北京交通大學

本科毕业设计（论文）

Bachelor Thesis

FPGA Design of UART Instruction Controller

A project submitted in partial
fulfillment of the requirements for the degree of
Bachelor in Communication Engineering

By

HABTE ENDALE LEGESSE

17219003

Supervised By

Li Jincheng

Undertaking

This is to declare that the project entitled "FPGA DESIGN OF UART INSTRUCTION CONTROLLER" is an original work done by undersigned, in partial fulfillment of the requirements for the degree "Bachelor of Science in Communication Engineering" at Communication Engineering Department, School of Electronics and information Engineering, Beijing Jiao tong University.

All the analysis, design and system development have been accomplished by the undersigned. Moreover, this project has not been submitted to any other college or University.

Contents

Chapter One: Introduction

1.1 Background of Research

In the 21st century, with the rapid development of electronic technology, high and new technologies are changing with every passing day. The traditional design method is gradually withdrawing from the stage of history, replaced by chip design technology based on EDA technology, which is becoming the mainstream of electronic system design. Field programmable gate array (FPGA) is one of the most widely used types of programmable application specific integrated circuits (ASIC). FPGA has good performance, high reliability, large capacity, small size, micro power consumption, fast speed, flexible use, short design cycle, low development cost, static reprogramming, dynamic in system reconfiguration, hardware functions can be modified by programming like software, which greatly improves the flexibility and versatility of electronic system design. Electronic engineers and scientific researchers can use this kind of devices to design the required ASIC in the office or laboratory, which greatly shortens the product development cycle and reduces the cost.

1.2 Literature Review

UART (Universal Asynchronous Receiver Transmitter) is a popular methodology of serial asynchronous communication [1]. Serial communication is vital to computers and allows them to communicate with the low-speed devices such as keyboard, mouse, modems etc[2]. Asynchronous serial communication requires less transmission lines, high reliability and low cost. It is widely used in data exchange between microcomputer and peripherals. UART is a key module for communication between various devices[3]. it allows full-duplex communication in serial link, thus has been widely used in the data communications and control system[4]. When one device needs to communicate with another device, digital signal is usually used. This made UART the first choice to be implemented among data communication Methods. Because of its high reliability, long transmission distance and the simple line, UART is becoming more extensive serial data communication circuit[5]. UARTs like RS-32 are a standard concerning electrical level of data and control signals between data terminal equipment (DTE) and data communication equipment (DCE)[6].

To simplify its complexity and to reduce the cost, UART can be integrated into FPGA[5]. This method of UART can significantly enhance its flexibility, and its portability and also can reduce the waste of energy and design materials. A UART that is designed for FPGA can serve as an interface for FPGA based embedded systems which uses soft core processors[7]. This reduces external routing problems and cost as FPGA has huge number of logic gates unused.

FPGAs could be the best solution for reconfiguration of the system hardware modification[7]. FPGAs are a logic device that contains a two-dimensional array of generic logic cells and programmable switches. The logic cell of FPGA can be configured (i.e., programmed) to perform a simple function, and a programmable switch can be customized to provide interconnections among the logic cells[2]. This programmability and configurability features of FPGAs have let them to be the most preferable digital chip for many researchers and digital system designers. Hence, they are the first choice in designing of this project too.

Hardware description language which is abbreviated as HDL is very helpful in formalizing and representing a digital system at hand. Hence, it can be implemented on a target FPGA platform[8]. The most widely used HDLs are Verilog HDL and VHDL (Very High-Speed Integrated Circuit Hardware Description Language). This project design has used the former HDL, which is Verilog HDL to design the UART instruction controller modules. VHDL means very high-speed Integrated circuit hardware description language and Verilog - means verify the logic. Verilog is an alternative language to VHDL for specifying RTL for logic synthesis VHDL similar to Ada programming language in Syntax Verilog similar to C/Pascal programming language[9].

1.3 Research Scope and design specifications

This research paper introduces a method to design UART instruction controller based on Field Programmable Gate Array (FPGA). The design specifications are stated as follow:

- The port type to be designed is：UART
- The system clock frequency is：24MHz
- The type of FPGA used is : Altera Cyclone
- The baud rate used in the design is: 9600
- The Information Set Architecture specified for this design are: a+b, ~a, a^b, a*b

This research paper is comprised of five chapters. Chapter one is an introduction section where the research background, the literature review and design scope and specifications are introduced. Chapter two is the section where basic required knowledge for the design is addressed. The UART introduction, the FPGA introduction and the Verilog HDL introduction are the core topics instigated in this chapter. Chapter three of this research paper clarifies the proposed design of this project. The proposed design is designed in total of four Verilog HDL modules namely UART_RX, UART_TX, cmd_pro and UART_TOP modules. Each of those modules are briefly explained consequently. Chapter four is to introduce readers of this paper to the FPGA testing of the proposed design. The testing development board Altera Cyclone and the serial debugging assistant are briefly introduced in this chapter. Finally, chapter five concludes the whole research and provides author experiences and problem faced during design process and in writing this research paper.

Chapter Two: UART, FPGA and Verilog HDL Instruction Controller

2.1 UART (Universal Asynchronous Receiver Transmitter)

For many years, serial port has been an integral part of most computer operating systems in use today and has been considered to be one of the most basic external connections in the computer and those Serial ports rely on a special controller chip called Universal Asynchronous Receiver/Transmitter (UART)[10]. UART is a hardware communication protocol that uses asynchronous serial communication with configurable speed. It is a universal serial data bus for asynchronous communication. The bus has two-way communication and can realize full duplex transmission and reception[11]. In embedded design, UART is widely used to communicate with computers and some peripheral devices with low-speed requirements.

UART consists two major blocks that is receiver and transmitter and is asynchronous because the receiver and the transmitter clock are not synchronized with each other. Hence, the word asynchronous transmitter is based on the start and stop bit to receive or transmit data[12]. An Asynchronous means there is no clock signal to synchronize the output bits from the transmitting device going to the receiving end.

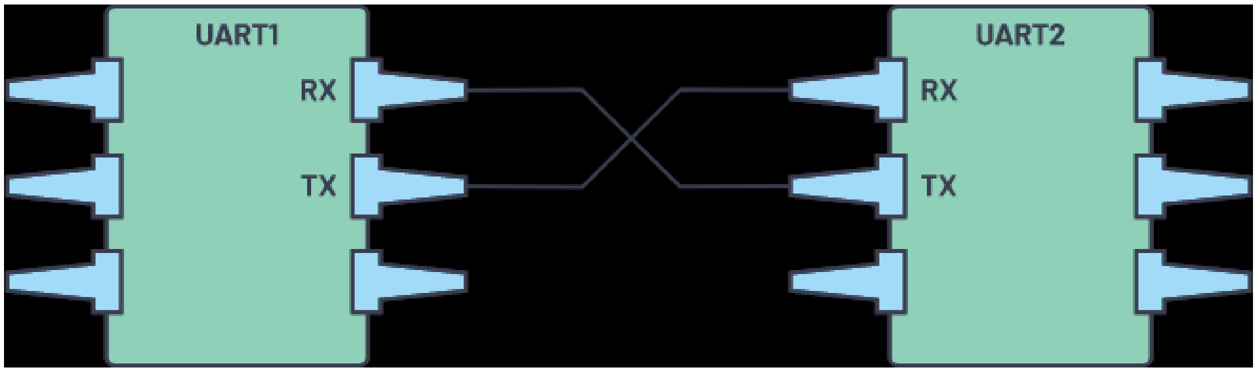


Fig 2.1.1 Representation of UART communication

A single UART has two signal pins called the transmitter pin (TX) and the receiver pin (RX). Thus, the devices which want to communicate with UART protocol have to have respective transmitter and receiver pins to communicate each other. As it has been shown in the above figure, the UART1 RX pin connected to the UART2 TX pin and the UART1 TX pin is connected to the UART2 RX pin. The main purpose of a transmitter and receiver line for each device is to transmit and receive serial data intended for serial communication.

The UART transmitter is connected to a controlling data bus that sends data in a parallel form. Then, the data will be transmitted on the transmission line (wire) serially, bit by bit, to the receiving UART. The UART receiver, in turn, will convert the serial data into parallel for the receiving device. UART lines, the TX line and the RX line serves as the communication medium to transmit and receive one data to another.

UART and some other serial communications needs a signal called Baud Rate. It is the rate at which information is transferred to a communication channel. The baud rate needs to be set the same on both the transmitting and receiving device of UART. In serial port context, the set baud rate will serve as the maximum number of bits per second to be transferred.

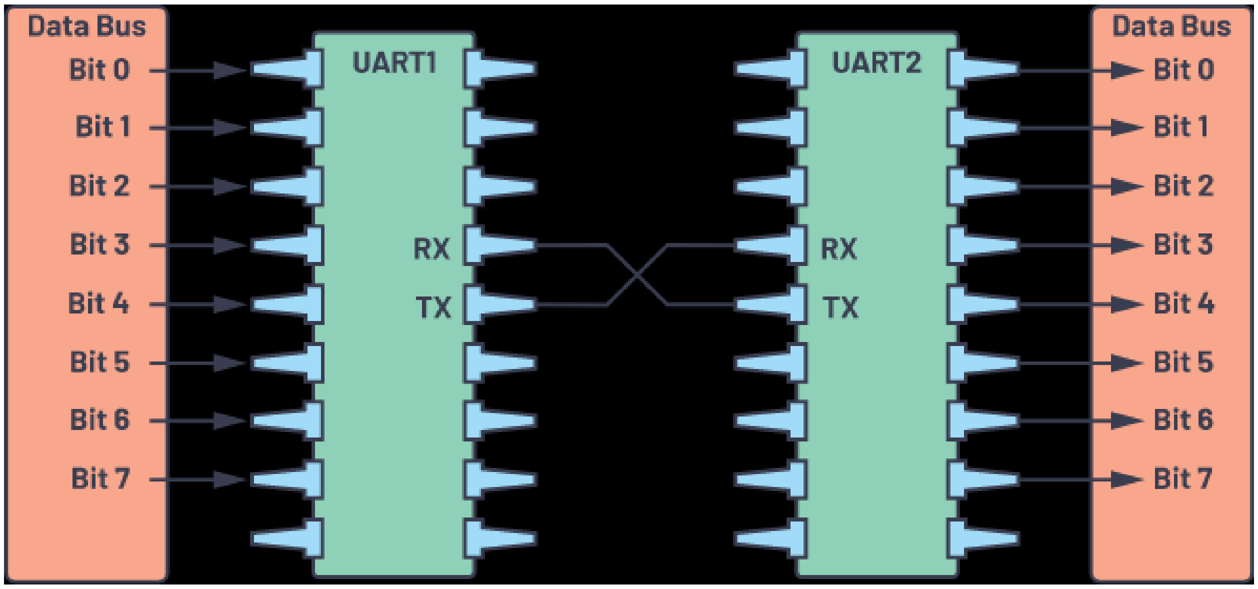


Fig 2.1.2 UART with data bus

There are two basic ways of serial communication: synchronous serial communication and asynchronous serial communication.

Synchronous serial communication means that under the same data transmission rate, the communication frequency of sender and receiver keeps strict synchronization. Because there is no need to use start bit and stop bit, the data transmission rate can be improved, but the cost of transmitter and receiver is higher.

Asynchronous serial communication means that under the same baud rate, the sender and the receiver do not need to be strictly synchronized, and relative time delay is allowed, that is, the frequency deviation between the sender and the receiver is less than 10%, which can ensure the correct communication.

When asynchronous communication does not send data, the data signal line always presents a high-level state, which is called idle state (also known as MARK state). When there is data transmission, the signal line becomes low level and lasts for one bit, which is used to indicate the start of transmission character. This bit is called start bit, also known as space state. After the start bit, each character data to be transmitted appears on the signal line in turn, and is transmitted bit by bit in the order of low bit first and high bit later. Different character coding schemes are adopted, and the number of bits of each character to be sent is different such that 5, 6, 7, 8 or 9 bits, but in this paper only 8 bits data are used. The data bit can be followed by a parity bit or not, which is specified by programming. The last transmission is stop bit, generally 1 bit, 1.5 bit or 2 bits.

Sending data: The CPU writes the data to UART, and UART is sent out serially on a wire according to a certain format.

Receiving data: UART detects the signal on another wire, puts the serial data in the buffer, and the CPU can read the data obtained by UART

2.1.1 Data transmission mode in UART

Simplex mode: a -> b

Simplex mode uses a data transmission line, which only allows data to be transmitted in a fixed direction.

Half duplex mode: a -> b, a <- B

Half duplex mode adopts a data transmission line, which allows data to be transmitted in two directions time-sharing, but not in two directions at the same time.

Full duplex mode: a <-> b

The full duplex mode adopts two data transmission lines, which allows data to be transmitted in two directions at the same time.

UART uses standard TTL/CMOS logic levels (0 ~ 5V, 0 ~ 3.3V, 0 ~ 2.5V or 0 ~ 1.8V) to represent data. High level represents 1 and low level represents 0. In order to enhance the ability of data interference and improve the transmission length, TTL / CMOS logic level is usually converted to RS-232 logic level. 3 ~ 12V represents 0 and - 3 ~ - 12 represents 1.

TX and RX data lines transmit data in "bit" as the smallest unit. A frame is composed of a number of bits with complete meaning and indivisible. It includes start bit, data bit, check bit (not necessary) and stop bit. Before sending, UARTS should agree on the data transmission rate (that is, the time occupied by each bit, the reciprocal of which is called baud rate) and data transmission format.

2.1.2 UART Frame Format

Implementation of UART and working principle of UART and frame format are important in retrieving the data bits after they are transmitted into the channel[13]. The mode of transmission in UART is in the form of a packet. Pieces of UART that connect the transmitter and receiver includes the creation of serial packets and controls those physical hardware lines. The packet consists a start bit, data frame, a parity bit, and stop bits.

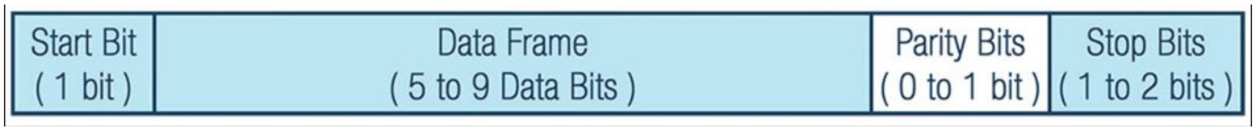


Fig 2.1.3 UART packet.

Starting Position (Start Bit):

In the idle state, the serial port always maintains a high-level state. When there is a data packet to be sent, the serial port first sets the power to lower the transmission time of one bit to inform the other party that it is ready to receive data. There needs to be a start bit in front of each data packet.

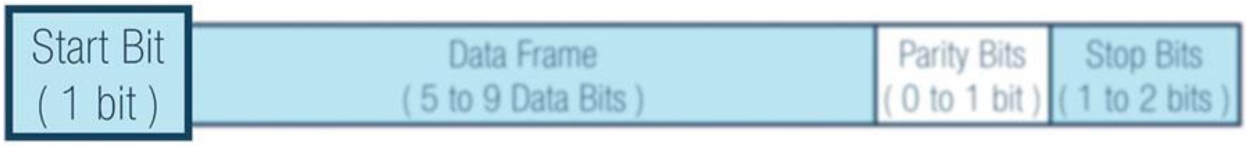


Fig 2.1.4 Start bit

Data Bits:

The number of bits used to determine the significant data bits contained in each byte of the transmitted data. This data can be 5, 6, 7 or 8 bits if parity bit is added and can also be 9 bits if no parity bit is added. For example, if the transmission is ASCII code, which is less than 0x80, that is, the highest bit is useless, then 7-bit transmission mode can be selected to improve the transmission speed.

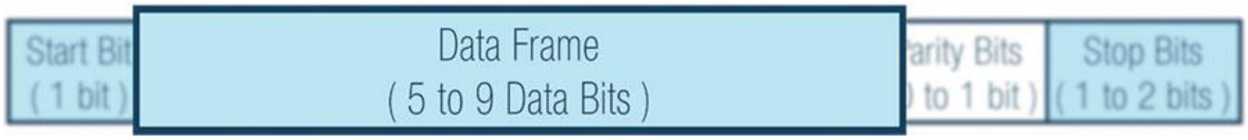


Fig 2.1.5 Data frame

Check Bit (Parity Bit):

In order to ensure the correctness of the transmitted data, some data check codes are often added in the communication process to check the correctness of the data by the receiver. Some simple checks are supported in UART, and no check bits are allowed. When parity check is required, the serial port will add a bit at the end of the data bit to represent that there are odd or even high bits in the transmitted data. The receiver will check this bit and give an error message if there is an error. Thus, parity bit is a method for the receiving UART to tell if any data has changed during transmission. Data bits can be changed by electromagnetic radiation, mismatched baud rates, or long-distance data transfers.

After receiving UART completes reading the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the check or parity bit is a 0 (even parity), the 1 or logic-high bit in the data frame should total to an even number. If the check or parity bit is a 1 (odd parity), the 1 bit or logic highs in the data frame should total to an odd number.

When the check or parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd, or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

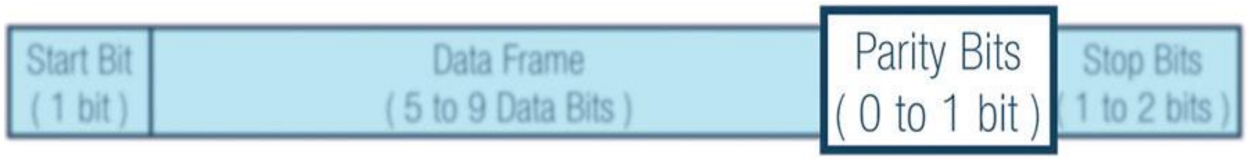


Fig 2.1.6 Check or Parity bit

Stop Bit:

Add a certain period of high level at the end of the packet to indicate the end of the data, which can be 1 / 1.5/2 data bit long clock cycle.

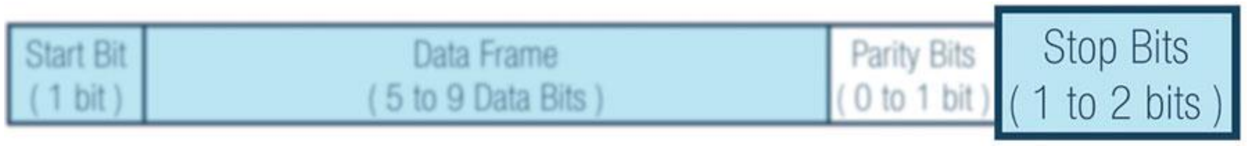


Fig 2.1.7 Stop bits

2.1.3 Steps of UART Data Transmission and Reception

The transmitting UART first receives data in parallel from the data bus of the sending device. This can be seen in the following figure.

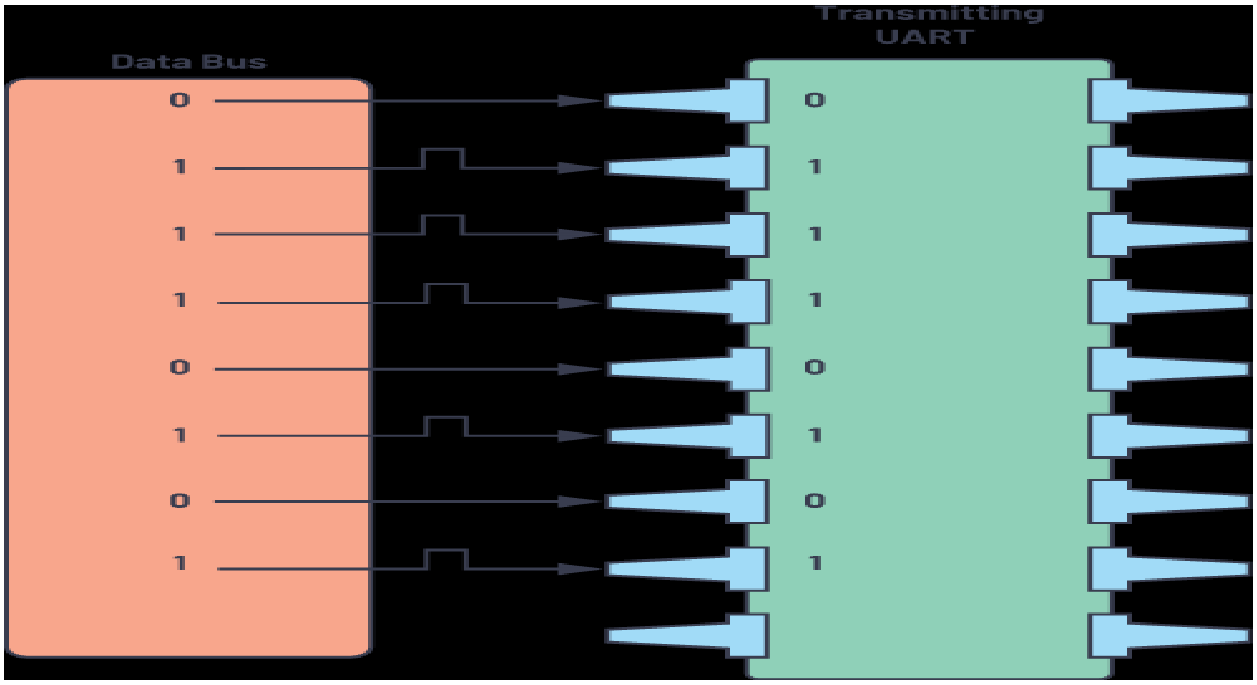


Fig 2.1.8 Data bus to the transmitting UART

After the transmitting UART receives the parallel data from the data bus of the sending device and convert the parallel data in serial data format, the transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame.

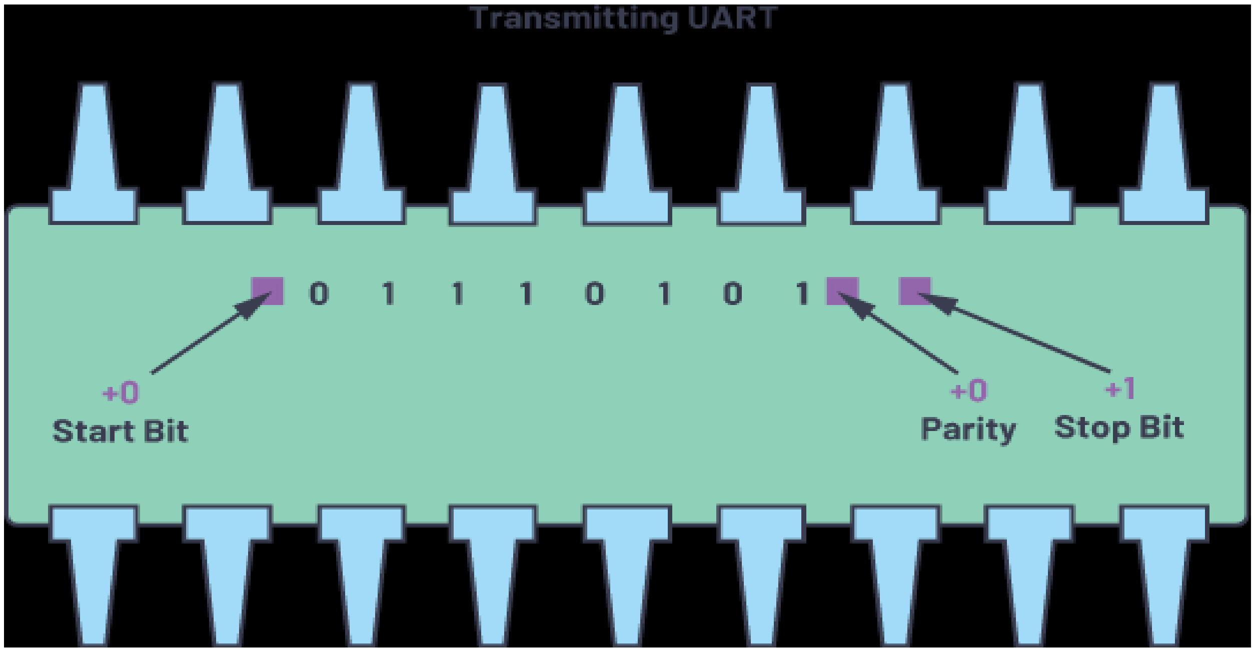


Fig 2.1.9 UART data frame at the Tx side

After having a complete data packet set up, the entire packet is sent serially starting from start bit to stop bit from the transmitting UART to the receiving UART. The receiving UART samples the data line at the preconfigured baud rate.

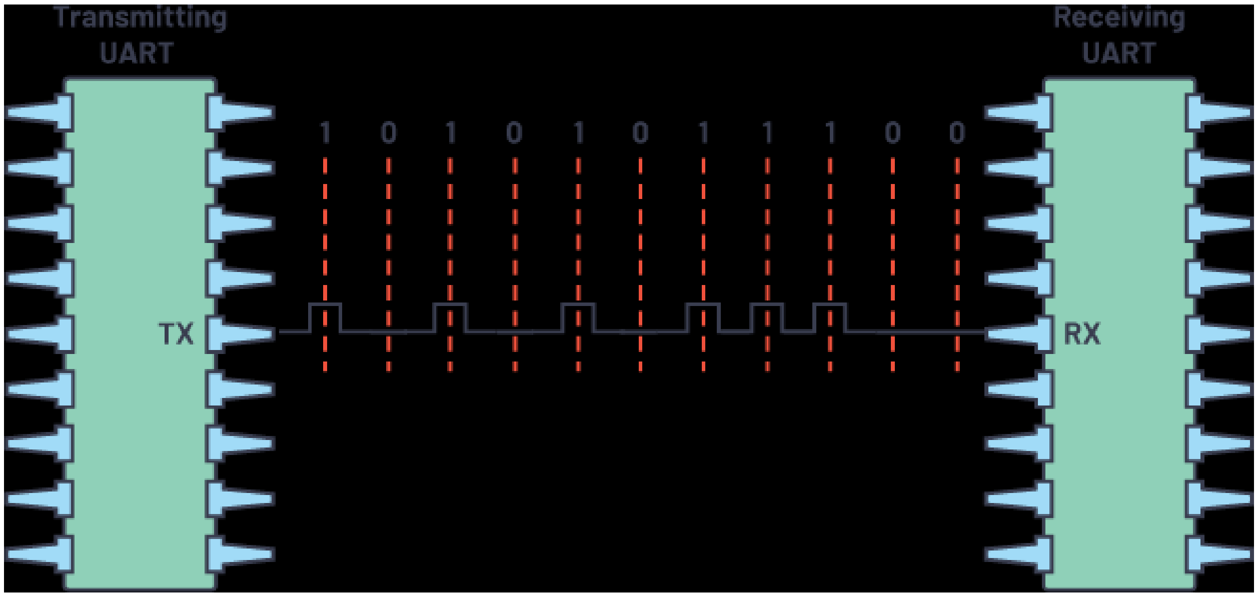


Fig 2.1.10 UART transmission

Receiving UART discards the start bit, parity bit, and stop bit from the data frame and only the data bits will be ready to be converted back to parallel data format.

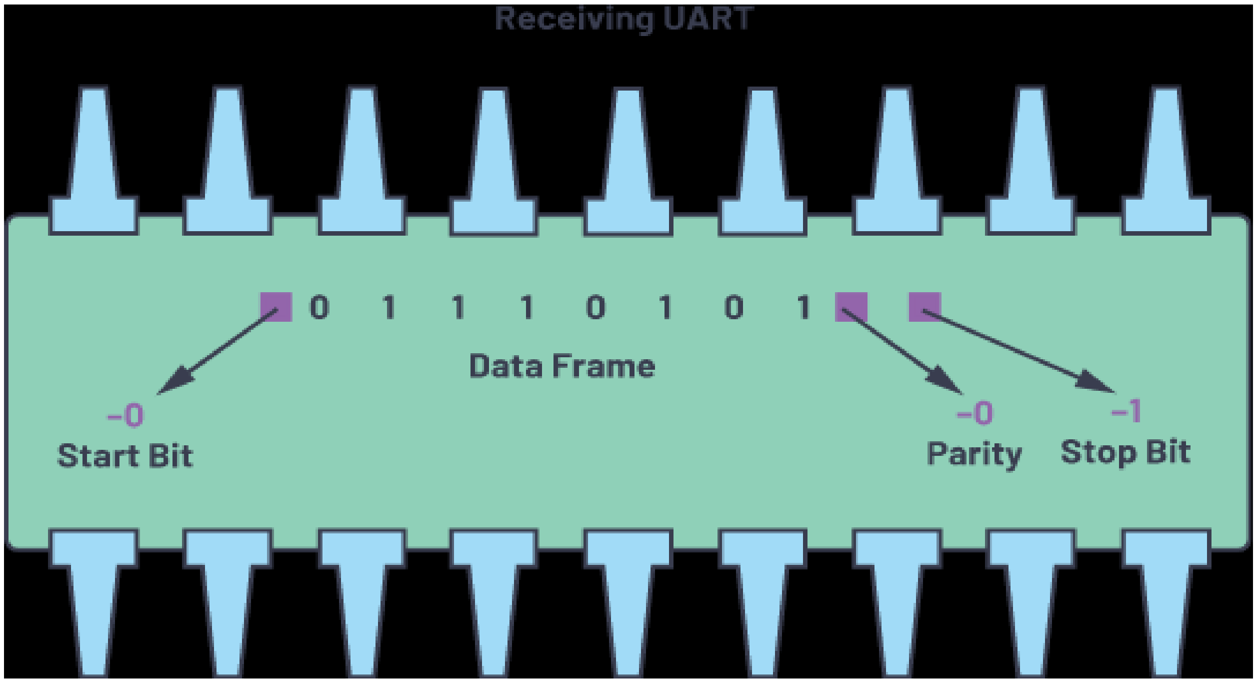


Fig 2.1.11 The UART data frame at the RX side

Then afterward the receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end. This can be shown in the figure below.

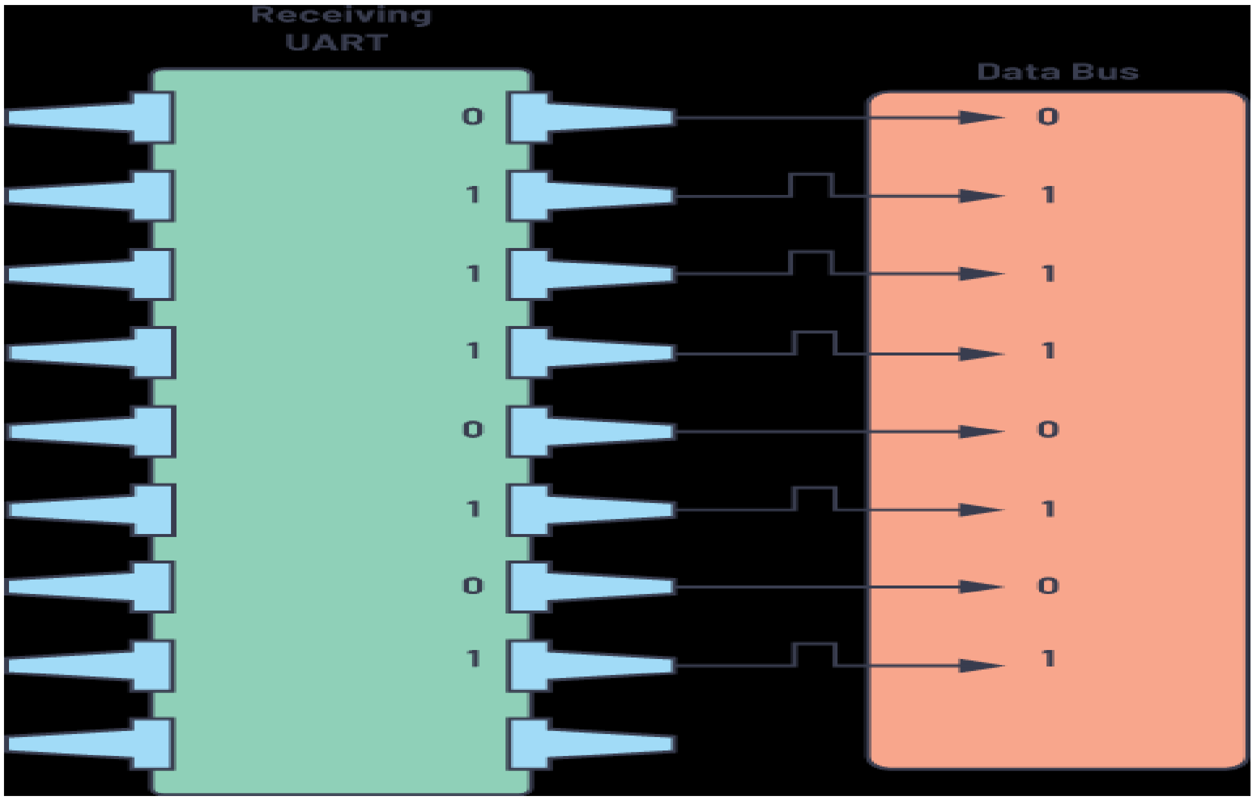
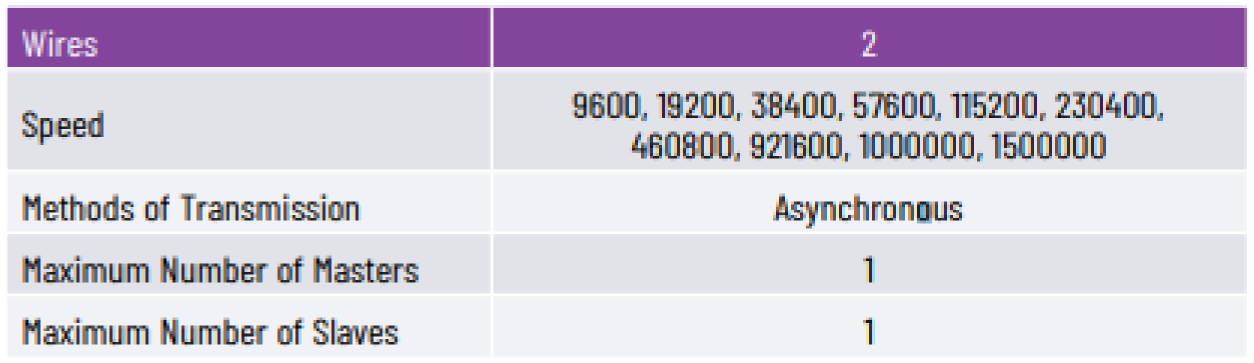


Fig 2.1.12 Receiving UART to data bus

UART Baud Rate:

Baud rate is used to identify the speed of UART communication. It represents the number of bits transmitted per second. For example, 9600 baud means 9600 bits per second. The higher the baud rate is, the faster the communication speed is. But the baud rate is inversely proportional to the distance and stability of communication. The faster the speed is, the higher the possibility of interference is. Therefore, with the attenuation of the signal in the transmission process, the worse the stability is.



Wires	2
Speed	9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600, 1000000, 1500000
Methods of Transmission	Asynchronous
Maximum Number of Masters	1
Maximum Number of Slaves	1

As the UART interface doesn't use clock signal to synchronize the transmitting device and the receiving devices; it transmits data asynchronously. Thus, instead of using clock signal, the transmitter generates a bitstream based on its clock signal while the receiver is using its internal clock signal to sample the incoming data. The point at which the synchronization to be done is managed by having the same baud rate on both transmitting and receiving devices. Failing to do so would affect the timing of sending and receiving data that can cause discrepancies during data handling. The possible allowable difference of baud rate between the transmitting UART and receiving UART is up to 10% before the timing of bits gets too far off.

2.2 FPGA (Field Programable Gate Array)

A field-programmable gate array (FPGA) is a logic device that contains a two-dimensional array of generic logic cells and programmable switches[2]. It is an integrated circuit that consists of internal hardware blocks with user-programmable interconnects to customize operation for a specific application. The interconnects can readily be reprogrammed, allowing an FPGA to accommodate changes to a design or even support a new application during the lifetime of the part.

The FPGA has its roots in earlier devices such as programmable read-only memories (PROMs) and programmable logic devices (PLDs). These devices could be programmed either at the factory or in the field, but they used fuse technology (hence, the expression “burning a PROM”) and could not be changed once programmed. In contrast, FPGA stores its configuration information in a re-programmable medium such as static RAM (SRAM) or flash memory. Furthermore, it is highly integrated, and its device density varies from tens of thousands of system gates to tens of millions of system gates. It can complete extremely complex combinational and sequential logic circuit functions, and is suitable for high-speed and high-density, high-end digital logic circuit design.

FPGA can be used in high-speed, high-performance tasks such as image processing, telecommunications, digital signal processing, high-frequency stock market trading, and many others. Using FPGA technology to design a variety of large-scale or super large-scale combinational and sequential logic circuits can not only make the designer avoid direct contact with the selection and wiring of some complex and tedious circuit components, but also make the modular design method similar to software development used in the development and design process more convenient for many people to cooperate in design and development, which has greatly improves the efficiency of product planning and design. Therefore, FPGA technology has become an international standard line of hardware chip product design and development of the mainstream.

Now a days, the general FPGA chip contains large capacity ROM and RAM unit, programmable logic unit, rich wiring resources and other basic components. These resources can fully meet the needs of the vast majority of design and development. With the development of process technology and market demand, new FPGA with super large scale, high speed and low power consumption is constantly emerging. Some advanced FPGA smart chips even contain CPU and embedded dedicated hard core. The main device suppliers of FPGA are Xilinx, Altera, lattice and Actel.

2.2.1 What is an ASIC?

ASIC which stands for Application Specific Integrated Circuit is similar in theory to FPGA, with the exception that it is fabricated as a custom circuit. This means that unlike FPGA it is not reprogrammable, so we had better get it right the first time. Since ASIC is custom circuit, it is very fast and use less power than an FPGA. This can be critical in power-sensitive applications such as cell phones, mp3 players, and other battery-operated devices.

What makes FPGA and ASIC special is that they are very good at performing a large number of operations in parallel (at the same time).

2.2.2 FPGA Design Flow Chart

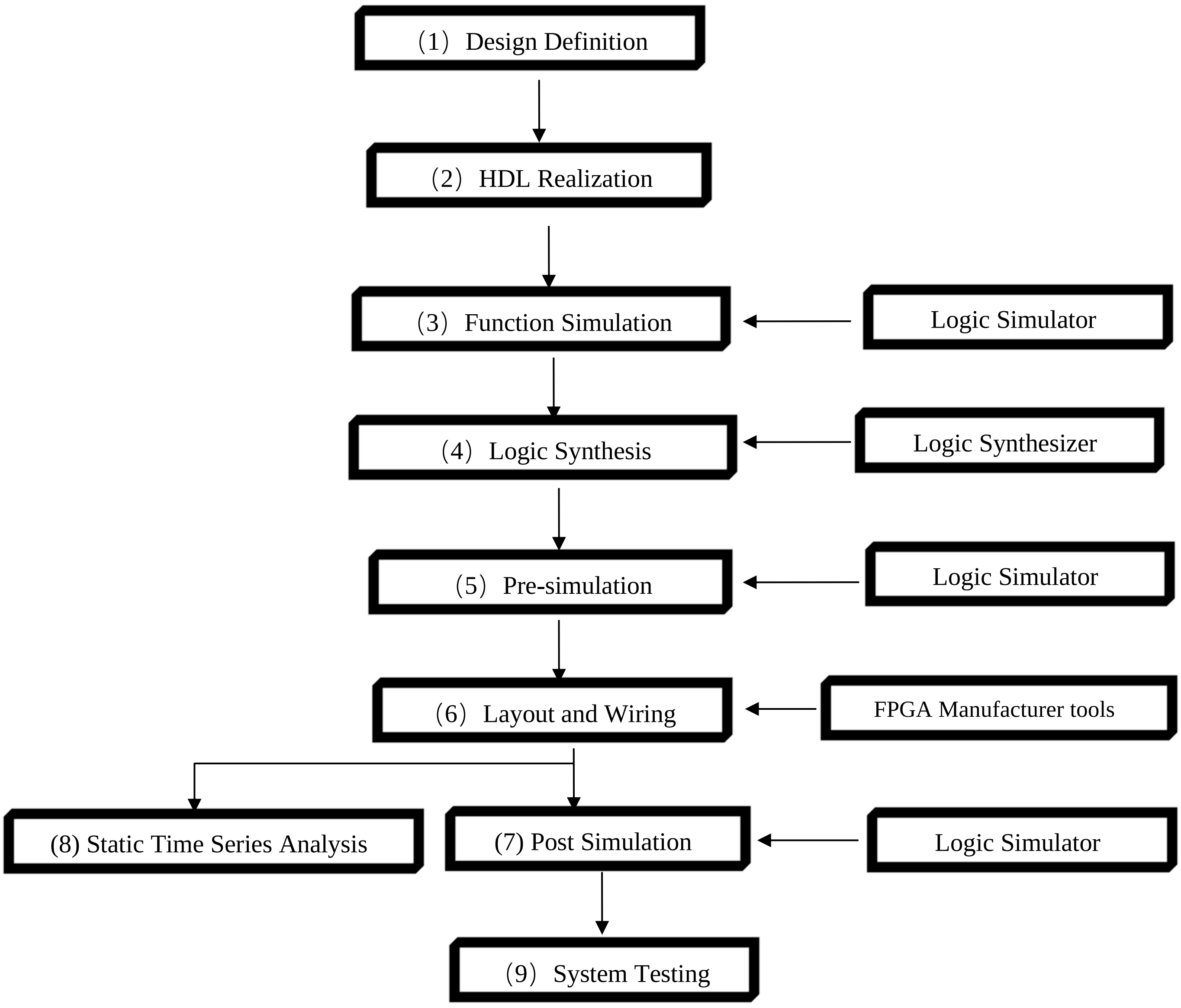


Fig 2.2.1 FPGA design flow chart

Here, Logic simulator mainly refers to the Modelsim Simulator that we are using while doing this project and the Verilog HDL which the hardware description language that we are using for coding. Logic synthesizer mainly refers to Leonardo spectrum, Synplify, FPGA express / FPGA compiler, etc.

FPGA manufacturer tools refer to MAX + PLUSII and Quartus II of Altera, foundation, alliance and ise4.1 of Xilinx, etc. The FPGA type we are using in this project is Altera cyclone II, which is Quartus II of Altera family.

2.2.3 Function Simulation

After designing the design code and create the testbench code to test our design code, functional simulation can be performed on the design. Functional simulation is an iterative process, which may require multiple simulations to achieve the desired end functionality of the design. After the desired functionality is achieved, use the output data to create a self-checking testbench code. This allows for automated testing and reduces the change of a functional regression in the design due to seemingly unrelated changes. It is important to set up the proper infrastructure for this type of simulation, spending time up front may save more time in back-end design debugging.

The following are a general recommended point in performing functional simulation;

- Spend time to create a good testbench code; Creating a good testbench that can be used for both functional and timing simulation can save substantial time and effort. Even if we do not intend to perform timing simulation, it may become necessary during debugging, which makes a dual-purpose testbench highly beneficial.
- Ensure that your libraries are properly compiled and mapped; In our project we are using ModelSim simulator, where libraries are properly compiled and mapped after providing project path. For other type of simulator, it is necessary to perform the above task.
- Automate the compilation and elaboration simulation steps; When we invoke the simulator within Project Navigator, the ISE® tools automatically run these steps. But, if we are running the simulator outside of Project Navigator, it is recommended that we create a script or use another method to automate these steps.
- Customize the simulator interface to present the information needed to debug the design; We may want to include the information console, the structure or hierarchy view, and the waveform viewer as well as other facilities to evaluate the simulation. Customization can improve the simulation experience and can be tied into the automation of the compilation and elaboration steps. If we are using a waveform viewer as a part of simulation debugging, organize the waveform view to display the proper signals in the proper order with the proper radices. This saves time and helps prevent confusion in interpreting simulation results.

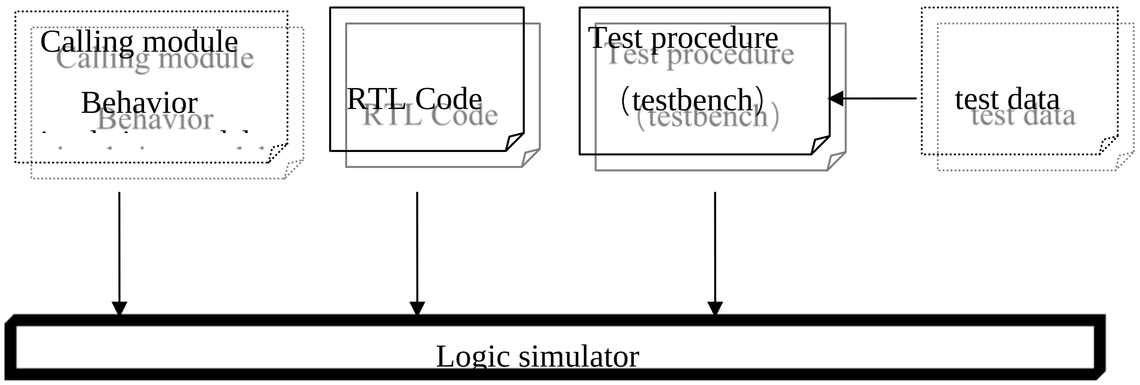


Fig. 2.2.2 Function simulation of FPGA design flow

Here, the behavior simulation model of calling module refers to the behavioral model of macro module / IP provided by manufacturer in RTL code, such as multiplier, memory and other components in LPM library provided by Altera.

2.2.4 Logic Synthesis

Logic synthesis is a process by which an abstract specification of desired circuit behavior, typically at register transfer level (RTL), is turned into a design implementation in terms of logic gates. It is a subject about how to abstract and represent logic circuits, how to manipulate and transform them, and how to analyze and optimize them.

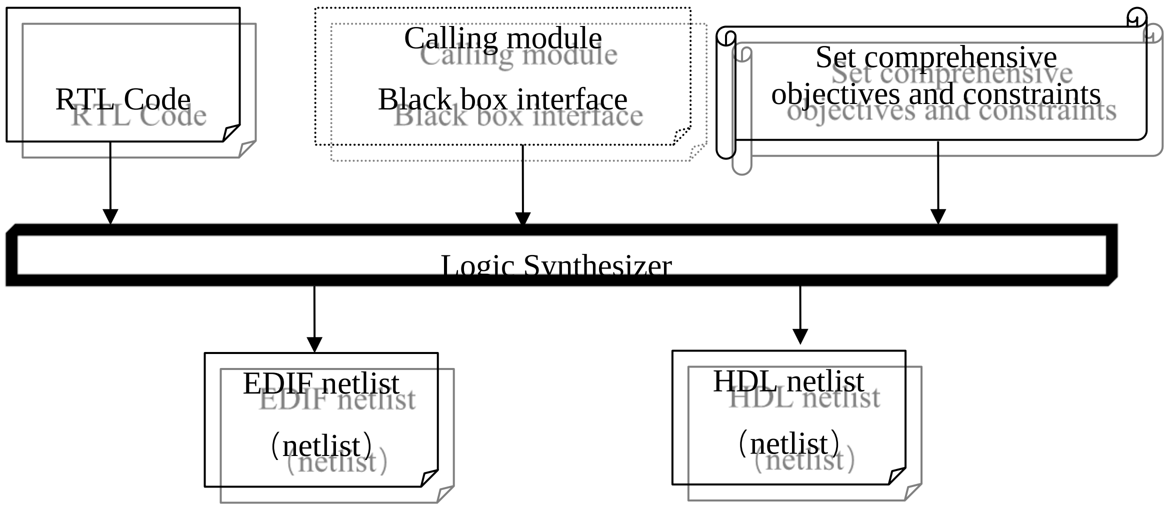


Fig. 2.2.3 *Logic synthesis of FPGA design flow*

The import of "calling module black box interface" is due to RTL code calling some external modules, which cannot be integrated or need not be integrated, but the logic synthesizer needs the definition of its interface to check the logic and retain the interface of these modules.

Logic synthesis transforms HDL code into a netlist describing the hardware (e.g., the logic gates and the wires connecting them). The logic synthesizer might perform optimizations to reduce the amount of hardware required. The netlist may be a text file, or it may be drawn as a schematic to help visualize the circuit.

2.2.5 Pre-simulation

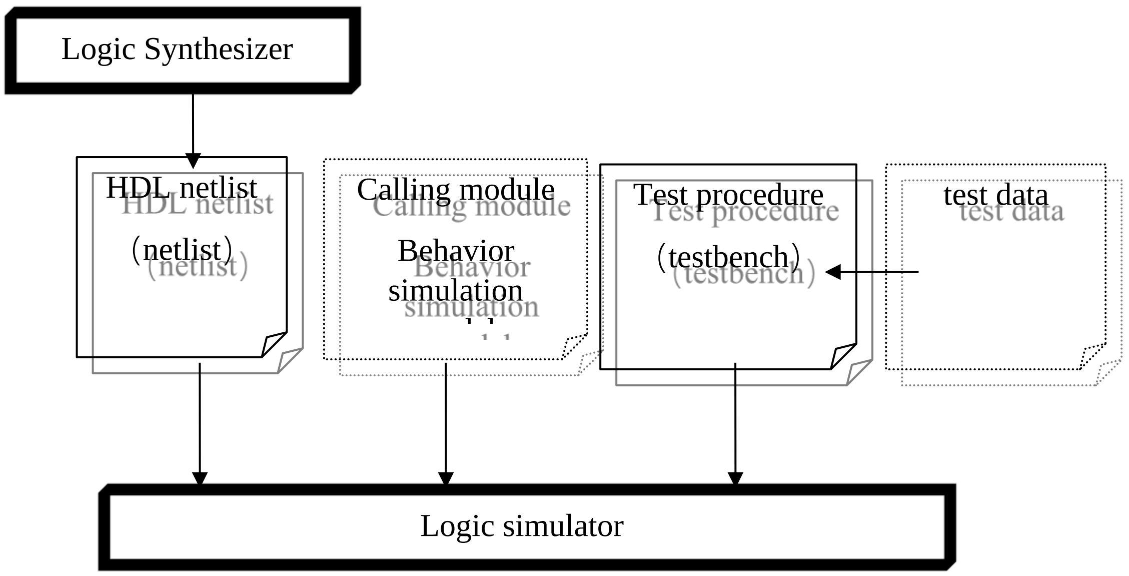


Fig. 2.2.4 Simulation of FPGA design process

The step of FPGA design can be used to debug whether there are problems or not.

2.2.6 Layout and Wiring

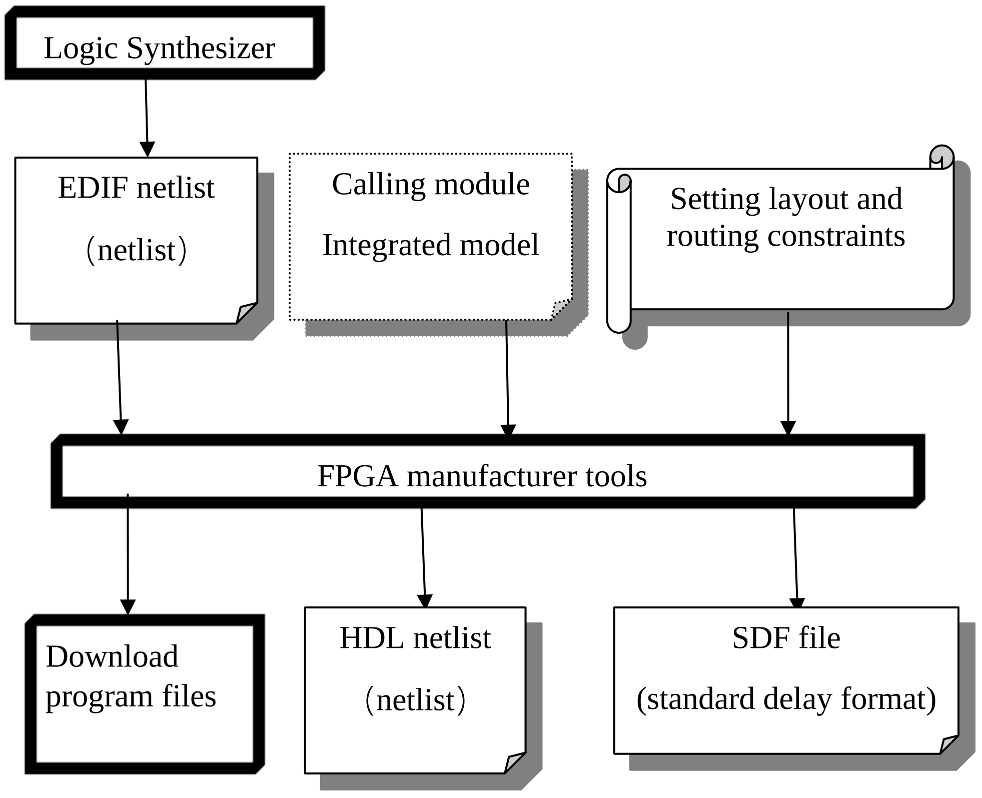


Fig. 2.2.5 Layout and routing of FPGA design flow

2.2.7 Post simulation (timing simulation)

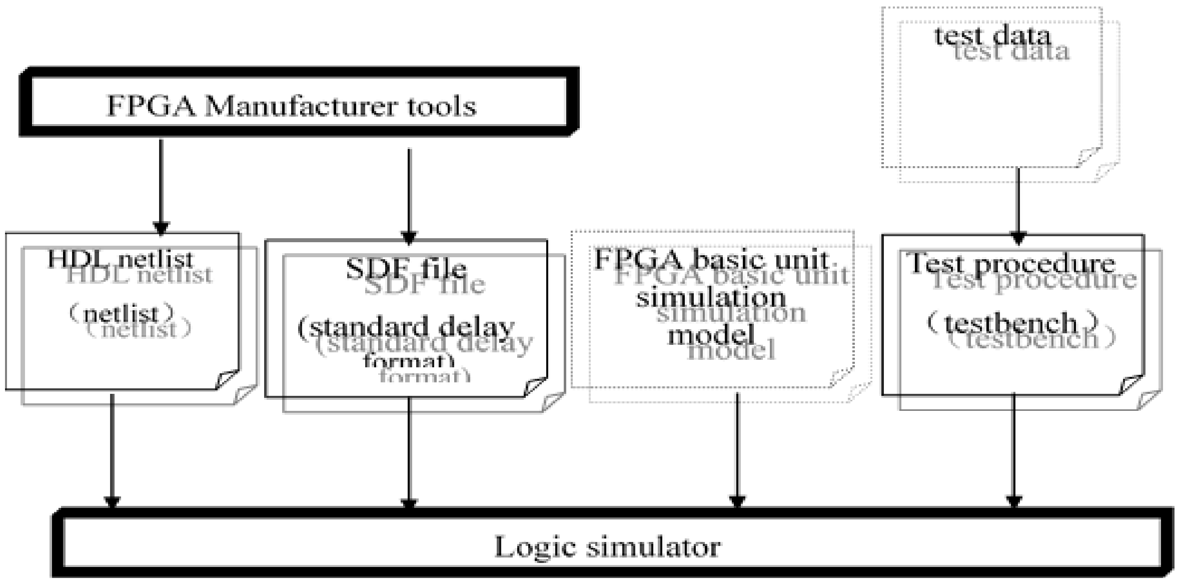


Fig. 2.2.6 Simulation of FPGA design process

With the wide application of microcomputer system and the great development of microcomputer network, UART (universal asynchronous receive transmitter) has been widely used in data communication and control system. 8250, NS 16450 and other chips are common UART devices. These chips have been quite complex, some of them contain many auxiliary modules (such as FIFO), but sometimes only need to use part of the functions of UART in practice, which will cause a certain waste of resources.

With the wide application of FPGA in modern electronic design, we can make full use of its resources and integrate UART function modules on the chip. In this way, there is no need to connect external special UART chip, so as to simplify the circuit, reduce the volume and make the design more flexible.

2.3 Verilog HDL

Verilog HDL is a kind of hardware description language, which is used to model digital system at various abstract design levels from algorithm level, gate level to switch level. It is used to write a digital circuit followed by logic synthesis to get the gate-level netlist to realize the descripted logic. The complexity of the digital system object can be between a simple logic gate and a complete electronic digital system. Digital systems can be described hierarchically and time series modeling can be performed explicitly in the same description. Verilog HDL language has the following description capabilities: the behavior characteristics of the design, the design processes, the data flow characteristics of the system, the structure of the design and the time delay and waveform generation mechanism including response monitoring and design verification are introduced. All of these use the same modeling language. In addition, Verilog HDL language provides programming language interface, through which the design can be accessed from outside during simulation and verification, including the specific control and operation of simulation. Verilog HDL language not only defines syntax, but also defines clear simulation semantics for each syntax structure.

Thus, the model written in this language can be verified by using the Verilog simulator. The Verilog simulator we have used in this project is Modelsim software. The Verilog HDL language inherits many operators and structures from C programming language. Verilog HDL provides extended modeling capabilities, many of which are difficult to understand at first. However, the core subset of Verilog HDL language is very easy to learn and use, which is enough for most modeling applications. Of course, a complete hardware description language is enough to describe everything from the most complex chip to the complete electronic system.

2.3.1 History of Verilog HDL

Verilog HDL is a hardware modeling language developed by gateway design automation company for its simulator products in 1983. At that time, it was just a special language. Due to the wide use of their simulation and emulator products, Verilog HDL, as a convenient and practical language, is gradually accepted by many designers. In an effort to increase the popularity of language, Verilog HDL language was introduced into the public domain in 1900. Open Verilog International (Ovi) is an international organization that promotes the development of Verilog. In 1962, Ovi decided to promote Verilog Ovi standard to become I e standard. Verilog language became IEEE Std 1364-1895 in 1955. The complete standard is described in detail in the hardware description language reference manual.

2.3.2 Verilog HDL Capabilities

The main capabilities of Verilog HDL are listed below:

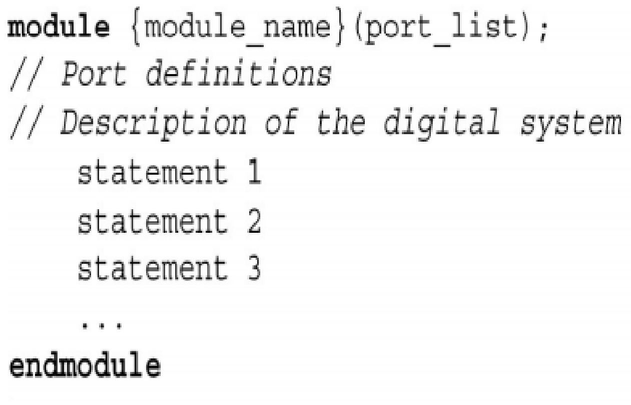
- Basic logic gates, such as AND, OR and NAND, are built into it.
- Flexibility in the creation of user-defined primitives (UDP). User defined primitives can be combinational logic primitives or temporal logic primitives.
- Basic structure models of switch level, such as PMOS and NMOS, are also built into it.
- Provide explicit language structure to specify port-to-port delay and path delay in the design and timing check of the design.
- The design can be modeled in three different or mixed ways. These methods include: behavior description method using procedural structure modeling, data flow method using continuous assignment statement modeling, structured method using gate and module instance statement description modeling.
- There are two types of data in Verilog HDL: wire network data type and register data type. The wire net type represents the physical connection between components, while the register type represents the abstract data storage element.
- Verilog HDL can describe the hierarchical design, and can use the module instance structure to describe any level.
- The scale of the design can be arbitrary; the language does not impose any restrictions on the scale (size) of the design.
- Verilog HDL is now an inclusive hardware description language standardized by IEE.
- Both human and machine can read Verilog language, so it can be used as an interaction

language between EDA tools and designers.

- The description ability of Verilog HDL can be further extended by using programming language interface (PLI) mechanism. PLI is a collection of routines that allow external functions to access information in Verilog modules and allow designers to interact with simulators.
- The design can be described at multiple levels, from switch level, gate level, register transfer level (RTL) to algorithm level, including process and queue level.
- The ability to use built-in switch level primitives to fully model the design at the switch level.
- The same language can be used to generate simulation incentives and specify validation constraints for tests, such as the specification of input values.
- Verilog HDL can monitor the execution of simulation verification, that is, the designed values can be monitored and displayed during the execution of simulation verification. These values can also be used to compare with expected values and print report messages in case of mismatches.
- In the behavior level description, Verilog HDL can not only describe the design on the RT L level, but also describe the design. It can describe the design in architecture level description and algorithm level behavior.
- It can use gate and module instantiation statements to describe the structure at the structure level. Build modeling capability, that is, in a design, each module can be modeled at different design levels.
- Verilog HDL also has built-in logic functions such as & (bitwise AND) and | (bitwise OR).
- High level programming language structure, such as conditional statement, situation statement and loop statement, can be used in Verilog.

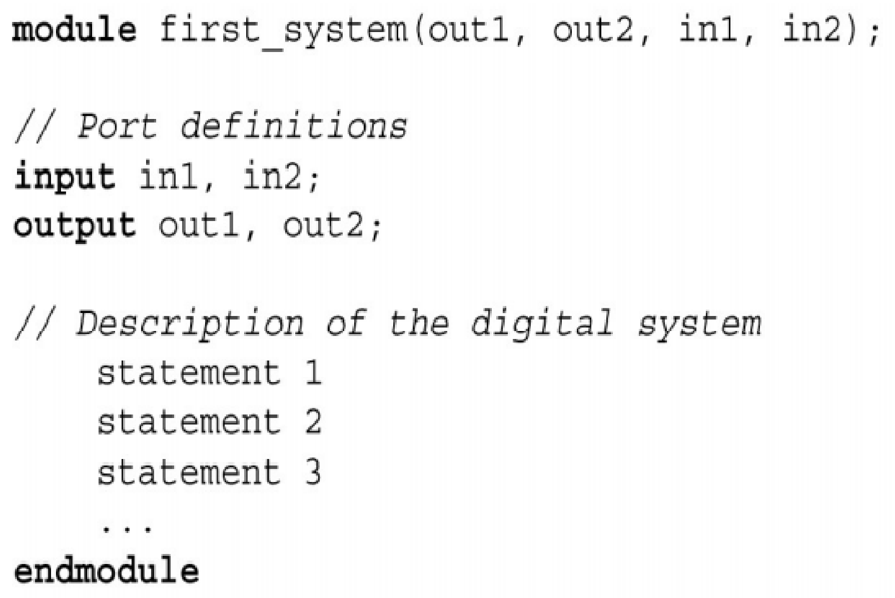
2.3.3 Module Representation

Module is the basic description unit of Verilog, which is used to describe the function or structure of a design and the external port for communication with other modules. The structure of a design can be described by switch level primitives, gate level primitives and user-defined primitives. The data of the design can be described by continuous assignment statements. The temporal behavior can be described by process structure. One module can be used in another. Below is a general format with explanation:



```
module {module_name}(port_list);
// Port definitions
// Description of the digital system
    statement 1
    statement 2
    statement 3
    ...
endmodule
```

First, the module should have a unique name which should not be the same as any of the predefined Verilog keywords. In the above description, we set the name as module_name. Second, the module should have input and output ports assigned to it. We represent these ports as port_list in the above description. The port list does not have a specific order. Therefore, input and output ports can be represented in any order within the list. For convenience, we suggest representing output ports first. At this stage, definition of the module is done. Next comes internal structure of the module. Here, we first define port elements within the module. Each element can be input, output, or inout. As the name implies, the input keyword declares that the related port will get data from outside world. The output keyword declares that the related port will feed data to outside world. The inout keyword declares that the related port can be used for both input and output purposes. Then, we describe the digital system. This is indicated by statement 1, statement 2, and statement 3 above. It is important to remember that order of statements is not important in the description since they will be represented by hardware elements in the FPGA. Afterward, we close the module by keyword endmodule. Note that we can use the symbol // to add a comment to the Verilog description. Below is a simple example of module named as first_sytem which has two input ports in1 and in2 and two output ports out1 and out2:



```
module first_system(out1, out2, in1, in2);

// Port definitions
input in1, in2;
output out1, out2;

// Description of the digital system
    statement 1
    statement 2
    statement 3
    ...
endmodule
```

As can be seen here, the module name for this description is first_system. The port list is composed of out1, out2, in1, in2. Ports in1 and in2 are defined as input in the following line. Similarly, ports out1 and out2 are defined as output in the next line.

2.3.4 Modeling in Verilog HDL

There are three different methods of modeling in describing a digital system in Verilog. Those are structural modeling, dataflow modeling, and behavioral modeling. Each of those modeling are explained in detail in the following subsections.

2.3.4.1 Structural Modeling

The first method in describing a digital system is using structural modeling. In this method, each element to be used in the description statement should have been defined under Verilog as a structure. Since logic gates are extensively used in Verilog descriptions, they have been defined beforehand. Therefore, this description method is also called gate-level modeling. Each gate is represented by the following structure in this method. First, gate type is defined by the corresponding Verilog keyword. Then, a name for the gate is assigned. Note that name assignment is not mandatory. Finally, output and input ports for the gate are defined within parenthesis. Therefore, the structural model of a logic gate will be as gate_keyword name (port_list). The port list should be such that output of the structure is defined first. For example, let’s call the module name as first_system, this module will be used in subsequent sections for emphasizing. The circuit diagram of first_system module can be drawn as follow:

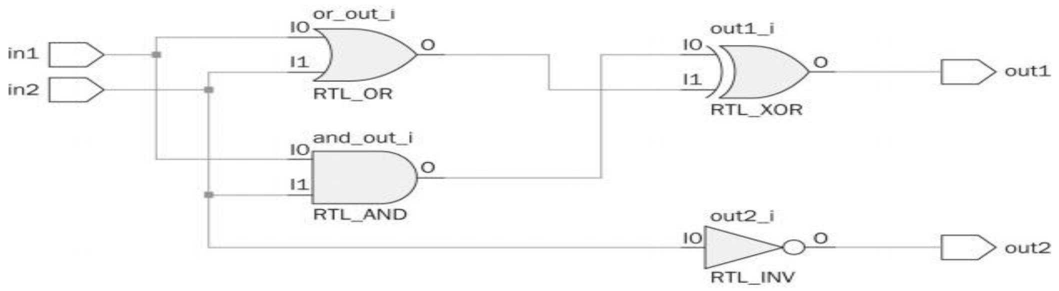


Fig 2.3.1 RTL schematic view of first_system module

As can be seen in this figure, four gates are used in this system as AND, OR, NOT, and XOR. Corresponding Verilog keywords for these are and, or, not, and xor, respectively. Let’s give a name to each logic gate to be used in the description as gate_and, gate_or, gate_not, and gate_xor, respectively. Using these, we can construct the structural model. There is one issue to be solved in describing the digital system. Inputs of the XOR gate are output of the AND and OR gates. We should define variables using the Verilog keyword wire to make this connection. In fact, the user can remember this easily as if we are adding a wire between logic gates. Based on these, we can form the structural model of the digital system as follow:

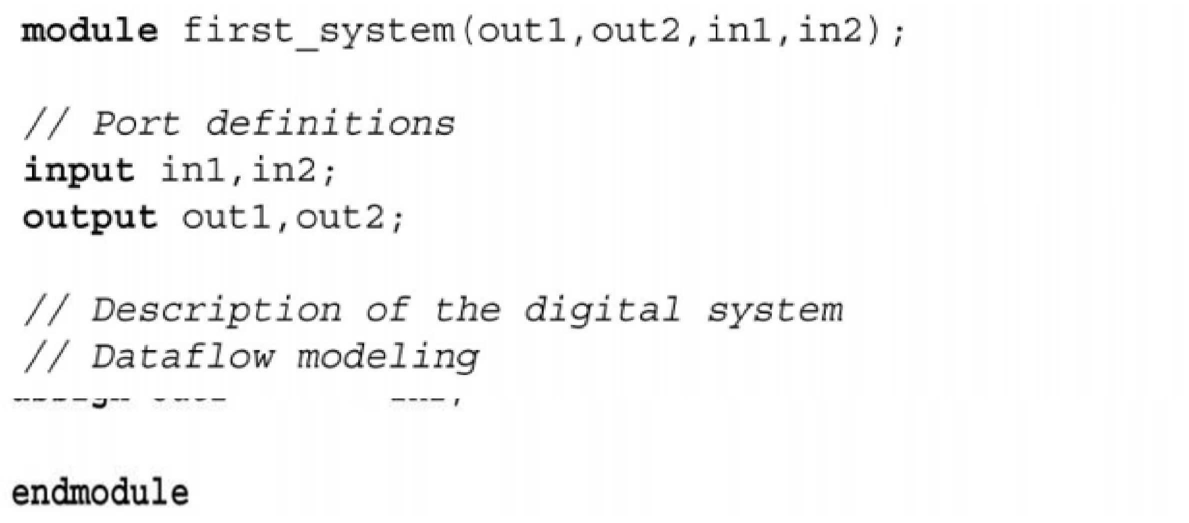
```
module first_system(out1,out2,in1,in2);

// Port definitions
input in1,in2;
output out1,out2;

// Description of the digital system
// Structural modeling

wire and_out,or_out;

and gate_and(and_out,in1,in2);
or  gate_or(or_out,in1,in2);
xor gate_xor(out1,and_out,or_out);
not gate_not(out2,in2);

endmodule
```

2.3.4.2 Dataflow Modeling

The second method in describing a digital system in Verilog is using dataflow modeling. In this method, the relation between input and output ports is formed as a function. Therefore, this description method is also called functional modeling.

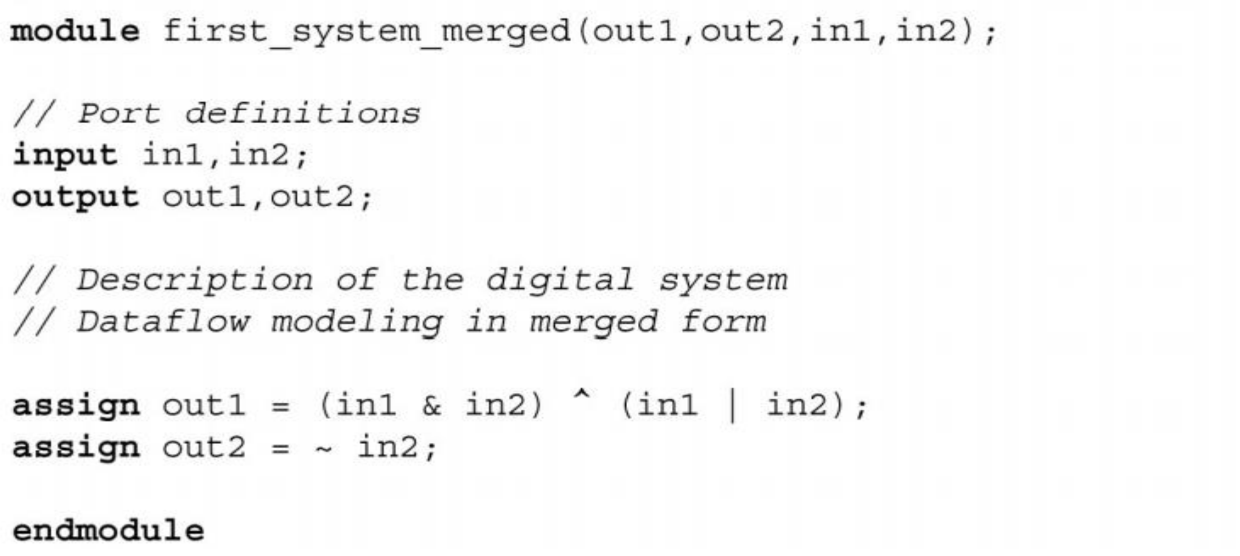
The main keyword in dataflow modeling is assign. The syntax here is assign output = function of inputs. Output in this representation must always be a scalar or vector. Here, the function may be formed by logic gate representations. As in structural modeling, only logic gate AND, logic gate OR, logic gate NOT and logic gate XOR are considered here. Corresponding operators to be used in dataflow modeling are &, |, ~, ˆ respectively. As in structural modeling, wire keyword can be used in this description to connect input and output of logic gates.

The digital system we described above in structural modeling can be now described by dataflow modeling as follow:



```
module first_system(out1,out2,in1,in2);

// Port definitions
input in1,in2;
output out1,out2;

// Description of the digital system
// Dataflow modeling
[illegible]

endmodule
```

Dataflow modeling allows merging functions, which leads to a more compact representation.



```
module first_system_merged(out1,out2,in1,in2);

// Port definitions
input in1,in2;
output out1,out2;

// Description of the digital system
// Dataflow modeling in merged form

assign out1 = (in1 & in2) ^ (in1 | in2);
assign out2 = ~ in2;

endmodule
```

As can be seen here, output out1 is defined in one merged line. Therefore, wire definitions are discarded from the description.

2.3.4.3 Behavioral Modeling

The third method in describing a digital system in Verilog is using behavioral modeling. In this method, digital system at hand is represented by its behavior. In other words, Verilog keywords corresponding to conditional and recursive statements can be used within the model.

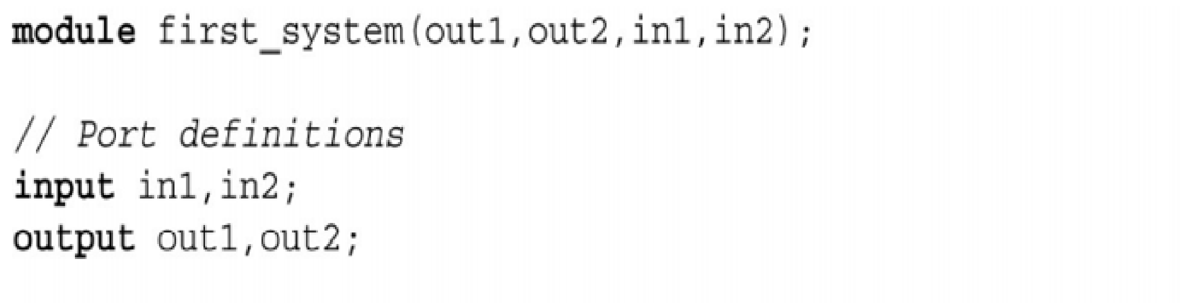
In behavioral modeling, statement (or statements) to be executed should be triggered by a signal (or signals) to operate. The keyword always is used to indicate this triggering operation. Once the signal changes its state, the statement is executed. If there is more than one statement to be executed, then they should be encapsulated by begin and end keywords. Hence, syntax for this representation becomes as follows:

```
always @ (sensitivity_list)
     begin
     // behavioral description
          statement 1
          statement 2
          statement 3
          ...
     end
```

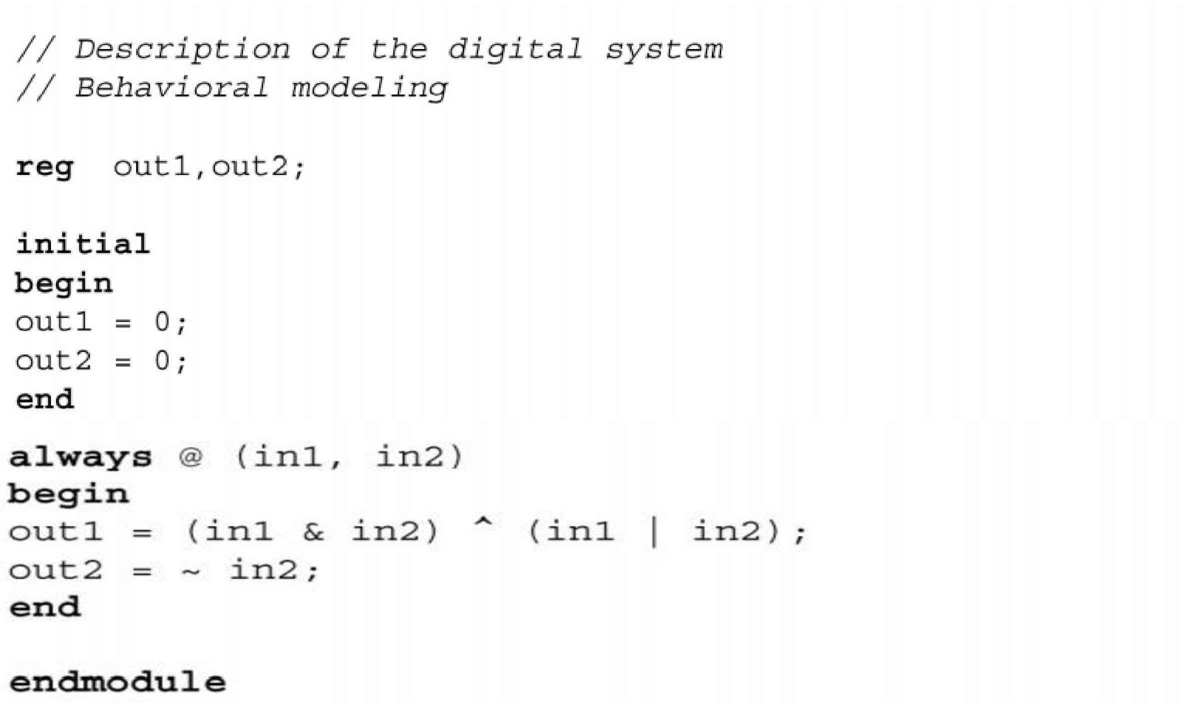
Here, sensitivity_list stands for triggering signal(s). The sensitivity list can be formed of signals separated by comma or combined by or keyword. If the behavioral description is to be executed for any input changes, then * sign can be used instead of the sensitivity list. Here, whenever one of the signals in the sensitivity list changes its state, the behavioral description is executed. Again, order of statements is not important in behavioral modeling. One other important Verilog keyword for behavioral modeling is initial. Via this keyword, an initial block can be formed which is executed at time zero. Syntax of the initial block is as follows:

```
initial
    begin
      statements
    end
```

Let's describe the digital system of the above module called first_system using behavioral modeling. Behavior of the system will change when the first or second input changes. Therefore, at the beginning of the always block, the sensitivity list will consist of inputs in1 and in2. We can represent the relation between input and output of the system as in dataflow modeling. However, the assign keyword will not be used in behavioral modeling. Since there is more than one statement to be executed, they are encapsulated within begin and end keywords. As a result, behavioral model of the first system will be as follow:



```
module first_system(out1,out2,in1,in2);

// Port definitions
input in1,in2;
output out1,out2;
```



```
// Description of the digital system
// Behavioral modeling

reg  out1,out2;

initial
begin
out1 = 0;
out2 = 0;
end
always @ (in1, in2)
begin
out1 = (in1 & in2) ^ (in1 | in2);
out2 = ~ in2;
end

endmodule
```

We should take a closer look at the description in above behavioral modeling. The always keyword executes the beneath description block (encapsulated by begin and end keywords) whenever in1 or in2 changes. If there is no change in these variables, output will not be provided by the system. Therefore, we have to save previous output values. This can be done by the Verilog keyword reg. We used this keyword to keep the previous value of out1 and out2. We also initialized these variables to logic level zero using the initial keyword.

There are two assignment types in behavioral modeling. These are called blocking and nonblocking. Statements having blocking assignment are executed one by one in sequential order. Therefore, as the name implies, each assignment blocks the execution of the next in hierarchy. Operator for the blocking assignment is =. Statements having nonblocking assignment are executed concurrently. Therefore, they don’t block each other. Operator for the nonblocking assignment is <=. It is strongly suggested in literature that blocking assignments should be used in combinational circuits and nonblocking assignments should be used in sequential circuits.

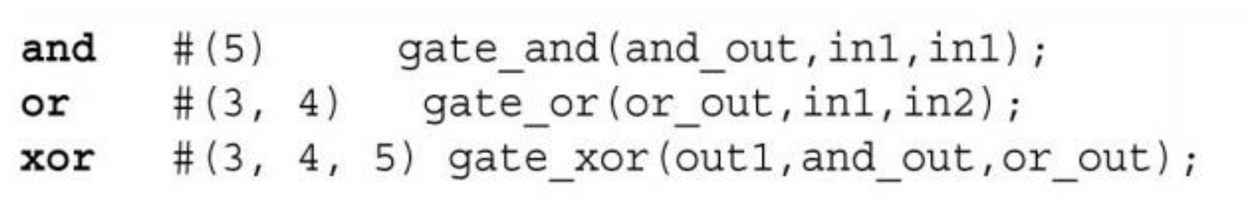
2.3.5 Timing and Delays in Verilog

The simulator used in this project is ModelSim Simulator. ModelSim is a multi-language environment by Mentor Graphics, for simulation of hardware description languages such as VHDL, Verilog and SystemC, and includes a built-in C debugger. ModelSim can be used independently, or in conjunction with Intel Quartus Prime, Xilinx ISE or Xilinx Vivado.

ModelSim simulates behavioral, RTL, and gate-level code - delivering increased design quality and debug productivity with platform-independent compile. Single Kernel Simulator technology enables transparent mixing of VHDL and Verilog in one design.

ModelSim allows adding simulation timings in Verilog descriptions. Moreover, if a blank Verilog file is to be opened, Modelsim adds the first line automatically as ’timescale 1ns / 1ps. These are the default timing values such that the first one(1ns) indicates the reference time unit. Whenever a time value is added to the Verilog description, it will be in the order of one nanosecond. The second timing value (1ps) indicates the smallest precision that can be achieved. Hence, the default smallest precision in simulation is one picosecond. Again, these values will be of use during simulation. They will have no effect in the actual FPGA realization step.

In Simulation, it is necessary to take the physical characteristics of logic gates into account. In other words, we not assume all delay times to be zero within logic gates. If the user wants to obtain accurate results (especially in timing diagrams) of the implemented digital system, then delay values should be added to the Verilog description. These can be done in connection with the reference time unit. There are three delay types that can be added to a digital device in Verilog. These are rise delay, fall delay, and turn-off delay. The rise delay indicates the transition time needed from any logic value to logic level one. The fall delay indicates the transition time needed from any logic value to logic level zero. The turn-off delay indicates the transition time needed from any logic value to high impedance. Here, are examples of delays in structural modeling;



```
and    #(5)       gate_and(and_out,in1,in1);
or     #(3, 4)    gate_or(or_out,in1,in2);
xor    #(3, 4, 5) gate_xor(out1,and_out,or_out);
```

We can also apply delay values in dataflow modeling. Such an example is; assign #10 and_out = in1 & in2. Here, #10 indicates that the assignment will be performed by a 10-time-unit delay. This will correspond to 10-ns delay with respect to the default reference time.

2.3.6 Hierarchical Module Representation in Verilog HDL

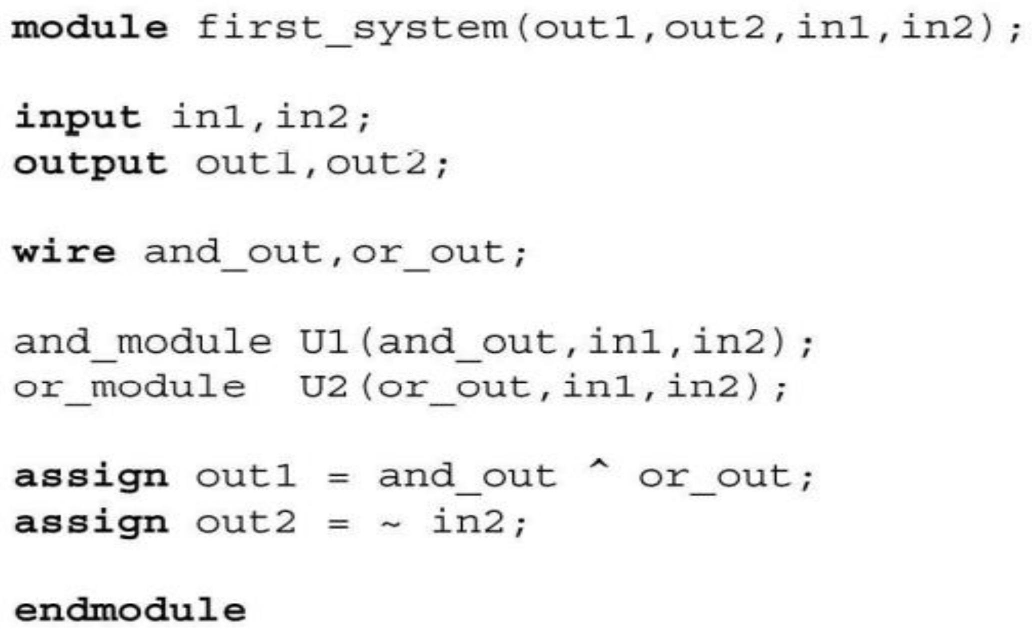
In larger projects, the number of modules may be more than one. In this section, we will show how a project with more than one module can be handled.

Let’s reconsider dataflow model of the first_system that we created earlier in this article. The same description can be represented as a combination of three modules such that AND and OR gates are described in different modules. Let’s call these as and_module and or_module, respectively. These should be formed as valid modules with their input/output ports and descriptions. We should instantiate the and_module and or_module in the top module first_system. This can be done as if structural modeling is used. In other words, the and_module should be represented within the first_system module as “and_module instantiation_name (port_list)”.

There are two options in forming port list correspondence between module to be instantiated and the top module using it. The first one is using locations. Here, the port list order in the top module and instantiation should be the same. The second method in forming the port list correspondence is using the declaration. sub_module_name (top_module_name). Here, port in the module to be instantiated is declared as sub_module_name. The corresponding port in the top module is declared as (top_ module_name). This operation should be done for all input/output ports. We will use both declarations throughout this article, although the second one should be picked whenever possible.

The Verilog description of the First System in Hierarchical Module Representation can be done as follow:

```
module and_module(and_out,in1,in2);

input in1,in2;
output and_out;

assign and_out = in1 & in2;
endmodule

module or_module(or_out,in1,in2);

input in1,in2;
output or_out;

assign or_out = in1 | in2;
endmodule
```



```
module first_system(out1,out2,in1,in2);

input in1,in2;
output out1,out2;

wire and_out,or_out;

and_module U1(and_out,in1,in2);
or_module  U2(or_out,in1,in2);

assign out1 = and_out ^ or_out;
assign out2 = ~ in2;

endmodule
```

2.3.7 Testbench Formation in Verilog

Characteristics of a digital system can be analyzed in Vivado by using a testbench. A testbench allows us to verify the functionality of a design through simulations. It is a container where the design is placed and driven with different input stimulus.

- A Verilog testbench file is composed of five parts as follows:
- Testbench module declaration
- Input/output port declaration Instantiation of the unit under test (UUT)
- Providing input to the UUT
- Displaying test results

It has the following main functions:

- Generate different types of input stimulus
- Drive the design inputs with the generated stimulus
- Allow the design to process input and provide an output
- Check the output with expected behavior to find functional defects
- If a functional bug is found, then change the design to fix the bug
- Perform the above steps until there are no more functional defects

The testbench is itself a Verilog module. Therefore, it needs valid module and input/output port declarations This is the first step in testbench formation. These declarations are done as follow:

```
'timescale 1ns / 1ps

module first_system_tb;

// Inputs
reg in1t, in2t;

// Outputs
wire out1t, out2t;
```

Here, first the simulation timing value is declared by the timescale keyword. Then, the testbench module is declared as module first_system_tb. We specifically assigned such a name to the testbench module to associate it with the top module to be tested. The reader is free to choose any valid name here. Next, input and output ports of the testbench module are declared as reg in1t, in2t and wire out1t, out2t. Again, the reader can pick any valid name for each input or output port in the testbench module.

The second step in testbench formation is associating the module to be tested (unit under test) with the testbench module. This can be done by instantiation as follow:

```
// Instantiate the Unit Under Test (UUT)
first_system UUT (.out1(out1t),.out2(out2t),.in1(in1t),.in2(in2t));
```

Here, as in hierarchical module declaration, the module to be tested (for our case first_ system) is instantiated in the testbench module with the name UUT. Then, each port in the testbench module and the module to be tested are associated (or connected) such as .out1(out1t). Here, the port in the module to be tested is declared as .out1. The corresponding port in the testbench module is declared as (out1t). This operation is done for all input/output ports.

The third step in testbench formation is providing input to the UUT. This can be seen as follow:

```
//Providing input to the UUT
initial begin
// Initialize Inputs
in1t = 0;
in2t = 0;

// Wait 100 ns for global reset to finish
#100;

// Add stimulus here
repeat (4)
#100 {in1t,in2t} = {in1t,in2t} + 1'b1;
end
```

Here, testbench input ports (in1t and in2t) are initialized first. Then, a delay of 100 ns is added by the command #100. This delay is added such that the module to be tested is reset properly. Otherwise, some undesired effects may occur during simulation. Next, input values are fed to the UUT. This is done in two lines as follows. The first line contains the command repeat (4). This indicates that the following line will be repeated four times. The second line contains the command #100 [in1t, in2t] = [in1t, in2t] + 1'b1. This indicates that inputs will be incremented one by one sweeping the pattern 00, 01, 10, and 11. Transition between each input combination is done after a 100-ns delay.

2.4 Chapter Summary

This project aimed to design UART instruction controller based on FPGA. Hence, the main focused on FPGA design flow and Verilog HDL coding. This chapter has briefly explained the background knowledge of FPGA and their design flow. The chapter also has briefly introduced Verilog HDL coding. Module representation, Modeling, Timing, hierarchical representation of modules and testbench formation of modules are the core idea that have been briefly discussed in the Verilog HDL study section of this Chapter. At the beginning of this chapter, UART (Universal Asynchronous Receiver / transmitter) has been also fleetingly elucidated. The generalized working principle of UART has been stated.

Having the basic understanding of about UART, FPGA and coding in Verilog HDL, the proposed project design is designed in the following chapter (chapter three) by the integrated knowledge of UART, FPGA and Verilog HDL.

Chapter Three: Design and Simulation of UART Instruction Controller

3.1 The Function and the System Architecture of UART instruction Controller

The UART instruction controller designed in this paper is capable of receiving data from a personal computer via the receiving UART and perform the designed arithmetic or logic operations on the FPGA development board and then return the result of the operations to the computer via the transmitting UART. Theis design is focused on the Verilog HDL coding and the FPGA design flow, so a complex instruction set is not necessarily required. The Verilog modules is simulated by Modelsim simulating software and tested on an FPGA. The FPGA development board used in this design is Altera Cyclone I.

Design Specifications:

Port type： UART

System clock frequency： 24MHz

FPGA: Altera cyclone I

Baud rates： 4800,9600

ISA: a+b, ~a, a^b, a*b

Thus, the complete design will perform three specific actions.

Those are: -

- Data Receiving Operations
- Instruction Execution / Arithmetic or Logic Operations
- Data Transmitting Operations

The data reception operations are done by the UART Receiver module. This module is named as UART_RX. The instruction execution operations are done by the UART Instruction Controller module. This module is named as cmd_pro. And the data transmission operations are done by the UART Transmitter module. This module is named as UART_TX.

The generalized system architecture can be seen in the figure below:

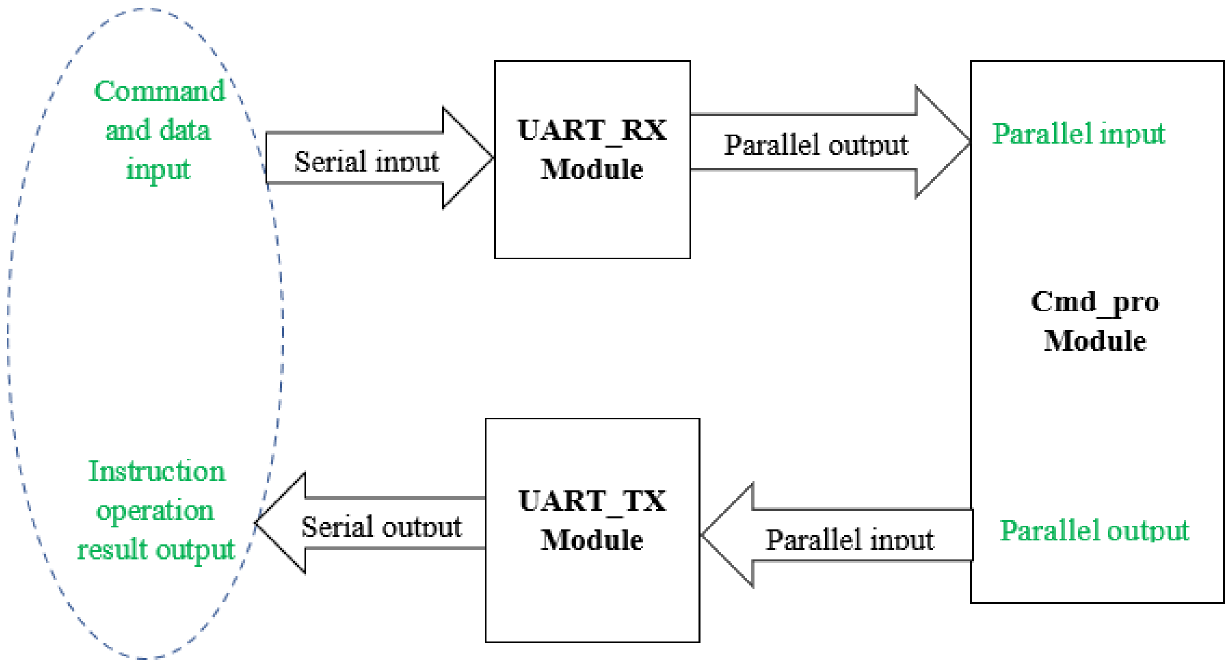


Fig 3.1.1 Generalized system architecture of UART instruction controller

The complete Verilog HDL code of this design is comprised from four Verilog HDL modules. Those are UART_RX Module, UART_TX Module, cmd_pro Module and the top-level module called UART_TOP Module. The UART_TOP Module is a module that embraces all the other three modules. It can be seen the system architecture as follow:

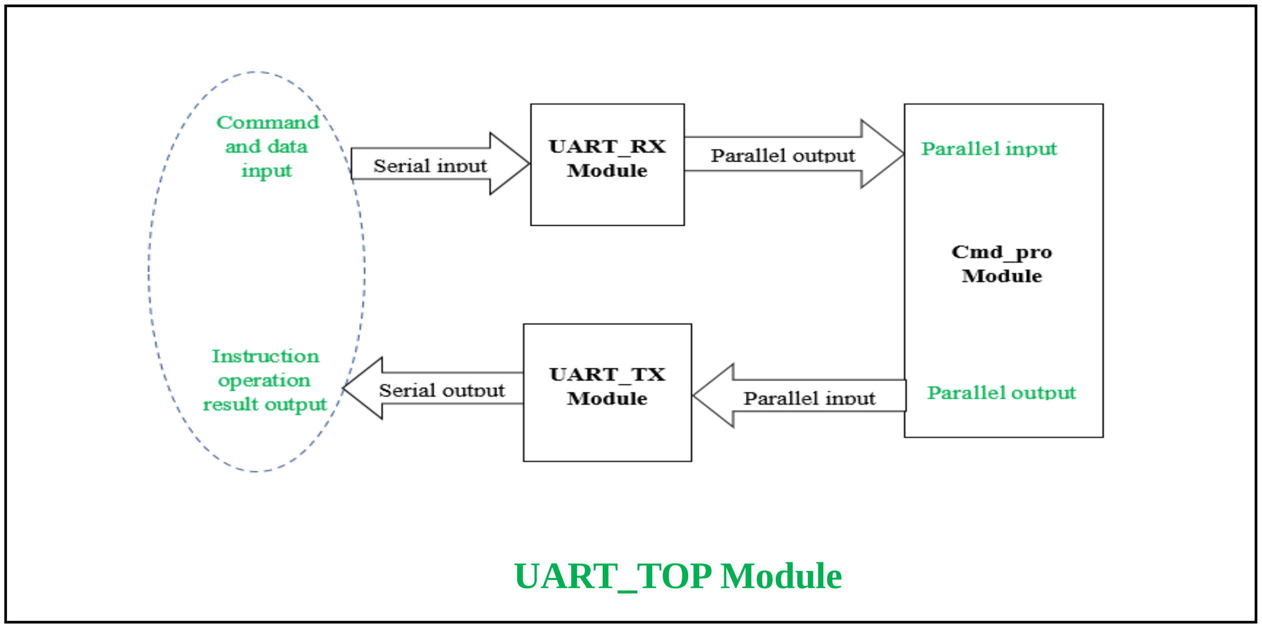


Fig 3.1.2 UART Top level module design

Generally, the UART instruction controller design is capable of performing the following tasks: -

(1) Realize two-way communication with PC via RX pin and TX pin;

(2) The data sent to PC can be defined on the development board through the input module;

(3) The data frame length is adjustable (8 bits);

(4) The baud rate of communication is adjustable, the one used in this design is 9600;

(5) The display of input data and received data is realized on the digital tube.

3.2 Design and Simulation of UART Receiver module (UART_RX)

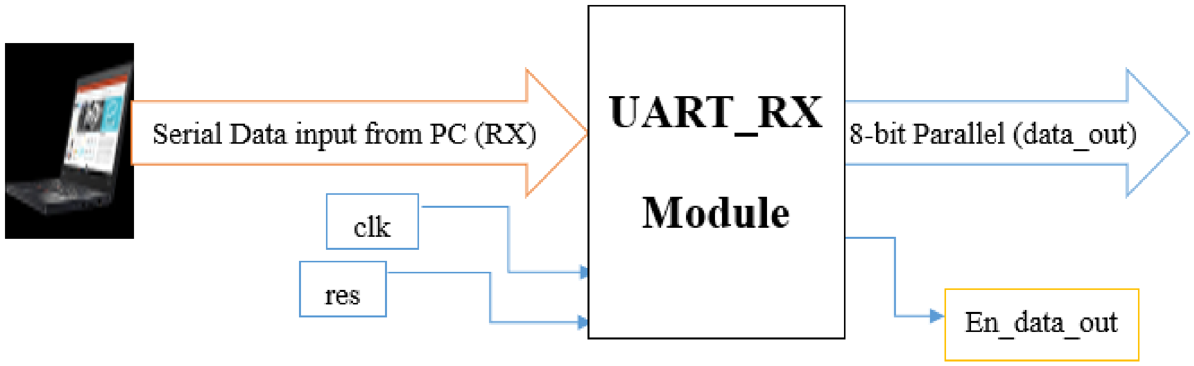


Fig 3.2.1 UART_RX Module diagram

The UART_RX module is designed to receive data inputs and commands from the personal computer. The module has clock signal clk and reset signal called res. It gets 8-bit serial data inputs from the pc via its RX pin. After it receives the 8-bit input data and shift them at a specific rate generated by the baud rate in order convert the serial type to parallel and it will hand the parallel data to the cmd_pro module when the enable data out signal is triggered.

3.2.1 UART Data Reception Principle

The serial data frame and the receiving clock are asynchronous. Jumping from logic 1 to logic 0 can be regarded as the beginning of a data frame, so the receiver must first determine the start bit. After determining the start bit, the receiver starts receiving 8-bit data. The data to be received from the computer is in serial format. Thus, the receiver receives those serial data bit streams one by one in a way that the LSB (least significant bit) comes out to the UART_RX line first and all the way to the MSB (most significant bit) sequentially. After the receiver confirms that 8-bit data been received, the stop bit will be triggered to logic 1 and will be ready to receive the next data frame.

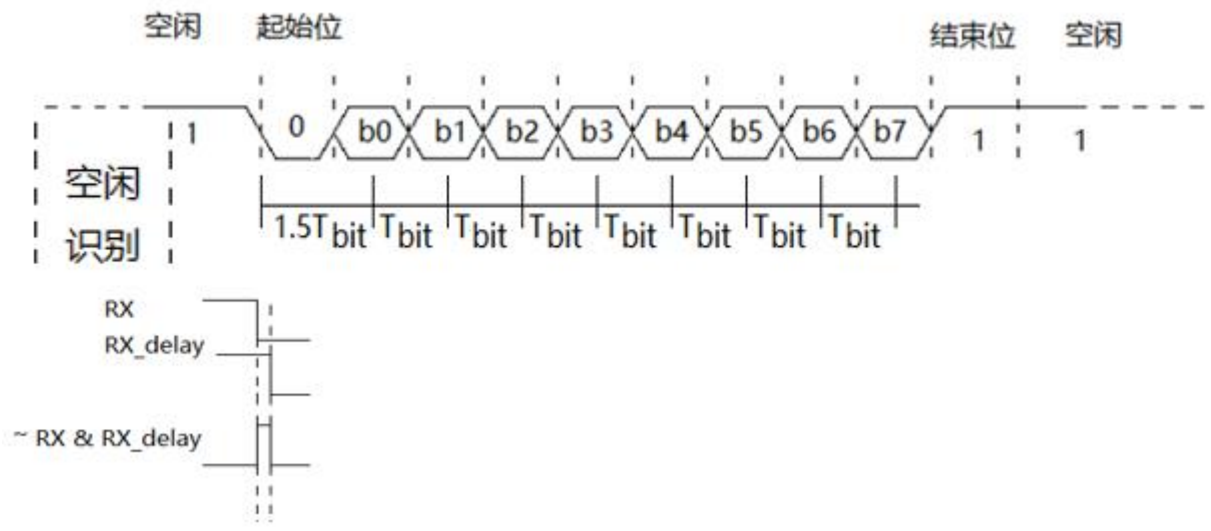


Fig 3.2.2 Serial data reception

3.2.1 State Machine Design of UART_RX

The data reception operation of UART_RX module can be designed as a simple state machine. A state machine is a behavior model which consists of a finite number of states and is therefore also called finite-state machine (FSM). Depending on the current state and the given inputs the machine performs state transitions and produces outputs. There are basic types like Mealy and Moore machines and more complex types like Harel and UML statecharts. Thus, the state machine design of UART_RX module is a simple design which can be designed based on the reception strategy of this module.

Figure 3.1.4 shows the serial data reception design of UART_RX module. The first step in the design is IDLE identification(空闲识别). As discussed before when the serial port RX is identified as high, it means the state machine is at IDLE state. Then the RX is pulled down to indicate that data transmission is about to start. The start bit is low for one-bit period T. This bit period is set by the baud rate. If the baud rate is slower, the bit period is longer, if the baud rate is faster, the bit period will be shorter. After the start bit is set up, the list significant bit (LSB) of the data bit will come out on the line first, then the next data bit will follow, and the most significant bit (MSB) will be received at last. In this fashion, the data bit (Bit 0, Bit 1, …, Bit 7) will be received serially. Finally, the stop bit, which is high will be received. Notice that we haven't implemented parity bit in our Verilog code design. Each data bit is separated by one-bit period T as set by the baud rate.

Because there is no clock in UART, it is up to the designer's design to find the middle of each data bit. For example, assume that the one-bit period T is one microsecond wide, it is necessary to sample the bit at a point when half of a microsecond has gone by to make sure that the sampling is not at the transition point. If the sampling occurs at the transition points, it could have a situation where it is sampling the wrong data. Thus, the best way to sample the right data is to sample the middle of the data bits. In the above figure we can see that the start bit is low for one-bit period T. The period 1.5T is the sum of the start bit Period T and the half bit period 0.5T of the next bit (Bit 0) as Bit 0 has been sampled at its middle point. After that, the one-bit period T will be counted from middle point of Bit 0 to the middle point of Bit 1 and the next one from the middle point of Bit 1 to the middle point of Bit 2 … in this pattern, it continues till middle point of Bit 7 then sampling is finished. So, When the complete data packet is received by the receiver module, the stop is pulled up to indicate the end of byte reception. Remember that the data packet can be 8 bits long or less but cannot be more that 8 bits in this design.

So, the entire state machine design can be emphasized as follow:

State 1: idle identification;

State 2: equal starting position;

State 3: receiving B0;

State 4: receiving B1;

State 5: receiving B2;

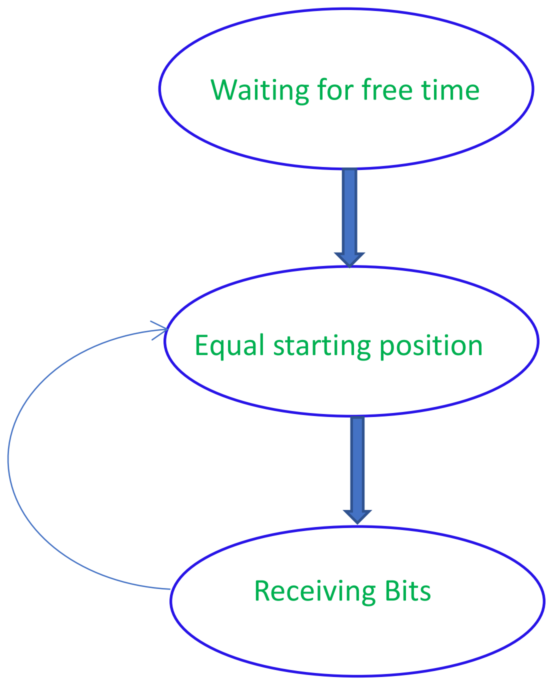
State 6: receiving B3.

State 7: receiving B4;

State 8: receiving B5;

State 9: receiving B6;

State 10: receiving B7;



→at the idle state, RX is high for long time

→when the falling age is identified, RX is low for 1 bit period indicating that it’s ready for reception

→after receiving all 8-bits, the system jumps back to the equal starting positions.

Fig 3.2.3   State machine design of UART_RX module

3.2.2 Verilog HDL Code design of UART_RX Module

The UART_RX module is the receiver module where six input output port lists assigned to it. The input port lists are RX, clk and res. The output port lists are data_out, en_data_out and led. The input port RX is the port where the 8-bit serial data and commands inputted to the UART_RX module of the UART and should be connected to the computer directly. Within the UART_RX module, the baud rate is defined as a parameter and set to 9600 bps and the system clock frequency is set 24MHz by default. The clk is used to enter the 24MHz clock of the FPGA development board (Altera cyclone I) to the module. This clock frequency must be the same for the transmitter module. The reset signal res is a signal which activate the UART, that is, when the res port is not high, everything will set to zero and when res port is high, the system proceeds to the waiting state where it could check other conditions. The output port data_out is the port where the converted 8-bit parallel data is handed to the cmd_pro module of the design. The en_data_out is a signal which let the data_out port active. When en_data_out is low, the system receives all 8 bits and hand all of them to the cmd_pro module then when finished, the en_data_out signal triggered high.

The UART_RX code is designed as follow:

```
 //2021_UART project
//UART Receiver Module
`timescale 1ns/10ps
module UART_RX(
                                        clk,
                                        res,
                                        RX,
                                        data_out,
                                        en_data_out,
                                        led
                                        );
input                     clk;
input                      res;
input                      RX;//Received data
output[7:0]                data_out;//8bits data output
output                     en_data_out;//Output data enable
output[7:0]                  led;
```

```
reg[7:0]                                                        led;

parameter                                   freg_clk=24;//Clock frequency;
parameter                                   baud_rate=9600;//Baud rate;
parameter                                   bit_width=freg_clk*1000000/baud_rate;//bit width

reg[7:0]                                                      data_out;
reg[7:0]                                                      state;//Master state machine;
reg[14:0]                                                     con;//Counter;
reg[3:0]                                                      con_bits;//Data bit counter;

reg                                                           RX_delay;//RX delay 1 clock;
reg                                                                       en_data_out;
always@(posedge clk or negedge res)
if(~res) begin
        state<=0;con<=0;con_bits<=0;RX_delay<=0;
        data_out<=0;en_data_out<=0;led<=0;
end
else begin
        RX_delay<=RX;
        case(state)
        0://Waiting for free time;
        begin
                if(con==bit_width-1) begin
                        con<=0;
                end
                else begin
                        con<=con+1;
                end
                if(con==0) begin
                        if(RX) begin
                                con_bits<=con_bits+1;
                        end
                        else begin
                                con_bits<=0;
                        end
                end
                if(con_bits==12) begin
                        state<=1;
                end
        end
        1://Equal starting position;
        begin
                en_data_out<=0;
                if((~RX)&RX_delay) begin
                        state<=2;
                end
        end
        2://Receive data_ out[0];
        begin
                if(con==bit_width-1) begin
                        con<=0;
                        data_out[0]<=RX;
                        state<=3;
                end
                else begin
```

```
                con<=con+1;
        end
end
3://Receive data_ out[1];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[1]<=RX;
                state<=4;
        end
        else begin
                con<=con+1;
        end
end
4://Receive data_ out[2];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[2]<=RX;
                state<=5;
        end
        else begin
                con<=con+1;
        end
end
5://Receive data_ out[3];
begin
        if(con==bit_width-1) begin
```

```
                con<=0;
                data_out[3]<=RX;
                state<=6;
        end
        else begin
                con<=con+1;
        end
end
6://Receive data_ out[4];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[4]<=RX;
                state<=7;
        end
        else begin
                con<=con+1;
        end
end
7://Receive data_ out[5];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[5]<=RX;
                state<=8;
        end
        else begin
                con<=con+1;
        end
end
8://Receive data_ out[6];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[6]<=RX;
                state<=9;
        end
        else begin
                con<=con+1;
        end
end
9://Receive data_ out[7];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[7]<=RX;
                state<=10;
        end
        else begin
                con<=con+1;
        end
end
10://Sending data enable signal;
begin
        en_data_out<=1;
        led<=data_out;
```

```
                state<=1;
        end
        default://
        begin
                state<=0;
                con<=0;
                con_bits<=0;
                en_data_out<=0;
        end
        endcase
end
endmodule
```

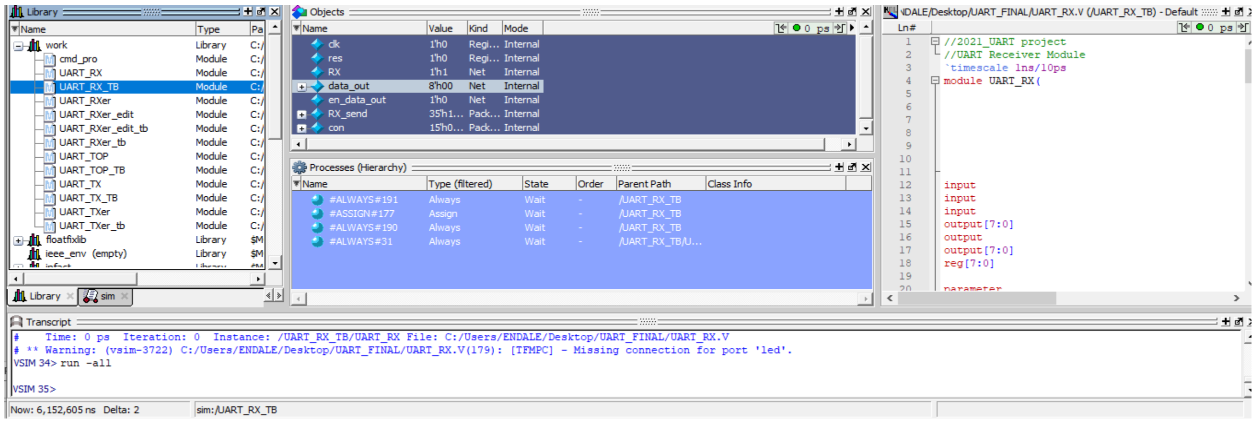
The UART_RX module a bit complicated module in this project design. There are three variables defined as a parameter in this module such that freq_clk, which is set to 24MHz, baud_rate, which is set to 9600bps and bit_width, which is set as bit_width = freq_clk*1000000/baud_rate.

Thus, in the always block, the system first checks the reset signal. If res is not asserted, everything is set to zero. If res is asserted, the RX_delay is loaded with the RX value and then the system jumps to the case statement based on the state machine viable called state. Case 0 is the waiting state. In this state, case statement parallelly checks the counter signal con whether it is bit_width -1 = 2500. If it is, con is loaded with 0, if not, it is set to con=con+1, to increment until it reaches this condition. Then if con=0, the system checks if RX is asserted. If asserted, bit counter signal con_bits is set to increment. If not, con_bits is loaded with 0. The system then checks the third condition, which allows the state machine to jump to the next state. Thus, if the con_bits = 12, the state machine jumps to state 1. State 1 is the equal starting position. In this state, the en_data_out signal is set 0 and the system checks if the RX signal is not asserted and the RX_delay signal is asserted, if the two conditions happen, the state machine will jump to the state 2. At the state 2, the receiver starts receiving data bits one by one based on the conditions. The first data bit to be received is LSB and the last one is MSB. Thus, when the counter reaches the condition bit_width-1, the counter over starts from 0 and the LSB of data_out is loaded with RX and the state will jump to the next state. Otherwise, the counter will keep incrementing. In this fashion, all 8 bits will be received sequentially. When the MSB is received, that means at state 9, the state jumps to state 10 and at state 10, the en_data_out is triggered to high and the state returns back to the equal starting position, which is state 1. The default condition is to set the baud rate counter con, the bit counter con_bits, the state and the data_out enable signal en_data_out all to zero. After this, the case statement is ended by the endcase keyword and the module also ended by the endmodule keyword.

3.2.3 Simulation of UART_RX Module

The simulation of each module in this project is done on the ModelSim simulator software. ModelSim Software is a multi-language environment simulator by Mentor Graphics, for simulation of hardware description languages such as VHDL, Verilog and SystemC, and includes a built-in C debugge. It can be used independently, or in conjunction with Intel Quartus Prime, Xilinx ISE or Xilinx Vivado. ModelSim Simulation is done by using the graphical user interface (GUI), or automatically using scripts.

The UART_RX module code is designed as shown in the above section, but it still necessary to design its testbench code to simulate it on the ModelSim simulator software. The testbench code is a separate module than UART_RX module. To simulate on ModelSim and check whether the designed module is functional, the UART_RX module and its testbench module can be concatenated under the same file but should be separated from each other. During simulation time, after compiling the file that contains both modules, a library called work will be created just after compilation. This library shows both modules, the main module and the testbench module, inside it. It can be seen like this.



Library

Name	Type	Pa
work	Library	C:/
cmd_pro	Module	C:/
UART_RX	Module	C:/
UART_RX_TB	Module	C:/
UART_RXer	Module	C:/
UART_RXer_edit	Module	C:/
UART_RXer_edit_tb	Module	C:/
UART_RXer_tb	Module	C:/
UART_TOP	Module	C:/
UART_TOP_TB	Module	C:/
UART_TX	Module	C:/
UART_TX_TB	Module	C:/
UART_TXer	Module	C:/
UART_TXer_tb	Module	C:/
floatfixlib	Library	$M
ieee_env (empty)	Library	$M

Objects

Name	Value	Kind	Mode
clk	1'h0	Regi...	Internal
res	1'h0	Regi...	Internal
RX	1'h1	Net	Internal
data_out	8'h00	Net	Internal
en_data_out	1'h0	Net	Internal
RX_send	35'h1...	Pack...	Internal
con	15'h0...	Pack...	Internal

Processes (Hierarchy)

Name	Type (filtered)	State	Order	Parent Path	Class Info
#ALWAYS#191	Always	Wait	-	/UART_RX_TB	
#ASSIGN#177	Assign	Wait	-	/UART_RX_TB	
#ALWAYS#190	Always	Wait	-	/UART_RX_TB	
#ALWAYS#31	Always	Wait	-	/UART_RX_TB/U...	

NDALE/Desktop/UART_FINAL/UART_RX.V (/UART_RX_TB) - Default

```
//2021_UART project
//UART Receiver Module
`timescale 1ns/10ps
module UART_RX(







input
input
input
output[7:0]
output
output[7:0]
reg[7:0]

parameter
```

Transcript

```
#    Time: 0 ps  Iteration: 0  Instance: /UART_RX_TB/UART_RX File: C:/Users/ENDALE/Desktop/UART_FINAL/UART_RX.V
# ** Warning: (vsim-3722) C:/Users/ENDALE/Desktop/UART_FINAL/UART_RX.V(179): [TFMPC] - Missing connection for port 'led'.
VSIM 34> run -all

VSIM 35>
```

Now: 6,152,605 ns Delta: 2 sim:/UART_RX_TB

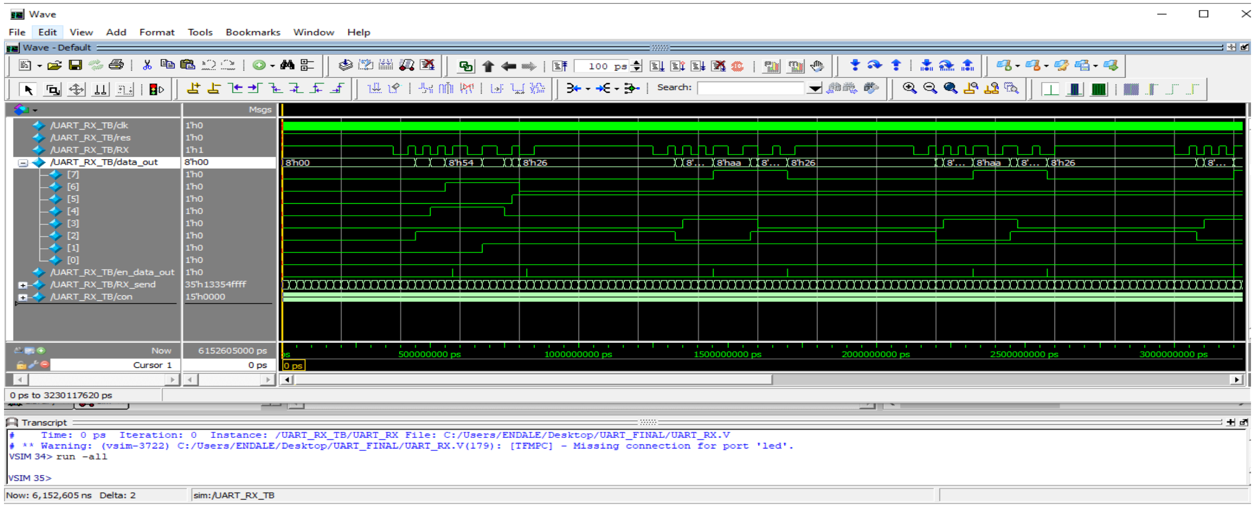
Fig 3.2.4 Work Library Created during Simulation

Thus, the testbench code of UART_RX has been design as shown down.

```
//------testbench for UART_RX module------

module UART_RX_TB;

reg             clk,res;

wire            RX;

wire[7:0]       data_out;

wire            en_data_out;
```

```
reg[34:0]                                   RX_send;//It contains serial port byte to send data
assign                                       RX=RX_send[0];//Connect RX;
reg[14:0]                               con;
UART_RX        UART_RX(    //instantiation of the same name
                                 clk,
                                  res,
                                   RX,
                                    data_out,
                                     en_data_out
                                  );
initial begin
        clk<=0;res<=0;RX_send<={1'b1,8'h26,1'b0,1'b1,8'haa,1'b0,16'hffff};con<=0;
                        #17                       res<=1;
end
always #5 clk<=~clk;
always@(posedge clk)  begin
        if(con==2500-1) begin
                con<=0;
        end
        else begin
                con<=con+1;
        end
        if(con==0) begin
                RX_send[33:0]<=RX_send[34:1];
                RX_send[34]<=RX_send[0];
        end
end
endmodule
```

The testbench module designed is named as “UART_TX_TB”. In the instantiation section, it instantiates the same port names as defined in the main module port lists. After instantiation, the initial block will begin and set the clk and res signal to 0. The predefined register is also set 0. The other predefined register called RX_send is loaded with the data frames that sent from pc. When all done, the res signal will be triggered and is set to high after passing 17 ns. At the always block, the clk signal is set to oscillate up down every 5 ns. The other always block is designed to check the bit_width condition. Bit_width is equal to the clk(24MHz) divide by the baud rate (9600 bps), which is equal to 2500. Thus, when the counter signal con reaches 2500-1, it is set back to 0 and will recycle in this fashion, otherwise it is set as con=con+1, to increment at every clock signal. The other condition is to be checked is what to do when con=0? Thus, when con=0, the RX_send register, which was loaded with data frames at the initial block will shift the loaded bits to the left in order to change the serial data type to the parallel. By doing so, the loaded data frames are converted to parallel data type and the data_out port of UART_RX will hand them to the cmd_pro module for the command execution.



/UART_RX_TB/clk	1'h0
/UART_RX_TB/res	1'h0
/UART_RX_TB/RX	1'h1
/UART_RX_TB/data_out	8'h00
[7]	1'h0
[6]	1'h0
[5]	1'h0
[4]	1'h0
[3]	1'h0
[2]	1'h0
[1]	1'h0
[0]	1'h0
/UART_RX_TB/en_data_out	1'h0
/UART_RX_TB/RX_send	35'h13354ffff
/UART_RX_TB/con	15'h0000
Now	6152605000 ps
Cursor 1	0 ps

0 ps to 3230117620 ps

```
#    Time: 0 ps  Iteration: 0  Instance: /UART_RX_TB/UART_RX File: C:/Users/ENDALE/Desktop/UART_FINAL/UART_RX.V
# ** Warning: (vsim-3722) C:/Users/ENDALE/Desktop/UART_FINAL/UART_RX.V(179): [TFMPC] - Missing connection for port 'led'.
VSIM 34> run -all

VSIM 35>
```

Now: 6,152,605 ns Delta: 2   sim:/UART_RX_TB

Figure 3.2.4 Simulation result of UART_RX Module

As it can be seen on the above figure, the RX_send register was loaded with 36-bit hexadecimal data frames as RX_send<={1'b1,8'h26,1'b0,1'b1,8'haa,1'b0,16'hffff}. The data bits to be received in this data frames are two 8-bit data such that 8'h26 and 8'haa and it can be seen that the data_out port has received both 8-bit data as shown in the wave form of the data_out port.

3.3Design and Simulation of UART Transmitter module (UART_TX)

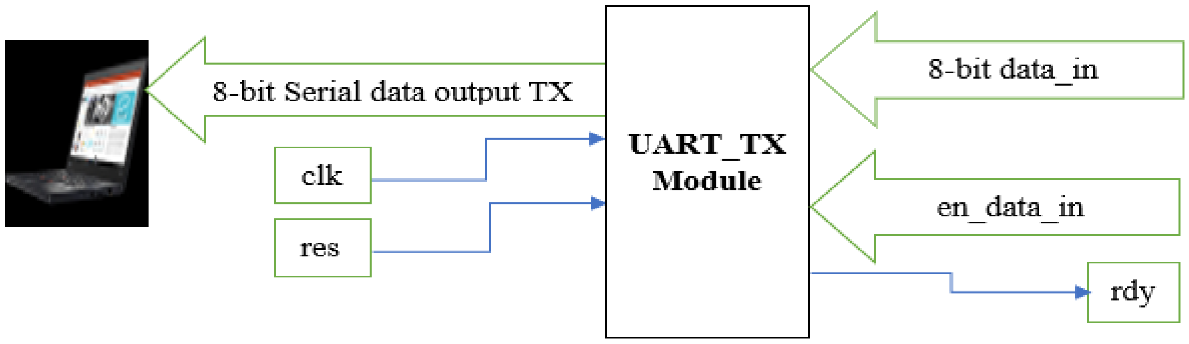


Fig 3.3.1 UART_TX Module diagram

UART_TX module is basically a shift register that loads parallel data from the cmd_pro module via the 8-bit data_in port and shifts it in a specific rate through TX pin of the computer. The module consists of six input output port lists, four input ports and two output ports. The input ports are data_in, en_data_in, clk and res. The port data_in is a port where the 8-bit parallel data inputted from cmd_pro module after command execution. The en_data_in port is a port which is used to get data_in enabler signal from the cmd_pro module. clk and res signals are the same thing as explained in the UART_RX module. The two output ports are TX and rdy. TX is a serial output pin where the 8-bit data will be transmitted through and rdy is an idle flag signal to conform that the transmitter is ready to transmit and it is high when the byte is sent.

3.3.1 State Machine Design of UART_TX Module

The transmission operation of UART_TX module can also be explained as a simple state machine. The TX pin of UART_TX Module is at logic level 1 when the transmitter is in idle state and when a data frame is ready to be sent, TX pin is pulled down to logic level 0 for one bit period. When the transmission starts, a falling edge is created on the data transmission line which wakes up the receiver of the computer. After that, the clock is set according to the baud rate and all bits are sent one by one in every clock cycle in the transmitter side of UART_TX module. When the transmission operation finalizes, the TX pin of UART_TX module is set to logic level 1 for one or two-bit widths to inform the receiver that the transmission is done. These are also called stop bit(s). The number of stop bits needs to be predetermined so that the transmitter and receiver have same settings. Within the UART_TX module, the baud rate is defined as a parameter and set to 9600 bps and the system clock frequency is set 24MHz by default.

Thus, the state machine of UART_TX module can be designed in a way that if the reset signal is low, the transmitter will be at idle state, which means TX=1, otherwise the transmitter will jump to the next state, which is waiting for en_data_in to be logic level 1. If so, it will fill in the send register to shift and send and finally it will shift the send register to the right and send serially. The LSB bit comes first to the TX pin.

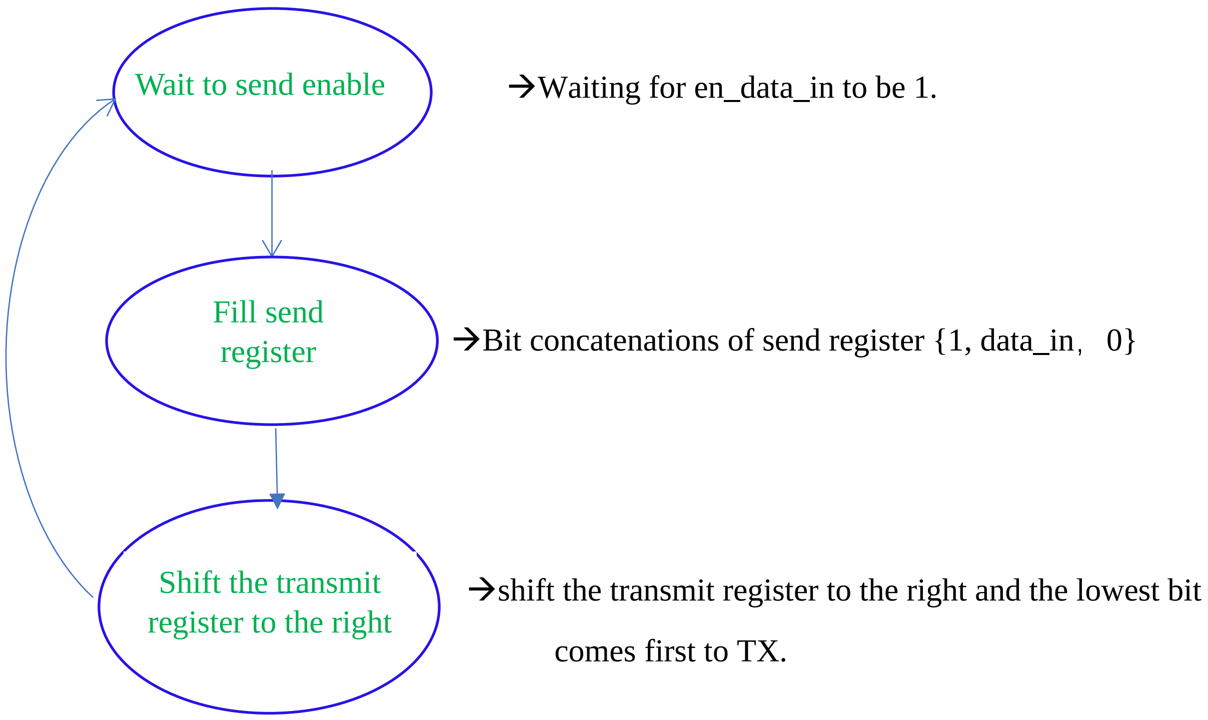

Wait to send enable
→Waiting for en_data_in to be 1.
Fill send register
→Bit concatenations of send register {1, data_in， 0}
Shift the transmit register to the right
→shift the transmit register to the right and the lowest bit comes first to TX.

Fig 3.3.2 State machine design of UART_TX module

3.3.2 Verilog HDL Code design of UART_TX Module

The Verilog code design of UART_TX module is relatively simpler than that of UART_RX module. The main goal of this module is to be able receive the 8-bit data from the instruction controller module and send out to the computer. It is connected with cmd_pro module via the rdy signal in order to identify whether the transmitter is busy or not. If not busy, the rdy signal will be set to logic level low and if busy, it will be set to logic level 1. The clk and res signal are common signal with the receiver module. The complete code is designed as follow:

```
//2021 UART Project

//UART Transmitter Module

`timescale 1ns/10ps

module UART_TX(

                                        clk,

                                        res,

                                        data_in,

                                        TX,
                                         Rdy
                                        en_data_in,
                                         );
input                                            clk;
input                                            res;
input[7:0]                                       data_in;//Bytes to be sent;
input                                            en_data_in;//Send enable signal;
output                                           TX;
output                                           rdy;//higher bit is busy;
parameter                                        baud_rate=9600;//Hz;
parameter                                        frequency_clk=24;//MHz;

reg                                             state;//Main state machine;
reg                                              TX;
reg                                               rdy;
reg[9:0]                                         data_buf;//Cache the input byte data;
reg                                              bit_pulse;
reg[31:0]                                        con;//For counting;
reg[3:0]                      con_bit_send;//The number of bits sent;
//assign                      TX=data_buf[0];//TX is data_ The lowest position of buf;
always@(posedge clk or negedge res) begin
if(~res) begin
    state<=0;rdy<=0;data_buf<=10'h0;bit_pulse<=0;con<=0;con_bit_send<=0;TX<=1;
end
else begin
    case(state)
    0://Wait for the enable signal to be sent;
    begin
```

```
                con_bit_send<=0;
                if(en_data_in) begin //Wait until the transmission is enabled;
                        data_buf<={1'b1,data_in,1'b0};//The highest end bit, the lowest start bit,
and the middle 8 bits are data;
                        rdy<=1;//busy;
                        state<=1;
                end
                else begin
                        rdy<=0;//Not busy;
                end
        end
        1://Sending data;
        begin
                if(con==(frequency_clk*1000000)/baud_rate) begin //Generate bit_ Pulse;
                        bit_pulse<=1;
                        con<=0;
                end
                else begin
                        bit_pulse<=0;
                        con<=con+1;
                end
                if(bit_pulse) begin //data_ BUF moves to the right;
                        TX<=data_buf[0];
                        data_buf[8:0]<=data_buf[9:1];
                        con_bit_send<=con_bit_send+1;//Record the number of bits sent;
                end
                if(con_bit_send==10) begin //Sending completed;
                        state<=0;rdy<=0;//Not busy;
                end
        end
        endcase
    end
    end
    endmodule
```

Thus, what this Verilog code is that, it first checks whether the reset signal is logic level low or high at the always block. If low set everything to 0, that is “if(~res) begin

state<=0;rdy<=0;data_buf<=10'h0;bit_pulse<=0;con<=0;con_bit_send<=0;TX<=1;

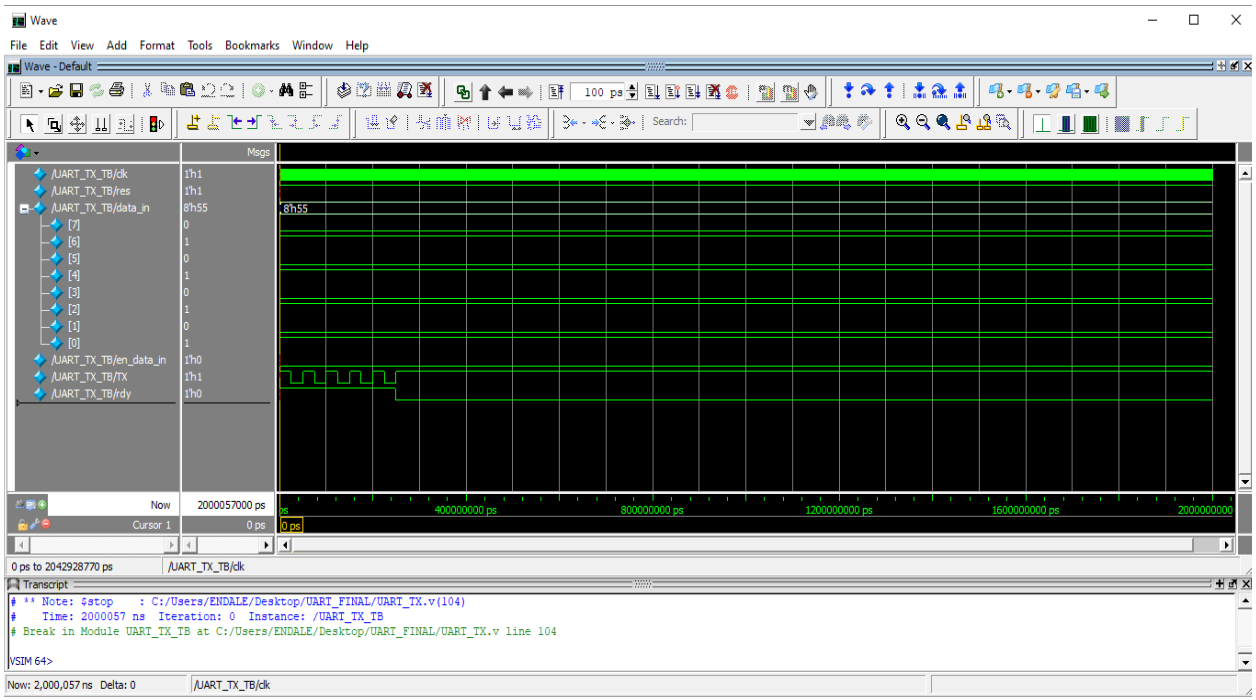
end” otherwise jump to the state machine cases. Case 0 of the state machine is the waiting state. In this state, it waits for the enable signal en_data_in to be sent. If the enable signal is received, it begins loading the data frame in to the data_buf register in a way that the start bit 1`b0 comes fist and then 8-bit data followed by the 1`b1 stop bit. If done so, the rdy signal is set to high indicating the transmitter is busy and the state machine jumps to state 1, otherwise the rdy signal is set to low indicating the transmitter is not busy. Case 1 of the state machine stars sending data according to the clock cycle and baud rate set as parameters. It first generates bit pulses based on the condition that if the counter is equal to the clock divide by baud rate. If the counter con meets the condition and bit pulse is generated, the data_buf register loaded with data frame is shifted to the right to convert the parallel data type to serial type. The ending of sending is checked by the other condition, which is con_bit_send signal. The data frame length to be sent is 10 bits, which is 1`b1, 8`b data_in, 1`bo, totally 10 bits. Thus, if the con_bit_send is equal to 10, it means the data frame has been set already and state machine will return back to the waiting state, which is case 0, and the rdy signal is set to 0, indicating the transmitter is ready to transmit the next data frame or not busy. This is all what the UART_TX module is designed to do in this project.

3.3.3 Simulation of UART_TX Module

The testbench code of UART_TX module is designed as follow to realize whether this code is correct or not.

```
//-----testbench for UART_TX------
module    UART_TX_TB;


reg                                           clk,res;
reg[7:0]                                      data_in;
reg                                           en_data_in;
wire                                          TX;
wire                                        rdy;
UART_TX    UART_TX(  // instantiation of port lists
                                                        clk,
                                                        res,
                                                        data_in,
                                                        en_data_in,
                                                        TX,
                                                        rdy
                                                        );

initial begin
                                    clk<=0;res<=0;data_in<=8'h55;en_data_in<=0;
                #17                 res<=1;
                #30                 en_data_in<=1;
                #10                 en_data_in<=0;

                #2000000            $stop;
end
always #5 clk<=~clk;

endmodule
```

This realization testbench code is designed load 8-bit hexadecimal number 55 as input data. The simulation period is set to 2000000 ns and clk signal is oscillating up down every 5 ns as it is indicated by the always block, “always #5 clk<=~clk”. The result of simulation can be seen in the following figure.



```
# ** Note: $stop    : C:/Users/ENDALE/Desktop/UART_FINAL/UART_TX.v(104)
#    Time: 2000057 ns  Iteration: 0  Instance: /UART_TX_TB
# Break in Module UART_TX_TB at C:/Users/ENDALE/Desktop/UART_FINAL/UART_TX.v line 104

VSIM 64>
```

Fig 3.3.3 Simulation Result of UART_TX Module

As it can be seen clearly that the testbench inputs data-in of 8-bit hexadecimal number 55 and it shows that the TX pin has transmitted the data_in. The binary equivalent of 55 is 01010101and the wave of TX shows this binary number.

3.4 Design and Simulation of Instruction Controller Module (cmd_pro)

The cmd_pro module is a module where the command execution will be done. This module is designed to perform the instruction set architecture (ISA) specified in this project. The ISA specified for this project are the following for operations.

ISA:

No.	Command	Result
1	Addition	A+B
2	Negation	~A
3	XOR operation	A^B
4	Multiplication	A*B

Table 3.4.1 Instruction Set Architecture of cmd_pro module

The cmd_pro module must be capable of executing the above four commands. This module gets data A and B from the receiver module (UART_RX). The command also received from the computer via the receiver module based on the code design of cmd_pro module. After the execution is completed, this module must also be able to return to the computer via the transmitter (UART_TX) module. Thus, the cmd_pro module gets its input data and also the input data enable signal from the UART_RX module. The diagram of cmd_pro module can be seen as follow:

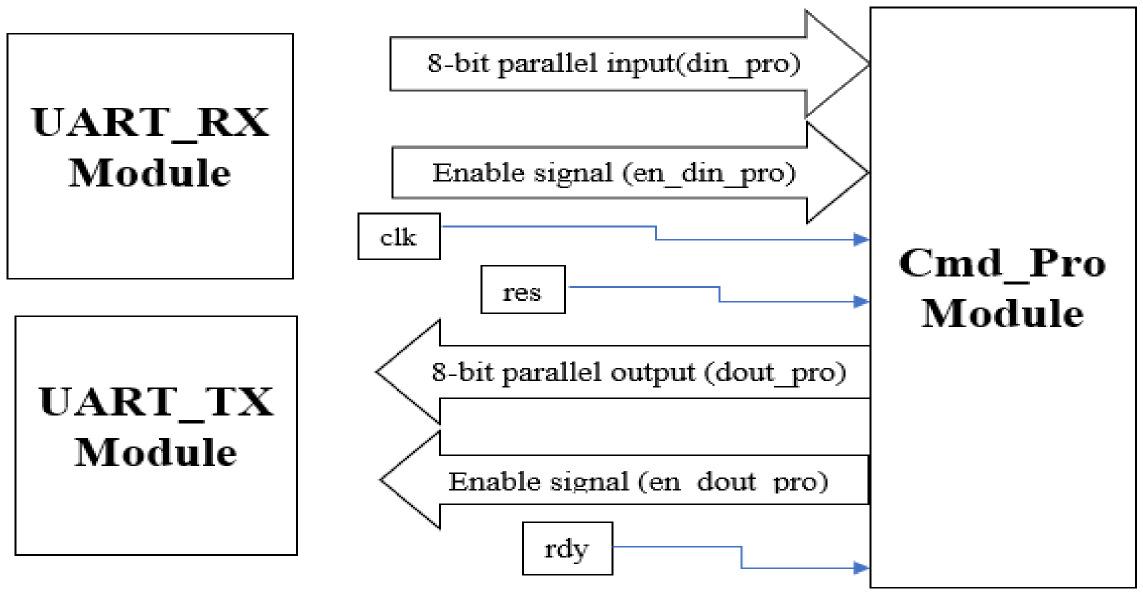

UART_RX Module
UART_TX Module
8-bit parallel input(din_pro)
Enable signal (en_din_pro)
clk
res
8-bit parallel output (dout_pro)
Enable signal (en_dout_pro)
rdy
Cmd_Pro Module

Fig 3.4.1 cmd_pro module input output diagram

3.4.1 State Machine Design of cmd_pro Module

The state machine design of cmd_pro module can be thought in performing three tasks in different state. Those tasks are data and instruction reception, data and instruction processing and result transmission. This can be drawn as shown in the figure below.

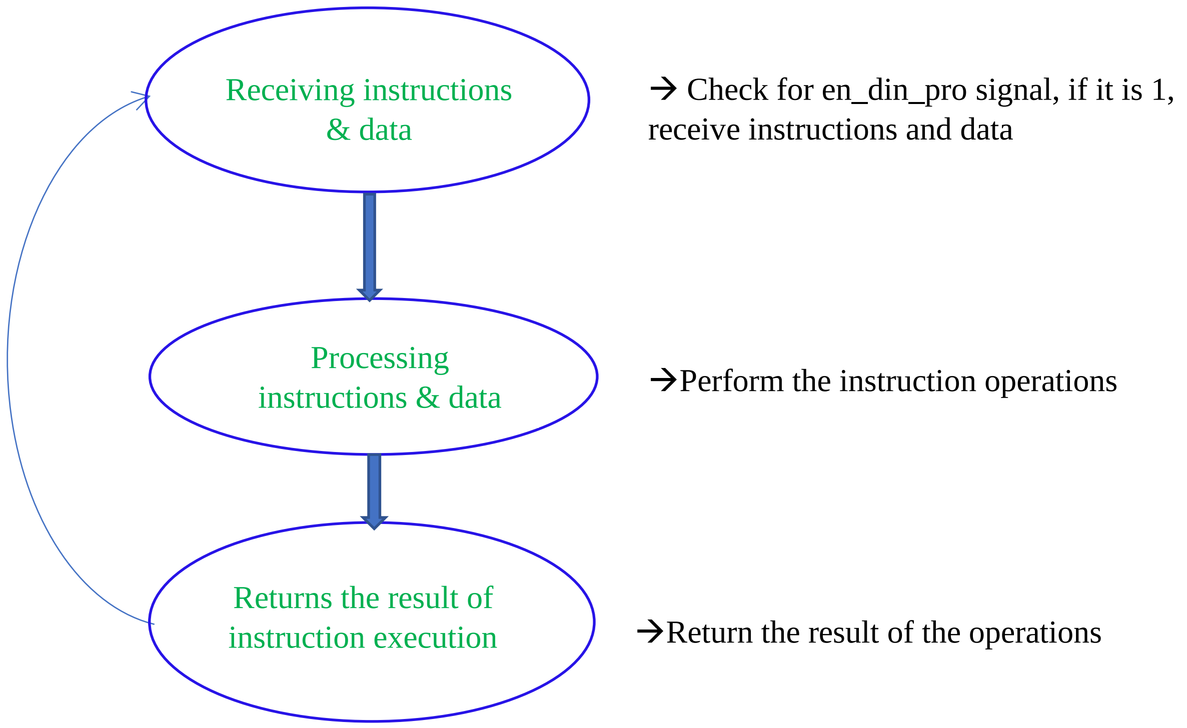

Receiving instructions & data
→ Check for en_din_pro signal, if it is 1, receive instructions and data
Processing instructions & data
→Perform the instruction operations
Returns the result of instruction execution
→Return the result of the operations

Fig 3.4.2 State Machine diagram of cmd_pro module

3.4.2 Verilog Code Design of cmd_pro Module

As it can been seen from the input output diagram of cmd_pro module, it has total of seven input output ports. The input ports are din_pro, where the 8-bit parallel input data comes through, en_din_pro, the input data enabler, clk, where the system clock signal entered through, res, the reset signal and rdy, where the ready signal comes through from the transmitter module. The output ports are dout_pro, where the 8-bit execution result data transmitted to the transmitter and en_dout_pro, the dout_pro enabler.

The main idea in designing this instruction controller module is to design a code that is capable of accepting commands as a first byte, accepting parameter "a" as a second byte and accepting parameter "b" as a third byte from the computer via the receiver module. The first byte can be stored temporarily in a register and the second byte and the third byte also do the same. Thus, we can define three register variables using reg keyword as reg [7:0] command, a, b; the first byte will be stored in command, second byte will be stored in a and third byte will be stored in b.

Another register variable called reg_result can also be defined in order to store the result of execution temporarily for each command executions. Thus, reg_result can be declared as reg[15:0] reg_result; The array size set to [15:0] because the operation results may be beyond 8 bits. For example, the result of multiplication operation, which is a*b, will be 16 bits. Considering this, the reg_result should be given enough array size where it could store all the result bits.

The complete Verilog code of cmd_pro module is designed as follow;

```
//2021 UART Project

// UART instruction controller module

module cmd_pro(

                                        clk,

                                        res,

                                        din_pro,

                                        en_din_pro,

                                        dout_pro,

                                        en_dout_pro,

                                        rdy

                                        );

input                              clk;
input                                   res;
input[7:0]                        din_pro;//the byte received from the RXer;
input                             en_din_pro;//inform the com_pro to receive the data_in;
output[7:0]                        dout_pro;//the instruction result;
output                              en_dout_pro;//enable the sending of the result;
input                                rdy;//0,when ready, 1, when busy;

reg                                     en_dout_pro;
reg[7:0]                          dout_pro;
reg[2:0]                          state;//the state machine;
reg[7:0]                          command,a,b;
reg[15:0]                         reg_result;//the buffer of the result;

/*          command ISA defination
                        command     result
                        8'h00             a+b
         8'h01              ~a
         8'h02              a^b
         8'h03              a*b
*/

always@(posedge clk or negedge res)
if(~res) begin
        state<=0;command<=0;a<=0;b<=0;reg_result<=0;en_dout_pro<=0;
end
else begin
        case(state)
        0://waiting for the first byte(command) from the PC;
```

```
begin
        if(en_din_pro) begin
                command<=din_pro;
                reg_result<=0;
                state<=1;
        end
end
1://waiting for the second byte(parameter a) from the PC;
begin
        if(en_din_pro) begin
                a<=din_pro;
                state<=2;
        end
end
2://waiting for the third byte(parameter b) from the PC;
begin
        if(en_din_pro) begin
                b<=din_pro;
                state<=3;
        end
end
3://process the command to get the result;
begin
        state<=4;
        case(command)
        8'h00://a+b
        begin
                reg_result<=a+b;
        end
        8'h01://~a
        begin
                reg_result<={8'h00,~a};
        end
        8'h02://a^b
        begin
                reg_result<={8'h00,a^b};
        end
        8'h03://a*b
        begin
                reg_result<=a*b;
        end
        default://do nothing
        begin
                reg_result<=0;
        end
        endcase
end
4://sending the result, the LSB byte;
begin
        if(~rdy) begin
                dout_pro<=reg_result[15:8];
                en_dout_pro<=1;
                state<=5;
        end
end
5://setting en_send back to 0;
```

```
        begin
                en_dout_pro<=0;
                state<=6;
        end
        6://sending the result, the MSB byte;
        begin
                if(~rdy) begin
                        dout_pro<=reg_result[7:0];
                        en_dout_pro<=1;
                        state<=7;
                end
        end
        7://setting en_send back to 0;
        begin
                en_dout_pro<=0;
                state<=0;
        end
        endcase
    end
    endmodule
```

Thus, as it can be seen in the code, after defining the register variable, command, a, b and reg_result, an always block is stated. In the always block, the system first checks if the reset signal res is asserted or not. If not asserted, everything is set 0, otherwise the system begins a case statement based on the state variable of state machine. Thus, at state 0, the system checks the input data enable signal en_din_pro. If it is asserted, the system begins accepting the first byte, which is the command, through din_pro port as “command<=din_pro;” and the reg_result is still set to zero as the system has not accepted the parameters yet. Then, the state jumps to state 1. At state 1, the system again checks the enable signal and if asserted, the second bye, which is parameter “a” will be received. Then, the state jumps to state 2. Here the reg_result is not set to 0, because operation can be made on parameter “a” based on the command received at first byte. At state 2, the system again checks the enable signal and if asserted, the third byte, which is parameter “b” will be received and state jumps to state 3. State 3 is the command execution state.

Thus, at this state a new case statement based on command is begun. To perform the four command, the code can be designed in a way that the command variable should receive 8-bit hexadecimal number 00 (8'h00) to perform the addition operation. In the same fashion, 8'h01 to perform negation, 8'h02 to perform logic XOR operation and 8'h03 to perform multiplication operation. Thus, the first byte to be inputted should be either of the above four hexadecimal number as per the required command to be executed. The second byte and the third byte can be any random hexadecimal number. A default statement is introduced to let the system do nothing. The default statement begins and the reg_result is set to zero. Then the case statement is ended by the endcase keyword and state jumps to state 4. State 4 is a state where the LSB byte will be sent out. At this state, the system first checks if the rdy signal from the transmitter module is asserted or not. If not asserted, means the transmitter is ready to transmit 8-bit data, the system loads the dout_pro with the LSB byte of reg_result as “dout_pro<=reg_result [15:8];” and the dout_pro enabler signal en-dout_pro is triggered high to let out the LSB byte moves out, then the state jumps to state 5. At state 5, the en_dout_pro set back to 0 to transmit the MSB byte and then state jumps to state 6. At state 6, the system again checks if the rdy signal is not asserted, if so, dout_pro is loaded with MSB byte of reg_result as “dout_pro<=reg_result [7:0];” and the en_dout_pro triggered high. Then the state jumps to the final state which is state 7. At this state, the system set the en_dout_pro to 0 and set state to 0 as well in order to return the state machine back to the data and command receiving state. Then, the case statement is closed with endcase and so does the main module with endmodule.

3.4.3 Simulation of cmd_pro Module

A stimulus testbench code can be designed for the cmd_pro module in order to verify whether the designed code works correctly. The cmd_pro module has five input port lists. It is necessary to have a close look at those port lists in order to design the testbench code. Thus, the first four input ports, din_pro, en_din_pro, clk and res ports has to be defined as a reg in the testbench module.

The cmd_pro module is designed to get 3 8-bit data as a command, parameter a and parameter b. Therefore, it is necessary to define a temporary register variable which can store 24-bit data. This can be defined as “reg [23:0]   data_reg;”. Also need to define another register variable to implement a case statement to let the command variable get the first byte of 24-bit data, which is data_reg [23:0] and then to let parameter a get the second byte, which is data_reg [15:8], finally to let parameter b get the third byte, which is data_reg [7:0]. The complete testbench code has been designed as shown below;

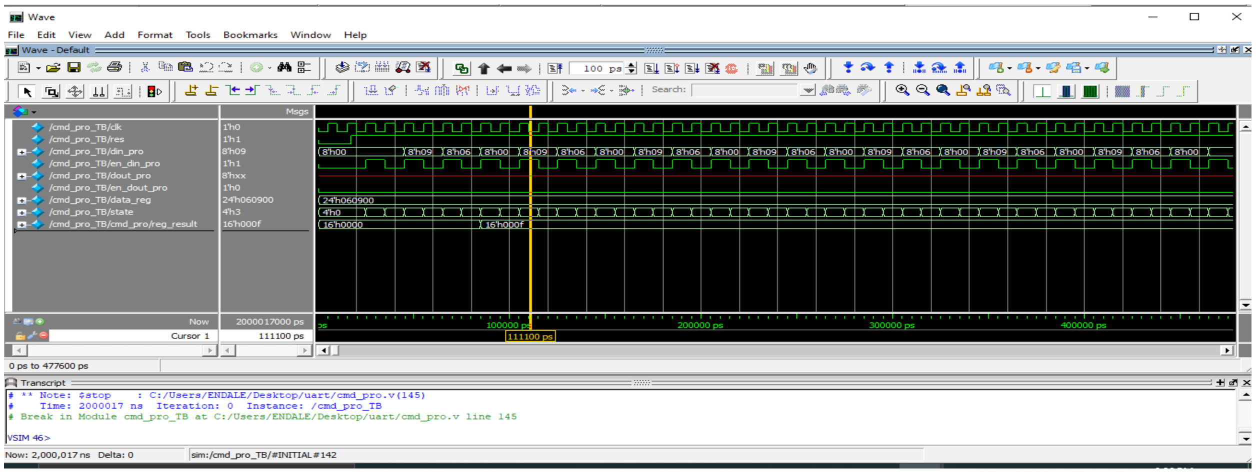
```
//----------testbench code for cmd_pro----

module cmd_pro_TB;

reg                   clk,res;

reg[7:0]              din_pro;

reg                                                   en_din_pro;
```

```
wire[7:0]                    dout_pro;
wire                                 en_dout_pro;
reg[23:0]                            data_reg;
reg[3:0]                             state;
cmd_pro    cmd_pro(
           .clk(clk),
           .res(res),
           .din_pro(din_pro),
           .en_din_pro(en_din_pro),
           .dout_pro(dout_pro),
           .en_dout_pro(en_dout_pro),
           .rdy(1'b1)
           );
initial begin
              clk<=0;res<=0;data_reg<={8'h06,8'h09,8'h00};
                     #17                        res<=1;
                     #2000000           $stop;
end
 always #5 clk<=~clk;
always@(posedge clk or negedge res)
  if (!res)begin
         din_pro<=0;
         en_din_pro<=0;
         state<=0;
  end
 else begin
       case(state)
       0:
```

```
begin
  din_pro<=data_reg[7:0];
  en_din_pro<=1;
  state<=1;
end
1:
begin
  en_din_pro<=0;
  state<=2;
end
2:
begin
   din_pro<=data_reg[15:8];
   en_din_pro<=1;
   state<=3;
end
3:
begin
  en_din_pro<=0;
  state<=4;
end
4:
begin
  din_pro<=data_reg[23:16];
  en_din_pro<=1;
  state<=5;
end
5:
```

```
            begin
            en_din_pro<=0;
            state<=0;
            end
            endcase
    end
endmodule
```

As there are four operation that the cmd_pro module is expected to perform, however, the module is designed three 8-bit data at a time. Hence, we can simulate the testbench code four times for each operation by switching the first byte in the data_reg register, which is data_reg [23:16] of the testbench code to 00, 01, 02 and 03. The above code performs the addition as the data_reg [23:16] is 00. The result has been seen in the figure below and because the value of a is 09 and the value of b is 06 and the operation done is addition. Hence the result should be 09+06 = 015= 0F. The result has been seen as 16'h000F.



```
/cmd_pro_TB/clk                  1'h0
/cmd_pro_TB/res                  1'h1
/cmd_pro_TB/din_pro              8'h09
/cmd_pro_TB/en_din_pro           1'h1
/cmd_pro_TB/dout_pro             8'hxx
/cmd_pro_TB/en_dout_pro          1'h0
/cmd_pro_TB/data_reg             24'h060900
/cmd_pro_TB/state                4'h3
/cmd_pro_TB/cmd_pro/reg_result   16'h000f

Now       2000017000 ps
Cursor 1  111100 ps
0 ps to 477600 ps
```

```
# ** Note: $stop    : C:/Users/ENDALE/Desktop/uart/cmd_pro.v(145)
#    Time: 2000017 ns  Iteration: 0  Instance: /cmd_pro_TB
# Break in Module cmd_pro_TB at C:/Users/ENDALE/Desktop/uart/cmd_pro.v line 145

VSIM 46>
```

```
Now: 2,000,017 ns  Delta: 0    sim:/cmd_pro_TB/#INITIAL#142
```

Fig 3.4.3 Addition operation of cmd_pro

The next step is to switch the first byte of data_reg to 01 to perform the negation of a. The negation of a is ~ (00001001) = (111111001) = F9, this can be seen as 16'h00F9 in the figure below.

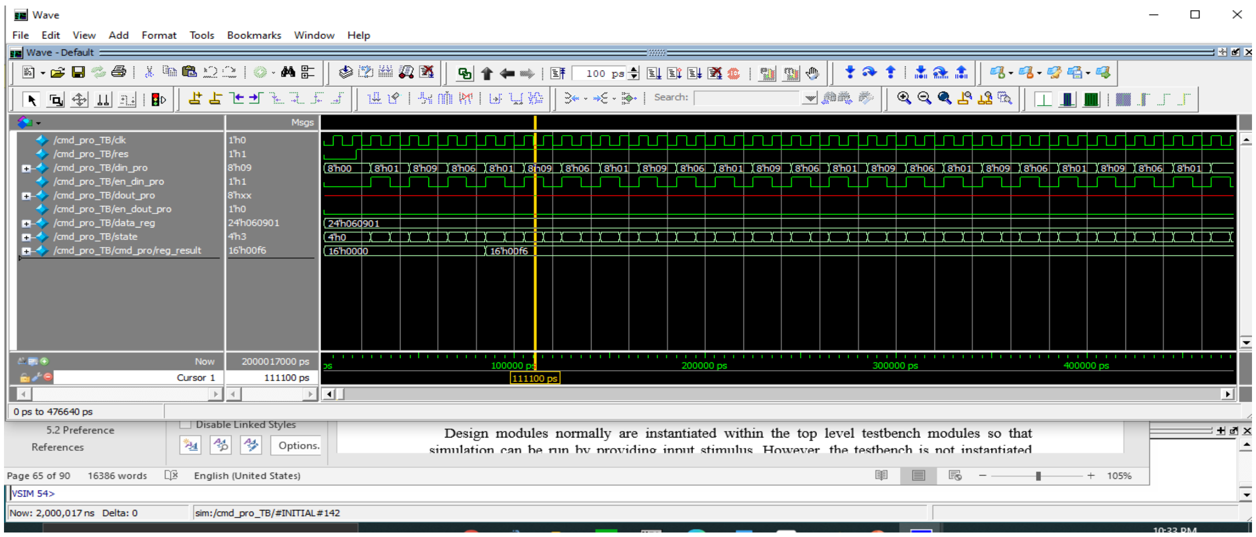
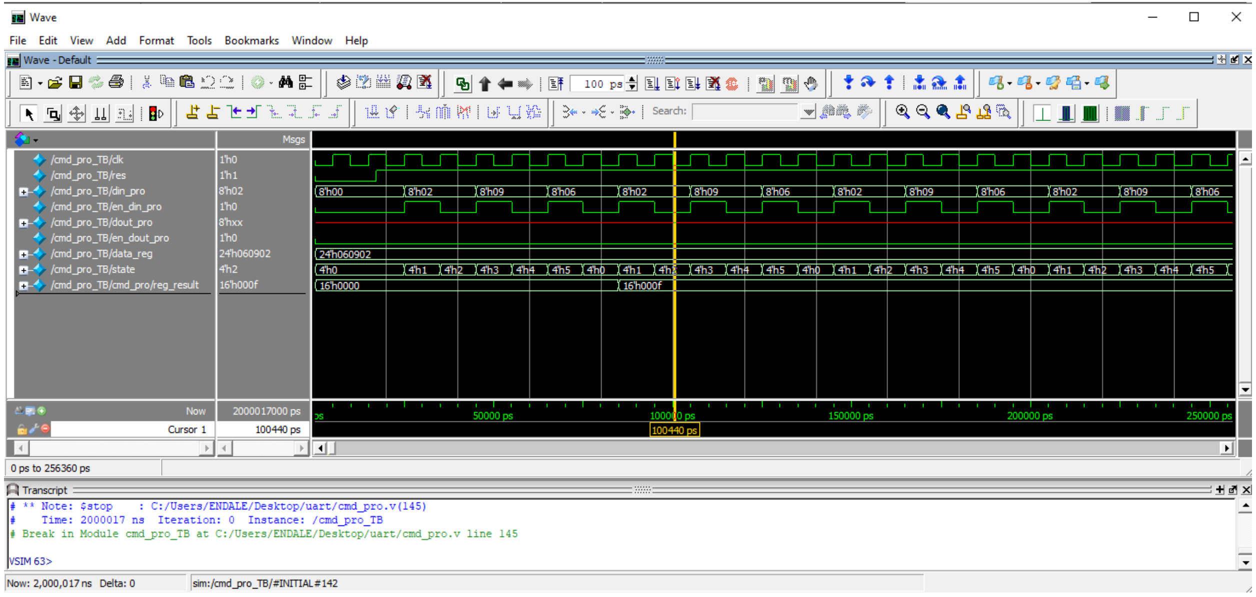

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Msgs
/cmd_pro_TB/clk 1'h0
/cmd_pro_TB/res 1'h1
/cmd_pro_TB/din_pro 8'h09
/cmd_pro_TB/en_din_pro 1'h1
/cmd_pro_TB/dout_pro 8'hxx
/cmd_pro_TB/en_dout_pro 1'h0
/cmd_pro_TB/data_reg 24'h060901
/cmd_pro_TB/state 4'h3
/cmd_pro_TB/cmd_pro/reg_result 16'h00f6
Now 2000017000 ps
Cursor 1 111100 ps
100000 ps
200000 ps
300000 ps
400000 ps
0 ps to 476640 ps
Now: 2,000,017 ns Delta: 0
sim:/cmd_pro_TB/#INITIAL#142
VSIM 54>

Fig 3.4.4 Negation Operation of cmd_pro

The next operation is logic XOR. This can be done by switching data_reg [23:16] to 02. The result should be (09)^(06) = (00001001)^(00000110) =(00001111) = 0F. This can be seen as 16'h000F in the reg_result in the figure below.

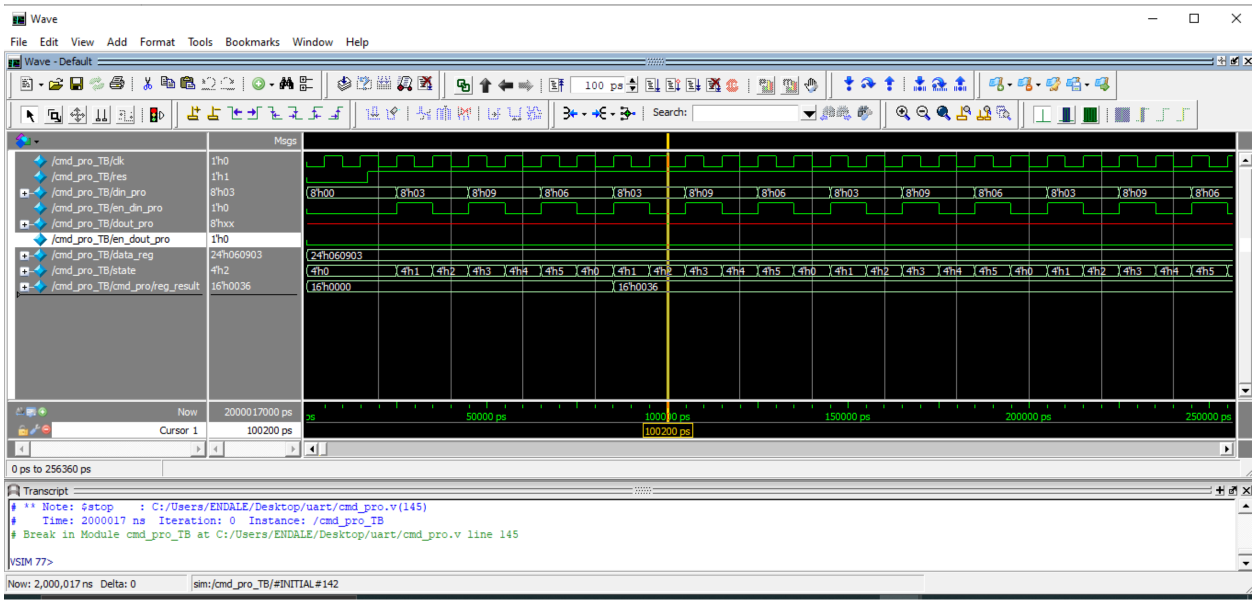

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Msgs
/cmd_pro_TB/clk 1'h0
/cmd_pro_TB/res 1'h1
/cmd_pro_TB/din_pro 8'h02
/cmd_pro_TB/en_din_pro 1'h0
/cmd_pro_TB/dout_pro 8'hxx
/cmd_pro_TB/en_dout_pro 1'h0
/cmd_pro_TB/data_reg 24'h060902
/cmd_pro_TB/state 4'h2
/cmd_pro_TB/cmd_pro/reg_result 16'h000f
Now 2000017000 ps
Cursor 1 100440 ps
50000 ps
100000 ps
150000 ps
200000 ps
250000 ps
0 ps to 256360 ps
Transcript

```
# ** Note: $stop    : C:/Users/ENDALE/Desktop/uart/cmd_pro.v(145)
#    Time: 2000017 ns  Iteration: 0  Instance: /cmd_pro_TB
# Break in Module cmd_pro_TB at C:/Users/ENDALE/Desktop/uart/cmd_pro.v line 145

VSIM 63>
```

Now: 2,000,017 ns Delta: 0
sim:/cmd_pro_TB/#INITIAL#142

Fig 3.4.5 Logic XOR operation of cmd_pro

The last operation is the multiplication which is done by switching data_reg [23:16] to 03. The result of the multiplication of a and b is a*b = (09)*(06) = (00001001)*(00000110) = 16'h0036. This has been verified in the figure below.



```
/cmd_pro_TB/clk                   1'h0
/cmd_pro_TB/res                   1'h1
/cmd_pro_TB/din_pro               8'h03
/cmd_pro_TB/en_din_pro            1'h0
/cmd_pro_TB/dout_pro              8'hxx
/cmd_pro_TB/en_dout_pro           1'h0
/cmd_pro_TB/data_reg              24'h060903
/cmd_pro_TB/state                 4'h2
/cmd_pro_TB/cmd_pro/reg_result    16'h0036

Now         2000017000 ps
Cursor 1    100200 ps
0 ps to 256360 ps
```

```
# ** Note: $stop    : C:/Users/ENDALE/Desktop/uart/cmd_pro.v(145)
#    Time: 2000017 ns  Iteration: 0  Instance: /cmd_pro_TB
# Break in Module cmd_pro_TB at C:/Users/ENDALE/Desktop/uart/cmd_pro.v line 145

VSIM 77>
```

```
Now: 2,000,017 ns  Delta: 0    sim:/cmd_pro_TB/#INITIAL#142
```

Fig 3.4.6 Multiplication operation of cmd_pro

3.5 Design and Simulation of Top Level UART

The top-level module is a module which embraces all the other three modules, UART_RX, UART_TX and cmd_pro modules. It is designed to realize the functionality of the whole design. This module is not instantiated with any of the other three modules. It rather instantiates all the other three.

Design modules normally are instantiated within the top level testbench modules so that simulation can be run by providing input stimulus. However, the testbench is not instantiated within any other modules because it is a block that encapsulates everything else and hence the top-level module. Thus, the top-level module can be designed in a way that the other three modules instantiated in it.

Because the top-level UART design is just instantiation of other predesigned modules, there is no need of state machine design and other complex design. It is the simplest module from all the rest.

3.5.1 Verilog Code Design of UART_TOP Module

The complete module has been designed as follow;

```
//20210510_FPGA Design of Instruction controller


`timescale 1ns/10ps

module UART_TOP (
                                        clk,
                                        res,
                                        RX,
                                        TX,
                                        led
                                        );

    input                                         clk;
    input                                         res;
    input                                         RX;
    output                                  TX;
    output[15:0]            led;

    wire[7:0]                               din_pro;
    wire                                en_din_pro;
    wire[7:0]                               dout_pro;
    wire                                en_dout_pro;
    wire                                    rdy;
    assign                                        led[7:0]=din_pro[7:0];

  UART_RX    UART_RX(
                                              .clk(clk),
                                              .res(res),
                                              .RX(RX),
                                               .data_out(din_pro)
                                               .en_data_out(en_din_pro),
                                              .led(led[15:8])
                                              );
```

```
UART_TX     UART_TX(
                                                .clk(clk),
                                                .res(res),
                                                .data_in(dout_pro),
                                                 .en_data_in(en_dout_pro),
                                                .TX(TX),
                                                .rdy(rdy)
                                                );
    cmd_pro      cmd_pro (
                                              .clk(clk),
                                               .res(res),
                                               .din_pro(din_pro),
                                                 .en_din_pro(en_din_pro),
                                               .dout_pro(dout_pro),
                                               .en_dout_pro(en_dout_pro),
                                               .rdy(rdy)
                                                );



    endmodule
```

This module is just need to have clk, res. RX and TX as input output port in the port list declaration. The other port led is added additionally just to check the output display on the FPGA board during FPGA testing. RX should be defined as input and TX should be defined as output port. din_pro and en_din_pro have to defined as wire, because they are the ports where the cmd_pro module gets its input through and at the same time they are the output from the UART_RX module. dout_pro and en_dout_pro also need to be defined as a wire because they are the ports where they UART_TX nodule gets its inputs though and at the same time they are the output signal from the cmd_pro module. The rdy signal is a wire that will notify the cmd_pro module whether the UART_TX is busy or not busy. The next step in designing UART_TOP module is to instantiate UART_RX port lists, then instantiate UART_TX port lists and then instantiate cmd_pro port lists. After that everything is done with UART_TOP module and the module will be ended with endmodule keyword.

3.5.2 Simulation of UART_TOP Module

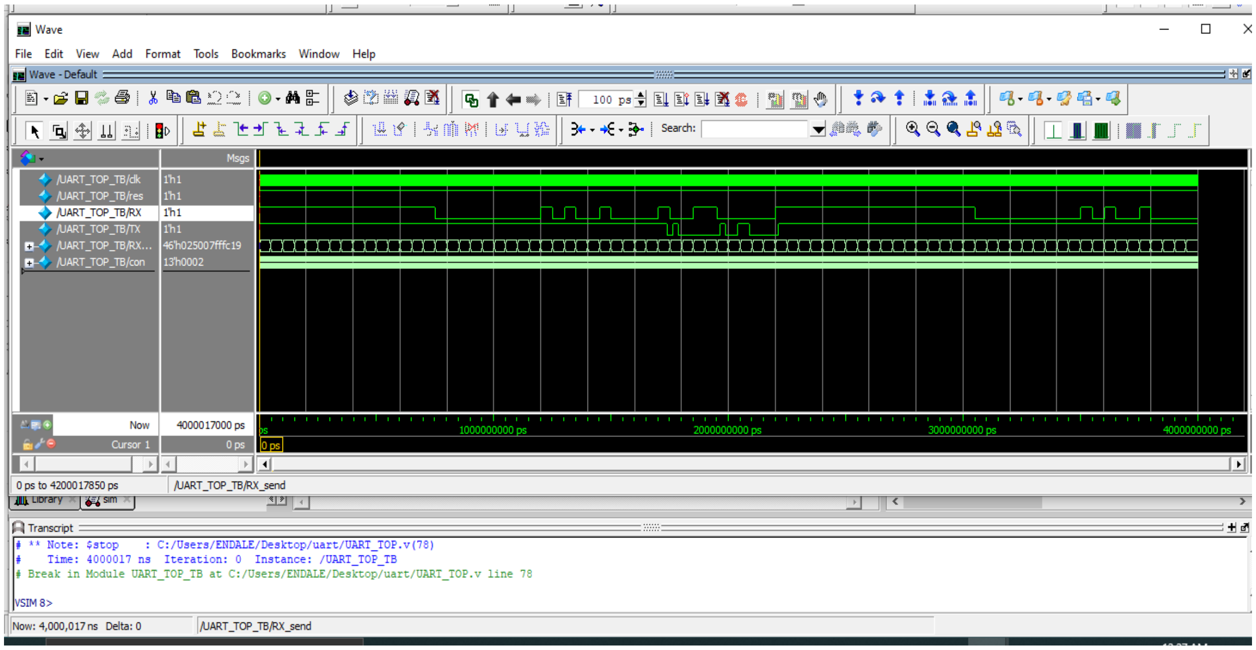
UART_TOP module can easily be verified its functionality by designing a stimulus testbench code. This test bench code is almost the same with that of UART_RX testbench module except some changes. Here, it is necessary to make sure the defined register variable have enough array size as the modules need three bytes hexadecimal number and the instantiation step should instantiate UART_TOP module itself.

The testbench module of UART_TOP module has been designed as follow;

```
/------testbench of UART_TOP-----
module UART_TOP_TB;
reg                                   clk,res;
wire                                  RX;
wire                                  TX;

reg[45:0]                             RX_send;
assign                            RX=RX_send[0];
reg[12:0]                         con;
UART_TOP   UART_TOP(
                                          clk,
                                          res,
                                          RX,
                                          TX
                                          );
initial begin
clk<=0;res<=0;RX_send<={1'b1,8'h06,1'b0,1'b1,8'h09,1'b0,1'b1,8'h00,1'b0,16'hffff};con<=0;
          #17                     res<=1;
          #4000000                $stop;
end
always #5 clk<=~clk;
always@(posedge clk)  begin
   if(con==2500-1) begin
```

```
            con<=0;
        end
        else begin
            con<=con+1;
        end

        if(con==0) begin
            RX_send[44:0]<=RX_send[45:1];
            RX_send[45]<=RX_send[0];
        end
    end
    endmodule
```

Here, the RX_send register has been defined as a 46-bit sized register so that it can have enough space to temporarily store all bit streams assigned to it. The RX port is assigned to this register so that the receiver module will be able to receive bytes. In the always block, the system checks the baud rate counter con and if con reaches 2500-1 condition, they system set con to 0 otherwise increment con. And if con is equal to 0, the data bit in RX_send will be shifted to the right in order to convert to parallel data type. Inn this way the test bench code has been designed and simulated. The result of simulation can be seen in the figure below.



```
/UART_TOP_TB/clk      1'h1
/UART_TOP_TB/res      1'h1
/UART_TOP_TB/RX       1'h1
/UART_TOP_TB/TX       1'h1
/UART_TOP_TB/RX...    46'h025007fffc19
/UART_TOP_TB/con      13'h0002

Now        4000017000 ps
Cursor 1   0 ps

0 ps to 4200017850 ps    /UART_TOP_TB/RX_send
```

```
# ** Note: $stop    : C:/Users/ENDALE/Desktop/uart/UART_TOP.v(78)
#    Time: 4000017 ns  Iteration: 0  Instance: /UART_TOP_TB
# Break in Module UART_TOP_TB at C:/Users/ENDALE/Desktop/uart/UART_TOP.v line 78

VSIM 8>
```

```
Now: 4,000,017 ns  Delta: 0    /UART_TOP_TB/RX_send
```

Figure 3.5.1 Simulation of UART_TOP module

3.6 Chapter summary

This chapter covers the main project work in this research paper. As the main goal of the project is to design a UART instruction controller module using Verilog HDL coding, this chapter has briefly addressed the design and simulation of each modules one by one. The function and the system architecture of UART instruction controller have also been addressed clearly at the begging of the chapter.

Chapter four: FPGA Testing

4.1 Introduction of FPGA Testing Board

The emergence of field programmable gate array (FPGA) is the result of the development of very large-scale integrated circuit (VISI) technology and computer aided design (CAD) technology. FPGA has the advantages of high integration, small size and special application through user programming. It allows circuit designers to use computer-based development platform, through design input, simulation, testing and verification, until the desired effect is achieved. Using FPGA can greatly shorten the development cycle of the system and reduce capital investment. What's more attractive is that the original circuit board level products can be integrated into chip level products by using FPGA devices, so as to reduce power consumption and improve reliability. At the same time, the design can be easily modified online. FPGA has become an ideal device for research and development, especially suitable for product prototype development and small batch production, so people also call FPGA programmable ASIC.

Altera cyclone series FPGA was launched by Altera company in September 2003, based on 1.5V, 0.13 μm processor. Altera Cyclone is a cost-effective FPGA family. Cyclone device series has the following characteristics features:

- ✓ 2910 - 20060 LE
- ✓ Up to 294,912 RAM bits (36864 bytes)
- ✓ Support configuration with low-cost serial configuration device
- ✓ Supports LVTTL, LVCMOS, SSTL-2 and SSTL-3 I / O standards
- ✓ Supports 66 - and 33 MHz 64 bit and 32-bit PCI standards

- ✓ High speed (640 Mbps) LVDS I / O support
- ✓ Low speed (311 Mbps) LVDS I / O support
- ✓ 311-MbpsRSDS I / O support
- ✓ Up to two PLLs per device provide clock frequency multiplication and phase modulation shift
- ✓ Up to eight global clock lines and six clock resources are available for Logical array module (Lab) row
- ✓ Supports external memory, including DDR SDRAM (133 MHz), FCRAM, and single data rate (SDR) SDRAM
- ✓ Supports a variety of IP cores, including features and Altera macro partners） Macro function

EP1C12Q240C8 is a member of the cyclone series. It has 20060 logic units and 288k bit RAM. There is such a phase locked loop (PLL) for the clock and a dedicated double data rate (DDR) interface to meet the requirements of DDR SDRAM and FCRAM memory requirements Point. This Cyclone devices provide cost-effective solutions for data channel applications Plan. This Cyclone devices support a variety of I / O standards, including LVDS data rates up to 640 megabits per second (Mbps) and 66 - and 33 MHz 64 bit and 32-bit peripheral component interconnect (PCI) used to interface and support ASSP and ASIC devices. Altera Cyclone also offers new low-cost serial configuration devices with cyclone devices

This design uses Altera cyclone series chip, the chip model is EPlC3T144C8, because the chip is a low price, high-capacity FPGA, which is widely used in practical applications with low price, excellent characteristics and rich on-chip resources. These are incomparable with other similar products.

EPlC3T144C8 chip adopts 1.5V core voltage, 0.33μm SRAM processor. Compared with other similar products, it has the following characteristics:

(1) The number of logic units (LE) is 12060.

(2) There are 173 available I / O pins, the I / O output can be adjusted according to the needs, and has the functions of swing rate control, three state buffer, bus hold, etc. The I / O pin of the whole device is divided into four zones, each zone can independently use different input voltage, and can provide I / O output of different voltage levels.

(3) Multi voltage interface, support LVTTL， LVCMOS， LVDS and other I / 0 standards.

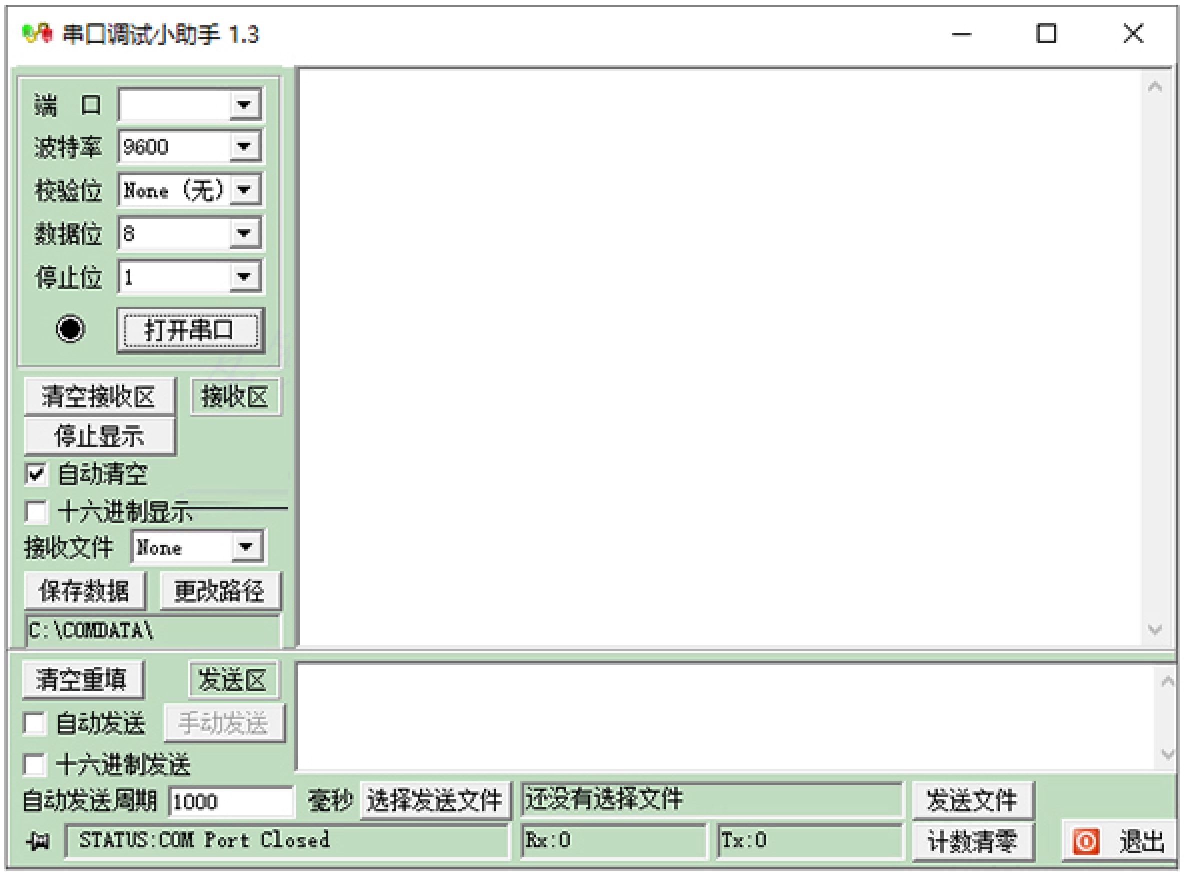
(4) Flexible clock management, with a phase lock loop (PLL) circuit in the chip, which can provide 1-32 times or frequency division of the input clock, 156-417ps phase shift and variable duty cycle clock output. The characteristics of the output clock can be directly set in the development software Quartos II. The clock signal output by PLL can be used as internal global clock, and can also be output to other circuits.

(5) There is a SignalTap embedded logic analyzer inside, which greatly facilitates the designer to check the internal logic of the chip without outputting the internal signal to the I / O pin.

4.2 The UartAssist (serial debugging assistant)

A Serial debugging assistant also called UartAssist is a debugging tool for modifying serial port data. It supports modifying serial port baud rate, check bit, data bit and stop bit, setting receiving area and sending area and increase the automatic identification function of the serial port, so that the restriction of serial port number is no longer bothered. The user-defined baud rate function of the software can change the list box from read-only to writable. we can input the desired baud rate directly without manual setting, and the list box will return to the read-only state when you select other preset baud rate. In addition, the internal files of the software are stored in binary system, which makes the storage speed of the software is much faster than that of similar software. It is worth mentioning that the software also supports the output of shortcut keys in the editing area, which makes serial editing as simple as word.

The figure below shows the UartAssist during run time.



端 口
波特率 9600
校验位 None (无)
数据位 8
停止位 1
☑ 自动清空
☐ 十六进制显示
接收文件 None
C:\COMDATA\
☐ 自动发送
☐ 十六进制发送
自动发送周期 1000 毫秒
还没有选择文件
STATUS:COM Port Closed
Rx:0
Tx:0

Fig 4.2.1 UartAssist

Function introduction:

1. Custom baud rate. Select "custom" in the baud rate list, the list box will be changed from read-only to writable, and directly input the baud rate wanted; When other specified baud rates are selected, the list box will return to read-only state.

2. Automatic acquisition of serial port list

3. Send edit area shortcut key: Alt + a send all contents, Alt + s sends data of cursor line, sends the data of the line where the cursor is located, and sends carriage return for line feed, Alt + C clear send area.

4. Display of Chinese characters in receiving area

5. The received hexadecimal data is saved in binary format for receiving files

Update log:

1. The parameters of serial port can be modified dynamically

2. Using thread to send files

3. The loop queue is used to receive data, and the thread is used to display data to avoid the program not responding when receiving a large amount of data

4. Fixed that version 1.2 could not open the serial port above com10

Usage method:

How to use serial debugging assistant

1. Connection: connect the ch340 to the USB interface of the computer. This is a ch340 module you bought. You can get more than ten pieces online.

2. Check: find the "device manager" of the computer: open the start menu and find the "computer";

3. Check: find the "device manager" of the computer: right click and select "device manager" in the drop-down menu.

4.3 System Testing

The Verilog HDL codes which have been designed in the previous chapter can now be tested on the Altera cyclone using the UartAssist software. In the previous chapter, each module was simulated on the ModelSim simulator separately and have been realized by designing a stimulus testbench module for each module. Here, the top-level module (UART_TOP) is used for testing on the development board. The top-level module needs to instantiate all the other three module and also needs to cascade each module within it. This can be seen in the following code. //20210513_FPGA

```
//final FPGA

module UART_TOP (

                                        clk,
                                        res,
                                        RX,
                                        TX,
                                        led
                                        );
input                                   clk;
input                                   res;
input                                   RX;
output                             TX;
output[15:0]                      led;
wire[7:0]                               din_pro;
wire                                 en_din_pro;
wire[7:0]                               dout_pro;
wire                                 en_dout_pro;
wire                                    rdy;
wire                                    reset;
assign                                 reset=!res;
assign                                  led[7:0]=din_pro[7:0];

cmd_pro     cmd_pro (
                                  .clk(clk),
                                  .res(reset),
                                   .din_pro(din_pro),
                                   .en_din_pro(en_din_pro),
                                   .dout_pro(dout_pro),
                                   .en_dout_pro(en_dout_pro),
                                    .rdy(rdy)
                                      );
```

```
UART_RX   UART_RX(
                                                  .clk(clk),
                                                  .res(reset),
                                                  .RX(RX),
                                                  .data_out(din_pro),
                                                  .en_data_out(en_din_pro),
                                                        .led(led[15:8])
                                                        );
UART_TX    UART_TX(
                                                  .clk(clk),
                                                  .res(reset),
                                                  .data_in(dout_pro),
                                                  .en_data_in(en_dout_pro),
                                                  .TX(TX),
                                                  .rdy(rdy)
                                                  );

endmodule

module cmd_pro(
                                                  clk,
                                                  res,
                                                  din_pro,
                                                  en_din_pro,
                                                  dout_pro,
                                                  en_dout_pro,
                                                  rdy
                                                  );
input                                     clk;
input                                   res;
input[7:0]                              din_pro;//the byte received from the RXer;
input                                   en_din_pro;//inform the com_pro to receive the data_in;
output[7:0]                     dout_pro;//the instruction result;
output                                  en_dout_pro;//enable the sending of the result;
input                                   rdy;//0,when ready, 1, when busy;
reg                                             en_dout_pro;
reg[7:0]                                dout_pro;
reg[2:0]                                state_machine;//the state machine;
reg[7:0]                                command,a,b;
reg[15:0]                               reg_result;//the buffer of the result;
/* command ISA defination
                        command     result
                        8'h00          a+b
        8'h01           ~a
        8'h02            a^b
        8'h03             a*b
*/

always@(posedge clk or negedge res)
if(~res) begin
        state<=0;command<=0;a<=0;b<=0;reg_result<=0;en_dout_pro<=0;
end
else begin
        case(state)
        0://waiting for the first byte(command) from the PC;
        begin
                if(en_din_pro) begin
                        command<=din_pro;
                        reg_result<=0;
                        state<=1;
                end
        end
        1://waiting for the second byte(parameter a) from the PC;
        begin
                if(en_din_pro) begin
                        a<=din_pro;
                        state<=2;
                end
        end
        2://waiting for the third byte(parameter b) from the PC;
        begin
                if(en_din_pro) begin
                        b<=din_pro;
                        state<=3;
                end
        end
        3://process the command to get the result;
        begin
                case(command)
                8'h00://a+b
                begin
                        reg_result<=a+b;
                end
                8'h01://~a
                begin
```

```
                reg_result<={8'h00,~a};
        end
        8'h02://a^b
        begin
                reg_result<={8'h00,a^b};
        end
        8'h03://a*b
        begin
                reg_result<=a*b;
        end
        default://do nothing
        begin
                reg_result<=0;
        end
        endcase
        state<=4;
end
4://sending the result, the LSB byte;
begin
        if(~rdy) begin
                dout_pro<=reg_result[15:8];
                en_dout_pro<=1;
                state<=5;
        end
end
5://setting en_send back to 0;
begin
        en_dout_pro<=0;
```

```
                state<=6;
        end
        6://sending the result, the MSB byte;
        begin
                if(~rdy) begin
                        dout_pro<=reg_result[7:0];
                        en_dout_pro<=1;
                        state<=7;
                end
        end
        7://setting en_send back to 0;
        begin
                en_dout_pro<=0;
                state<=0;
        end
        endcase
end
endmodule

module UART_RX(
                                                                                        clk,
                                                                                        res,
                                                                                        RX,
                                                                                        data_out,
                                                                                        en_data_out,
                                                                                        led
                                                                                        );
input                                                                           clk;
input                                         res;
input                                         RX;//Received data
output[7:0]                             data_out;//8bits data output
output                                    en_data_out;//Output    data    enable
signal
output[7:0]                       led;
reg[7:0]                          led;

parameter                        freg_clk=24;//Clock frequency
parameter                        baud_rate=9600;//Baud rate
parameter                        bit_width=freg_clk*1000000/baud_rate;//Bit rate

reg[7:0]                                  data_out;
reg[7:0]                                  state;//Master state machine
reg[14:0]                                 con;//Counter
reg[3:0]                                  con_bits;//Data bit counter

reg                                           RX_delay;//RX delay 1 clock
reg                                           en_data_out;
always@(posedge clk or negedge res)
if(~res) begin
        state<=0;con<=0;con_bits<=0;RX_delay<=0;
        data_out<=0;en_data_out<=0;led<=0;
end
else begin
        RX_delay<=RX;
        case(state)
        0://Waiting for free time
        begin
        if(con==bit_width-1) begin
                con<=0;
        end
        else begin
                con<=con+1;
        end
        if(con==0) begin
                if(RX) begin
                        con_bits<=con_bits+1;
                end
                else begin
                        con_bits<=0;
                end
        end
        if(con_bits==12) begin
                state<=1;
        end
end
1://Equal starting position
begin
        en_data_out<=0;
        if((~RX)&RX_delay) begin
                state<=2;
        end
end
2://Receive data_ out[0];
begin
        if(con==bit_width-1) begin
```

```
                con<=0;
                data_out[0]<=RX;
                state<=3;
        end
        else begin
                con<=con+1;
        end
end
3://Receive data_ out[1];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[1]<=RX;
                state<=4;
        end
        else begin
                con<=con+1;
        end
end
4://Receive data_ out[2];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[2]<=RX;
                state<=5;
        end
        else begin
                con<=con+1;
        end
end
5://Receive data_ out[3];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[3]<=RX;
                state<=6;
        end
        else begin
                con<=con+1;
        end
end
6://Receive data_ out[4];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[4]<=RX;
                state<=7;
        end
        else begin
                con<=con+1;
        end
end
7://Receive data_ out[5];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[5]<=RX;
                state<=8;
        end
        else begin
                con<=con+1;
        end
end
8://Receive data_ out[6];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[6]<=RX;
                state<=9;
        end
        else begin
                con<=con+1;
        end
end
9://Receive data_ out[7];
begin
        if(con==bit_width-1) begin
                con<=0;
                data_out[7]<=RX;
                state<=10;
        end
        else begin
                con<=con+1;
        end
        end
        10://Sending data enable signal;
        begin
                en_data_out<=1;
                led<=data_out;
                state<=1;
        end
        default://
        begin
                state<=0;
                con<=0;
                con_bits<=0;
                en_data_out<=0;
        end
        endcase
end
endmodule

module UART_TX(
                                                                clk,
                                                                res,
                                                               data_in,

        en_data_in,
                                                                TX,
                                                                rdy
                                                                );

input                                                   clk;
input                                   res;

input[7:0]                          data_in;//Bytes to be sent;

input                                   en_data_in;//Send
enable signal;

output                               TX;

output                               rdy;//higher bit is busy;

parameter                            baud_rate=9600;//Hz;

parameter                            frequency_clk=24;//MHz;


reg                                   state;//Main state machine;

reg                     TX;

reg                                     rdy;

reg[9:0]                         data_buf;//Cache the input byte
data;

reg                                   bit_pulse;

reg[31:0]                          con;//For counting;

reg[3:0]                            con_bit_send;//The number
of bits sent;

//assign                                TX=data_buf[0];//TX
is data_ The lowest position of buf;

always@(posedge clk or negedge res) begin

if(~res) begin

        state<=0;rdy<=0;data_buf<=10'h0;bit_pulse<=0;con<=0;con_bit_send<=0;TX<=1;

end

else begin

        case(state)

        0://Wait for the enable signal to be sent;

        begin

                con_bit_send<=0;

                if(en_data_in) begin //Wait until the transmission is enabled;
```

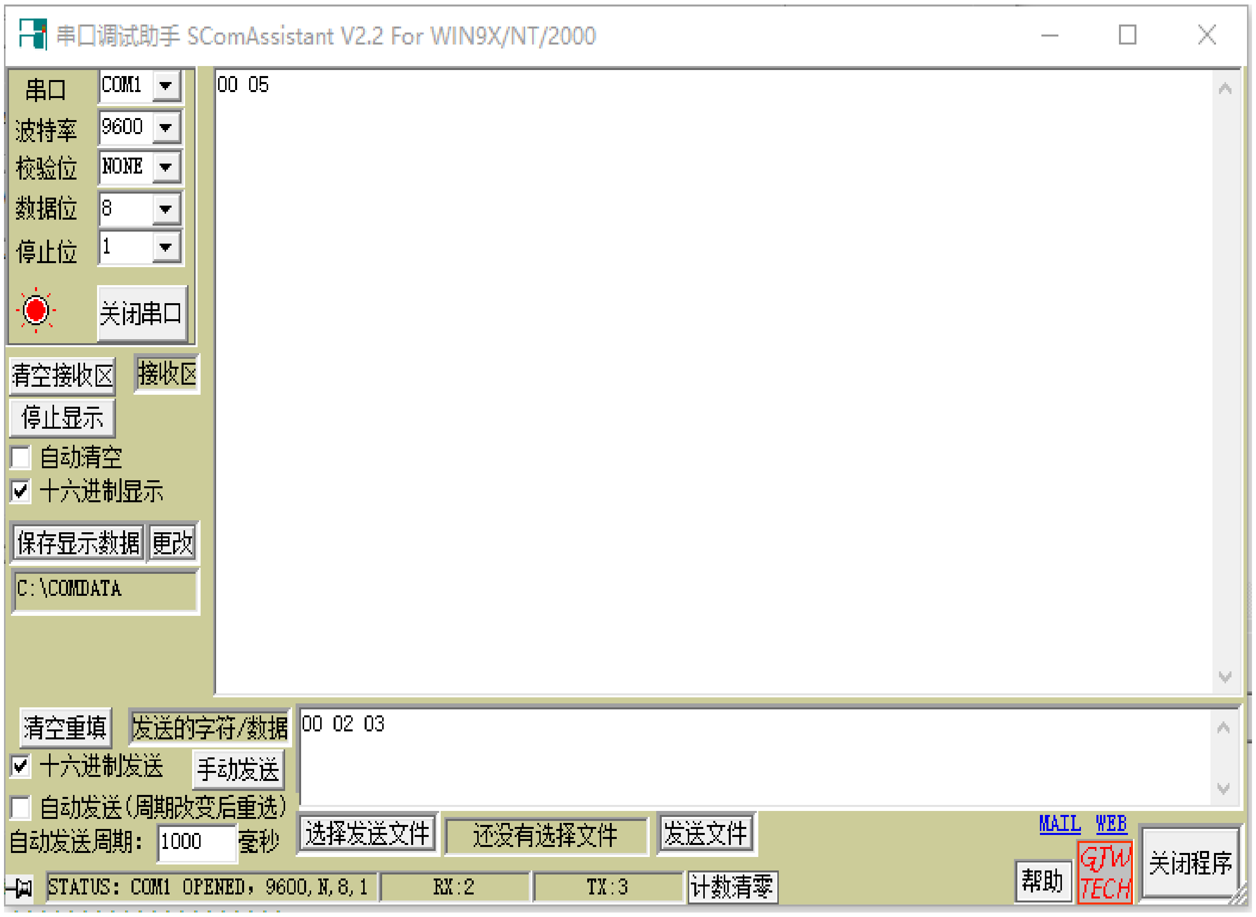
```
                    data_buf<={1'b1,data_in,1'b0};//The highest end bit, the lowest start bit,
and the middle 8 bits are data;
                    rdy<=1;//busy;
                    state<=1;
                end
                else begin
                    rdy<=0;//Not busy;
                end
            end
            1://Sending data;
            begin
                if(con==(frequency_clk*1000000)/baud_rate) begin //Generate bit_ Pulse;
                    bit_pulse<=1;
                    con<=0;
                end
                else begin
                    bit_pulse<=0;
                    con<=con+1;
                end
                if(bit_pulse) begin //data_ BUF moves to the right;
                    TX<=data_buf[0];
                    data_buf[8:0]<=data_buf[9:1];
                    con_bit_send<=con_bit_send+1;//Record the number of bits sent;
                end
                if(con_bit_send==10) begin //Sending completed;
                    state<=0;rdy<=0;//Not busy;
                end
            end
            endcase
```

```
end

end

endmodule
```

The top-level module here is to be tested on the FPGA development board So, it doesn’t need testbench code as in the simulation. After connecting the computer and the Altera cyclone FPGA, we can run the UartAssist software and input the three bytes as command, parameter a and parameter b from the computer keyboard with the help of UartAssist. The receiver module will receive each of those byte and hand to the instruction execution module and the instruction execution module perform its tasks and hand in the result of execution to the transmitter module. The above code has been tested and the following result has been achieved.

The commands expected to be executed in the command execution module are the for operation such that the addition, a+b, the negation, ~a, the logic XOR, a^b, and the multiplication, a*b. The command register has to be loaded with 00 to perform a+b, 01 to perform ~a, 02 to perform a^b, and 03 to perform a*b. hence, we need to input each of those four hexadecimal bytes one by one separately with parameter a and b.

Thus, the first command to be executed can be the addition and the command has to get 00 byte with a and b. This can be inputted as “00 02 03” as it can be seen in the figure bellow. The result of execution must give 02 + 03 = 05 as 00 05, where 00 represents the type of operation done and 05 is the result. This has been proved as it can be seen bellow.



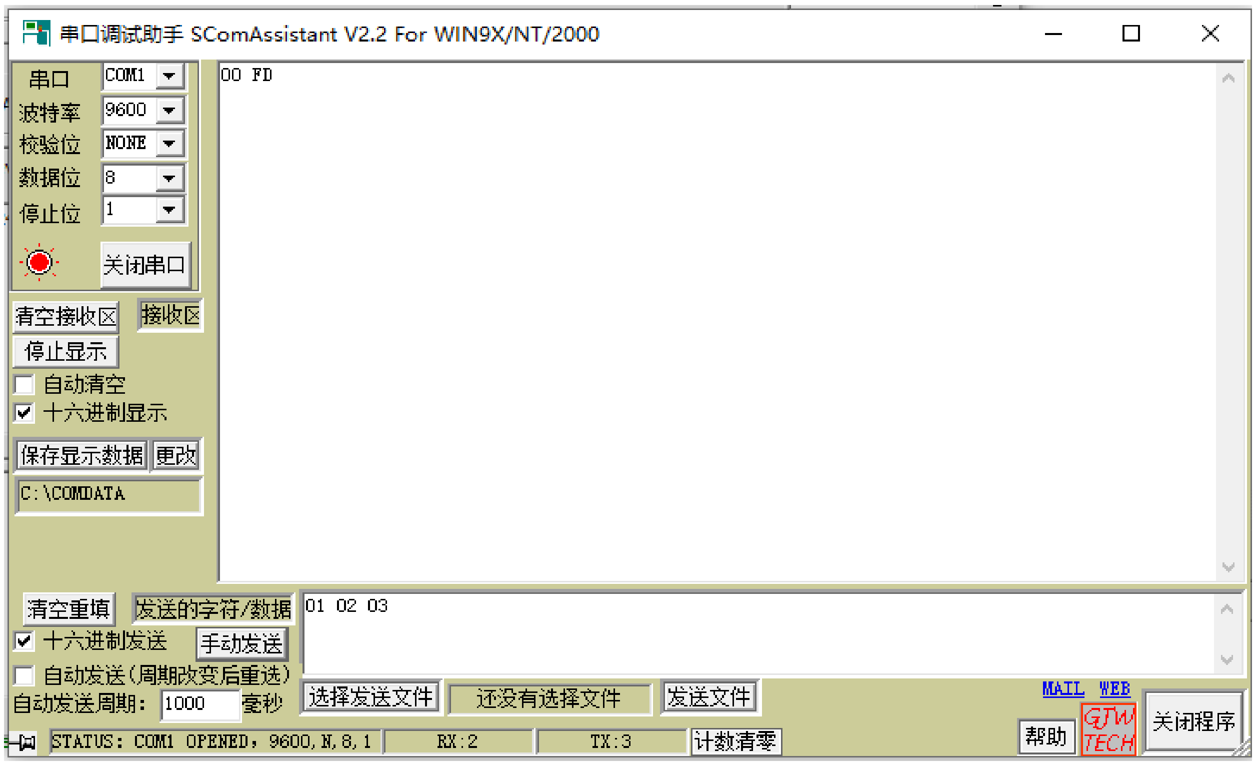
```
00 05
```

```
00 02 03
```

STATUS: COM1 OPENED, 9600, N, 8, 1   RX:2   TX:3

Fig 4.3.2 Addition operation

The next operation to be performed can be the negation, ~a. To perform this, the command has to get byte 01. As the negation operation is done only parameter a, parameter b can be set as 03 or any value as it won’t affect the operation. Thus, we can input the three bytes as “01 02 03) and the result must be {~ (02), which is ~ (00000010) =11111101 =FD} 01 FD. This has been proved as shown in the figure bellow;


串口调试助手 SComAssistant V2.2 For WIN9X/NT/2000

串口 COM1
波特率 9600
校验位 NONE
数据位 8
停止位 1
C:\COMDATA

```
00 FD
```

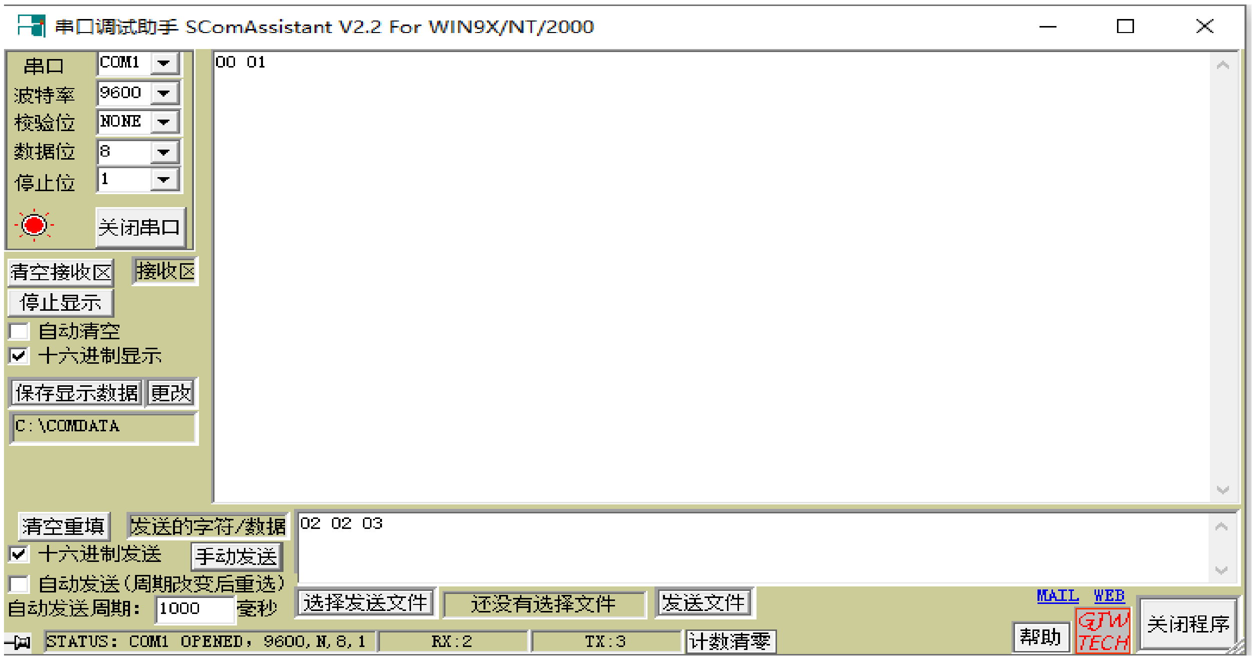
```
01 02 03
```

自动发送周期: 1000 毫秒

```
STATUS: COM1 OPENED, 9600, N, 8, 1    RX:2    TX:3
```

Fig 4.3.3 Negation Operation

The next operation would be logic XOR. For this the first byte should be 02 and a and b can be 02 & 03. This can be inputted as “02 02 03“the result is {(00000010 xor 00000011) =00000001} 00 01. This has been verified in the figure bellow.


串口调试助手 SComAssistant V2.2 For WIN9X/NT/2000

串口 COM1
波特率 9600
校验位 NONE
数据位 8
停止位 1
C:\COMDATA

```
00 01
```

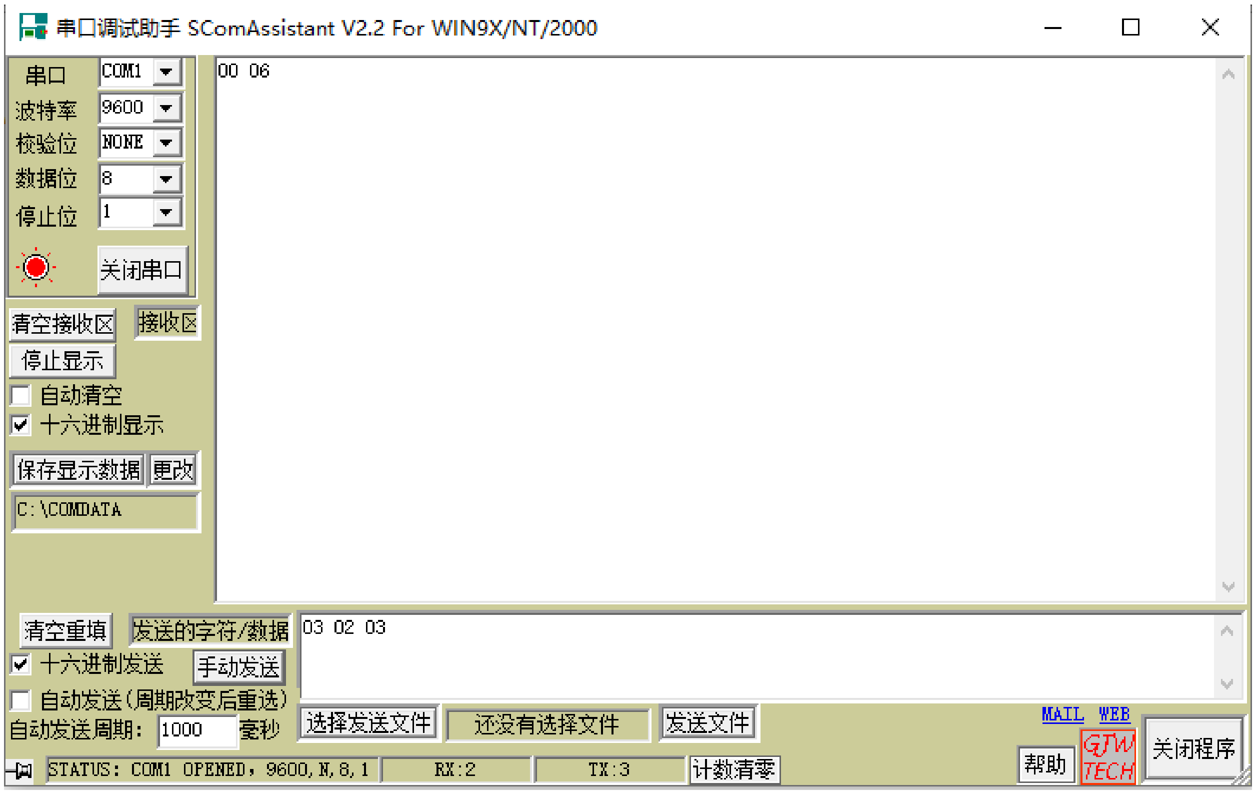
```
02 02 03
```

自动发送周期: 1000 毫秒

```
STATUS: COM1 OPENED, 9600, N, 8, 1    RX:2    TX:3
```

Fig 4.3.4 Logic XOR Operation

The last operation of the design is multiplication. To perform multiplication, the command has to get byte 03. Hence, we can input the three bytes as “03 02 03” and the result of execution must be {(02*03) = (06) = (00000010) * (00000011) = (00000110)} 00 06. This has been verified down in the figure.



串口 COM1
波特率 9600
校验位 NONE
数据位 8
停止位 1

```
00 06
```

[ ] 自动清空
[✓] 十六进制显示

C:\COMDATA

```
03 02 03
```

[✓] 十六进制发送
[ ] 自动发送(周期改变后重选)
自动发送周期: 1000 毫秒

STATUS: COM1 OPENED, 9600, N, 8, 1   RX:2   TX:3

Fig 4.3.5 Multiplication Operation

3.6 Summary of Chapter

The main objectives of this chapter were to introduce the FPGA testing development board, which is Altera Cyclone, the serial debugging assistant software, which is UartAssist and finally to test the designed UART instruction controller. Hence, the chapter has briefly introduced the Altera Cyclone FPGA and also the serial debugging software. And the designed UART instruction controller has been tested successfully and the result of executions have been seen all correct and verified.

Chapter five: Conclusion and Preference

5.1 Conclusion

The design FPGA Design of UART Instruction is useful in simplifying the complex instruction set required digital chip development significantly. Therefore, this design has been done based on the core understanding of Verilog HDL coding and FPGA design flow so that the complex instruction set are not necessarily required in the chip development. Chapter two of this paper has briefly introduced the fundamental understanding of coding in Verilog HDL and has also briefly introduced FPGAs and their design flow schemes.

The instruction set architecture of this design is designed in a way that the system would able to perform four, addition, negation, XOR and multiplication operations by receiving input data from the computer and return the result of execution back to the computer. Thus, the complete task has been done by designing three Verilog HDL modules namely; the receiver module (UART_RX), the transmitter module (UART_TX) and the command execution module (cmd_pro). The functionality of each of those modules has been verified by designing a stimulus testbench code for each module and has been simulated on the ModelSim simulator software. The result of the simulation of all modules has been seen in chapter three and has been verified that each of the designed module are working correctly.

Using FPGAs to implement UART designs can highly reduce the area of the design system. It can also reduce the power consumption of the system, improve the stability of the design, and make full use of the advantages of the FPGA technologies. This method of hardware and software designs have become an important part of FPGA as FPGA becomes leading trend in the field of electronic design. In many cases, the UART instruction controller can completely replace the special IC chip. Hence, the design has been completed and downloaded to Altera cyclone series EP1C12Q240C8 and has been tested and verified that the design works perfectly.

Remarks: This design has introduced only the four mathematical operations namely, addition, negation, logic XOR and multiplication. However, the design can be expanded and can be designed in a way that other complex set of instructions can be included in the design in future work.

5.2 Preference

As an author of this paper and as a designer of this project, I have learned a lot of fundamental knowledge and new skills. I also have experienced and faced many problems and obstacles as well. In the design process of this project, I have encountered many problems, such as some incomprehensible syntax problems at the beginning of programming. For example, in division operations such as a < = B / 2, the data bit width of a and B is required to be the same, otherwise the operation is difficult to establish. It took me a lot of time to solve these problems, which reflects that my understanding of Verilog grammar is still not enough. At the software level, I still can't skillfully use many functions of Quartus, such as how to establish a direct connection between Modelsim and Quartus engineering, and how to write testbench to simulate in Modelsim. Another example is to use SignalTap to capture some signals of the development board when it is running. Skilled use of SignalTap can be very convenient to verify the timing of their own projects and remove faults; These functions are very powerful. I haven't mastered the software functions, which shows that I still need to learn more.

Language and software are just tools, and design idea is the most important core. Before learning and operating FPGA, I always thought that as long as I master the development language and software, I can be competent for the development work. However, in the actual operation process, I found that most of the logic functions can be realized by simple Verilog statements, and how to use language to realize a function, if there is a correct design idea, it can reduce the workload and improve the quality of development.

References

1. Zhe, H., J. ZHANG, and X.-l.J.C.J.o.A. LUO, *A novel design of efficient multi-channel UART controller based on FPGA.* 2007. **20**(1): p. 66-74.
2. Awedh, M. and A. Mueen, *Research Article Design and FPGA Implementation of UART Using Microprogrammed Controller.*
3. 彭圆圆 and 刘.J. 企业技术开发, *基于 FPGA 的 UART 模块设计及仿真*. 2010. **23**.
4. Mayank M. Bhanani 1, P.D.R.P., *FPGA Based Implementation of UART bus for*

AES application. IJARIIE-ISSN(O)-2395-4396, 2016. **2** (3).

5. 高军建 and 苗.J. 價值工程, *基于 FPGA 的 UART 模块化设计*. 2010(207): p. 148-149.
6. Kralev, J. *Design of UART controller for FPGA with Simulink®*. in *2014 3rd Mediterranean Conference on Embedded Computing (MECO)*. 2014. IEEE.
7. S, K., *Design and Implementation of UART using Verilog.* International Journal Of Engineering And Computer Science, 2015. **Volume – 4** (Issue - 12): p. 15240-15245.
8. Ünsalan, C. and B. Tar, *Digital system design with FPGA: implementation using Verilog and VHDL*. 2017: McGraw-Hill Education.
9. Priya, G.B., et al., *An Advanced Universal Asynchronous Receiver Transmitter (UART) Design & Implementation By Using VERILOG.* **3**.
10. Hora, J., et al., *Verilog HDL Implementation of a Universal Synchronous Asynchronous Receiver Transmitter.* 2014.
11. Gopal, P.B., K.H. Kishore, and B.P.J.I.J.o.A.E.R. Kittu, ISSN, *An FPGA Implementation of On Chip UART Testing with BIST Techniques.* 2015: p. 0973-4562.
12. Lee, Z.Y., *UART Design, Integration and Synthesis on FPGA*. 2016, UTAR.
13. Sowmya, K., S. Gomes, and V.R. Tadiparthi. *Design of UART Module using ASMD Technique*. in *2020 5th International Conference on Communication and Electronics Systems (ICCES)*. 2020. IEEE.