ECE430/530 Digital Hardware Design
Project 02 Fall 2022

Universal Asynchronous Receiver Transmitter (UART)

The *Universal Asynchronous Receiver* Transmitter (UART) is a hardware module used a simple interface to provide a communication protocol over a serial line. The objective of this project is to design, verify, and implement an UART interface, which will be tested, on a FPGA prototyping board.

Introduction

UART is a circuit that sends parallel data on a serial line. It includes a transmitter, a receiver and a CPU interface. The transmitter is essentially a special shift register that loads data in parallel and then shifts it out bit by bit at a specific rate. The receiver, on the other hand, shifts-in data bit by bit and then reassembles the data. Each block also contains a FIFO (created by the FIFO Generator in Vivado). The CPU interface acts as the controller and contains a set of registers used for configuring its operation.

Asynchronous V.s. Synchronous

- Synchronous transfer does not transfer extra bits. However, it requires clock signal as shown in Figure 1

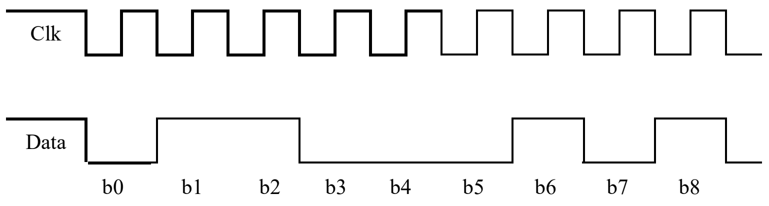

Figure 1 – Synchronous Data

- Asynchronous transfer does not require a clock signal. However, it transfers extra bits (Start and stop bits) during data communication as seen in Figure 2. LSB is transferred first at a given bit rate (BAUD rate = number of times signal changes state per second)

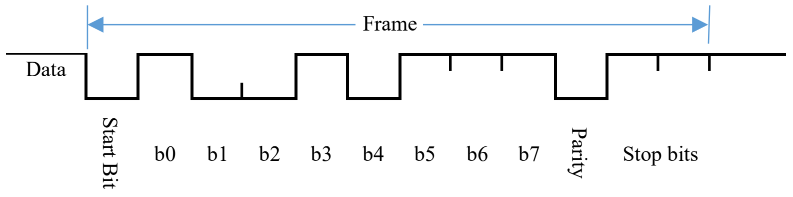

Figure 2 - Asynchronous Data

UART Overview

The UART consists of three main blocks as shown in Figure 3

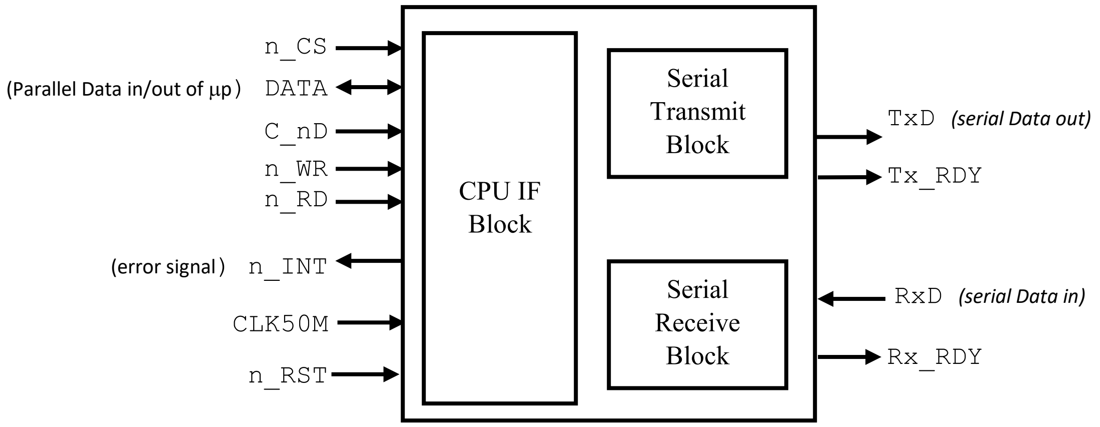

Figure 3 – Basic UART Block Diagram

Interface signal

The UART has the following CPU Interface signals:

- **CLK50M:** Clock input for the UART, nominally 50MHz. It is used to generate internal device timing. No external inputs or outputs are referenced to this clock.
- **n_RST:** Active-low reset input driven from the CPU. It places the UART in an idle mode until it is programmed.
- **DATA:** This is the 8-bit bi-directional multiplexed data bus interface to the CPU. It carries data received from or sent to the CPU. Two separate input and output buses should be implemented on the FPGA to avoid using an on-chip tri-state buffer.
- **n_RD:** A “low” on this input informs the UART that the CPU is reading data or status words from the UART during a read cycle.
- **n_WR:** A “low” on this input informs the UART that the CPU is writing data or control words to the UART during a write cycle.
- **C_nD:** This input, in conjunction with n_RD and n_WR inputs, indicates what kind of information is on the DATA bus. If “low”, the data bus is carrying a data character. Otherwise, it is a status information or a control word.
- **n_CS:** This is the chip select signal which enables the UART. No read or write occur unless this signal is “low”
- **n_INT:** Active-low interrupt output driven by the UART in case of errors.
- **Tx_RDY:** Active-high transmit-ready status output driven by the UART. This signal indicates that the UART is ready to accept a data character.
- **Rx_RDY:** Active-high receive-ready status output driven by the UART. This signal indicates that the UART contains a character that is ready to be input by the CPU.

It also has the following serial interface signals to external peripheral serial devices.

- **TxD:** The serial transmit line.
- **RxD:** The serial receive line.

Serial Data Frame

In general, a serial data frame has the format shown in Figure 4

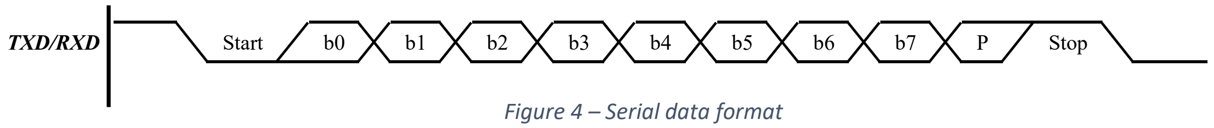

Figure 4 – Serial data format

- Signal is 1 (high voltage) when the system is idle
- A start bit is placed before the data; and, optionally, stop bit(s) are placed at the end of data
- Start bit is 0 and stop bit(s) are 1
- LSB is first transmitted or received
- **Baud rate**: number of signal or symbol changes that occur per second
- **Number of Data bits:** they can be either 7 or 8
- **Stop bits:** number of stop bits can be either 1 or 2
- **Parity bit**: are used for error checking; even, or odd, or no parity

Read and Write data Cycles

The formats required for read and write cycles are illustrated in Figure 5 and Figure 6 , respectively. These figures illustrate writing and reading data. Reading status word and writing control words are similar with the exception of the C_nD logic level. Pay attention to the timing or the n_WR and n_RD signals since they are used by the UART for clearing or setting some bits in the registers. (these cycles will be implemented in the testbench for testing, They are actually generated by the CPU unit.

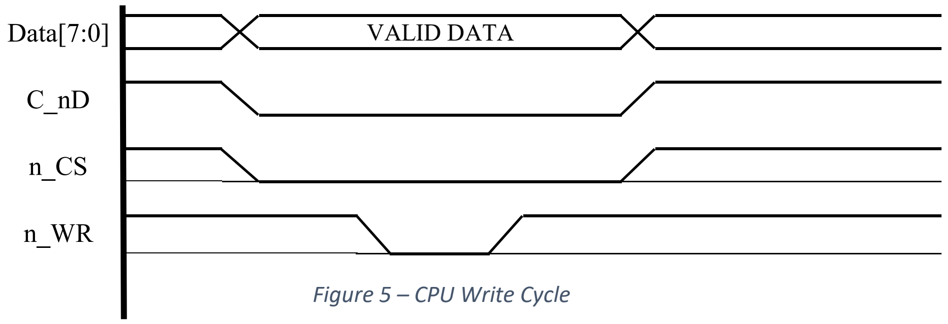

Figure 5 – CPU Write Cycle

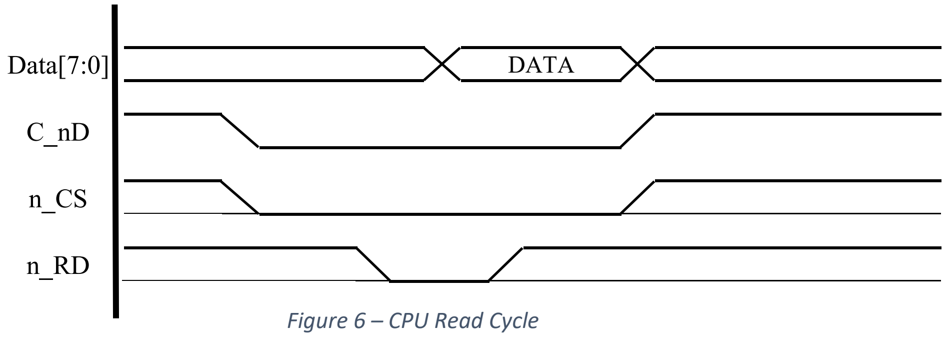

Figure 6 – CPU Read Cycle

Asynchronous Data Recognition and Baud Rate Generator

Since clock information in asynchronous data transmission is conveyed from the transmitted signal, the receiver can retrieve the data bits only by using predetermined parameters about the data frame.

An ***oversampling scheme*** should be used to estimate the middle points of transmitted bits and then retrieve them at these points accordingly. The most commonly used sampling rate is 16 times the baud rate, which means that each serial bit is sampled 16 times. While the receiver has no information about the exact onset time of the start bit, the estimation can be off by at most 1/16. The subsequent data bit retrievals are off by at most 1/16 from the middle point as well. Therefore, sampling ticks should be exactly 16 times the UART's designated baud rate. Because of oversampling, baud rate can be only a small fraction of the system clock rate and thus this scheme is not appropriate for high data rates.

Assume that the communication uses N data bits and M stop bits. The oversampling scheme works as follows:

1. Wait until the incoming signal becomes 0, the beginning of the start bit, and then start a sampling tick counter.
2. When the counter reaches 7, the incoming signal is at the middle point of the start bit, clear the counter to 0 and restart.
3. When the counter reaches 15, the incoming signal progresses for one bit and reaches the middle of the first data bit. Retrieve its value, shift its value into a register, and restart the counter.
4. Step 3 is repeated (N-1) times to retrieve the remaining data bits.
5. If the two parity bits are included, step 3 is repeated two more times.
6. Step 3 is repeated M more times to obtain the stop bits.

To avoid creating a new clock domain and violating the synchronous design principle, sampling signals should function as **enable pulses** to the receiver block rather than a clock signal. As an example, for the 19,200 baud rate, the sampling rate has to be 307,200 (i.e., 19,200 * 16) ticks per second. Since the system clock is 50 MHz, the baud rate generator needs a mod-163 (i.e. $\frac{50 * 10^6}{307{,}200}$) counter, in which a one-clock-cycle tick is asserted once every 163 clock cycles. Since the UART operates at different baud rates, a parameterized mode-m counter should be used for this process by setting its mod parameter to the desired value.

UART CPU Interface bus

The CPU I/F block is responsible for recognizing which cycle is in-progress then generates internal control signals required to perform these tasks. It is also capable of accessing the various UART registers and other components, explained later, and updating them. It receives a set of control interface signals from the CPU and interpret them. These different functions are described in Table 1 (if all inputs are zeros, it means NOP). Generally, in asynchronous data delivery and reception, the signals `DATA, C_nD, n_CS, n_WR, n_RD` as well as the `n_INT` should be synchronized with the system clock `CLK50M`. On the other hand, data on the serial lines `RxD, TxD` are sent/received according to the baud rate.
The bi-directional data bus should be split into two buses `Data_in` and `Data_out` since all tri-state buffers are only available on the I/O pads of the FPGA and they can't be used for simulation

Table 1 – Status of Control bits for UART operation

C_nD	n_RD	n_WR	n_CS	CPU IF function
0	0	1	0	Read Data: UART Rx_FIFO → DATA bus
0	1	0	0	Write Data: DATA bus → UART Tx_FIFO
1	0	1	0	Read Status Register: Status → DATA bus
1	1	0	0	Write Configuration Register: DATA bus → Config Reg.
Note: All other combinations are not valied.				

UART Registers

The UART contains a set of registers located in the CPU I/F unit to allow its configuration by the CPU and to control its operation.

The following is a description of the registers and their usage

1. ***Data Buffers:*** The UART has two separate buffers: a Transmitter FIFO (Tx_FIFO) placed in the transmitter block and a Receiver FIFO (Rx_FIFO) placed in the receiver block. Data, representing a character in ASCII format, is written into Tx_FIFO and read from the Rx_FIFO through the CPU IF block.
2. ***Configuration Register (CR):*** Prior to starting data transmission or reception, the UART must be loaded with a configuration word generated by the CPU. This defines the UART's operating parameters and must follow a Reset operation.

I_Rst	Clr_EF	Par1	Par0	S_num	D_num	Bd_rate1	Bd_rate0

- BD_Rate (1:0) – These two bits are used to specify the baud rate which can be:
  - 9600 for Bd_Rate = 0 0
  - 19,200 for Bd_Rate = 0 1
  - 57,600 for Bd_Rate = 1 0
  - 115,200 for Bd_Rate = 1 1
- D_num – This bit specify the number of data bits which can be
  - 7 if D_ num = 0
  - 8 if D_num = 1
- S_num – This bit specify the number of stop bits which can be
  - 1 if S-num = 0
  - 2 if S_num = 1
- Par (1:0) – These two bits are used to specify desired parity scheme
  - No parity if Par = 0 0
  - Odd parity if Par = 0 1
  - Even parity if Par = 1 0
  - Invalid if Par = 1 1
- Clr_EF – If set, all error flags are cleared.
- I_Rst – if set, an internal reset is issued to clear status register and returns the UART back to the configuration mode.

3. ***Status Register (SR):*** The contents of this register represent the status of the UART. This register can be cleared by the global reset n_RST signal or via an internal reset command. Some bits of these register have identical meaning to external output ports of the UART

0	0	0	OE_Fg	FE_Fg	PE_Fg	Tx_Rdy	Rx_Rdy

- Tx_Rdy – This bit is set to logic 1 to inform the CPU that the UART can accept a new character for transmission. i.e., the *TX_FIFO* in not full. This bit is reset to logic 0 when the *TX_FIFO* is full and data were not yet read by the external serial peripheral device.
- Rx_Rdy – This bit is set to logic 1 when at least one character is in the *RX_ FIFO*. It will go inactive when there are no more characters in the FIFO.
- PE_Fg – The Parity Error Flag is set to indicate that the data character in the FIFO does not have the correct parity.
- FE_Fg – The framing Error Flag is set to indicate that the character in the FIFO does not have valid stop bit(s) sequence detected.
- OE_Fg – The Overrun Error Flag is set when a character is being received by the receiver block while the RX_FIFO is full. An interrupt is generated and This character is lost. The interrupt routine will read all data from the receive FIFO

UART Other Components

1. ***Receiver Block:*** It consists of two components:

- ***Receiver Controller:*** This is a Finite State Machine (FSM) that process sample the serial data input from UART RxD line and place it in a receive FIFO. It has the following ports:
  - Inputs: CLK50MHZ, n_rst, RxD, Baud_tick,
    number of data bits, number of stop bits, and parity (even, odd, no parity)
  - Outputs: Rx_Done, an 8-bit Rx_Dout , error flags

  The controller has the following characteristics:
  - The FSM only changes the state only when the Baud_tick is asserted (as an enable signal)
  - The FSM should include two counters:
    - Tick_no: to Keep track of the number of ticks which are 7 at the start, 15 for each data bit, parity bit, and stop bits
    - Data_Stop_no: to keep track of the number of data bits or stop bits
  - Rx_Done signal is asserted for one clock cycle after the receiving cycle is completed. It should be connected to the FIFO's *RF_write* signal to write the corresponding *RX_Dout* into the receive FIFO through the *RF_data_in port.*

  The Serial Receive Block reads the serial incoming data from the *RxD* line at the desired baud rate, check it for data errors and places the data and errors into the receiver FIFO (*RX_FIFO*) as long as it is not full.

  The receiver activates the *Rx_RDY* signal when the receive FIFO's *RF_Empty* signal is not active to inform the external CPU that the receive FIFO has data to be read. The CPU obtains the data from the FIFO's *RF_data_out.* After the CPU receives the word, it asserts the FIFO's *RF_read* signal for one clock cycle to remove the corresponding item.

- ***Receive FIFO:*** This FIFO is 10-bit (8 data bits plus a parity bit and a framing error bit) by 4-words generated using Vivado IP catalog. Error bits are transferred to the status register when the character data bits are on the top of the FIFO. It has the following interface signals:
  - ***RF_clock –*** The global UART 50 MHz clock.
  - ***RF_reset*** – an active LOW input causes the FIFO to be reset and hence all data currently in the FIFO to be lost. Operation of this signal during serial data reception will potentially result in corrupt data.
  - ***RF_data_in*** – After the serial data frame is received on the RxD line. It gets processed by the receive block where the character is extracted and data errors are checked. Then they are placed on this bus to be written into the FIFO.
  - ***RF_write*** – An active high indicates that the data currently being applied to the RF_data_in port is to be written to the FIFO on the next rising edge of the RF_clock. A write operation will take place on every rising clock edge on which this signal is active. The FIFO ignores data should it becomes full and generated an overrun error.
  - ***RF_data_out*** – The parallel data out of the FIFO to be read by the CPU
  - ***RF_read –*** when this signal is activated the next data stored in the FIFO is pushed to the top.
  - ***RF_full*** – When the FIFO buffer is full, this output becomes active (high). serial data currently being received is ignored. An overrun error is generated.
  - ***RF_Empty*** – When the 4-word FIFO buffer is empty, this output becomes active (high) and CPU can't read any data form the UART. When it is not active.

2. ***Transmitter Block:*** The organization of the UART transmitting subsystem is similar to that of the receiver. Except that the "CPU IF" writes the Tx_FIFO buffer, and the UART transmitter reads from the Tx_FIFO FiFO butter. Therefore, it is essentially a shift register that shifts out data bits at a specific rate. The rate can be controlled by one-clock-cycle enable ticks generated by the baud rate generator. Because no oversampling is involved, the frequency of the ticks is 16 times slower than that of the UART receiver. Instead of introducing a new counter, the UART transmitter usually shares the baud rate generator of the UART receiver and its FSM uses an internal counter to keep track of the number of enable ticks to shift a bit out every 16 enable ticks.

   The *Tx_RDY* signals the external CPU that the UART transmitter is ready to accept a data character (Tx_FIFO is not full). The CPU writes it into the *(Tx_FIFO)*. The transmitter block reads It from the *Tx-FIFO*, adds a start bit (low) followed by the data bits (least significant bit first), inserts a parity bit (if needed), and the desired stop bit(s) to the character.

- ***Transmit FIFO:*** This FIFO is 8-bit by 4-words generated using Vivado IP catalog
  - ***TF_clock –*** The global UART 50 MHz clock.
  - ***TF_reset*** – an active LOW input causes the FIFO to be reset and hence all data currently in the buffer to be lost. Operation of this signal during serial data transmission will potentially result in corrupt data.

- ***TF_data_in*** – The parallel byte to be written into the FIFO.
- ***TF_write*** – An active high indicates that the data currently being applied to the *TF_data_in* port is to be written to the FIFO on the next rising edge of the *TF_clock*. A write operation will take place on every rising clock edge on which this signal is active. The FIFO ignores data should it becomes full.
- ***TF_read –*** an active high indicates that there is data in the FIFO needs to be read, serialized, and transmitted. Data should not be read from when FIFO is empty
- ***TF_data_out*** – The parallel byte read out of the FIFO and handed to the serial block for sterilization and transmission
- ***TF_full*** – When the FIFO buffer is full, this output becomes low and the TX_RDY signal will be low as well indicating that the CPU is not allowed to transmit any data to the UART.
- **TF_Empty** – When the FIFO buffer is empty, this output becomes active HIGH and no serial data available to be transmitted on the TxD line.

3. ***Clock Generator:*** Use the Vivado project manager under the flow navigator to add a clock generator IP using the command "IP catalog> FPGA Features and designs> Clocking> Clocking Wizard", Use this IP to generate the 50MHz clock and a copy of the input clock from the board Master Clock. Use these two outputs for the UART design and the OLED display, respectively. The generated instantiation template can be used to instantiate it into your top design.
4. ***Integrated Logic Analyzer:*** Refer to the enclosed document about the Vivado ILA to instantiate an ILA core in the design and connect all I/O, data busses, and serial Tx/Rx lines to it. It is a great tool for debugging and demonstrating your results. You may be asked to demonstrate the functionality of the UART through it.

Simulation of Individual Components

Simulate **each** of the subcomponents of the UART to confirm their correct functionality before connecting all the components to construct the UART. Report these simulations.

Function Verification via simulation

Write an automated testbench using tasks to verify normal operation and erroneous conditions for the UART. The test bench should be commented thoroughly explaining each part of the testbench and the test cases.

At minimum, the testbench should include the following tasks:

- Tasks to emulate the CPU read and write cycles
- Write a command file to emulate programming the UART and reading/writing configuration and data to the UART. (PClk, Cnfg, Send, Recv – each should be followed by a value)
- The testbench should read the commands from the file and use the tasks to test the following situations:

  - Reset operation
  - A normal byte transmission and reception for all combinations of number of data bits, stop bits, parity bits at different BAUD rates.
  - Check on Error flags: Parity error, Framing error, and Overrun error.

- The testbench also should use system tasks to print the value of the CPU data received and the serial data transmitted into a file as follows:
    - Configuration data is <data in binary or hex in the form 0X??>
    - RxD data is <data received in binary or hex in the form 0X??>
    - TxD data is <data transmitted in binary or hex in the form 0X??>

Design Testing via Implementation

- Implement the UART design on the FPGA prototyping board. loop back the TxD and RxD lines on the board.
- Data and configuration words should be represented by the switches [SW7 : SW0] (LSB switches are on the board and the MSB are on the PMOD)
- Use the following push buttons to represent the control signals coming from the CPU

  { PB0 – Reset ; PB1 → C-nD ; PB2 → n_RD ; PB3 → n_WR ; n_CS is always active }

- Use LED0 for Rx-Rdy, LED1 for Tx_Rdy, LED2 for parity error and LED3 for Framing error

Use this information to trigger the ILA With the correct Write Cycle and Read Cycle and take a snap shot of the waveforms of the TxD line and RX_FIFO lines showing the transferring and receiving of the following two test cases:

1. Baud rate 9600, 8 data bits, 1 stop bit and odd parity – Data Character is D = 0x68
2. Baud rate 115,200, 7 data bits, 2 stop bits, and even parity – Data Character is G = 0x71

Submission

A formal report is required, include simulation of every component of the design marked with markers placed at critical points with comments. The conclusion section should document your efforts in the design. What worked, and what didn't, what functions are dropped, along with any other things you want to tell me about the project

Include all Verilog design files and IPs used. The well commented testbenches are a must (If I am not able to understand your comments, I will not be able to grade it fairly).

A narrated video demonstrating the operation of the implemented design and an in-class Demo.