output	100 MHz clock to core logic
button_O	Output	Reset to core logic
RX_O	Output	Rx signal to core logic
TX_O	Output	Tx signal to terminal
switch_O[6:0]	Output	Baud Rate, EIGHT, PEN, OHEL configuration to core logic
LED_O[7:0]	Output	LED output

Verification

B. Asynchronous-in Synchronous-Out (AISO)

Description

AISO is used to prevent the potential effects of metastability caused by the reset button. Due to the nature of the signal being asynchronous and affecting synchronous elements like registers, AISO synchronizes the reset signal to give registers enough setup time to meet the timing requirements. AISO uses a synchronization flop in order to sync the reset with the rest of the design.

Block Diagram

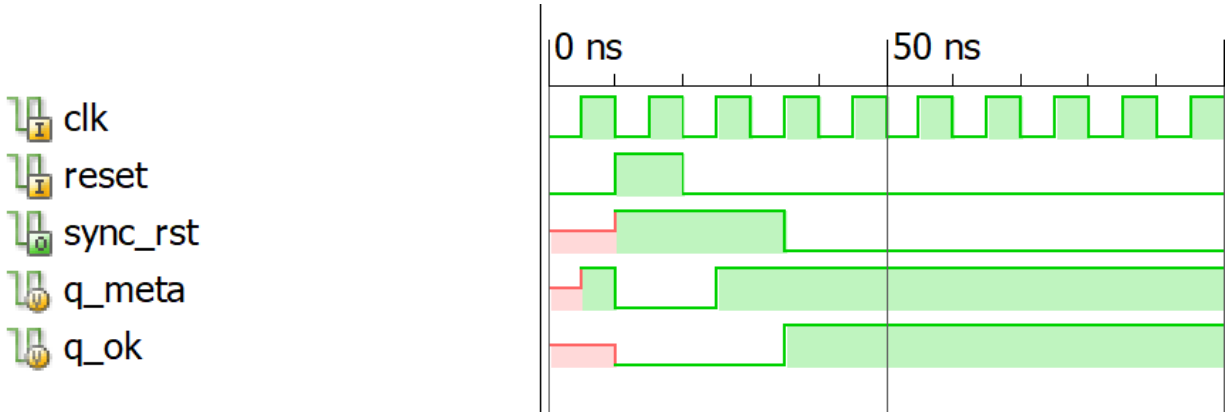
Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock frequency
reset	Input	Asynchronous reset from FPGA I/O
sync_rst	Output	Synchronized reset to core logic

Register Maps

Register	Bit Size	Description
q_meta	1	Metastable register, 0 when reset, V_{cc} otherwise
q_ok	1	Synchronization flop

Verification

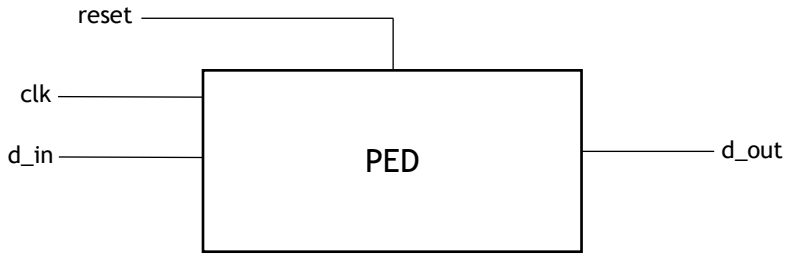


C. Positive-Edge Detect (PED)

Description

The PED block provides a one-clock-wide pulse when it detects a positive edge signal. It will only send one pulse until the input is asserted again, meaning another pulse will not be sent if the input remains high. This block is used to generate an interrupt signal to the TramelBlaze from the UART.

Block Diagram



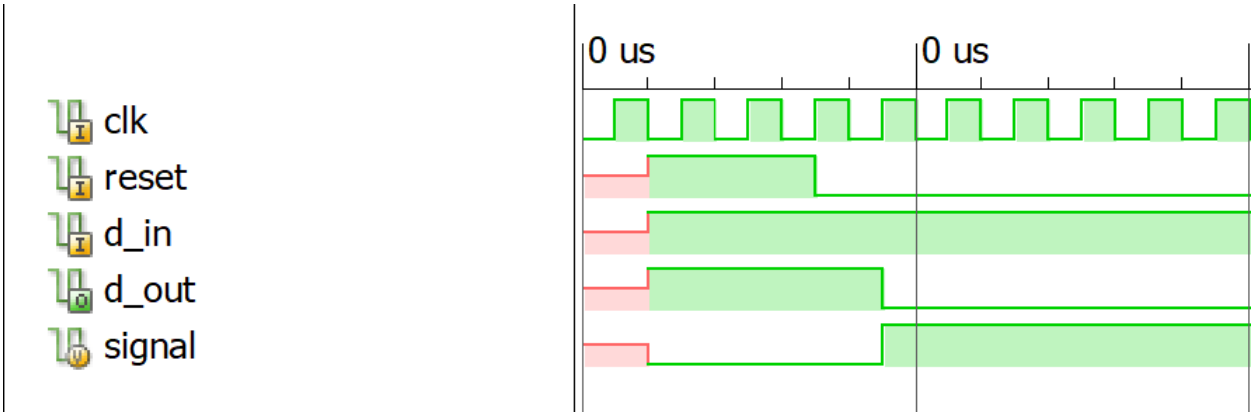
Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock
reset	Input	AISO reset
d_in	Input	Input signal from the TramelBlaze
d_out	Output	One-clock-wide pulse

Register Map

Register	Bit Size	Description
signal	1	Delay register

Verification

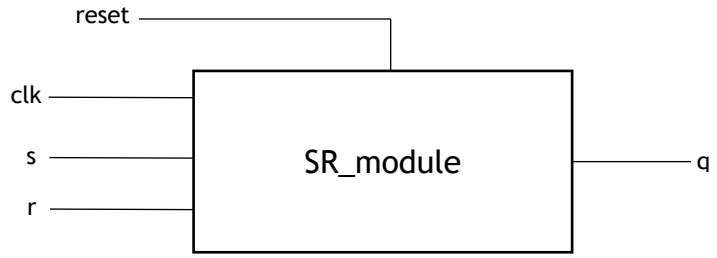


D. SR_module

Description

The SR flop has two one-bit inputs, Set and Reset. “Set” will set the output to a constant 1 and “Reset” will set it to a constant 0. The global reset signal will set it to 0 regardless of inputs.

Block Diagram



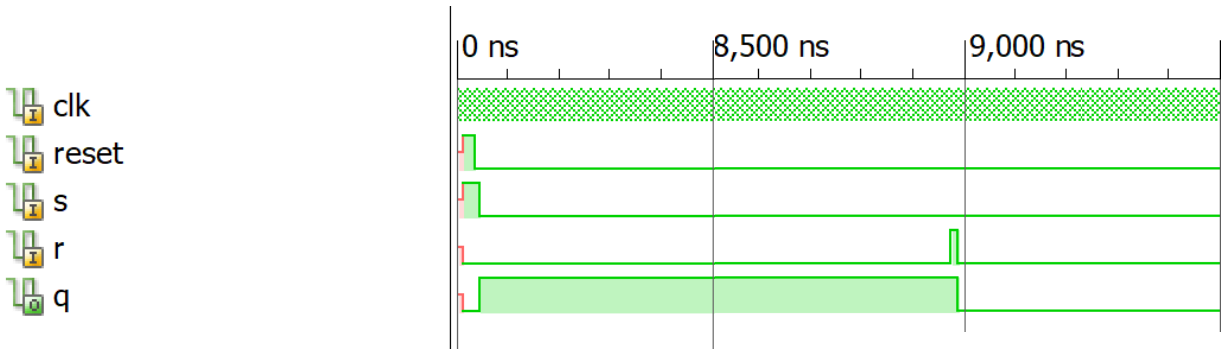
Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock
reset	Input	AISO reset
s	Input	Sets output to 1
r	Input	Resets output to 0
q	Output	SR flop output

Register Map

Register	Bit Size	Description
q	1	Register to be set or reset by inputs

Verification

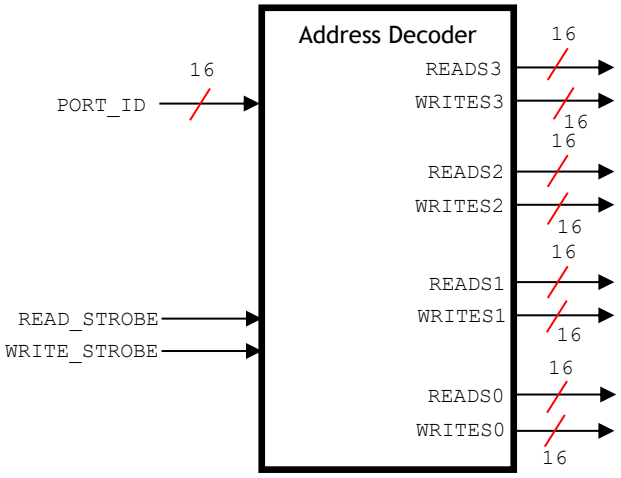


E. Address Decoder

Description

The Address Decoder block is for reading a writing data to and from the TramelBlaze. It consists of 4 16-bit Write lines and 4 16-bit Read lines for a total of 64 possible addresses to access memory. The decoder consists of to procedural blocks: one to decode the specified address with sensitivity to PORT_ID[3:0], and the second to choose which one of the 4 Write OR Read lines (depending on the Read and Write Strobe Inputs) with sensitivity to PORT_ID[15:14].

Block Diagram



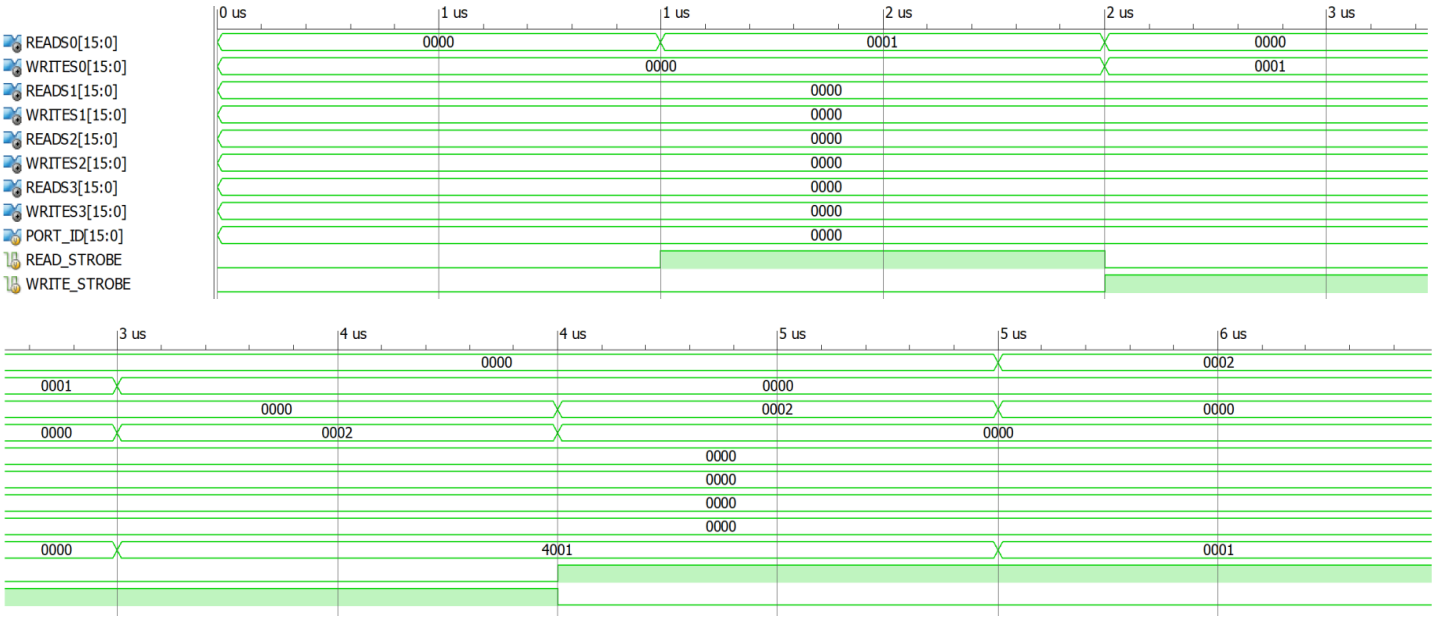
Inputs/Outputs

Signal Name	Direction	Description
PORT_ID[15:0]	Input	PORT ID signal from TramelBlaze
READ_STROBE	Input	Read Strobe from TramelBlaze
WRITE_STROBE	Input	Write Signal from TramelBlaze
READS3 [15:0]	Output	Read output signal of PORT_ID[15:14] = 11
READS2 [15:0]	Output	Read output signal of PORT_ID[15:14] = 10
READS1 [15:0]	Output	Read output signal of PORT_ID[15:14] = 01
READS0 [15:0]	Output	Read output signal of PORT_ID[15:14] = 00
WRITES3 [15:0]	Output	Write output signal of PORT_ID[15:14] = 11
WRITES3 [15:0]	Output	Write output signal of PORT_ID[15:14] = 10
WRITES3 [15:0]	Output	Write output signal of PORT_ID[15:14] = 01
WRITES3 [15:0]	Output	Write output signal of PORT_ID[15:14] = 00

Register Map

Register	Bit Size	Description
ADRS	16	Register to store specified address depending on PORT_ID[3:0]

Verification

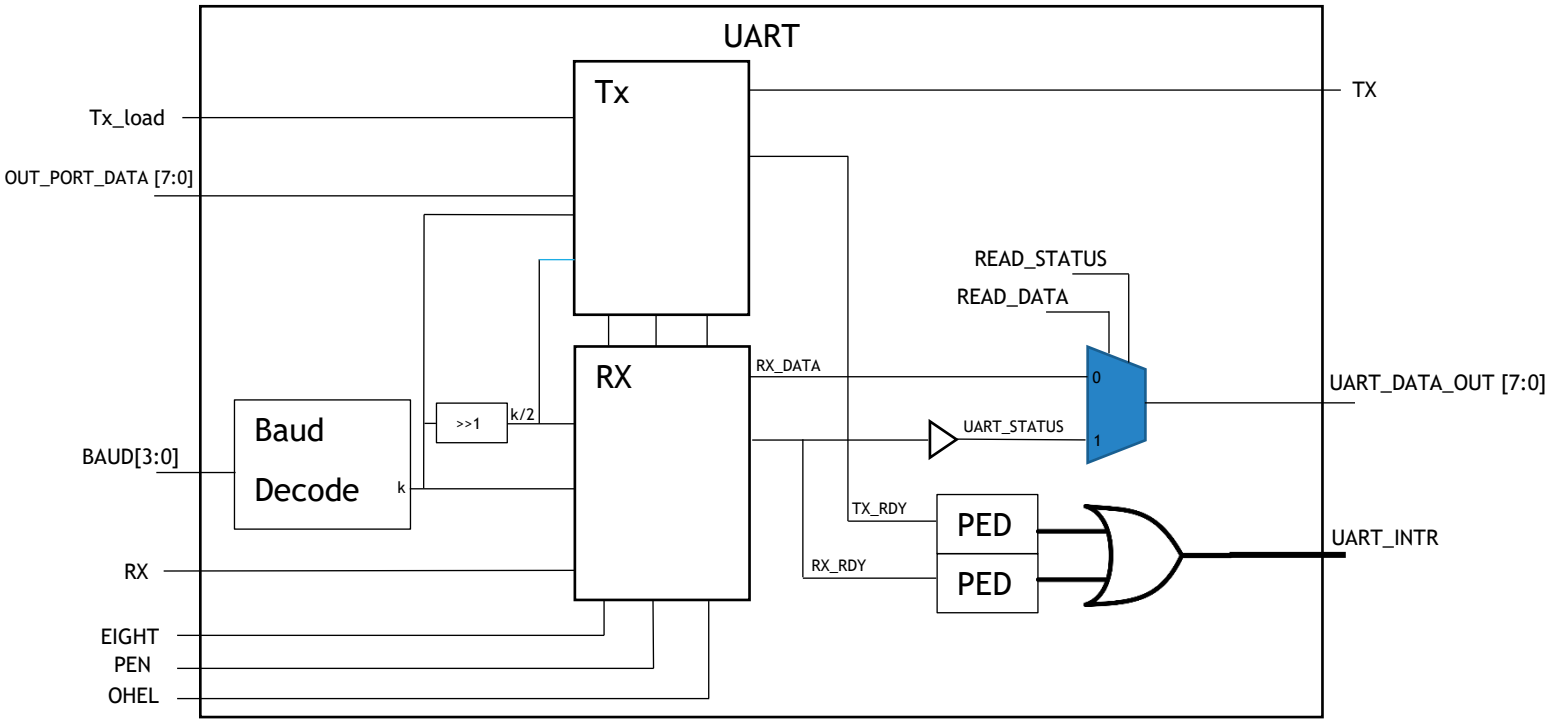


F. UART

Description

A complete UART module with both a Receive and Transmit Engine. The UART allows for both engines to share the same configurations such as baud rate, eighth bit enable, parity enable, and odd/even parity. It supports full duplex serial communication. The module also includes supplemental logic such as baud rate decoder to generate k and k/2, a mux selecting between UART data and UART status, and the TX_RDY and RX_RDY being fed through PED circuits to generate interrupt signals to the TramelBlaze.

Block Diagram



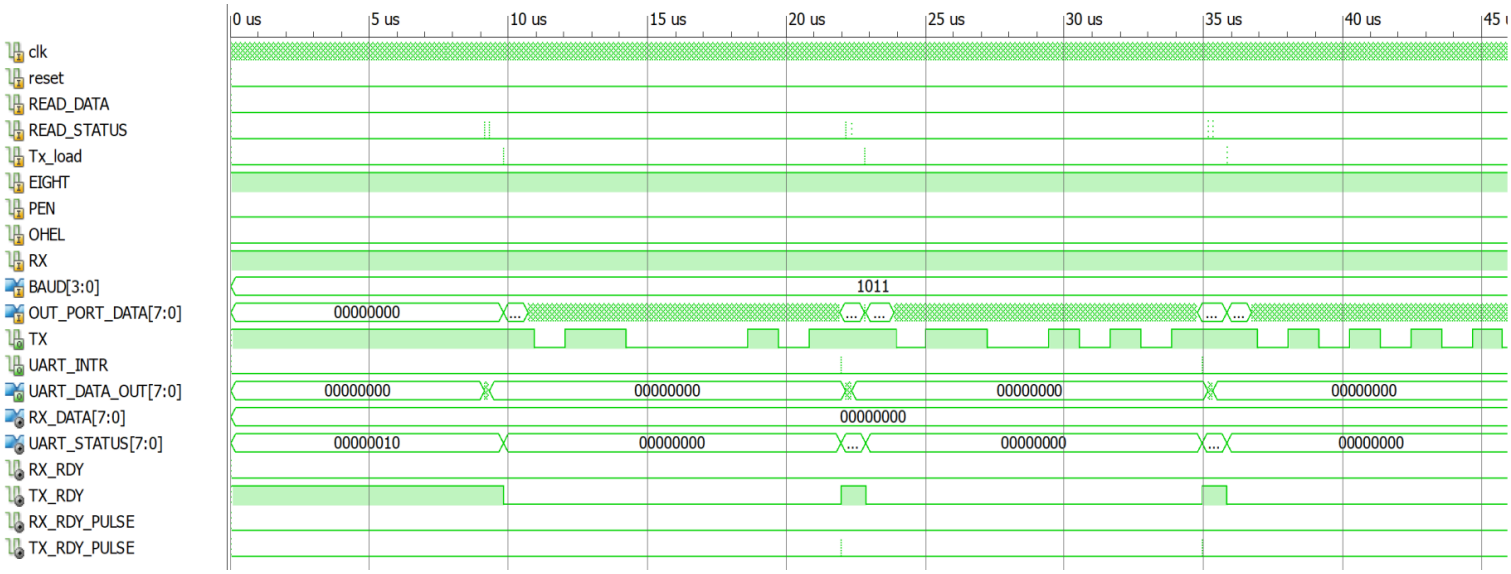
Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock
reset	Input	AISO reset
READ_DATA	Input	Select Line to read Rx Data
READ_STATUS	Input	Select Line to read Rx Status
Tx_load	Input	Loads Transmit Engine with data from the TramelBlaze
OUT_PORT_DATA[7:0]	Input	Data from the TramelBlaze to be transmitted
BAUD[3:0]	Input	Chooses between selectable baud rates
EIGHT	Input	Enables eighth bit processing
PEN	Input	Enables parity
OHEL	Input	1 enables odd parity, 0 enables even parity
RX	Input	Rx input signal
TX	Output	Tx output signal
UART_INTR	Output	UART interrupt request to the TramelBlaze
UART_DATA_OUT[7:0]	Output	UART data line to the TramelBlaze

Register Map

Register	Bit Size	Description
TX_RDY_PULSE	1	TX_RDY signal fed through a PED circuit
RX_RDY_PULSE	1	RX_RDY signal fed through a PED circuit

Verification

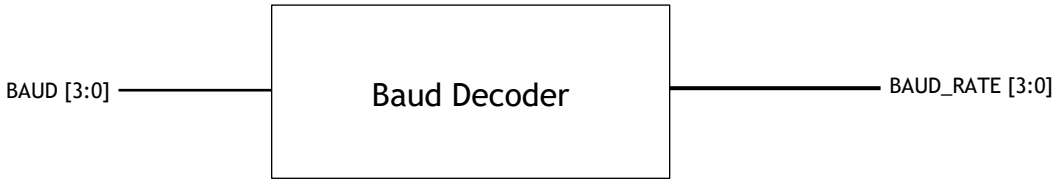


G. Baud Decoder

Description

The Baud Decoder is a decoder block that calculates the counter value for a specified baud rate with respect to a 100 MHz clock for the bit time counter. In order to calculate the counter value, simply divide 100,000,000 over the specified baud rate in bits per second. For example, a baud rate of 300 bits per second would calculate to 100,000,000 / 300 = 333,333 for the counter value.

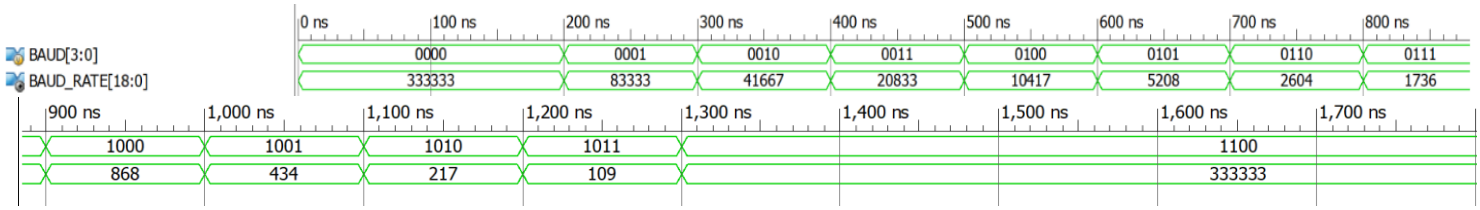
Block Diagram



Inputs/Outputs

Signal	Direction	Description
BAUD [3:0]	Input	Selectable baud rate configuration
BAUD_RATE[3:0]	Output	Calculated counter value for specified baud rate

Verification

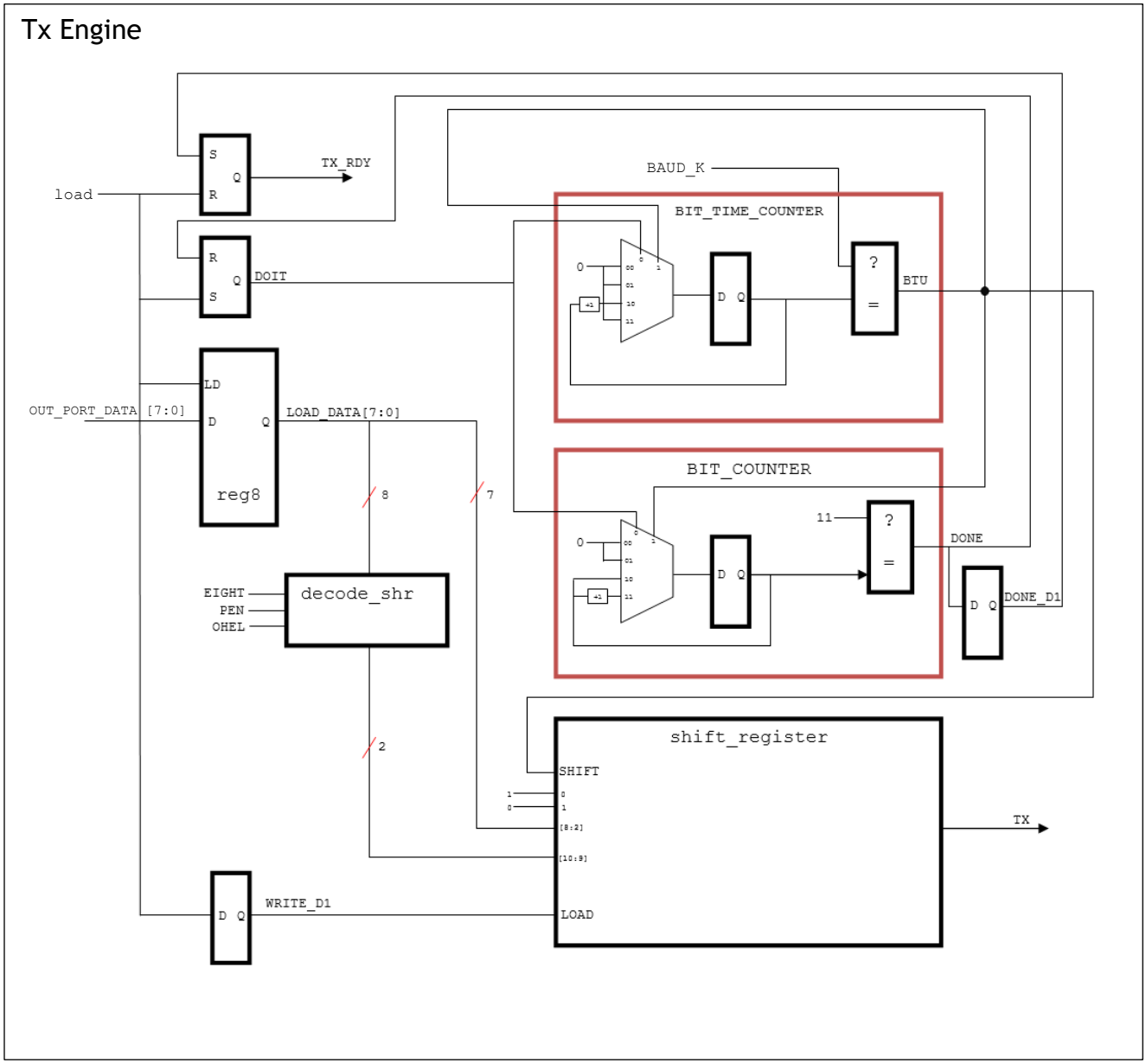


H. Tx Engine

Description

At the heart of the Transmit Engine of the UART is a 11-bit shift register that will be loaded with the correct data and outputted serially via the Tx signal. The transmit engine also outputs the TX_RDY signal, which is asserted upon reset and sends an interrupt to the TramelBlaze and initiates a transmission. During the transmission process, TX_RDY stays low until transmission is complete. At the point, TX_RDY will be asserted again and communicate to the TramelBlaze that it is ready to send more data.

Block Diagram



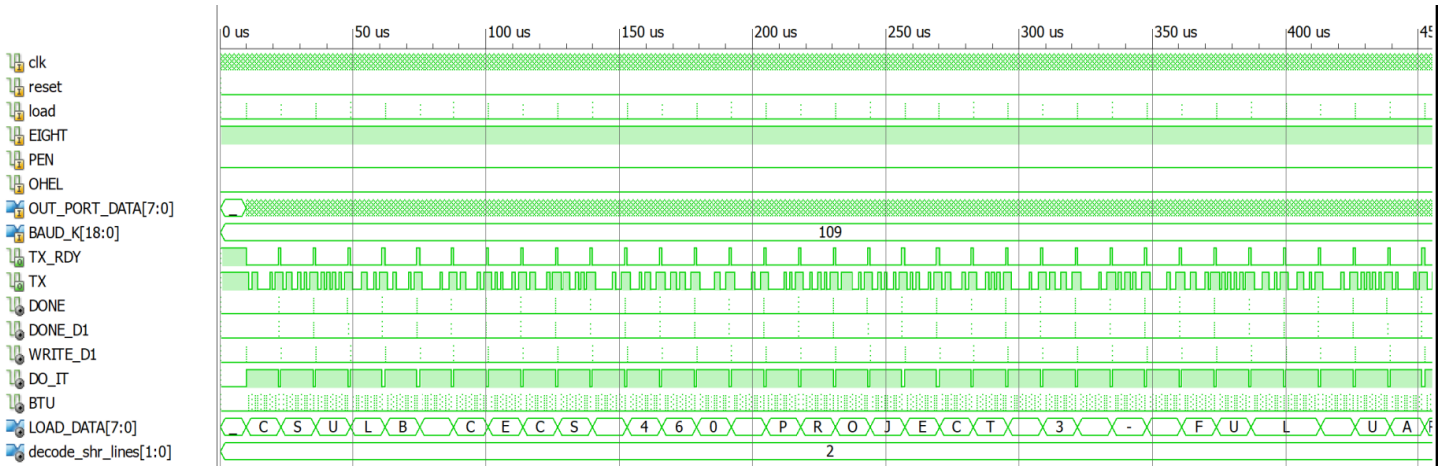
Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock
reset	Input	AISO reset
load	Input	Loads appropriates registers and initiates transmission
BAUD_K	Input	Baud rate counter constant
EIGHT	Input	Enables eighth bit
PEN	Input	Enables parity
OHEL	Input	1 for odd parity, 0 for even parity
OUT_PORT_DATA	Input	Data from the TramelBlaze to be transmitted
TX_RDY	Output	UART interrupt signal from the Tx Engine
TX	Output	TX output to terminal

Register Map

Register	Bit Size	Description
TX_RDY	1	UART interrupt signal from the Tx Engine
DO_IT	1	Initiates transmission in the Tx Engine
LOAD_DATA	8	Data from the TramelBlaze to be transmitted
WRITE_D1	1	Load signal for the 11-bit shift register
Shift_register	11	Serial output register to the Tx signal of the transmit engine
BTU	1	“Bit Time Up”; One-clock wide pulse done at the speed of the specified baud rate
DONE	1	One-clock wide pulse when transmission is complete
DONE_D1	1	One-clock wide pulse clocked after transmission is complete and asserts the TX_RDY signal

Verification

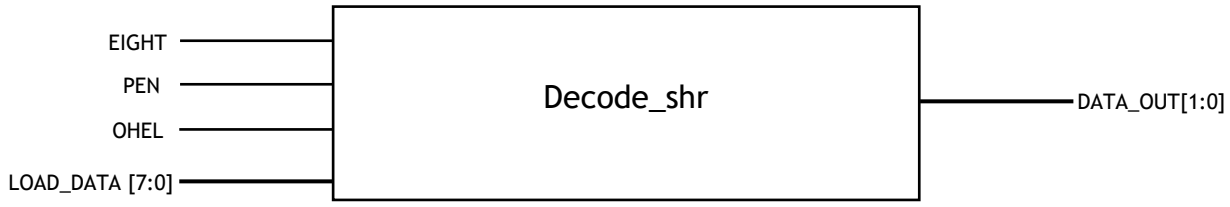


1. Decode_shr

Description

This decoder block of the Tx engine decides what data is loaded in the two most significant bits of the 11-bit shift register. Inputs for sending the eighth bit (EIGHT), Parity Enable (PEN), and Odd/Even Parity (OHEL) are determined by the user. The block uses bitwise operations for odd/even parity.

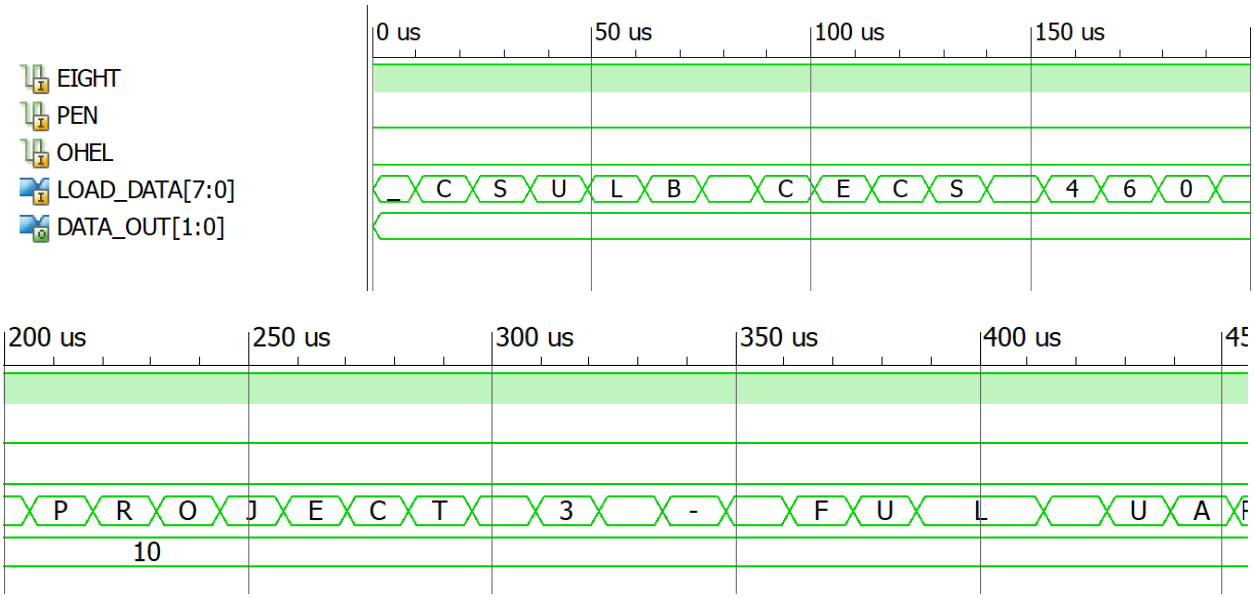
Block Diagram



Inputs/Outputs

Signal	Direction	Description
EIGHT	Input	Enable eighth bit
PEN	Input	Enables Parity bit
OHEL	Input	1 for odd parity, 0 for even parity
LOAD_DATA [7:0]	Input	Loaded data from the TramelBlaze to be transmitted
DATA_OUT [1:0]	Output	Data to be loaded to shift_register [10:9]

Verification

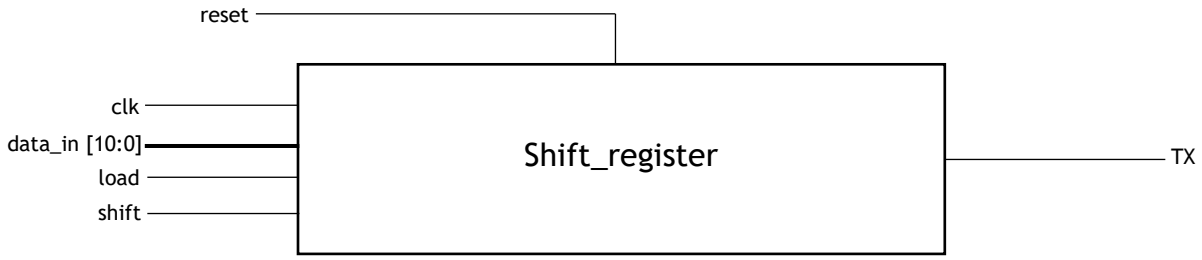


2. Shift_register (Tx)

Description

The Tx Engine shift register is responsible for being loaded with the correct information then being shifted and outputted the correct number of times to the Tx signal. The Tx signal is a one-bit output because the UART is parallel-in-serial-out. Upon reset, the shift register is loaded with “1’s”, marking the Tx signal as inactive. Only when a “0” is shifted into the Tx signal will it be considered the start bit and begin transmission.

Block Diagram



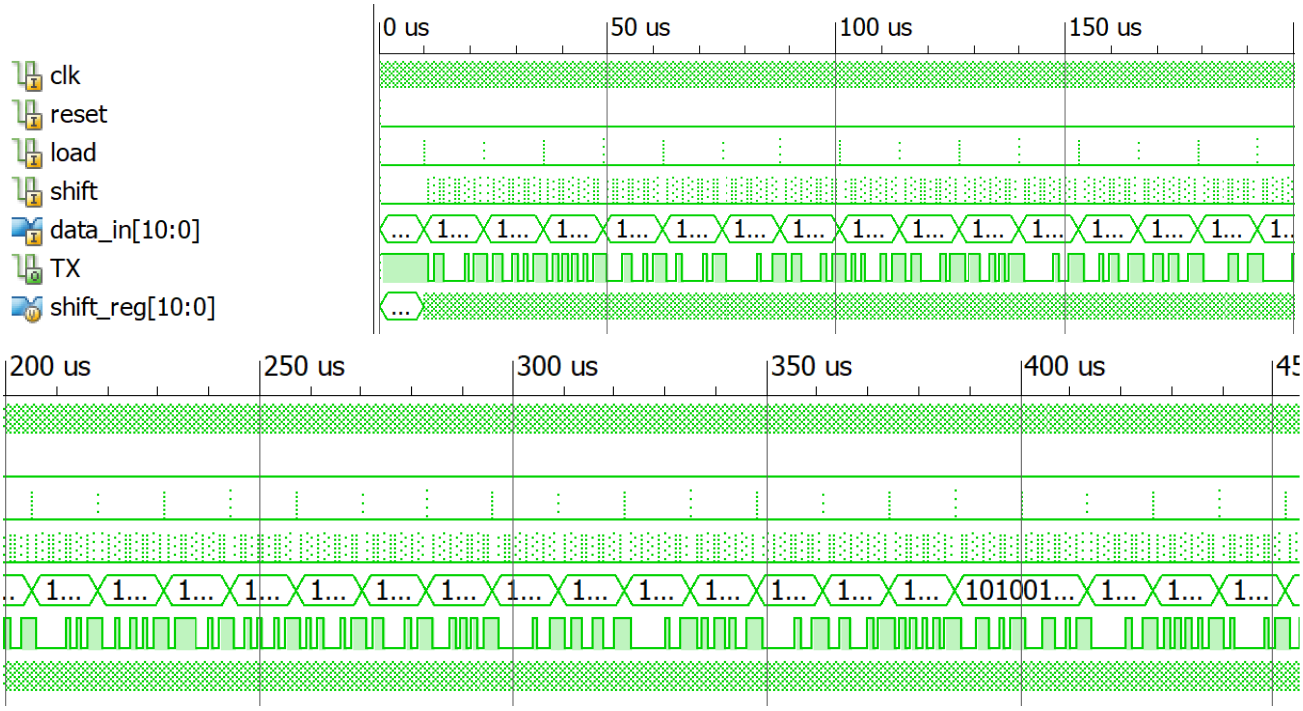
Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock
reset	Input	AISO reset
load	Input	Loads the shift register with a processed version of the TramelBlaze data
Shift	Input	Shifts the contents of the register to the right and loads the 11th bit with a “1”
data_in [10:0]	Input	Data loaded in to the shift register in the even of a load; [10:9] come from the decode_shr block, [8:2] are the original bits of the loaded data, and [1:0] = 01 by default.
TX	Input	Tx output signal to the terminal

Register Map

Register	Bit Size	Description
shift_register	11	Shift register responsible for being loaded with the correct data and outputted serially to the Tx output signal

Verfication

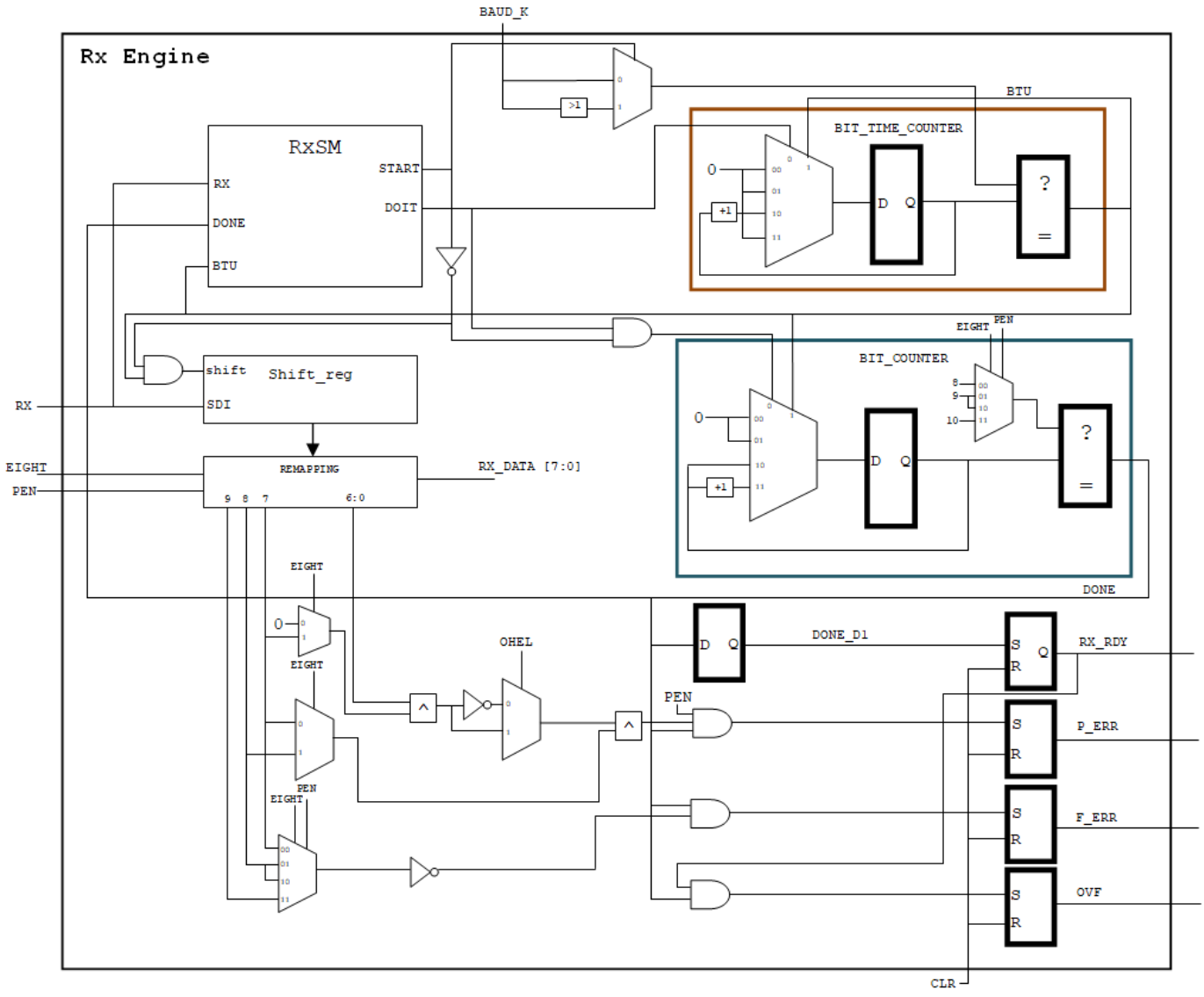


I. Rx Engine

Description

The goal of the Receive Engine is to synchronize the data collection with the Tx communication. The Rx engine is always polling the Rx input and looking for high-to-low transition, indicating the arrival of a start bit. The Rx engine also detects any errors when processing the data such as framing error, parity error, and overflow

Block Diagram



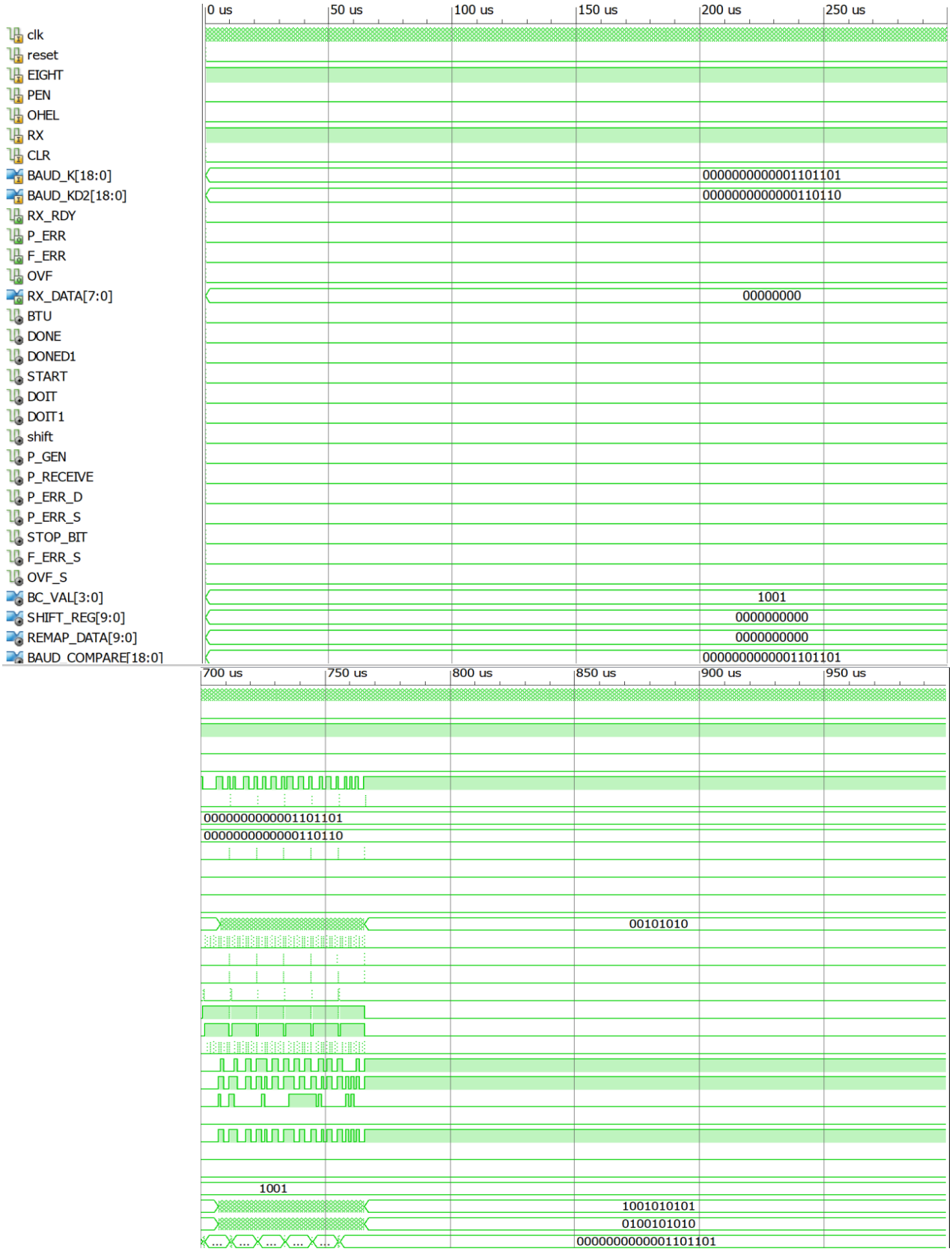
Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock
reset	Input	AISO reset
BAUD_K	Input	Calculated bit time counter value for a specified baud rate
BAUD_KD2	Input	Calculated bit time counter divided by 2
EIGHT	Input	Enable eighth bit
PEN	Input	Enables parity bit
OHEL	Input	1 for odd parity, 0 for even parity
RX	Input	Rx input signal from the terminal
CLR	Input	Clears various flops within Rx such as RX_RDY, Parity Error, Framing Error, and Overflow Error
RX_RDY	Output	Interrupt signal to the TramelBlaze signaling the data is done being processed
P_ERR	Output	Parity Error
F_ERR	Output	Framing Error
OVF	Output	Overflow Error
RX_DATA	Output	Processed Data from the Rx Engine

Register Maps

Register	Bit Size	Description
RX_RDY	1	UART interrupt signal from the Rx Engine
START	1	Signal to iterate through the UART protocol through half bit time
DOIT	1	Initiates Data processing within the Rx Engine
Shift_reg	10	Shift register responsible to read in data from the Rx input
BTU	1	“Bit Time Up”; One-clock wide pulse done at the speed of the specified baud rate
DONE	1	One-clock wide pulse when data receive and processing is complete
DONE_D1	1	One-clock wide pulse clocked after data processing is complete and asserts the RX_RDY signal
P_ERR	1	Parity error SR flop
F_ERR	1	Framing error SR flop
OVF	1	Overflow error SR flop

Verification

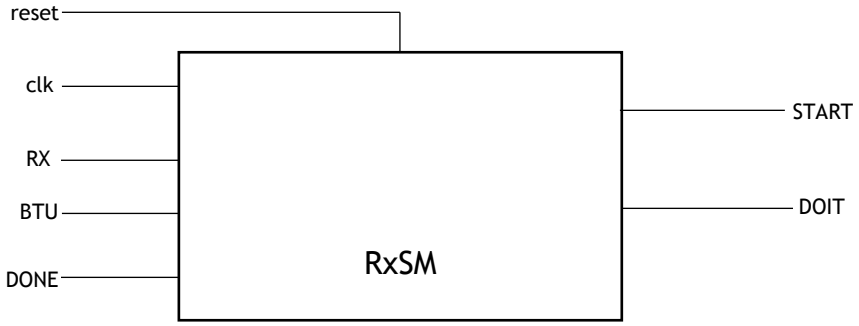


1. Rx State Machine

Description

The Rx State Machine for the Receive Engine is responsible for ensuring that the start bit is able to be sampled by remaining active until the mid-bit time. When half the bit time has elapsed, the receive engine will then continue to process data at normal bit time until finished. Outputs are START and DOIT: START indicates looking for the first bit, and DOIT indicates that the engine is reading and processing data. The Rx State Machine is designed as a modified Moore machine, with outputs also having a present and next state.

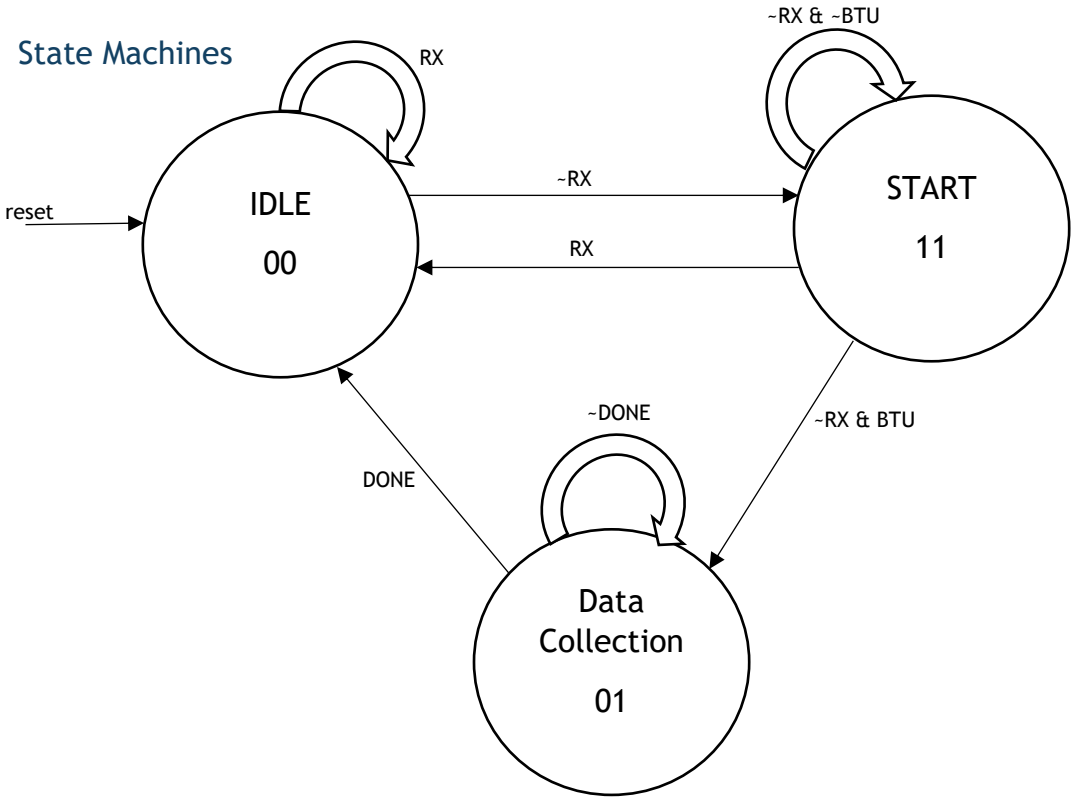
Block Diagram



Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock
reset	Input	AISO reset
RX	Input	Rx input signal from the terminal
BTU	Input	“Bit Time Up”; One-clock wide pulse done at the speed of the specified baud rate (or sometimes half)
DONE	Input	One-clock wide pulse for when data is done being received and processed in the Rx Engine
START	Output	Signals the occurrence of a start bit
DOIT	Output	Initiates data processing after the start bit has occurred

State Machines

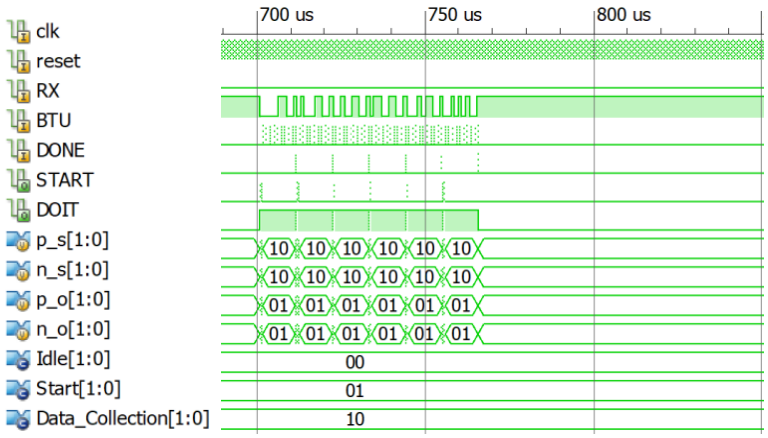


Outputs: {START, DOIT}

Register Map

Register	Bit Size	Description
p_s	2	Present state register
n_s	2	Next state to be clocked
p_o	2	Present output of the state machine (START, DOIT)
n_o	2	Next output to be clocked

Verification

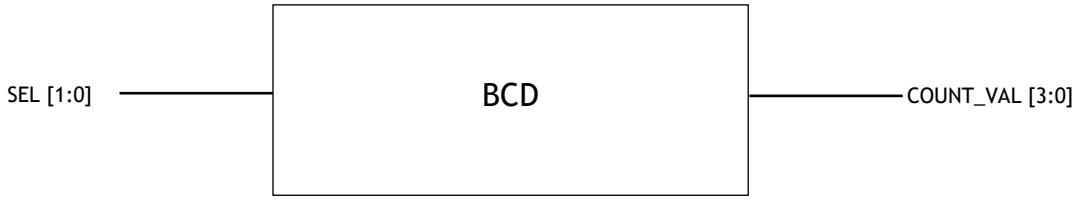


2. Bit Count Decoder (BCD)

Description

The Bit Count Decoder block in the Rx Engine calculates how many bits to count in the serial input, Rx, to the Rx Engine based on the inputs EIGHT and PEN.

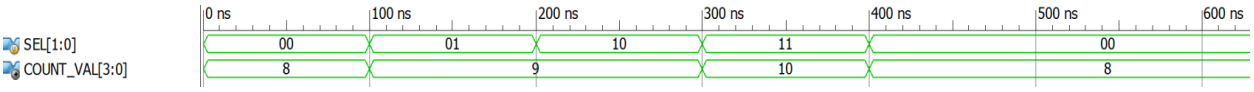
Block Diagram



Inputs/Outputs

Signal	Direction	Description
SEL [1:0]	Input	Inputs EIGHT and PEN, respectively
COUNT_VAL [3:0]	Output	Calculated bit count value

Verification

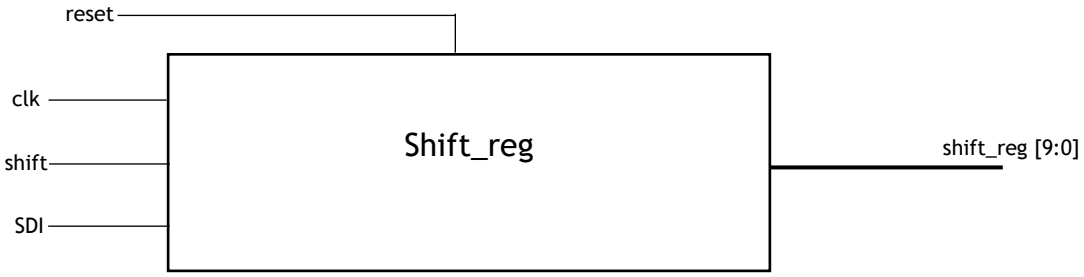


3. Shift_reg (Rx)

Description

The 10-bit shift register of the Rx Engine is responsible for reading in data from the Rx input signal and then being remapped and sent as an input to the TramelBlaze (IN_PORT). Since Rx is one bit wide, the shift register supports a serial-in-parallel-out protocol. Data is shifted to the right in the register whenever “Bit Time Up” (BTU) is asserted.

Block Diagram



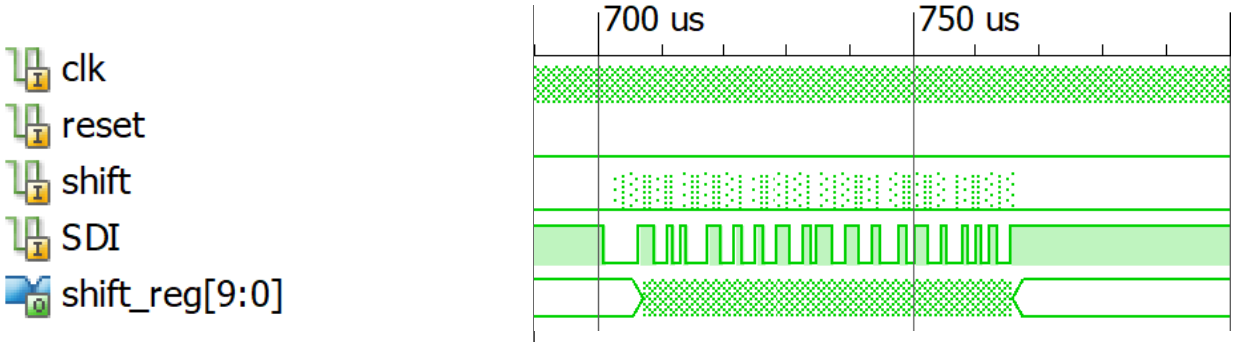
Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock
reset	Input	AISO reset
shift	Input	Shifts the contents of the register one space to the right
SDI	Input	“Shift Data In”; connected to the Rx input Signal; set as shift_reg [9] in the event of a shift
shift_reg[9:0]	Output	10-bit shift register contents

Register Map

Register	Bit Size	Description
shift_reg	10	Shift register responsible for being shifting data in from the Rx input signal and outputting it as parallel data

Verification

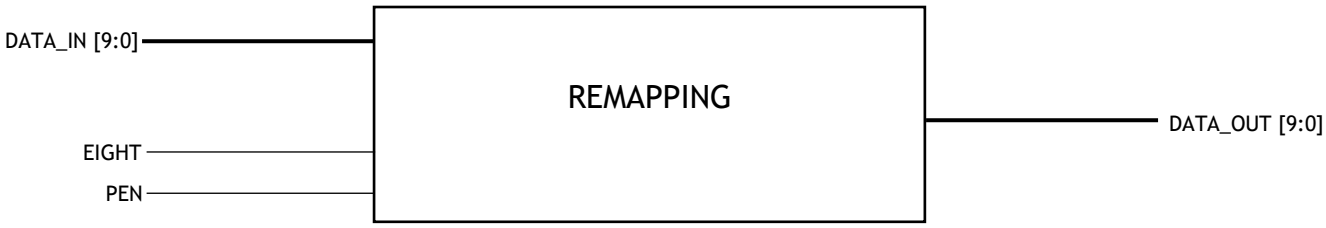


4. REMAPPING

Description

Remapping is responsible for “right justification” of the received data based upon UART configuration for the eighth bit and parity enable. The block outputs the data as RX_DATA of the Rx Engine.

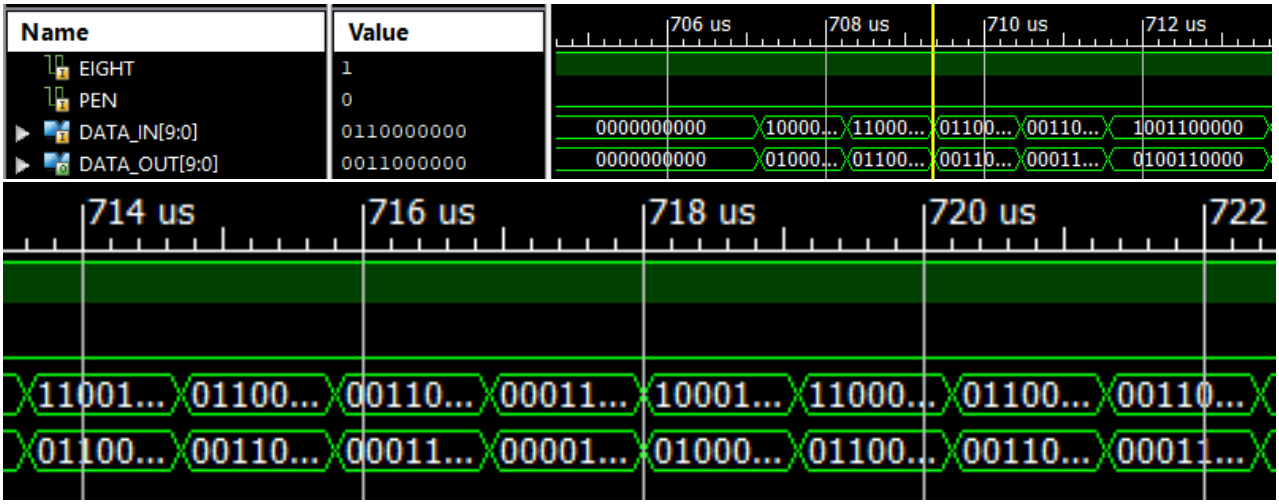
Block Diagram



Inputs/Outputs

Signal	Direction	Description
DATA_IN [9:0]	Input	Data loaded from the Rx Engine’s 10-bit shift register
EIGHT	Input	Enables eighth bit
PEN	Input	Enables Parity bit
DATA_OUT [9:0]	Output	Data that has been remapped to the UART configuration and considered valid

Verification

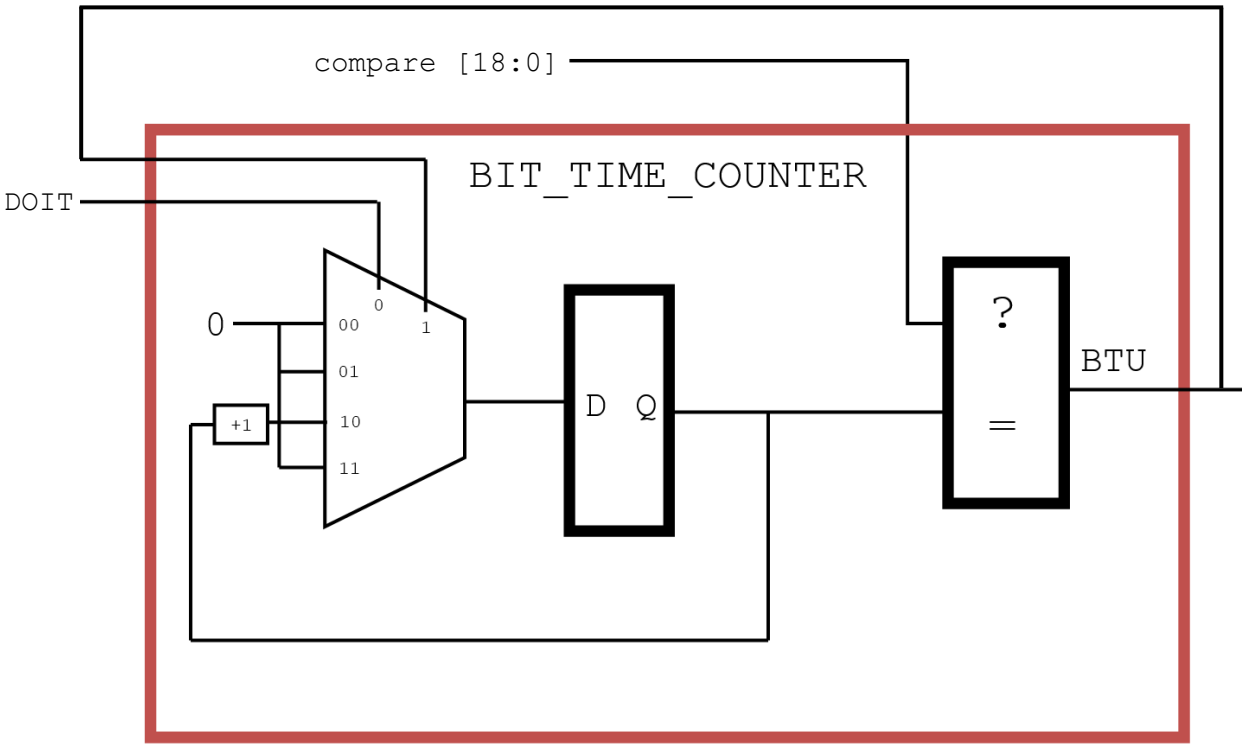


J. Bit Time Counter

Description

The Bit Time Counter block specifies how fast data is processed within both engines of the UART. In the Tx Engine, the counter determines how long a bit stays on the Tx output signal; for the Rx Engine, the counter determines how fast data is shifted in to the 10-bit shift register. The Baud Decoder block calculates the “compare” value.

Block Diagram



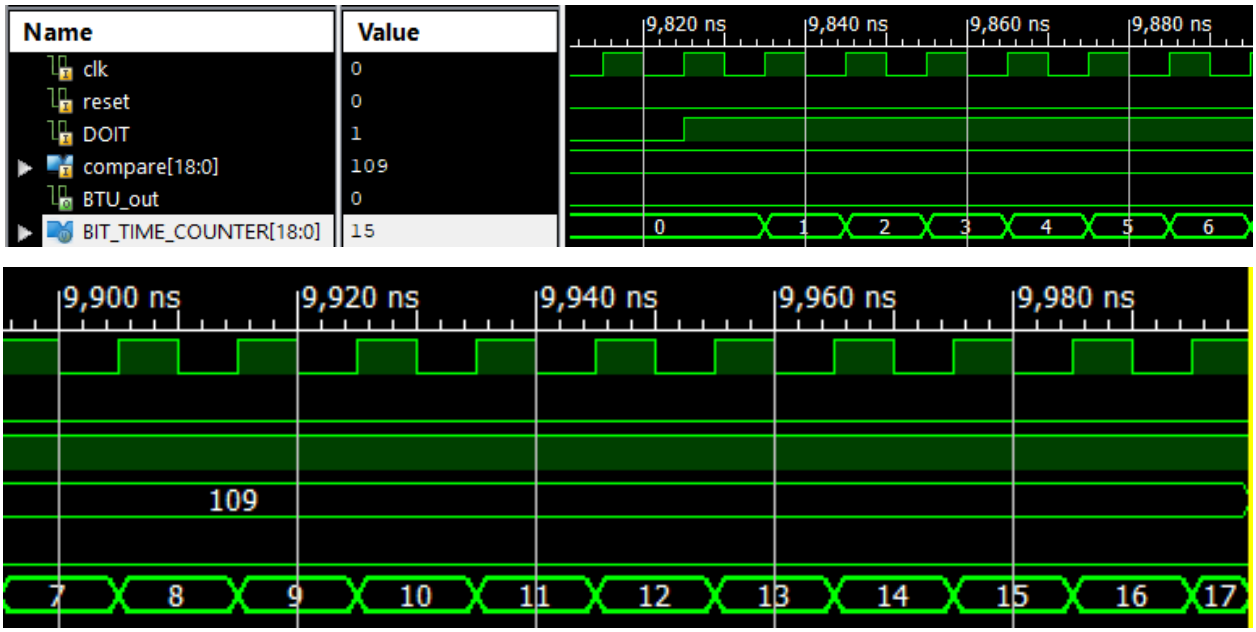
Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock
reset	Input	AISO reset
compare [18:0]	Input	Calculated counter value to the specified baud rate
BTU_out	Output	“Bit Time Up”; One-clock wide pulse done at the speed of the specified baud rate

Register Map

Register	Bit Size	Description
BIT_TIME_COUNTER	19	Counter register that increments on every clock cycle while the DOIT signal is asserted. When the counter reaches the specified “compare” value, the “BTU” signal is asserted for one clock and the counter resets to 0 and continues to increment and reset until data processing completes.

Verification

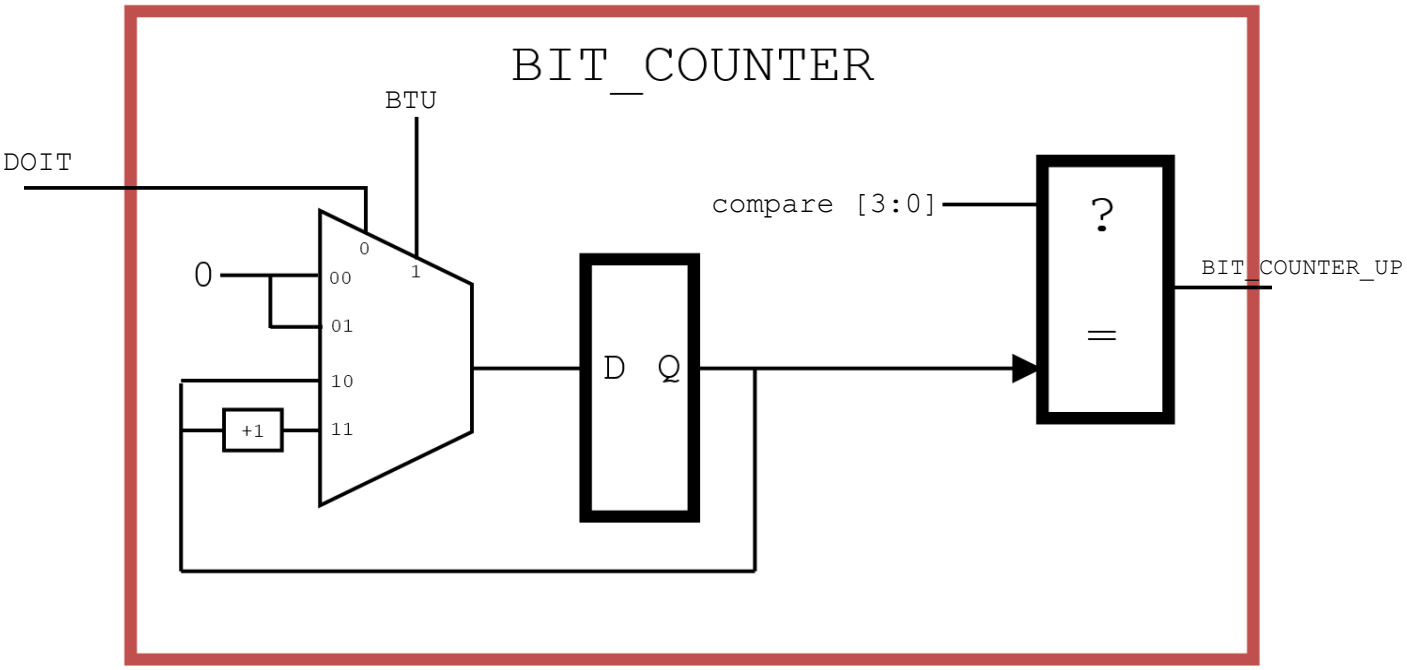


K. Bit Counter (Rx)

Description

The Bit Counter block keeps track of how many bits are to be processed within the engines of the UART. In the Tx Engine, the counter keeps track how many bits are being sent to the Tx signal; for the Rx Engine, the counter keeps track of how many bits the Rx Engine reads before the data is remapped and set as the RX_DATA. In both engines, when the counter reaches the counter value, the TX_RDY or RX_RDY signal is asserted on the next clock. It is important to note that the Bit Counter's compare value within the Rx Engine changes depending on the configurations for EIGHT and PEN while the Bit Counter's value within the Tx Engine is hardwired to 11 bits.

Block Diagram



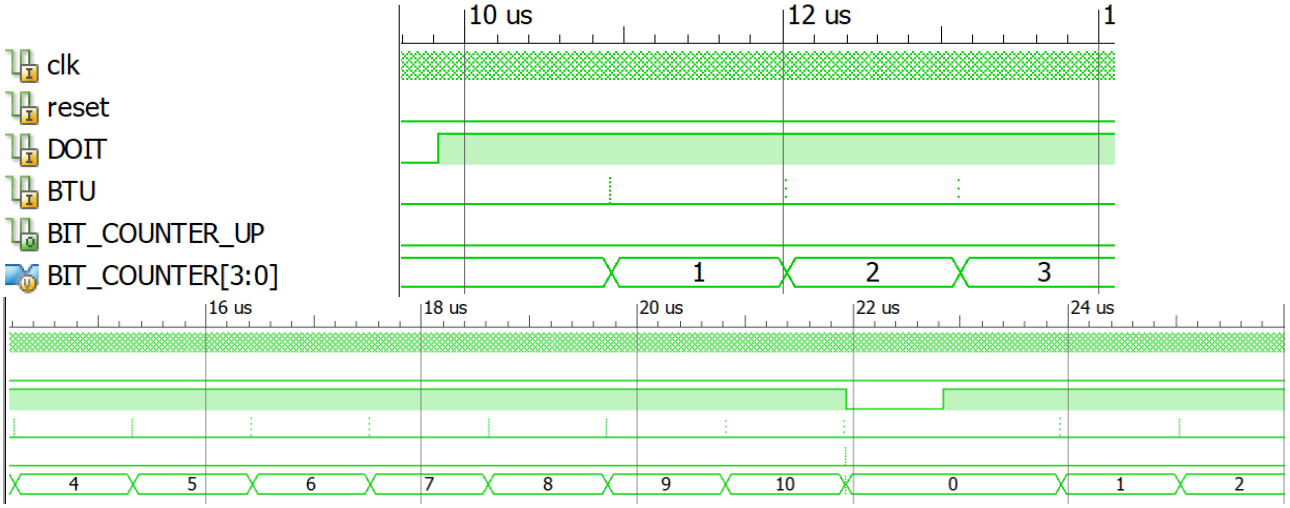
Inputs/Outputs

Signal	Direction	Description
clk	Input	100 MHz clock
reset	Input	AISO reset
compare [3:0]	Input	Calculated counter value depending on the configuration for EIGHT and PEN. Hardwired to 11 in the Tx Engine
BTU	Input	“Bit Time Up”; One-clock wide pulse done at the speed of the specified baud rate
BIT_COUNTER_UP	Input	One-clock wide pulse to signal when data processing is complete for either Tx or Rx Engine

Register Map

Register	Bit Size	Description
BIT_COUNTER	4	Counter register that increments whenever both BTU and DOIT are asserted. When the counter reaches the specified “compare” value, the DONE signal is asserted and TX_RDY/RX_RDY are asserted on the next clock

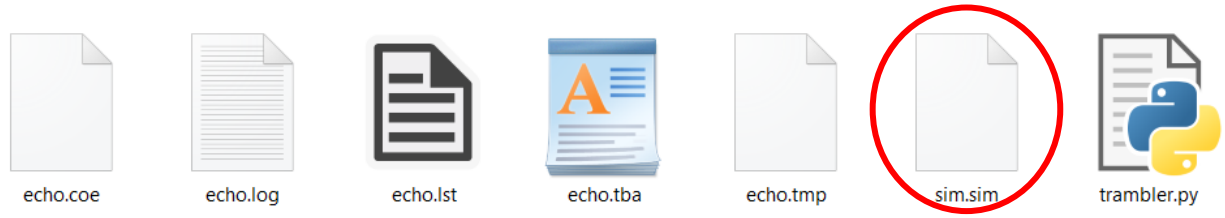
Verification



VII. CHIP LEVEL VERIFICATION

Verification of the SOPC design is done using a test fixture with a variety of different verification elements. The first thing to note is because of the time-consuming task of instantiating instruction memory whenever changes are made to the software, the top level TramelBlaze module also has an alternative version where it runs the “sim.sim” file of the assembly directory as its instruction set rather than the instruction ROM block. The sim.sim file emulates the instructions that would normally be instantiated in the instruction ROM for simulation purposes.

Sim.sim file in the Assembly Directory After Successful Assembly

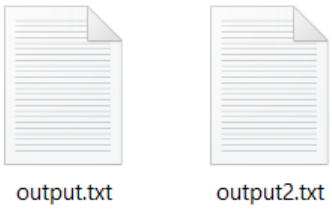


Any time the .tba file is assembled, a sim.sim file will be generated for simulation and verification purposes.

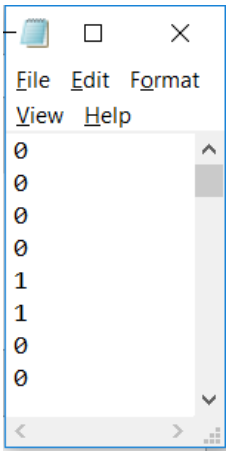
Next, when writing the test fixture, the main goal is to simulate particular inputs to the RX signal while also following the timing requirements of the baud rate. For simulation purposes, the baud rate has been turned up to the highest setting (921,600 bits/sec), meaning that one bit is processed every 1,085 nanoseconds by a 100 MHz clock. Therefore, the simulation will send bits to the RX input every 1,085 nanoseconds with the least significant bit inputted first. Data will consist of various ASCII characters to be displayed in waveforms within the simulation.

These bits of data are written into text (.txt) files that are to be read into temporary memory blocks and fed into the RX signal.

Example text files for RX bits



Bits represent 0x30, “0” in ASCII



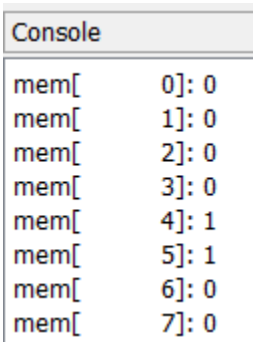
Bits Read into Memory array

```
$readmemb("output.txt", mem);
```

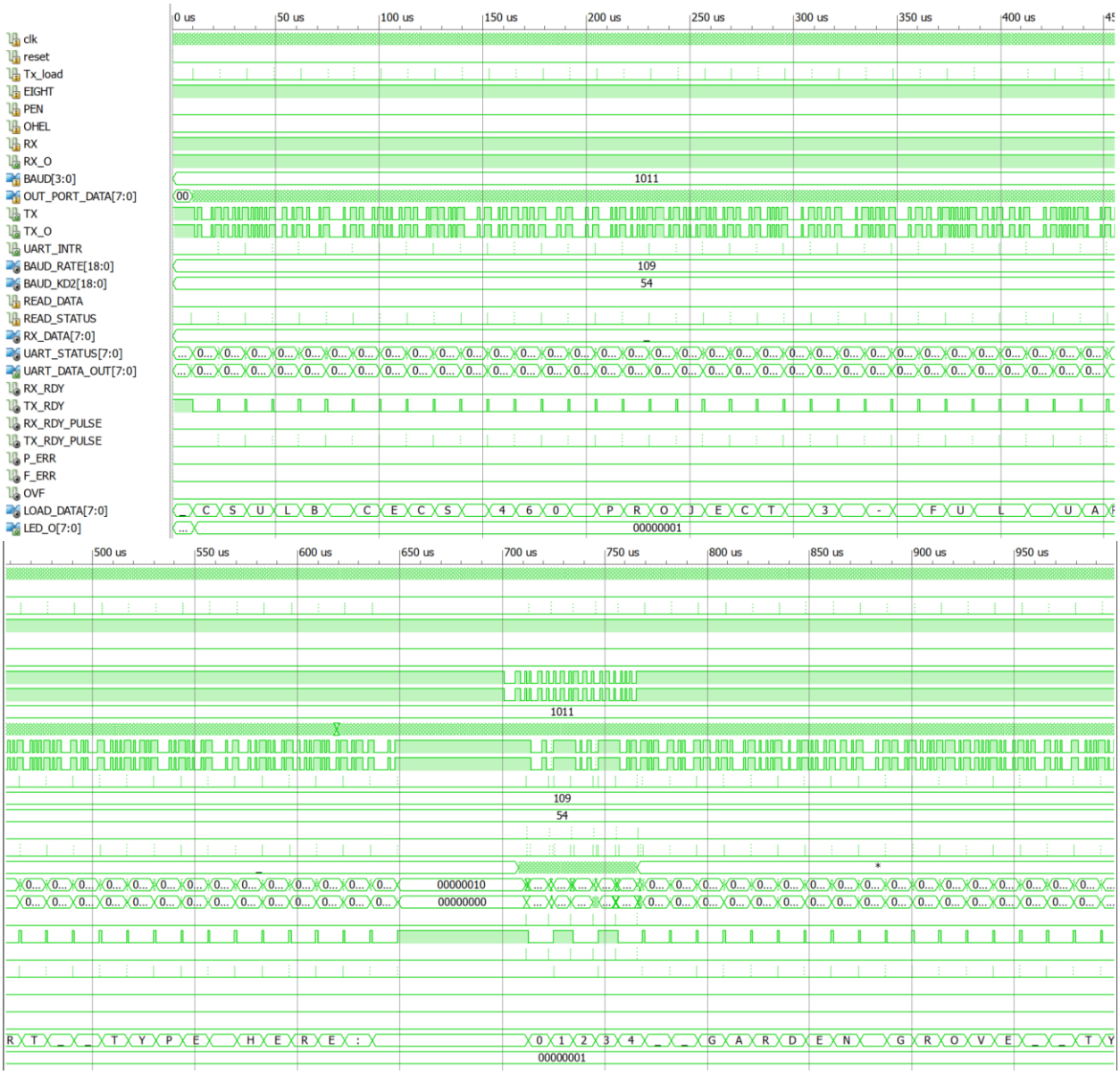
A for-loop cycles through the memory array and inputs that data into the RX signal. It also ensures that each byte of data begins with start bit and ends with a stop bit. Other configurations such EIGHT, PEN, and OHEL are accounted for in the test fixture code; however, the bits written in the text file must be changed accordingly to fit the given configurations.

Verification on the SOPC requirements is done through analysis of the waveform simulation.

Memory Contents Being Verified in the Simulation Console



Waveform Simulation (5 characters, display hometown)

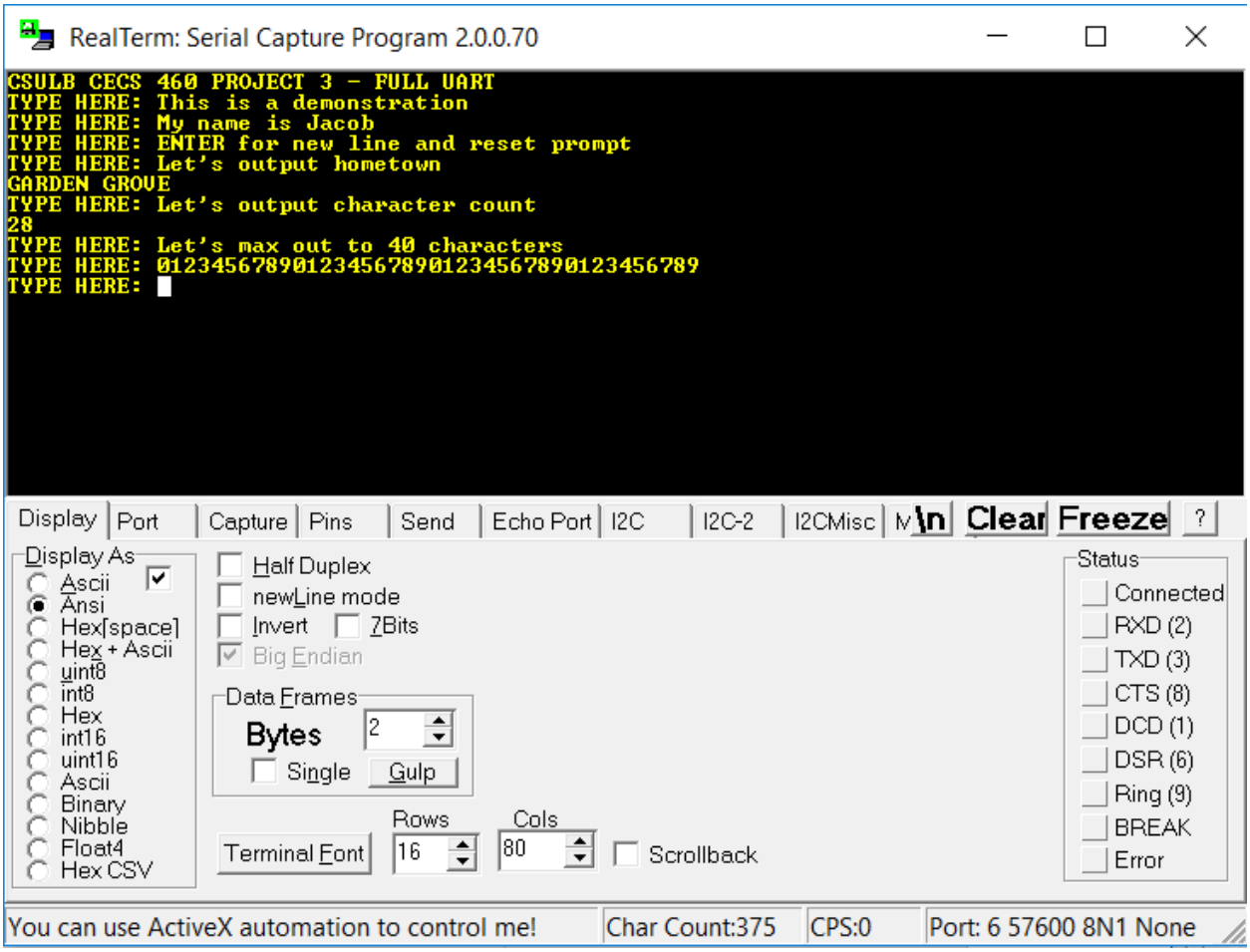


One of the benefits of verifying via waveform simulation is the ability to see the "inner-workings" and the various signals of the design that would normally be obscure if immediately programmed to a FPGA board.

In the waveform simulation above, particular signals that can be observed to meet the requirements are the LOAD_DATA [7:0], showing that it is properly getting loaded with the correct ASCII character and outputted to the terminal via the Tx output signal, and the error signals, P_ERR, F_ERR, and OVF showing no signs of data processing errors within the UART.

VIII. CHIP LEVEL TEST

Demonstration of SOPC Design in RealTerm serial terminal



IX. APPENDIX

All Verilog source code, test fixtures, and assembly code are within this section of the document.