

ECE/CS 3700 — Digital System Design

Final Project — A Simple UART

Documentation and circuit demo due by the week of April 20th

Overview

This lab consists of building a simple Universal Asynchronous Receiver/Transmitter (UART) circuit that can communicate with a standard computer terminal. Your final circuit should be able to accept serial ASCII data from a terminal (when you press a key on the terminal, serial ASCII data is sent out the serial connection) and collect it into a parallel byte of information. This byte of data will then be sent back to the terminal as serial data so that it is echoed on the terminal screen. Although this might sound like a very complicated project at first (or even second) glance, it is just an application of using state machines to control counters and shift registers. Since you know all about state machines by now, this should not be much different than the types of things you've already done. However, it has three particularly interesting features:

1. It is a much more open-ended problem than previous labs and homework problems. By this We mean that there will be *many* ways to solve this problem! There is no right way and wrong way, only ways that work and ways that don't work. However, you need to take some time to think carefully about the problem before launching into the design phase so that you don't end up with a huge bulky solution.
2. It is an example of a Finite State Machine interacting with data path elements. The FSM is the controller, and the counters and shift registers are the data path elements. This is the big picture for digital systems: finite state controllers controlling data paths. A computer is nothing but a finite state controller controlling a data path. The data path is the registers, ALU, I/O controller, etc. and the finite state controller is the state machine that looks at the machine code (translated from the assembly code, which was translated from the source code) and figures out what to do in the data path to execute each instruction. Our UART is simpler than a complete processor, but it has all the features that demonstrate what digital systems can do.
3. It connects and communicates with real hardware. You'll connect your UART to a Teleray 10 terminal and use your UART to echo key strokes on the terminal screen. This means that you have to get the circuit built so that it operates correctly according to someone else's specification. You can't change the way the terminal operates, so you have to get the protocol correct in your UART circuit.

The implementation technology that will be used for this lab includes the Xilinx Field Programmable Gate Array (FPGA). As you will see, the project is very neatly divided into two main parts so it should be easy to divide up the design.

For this lab you can use any of the tools that we've used in this class, including K-maps, Verilog, Synopsys VCS, Synopsys FPGA Compiler II and Xilinx ISE.

Remember to include all written documentation, source files, log files, waveforms etc. that you used when you document your circuit.

What is a UART?

A UART is a circuit that performs serial data communication between two different circuits. If this serial data transmission follows a particular standard protocol, then any device that also uses that protocol will be able to talk to it. One very common standard is the ANSI/EIA-232 standard (formerly known as RS-232). This defines how serial data should be passed from one device to another in terms of transmission speed (baud rate) data format, how to recognize the begin and end of a transmission, how many wires are required and what their meaning is related to the data transfer, the electrical characteristics of the signals that are passed and of the equipment used to pass them, and even what type of connectors can be used and how signals are mapped to pins on the connectors. This standard is technically titled “Interface Between Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE) Employing Serial Binary Data Interchange.” For our purposes, your circuit is the Data Circuit-Terminating Equipment (DCE) (i.e. the computer), and the Data Terminal Equipment (DTE) will be a Teleray 10 terminal.

Like most standards, there is a great deal of information, but fortunately not all of it is necessary for any particular application. In this case, you will be conforming to the the standard format for data interchange, but will be ignoring almost all of the rest of the standard to make things simpler. For example, the EIA-232 standard defines a signal to be off (logic 0) (called “space” or “spacing” in EIA-232-speak) if it is between 3 and 25 volts with respect to the interface ground, and on (logic 1) (called “mark” or “marking” in EIA-232 lingo) if it is between -3 and -25 volts with respect to the interface ground. Note that this mapping is different than you might expect. The on-state is a negative voltage and the off-state is a positive voltage. In our case, the XST-2 board takes care of converting the RS232 logic voltage levels to the appropriate voltage levels of the FPGA. The EIA-232 standard also defines 21 signals that might be part of the serial interchange circuit (control wires of various sorts). In our case, we will only use three wires: one to send serial data, one to receive serial data, and one ground wire. We will also use one standard baud rate of 9600 baud for the communication and not worry about how to select between different baud rates.

Essentially, your UART will be broken down into two major pieces: a sender circuit that takes a byte of data and sends it serially to the terminal, and a receiver circuit that receives serial data from the terminal and collects it into a byte. These state machines should be constructed in a way that allows them to communicate with each other using a 4-cycle handshake such that whatever is received by the receiver is then sent out by the sender. In this configuration your UART will act as an echo circuit for the terminal. Whatever key you press on the terminal will be echoed back to the terminal screen.

Serial Data Format

The serial data that we are interested in sending to and from the terminal is byte-wide ASCII data. ASCII is a standard code for sending alphanumeric data and is actually only 7-bits wide. The 8th bit is often used to indicate the parity of the 7-bit data word and used for error detection. For our circuit, the high-order bit will always be 0, but you should always send it anyway. So, each packet that is sent will consist of 8 bits, 7-bits of ASCII and one 0 in the high-order bit. A table of the ASCII code is shown in Figure 1.

Once we agree to send ASCII, we should also agree on how that data should be send as a serial data stream. The protocol we will use defines that the bits in the byte are passed least-significant bit first in a serial stream. So, send the bits one at a time, starting with the least-significant. How long each bit is asserted in this serial stream depends on how fast your baud rate is. At 9600 baud, for example, each bit will be asserted for $1/9600$ of a second, or about $104 \mu\text{sec}$. Now the problem is how to decide when to look at the data wire in order to see the bit. The DCE and DTE will not be synchronized to a common clock, so

$B_3B_2B_1B_0$	$B_6B_5B_4$							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(8	H	X	h	x
1001	HT	EM)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[k	{
1100	FF	FS	,	<	L	\	l	—
1101	CR	GS	-	=	M]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	-	o	DEL

Figure 1: ASCII Character Codes (See also Table 5.4 on page 290 of your text)

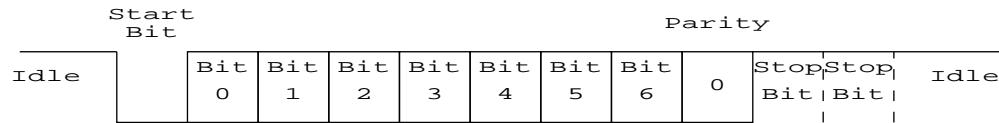


Figure 2: Data Byte Transmission Format

in order to decide when to look at the data line to see new data being passed, they must synchronize with each new byte that is being passed. This is why the UART is “asynchronous” in operation. The data is being passed at a known frequency, but the starting time of each new byte is unknown. So, the receiving circuit must resynchronize at the start of each new byte.

In practice this is quite easy. You only need some sort of protocol that tells you when to expect new data. For our system (and most asynchronous serial protocols in general) the data line must be held in a 1 state (+5v in our case) until a byte is ready to be passed. When a byte is to be sent, the data line drops to 0 (gnd in our case) for one bit time to signal that a byte will follow. This is the *start bit* and its purpose is to wake up the receiver and alert it to the byte that is about to be sent. The 8 data bits then follow, least significant bit first, each asserted for one bit time (which depends on the baud rate). Finally, one or two *stop bits* are sent to indicate the end of the byte. Stop bits are 1-bits that are asserted for one bit time each. In our system the receiver will assume that only one stop bit is sent, and the sender will send two stop bits. This is the most general and safest solution. For example, if the receiver expects a single stop bit and two are sent, nothing bad happens except that some extra time elapses between that byte and the next due to the extra stop bit. On the other hand, if the sender sends only a single stop bit but the external receiver expects two, a mistake might be made. So, it's safer to expect only a single stop bit, but always send two (note that in practice, most terminals and other communication devices have settings to control how many stop bits are sent or expected.) A picture of this data protocol is shown in Figure 2.

Framing Error

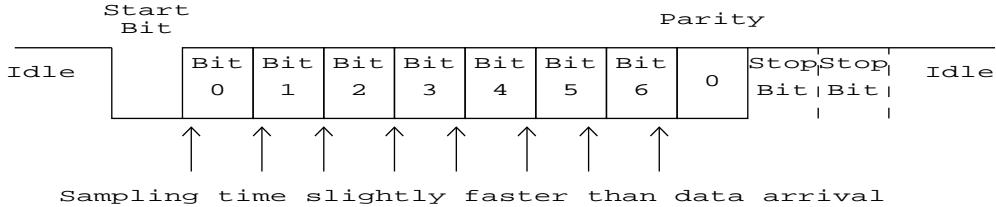


Figure 3: Framing Error

Once you have decided on a baud rate, you know how long each data bit should asserted in order for the receiving circuit to understand the message. So, you need a clock that is running at exactly that baud rate so that once you see a new byte, you know when to look at the bits, right? Well, it's not quite as simple as that. Remember that the DCE and DTE are running at the same baud rates, but they are not synchronized with each other. The clocks might be out of phase, and they might not even be at exactly the same frequency to begin with. Imagine that the clock rates of the DCE and DTE were slightly off. If you are looking at the incoming bits at your rate, but they were being sent just slightly slower than your clock, you might end up looking at bit3 twice, and missing bit7 completely! See Figure 3 for an example. This is known as *framing error* (because the entire word including the start and stop bits is sometimes called a “frame”) and can be a real headache.

The answer is to sample each bit in the middle of its bit time instead of right at the beginning of its bit time. That way, if the timing is off by just a little, you will still get the right result. The easiest way to do this is to run the state machines and whatever other parts you are using at a clock speed that is higher than the actual baud rate of the transmission. That way you can start things up when you see the start bit, and then count into the middle of the bit before you look at what the datum is for the following bit. In your circuit, you should use a system clock that is eight times faster than the baud rate. So, if the baud rate is 9600, then your clock should run at 76.8kHz. The clock on your XSA board runs at a maximum of 100Mhz. So, you need to divide it down, by using the XSTOOLS and within your behavioral code, so that you end up with a system clock of 76.8kHz.

Sender and Receiver Circuits

Your circuit should be split into two main parts: a circuit to receive data from the terminal and collect them into a byte, and a circuit that takes a byte and sends it to the terminal as a serial data stream. These circuits will each look like a state machine that controls some other components, shift registers and counters seem like the most likely choices. The sender and receiver can each be designed completely independently of each other, although the finished circuits will be connected so that they communicate with each other at a higher level. So, let's consider each machine by itself first, and then consider how they communicate.

Sender Circuit

This circuit should take a byte of data and send it serially on a serial line. The circuit will use a four-cycle handshake on the DCE side so that it knows when it has new data to send, and can report when that byte has been sent. On the DTE side there will be a single wire for the serial data communication. A diagram of the sender interface is shown in Figure 4.

The handshake on the DCE side operates as follows: When the DCE has data to send it will raise the *XMT-REQ* line. This is a signal for the sender machine to take the 8 bits on the input and send them out

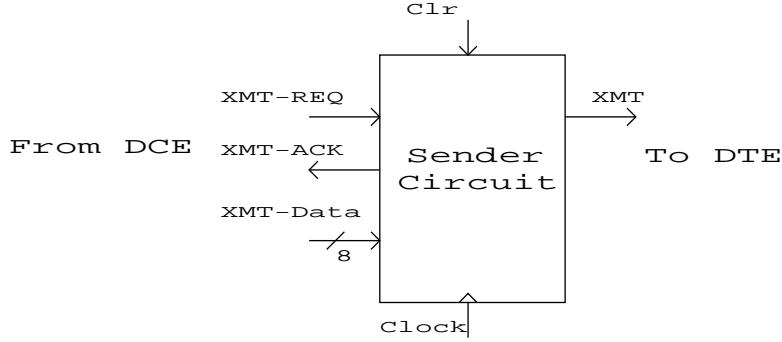


Figure 4: Sender Machine Interface

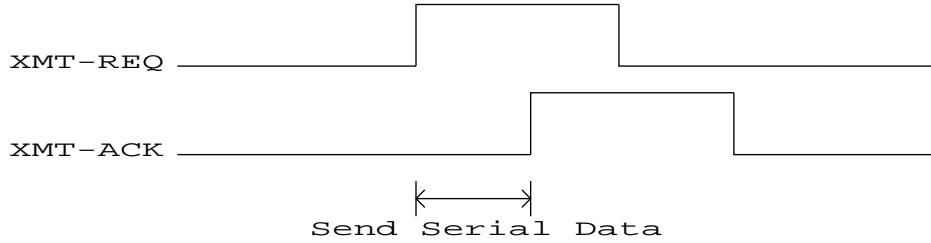


Figure 5: Four-Cycle Handshake for Sender Circuit

to the *XMT* line to the DTE. When the bits have all been sent, the sender will raise the *XMT-ACK* line and wait for the *XMT-REQ* line to go low. When it does go low, sender should lower *XMT-ACK* and go back to waiting for a new byte and another *XMT-REQ* to start things up again. A timing diagram of the four-cycle handshake is shown in Figure 5.

Receiver Circuit

The receiver circuit should monitor the incoming serial line for new data (indicated by a start bit). When it sees that, it should capture the data into a byte and signal that the data has arrived with a four-cycle handshake. The interface for the receiver is shown in Figure 6.

The operation is as follows. The receiver monitors the incoming *RCV* line from the DTE until it sees a start bit. When it sees a start bit it captures the incoming serial data according to the protocol defined earlier. Once the entire 8-bit byte has been captured, the receiver raises the *RCV-REQ* signal to indicate to

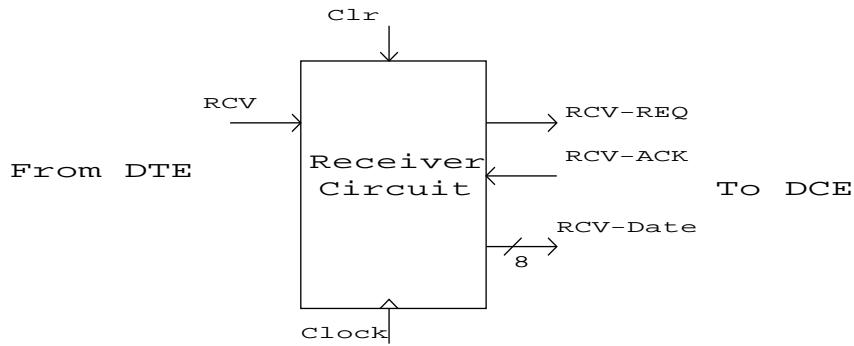


Figure 6: Receiver Machine Interface

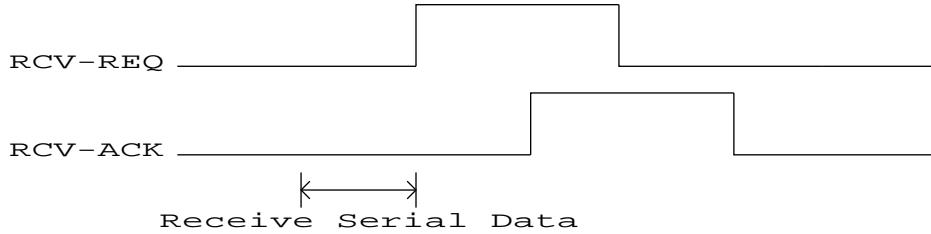


Figure 7: Four-Cycle Handshake for Receiver Circuit

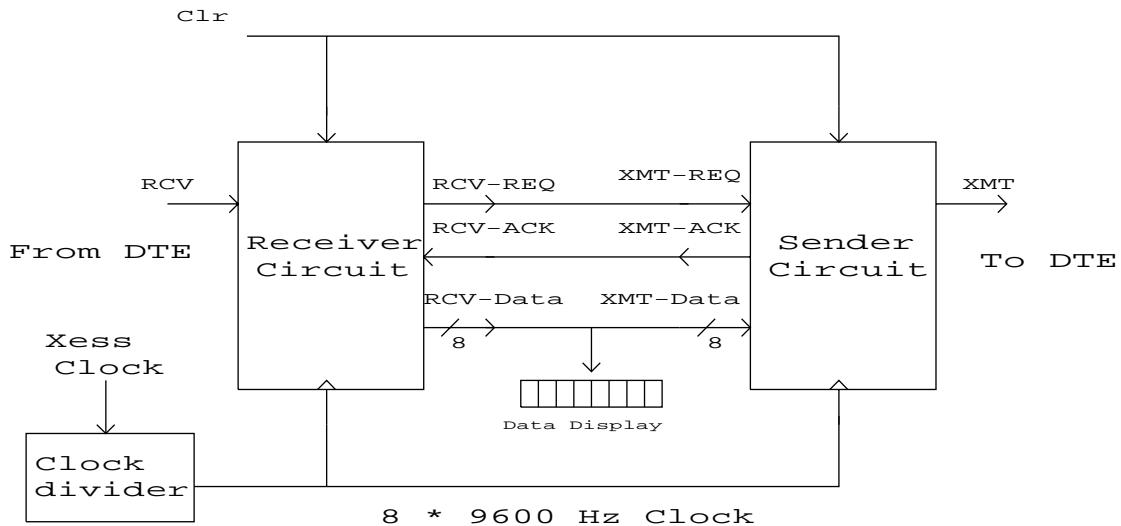


Figure 8: Entire UART System(replace 24 in figure by the appropriate clock dividing factor

the DCE that a new byte is available. When the DCE circuit is done looking at that byte it raises the *RCV-ACK* line. At this point the receiver lowers *RCV-REQ* and waits for *RCV-ACK* to go low before going back to a state where it is waiting for new serial data from the DTE. A timing diagram for the receiver is shown in Figure 7.

Connecting the Sender and Receiver

Once the two machines are working, they can be connected together as shown in Figure 8. In this configuration, data from the terminal will be received on the *RCV* wire by the receiver circuit. Once the data are received they will be passed to the sender circuit and sent back to the terminal on the *XMT* line. So, whatever key is pressed on the T10 terminal will be echoed back to the terminal and displayed on the screen. In addition, just to make sure that everything is working correctly, you should make an 8-bit display on your board that shows the byte that was received by your receiver machine. It should match the character that is being echoed to the terminal. Notice how nicely the four-phase handshake works to allow the sender and receiver circuit communicate. A four-cycle handshake is a very common communication scheme for getting state machines to work together.

Generating the Clock

One required feature of this lab is a clock that runs at a specific frequency. In particular, for data being

transferred at 9600 baud, and a system clock for your project that is eight times faster than this, you need a clock that runs at 76.8kHz. As described previously use XSTOOLS and/or clock divider inside your behavioral code to derive a 76.8kHz clock from the 100 MHz oscillator on the XSA board.

Connecting to the T10 Terminal

Once everything is built and working, you will connect to the T10 terminal to demonstrate your circuit. This connection will be made through the RS232 connector on your XST-2 board. The interface to the terminal is three wires, the *XMT* wire which sends data from your circuit to the terminal, the *RCV* wire which sends data from the terminal to your circuit, and a ground wire so that your circuit and the terminal will be on a common ground. These three wires are brought from the terminal through a 25 pin connector to a 9 pin connector for the RS232 port on your XST-2 board. The *GND* wire is on pin 5, the *XMT* wire will be on pin 3, and the *RCV* wire will be on pin 2. Do necessary pin mapping to your design wrt these pins of RS232 port on the XST-2 board (pin 5 of RS232 connector is internally connected to the ground bus on the XST-2 board).

Whew! This is beginning to sound like a LOT of work. Don't worry, there's a lot of text here, but once you understand the problem it's not that hard. The first thing you should do is make sure that you completely understand the serial data protocol. Now think about how you might use a state machine to control counters and shift registers to make that happen. Once you understand this, it's just a matter of state machine design and you know how to do that (plus a few other components like counters and shift registers, but you know how to do that too). Remember that you must follow four-cycle handshaking in your state machines so that you can get your machines to talk to each other at the end.

The *minimum* requirements for extra circuits are a reset switch (toggle or pushbutton), and an 8-bit LED display that shows the current data that has been grabbed by the receive circuit. You should connect to RS232 port on the XST-2 board and communicate with the terminal using the *GND*, *XMT*, and *RCV* wires and connect the machines to echo back any value received from the terminal. The external connections should be named *XMT*, *RCV*, *CLOCK*, *D[7:0]* (*D7* is the high order bit, and *D0* is the low order bit), and *RESET*. Other names for extra signals are up to you.

Anything else you want to include is up to you. What are some things that you might want to consider? You might want to include lights on the state bits for each of the state machines. If you do this you might also want to include LEDs on the handshake signals between the two machines so you can tell if things are communicating correctly. You might even want to include a set of switches for the sender so that you can choose what to send to the terminal instead of always sending back what you got. This might also include a mode switch that switches between echo-mode and send-the-value-on-the-switches mode. You might want a mode that modifies the value that's being transmitted through your UART just for fun. There might be other things that would be interesting too. This is all up to you.

Keep in mind that if your circuit works completely, that is all the evidence the TA will need (for the demo at least). However, if it does NOT work completely you can demonstrate (for partial credit) that communication is happening by showing the states/data on the LEDs. That means that without LEDs that show the current state of each state machine, it will be hard to argue that your machine is doing anything if it's not working completely.

Final Lab Documentation

The final lab documentation consists of the documentation of everything you did for the UART project. Print out your design documentation and hand it in to your TA. Since this is a final project a more formal well prepared document is expected (with title page, contents sheet, typed/neatly hand written). The things

you need are:

1. A block diagram of the system showing each of the two main parts in detail. Show the State machine as a block and how it connects to the other parts (counters, shift registers, etc) that you used.
2. Complete, documentation of your design approach, the state machines, including state diagram, state transition table, state assignment, encoded state transition table, next state equations, Karnaugh maps for the next state logic, and the output logic.
3. Documented, commented Verilog code.
4. A circuit simulation using Synopsys VCS. The testbench file that applies inputs and checks exhaustively turned in along with a log file of simulation results. If you use a program to generate tests, include the source for that program.

Both the final documentation and the final circuit demo are due in the week of April 20th in your respective labs.