Asynchronous Serial Interface

Introduction

An Asynchronous Serial Interface is a simple low-speed serial communication interface. Today it's mostly used by embedded devices. The simplest version uses a single data line in each direction (Transmit Data and Receive Data) and a ground pin. The RS-232 standard specifies levels of at least ±5 volts but embedded systems often use logic-level signals.

In this lab you will design and implement the transmitter portion of a UART (Universal Asynchronous Transmitter-Receiver) that transmits an 8-bit binary value serially.

A supplied control module will test your UART module by using it to transmit a nine-character text string that is a printable version of your BCIT ID encoded using the ASCII encoding[1].

You will use the Analog Discovery 2 (AD2) oscilloscope, logic analyzer and "UART" protocol analyzer functions to display the waveforms and data.

You must implement the transmitter as a state machine. Quartus will generate a state transition diagram and a state transition table from your HDL description.

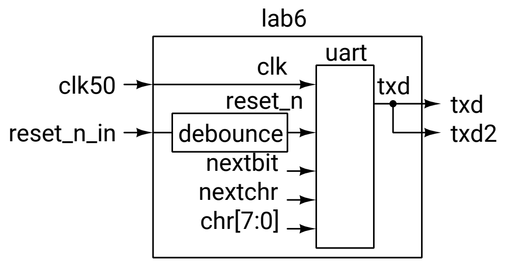
The supplied **lab6.qar** Quartus project archive contains the required code except for the **uart** module, which you must write.

Specifications

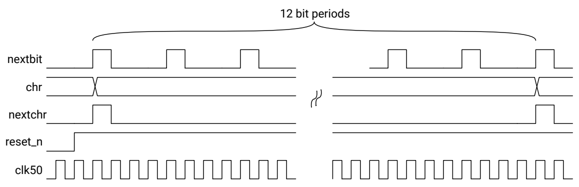
The supplied **lab6** module has a 50 MHz **clk** clock input, an active-low **reset_n_in** input and one-bit **txd** transmit data output .

Your **uart** module will have a 50 MHz **clk50** input, a debounced reset signal named **reset_n**, a **nextchr** signal that is asserted when the UART should start sending the 8-bit character **chr** and a **nextbit** signal to indicate when the next bit of the character should start.

[1] An encoding is a mapping of characters ("glyphs") to numbers. Unicode is the most widely used. ASCII, American Code for Information Interchange, is a subset of Unicode that covers only English-language characters and numbers.



The following diagram shows the timing relationship of the signals at the input to the **uart** module:



Your **uart** module must implement a state machine that operates as follows:

- If **reset_n** is asserted the **uart** is reset; any transmission in progress is halted.
- If **nextchr** is asserted, transmission of a new character is started; **chr** contains the value of this character. You do not need to store it; the value will remain valid until the next character.
- If **nextbit** is asserted, the next bit of the character should be output.

State changes must only take place on the rising edge of **clk50**.

Eight bits per character must be transmitted in order from least-significant bit to most-significant bit. A "1" bit should be transmitted as a high logic level and a "0" bit as a low logic level. An extra "0" ("start bit") must be transmitted before the data bits and an extra "1" ("stop bit") must be transmitted after the data bits. **txd** should be high when the state machine is in the idle state and no data is being transmitted.

The following logic analyzer screen capture shows the **txd** waveform when the character "A" (**8'h41**) is transmitted:

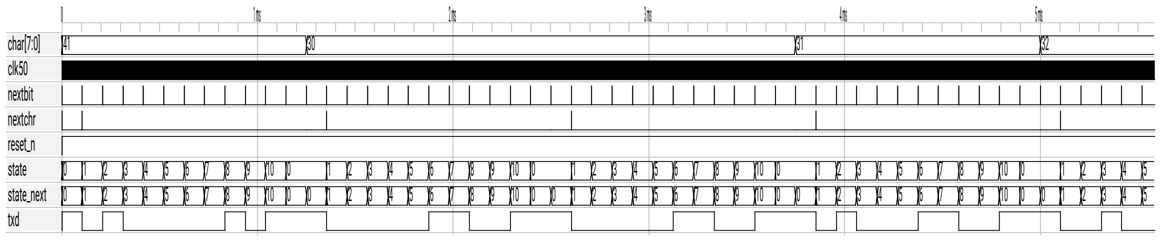


Figure 1: UART Simulation Waveforms.

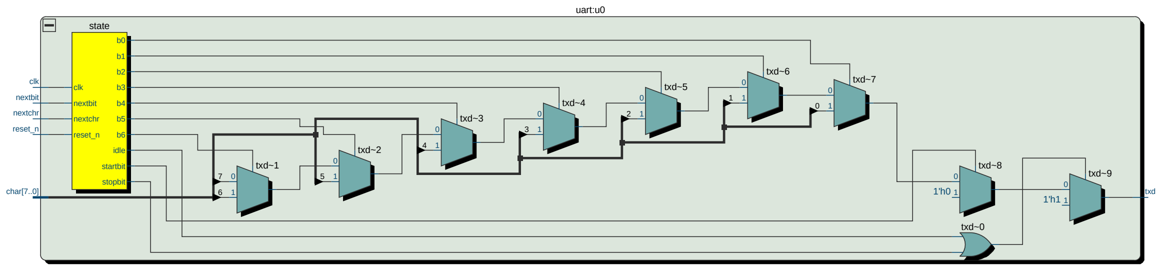


Figure 2: RTL Schematic of UART Module.

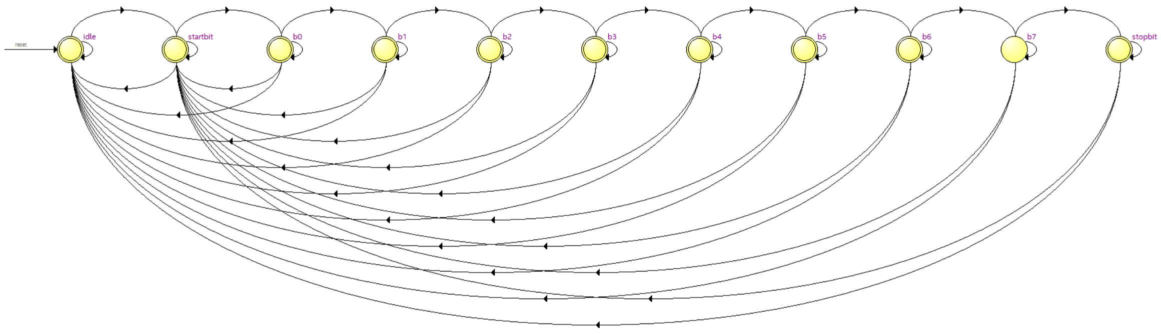


Figure 3: Sample State Transition Diagram.

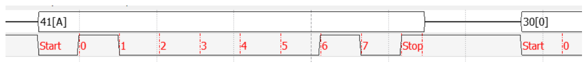


Figure 1 shows the simulation waveforms at the input and output of the UART module. The 50 MHz clock waveform is not visible since it is so much faster than the data. The **nextchr** and **nextbit** signals have a duration of one **clk50** period and are aligned.

Quartus must recognize your design as a state machine. This requires that you follow specific requirements listed in the Quartus Recommended HDL Coding Styles. In particular:

- use an enumerated type of unsigned integer type to define the states (example below),
- do not use the state variable as an output
- keep other operations in the module (e.g. computations) separate from the state machine logic
- include a synchronous reset

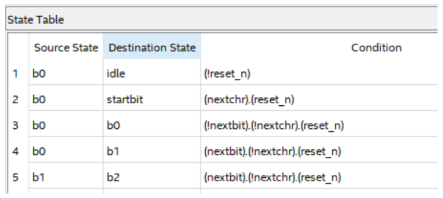
Quartus can recognize state machines implemented using only **assign** and **always_ff** concurrent statements.

An example of a declaration of a suitable enumerated type would be:

```
typedef enum int unsigned
  { idle, startbit, b0, b1, ..., stopbit } state_t ;
state_t state, state_next ;
```

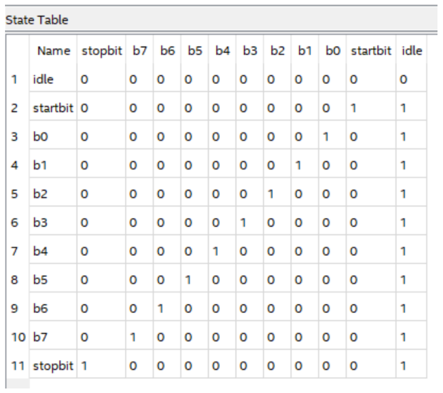
The Quartus RTL netlist viewer will be able to display the state machine as a separate yellow logic block as shown in Figure 2 if you follow the coding guidelines above.

Double-clicking the state machine block will display a state transition diagram similar to that in Figure 3 along with tables showing the state transition conditions and state encodings[2]:



State Table

	Source State	Destination State	Condition
1	b0	idle	(!reset_n)
2	b0	startbit	(nextchr).(reset_n)
3	b0	b0	(!nextbit).(!nextchr).(reset_n)
4	b0	b1	(nextbit).(!nextchr).(reset_n)
5	b1	b2	(nextbit).(!nextchr).(reset_n)



State Table

	Name	stopbit	b7	b6	b5	b4	b3	b2	b1	b0	startbit	idle
1	idle	0	0	0	0	0	0	0	0	0	0	0
2	startbit	0	0	0	0	0	0	0	0	0	1	1
3	b0	0	0	0	0	0	0	0	0	1	0	1
4	b1	0	0	0	0	0	0	0	1	0	0	1
5	b2	0	0	0	0	0	0	1	0	0	0	1
6	b3	0	0	0	0	0	1	0	0	0	0	1
7	b4	0	0	0	0	1	0	0	0	0	0	1
8	b5	0	0	0	1	0	0	0	0	0	0	1
9	b6	0	0	1	0	0	0	0	0	0	0	1
10	b7	0	1	0	0	0	0	0	0	0	0	1
11	stopbit	1	0	0	0	0	0	0	0	0	0	1

You must modify the **`lab6.sv`** file to substitute your BCIT ID for the **`A00123456`** value.

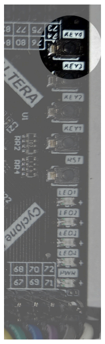
The **`lab6.sv`** file contains a testbench named **`lab6_tb.sv`** that you can use to simulate your design.

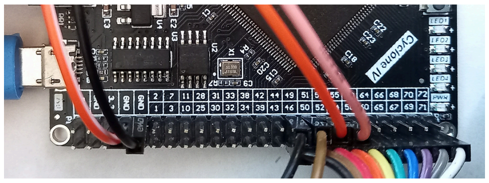
FPGA I/O

The default pin assignments for this lab use the on-board pushbutton **`KEY4`** to trigger the transmission of a string and output the serial data on FPGA pins **`53`** (**`txd`**) and **`55`** (**`txd2`**).

The photo below shows how the AD2 connects to the FPGA pins: the analog negative reference and the digital ground (orange/white and black leads respectively) connect to two ground pins (labelled **`GND`**), the AD2 'scope's channel 1 (orange) connects to pin 53 and the digital input **`DI0`** (pink) connects to pin 55:

[2]Note that in this case Quartus has chosen a one-hot encoding in which the idle (reset) state is active-low.





Use of AD2

You will use the AD2 oscilloscope, logic analyzer and protocol windows to verify and troubleshoot the operation of your design.

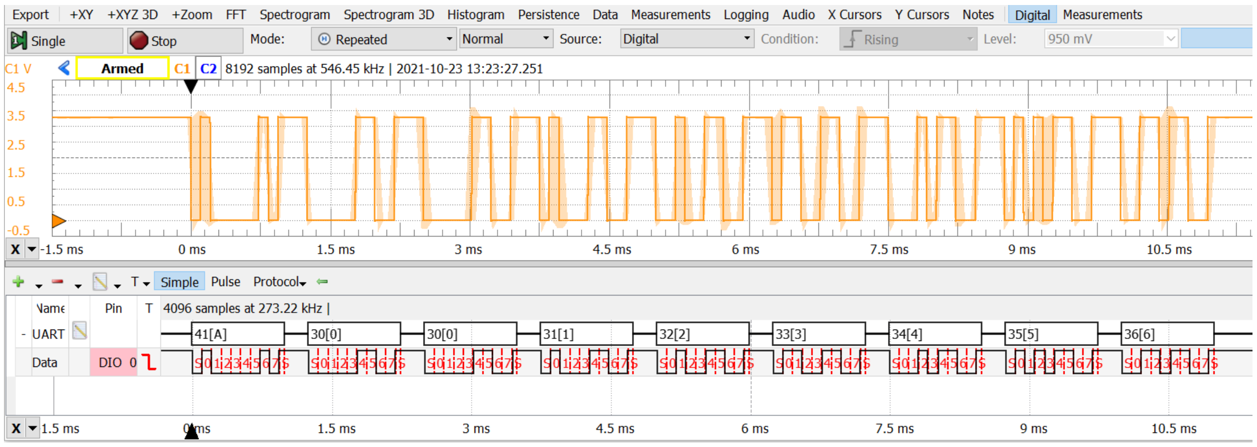
Oscilloscope. A 'scope channel can be connected to the **`txd`** output to verify the voltage levels and check for signal integrity issues such as noise, glitches or ringing. You can trigger on the falling edge of **`txd`** to capture the start of the transmitted waveform.

Logic Analyzer. The logic analyzer can be used to display digital signals. It can display multi-bit bus values (not relevant in this case) and multi-bit-duration values (as for the UART configuration used here). Figure 4 shows the transmitted serial data using a configuration called "UART."

The trigger (T column) has been set to the falling edge of the Data signal and the Protocol options have been set for 9600 bits per second, 8 bits per character and normal polarity (low for a "1" bit).

Protocol Analyzer. The protocol analyzer can be used to decode even more complex protocols such as those including device addresses and variable-length fields (neither used by asynchronous interfaces). The following screen captures show the "UART" decoding. Figure 5 shows an example of the protocol analyzer display showing the received characters (the reset button was pressed twice).

Note: There seems to be a conflict between the AD2 and the USB-Blaster drivers. You may need to disconnect the AD2 to program the FPGA.

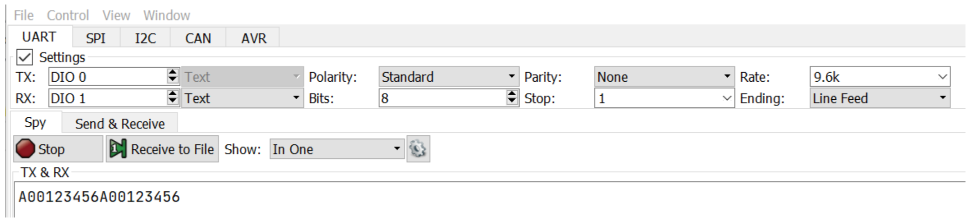


Mode: Repeated Normal Source: Digital Condition: Rising Level: 950 mV

8192 samples at 546.45 kHz | 2021-10-23 13:23:27.251

4096 samples at 273.22 kHz |

Figure 4: ’Scope and Logic Analyzer Display.



Settings

TX: DIO 0 Text Polarity: Standard Parity: None Rate: 9.6k

RX: DIO 1 Text Bits: 8 Stop: 1 Ending: Line Feed

Show: In One

TX & RX

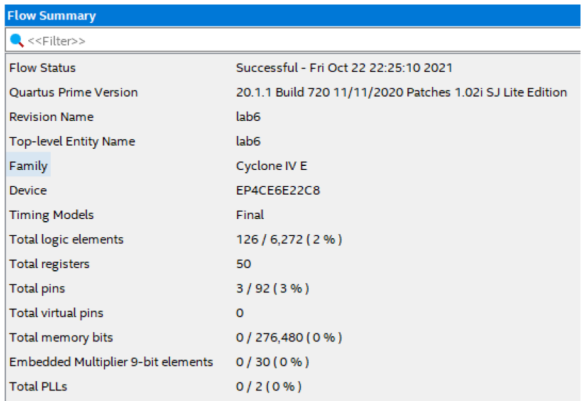
```
A00123456A00123456
```

Figure 5: Protocol Analyzer Display.

Submission

To get credit for completing this lab, submit a PDF document containing the following to the Assignment folder for this lab on the course website:

1. A listing of your **`uart.sv`** System Verilog file.
2. The RTL schematic (Tools > RTL Netlist) similar to Figure 2.
3. A screen capture of the state transition diagram similar to Figure 3.
4. A screen capture of your compilation report similar to:



Flow Summary

<<Filter>>

Flow Status	Successful - Fri Oct 22 22:25:10 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 Patches 1.02i SJ Lite Edition
Revision Name	lab6
Top-level Entity Name	lab6
Family	Cyclone IV E
Device	EP4CE6E22C8
Timing Models	Final
Total logic elements	126 / 6,272 (2 %)
Total registers	50
Total pins	3 / 92 (3 %)
Total virtual pins	0
Total memory bits	0 / 276,480 (0 %)
Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)

5. Screen captures of the AD2 logic analyzer and protocol analyzer similar to those in Figures 4 and 5 demonstrating the operation of your interface. They should both show your full BCIT ID.