**ECE 554 Mini-Project 1 – Special Purpose Asynchronous Receiver/Transmitter**

**Functional Description**

The Special Purpose Asynchronous Receiver/Transmitter, or SPART, is used for serial input and output between a Xilinx Vertex II FPGA prototyping board and a workstation with a terminal program. For the mini-project, the objective was to create an implementation that would enable a character to be sent from the terminal program on the workstation to the FPGA via a COM port, and have the FPGA echo the character entered back onto the workstation’s terminal. For abstraction and debugging purposes, we chose to base our implementation on a modular design. This required separation of functionality within the SPART with the added benefit of separating complexity into separate modules, allowing us to abstract away blocks already deemed functional.

The SPART was separated into four functional blocks: the bus interface, the baud generator, the transmitter, and the receiver as shown in Figure 1. All Verilog HDL is included within the project package.



Figure 1: SPART block diagram

The bus interface was implemented using combinational logic and served as the primary interface between the processor and the SPART module. In addition to providing the processor with access to the transmit and receive buffers, the status register also had to be made available for reading to see if there was data that needed to be read, or whether the transmit buffer was able to accept a new transmission. In addition, the bus interface had to manage the 8-bit tristate data bus and ensure that only one module had access to it at any given time. Finally, the interface also provided control signals to the baud generator and transmitter to control transactions to those modules, as well as control signals to the processor to signify that data was able to read or that the SPART was able to accept data to write.

The baud generator was implemented using sequential logic and was responsible for generating sampling and transmit/receive signals for the SPART to operate within. As the SPART implies asynchronous behavior (from the receiving end, between the terminal and the SPART), the serial data coming in must be sampled in order to recognize a start bit to begin a receive operation. The receive sampling signal samples at 16x the baud rate. The baud rate is simply the rate at which data is transferred between the SPART and the workstation terminal. This was accomplished using a counter that divided the input clock that runs at a frequency of 100 MHz. The counter was loaded with a pre-calculated value based on the SPART’s operational baud rate (4800, 9600, 19200, 38400), and was decremented until it reached 0. At this point, a sampling signal was toggled, and for every 16th sampling signal, a baud rate signal was toggled.

The transmitter was also implemented using sequential logic and was solely responsible for transmitting data out serially to the workstation. This was accomplished using a state machine as well as a shifter. When a transaction was set to begin, a shifter was loaded with the data that would be transmitted as well as a start and stop bit so the workstation terminal could process the data on its end. With every baud rate signal the data was shifted out starting with the start bit, and proceeding with the least significant bit of the data to be transmitted. During any given transmission, the signal dictating that the transmitter was ready to receive another transmission was deasserted. This ensured that transmit data would not be overwritten.

The final module included within the SPART was the receiver. This was implemented using sequential logic, and in many ways is a direct inverse of the transmitter with the addition of a sampling module. The sampling module was responsible for the recognition of a start bit, as well as dictating when the receiving module should take in a data bit. This was the most integral part of the design. The sampling module waits for the first 0 bit that it sees, indicating a potential start bit. Sampling at the aforementioned sampling rate, the sampling module asserts a start signal if the next 8 samples are also 0. At this point, the sampling module has located the middle of a start bit (as each bit is transferred at the baud rate, or contains 16 samples). The sampling module then outputs an enable signal for every 16 samples 8 times. Outputting at every 16 samples ensures that the 8 data bits are also sampled in the middle of the bit as well. The receiving module uses a state machine to shift the serial data in serially and when the shifting is complete, the receive buffer is loaded with the data that was received.

Also included within the mini-project was a driver module which took the place of a processor. This driver module was responsible for controlling the operations of the SPART. This was done using a simple state machine that used the following state diagram:

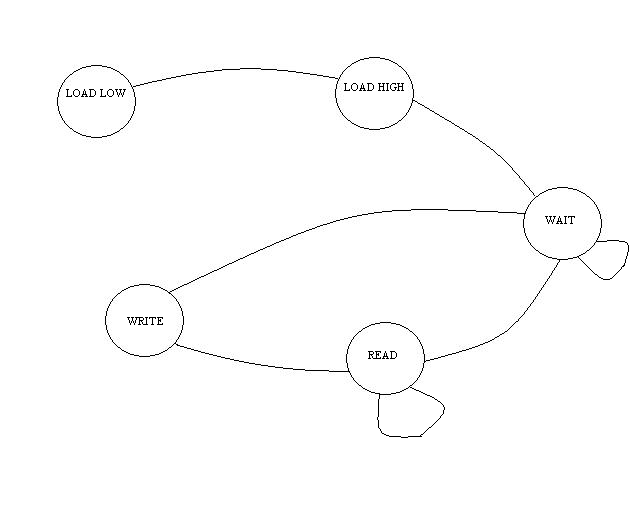


Figure 2: Simple driver state diagram

Additional control signals were used to dictate the state and next state logic. First, the baud generator counter has to be loaded with a value. This counter takes a 16-bit value. Since our data bus is only 8-bits wide, this had to be broken into two transactions. First the lower byte is written, followed by the higher byte. The driver then enters a wait state, where it waits for received buffer data to be available after a receive has been established. The driver will then enter a read state in which the data is read. It will wait here until a point that the transmitter is able to write the data back out to the workstation terminal. The driver will then enter the wait state to await further transactions.

The functionality of the overall SPART, as well as the included driver is finally wrapped into a top level module which instantiates both modules and handles the connections to various peripherals on the FPGA prototyping boards. These include dip switches to allow the user to reset the SPART and configure different baud rates, as well as LEDs to indicate other various signals.

**Testing**

Initial testing took place within the ModelSim development environment. ModelSim provides several advantages over programming and debugging from the FPGA board. If there was a functional error, it is much more visible on a waveform within ModelSim than once the FPGA is programmed as one can interpret any signal within the design. Also, the time to program the FPGA board often takes at least 8-10 minutes, which isn’t ideal for rigorous testing before proper functionality if achieved within ModelSim.

Within ModelSim, every module was tested with a testbench. This enabled the testing of every module by inputting activation signals into the system, as well as being able to abstract away the modules that were already functional. After every module was verified, the design was combined into a top level design, and further testing was done on the system by combining two SPARTS and two driver modules. One SPART/driver initiated a transfer of our choosing serially to the other SPART/driver, where it was received. The other SPART/driver then echoed the transmission back to the first SPART/driver as it should. Once the data came full circle back to our first SPART/driver, a check was done to make sure that the proper data was received.

Simple testing was done to ensure functionality of the SPART design. The figure below shows a sample of input to the HyperTerminal program running on the workstation as the data is echoed back to the terminal from the SPART:

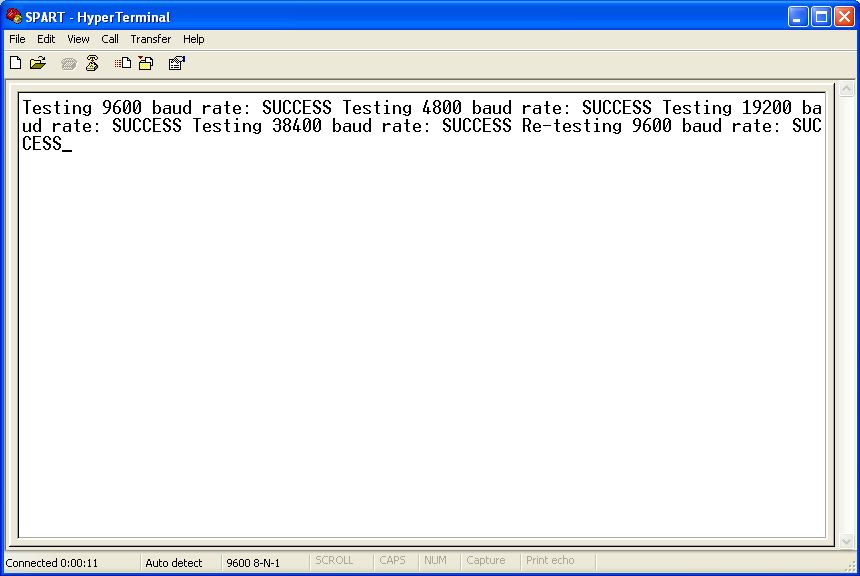


Figure 3: HyperTerminal output from the SPART

Below, the same simple test was repeated, but with the terminal echoing its own output as well as the output received back from the SPART showing what is sent and what is received back:

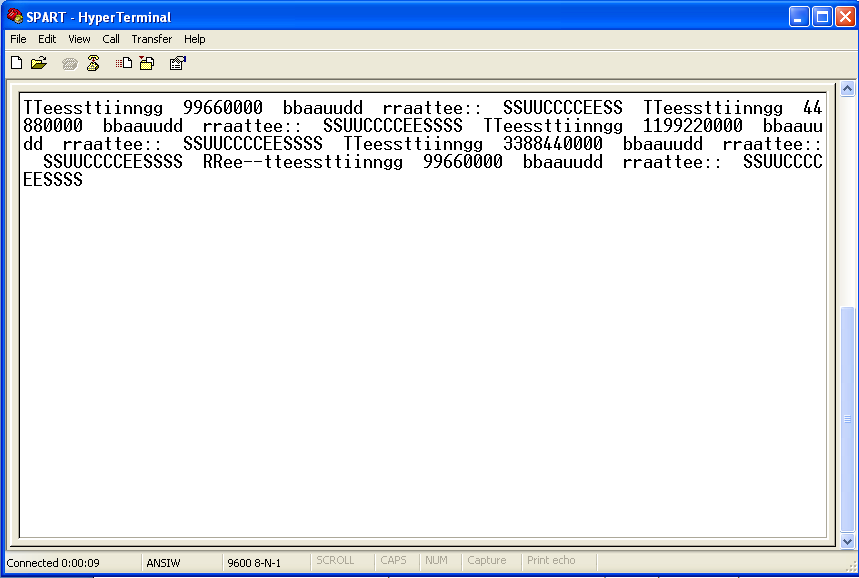


Figure 4: HyperTerminal output from the SPART with HyperTerminal echo verification

**Issues Encountered**

The design of the SPART proved to be a challenge in several areas. The first issue was that proper care wasn’t taken to ensure that the data would be sampled at the middle of every data bit. This led to an overhauling of the sampling module to entirely dictate when the receiver was to actually receive a character.

In addition, it was seen that sampling was beginning again immediately after the final receive enable signal for a transaction was given. This was leading to preemptive sampling, and if the last data bit was a 0, an inaccurate transaction may be interpreted. This was fixed by ensuring that sampling would not resume until the following baud rate signal was toggled.

Once these issues were fixed and functional verification was confirmed in a ModelSim simulation environment, operation was still not functional when programmed onto the FPGA board. At this point, we took note of the warnings given by the Xilinx IDE software. There were several warnings that indicated inferred latches as well as some clock gating going on which generally overlapped with those latches that were inferred. The Verilog HDL was then cleaned up to ensure that no inferred latch behavior was present. At this point, the board would output some character when any key was entered, but it was never the matching character. Upon inspection of an ASCII table, it was determined that the data was simply being transmitted most significant bit first. We reversed the transfer operation, and the SPART worked as expected.

**Verilog Code**

*Attached below*