www.ijecs.in
International Journal Of Engineering And Computer Science
Volume1 Issue 1 Oct 2012 Page No. 28-36

Design Of High Speed Uart For Programming Fpga

Hazim Kamal Ansari, Asad Suhail Farooqi

M.Tech Scholar, AFSET, Dhauj, Fbd.

hazimkamal@gmail.com

asadsuhail2003@gmail.com

Abstract.

FPGA (Field Programmable gate Array) devices are one of the modern technologies that are changing the electronic industry. They are riding the same integrated circuit process curves as processors and memories and keep getting larger, faster, and cheaper and are now common in low and mid volume embedded products With the advances of FPGA technology, many engineers now have the opportunity to do medium scale digital design. They can be found in primary and secondary surveillance radar, satellite communication, automotive, manufacturing, and many other types of products. Time triggered communication within FPGA is enhanced with the help of UART (Universal Asynchronous Receiver Transmitter).It translates between serial and parallel bits of data. The device changes incoming parallel information to serial data which can be sent on a communication line. A second UART can be used to receive the information. The UART performs all the tasks, timing, parity checking, etc. needed for the communication. UART is designed in VHDL. The design process involves converting the requirements into a format that represents the desired digital function(s).

Key words: *FPGA, EMBEDDED, UART, VHDL.*

Introduction

Over the past several years, high capacity Field Programmable Devices (FPDs) have enjoyed a rapidly expanding market, and have become widely accepted for implementation of small to moderately large digital circuits. FPGA is one of the upcoming FPDs available in the market. FPGAs are used in low-volume products, where design costs comprise a significant part of the budget With FPGAs now exceeding the 10 million gate limit you can really dream big. FPGAs are now common in low and mid volume embedded products where they offer the following advantages:

- Fast time to market
- Better integration
- In system programmability
- FPGAs tend to have long life cycles and are usually replaced with pin compatible parts.

To maintain time triggered communication within FPGA a separate controller is designed within it. This controller is called UART. It is a kind of serial communication circuit which is used widely. A universal asynchronous receive/transmit (UART) is an integrated circuit which plays the most important role in serial communication. It handles the conversion between serial and parallel data. Serial communication reduces the distortion of a signal, therefore makes data transfer between two systems separated in great distance possible. It contains a parallel-to serial converter for data transmitted from the computer and a serial to parallel converter for data coming in via the serial line. The UART also has a buffer for temporarily storing data from high speed transmissions. In addition to the basic job of converting data from parallel to serial for transmission and from serial to parallel on reception, a UART will usually provide additional circuits for signals that can be used to indicate the state of the transmission media and to regulate the flow of data in the event that the remote device is not prepared to accept more data. UART must have a larger internal buffer to store data coming from the modem until the CPU has time to process it. The design entry of UART is done in VHDL.

I FPGA

In 1985, Xilinx introduced a completely new idea: combine the user control and time to market of PLDs with the densities and cost benefits of gate arrays. Customers liked it, and the FPGA was born. Today Xilinx is the number one FPGA vendor in the world. An FPGA is a regular structure of logic cells (or modules) and interconnect, which is under your complete control. This means that you can design, program, and make changes to your circuit whenever you wish. FPGAs are used in low-volume products, where design costs comprise a significant part of the budget With FPGAs now exceeding the 10 million gate limit you can really dream big. The most common FPGA architecture consists of an array of logic blocks (called Configurable Logic Block, CLB, or Logic Array Block, LAB, depending on vendor), I/O pads, and routing channels. As their size, capabilities, and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SoC). Particularly with the introduction of dedicated multipliers into FPGA architectures in the late 1990s, applications, which had traditionally been the sole reserve of DSPs, began to incorporate FPGAs instead. FPGAs especially find applications in any area or algorithm that can make use of the massive parallelism offered by their architecture.

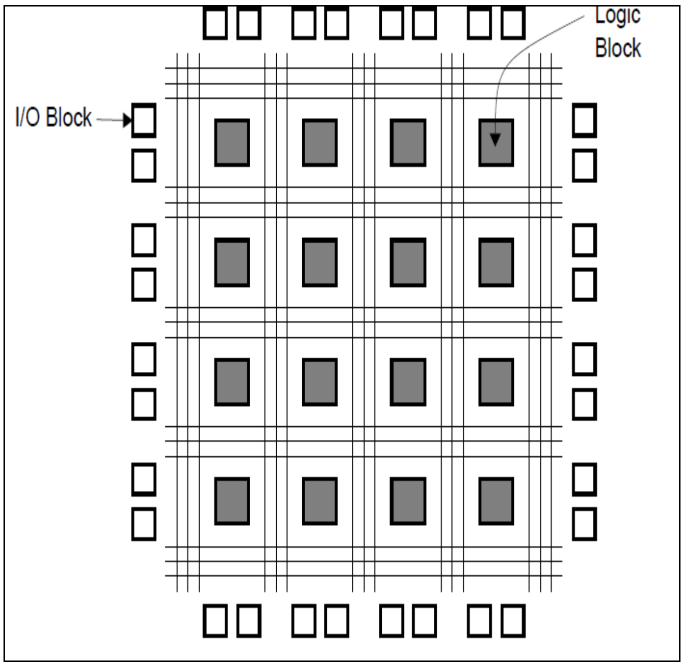


Fig. 1 Generalized Architecture of FPGA

II SYSTEM DESIGN ISSUES

A FPGA designer should understand several key points about the system the FPGA is going to be used in. Determine what the FPGA is going to do for the system and how it will interact with the rest of the system. The classic approach is to break a digital design down into data path and control logic. Generally the purpose of digital logic in a system is to move data. If the FPGA is involved in a data path, design this first. Figure out what data has to moved and how fast. Understand and document the data path requirements fully before designing the control architecture. Two important issues that are usually dictated by the system is the reset and clock strategy. Bandwidth of the data path and available system clocks will probably dictate the clock speed. Signals that connect two different asynchronous clock domains must be synchronized. Synchronization costs clock cycles and sometimes requires FIFOs to maintain data throughput. The reset strategy is a simple, but very critical aspect of any digital design. In any reset scenario, reset should be deasserted synchronously with the clock so that all flops exit reset on the same clock cycle. There are two basic scenarios. Reset scenario number one is where the system comes out of reset and then the clock starts running. This scenario can occur when using a clock output from a microcontroller to clock a FPGA. The system is taken out of reset and then the oscillator circuitry in the microcontroller starts running. An asynchronous reset is required in this case. Designers should be aware that some clock ICs output a series of runt pulses while the oscillator starts up. If the clock does not start up cleanly, the FPGA should be held in reset until the clock is stable. Reset scenario number two is where reset is odeasserted after the clock is running. In this case, it is necessary for reset to be synchronous to the clock so that all fops exit reset on the same clock. If the reset is asynchronous it is safest to synchronize reset inside the FPGA is circuit, the internal reset to the FPGA can be asynchronously set, but reset does not deactivate until a clock edge occurs. This allows the chip to be put in reset without the clock running, but forces the chip to come out of reset synchronously.

III FPGA DESIGN

Now that the system data path, control architecture, reset, and clock strategies are understood and documented, the actual FPGA design can start. The three primary considerations for selecting a FPGA are speed, pin count, and amount of logic. If there is time for 5-7 levels of combinational logic between flops, the design should be fairly straightforward. More levels of combinational logic between flops will require more careful design and more pipelining effort. There are numerous other considerations that are more device specific such as I/O buffer features, on chip PLLs, available tools and cores, on chip RAM, etc. It is a good idea to write a functional specification for the design. This forces the designer to understand what the chip is supposed to do before implementing large amounts logic and will lead to cleaner implementations. A functional specification is also very useful for the system designer who will eventually use the FPGA. The partitioning of the FPGA design falls out after the functional specification. Partitioning a design by functional blocks is a logical partitioning strategy, but it may also be necessary to partition parts of the design by routing strategy if there are performance critical parts of the design. A rough guideline that seems to work well is that the lowest level blocks in the design should be 2000 to 5000 gates in size. The top level module should be used only to connect lower level blocks. Avoid logic in the top level module if possible. The design entry and implementation is where the tools start to become important. VHDL and Verilog are industry standard HDLs (hardware description languages) for digital design entry and simulation. Text based code still seems to be the preferred way to express large, complex designs as evidenced by the fact the most people still write software in text based languages. The terms RTL (Register Transfer Level) and Structural HDL are often used to describe two different levels of HDL. The core of the design entry code is usually written at the RTL level but also contains structural HDL code to connect modules together. Structural HDL is essentially a netlist that describes how different components are tied together. A HDL file of a routed design created by place and route tools is a structural HDL level file. Behavioural HDL is a third level of HDL that is used to model the behaviour of a device and is used only during simulation. Behavioural HDL is generally not used for design entry as it is difficult to synthesize. Schematic entry is still used for FPGA design entry but is cumbersome for large designs and is more difficult to simulate and verify. The general flow for a HDL based design is shown in Illustration. There are additional steps such as timing simulation of the routed design and timing analysis at the synthesis level, but the majority of the time will be spent iterating the steps show in Illustration. Investing some time in setting up a good simulation environment is necessary for a large design and will save you time on any design. Some of the low end FPGA design tools do not include a source level HDL simulator which must be purchased separately. It is a good idea to synthesize each block as it is coded so that inefficient and slow designs can be fixed before the entire chip is synthesized. Registering the outputs of blocks will make meeting timing requirements easier but might require more FPGA resources. Timing and related tools can be somewhat of a mystery in FPGAs. For synchronous logic, you must verify that the inputs of flops arrive before the clock does. This can be controlled in a large part by carefully coding the design to minimize the levels of combinational logic between flops. After synthesizing some code and looking at the timing results, you quickly learn how to code for synthesis. Pipelining is a standard technique for speeding up a design and is done by breaking up large amounts of combinational logic into several stages with flops between the stages. Medium speed designs (30MHz +) will probably require some degree of pipelining. Timing can also be improved by entering timing constraints that the place and route tools try to meet. Timing constraints should be standard part of any FPGA design entry. Higher end synthesis tools can also use timing constraints. Static timing analysis is the primary timing verification tool that checks the worst case delays against the timing constraints entered by the designer. Timing simulations of routed designs are useful for debugging and sanity checks, but can be misleading because a timing simulation will most likely not exercise every possible combinational path. After the design is implemented in real hardware, the debug phase starts. FPGA designs are difficult to debug on the bench because internal signals are not visible unless routed to unused pins or analyzed by special FPGA debug equipment. This is where a good verification environment pays off. The design should be mostly correct if you did the functional simulations and static timing analysis. Chances are some corners where cut and some problems will come up in unverified parts of the design. Once the symptoms are identified, the designer can go back to the functional simulation, recreate the scenario and observe what is going on inside the chip. If timing problems are suspected, a timing simulation can be run on a structural model of the chip.

IV Need For Uart

In FPGAs to increase speed of operation parallel processing of data is required. At the FPGA a UART needs to be designed which converts serial bits into parallel bits of data. Serial communication is often used either to control or to receive data from an embedded microprocessor/programmable logic. A UART or Universal Asynchronous Receiver-Transmitter is a piece of computer hardware that translates between parallel bits of data and serial bits. A UART is usually an integrated circuit used for serial communications over a computer or peripheral device serial port. Serial communication is a form of I/O in which the bits of a byte begin transferred appear one after the other in a timed sequence on a single wire. An UART, universal asynchronous receiver / transmitter is responsible for performing the main task in serial communications with computers. The device changes incoming parallel information to serial data which can be sent on a communication line. A second UART can be used to receive the information. The UART performs all the tasks, timing, parity checking, etc. needed for the communication. The only extra devices attached are line driver chips capable of transforming the TTL level signals to line voltages and vice versa. To use the UART in different environments, registers are accessible to set or review the communication parameters. Settable parameters are for example the communication speed, the type of parity check, and the way incoming information is signalled to the running software .The UART controller is the key component of the serial communications subsystem of a computer. The UART takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes. Serial transmission of digital information (bits) through a single wire or other medium is much more cost effective than parallel transmission through multiple wires. A UART is used to convert the transmitted information between its sequential and parallel form at each end of the link. Each UART contains a shift register which is the fundamental method of conversion between serial and parallel forms. The UART usually does not directly generate or receive the external signals used between different items of equipment. Typically, separate interface devices are used to convert the logic level signals of the UART to and from the external signalling level.

V The Uart Modules

The UART module probably can be divided into two parts the transmit and receive parts as shown by figure 1:

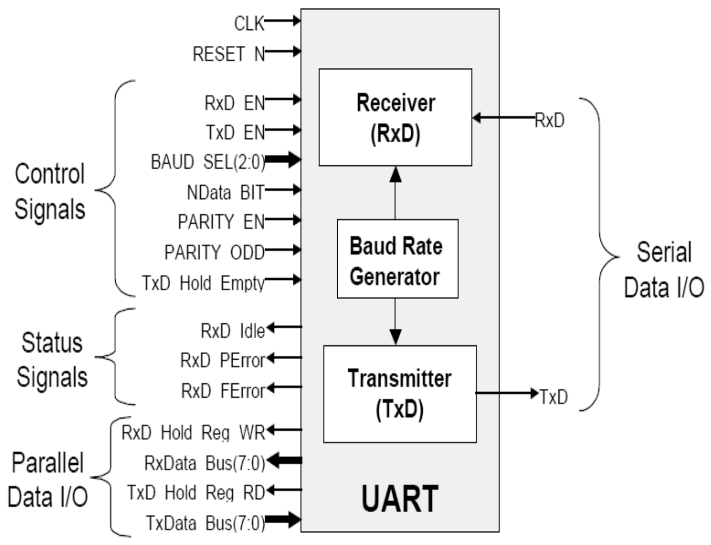


Fig. 2 Complete UART module

There is complete synchronization between the two and although they work independently. Working of UART is based on asynchronous transmitting and receiving. In asynchronous transmitting, teletype-style UARTs send a "start" bit, five to eight data bits, least-significant-bit first, an optional "parity" bit, and then one, one and a half, or two "stop" bits. The start bit is the opposite polarity of the data-line's idle state. The stop bit is the data-line's idle state, and provides a delay before the next character can start. (This is called asynchronous start-stop transmission). In mechanical teletypes, the "stop" bit was often stretched to two bit times to give the mechanism more time to finish printing a character. A stretched "stop" bit also helps resynchronization. The parity bit can either makes the number of "one" bits between any start/stop pair odd, or even, or it can be omitted. Odd parity is more reliable because it assures that there will always be at least one data transition, and this permits many UARTs to resynchronize. In synchronous transmission, the clock data is recovered separately from the data stream and no start/stop bits are used. This improves the efficiency of transmission on suitable channels since more of the bits sent are usable data and not character framing. An asynchronous transmission sends no characters over the interconnection when the transmitting device has nothing to send—only idle stop bits; but a synchronous interface must send "pad" characters to maintain synchronism between the receiver and transmitter. The usual filler is the ASCII "SYN" character. Asynchronous transmission allows data to be transmitted without the sender having to send a clock signal to the receiver. Instead, the sender and receiver must agree on timing parameters in advance and special bits are added to each word which is used to synchronize the sending and receiving units. When a word is given to the UART for Asynchronous transmissions, a bit called the "Start Bit" is added to the beginning of each word that is to be transmitted. The Start Bit is used to alert the receiver that a word of data is about to be sent, and to force the clock in the receiver into synchronization with the clock in the transmitter. These two clocks must be accurate enough to not have the frequency drift by more than 10% during the transmission of the remaining bits in the word. (This requirement was set in the days of mechanical teleprinters and is easily met by modern electronic equipment.) After the Start Bit, the individual bits of the word of data are sent, with the Least Significant Bit (LSB) being sent first. Each bit in the transmission is transmitted for exactly the same amount of time as all of the other bits, and the receiver ―looks‖ at the wire at approximately halfway through the period assigned to each bit to determine if the bit is a 1 or a 0. For example, if it takes two seconds to send each bit, the receiver will examine the signal to determine if it is a 1 or a 0 after one second has passed, then it will wait two seconds and then examine the value of the next bit, and so on. The sender does not know when the receiver has ―looked‖ at the valu e of the b it. Th e sender only knows when the clock says to begin transmitting the next bit of the word. When the entire data word has been sent, the transmitter may add a Parity Bit that the transmitter generates. The Parity Bit may be used by the receiver to perform simple error checking. Then at least one Stop Bit is sent by the transmitter. When the receiver has received all of the bits in the data word, it may check for the Parity Bits (both sender and receiver must agree on whether a Parity Bit is to be used), and then the receiver looks for a Stop Bit. If the Stop Bit does not appear when it is supposed to, the UART considers the entire word to be garbled and will report a Framing Error to the host processor when the data word is read. The usual cause of a Framing Error is that the sender and receiver clocks were not running at the same speed, or that the signal was interrupted. Regardless of whether the data was received correctly or not, the UART automatically discards the Start, Parity and Stop bits. If the sender and receiver are configured identically, these bits are not passed to the host. If another word is ready for transmission, the Start Bit for the new word can be sent as soon as the Stop Bit for the previous word has been sent. Because asynchronous data is ―self synchronizing, if there is no data to transmit, the transmission line can be idle.

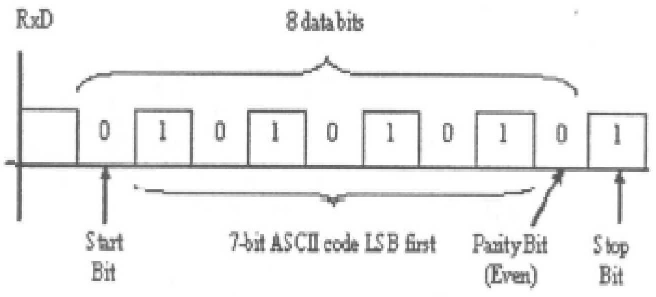


Fig. 3 UART data format

VI Implementation Of Uart Using Vhdl

As integrated circuit technology has become more complex, detailed design of systems at the gate and flip flop level has become very tedious and time consuming. For this reason, use of hardware description languages in the digital design process has significantly improved in the last few years, especially for FPGA design. A hardware description language allows a digital system to be designed and debugged at a higher level before conversion to the gate and flip-flop level. One of the most popular hardware description languages is VHISC hardware description language (VHDL). It is used to describe and simulate the operation of a variety of digital systems, ranging in complexity from a few gates to an interconnection of in any complex integrated circuits. There are in any excellent hardware description languages (HDL) were prior to VHDL but VHDL offers a number of benefits over other HDL's .Therefore the use of VHDL (Very High Speed Integrated Circuit Hardware Description Language) is preferred to design such circuits especially for FPGA design. VHDL can be used to describe and simulate the operation. of digital circuits ranging from few gates to more and more complex gates. VHDL can be used for the behavioural level design implementation of a digital UART and it offers several advantages. Below are the advantages of using VHDL to implement UART.

1) VHDL allows us to describe the function of the transmitter in a more behavioural manner, rather than focus on its actual implementation at the gate level.
2) 2) VHDL makes the design implementation easier to read and understand, they also provide the ability to easily describe dependencies between various processes that usually occur in such complex event-driven systems.
3) It is easier to test the UART by the VHDL simulation and find out if any discrepancy occurs.
4) VHDL as a Standard Language.
5) Better design.
6) Reusability for new technology.
7) Tools independence.
8) Minimum cost 2nd time.
9) Increase productivity.

VII Design Steps For Implementing Uart In Fpga

In embedded systems, the processor that we choose for our design may not come with built-in peripherals. Therefore, designers will have to implement these devices in hardware keeping in mind that they will need to interface to the processor. , it’s possible to perform a logic simulation on every design level. Correct simulation can save a lot of time. Good practice is to simulate at least the top level design files. The top-level simulation is called functional. For larger design is suitable to simulate post-place & route netlist. This simulation is called timing. When you design some FPGA application you will usually do following several steps:

1. You enter your logic description using an HDL (hardware description language) such as VHDL. You can also use schematic description but it’s complicated for complex designs.

2. Now you can use logic synthesis tool for translating your description to a netlist. The netlist is description of generic logic primitives like multiplexers, registers, gates, etc. The netlist also contains information about connection of the primitives.

3. After the synthesis is completed you can use implementation tools. These tools consist of a map tool and a place & route tool. The map tool converts an RTL netlist from synthesis to another netlist. This netlist contains FPGA primitives only. The place & route tool interconnects these primitives using FPGA routing resources.

4. To allow configuration of an FPGA it is necessary to create a bitstream. The bitstream is a file that describes especially states of electronic switches of an FPGA.

5. The bitstream can be downloaded into an FPGA chip. After the downloading the FPGA will perform the operation specified by description from first point.

Therefore the steps of implementation must be carried out in this order:

1. Synthesize

2. Design implementation (Map, Place and Route)

3. Timing Simulate

4. Program.

Synthesize: The synthesis tool will only attempt to synthesize the file highlighted in the Source window. The synthesis tool recognizes all the lower level blocks used in the top-level code and synthesizes them together to create a single netlist. The synthesis tool will never alter the function of the design, but it has a huge influence on how the design will perform in the targeted device.

Design Implementation (Map,Place And Route): The place-and-route tools (PAR) automatically provide the implementation flow described in this section. The practitioner takes the EDIF netlist for the design and maps the logic into the architectural resources of the FPGA (CLBs and IOBs, for example). The placer then determines the best locations for these blocks based on their interconnections and the desired performance. Finally, the router interconnects the blocks. The PAR algorithms support fully automatic implementation of most designs. For demanding applications, however, the user can exercise various degrees of control over the process. User partitioning, placement, and routing information is optionally specified during the design-entry process. The implementation of highly structured designs can benefit greatly from basic floor planning. The implementation software incorporates timing-driven placement and routing. Designers specify timing requirements along entire paths during design entry. The timing path analysis routines in PAR then recognize these user-specified requirements and accommodate them. Timing requirements are entered in a form directly relating to the system requirements, such as the targeted clock frequency, or the maximum allowable delay between two registers. In this way, the overall performance of the system along entire signal paths is automatically tailored to user generated specifications. Specific timing information for individual nets is unnecessary.

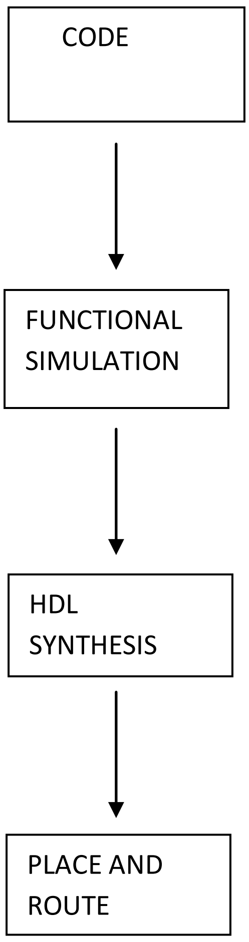
Place and Route For FPGAs, the Place and Route programs are run after Compile. “Place” is the process of selecting specific modules or logic blocks in the FPGAs where design gates will reside. “Route” as the name implies, is the physical routing of the interconnect between the logic blocks. Most vendors provide automatic place and route tools so the user does not have to worry about the intricate details of the device architecture. Some vendors have tools that allow expert users to manually place and/or route the most critical parts of their designs and achieve better performance than with the automatic tools. Floor planner is a form of such manual tools. These two programs require the longest time to complete successfully since it is a very complex task to determine the location of large designs, ensure they all get connected correctly, and meet the desired performance. These programs however, can only work well if the target architecture has sufficient routing for the design. No amount of fancy coding can compensate for an ill-conceived architecture, especially if there is not enough routing tracks. If the designer faces this problem, the most common solution to is to use a larger device. And he will likely remember the experience the next time he is selecting a vendor. A related program is called Timing-Driven Place & Route (TDPR*)*. This allows users to specify timing criteria that will be used during device layout. A Static Timing Analyser is usually part of the vendor’s implementation software. It provides timing information about paths in the design. This information is very accurate and can be viewed in many different ways (e.g. display all paths in the design and rank them from longest to shortest delay). In addition, the user at this point can use the detailed layout information after reformatting, and go back to his simulator of choice with detailed timing information. This process is called Back-Annotation and has the advantage of providing the accurate timing as well as the zeros and ones operation of his design. In both cases, the timing reflects delays of the logic blocks as well as the interconnect. The final implementation step is the Download or Program.

Timing Simulate: Timing and related tools can be somewhat of a mystery in FPGAs. For synchronous logic, you must verify that the inputs of flops arrive before the clock does. This can be controlled in a large part by carefully coding the design to minimize the levels of combinational logic between flops. After synthesizing some code and looking at the timing results, you quickly learn how to code for synthesis. Pipelining is a standard technique for speeding up a design and is done by breaking up large amounts of combinational logic into several stages with flops between the stages. Medium speed designs (30MHz +) will probably require some degree of pipelining. Timing can also be improved by entering timing constraints that the place and route tools try to meet. Timing constraints should be standard part of any FPGA design entry. The following groups of constraints are typical:

1. Input pin to flop input
2. Flop to flop (must be less than the clock period)
3. Flop to output pin
4. Pin to pin (asynchronous stuff)
5. Higher end synthesis tools can also use timing constraints

Static timing analysis is the primary timing verification tool that checks the worst case delays against the timing constraints entered by the designer. Timing simulations of routed designs are useful for debugging and sanity checks, but can be misleading because a timing simulation will most likely not exercise every possible combinational path.

Program: After the design is implemented in real hardware, the debug phase starts. FPGA designs are difficult to debug on the bench because internal signals are not visible unless routed to unused pins or analyzed by special FPGA debug equipment. This is where a good verification environment pays off. The design should be mostly correct if you did the functional simulations and static timing analysis. Chances are some corners where cut and some problems will come up in unverified parts of the design. Once the symptoms are identified, the designer can go back to the functional simulation, recreate the scenario and observe what is going on inside the chip. If timing problems are suspected, a timing simulation can be run on a structural model of the chip.



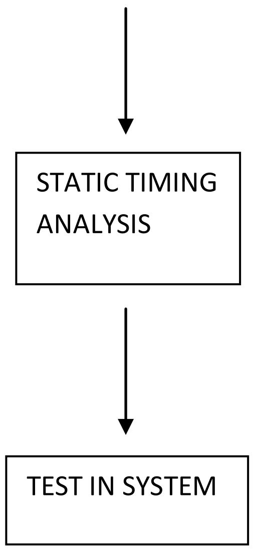


Fig.4 Design steps

Conclusion

FPGAs are suited to very fast, I/O intensive operations. Partitioning your design across the two devices can increase overall system speeds, reduce costs, and potentially absorb all of the other discrete logic functions in a design – thus presenting a truly reconfigurable system. The design process for a microcontroller is very similar to that of a programmable logic device. This permits a shorter learning and designing cycle. UART controller is designed within FPGA based on SRAM with high speed and high reliability. It is used for programming FPGA. The controller can be used to implement communications in complex system And it also can be used to reduce time delays between sub-controllers of a complex control system to improve the synchronization of each sub-controller. The controller is reconfigurable and scalable. FPGAs are replacing conventional programmable logic devices and within next few years they would be required at each and every application.

References

[1] Gina R. Smith.”The basics of constructing FPGA.” eetimes, Jan 2008.

[2] Cliff Brake.”Digital Design Basics”, Nov 2002.

[3] Wilfried Elenmenriech, Martin Delvai."Time triggered communication with UARTs." 4th IEEE international workshop on factory communication system, Sweden, August 2002.

[4] Raffaele Gallo, Martin Delvai, Wilfried Elmenreich, Andreas Steininger." Revision and Verification of an Enhanced UART", Austria, 2004.

[5] M. Delvai, U. Eisenmann, W. Elmenreich, "Intelligent UART Module for Real-Time Applications." First Workshop on Intelligent Solutions in Embedded Systems (WISES), pages 177-185, Vienna, Austria, June 2002.

[6] Brian C. O'Neill, Steve Clark, and Kar L. Wong. "Serial Communication circuit with optimized skew characteristics." IEEE communications letters, VOL. 5, NO. 6, June 2001.

[7] R. Hotchkiss, K. L. Wong, B. C. O'Neill, G. C. Coulson, S. Clark, and P. D. Thomas, "The building blocks for a parallel network incorporating The Strong ARM microprocessor," in *PDPTA'98 Conf.*, July 1998,pp.1863–1870.

[8] J. Norhuzaimin and H.H Maimun,"The design of high speed UART," Asia –Pacific conference on Applied Electromagnetic, Malaysia, Dec 2005.

[9] Karalis, Edward, "Digital Design Principles and Computer
Architecture." Prentice-Hall, United States of America, 1997.

[10] Mohd Yamani Idna Idris, Mashkuri Yaacob, "A VHDL

[11] Implementation of BIST Technique in UART Design." Faculty of Computer Science And Information Technology, University Of Melaya, 2003.

[12] S. Brown, R. Francis, J. Rose, Z. Vranesic, "Field-Programmable Gate Arrays" Kluwer Academic Publishers, May 1992.

[13] S. Trimberger, Ed., "Field-Programmable Gate Array Technology." Kluwer Academic Publishers, 1994.

[14] J. Rose, A. El Gamal, A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays." in Proceedings of the IEEE, Vol. 81, No. 7, July 1993,pp. 1013-1029.