UNIVERSITY OF CINCINNATI

25th Nov. , 20 02

I, Faisal Muslehuddin **,**

hereby submit this as part of the requirements for the degree of:

Master of Science

in:

Computer Engineering

It is entitled:

Bit Stream Modification to Improve the Debugging Capabilities of Re configurable Computing Systems

Approved by:

Dr. Karen Tomko

Dr. Harold W. Carter

Dr. Carla Purdy

Bit Stream Modification to Improve the Debugging Capabilities of Reconfigurable Computing Systems

A Thesis submitted to the

Division of Research and Advance Studies
of the University of Cincinnati

In partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

In the Department of Electrical and Computer Engineering
and Computer Science of the College of Engineering

November 2002

By

Faisal Muslehuddin

Bachelor of Engineering
In
Electronics and Communication
Osmania University, Hyderabad, INDIA, 1999
Thesis Advisor and Committee chair: Dr. Karen Tomko

Abstract

With the increasing design complexity of applications implemented on Field Programmable Gate Array (FPGA) based hardware platforms the time for debugging becomes a major bottleneck in achieving early time to market goals. Traditionally designs implemented on SRAM based FPGAs have been debugged in a manner similar to ASIC designs using simulation during the early stages of the design process. However, FPGA device features such as configuration readback, reprogramability, and clock stepping support the debugging of designs directly on the FPGA-based target platform thus speeding up the debugging and verification process.
Recently FPGA vendors and academic researchers have introduced integrated logic analyzer (ILA) cores that are added to a user's design in support of such in-situ debugging. In this thesis we have developed a method to enable addition of such cores at the last step of the FPGA design flow. This can ease and expedite the process of debugging designs implemented on FPGA based systems by improving their execution control, reducing the time taken to modify the debugging functionality and hence aid in achieving early time to market goals. The process of developing these debugging techniques was divided into two phases.

1. In the first phase a symbol table is created which contains information of the mapping of the logical design synthesized by the synthesis tool to the physical FPGA resources.
2. The goal of the next phase is to improve the execution control of the design. This is achieved by adding bitstream generated Integrated Logic Analyzer Cores into the designs using the JBits and JRoute bitstream modification tools.

The process of ILA core generation and addition was automated as a part of this research. From the experiments that were conducted it was observed that the time taken to add these ILA cores to the FPGA design was quite small. On average the technique developed took 8-10 times less time for ILA core addition when compared to the other techniques. In the case of complex designs the time saved was even more significant.

Dedicated to my dear Parents

Acknowledgements

I thank the Almighty, the most beneficent, the most merciful for all the blessing He bestowed on me and for fulfilling my dreams and aspirations.

I sincerely and wholeheartedly thank my advisor, Dr. Karen Tomko for her guidance and support during the course of my research at University of Cincinnati. It was always a pleasure to work with her. I am also thankful to Dr. Hal Carter and Dr. Carla Purdy for devoting some valuable time from their busy schedules for being on my defense committee.

It has been a pleasure to be a part of the Advanced Computing lab (ACL) where my colleagues Hananiel, Sriram, Anurag, Sinthel, Priya, Sherif, Ellen and Amr were a constant source of inspiration and always ready to help. I thank them for maintaining a motivating and pleasant atmosphere in the lab. I am also thankful to my roommates Jawad and Akber for their invaluable advice and suggestions during the course of my research and I really enjoyed my stay with them.

I would like to thank my parents from the deepest of my heart for their guidance and support through the entire course of my career. They were always there to help me in everyway they could whenever I needed them and I do not have enough words to express my gratitude towards them. I would also like to convey my thanks to my sisters for all their love and good wishes.

Copyrights and Trademarks Notice

- *Xilinx, Virtex, Virtex-II, Chipscope ILA and Xilinx Alliance tools are trademarks of Xilinx Inc.*
- *Signal Tap Mega function ELA is trademark of Altera Corporation.*
- *JHDL is trademark of BYU.*

Contents

List of Figures

List of Figures

CHAPTER 1
INTRODUCTION

1 Introduction:

1.1 Introduction to Reconfigurable Computing: Reconfigurable Computing is an innovative approach in the area of Computing Systems design, inorder to cope with the drawbacks of the conventional Computing Systems, due to their general purpose nature. It aims at reducing the gap between Hardware and Software Computing by trying to achieve the advantages of both i.e. the flexibility of Software and the efficiency of Hardware at the same time. In recent years there have been many instances in which the potential of Reconfigurable Computing Systems has been demonstrated in many different application domains. Survey [39] describes many FPGA based systems which have demonstrated excellent performance for different applications. The following are some of the systems which are mentioned in this survey.

a. The Splash system [40] gave 200 times better performance on genetic string matching algorithms when compared to supercomputers implementations.
b. The DECPeRLe-1 system [41] achieved a very fast encryption rate of 185 kbps with 970 bit keys and 600 kbps with 512 bit keys for data encryption based applications.

An FPGA in general can be viewed as a two dimensional network of Configurable Logic Blocks which are surrounded by the Input Output Blocks on the periphery of the chip. The different components that make up an SRAM based FPGA are Configurable Logic Blocks (CLB), Input/Output Blocks (IOB), Block Rams, buffers and the configurable interconnect resources that are used to connect these primitives together. The CLBs consists of Look up Tables (LUTs) which can be configured to implement

different Boolean functions and Flip Flops to implement components of the design that need to save state such as registers, counters, control logic, etc. The Block Rams are used to provide on chip memory for DSP and other applications [42] that require low memory latency and high bandwidth as the data in the memory is accessed frequently.

With the advent of multimillion gate FPGAs like the Vertex [1] FPGA architecture from Xilinx which consists of more than 10 million gates the amount of reconfigurable resources that are available on these devices have increased drastically. With improvement in the VLSI chip technology and improvement in the Mapping, Placement and Routing algorithms the execution speed of the designs that are mapped on the FPGAs have also improved rapidly resulting in implementation of complex and high frequency designs on FPGA Based Systems. Recently, Xilinx has introduced a new FPGA architecture, the Vertex II [43] which in addition to providing traditional configurable resources like CLBs, IOBs and Block Rams also includes dedicated resources such as Multipliers, Adders, etc to speed up DSP and other applications which require fast arithmetic computations.

1.2 Design Verification techniques for FPGA Based Systems:

Design verification is a very important and time consuming step in the process to bring a new system design to successful completion. Designs are verified at different stages of the design process. In the case of FPGAs the different stages at which designs are verified is depicted in Fig 1.1. Once the design is described in a Hardware Description Language the first step is to verify the functional correctness as per the given specifications, which is generally performed using a simulation environment. The

later in the design cycle the bugs are detected the more time consuming and expensive, it is to correct them. Hence it is desirable to detect and correct the errors in the early stages of the design process which will help in reducing both the time and design effort.

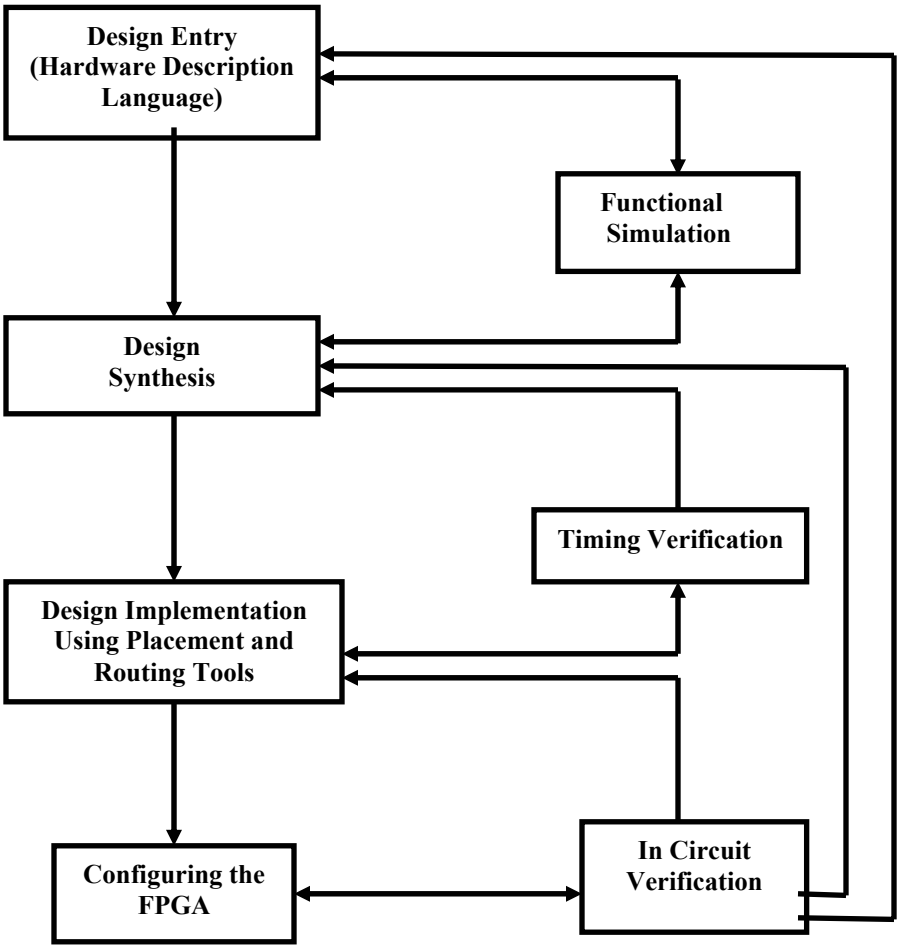


Fig 1.1 The FPGA design Flow

There are some design errors, however, that can be detected only in the later stages of the design process such as the timing violation which can be detected only after logic synthesis or after the design is mapped onto an FPGA based System. The following is a

brief overview of the various verification techniques that are employed during the development of FPGA based Systems.

1.2.1 Functional Verification:

Functional verification is the process of verifying the functional correctness of the design. Designs are verified by simulation tools which take the HDL description of the design at various levels of design abstraction such as the Behavioral level, the RTL level or the Gate level as input and verify the functional correctness by applying a set of test vectors and verifying the output generated against the expected output for each of the test vectors. The strengths of simulation include high controllability, observibility and control over the execution of the design. But a linear increase in the complexity of the design produces an exponential increase [48] in the number of test vectors required to verify the functional correctness of the design making the process of simulation impractical for complex designs since it becomes very time consuming to simulate for all the test vectors that are necessary to exhaustively test the design functionality.

An alternative technique to simulation is emulation, also called Rapid Prototyping, where the designs are verified for their functionality directly in hardware by mapping the synthesized design to an FPGA based emulator [43] and then verifying their functionality by applying the set of test vectors as in the case of simulation. The advantage of emulation is a reduction in execution time as the design is implemented and executed directly on the hardware. There has been significant amount of work reported in the literature [44] which has demonstrated the capabilities of in circuit emulators.

1.2.2 Timing Verification:

Timing verification is the process of verifying that the system meets the desired timing goals [45] such as achieving the desired clock frequency, satisfying the setup and hold times of flip flops, etc. Timing verification is performed later in the design process as can be observed from Fig 1.1 after the design is synthesized and also after the design is mapped onto an FPGA. Timing analysis that is performed after the design is synthesized is termed as static timing analysis where the critical paths in the design are identified and this information is passed onto the Placement and Routing tools so that the components that are a part of the critical path can be placed in close proximity to each other to satisfy the timing goals and avoid any timing violations. In this step the delays due to FPGA interconnect are not considered, so its accuracy is not as good as the post placed and routed timing analysis. The next instance where the timing analysis is performed is after the design has been mapped to the FPGA and this includes the delays due to the FPGA interconnect. Thus the goal of timing analysis at different stages in the FPGA design flow is to make sure that designs implemented on FPGA Based Systems satisfy the desired timing specifications and design goals.

1.3 Debugging:

The process of debugging a design consists of error detection, error diagnosis and error correction [46]. Error detection is the process of detecting the location in the design where the error has occurred which is responsible for the malfunctioning of the design. Error diagnosis is the process of detecting the cause of the error and error correction is the process of coming up with a remedy to resolve the problem or error.

For a design to be debugged efficiently it should have good controllability, observibility and control over the execution of the design. The following is a brief description emphasizing the importance of each of these techniques.

1.3.1 Controllability:

Controllability is defined as the ability to set the value of any internal signal and components in the design to a desired value. This is important when one is interested in testing a given section of the design whose input cannot be controlled directly from the primary input of the design. This is important when a designer is trying to detect an error and inorder to narrow down the search of the actual location of the error the designer might be interested in controlling the values of certain internal signals. In the case of Application Specific Integrated Circuits (ASICs) this capability is provided by the addition of Scan Chains and other ad hoc debugging techniques. There has been a lot of research that has been reported in this area and one of the famous techniques for scan chain design is the Level Sensitive Scan design [47] technique being employed at IBM.

1.3.2 Observibility:

Observibility is defined as the ability to observe the value of any of the internal signal and the components in a given design. This feature is important when the designer is interested in monitoring the internal signals during the execution of the design. There are some nodes in the design which are difficult to monitor and it is difficult to generate test patterns so that the values on those nodes can be observed at the primary output. In such cases the observibility of the design can be improved by adding debugging logic

such as scan chains, multiplexers, etc to the design. Fig 1.3 demonstrated how the observibility of the circuit given in Fig 1.2 can be improved by connecting a wire from

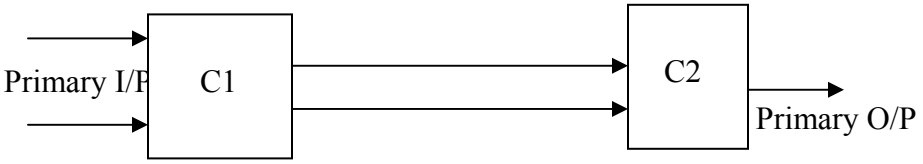


Fig 1.2 An example design implemented on FPGA

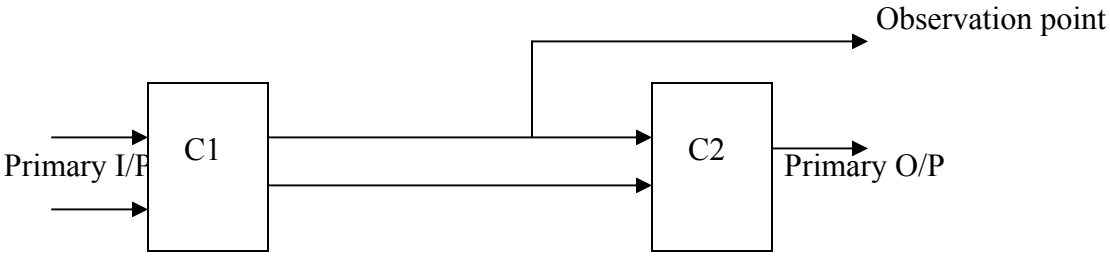


Fig 1.3 Circuit with improved Observibility

one of the output of C1 which is a internal node and connecting it to the primary output if any unused I/O pin is available. In case of FPGA based designs the observibility can be improved through readback which is the ability to read back the state of various elements in the FPGA like flip flops, Block Rams, etc. The different families which support this feature include the XC4000, Virtex and Spartan from Xilinx and Lucent FPGAs.

Debugging of designs in case of ASICs differ from FPGAs as ASICs are tested for manufacturing defects after the design has been fabricated, along with the debugging logic that has been added to the design inorder to improve its debugging capabilities. These design for testability (DFT) techniques help in improving the controllability,

observibility and execution control of the design during the testing process after the chip has been fabricated. This extra logic that is added in ASIC designs cannot be removed from the chip after the design has been tested. The goal that needs to be achieved by the addition of these debugging logic cores differ in ASICs and FPGAs. In case of ASICs as discussed above the purpose of debugging logic is to aid in the testing of the chip for manufacturing defects after fabrication but in case of FPGAs the purpose of debugging logic is to aid in the process of functional and timing verification during the development of the system and the debugging logic can be removed once the functionality and timing of the system has been verified and this is possible due to the reprogrammable nature of FPGAs.

1.4 Motivation:

Traditionally designers who implement their designs on FPGA Based Systems follow the same design flow as used for design of ASICs. During the initial stages of the design process when the specifications of the design are available, designers try to describe the design at the behavioral or RTL level depending on the complexity of the design using a Hardware Description language and try to simulate the design using a circuit simulator. The attractive feature of circuit simulation is in its ability to provide almost complete visibility and control over the execution of the design during simulation. These features include monitoring of internal signals, forcing the values of certain signals as and when desired, halting the execution of the design on the occurrence of certain trigger conditions, etc. These features are very important when the design is in its initial stages of development when there are many issues that need to be resolved.

With the arrival of multimillion gate FPGAs into the market from vendor like Xilinx and Altera the complexity of designs that are implemented on these FPGA based systems is increasing rapidly. The time required for simulating these systems is high due to the reasons discussed in section 1.2.1.

An alternative technique which tries to mitigate the long simulation times of these complex systems is Rapid Prototyping or in circuit emulation which was also discussed in section 1.2.1. But there are some issues that have to be addressed when considering in circuit emulation. Controllability and observibility must be available during execution of the designs to make it a viable alternative to simulation.

Designs tools for implementing circuits with FPGA based system as their final target platform similar to ASIC design systems which are verified using the emulation environments have to address this problem of poor debugging capabilities. SRAM based FPGAs with flexible resources like CLBs, IOBs, Block Rams and Configurable interconnect and features such as configuration readback, configuration writeback, reprogramability and JTAG Boundary Scan interface have the potential for supporting the desired debugging capabilities as discussed above. Towards achieving the goal for supporting improved debugging capabilities for FPGA based designs using these unique features of FPGAs many researchers in both the academic and commercial domains have come up with debugging circuits called Integrated Logic Analyzers (ILA) which can be added to the design to improve its observibility and execution control. One of the prominent academic tools include the ILA cores developed by researchers at BYU [25] that support different debugging capabilities and the techniques used to add them to the design at various stages of the design flow. This work supports the addition of ILA cores

at the HDL stage before the commencement of logic synthesis and also in the logical database before commencement of Xilinx design flow. Similarly the tools in the commercial domain include the Chipscope ILA [11] [12] from Xilinx and Signal Tap Logic Analyzer [8][9][10] form Altera Corporation. It is observed that the logic analyzers that each of these tools support provide good debugging capabilities. But the problem with these tools lies in the time taken to add and modify the ILA cores based on the designer's requirements, as the entire Xilinx flow has to be repeated whenever these changes are made. It is observed that as the complexity of designs increases the time taken to make these changes also increases drastically. Hence it is desirable to develop a technique that can be used to add the ILA cores to the design as late as possible so that the ILA core addition or modification takes less time. The latest stage at which the ILA cores can be added is at the bitstream stage after the process of bitstream generation which is last stage before the device is configured. Hence, the aim of this thesis is to develop techniques to add the ILA cores to the design at the bitstream stage so that the time taken for addition and modification of these ILAs cores is minimal. By developing such a technique which may be a part of a comprehensive Hardware debugging environment, along with the speed up that is achieved by debugging the design directly on the FPGA hardware, the designer can improve the debugging capabilities of the design by adding and modifying the ILA cores as needed and hence have a rich and flexible investigative tool for his/her debugging needs.

1.5 Contribution of this thesis: In this thesis we have investigated a technique to add and modify ILA cores to the design which takes very less time when compared to the

conventional approaches in both the academic and commercial domains to support a hardware debugging environment so that designs can be debugged directly on the FPGA hardware which is their target platform. The main contribution of this thesis has been the demonstration of the ability to add Integrated Logic Analyzer cores to the design at the bitstream level using the bitstream modification tools such as JBits and JRoute after the user has been provided with a logical view of the design through the process of logical to physical design mapping for the creation of symbol table, which is implemented as a part of this thesis. The process of logic to physical design mapping is also useful to find the state of the logical elements such as flip flops, Ram, etc during hardware execution when readback is carried out, which is another way of improving the observibility of designs. So instead of simulating designs using cycle based or event based simulators which are very slow in simulating complex designs, a hardware debugging environment with the techniques investigated in this thesis and several other capabilities demonstrated by other systems can be developed to verify the functional correctness of the design directly on the target platform.

The following are the steps taken during the process of adding ILA cores to a design as shown in the context of FPGA design flow in Fig 1.4 the goal of each step is given along with a brief description.

1. Logical to Physical design mapping to provide the user with the logical view of the design.
2. Detecting the free resources in the FPGA so that these resources can be used for the addition of debugging logic.
3. Modifying the designs at the bitstream level using JBits and JRoute for adding the

necessary debugging logic.

4. Configuring the FPGA chip using the new debug modified bitstream.

This process speeds up the modification of ILA logic while maintaining the same level of observibility and execution control.

1.5.1 Logical to Physical Design Mapping:

Logical to Physical design mapping is defined as the process of creating a symbol table from the information of the mapping of the logical netlist of the design generated by the synthesis tools to the physical resources in the FPGA after the design has been placed and routed to the device by the FPGA vendors CAD tools. This mapping is important when designs are debugged directly on the FPGA based platform, as the user is familiar with the components in the logical netlist and he/she doesn't have a clear picture of how the design has been mapped to the primitives in the FPGA. Along with the generation of a bitstream used to configure the FPGA CAD tools also generate several files which have information of how the logical components or instances in the design are mapped to their physical counterparts in the FPGA. To achieve good observibility and control over the execution of the design it is necessary to provide a mapping of all the signals and components in the FPGA that are a part of the netlist to their logical counterparts so that debugging can be carried out efficiently. Since the user is familiar with the logical design he/she should be provided with the logical view even when the design is being executed on the FPGA hardware. For instance if the designer is interested in reading the content of a particular flip flop in the logical design then the user should not need to know what it corresponds to in the physical FPGA. This should be taken care

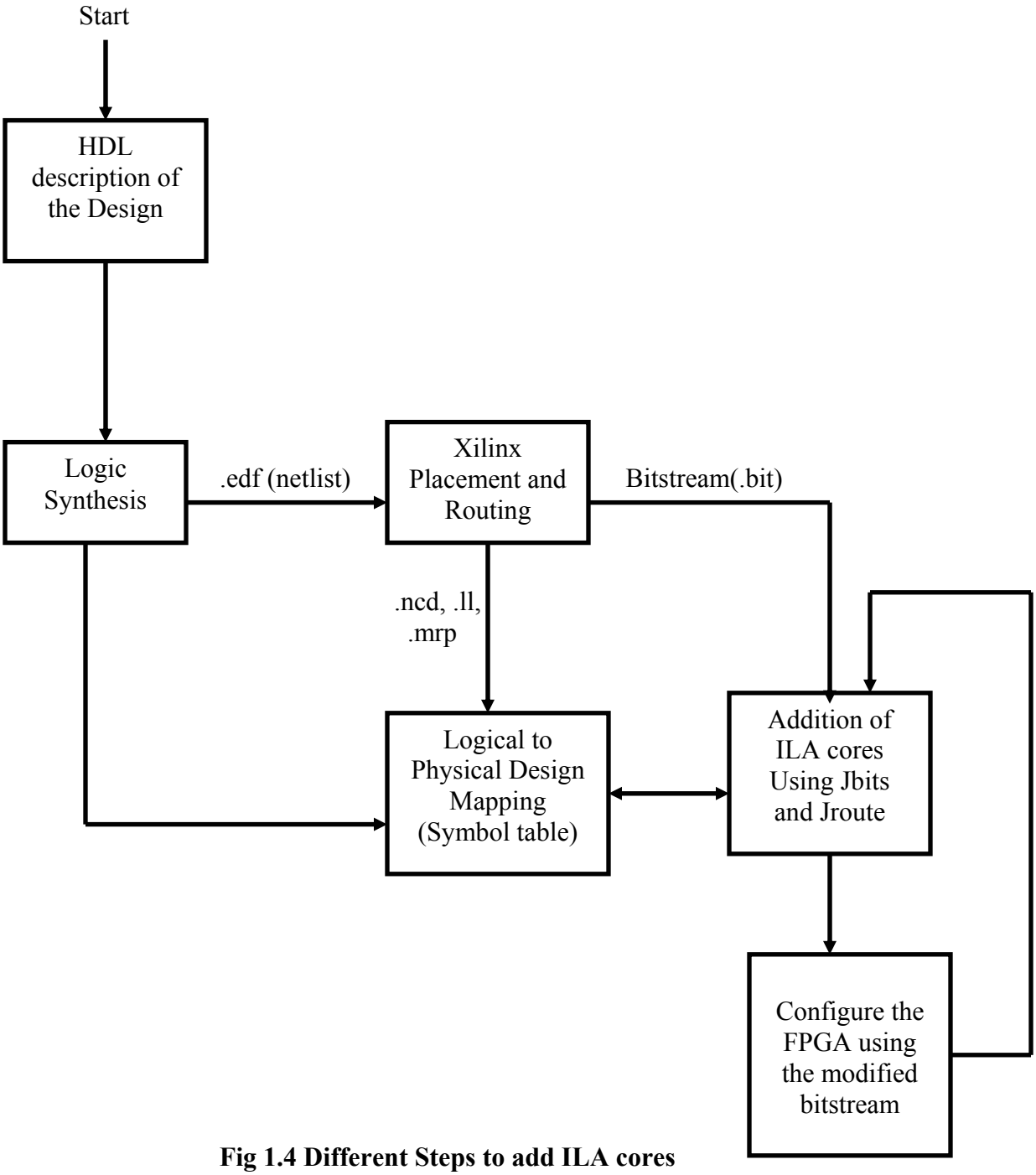


Fig 1.4 Different Steps to add ILA cores

of by the design environment so he/she can concentrate on the more important task of error detection at hand. Thus this information of logical to physical design mapping is

gathered in a Symbol Table which will be referred to by the Hardware Debugging Environment whenever the information is desired.

1.5.2 Detection of free resources:

Once the design has been mapped to the FPGA based system not all of the resources in the FPGA are used. Depending on the complexity of the design being implemented different percentage of resources remain free. For instance the user might be interested in controlling some of the signals in the design. An example of such improved observability is depicted in Fig 1.3. Inorder to achieve this some of the free resources inside the FPGA along with the unused interconnect can be used to implement debugging logic cores which provide this improved debugging capability. Inorder to do this first the free resources in the FPGA have to be identified. The information of the resources utilized to implement a particular design are present in Native Circuit Description (NCD) file and by parsing this file a table with the free resources can be created which is used to guide the process of debugging logic addition without disturbing the already implemented design on the FPGA. In this thesis we have implemented a parser which parses this NCD file and generates a table with the information of the free resources which is utilized for the addition of debugging logic.

1.5.3 Addition of Integrated Logic Analyzer Cores at Bitstream level using JBits & JRoute:

Once the Logical to Physical design mapping symbol table is created and the free resources in the FPGA are identified the next step involves the improvement in the

debugging capabilities of the design. The debugging capabilities required for a rich interactive tool includes features like single or multi stepping the clock, reading back the state of the intermediate and internal signals, generation of interrupt signals on the occurrence of certain trigger conditions, saving the intermediate state of certain signals in data buffers, stopping the clock on the occurrence of certain trigger condition, forcing the values of certain internal signals, etc. Inorder to achieve these goals Integrated Logic Analyzers (ILA) have to be added to the FPGA designs[9][10][11][12][25]. There are many stages in the design flow where these ILA cores can be added. But since one might be interested in making changes to these ILA cores depending on the debugging capabilities that the user wants while diagnosing a problem, the best case scenario would be to add it in the latter stages of the design process so that the overhead in terms of time taken can be minimized. Towards achieving this goal we have developed techniques to add the ILA cores at the bitstream level after the configuration bitstream for the design is generated. From our experiments it has been observed that on the average it takes 8 times less time to add the ILA cores at the bitstream level when compared to the time taken for the addition of the debugging logic at the earlier stages of the design flow.

1.6 Organization of the thesis:

The following is the organization of the rest of this thesis. Chapter 2 will provide an introduction to the different features of FPGAs, the different capabilities that are available in FPGA Based System that are utilized, the design tools used in this thesis and the background to the previous work done. Chapter 3 will describe in detail the process of logic to physical design mapping for the creation of a Symbol table. Chapter 4 will

explain the process of modifying the designs at the bitstream level using JBits and JRoute inorder to add the Integrated Logic Analyzer cores to improve the debugging capabilities of the designs. Chapter 5 will describe the experiments that were conducted and the insight gained from those experiments. Finally, Chapter 6 will conclude the thesis and provides a brief introduction to the potential future work related to this thesis.

CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter will give a detailed description of the Virtex™ FPGA architecture from Xilinx, the different configuration techniques supported by Virtex™ FPGAs and the different features available in this architecture that are utilized in this thesis to improve the debugging capabilities of designs. It also gives an introduction to JBits and JRoute, the Java Based APIs developed by Xilinx which are used in this thesis for the modification of designs at the bitstream level inorder to add the desired debugging logic to the design. Finally, the chapter concludes with a description of the previous work done in this area.

2.1 Introduction to Virtex™ FPGA Architecture:

The different components that make up a Virtex™ [1] FPGA architecture include Configurable logic Blocks (CLBs), Block Rams, Input Output Blocks (IOBs), buffers and the configurable routing resources. A brief description of each of the components is given below.

2.1.1 Configurable Logic Blocks (CLBs):

Each Virtex™ CLB consists of four Logic cells (LC) which are the basic component that make up a CLB. A CLB consists of two slices where each slice contains two Logic Cells and some other configurable resources that are a part of the CLB, which can be used to handle different situations and requirements as discussed below.

Each Logic cells consists of a 4-input Lookup table (LUT), carry logic and the storage element. Figure 2.1 gives a schematic of a typical LC of a Virtex™ CLB. The LUT in a LC can be used to implement any function of four variables. It can also be used as a 16x1 SRAM or as a 16 bit shift register. In order to implement functions of more than four variables the LUTs from different LCs can be combined to handle this situation. For instance a function of five variables can be implemented by combining the outputs of the two 4 input LUTs and passing them as input to a

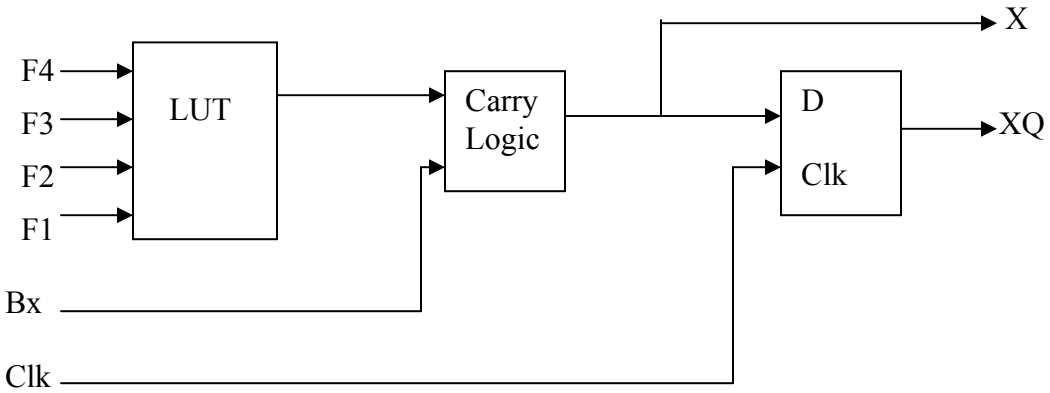


Fig 2.1 A Schematic of the Logic Cell in a Virtex™ CLB

multiplexer which is a part of the CLB. Similarly functions of more than five variables can also be implemented using similar techniques. Thus the availability of ample configurable resources in the CLB provides a great deal of flexibility.

The LCs also contains dedicated carry logic which is used to expedite the carry operation in case of arithmetic computations for DSP and other applications. Each slice of the CLB implements 2 bits of the carry logic chain. Some extra resources like an XOR gate are available in each LC which can be utilized to implement a 1 bit full adder in each of the LCs.

The storage element available in each LC can be used either as an edge triggered flip flop or level sensitive latche. The input to the storage element can be the output of the LUT or a direct input to the slice.

2.1.2 Block RAMs:

For applications which access the contents of memory frequently such as DSP applications it is desirable to provide on chip RAM to reduce the latency of memory access. Towards serving this goal Xilinx has provided on chip memory in the form of Block Rams modules each containing 4096 bits of storage space. Fig 2.2 shows a block diagram of a Virtex™ Block Ram. Each of the 4096 bit blocks can be configured as memory module with varying aspect ratios ranging from 4096x1 to 256x16. The table in Fig 2.3 gives the different aspect ratios to which each of the Block Rams can be configured.

2.1.3 Input Output Blocks (IOBs):

The Input Output Blocks in the FPGA act as an interface between the IO pads and the internal FPGA resources. Each IOB contains three storage elements each of which can be used as a D flip flop or a level sensitive latch. The IOBs have capabilities to support various I/O modes.

2.2 Different Configuration Techniques:

Configuration is defined as the process of storing the bitstream generated by the FPGA CAD tools for configuring the FPGA in its configuration memory. The

configuration bitstream includes data which is used to configure various configurable resources in the FPGA like LUTs to implement a particular Boolean function or as a

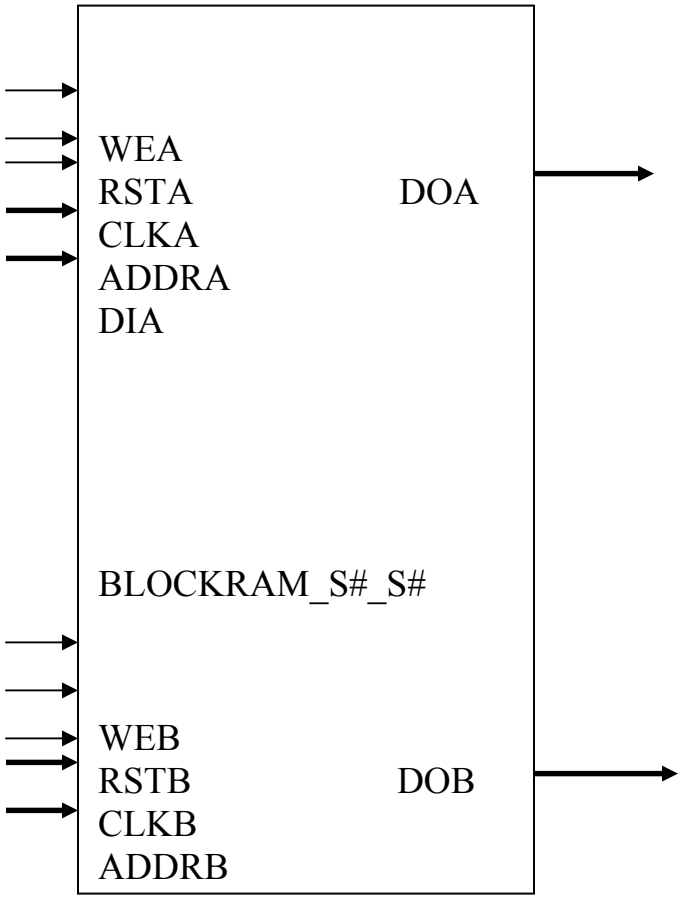


Fig 2.2 Schematic of Block Select Ram taken from the application notes [1]

Data Width	Data Depth	Address Width
1	4096	12
2	2048	11
4	1024	10
8	512	9
16	256	8

Fig 2.3 Table showing different aspects Ratios for Block Rams

16x1 RAM, Block Rams as memory elements with different aspect ratios as discussed in the previous section, a particular IOB as input or output port, configuring the interconnect inorder to connect the FPGA configurable resources together as required by the current design, etc. The various Configuration techniques that are employed to configure the FPGAs can broadly be classified into static Reconfiguration and Runtime or Dynamic reconfiguration. The following is a brief overview of each of these techniques and issues related to each one of them.

Static reconfiguration is the process of reconfiguring the FPGA after the design has been synthesized by the synthesis tools and mapped, Placed and Routed using the FPGA vendors CAD tool. The bitstream generated by the FPGA vendors CAD tool is used to configure the FPGA and once this is done there are no changes made in the configuration of the device when the design is being executed on the FPGA Based System. This process of configuring the device once and not altering its configuration till the execution is complete is termed as static reconfiguration. The advantage of static reconfiguration lies in the time saved as the device is configured once for a single run of the execution. But if a design is complex and does not fit on the available FPGA resources then such designs cannot be implemented on the given FPGA system using static or compile time reconfiguration. This drawback can be addressed by Runtime or Dynamic Reconfiguration. Using Dynamic reconfiguration [2][3] designs which cannot be accommodated on the available FPGA hardware can also be implemented. Here at a given instance only those components of the design are using during computations are mapped to the FPGA and the other components are swapped in and out of the system as and when required.

The Configuration bitstream is divided into frames each of which is used to configure a different portion of the FPGA. A frame is a smallest possible set of bitstream data that can be used to configure a particular section of the FPGA. So incase of partial reconfiguration the smallest possible unit of reconfiguration that can be used is a frame. This feature of reading back or configuring the device using individual frames is important in many different applications. In some applications which support dynamic or partial reconfiguration only a portion of the FPGA needs to be reconfigured and only those frames need to be used which are responsible for configuring the corresponding portions of the FPGA. Similarly incase of debugging designs if the designer is interested in checking the content of a particular flip flop in the design he/she can figure out the frame that contains the content of that particular storage element and then read back only that particular frame. This feature is very useful as reading back the entire configuration of the device would be unnecessary for checking the content of a single storage element. Application note [5] for Virtex™ FPGA gives a detailed description of each frame and their arrangement in the configuration bitstream.

2.3 FPGA features useful in debugging:

Inorder to debug designs on FPGA based systems there are many features available in present day FPGAs which can be utilized towards achieving this goal. These features include configuration readback, writeback, reprogrammability, JTAG Boundary Scan interface, etc. A brief introduction to each of these features available in Virtex™ FPGA and their significance with respect to improving the debugging capabilities of the designs is given below.

2.3.1 Configuration Readback:

Readback is defined as the process of reading back the configuration of the FPGA [4]. Reading back the configuration is useful in many instances. One application of readback is for checking whether the FPGA has been configured using the proper data and any unwanted bit inversion during configuration of the device can be detected through readback.

Configuration readback is used as a means of tracing the content of memory components like Block Select Rams, Flip Flops, and LUTs during the execution of the design. Inorder to relate the content of these components to the data read back one needs to be familiar with the layout of data in the frames of the configuration bitstream and their distribution. The configuration bitstream for Virtex™ FPGAs is divided into columns with each column containing several frames as shown in Fig 2.5 for a Virtex XCV50 device.

Each column has a Major Address and each frame within the column will have a Minor Address. Hence each frame in the configuration bitstream is identified using a combination of Major and Minor addresses. As can be observed in Fig 2.4 different frames in the bitstream are used to configure different resources in the FPGA. For instance the left most and the right most columns contain frames for configuring the IOBs in the left and right column. Similarly the second from the left and the second from the right columns are used to configure the Block Rams in the designs and the remaining frames are used to configure the CLBs, the IOBs in each column and the clock driver as

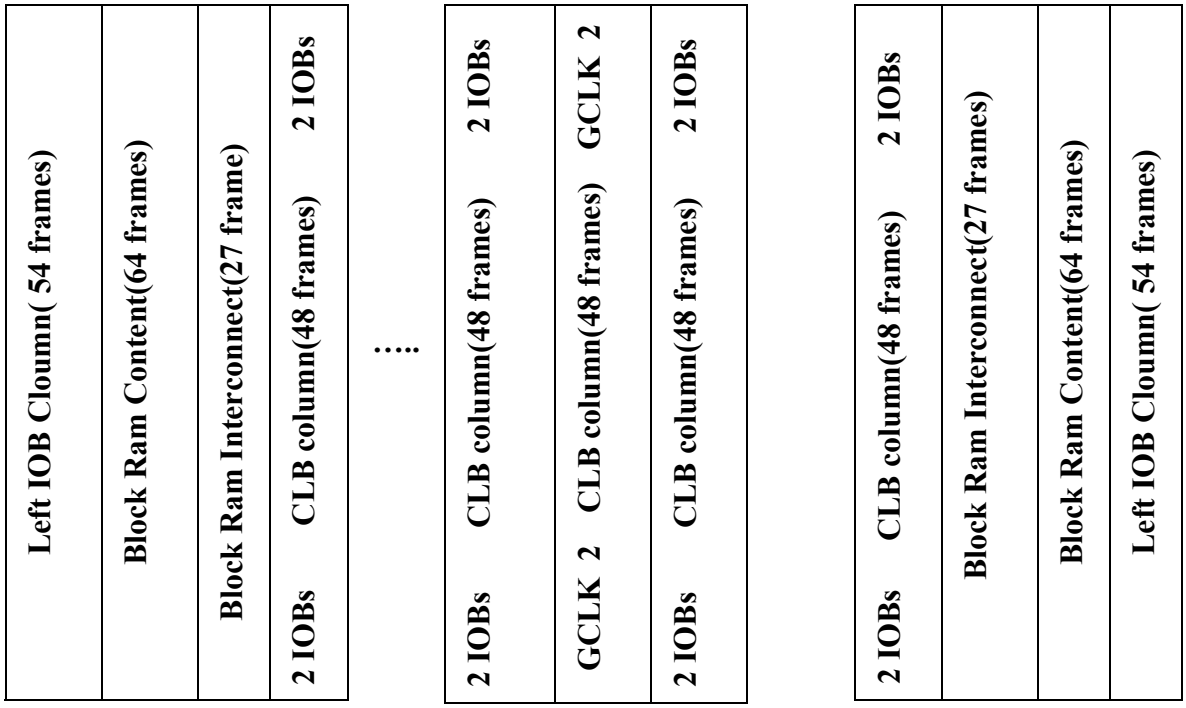


Figure 2.4 Distribution of Frames in the Bitstream for Virtex XCV50 device taken Form the application notes [5]

the Virtex FPGA provides resources for different clock configurations. Different frames in an FPGA contain a different no. of configuration bits depending on the type of resource the particular frame is used to configure. Another point to be noticed is that each frame contains padding bits at the beginning and the end of the frame for demarcation of different frames. The number of configuration bits used to configure different devices in the Virtex™ family is given in the table of Fig 2.5. In summary configuration readback is a very important feature available in the Xilinx Virtex™ FPGA family which can be used for debugging designs during design execution.

Device	No of Configuration bits
XCV50	559,200
XCV100	781,216
XCV150	1,040,096
XCV200	1,335,840
XCV300	1,751,808
XCV400	2,546,048
XCV600	3,607,968
XCV800	4,715,616
XCV1000	6,127,744

Fig 2.5 Table showing the no. of configuration bits for each Virtex device taken from Application notes [5]

2.3.2 JTAG Boundary Scan Interface:

JTAG is an acronym for Joint Test Action Group. It is an IEEE standard which was introduced to support board level testing of designs. Virtex™ FPGAs support this standard and hence contain the circuitry which implements the Boundary Scan controller and the logic which is required to support the operation of the Boundary Scan circuitry within the FPGA. A Virtex™ FPGA contains four ports which are used to communicate with boundary scan circuitry inside the FPGA. They are TDI(test data input) which is used to input the data serially to the data and instruction registers of the Boundary Scan circuitry present in the FPGA, TDO(test Data Output) is the test data output port which

acts as an output interface to the test bus circuitry, TMS(test mode select) is used to control the state of the Boundary scan controller which is in one of the
16 possible states and Tclk which is the clock signal used for clocking the Boundary Scan JTAG interface. JTAG interface is used for configuring the FPGA serially as well as for configuration readback. No extra FPGA pins are used for configuring the device as JTAG interface has the dedicated JTAG pins which are sufficient to carry out these tasks of configuring the device and reading back the configuration. In addition to configuration and readback of designs the Boundary Scan interface also supports many other debugging capabilities. Fig 2.6 gives a list of instruction that are supported by the JTAG interface which facilitate both chip level and board level debugging of designs implemented on FPGA based systems.

2.4 JBits and JRoute:

JBits [6] is a Java Based Application Program Interface (API) which can be used to configure the Virtex™ device at the bitstream level. As JBits works at the bitstream level one should be familiar with the Virtex™ architecture inorder to use this tool efficiently. It supports both static and dynamic or Runtime reconfiguration. Designs implemented using traditional CAD tools can be modified using JBits which takes as input the original bitstream generated by the normal design flow and can make the necessary changes to the design at the bitstream level inorder to provide the desired functionality. So JBits can be used to construct new designs or modify existing designs. Fig 2.7 shows the various steps involved in bitstream modification using JBits and JRoute. The CLBs inside a Virtex™

JTAG Commands	Opcode 5 bit	Description of Command
Extest	00000	Enable boundary Scan external test for testing inter chip interconnect
Sample/Preload	00001	Enable boundary Scan sample/ preload
USER1	00010	Command for accessing user defined Register1
USER2	00011	Command for access user defined Register 2
CFG_OUT	00100	The configuration bus is accessed for read operation
CFG_IN	00101	The configuration bus is accessed for Write operation
INTEST	00111	Test the internal components of the FPGA
USERCODE	01000	Enable shifting the user code to the output
IDCODE	01001	Enable shifting the content of instruction register
HIGHZ	01010	3 state output pin while enabling the bypass register
JSTART	01011	Clock the startup sequence
BYPASS	11111	Enable bypassing of data
RESERVED	All other cases	Other reserved instructions

Fig 2.6 This table gives a brief description of each of the JTAG commands

Taken from the Virtex Data Book [4]

FPGA are represented in JBits as a two dimensional array of configurable logic blocks. Each CLB in the FPGA is represented by an X and Y coordinate which is used to identify the particular CLB. Similar is the case with Block Rams and IOBs.

JBits can be used to implement designs using the Runtime Parameterizable (RTP) cores at the bitstream level that have been provided along with the API. These cores include adders, multipliers, comparators, counters, registers, etc whose bus width can be customized to implements designs with the desired specification. Using these cores to implement the desired design is easier as the designer does not need to worry about the details of the Virtex™ architecture as he/she uses the RTP cores and connects them

together using the JRoute API calls. Using these RTP cores enables the designer to work at a higher level of abstraction which is desirable when the designs are complex. JRoute is a java based API used to route the nets in a Virtex™ device. It can be used to route a single net or a group of nets using a single command. It can also be used to unroute nets in case the need arises using simple API calls. If one is modifying designs using JBits and JRoute then it is important that the routing resources that are already used should not be reused. Inorder to prevent JRoute from reusing the reserved resources the JRoute database is preloaded with the used resources.

2.4.1 Drawbacks of JBits and JRoute:

The following are some of the drawbacks of using JBits and JRoute which were realized while using them during the course of this thesis.

1. There is no way of finding the maximum frequency at which the design can be executed after it has modified using JBits and JRoute.
2. Limited to the Virtex™ chips.
3. No way of determining which routing resources have been used when a net is added with Jroute.
4. User needs to be familiar with the FPGA device architecture to use these tools as one works at a low level of abstraction.

2.5 Related Work: In this section we will try to give an overview of the previous work that has been done in the area of debugging designs on FPGA based systems. We will try

to discuss the FPGA Based Systems and software that have been developed to improve the debugging

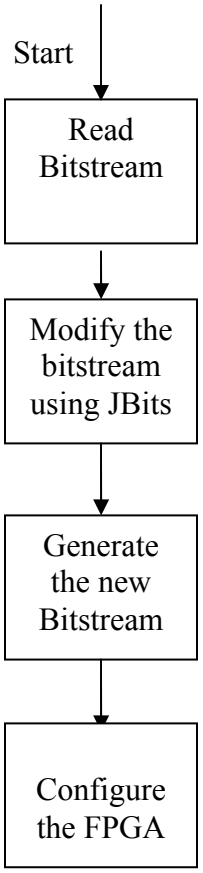


Fig 2.7 The different steps involved in modifying designs using JBits

capabilities of designs and the research that has been done in this area. We will discuss both the commercial and academic systems and software that have been developed and features among controllability, observibility and execution control that they support.

2.5.1 Device Level Support for Debugging: There are many FPGA devices available in the market which supports features that are useful in debugging designs. FPGA vendors

like Xilinx and Altera provide many features like configuration readback and JTAG Boundary Scan interface as discussed in the previous sections which have been used successfully for debugging designs. For instance the Virtex™ FPGA from Xilinx provides the ability to configure the FPGA partially or completely. In addition it also provides the capability to readback the configuration partially as well as completely. Through readback the designer can observe the content of resources inside the FPGA like LUTs, Block Rams, IOBs and Flip Flops [5]. Thus readback helps in improving the observibility of the designs.

Similarly through partial reconfiguration that is supported by the Vitrex™ and Spartan™ FPGAs from Xilinx [5] one can improve the controllability of designs. For instance one can add some small debugging logic cores into the design which will help in controlling the internal values of signals. Also design level scan chains can be added to the design so that the content of the storage elements like flip flops can both be observed and controlled.

Recently introduced devices from Xilinx, the Virtex-II [7] series of devices have Phase Locked Loops (PLL), Delay Locked Loops (DLL) and Digital clock manager. These features are available for implementing system on a chip solution where there can be multiple clocks that are used by the system. They also provide support for single or multi stepping the clock, turning off the clock to certain parts of the system and configuring the clock on the fly through partial reconfiguration. These features can be utilized for controlling the execution of the design during debugging. They provide the ability to stop the clock on the occurrence of certain trigger condition, change the frequency of the clock during debugging through partial reconfiguration, etc.

Device from both Xilinx and Altera support the JTAG boundary Scan interface. Generally JTAG port is used for debugging designs by providing controllability and observibility. In the case of FPGAs in addition to supporting these above mentioned features it is also used for some other purposes.

In the Altera Apex 20k [8] device family the JTAG interface is used for configuring the device. In addition to this, Altera provides the capability of adding Embedded Logic Analyzer (ELA) cores called SignalTap Megafunction [9] [10] into the design to improve the observibility and execution control of the design. The user can communicate with the ELAs added to the design through the JTAG interface. These ELAs and the capabilities that they support are discussed in more detail in the following sections.

A similar type of capability is also supported by the Virtex™ FPGAs from Xilinx through the JTAG interface. Virtex™ devices use the JTAG port for configuration readback and write back. In addition to these features Xilinx also supports the addition of Integrated Logic Analyzer cores using the software called ChipScope [11] [12] into the designs which have quite powerful capabilities for debugging designs. These capabilities are discussed in detail in a separate section on the various software tools that are available for debugging FPGA Based designs.

2.5.2 FPGA Based Boards that support Debugging: There are many FPGA Based boards that have been developed which support different type of debugging capabilities. A brief discussion of these systems and the debugging capabilities that they support is discussed below.

Designs that are implemented on FPGA based boards are implemented in a Hardware Description Language and simulated using a simulation environment and once verified are implemented on the given target platform. In order to communicate with the FPGA devices on the board software support needs to be provided so that the features supported by the device like configuration readback, partial reconfiguration, clock control like single and multi stepping of clock, etc can be supported at board level. In order to support these features at the board Application Program interfaces (API) are developed in languages like C, C++, Java and through the API calls these features described above are supported.

There are many FPGA based Boards developed that support several debugging capabilities. The prominent among them are Wildfire [13], Pamette [14], Splash [15], Splash 2 [16], Teramac [17] and SLACC[18]. All these boards support configuration readback and write back. In addition the Wildfire and Splash 2 boards provide capability for relating the extracted symbols in the design to their values in the read back bitstream which is useful when debugging designs. The SLAAC series of FPGA boards support features like single and muti stepping the clock which is an important feature for supporting execution control and partial reconfiguration can be utilized for making changes in the design at runtime which can be used to improve the controllability of designs.

2.5.3 Hardware Debugging tools :

There have been many hardware debugging tools both commercial and academic that have been developed that support excellent debugging capabilities. These include

Signaltap Megafunction from Altera, ChipScope Logic Analyzer from Xilinx, JHDL Design Environment developed by researchers at BYU [19] and BoardScope [20] from Xilinx. Each of these tools and the features they support is described below.

SignalTap Megafunction:

The SignalTap megafunction is an Embedded Logic Analyzer (ELA) developed by Altera for the APEX II and APEX 20K devices (including APEX 20K, APEX 20KE, and APEX 20KC devices), which can be added to the design to improve the observibility of the design. The user can communicate with this ELA through the JTAG Boundary Scan interface of the FPGA. It can basically be considered to be a piece of hardware that is added to the design before the design is placed and routed by the placement and routing tools.

The ELA consists of three ports i.e. the debugging port, the ELA port and the triggering port. The debugging port connects the ELA to the data signal in the design that are captured on the occurrence of certain trigger condition. There can be several debugging ports in the design based on the number of data signals that need to be monitored. The debugging port can be connected internally to the logic analyzer so that the data can be captured on the occurrence of certain trigger condition or it can also be connected to the I/O pins of the FPGA so that data can be saved in a buffer external to the device if the resources on the device are scarce. On the FPGA the storage buffer is implemented using the Embedded System Blocks (ESB) which are the block Rams available inside the FPGA. Fig 2.8 gives an overview of the various components of the SignalTap MegaFunction ELA.

The ELA port is used for communicating with the ELA. In addition to the JTAG interface USB and other parallel ports can also be used to communicate with the ELA. An ELA functions in three different modes. They are the Run mode, the Autorun and Stop mode. In the Run mode data is saved in the buffer on the occurrence of the specified trigger and the ELA goes into the stop mode where the ELA is not active. In the case of Autorun mode the ELA continues sampling the data on the occurrence of the trigger condition and it continues doing so until the status is changed to the stop mode.

The list of possible trigger conditions that are supported by the ELA include the rising edge, the falling edge and the level trigger conditions. There are three different positions at which the trigger can be placed when saving the data on the occurrence of certain trigger condition. Since the buffer used is a circular buffer the trigger can be placed at 12%, 50% of 88% of the trigger point i.e. for instance in case of a 12% trigger condition, 12% of the data saved in the buffer is before the occurrence of trigger condition and 88 % comprises of data after the occurrence of the trigger condition. The trigger signals are connected to the ELA through the trigger ports and if necessary the trigger single signal can be connected to the I/O pin of the FPGA informing the user on the occurrence of certain trigger condition. But the trigger port itself cannot be connected to the I/O pins of the FPGA. The ELA cores are added to the design before placement and routing the design after the design has been synthesized by the synthesis tool. Thus no changes need to be made in the HDL source code inorder to add the ELA into the design. Modifications can be made to the ELA cores that are already present in the design without repeating the process of placement and routing the design. These changes include changing the trigger pattern, new signals can be connected to the ELA or the mode of the

ELA can be changed. But if the designer wants to increase the depth of the buffer, the width of the trigger bus, or the width of the data bus, these changes can be incorporated only by repeating the entire process of placement and routing as significant modifications have been made to the ELAs.

ChipScope:

Similar to Altera, Xilinx also provides the support to add Integrated Logic Analyzer (ILA) Cores into the design through their hardware debugging tool called Chipscope. This feature is supported for Virtex and Spartan II family of FPGAs. The different features of the ILA cores are as follows

There are from 1 to 256 user selectable data channels.

1. At a given time upto 15 independent ILA cores can be added to the design.
2. The trigger bus can be separate from the data bus with its width ranging from 1-64 bits.
3. All trigger and data operations are synchronous to the user clock.
4. The user can set multiple trigger setting simultaneously.

Using Chipscope ILA cores can be added to the design. A GUI is provided using which the user can specify the various trigger and data signals that need to be monitored, the depth of the storage buffer, the width of the data and trigger signals, etc. The Chipscope tool includes the Core Generator which is used to generate ILA cores depending on the various requirements supplied by the user through the GUI. The ILA cores generated can be in the form of an Edif netlist or the VHDL components, which can be added to the design at various stages of the design process. Once the ILA core is

generated it can be added to the design using a Core Inserter before or after synthesis. Adding the ILA core before synthesis is accomplished by instantiating the HDL

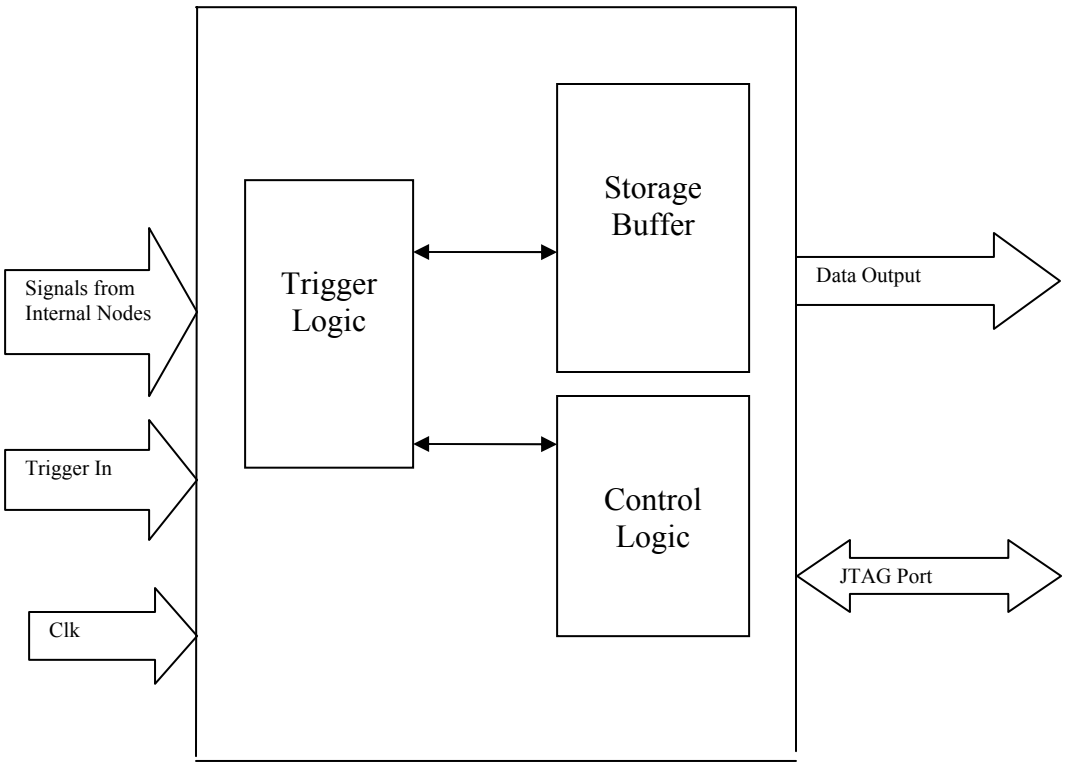


Fig 2.8 Different component of the ELA core [9][10]

components into the HDL sources code of the design. One can also add the ILA cores after synthesis directly into the Edif netlist of the design before placement and routing of the design begins.

The different components of Chipscope toolkit include the ICON and the ILA cores. ICON is the integrated logic controller which controls the operation of the various components of the ILA. The ILA core incorporates the hardware which contains the trigger logic and it communicates with ICON on the occurrence of certain trigger

condition so that the values on the data signals can be saved into the storage buffer. The triggering can occur on the rising or falling or both edges of the signal or at a stable value. The trigger logic can work in basic or extended trigger mode. In the case of basic mode there can be a single match unit or at the most two match units. For instance when the designer is interested in checking whether a signal lies in a particular range two match units are used. In the extended mode in addition to the individual match units for each ILA the output of each ILA triger unit is 'Ored' to give the overall trigger signal. This is useful when one is interested in relating the various trigger signals in the design. The data saved in the storage buffer can be analyzed using the Chipscope Analyzer which is a part of the tool. The analyzer comprises of a waveform viewer which displays the content of the storage buffer. The Chipscope Analyzer can communicate with the PC or workstation through the Xilinx MultiLINX™ and Parallel Cable III download cables.

Similar to Altera's SignalTap megafunction, the Chipscope tool allows small changes in the ILA core like the changes in trigger condition, the control and data signals being monitored without repeating the process of placement and routing of the design. For making significant changes like the alteration in the depth of the buffer, the width of the trigger and data ports, the process of Placement and Routing has to be repeated after making the necessary alterations at the HDL source file or the EDIF netlist stage.

JHDL Design Environment for Debugging: There has been a lot of research being done by researchers at BYU [21] [22] to improve the debugging capabilities of designs implemented on FPGA Custom Computing Machines (FCCMs). We will try to give a

brief description of the JHDL design environment that they have developed and how it has inspired the work that has been done in this thesis.

JHDL is a simulation environment that provides Java Based APIs. Designs implemented on FPGA based systems can be described and simulated using the tools supported by the JHDL Design Environment. Only structural designs can be described using JHDL which treats each of the components of the design as a java object. The researchers at BYU have developed a framework for integrating hardware execution and simulation [23] [24]. Here the information of the mapping of the logical design to the physical FPGA resources is stored in the form of a symbol table using the Java based APIs and once this is done the user can execute the design in a unified framework switching back and forth between hardware execution and software simulation when desired based on the requirements. For instance when the designer has reached a particular phase of the design execution where excellent observibility is desired then he/she can switch from hardware execution to software simulation which provides excellent observibility. The creation of a symbol table from the information of logical to physical design mapping which is a part of the process of developing an integrated debugging environment is also useful for many other purposes as will become clear in subsequent chapters.

In our thesis we have also developed an algorithm for the creation of a symbol table which contains the information of the logical to physical design mapping so that the user can be provided with a logical view of the design even after the design has been mapped to the FPGA and he/she doesn't need to worry about the mapping information as the symbol table contains this information. So whenever the designer refers to a particular

logical component in the netlist the correspondent resource in the FPGA is referred to using the information in the symbol table which is desired as the user is familiar with the logical design.

In addition to this work on Logical to Physical Design mapping there has been a considerable amount of work done at BYU on improving the debugging capabilities of design through the addition of debugging logic for better controllability , observibility and execution control [25] [26][27] of the design. In this work they have demonstrated the various stages of the design process where the debugging logic can be added and the trade offs involved.

Since there are some similarities between our work and the work done at BYU so we will try to explain the various aspects in which our work differs from theirs. The work at BYU targets the JHDL design environment whereas our design environment supports general HDLs like VHDL or Verilog which are the industry standard and used by most of the designers. Also to our knowledge in their work the debugging logic is added to the design through the logical database before the process of Placement and Routing, whereas we have been able to add the debugging logic directly at the bitstream stage using JBits and JRoute which is one of the main contributions of our work.

BoardScope:

There has been considerable amount of work reported in the literature [28] [29] [30] describing the various capabilities supported by Boardscope, a tool developed by Xilinx for debugging designs. After the bitstream is modified using JBits and JRoute the new bitstream generated can be debugged using BoardScope [31]. It comes as a part of JBits

package. A view of the BoardScope interface is given in Fig 2.9 which can be used to view the configuration information in different views which includes Core view, State View, Power View and Routing density View. Using Boardscope one can communicate with the design implemented on the FPGA board through the Xilinx hardware interface. BoardScope can be used to invoke the Virtex device simulator which is a simulation tool used to simulate designs implemented on the Virtex FPGAs. In this thesis we have used this Virtex Device Simulator to verify the functionality of the designs after the addition of Integrated Logic Analyzer cores. Virtex device simulator provides feature for debugging designs like single stepping or multi stepping the clock, reading back the content of Flip Flops, IOBs and Block Rams during simulation, etc. The designer can also use the command line interface for controlling the simulation of the design using DDT Script commands which are a part of the BoardScope interface. The current version of BoardScope is slow for complex designs.

Other Hardware Debugging Tools:

In addition to the debugging tools discussed above there are many other debugging tools that have been developed over a period of time. The prominent among them are the InnerView hardware debugger from the virtual wires group at MIT[32], the Splash reconfigurable coprocessor board which also supports a runtime debugging environment called T2 [16], the Teramac [17] FPGA based system developed at the HP laboratories also has a debugging tool using which the state of the resources inside the FPGA like flip flops, LUTs, etc can be read back. My colleague Anurag Tiwari has also studied different

techniques [33] [34] to add small and compact debugging logic cores called hardware watchpoints to speed up the debugging process of complex designs. He has also been

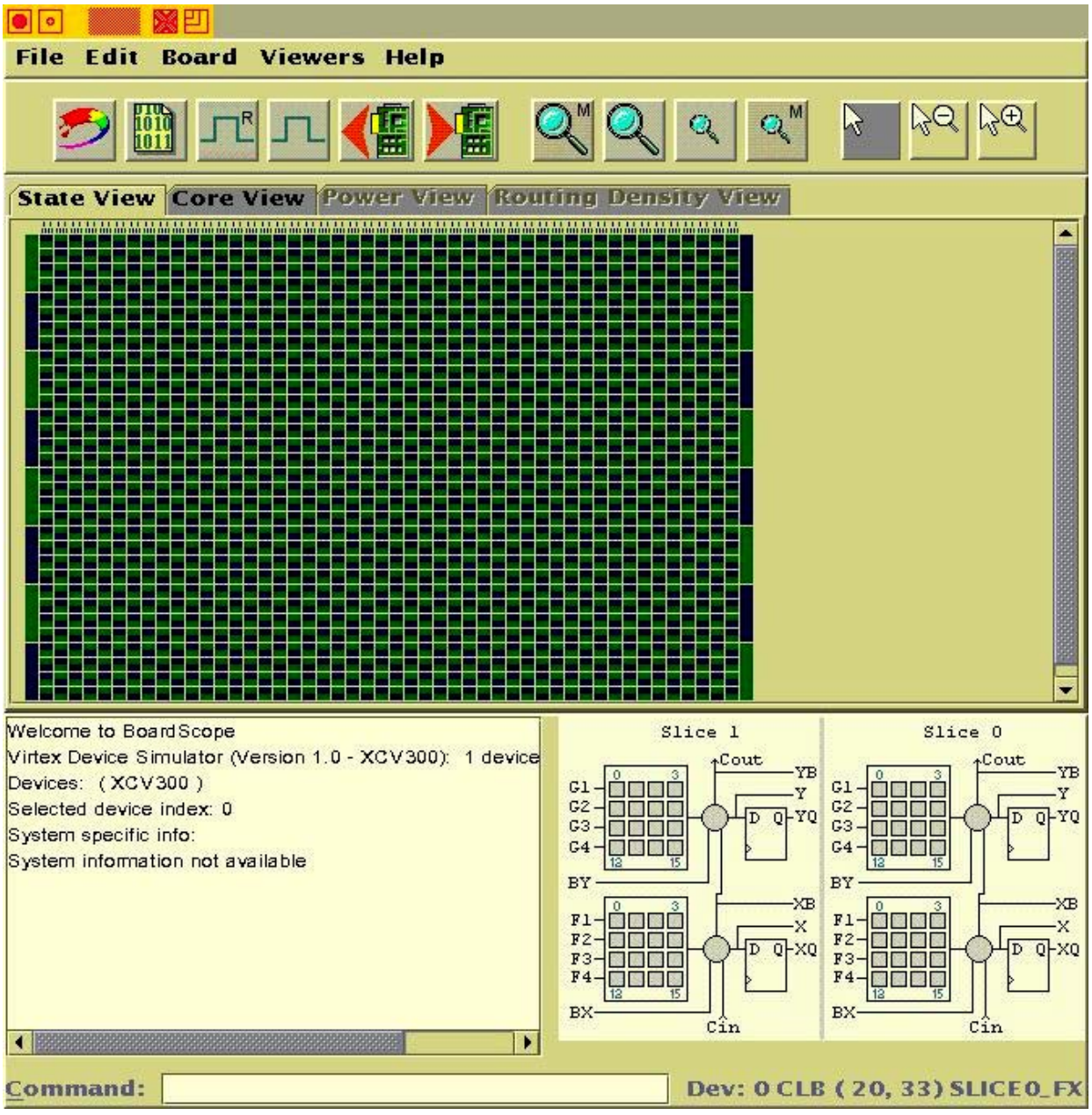


Fig 2.11 The BoardScope Interface

working on using the features of FPGAs to support debugging, such as LUTs being used as shift registers to add and make changes to the watchpoint logic at runtime to improve the debugging capabilities [35].

In this chapter we have given an overview of the various features of the Virtex™ FPGAs from Xilinx which can be used for improving the debugging capabilities of designs. In addition to this some FPGA based Hardware debugging systems and tools relevant to the work done in this thesis which have been developed over the past decade were also discussed.

Chapter 3

Process of Symbol Table Creation

In chapter 1 we have explained the importance of the process of logical to physical design mapping for the creation of a symbol table. This idea of symbol table creation was first proposed by researchers at BYU [24].As discussed this will help in providing the designer with a logical view of the design which the designer is familiar with, instead of the physical view after the design has been mapped to the FPGA. This is the first step towards the process of improving the debugging capabilities of the design by adding the Integrated Logic Analyzer (ILA) cores to the design at the bit stream level. Due to the presence of the mapping information in the symbol table the debugging environment can provide support for various debugging capabilities like selection of signals to be monitored by connecting them to the ILA, flip flops whose content is to be readback to improve the observibility of the design, etc using the logical view and the circuit modifications to support these capabilities can be carried out at the bitstream level using the bitstream modification tools such as JBits and JRoute.

3.1 Introduction:

The process of Logical to Physical design mapping for creating the symbol table is quite involved. In this chapter we will try to give a detailed explanation of this process for Virtex™ FPGAs and the issues involved. A similar mapping process is described in ref [24] and [25]. We will start with an overview of this process and then try to explain each step in detail. This process of Symbol table creation can be broadly divided into three steps which are as follows

1. The process of extracting the relationship between the FPGA state elements such as flip flops, LUTs, Block Rams, IOB, etc. to the location of their state in the readback bitstream.
2. The process of mapping the physical FPGA elements to their counterparts in the logical netlist.
3. The tracing of the LUT Ram address permutations and taking these permutations into account while determining the location of the LUT Ram state in the read back bitstream.

This process of logic to physical design mapping can be accomplished by using the information form various files generated by the synthesis and the Xilinx Placement and Routing tools. The table in Fig 3.1 gives a brief description of each of the files that will be used in the creation of the symbol table and their content and they are explain in detail below.

a. **.edf:** This EDIF (electronic design interchange format) file is generated by the synthesis tool. It is a standard format used to represent the logical netlist generated by the synthesis tool.

b. **.mrp:** This map report file is generated by the Xilinx Mapping tool. The ".mrp" file contains information of the various design optimization that have been performed by the mapping tool. It also provides information regarding the relationship between the logical instances and the FPGA resources used in implementing the design.

c. **.ll:** This logic location file is an ASCII representation of information of the corresponding flip flops and Ram state values in the readback bitstream. This file is

generated bt “bitgen” which is the bitstream generation program provided as part of the Xilinx CAD tool kit.

d. **.ncd:** This native circuit description file provides the information of all the used resources in the FPGA for implementing the particular design. As this file is in a non readable format Xilinx provides a utility called “ncdread” which can be used to generate a textual description of this file in ASCII format.

File name	Content of the file
.edf	It generated by the synthesis tool and contains the list of all the logical instances in the design
.ll	The logic location file which contains the information of the Sampled FPGA state in the readback bitstream
.mrp	This file contains contains information of the mapping of the the Logical instances in the design to the physical resources in the FPGA and the optimizations performed on the logical design.
.ncd	this is binary file which contains information of the resoures in the FPGA that have been used for the implementation of the given circuit
.ncdtext	This file is an ASCII representation of the FPGA resources used for implementing the given design.

Fig 3.1 List of files used for symbol table creation

3.2 Process of Logical to Physical Design mapping:

This section will describe the process of Logical to Physical design mapping in detail. This process will lead us to the creation of the symbol table that we refer to as “.rbstable”. We will start with a detailed description of the information that

is contained in each of the files and how this information is utilized for constructing the symbol table.

3.2.1 EDIF netlist file (".edf"):

The first file we will describe is the ".edf" file generated by the synthesis tool. This file contains information of the logical instances (which are the components from the FPGA design library used by the synthesis tool to implement the design) in the design and the way they are inter connected to each other to form the netlist of the design. We used an ".edf" parser which was developed at Rice University for a project on "Optimizing VHDL Intermediate Representations" [37]. The parser is available as open source and we have customized this parser for our research. A hash table called ".instancehash" is generated by parsing the .edf file. This hashtable is keyed on the name of the instances in the design. Each entry in the hashtable contains the name of the instance, the name of the each port of the instance and the direction of the ports (whether input or output or both) and the net connected to each of the ports. Fig 3.2 gives some sample entries of this hashtable. This includes an instance which is mapped to a flip flop and another instance which is mapped to a LUT. Fig 3.3 gives a block diagram of the ".instancehash". The information in this hashtable will be used for achieving the one to one mapping of the logical instance to the physical components in the FPGA.

3.2.2 Information extracted from Logic Location (.ll) file: The logic location file is used during the construction of the symbol table inorder to trace the position of the state

for various resources which includes Flip Flops, Block Rams and LUTs in the readback bitstream. Once a one to one mapping of the

A_BLOCK/ACOUNT/HRS_OUT[3] FDPE

ah(3) Q OUTPUT

hrs_out_8(3) D INPUT

CLK_c C INPUT

RESET_c PRE INPUT

A_BLOCK/ACOUNT/un7_inc_mins LUT4

am(5) I3 INPUT

am(4) I2 INPUT

G_31 I1 INPUT

un7_inc_mins O OUTPUT

un7_inc_mins_1 I0 INPUT

Figure 3.2 Sample entries of the .instancehash file

logical instances and the Physical FPGA resources is established the information in the ".ll" file will assist in tracing the location of the data read back from each component. There are several issues that are to be considered and resolved when using the information provided in the .ll file to construct a hashtable. We will explain in detail the

possible entries in a logic location file, the information available in this file and how this information is used.

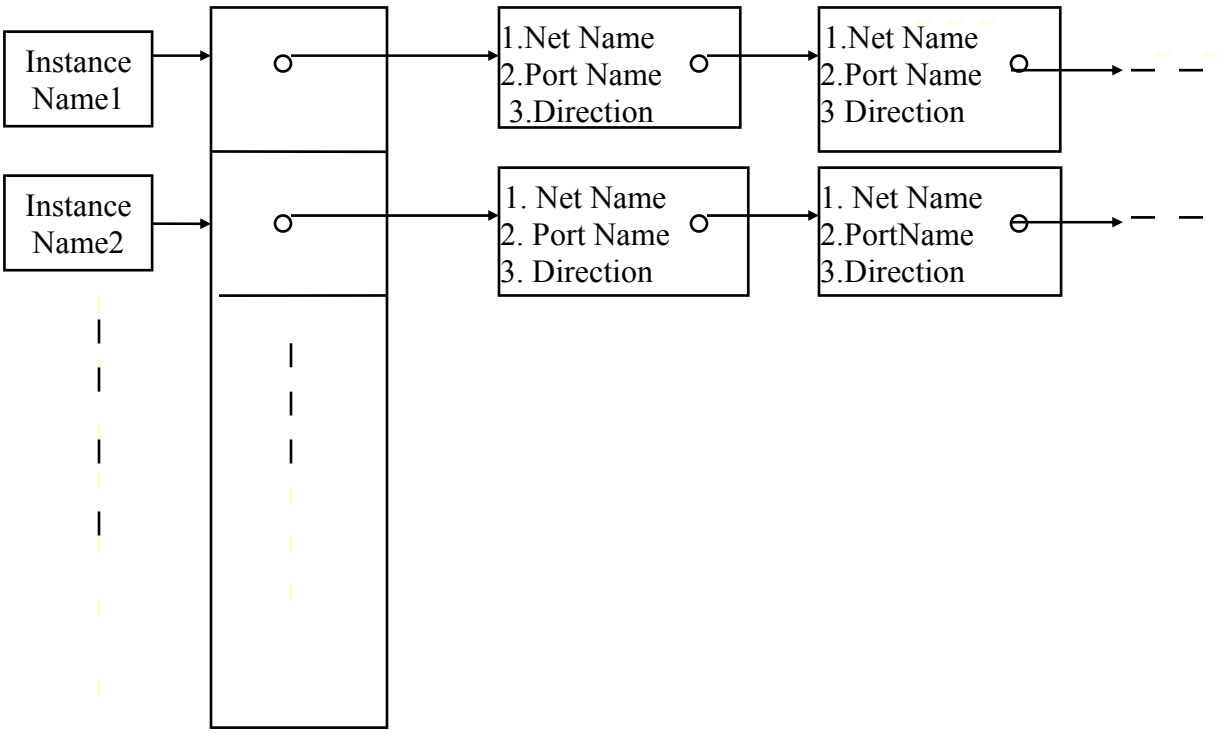


Fig 3.3 Block Diagram of the ".instancehash"

A list of sample entries of the Logic Location file is given in Fig 3.4. The first entry is the state of output flip flop of IOB AK25. As discussed in section 2.1.1 each Virtex CLB contains two slices designated by S0 and S1, each containing pair of Flip Flops and LUTs. The second entry in the sample .ll file gives the position of the FFY flip flop of CLB located on row 60 and column 16 in slice S0 . The other entries are for Block Rams and LUTs used as Ram. Here the .ll file provides information that can be used to calculate the position of the content of each Ram bit in the bitstream.

As discussed above the different possible readback entries include the state of latches and the content of LUT and Block Rams in the case of Virtex™ FPGAs. The

possible latch entries in the .ll file are given in Fig 3.5. They include the I,T ports which are the outputs of the IOBs and the IQ, O ports which are the inputs to the IOBs. In addition to this the latch entries include the state of flip flops which are a part of the Virtex™ CLB slice. The detailed functional diagram of the IOBs and CLB and block Ram can be found in reference [1].

Offset framenum frameoff

Bit 3938642 0x00843400 1170 Block=AK25 Latch=O
Net=b14_comp/addr_35_enl(18)
Bit 3954777 0x00844e00 1081 Block=CLB_R60C16.S0 Latch=YQ
Net=b14_comp/reg3(8)
Bit 204927 0x00b01000 160 Block=CLB_R9C6.S1 Ram=M:16
Bit 6122057 0x02027c00 585 Block=RAMB4_R7C1 Ram=B:BIT2027
Bit 6122058 0x02027c00 586 Block=RAMB4_R7C1 Ram=B:BIT2023

Fig 3.4 Sample .ll file entries

The process of relating LUT and Block Rams to their state in the readback bitstream is very involved. We will start by describing the different possible configuration in which the LUTs and Block Rams in the design can be used. The LUTs in the CLB slices can be used as individual 16x1 single port Rams or the two LUTs combine to form a single 32x1 single port Ram depending on the requirement. In case of

Virtex™ series of FPGA the F4 port represents the MSB of the address line and F1 represents its LSB. If the two LUTs are combined together to form a 32x1 Ram then the F LUT holds the lower 16 bit of data and the G LUT holds the higher 16 bit of the data and the BY port which is the input to the slice acts as the MSB of the address bus. If the LUTs in the slice are used as two individual 16x1 Rams that the entries in the .ll file are represented as Ram=F:<address> and Ram=G:<address> where the address ranges from 0 to 15. In the case when the two LUTs are combined to form a 32x1 Ram the Ram entries are represented as Ram=M:<address> where the address ranges form 0 to 31.

Possible latch entries	Explanation of each type
IQ	This is the input to the user circuit which is the Output of the IOB resgister in the FPGA
O	The output of the IOB (it's the output form the user circuit)
I	It's the input to the user circuit which is the output of the IOB
XQ, YQ	These are the output of the flip flops in the CLB slices
T	The IOB output which is the output form the user circuit. It's a tristate flipflop

Fig 3.5 Possible latch entries for Virtex device

As discussed in chapter 2 the block Rams are additional memory resources provided in the Virtex™ series of FPGA. The entries of the Block Rams are also available in the .ll file. The Block Rams can be configured with different aspect ratios

such as 4096x1 bit, 2048x2 bit, 1024x4 bit, 512x8 bit and 256x16 bit based on the users requirement.

The Xilinx Placement and Routing tools perform address signal permutations during the process of Placement and Routing for Rams implemented using the LUTs in the slices inorder to improve the routing. Fig 3.7 depicts possible address signal permutation that can be performed by the Xilinx PAR tool. In 3.7 (a) the logical instance has the address signal with F4 being the MSB and F1 being the LSB. But inorder to reduce the congestion during routing the Xilinx PAR tool might permute the signal as shown in Fig 3.7 (b). One need to take this permuation into account when performing the task of calculating the bit position of each Ram state. Fig 3.7 (c) depicts the mapping of the logical address to their physical counterparts inorder to relate the logical Ram state to their Physical counterpart. In case of block Ram, the Xilinx placement and Routing tool do not perform any address signal permutations due to the abundance of Routing resources.

A parser has been implemented for parsing the .ll file so that the information in the .ll file can be used during readback. Towards achieving this goal two hashtable are generated from this process. The first hashtable called ".rbblockhash" contains the information of all the latches in the FPGA that have been used in the design which includes the latches in the IOBs and the CLBs. The second hashtable called ".rbramhash" contains the readback information of the LUT and BlockRams used in the design. Each entry in the .ll file, as observed in Fig 3.4 includes the absolute bit offset (ao), the frame number (fn) and the frame offset (fo). This information along with the

information in references [4][38] is used to calculate the exact location of the state of the respective elements in the readback bitstream.

The formulae used to calculate the absolute position of the state bit in the readback bitstream for all the Rams and latches in the design is given in Fig 3.6. The different terms used in the formulae include the frame length *fl*, the frame number *fn* the bitmap length *bm* which varies for various Virtex™ devices as given in reference[4], the frame offset *fo* and the pad word length *pwl* which is 32 for the Virtex™ devices. The values of 63 and 93 used in the formulae for calculating the actual bit position for BlockRam came from reference [38] as this did not appear in the Xilinx documentation.

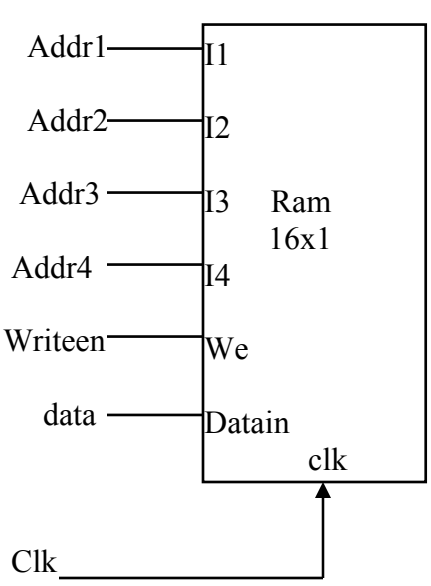
Actual position $= fl * fn + bm - fo + pwl$
(if the component is in CLB or IOB)

Actual position $= fl * (fn - 63) + bm - fo + pwl$
(if component is in Blockram column 0)

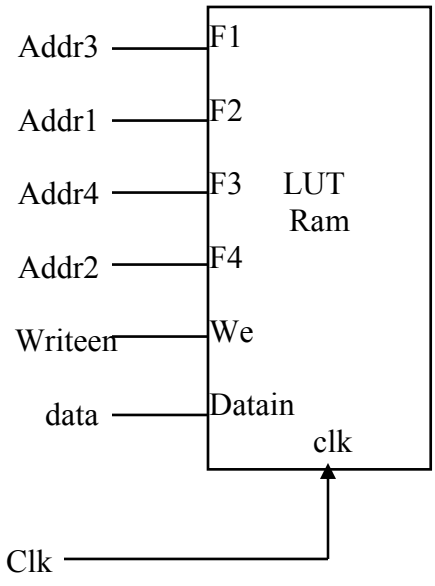
Actual position $= fl * (fn - 93) + bm - fo + pwl$
(if component is in Blockram column 1)

Figure 3.6 The formulae used to calculate the postion of bits in the readback bitstream

Once the bit position of each Ram Bit and latch in the readback bitstream is know the ".rbblockhash" and the ".llramhash" are created with the necessary information. Fig 3.8 represent information contained in the ".rbblockhash" for latches in the design and Fig 3.9 the information for the Ram entities. Thus these hashtable provide comprehensive



(a) Logical Instance



(b) Physical Ram Counterpart

Logical Address	Physical Address
0000	0000
0001	0010
0010	1000
0011	1010
0100	0001
0101	0011
0110	1001
0111	1011
1000	0100
1001	0110
1010	1100
1011	1110
1100	0101
1101	0111
1110	1101
1111	1111

(c) Process of Ram Address Mapping

Fig 3.7

information that is utilized to relate the Latch and Ram entries to their state in the readback bitstream.

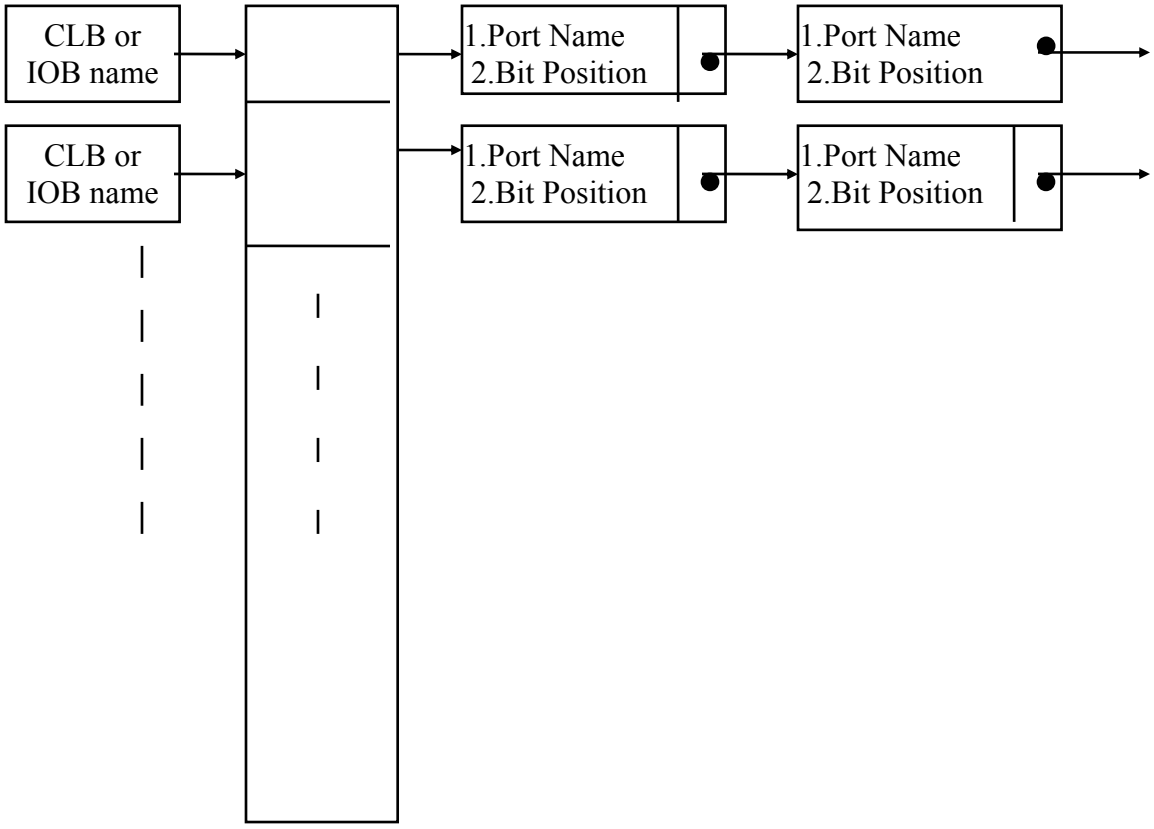


Fig 3.8 Block diagram of readback Block hash

3.2.3 Considering the Design Optimization (.mrp):

The map report file (.mrp) file generated by the Xilinx Cad tools contain information of the various design optimization that have been performed by the mapper while mapping the logical design to the physical FPGA resources. The "Merged Signal" section of this ".mrp" file contains information of the nets that have been removed or merged with some other nets during this process. Fig 3.10 shows some sample entries of

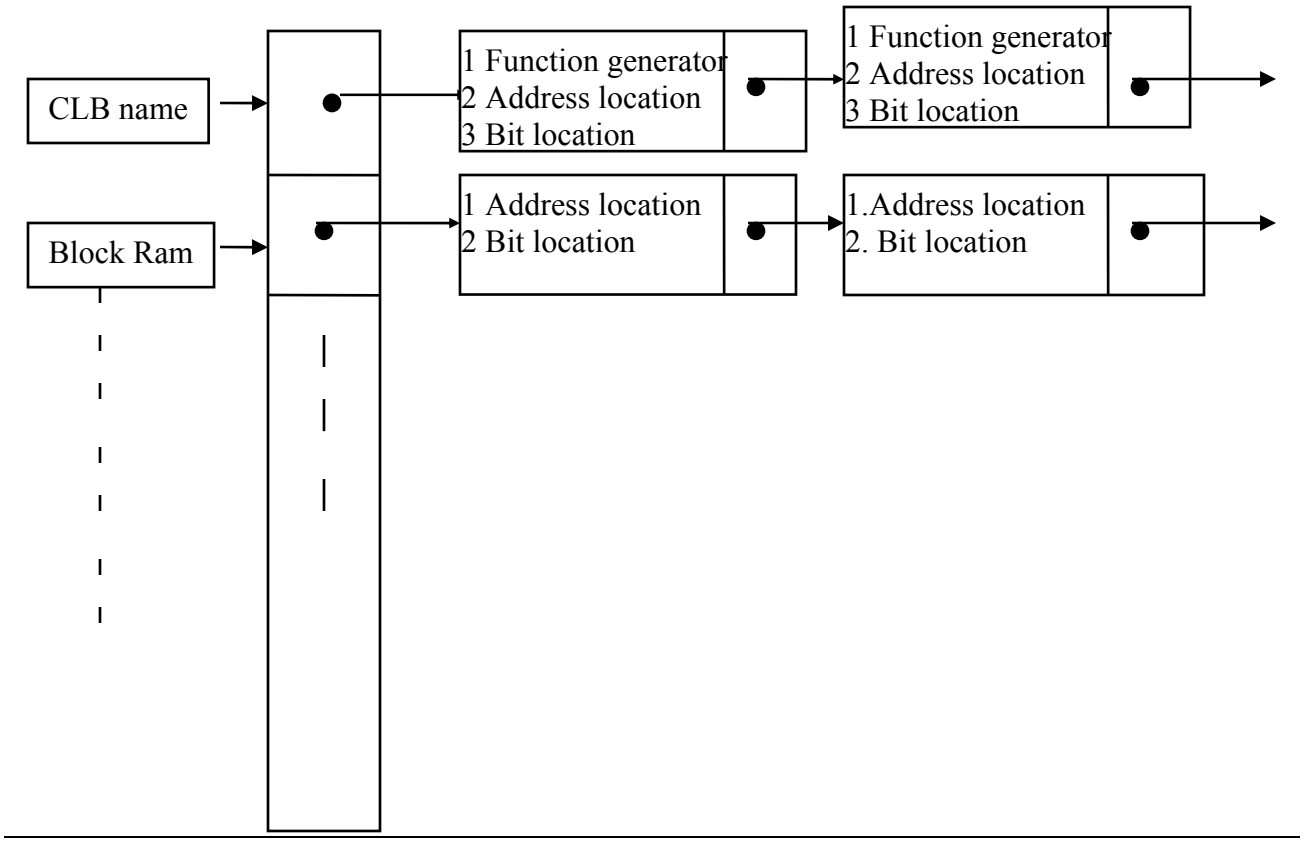


Fig 3.9 Block Diagram of readback Ram hash

this file where the information of the merged nets is provided. The possible net optimization that can be performed include the merging of two nets, for instance net n1 can be merged with net n2 . It may also happen that net n1 is merged into n2 and net n2 is merged into n3 so after optimization only n3 is left . So inorder to keep track of these optimizations we have implemented a parser for the map report file which creates a hash table called ".nethash" with each entry keyed on the old net and the nets that have been optimized will contain the name of the new net into which it has been merged as the entry. The algorithm used to keep track of these optimizations and the information

Merged Signal(s):

-The signal "PrgmCntr/un3_stack1_cry_2" was merged into signal "PrgmCntr/un3_stack1_cry_2/O".

-The signal "PrgmCntr/un3_stack1_axb_1" was merged into signal "pc(1)"

-

-

Symbol Cross-Reference:

- "DECODE/ALUOP[0]" (FDP) mapped to: aluop(0) (SLICE)

- "porta_iobuf[0]" (IOBUF) mapped to: porta(0) (IOB)

Fig 3.10 Sample entries of Map report file

contained in the hashtable generated is shown in Fig 3.11 and Fig 3.12 respectively. The algorithm implemented handles only two levels of merging, as it was observed from the map report files generated for different designs that the Xilinx CAD tools at the most performs two levels of merging. But it is not difficult to implement an algorithm which can handle more than 2 levels of merging.

The second hashtable generated from the map report file is called the ".instancemaphash". This hash table contains the information of one to one mapping of the logical instance in the design to the corresponding physical FPGA resources which is extracted from the "symbol cross reference" section of the map report file. The block diagram in Fig 3.13 depicts the information contained in this hash.

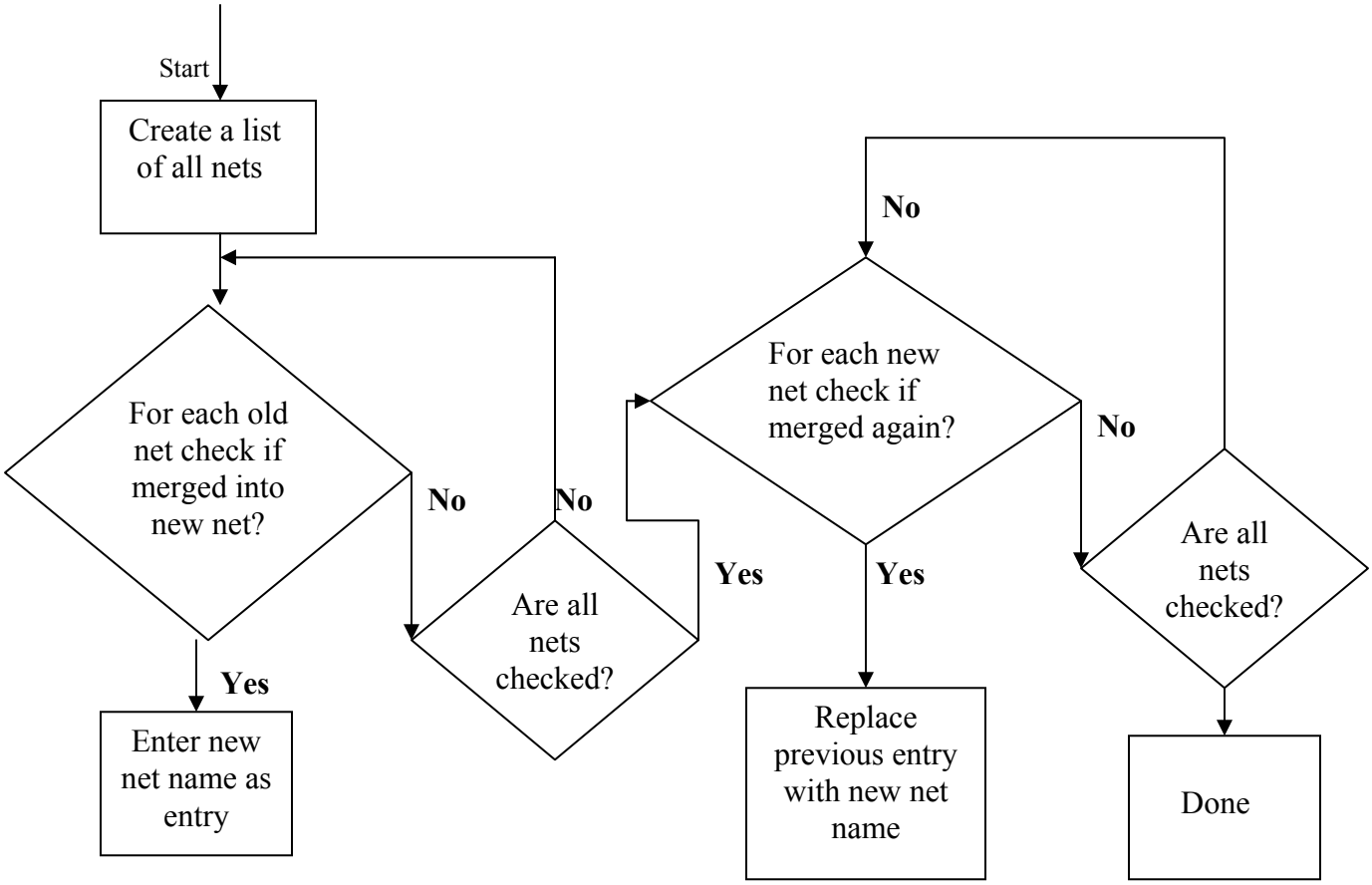


Fig 3.11 Algorithm for keeping track of Net Merging

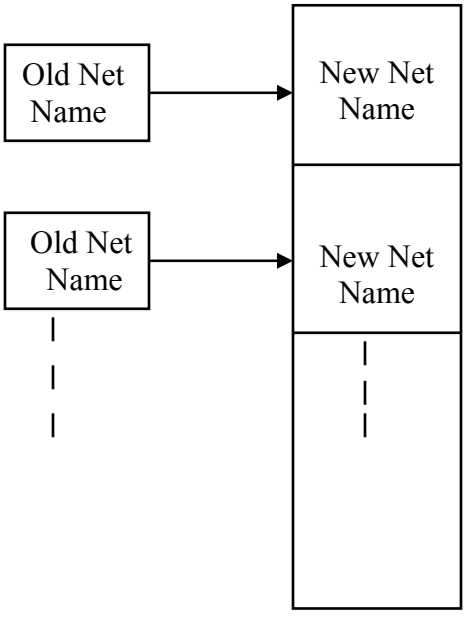


Fig 3.12 Block Diagram of Nethash

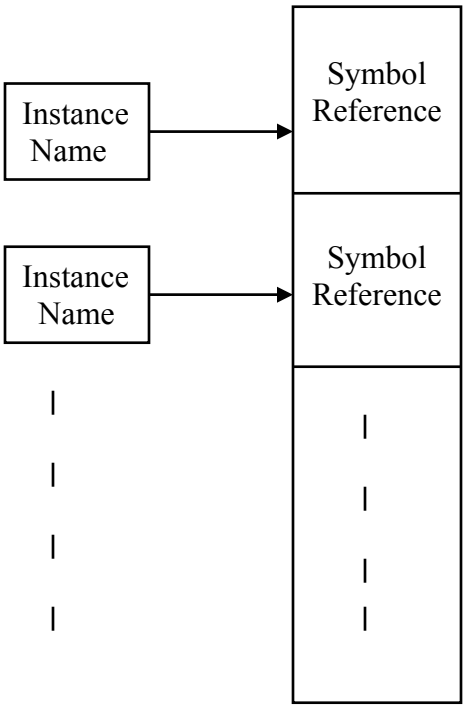


Fig 3.13 Block Diagram of Instancemaphash

3.2.4 Tracing the FPGA resource usage information (Native Circuit Description):

The native circuit description (".ncd") file generated by the Xilinx Placement and Routing tools contains information of all the resources of the FPGA used in the design. A textual description of this file is available in ASCII format which is parsed to create a hash table which provides information of the resources used. Fig 3.14 shows a sample entry of this file. It contains the information of the Physical resource which is a Slice S1 of the CLB located at Row 4, Column 13 and the ports of the CLB that have been used in the designs and the nets connected to these ports. Fig 3.15 gives a block diagram representing the information contained in this hashtable.

3.2.5 Putting it altogether: The information extracted from various files as described above is used to create a symbol table called ".rbstable" which provides detailed information of

NC_COMP:2 - <status_z_write> site = CLB_R4C13.S1

Config String: <CYSELF:#OFF CYSELG:#OFF CKINV:1 COUTUSED:#OFF

YUSED:0 XUSED:#OFF XBUSED:#OFF F5USED:#OFF YBMUX:#OFF CYINIT:#OFF

DYMUX:#OFF DXMUX:1 CY0F:#OFF CY0G:#OFF

F:#LUT:D=(~A1~A2*~A3*~A4)+(~A1*~A2*~A3*A4)+(~A1*~A2*A3*A4)*

G:#LUT:D=(~A3~A2*~A1) RAMCONFIG:#OFF REVUSED:#OFF BYMUX:#OFF*

BXMUX:#OFF CEMUX:#OFF SRMUX:SR_B GYMUX:G FXMUX:F SYNC_ATTR:ASYNC

SRFFMUX:0 INITY:#OFF FFX:#FF FFY:#OFF INITX:LOW>

23 pins -

pin 4 - CLK: <clock_c>

pin 6 - F1: <DECODE/G_142>

pin 7 - F2: <DECODE/N_162>

pin 8 - F3: <DECODE/N_259>

pin 9 - F4: <inst(7)>

pin 12 - G1: <inst(9)>

pin 13 - G2: <inst(8)>

pin 14 - G3: <inst(6)>

pin 16 - SR: <resetn_c>

pin 19 - XQ: <status_z_write>

pin 20 - Y: <DECODE/N_259>

Fig 3.14 Sample entry of the ASCII representation of .ncd file

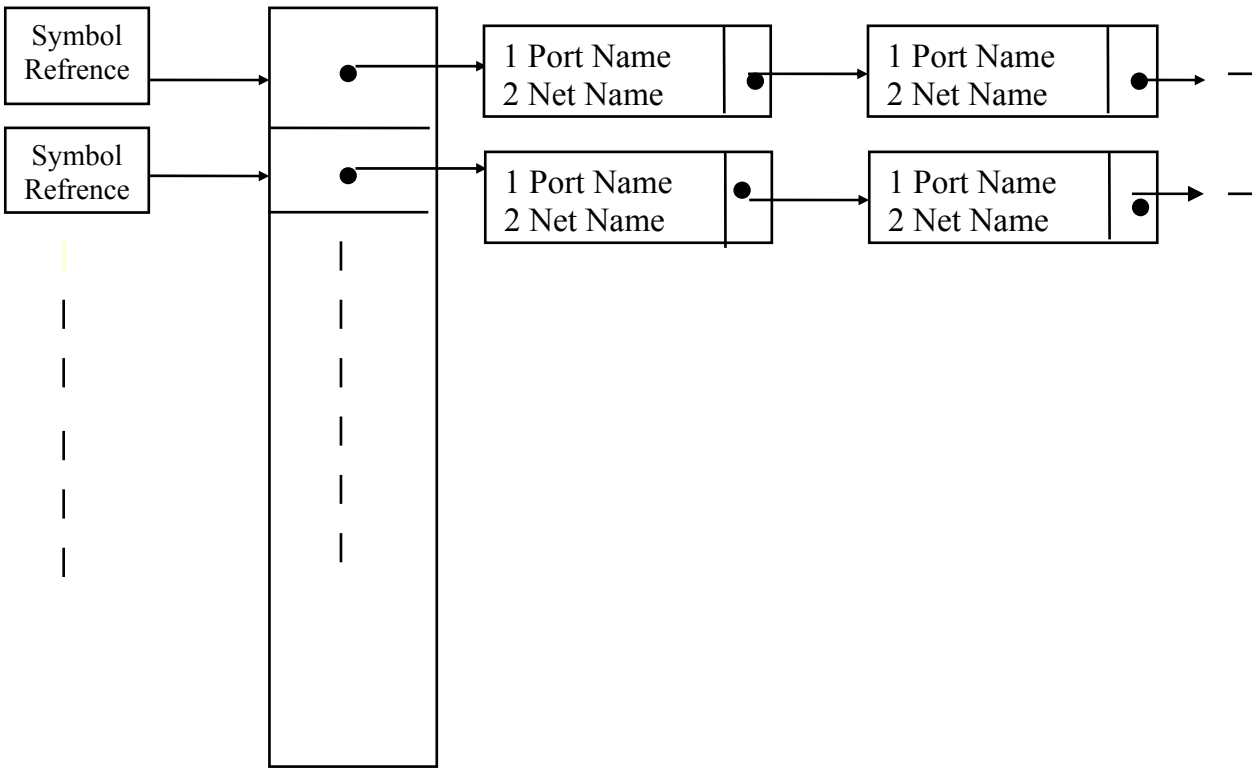


Fig 3.15 Block Diagram of NCD hash

the logical to physical design mapping. The table in figure 3.16 gives a list of all the intermediate hashtables generated during this process and the data contained in each of the hashtables that is used for the creation of final symbol table. This entire process which is depicted in Fig 3.17 begins by mapping all the logical instances in the ".instancehash" to their counterparts in the ".ncdhash" using the information in the ".maphash" and the ".ncdhash" which contains the information of this mapping to create the ".instancemaphash". Once this mapping is done the information in ".llblockhash" and the ".llramhash" is used to relate each of the Latches, Flip flops and Rams in the design

Hashtables generated	Key of the hashtable	Data entry in the hashtable
.instancehash	Instance name	It's a linked list of all the ports of this instances with the Direction of the ports
.rbblockhash	CLB or IOB latch Name	The location of the corresponding state in the readback bitstream
.rbramhash	LUT or block Ram	The location of the corresponding state of each of the Ram entries.
.maphash	Logical Instance name	Output Port of Each of the Corresponding physical resource
.nethash	Old net name	Net net name into which the old Net has been merged
.ncdhash	Name of the physical resource used in the implementing the design	A linked list of all the ports of this resource and the net nets connected to each of the ports
.instancemaphash	Logical Instance Name	Corresponding Phyical resource to which it has been mapped
.rbstable	Logical Instance Name	Corresponding physical instance, the corresponding port mapping, the corresponding net mapping for Rams, CLBs and IOBs.

Fig 3.16 A list of Hash table created during the process of Logical to Physical design Mapping

with the location of their state in the readback bitstream. Inorder to take the address signal permutation for LUTs being used as 16x1 or 32x1 Rams into account the

information from the ".ncdhash" and the ".instancehash" is used to compare the signals connected to the ports of these components and correspondence between the ports is established utilizing the information form the ".nethash" which contains information of

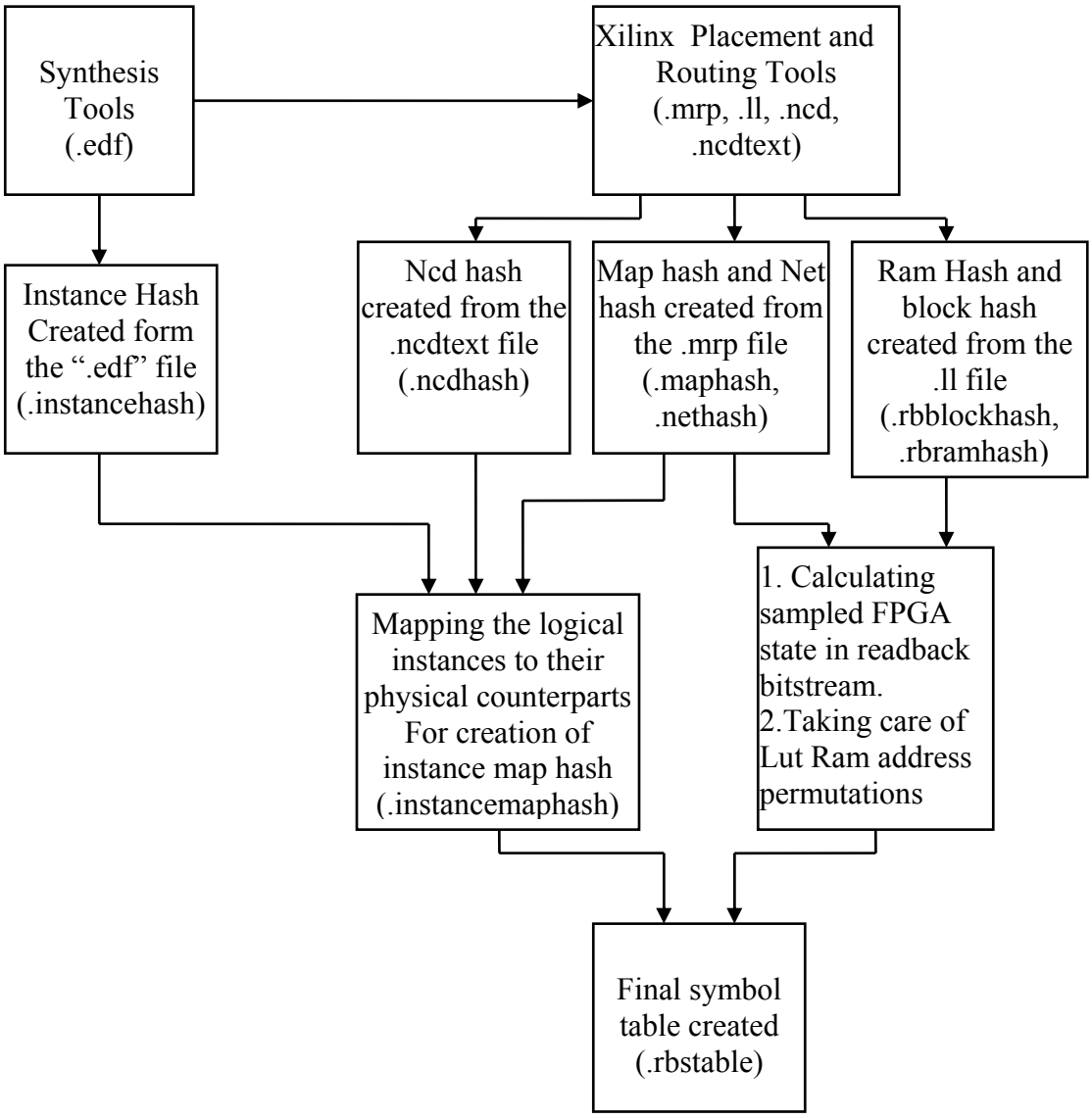


Fig 3.17 Process of Logical to Physical Design mapping

those signal that have been merged. With the information gathered through this procedure a symbol table as shown in Fig 3.18 is created which contains comprehensive information

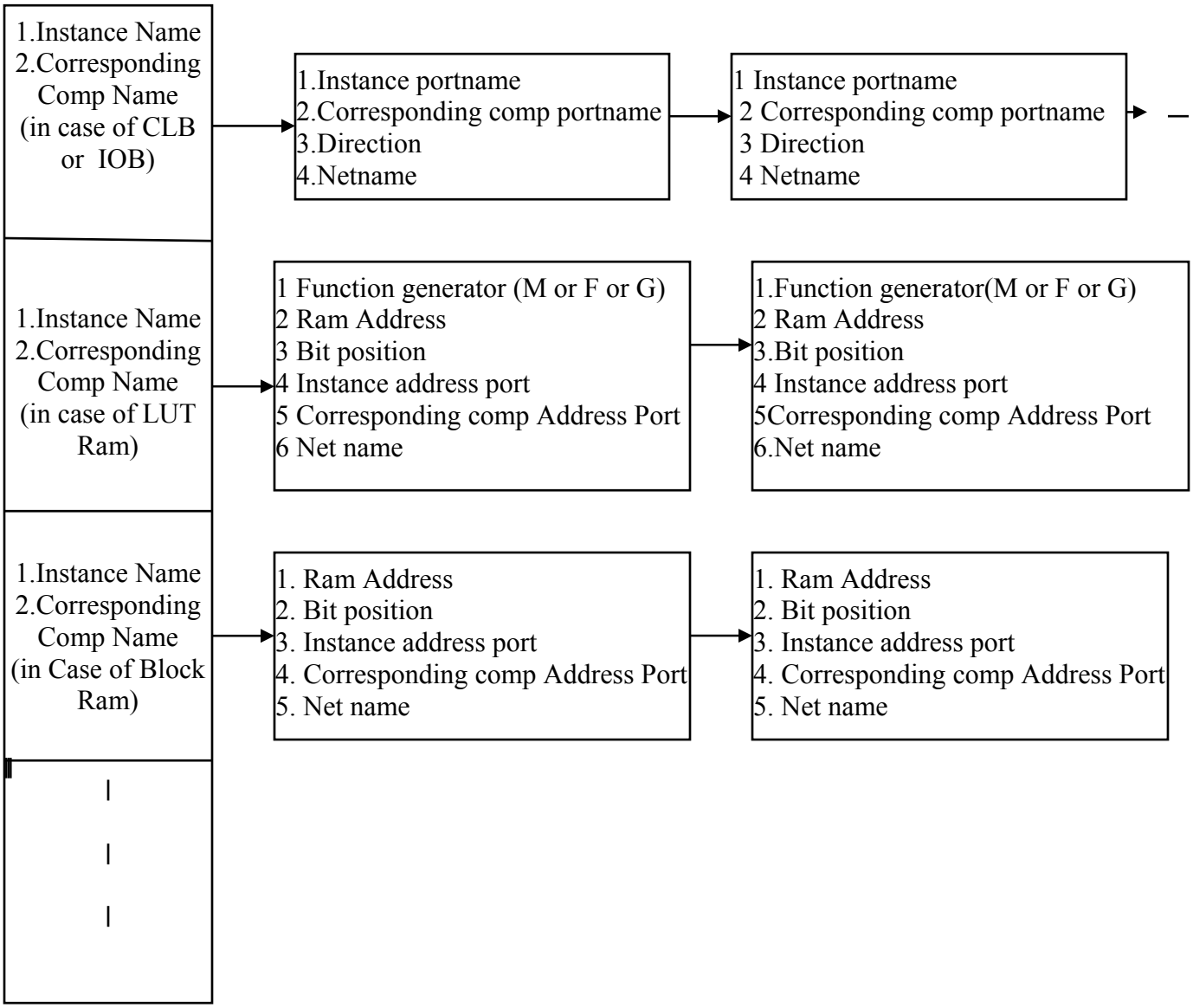


Figure 3.18 Block Diagram of the final Symbol table

of the mapping of the logical instances in the design to the corresponding resources of the FPGA. Some of the entries in the symbol table are empty as those instances in the design

are optimized by the Xilinx CAD tools. This information in the symbol table will be utilized during the process of ILA core addition using JBits and JRoute.

3.3 Limitation of Logical to Physical design Mapping technique: The limitation of this technique lies in the fact that the process of symbol table creation starts at the Edif netlist stage and the design optimizations performed by the synthesis tool are not accounted for by this technique. Some signals may be optimized by the synthesis tools and are not present in the design after synthesis. The symbol able does not contain any information about these. There is a greater chance that designs described at the behavioral stage will have many signal names changed or compared to designs described at the structural level using a HDL language. So the process of symbol table creation which starts after synthesis is more useful fro the case of designs described at the structural level.

Chapter 4

Adding ILA Cores to Improve the Debugging Capabilities of Designs

This chapter will give a detailed description of the process of adding Integrated Logic Analyzer (ILA) cores to the design at the bitstream level using JBits and JRoute, the Java based APIs that can be used to modify designs at this level. The main advantage of adding the ILA cores at the bitstream level lies in the time saved in making the necessary modification to the ILAs depending on the users requirement, such as the change in trigger condition, the change in the depth of the data buffer, etc as these changes are made late in the design process. Another advantage is that the placement and routing of the original design is not changed with the addition of debugging logic. Hence the placement is the same with or without the addition of debugging logic. Using this technique also provides flexibility which is a very important factor when the design is in the early stage of development and several iterations with modifications in the debugging logic are required.

4.1 Introduction:

ILA cores are a specialized form of logic that is added to the design inorder to improve the observibility and execution control of the design. They are in some ways similar to the Logic Analyzers that are used to test the fabricated chips. But there are some very important distinctions which are as follows

a. Logic analyzers for Chip testing are used after the system is already implemented and fabricated whereas ILAs are used in aiding the debugging of designs during the development of the system.

b. Logic Analyzers can only be used to probe the I/O pins of the design whereas ILA cores are added to the design due to which they are capable of probing even the internal signals. This is a very important feature as the no. of I/O pins available for testing purpose is very limited and it is desired that all the internal signals are observable.

There have been many instances in both commercial and academic domains which have demonstrated the capability of adding the ILA cores into the design to aid the debugging process. Prominent among the commercial tools are the SignalTap Logic Analyzer from Altera and the ChipScope Logic Analyzer from Xilinx. These tools have been discussed in chapter 2. Using SignalTap, an ILA cores can be added in the Logical Database before commencing the process of PAR carried out by FPGA vendors CAD tools. Any significant changes to be made in the debugging tool such as the change in the width of capture data, change in the trigger bus width, changes in the buffer depth require repeating the entire process of PAR which is quite time consuming. In the case of ChipScope Logic Analyzer from Xilinx the ILA core can be added by modifying the HDL source code or making the changes in the EDIF netlist before the process of PAR and this process has to be repeated whenever significant changes are made to the ILA as described above. An instance of an effort in the academic domain was carried out by researchers at BYU which targeted the JHDL design environment. It demonstrated the ability of ILA core addition in the JHDL source file and connecting the signals in the design to the ILA core using Jroute API calls. The time required for repeating this

process using the methods described is quite high in case of complex designs. To circumvent these excessive times for ILA core modifications, in this thesis we have developed a procedure for adding as well as modifying the ILA cores late into design process at the bitstream stage. By doing so a considerable amount of time is saved which is highly desirable.

ILA cores can be added to the design at various stages of the design process as depicted in Fig 4.1. The earliest stage at which the ILA cores can be added is in the HDL source code of the design. The Chipscope tool supports this feature. Adding the ILA core at this stage will result in design optimization being applied to the Logic Analyzer by the synthesis tools resulting in less resources being used by the ILA. But the main disadvantage is the inflexibility as any changes to be made to the ILA will require the repetition of the entire synthesis and PAR process. This results in the user being tempted to make the ILA very generic so that it can support various debugging capabilities as desired at the given instance. Such a generic ILA will use up lot of FPGA resources instead of supporting only those capabilities as desired at that instance. The next stage at which the ILA cores can be added is in the EDIF netlist generated by the synthesis. This capability is supported by ChipScope. Logic Analyzers can also be added after the process of Placement and Routing by the FPGA CAD tool. In case of Xilinx the modification can be made in the Native Circuit Description (NCD) file. The placed and routed design can be modified using the FPGA Editor from Xilinx which reads the NCD file of the design as input and a new NCD file is created after the necessary modification. A script can be written for modifying the ILA. The latest stage at which the ILA cores can be added is at the bitstream stage by using bitstream modification tools like JBits and

JRoute. The main advantage in using JBits and Jroute as discussed at the beginning of this chapter is in the time saved which is highly desirable. There are some disadvantages in making the modification so late in the design process which are as follows.

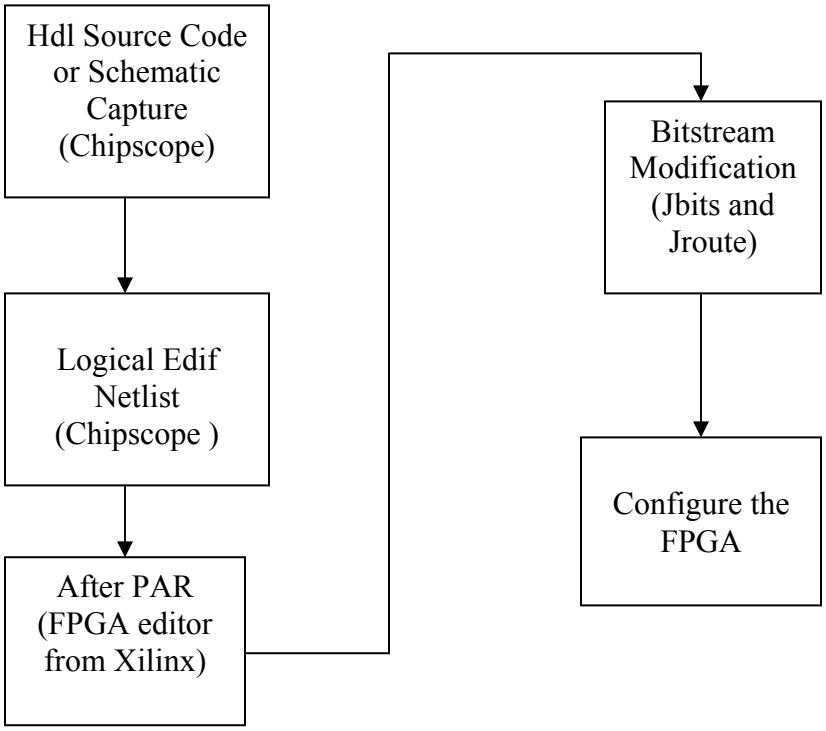


Fig 4.1 The different stages at which ILA cores can be added to the design

1. The modification made at the bitstream stage is less optimized in terms of area and speed.
2. Since modifications are made at the bitstream stage which is the lowest level of abstraction the user has to be familiar with the detailed architecture of the device.

4.2 Integrated Logic Analyzers: In this section we will give a detailed description of the various aspects of the ILA. Fig 4.2 gives a block diagram of an ILA core. As described in the previous section an ILA can be added at various stages in the design process. One of the aims in this thesis was to demonstrate that such ILA cores can be added to the design at the bitstream level in a minimum amount of time.

The components that make up an ILA include the ILA controller, the trigger logic, the address generator and the data buffer. This section will explain the overall functioning of the ILA. The trigger input to the trigger logic is checked for the occurrence of certain trigger condition. On the occurrence of the trigger condition the control logic enable the address generator. The address generator generates the address where the data signal is to be saved in the storage buffer. The control logic also enables the write enable signal of the storage buffer so that data can be written in the particular location. When the buffer is full the buffer full output of the control logic goes high. The data saved in the data buffer can be analyzed once the hardware execution is terminated. The following is a brief description of the different components that make up an ILA.

4.2.1 The ILA controller: An ILA controller is a state machine which controls the functioning of the ILA and coordinates the communication between different components. For instance on the occurrence of certain trigger condition the ILA controller asserts the write enable input of the data buffer so that data can be saved in it. When the buffer is full the control logic asserts a trigger signal indication that no more data can be sampled by the storage buffer.

4.2.2Trigger Logic: The trigger logic is used to support various types of trigger conditions. The different trigger conditions include

- Equal to
- Not equal to
- Greater than
- Greater than or equal to
- Less than
- Less than or equal to

On the occurrence of the trigger condition data can be captured on the

- The rising edge
- The falling edge
- The rising edge or the falling edge

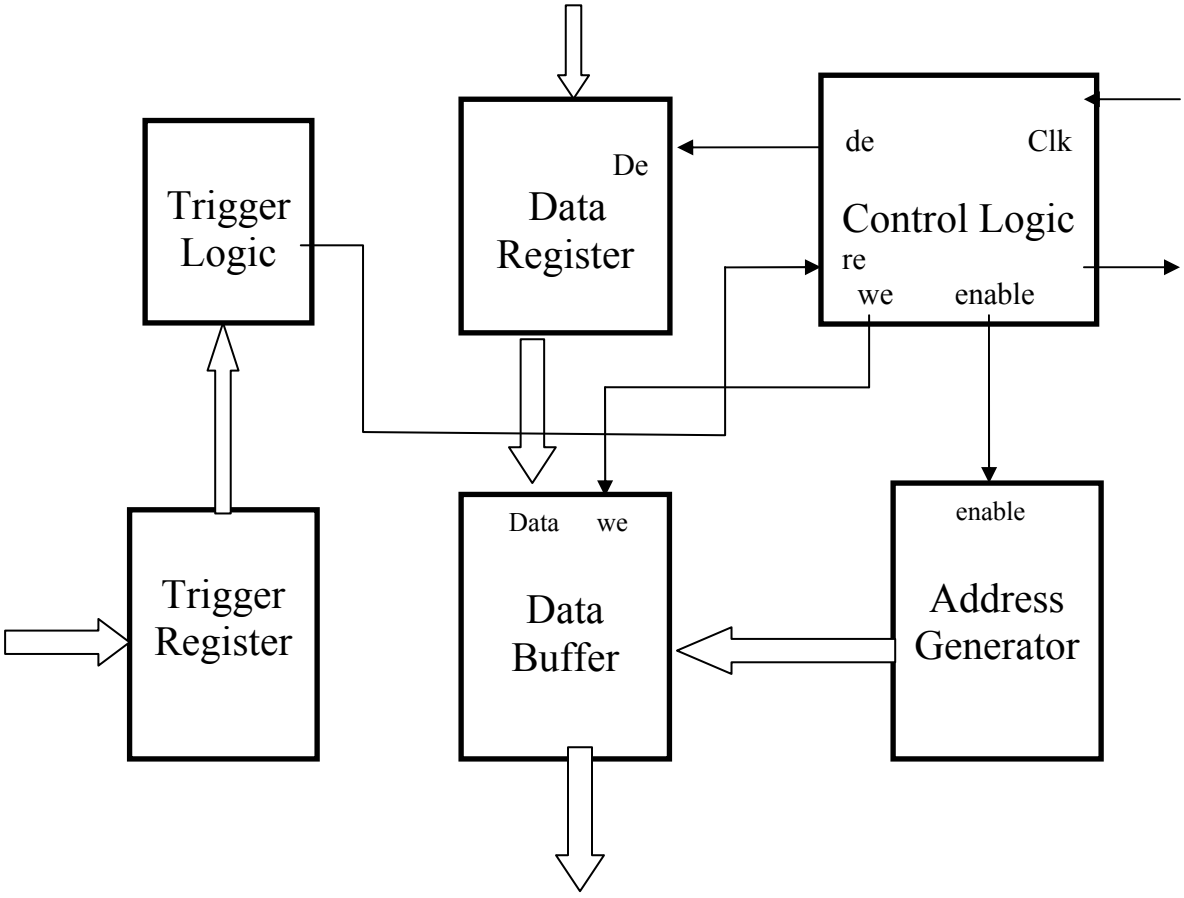


Fig 4.2 Integrated Logic Analyzer Core

Some instances of the trigger logic are shown in Fig 4.3 and Fig 4.4 which is used to implemented different types of trigger conditions. In Fig 4.3 the input trigger signal is checked for less than or equal to condition. When the given trigger condition occurs the output of the trigger logic goes high. A trigger logic which supports more than one trigger input can also be generated as shown the trigger logic in Fig 4.4. Here the trigger logic supports two input signal which can be tested for certain trigger condition and a logical operation between the individual output such as a logical OR in the given example will give the overall trigger output. The RTP cores supplied by JBits can be used to implement the trigger logic to support the different type of trigger conditions and different trigger widths. The different possible trigger widths are 8, 16, 24 and 32. Depending on the requirement the use can select any particular trigger width.

4.2.3 Data Storage Buffer:

The storage buffer is used to save the value of the signals on the data lines on the occurrence of a certain trigger condition. When the trigger occurs, the address generator increments the address on the address line and the control logic asserts the write enable input of the storage buffer so that the data on the data input can be saved in the storage buffer. The data storage buffer is implemented using the block Rams available in the Virtex™ devices. A single block Ram in a Virtex™ device has 4096 bits. Data buffers of different aspect ratios can be implemented using these Block Rams. So in our thesis we have implemented storage buffers of varying depths and data widths as shown in the table of Fig 4.5.

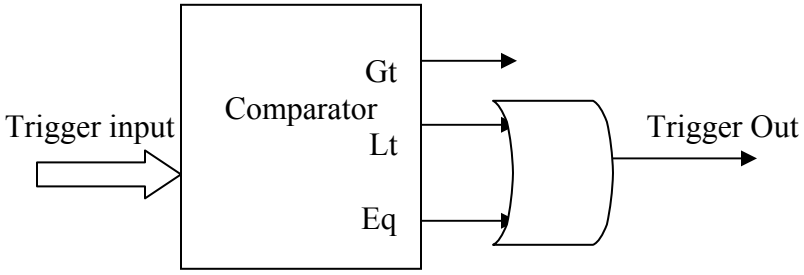


Fig 4.3 Trigger logic to test the less than or equal to condition

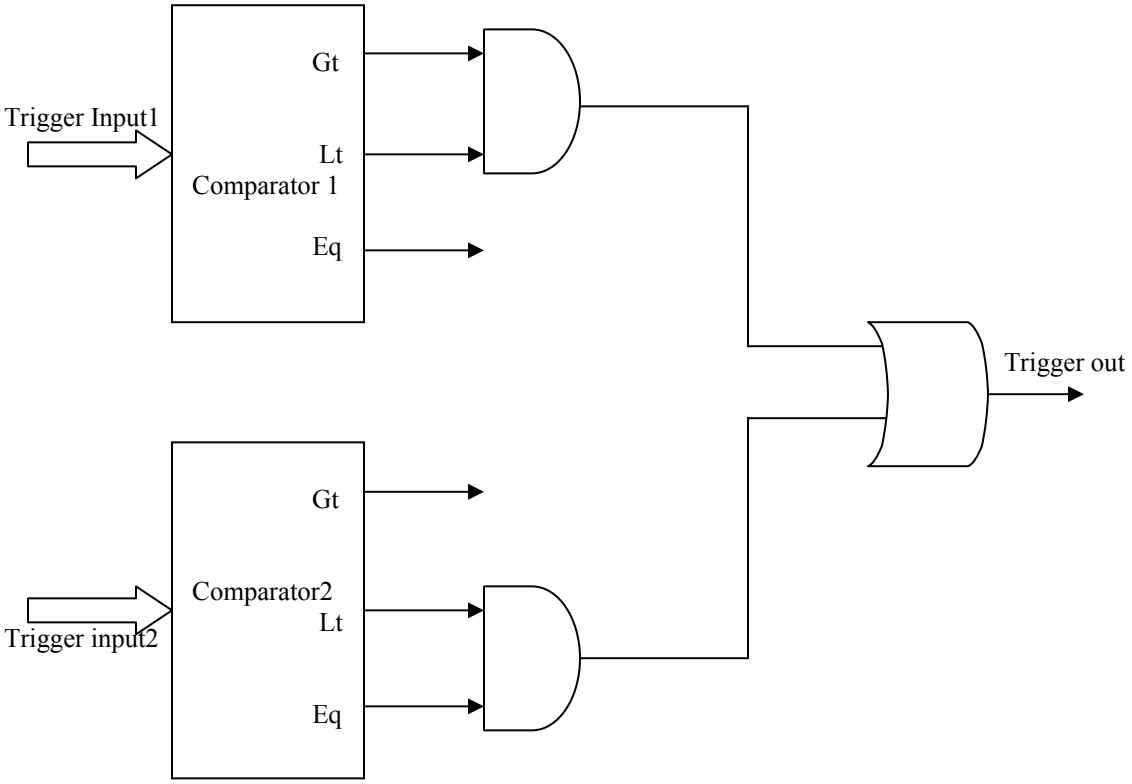


Fig 4.4 An instance where more than one trigger conditions can be tested simultaneously

4.2.4 Address Generator:

An address generator is used to generate the address of the memory location in the data buffer where the next data value is saved. This address generator is implemented

using a counter which is incremented on the occurrence of certain trigger condition and this value of the counter forms the address input to the storage buffer. The flow chart in Fig 4.6 depicts the functioning of the ILA core.

Data Width	Buffer Depth
16	256
8	512
4	1024
2	2048
1	4096

Fig 4.5 Table showing different buffer depths and Data widths

4.3 Process of adding ILA cores to the design using JBits and JRoute:

In this section we will describe the procedure of adding ILA cores to the design at the bitstream level. We have used JBits and JRoute to add the ILA cores at the bitstream stage. Towards achieving this goal we have used the Runtime Parameterizable (RTP) cores supplied with the JBits tool kit. These RTP cores include various components like counters, registers, comparators, adders, etc. Since these cores are parameterizable different types of debugging logic can be generated at the bitstream level using JBits and JRoute. These RTP cores enable us to work at a higher level of abstraction as against the case where the designer is required to configure the bitstream at

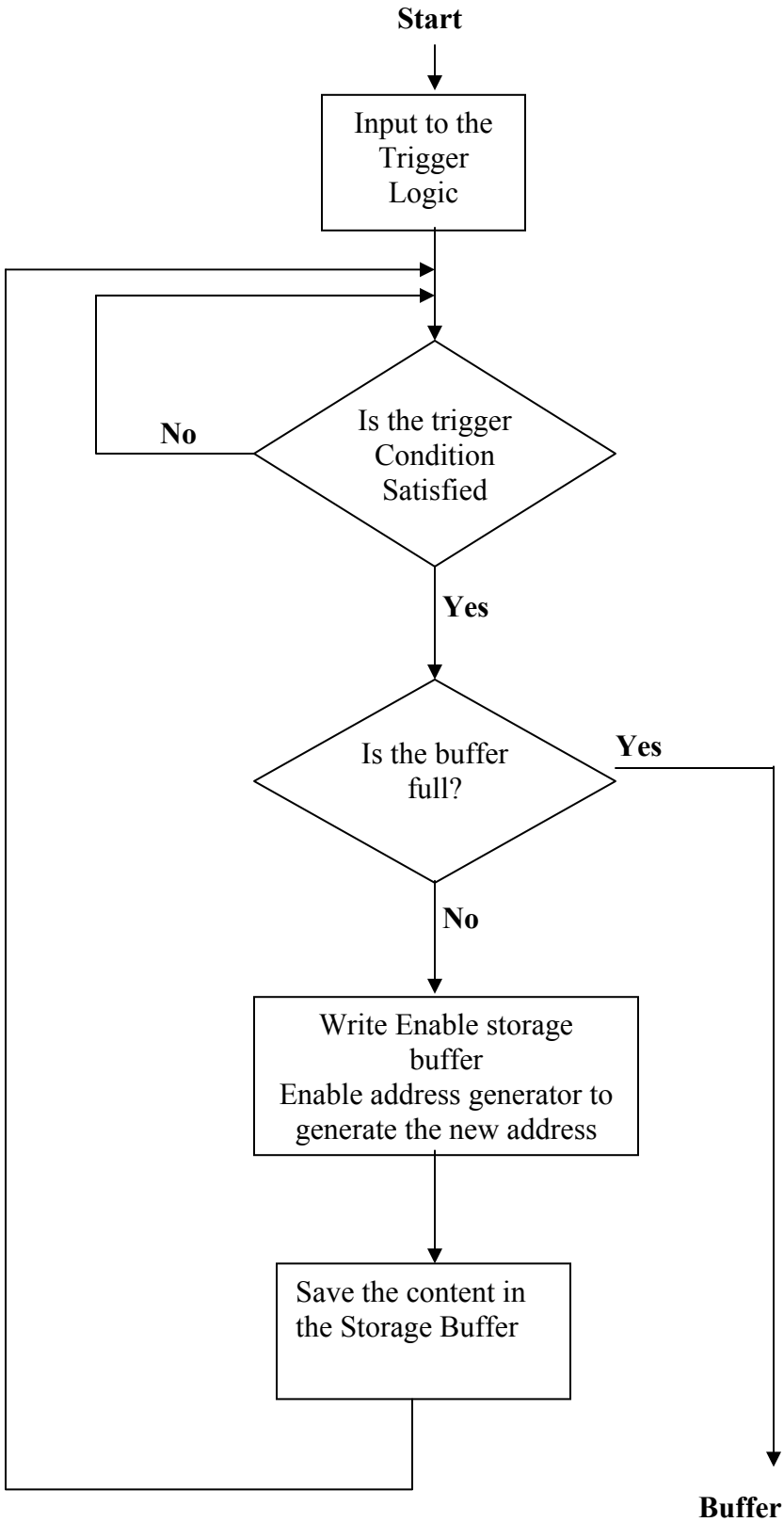


Fig 4.6 Flow chart showing the steps in the functioning of ILA

a very low level. Each of the steps involved is explained in detail below and the block diagram depicting this process is given in Fig 4.7.

4.3.1 Providing the user with a logical view of the design: After the design has been synthesized by the synthesis tool and then mapped, placed and routed by the Xilinx FPGA tools it is desired that the user is provided with a logical view of the design even after mapping the design to the physical FPGA resources. Towards achieving this goal we have generated a symbol called ".rbstable" which contains detailed information of the logical to physical mapping. This process of symbol table creation was described in detail in the previous chapter. The symbol table provides the mapping of the each of the instance in the design to their Physical counterparts and the nets connected to each of the ports for all the instances. When the user is provided with a logical view of the design he/she can select the signal to be monitored in the logical view and as the environment is familiar with the physical counterparts of these signals whose information is available in the symbol table the corresponding signals are monitored in the physical design. The information provided by the user for the creation of ILA cores is as follows.

1. The depth of the storage buffer.
2. The width of the data signal
3. The width of the trigger signal or signals depending on the no of trigger signals.
4. The data signal in the design that is to be monitored
5. The trigger signal or signals that are to be monitored.

Based on this information provided by the user ILA cores are automatically generated by the ILA core generator.

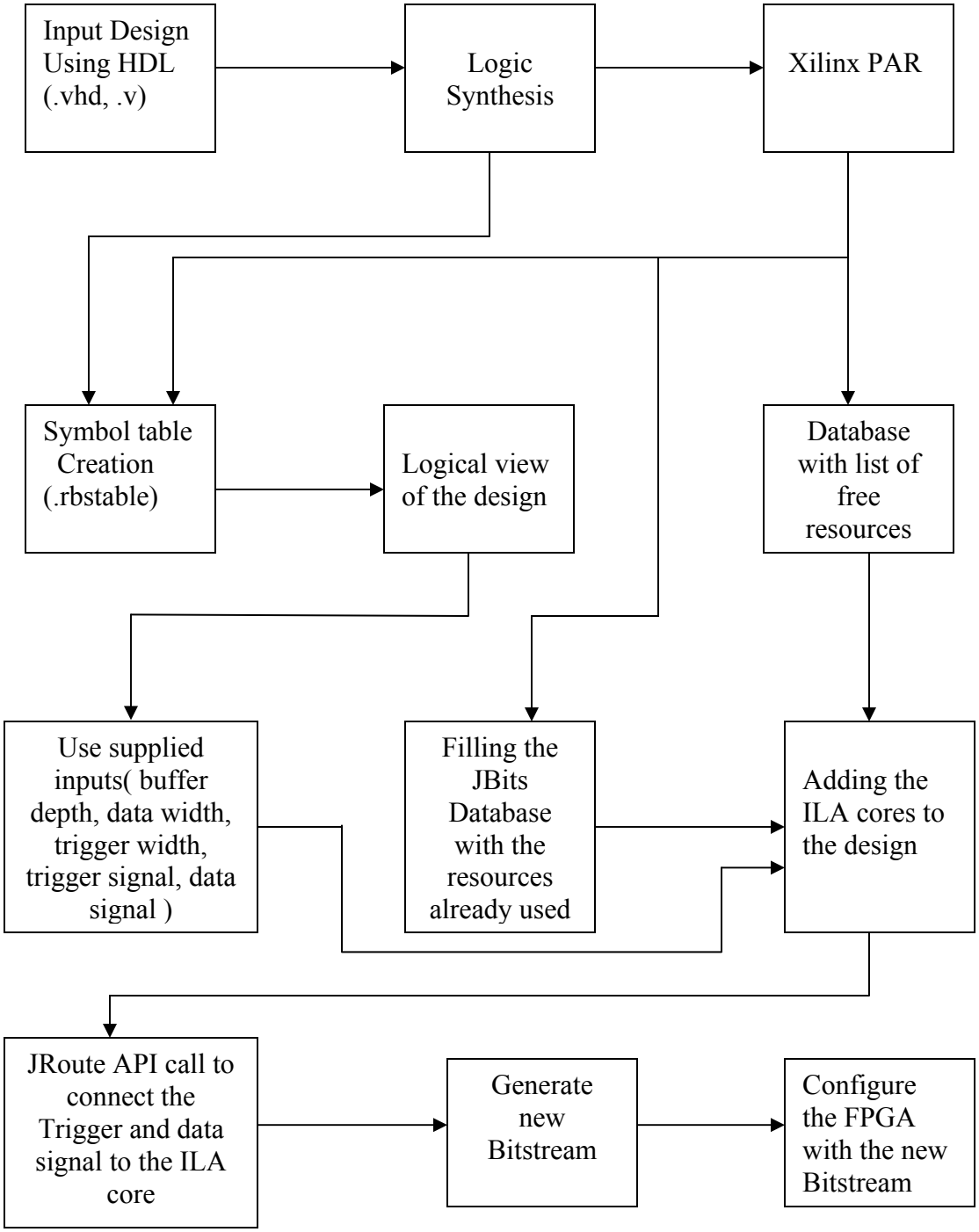


Fig 4.7 The different steps during the process of ILA addition

4.3.2 Filling the Database with the used resources: Before adding the ILA core to the design the designer has to make sure that the resources that are already used by the actual design are not disturbed during the process of adding the ILA cores. Towards achieving this goal the JBits and JRoute Database is filled with the information of the used resources before commencing the process of ILA core addition.

4.3.3 Tracking the free resources available in the device: Inorder to add the ILA core to the design the information of the free resources in the FPGA should be known. The NCD file generated by the Xilinx placement and routing tool contains information of the used resources in the FPGA. A textual format of this file is generated using a utility called "ncdread" provided by Xilinx. A 2 Dimensional array representing the CLB slices inside the FPGA is created and all of the entries are marked as free. The ncd file parser which is implemented as a part of this thesis parses this file and fills the 2D array with the information of the used resources as depicted in the diagram of Fig 4.8. Thus this array will contain the information of the free resources which is referred to while adding the ILA core to the design so that only the free resources that are left are used during the process of ILA core addition.

4.3.4 Algorithm used to add the ILA core: In this section we will describe the algorithm that we have used to add the ILA cores to the design. In the previous sections we have described different components that make up an ILA. If the designs are dense it might not be possible to add the entire ILA as a single entity but it has to be split and add at different locations in the FPGA based on the availability of resources. Fig 4.9 shows

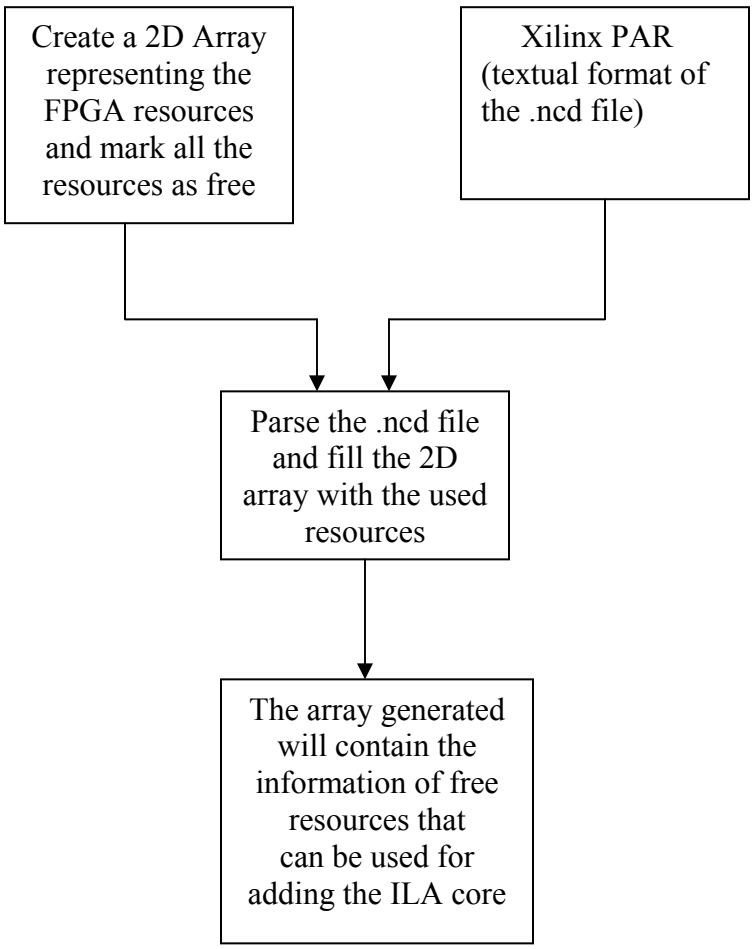


Fig 4.8 Block Diagram describing the process of tracking the free resources

the flow chart of the algorithm that we have used inorder to add the ILA cores to the design. First a list of all the components that make up the ILA such as the trigger logic, the address generator, the control logic, etc is created. Now the entire FPGA is traversed inorder to find the free resources inside the FPGA. When a free slice is found the algorithm tries to find all the slices that are free both in the X and Y direction with this slice located at the bottom left corner of other free resources. Now the component with

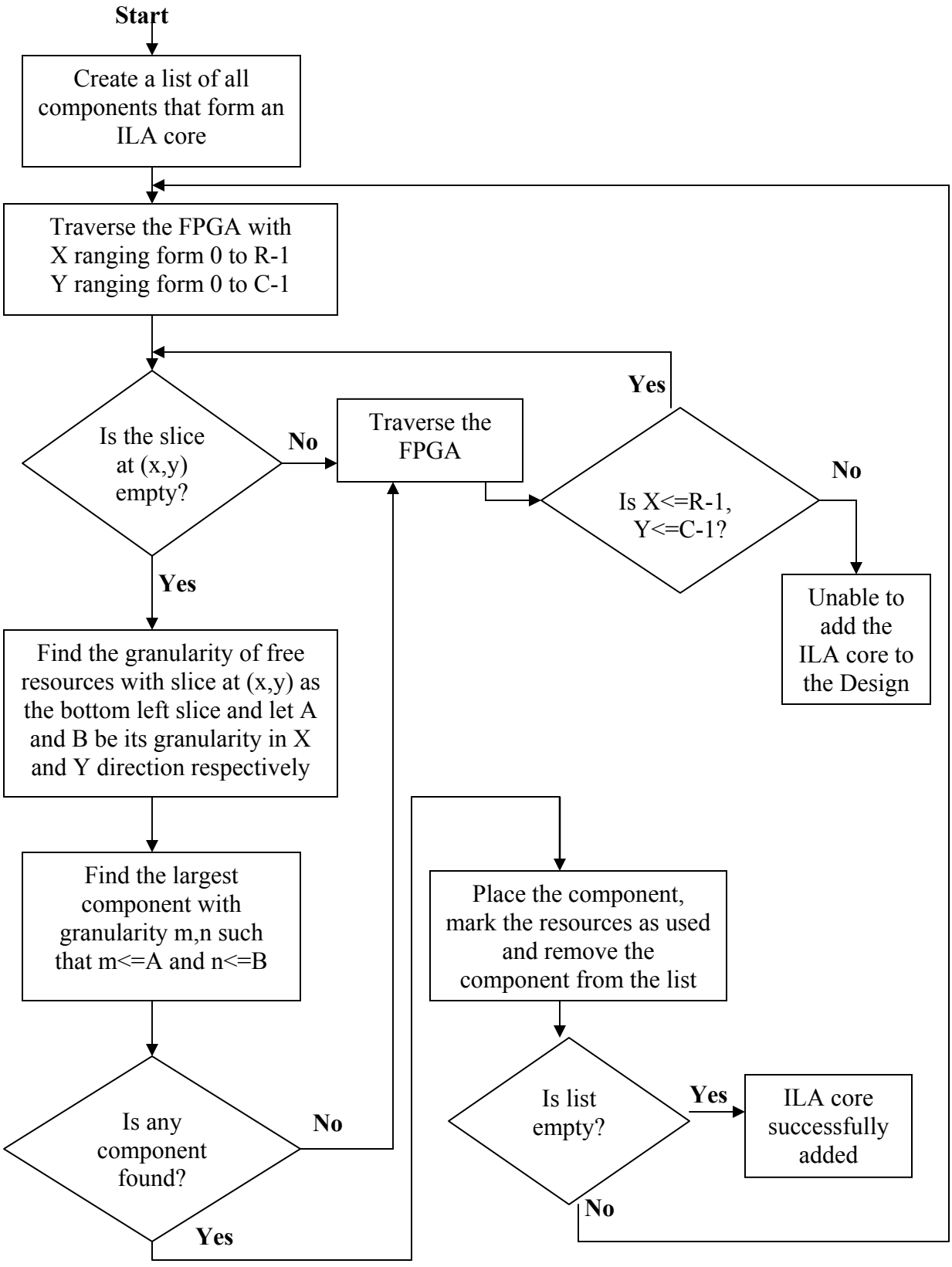


Fig 4.9 Flow chart describing the algorithm used to add the ILA core

the highest granularity in the X and Y direction that can fit into these free resources is selected from the list and placed in these free resources. Once this is done the placed component is removed form the list and the resources used are marked so that they won't be used again. This process is repeated until all the components of the ILA are placed, or the entire array has been searched. The algorithm terminates when all the components are placed and declares success but if sufficient resource are not available to accommodate all the components of the ILA the algorithm terminates by declaring a failure.

4.3.5 Connecting the data and trigger signal to the ILA: Once the ILA is added to the design the trigger and data signals are connected to the ILA so that the process of debugging can start. Inorder to connect these signals to the ILA core we use the JRoute tool to connect the nets. This tool provides support for API calls which take the end point of the net (i.e. the ports) as parameters so that the two ports can be connected via a net. The symbol table (".rbstable") that we have created contains the corresponding physical ports information of the logical ports the user has selected, which is provided as input to the JRoute function calls so that the trigger and data signals can be hooked to the ILA.

4.3.5 Creating the new bitstream and configuring the device: Once the ILA core is added and connected to the design a new bitstream is generated which contains the configuration data for the design with the ILA added. This new bitstream is used to configure the FPGA. Once the FPGA is configured the ILA core can be used for debugging the design.

4.4. Example showing the process of Debugging Design using an ILA: We will now describe the process of debugging a design using an ILA with the help of an example. Consider a design which contains two counters as shown in Fig 4.10 (a). One of the counters output is used as a trigger signal and the output of other counter is used as a data signal. Counter1 is a 4 bit counter. The output of this counter will be used as the trigger signal. Counter2 is an 8 bit counter whose output is used as the data signal that is sampled and saved in the data buffer on the occurrence of certain trigger condition.

First the signals to be monitored are identified which include the 4 bit trigger signal from counter1 and the 8 bit data signal from counter2. The ILA core is generated and added to the design as shown in Fig 4.10 (b). Now the 4 bit trigger signal

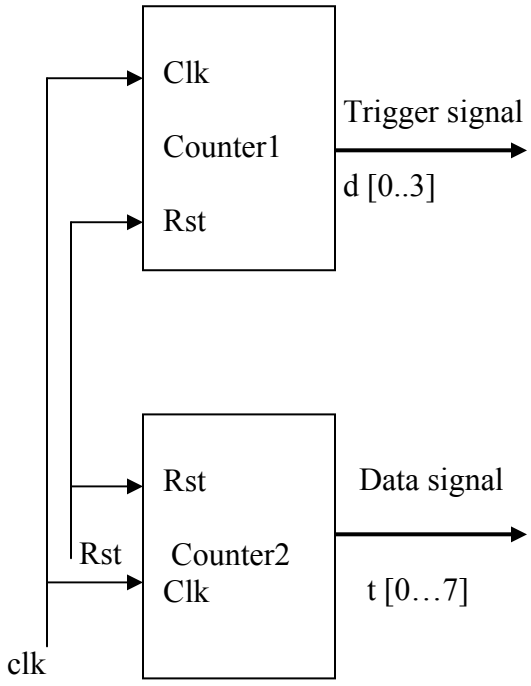


Fig 4.10(a) Original design

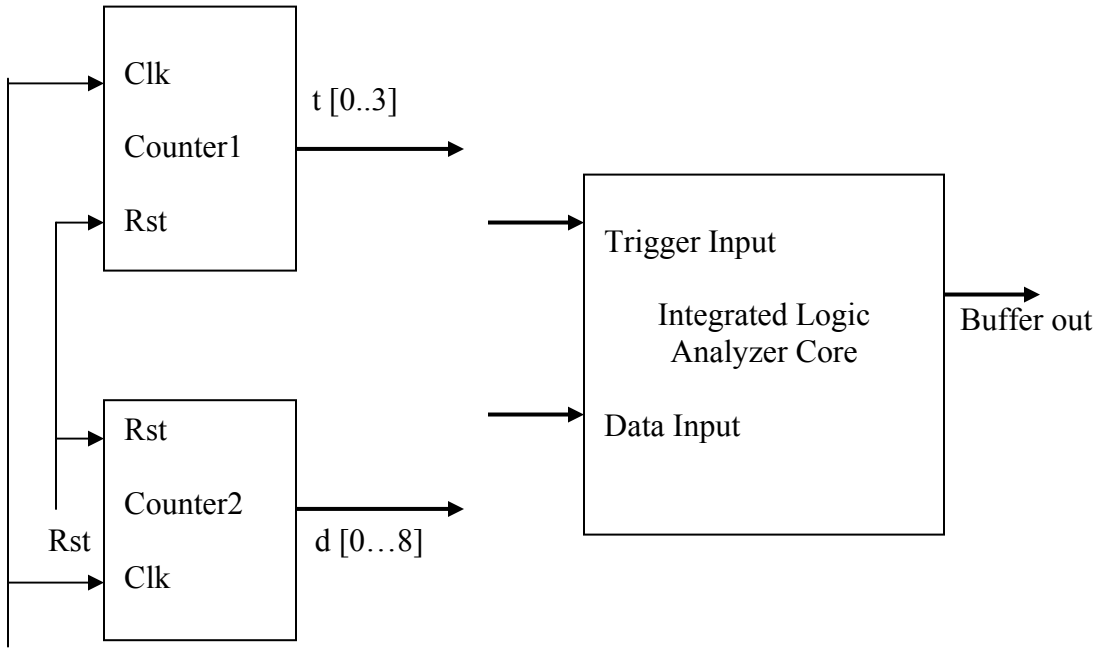


Fig 4.10(b) Adding the ILA core to the design using JBits

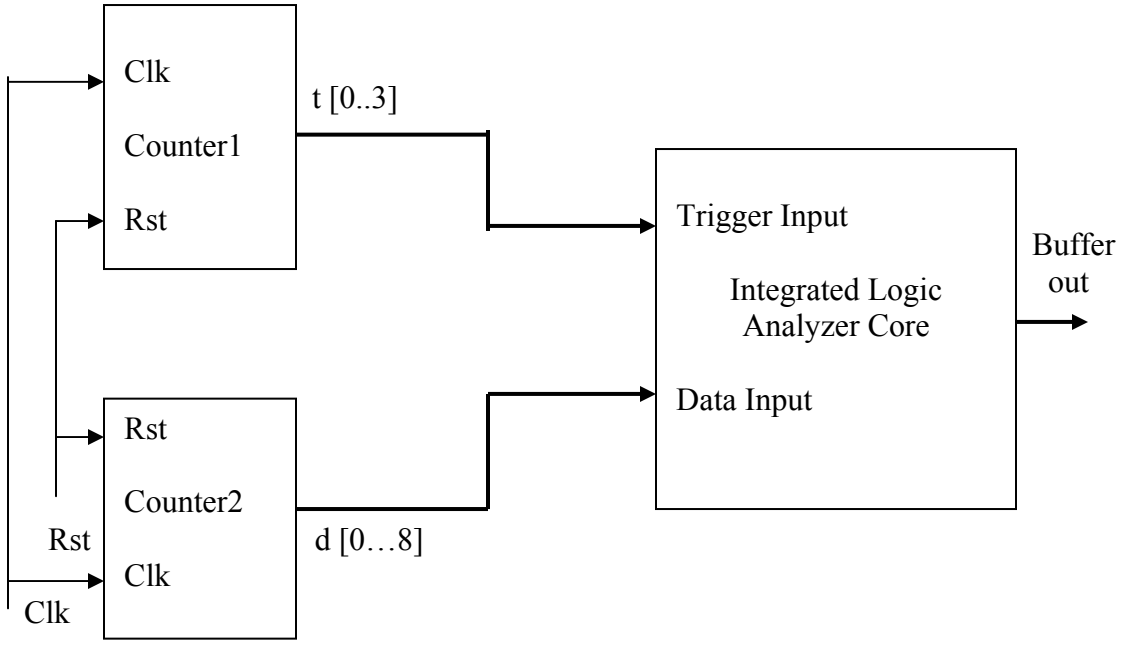


Fig 4.10 (c) Connecting the data and trigger signals to the ILA core using JRouteFig

Fig 4.10 Blocking diagram depicting the steps in addition of ILA core to a design

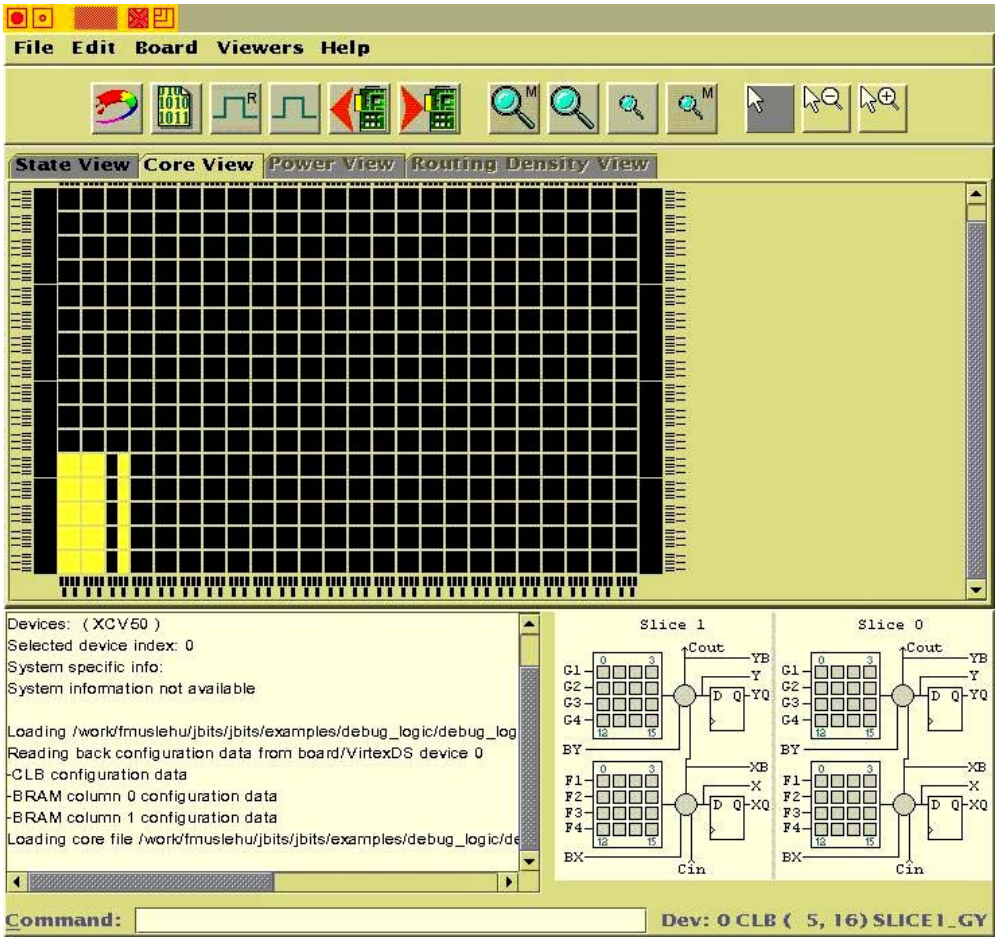


Fig 4.11 The core view of the ILA added to the design

Clock Cycle	Trigger Signal T [0…3] (hex)	Data Signal D[0…7] (hex)	Value saved in Data buffer (hex)
16	F	0A	0A
32	F	1A	1A
48	F	2A	2A
64	F	3A	3A
80	F	4A	4A
96	F	5A	5A

Fig 4.12 Table showing the various signal values during simulation

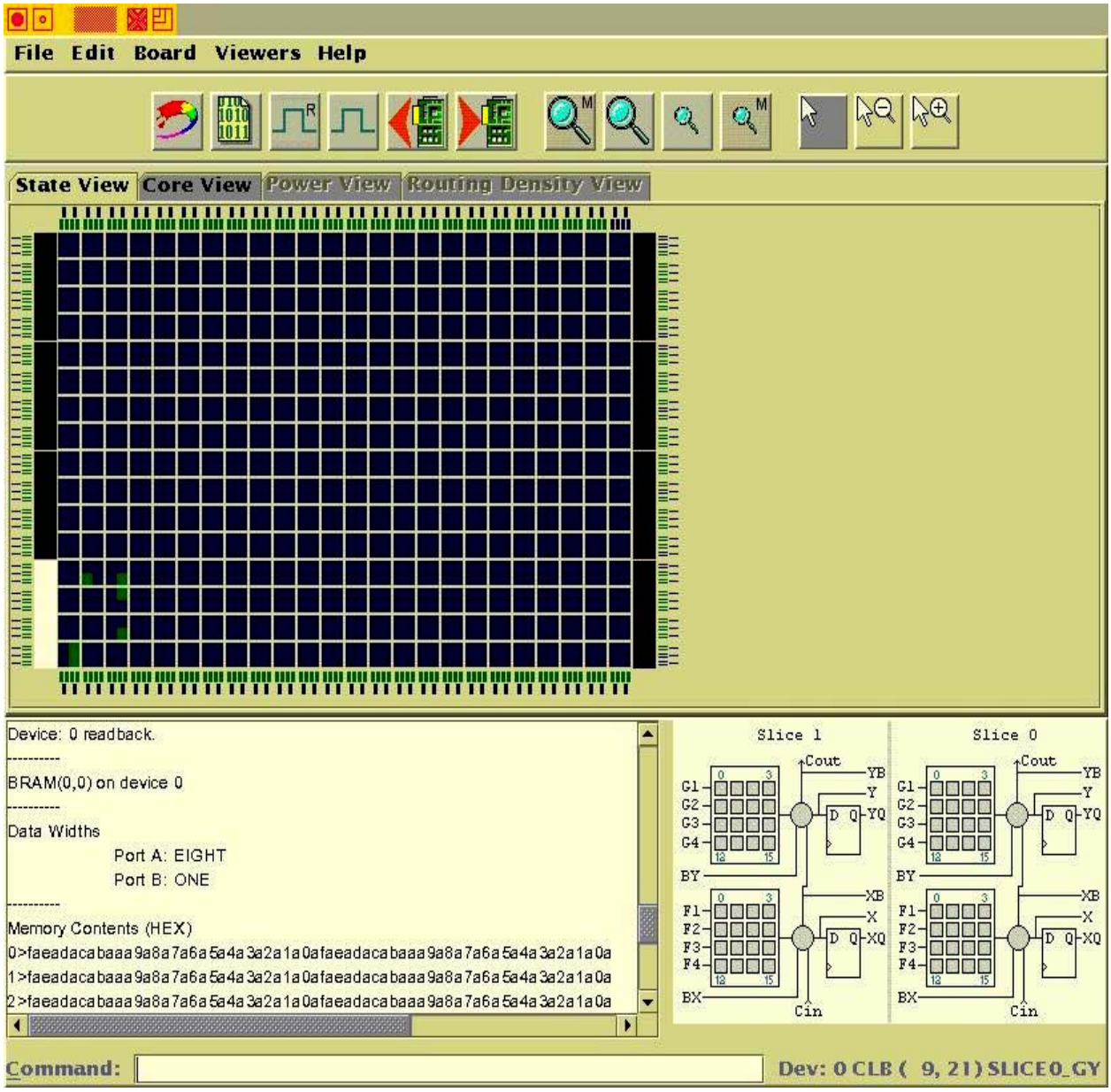


Fig 4.13 The state view of the ILA also showing the content of the Block Ram Used as Data Buffer

t [0…3] and the 8 bit data signal d [0…7] are connected to ILA using the JRoute API calls as shown in Fig 4.10 (c). Fig 4.11 shows the core view of the ILA in BoardScope after the ILA core is added. We have used the Virtex device simulator which s a part of the BoardScope tool kit to simulate the working of the ILA after it has been added to the

design. The trigger value in this example is set to 15. So whenever the value on the trigger bus reaches 15 the content of the counter2 or the content on the data bus is saved in the data buffer. Fig 4.13 shows the content of the data buffer after some 1500 clock cycles. The value on the trigger bus and the data bus and the content of the data buffer for certain sample clock cycles is shown in Fig 4.12.

Thus in this chapter we have explained this process of ILA core addition using bitstream modification tools like JBits and JRoute in great detail. Thus using the readback capability as discussed in chapter 3 and the features supported by the ILA cores as discussed in this chapter the observibility and execution control of the designs can be enhanced considerably.

Chapter 5

Results and Analysis

This chapter will give a detailed description of the experiments that were conducted and the conclusions and insight gained from this research. In chapters 3 and 4 we have described in detail the different steps that are involved to automate the process of ILA core addition and modification which has been developed as a part of this research to improve the observibility and execution control of the designs implemented on FPGA Based systems. This technique which was studied extensively consists of the process of ILA core addition by modifying the bitstream generated by the Xilinx PAR tools and making changes to these ILA cores at the bitstream stage as desired based on the particular debugging needs. Logical to Physical design mapping for creation of symbol table is an important step towards developing such a technique as discussed in the previous chapters. In this chapter we will try to summarize the different aspects of this technique and provide results obtained by using this technique and try to gain some insight.

.

5.1 Symbol table Creation for providing the user with a logical view of the designs during ILA core addition:

The process of Logical to Physical design mapping for the creation of Symbol table is very involved and this entire process has been automated as a part of this research. We have implemented the parsers for various files that are generated by the synthesis and the FPGA vendors CAD tools which are utilized during this process. These parsers were implemented using UNIX utilities, Lex and YACC. The information

gathered from various files is put together to create a symbol table. This symbol table created by the mapper provides a logical view of the design to the user with the information of all the logical instance in the design including the information of the ports of each of these instances and the nets connected to these ports even after the design has been mapped to the physical FPGA. With the information of the mapping available in the symbol table the designer can select the signals in the design that he/she is interested in and the corresponding signal in the physical FPGA which are recognized by the Jbits and Jroute tools is selected in the physical FPGA.

The information in the symbol table can be used for other debugging purposes such as readback which was demonstrated by the researchers at BYU [24] for the JHDL design environment. In this research we are targeting a design environment in which designs are described in a hardware description language like VHDL or Verilog. Using the information in the symbol table similar type of debugging capabilities can be provided for the VHDL or Verilog design environment.

5.2 Addition and Modification of ILA cores:

The process of adding the ILA cores and modifying them at the bitstream level using bitstream modification tools like JBits and JRoute have been explained in great detailed in the previous chapter. The main metric that was used to test the effectiveness of the techniques developed lies in the time saved in adding the ILA cores at the bitstream stage rather than before the start of PAR, which will result in drastic improvement. We used the BoardScope debugger from Xilinx to debug the designs modified using Jbits and Jroute at the bitstream stage for adding the ILA cores.

5.2.1 Description of the ILA cores added to the design:

We added ILA cores with different configurations to the designs using JBits and JRoute. The different configurations of the ILA cores depend on the width of the trigger and data signals that each of the ILA supports. In the experiments conducted we used the ILA cores with the following Trigger and Data widths.

ILA configuration	Trigger Width	Data Width
ILA_8_16	16	8
ILA_8_24	24	8
ILA_8_32	32	8
ILA_8_40	40	8
ILA_8_48	48	8
ILA_8_64	64	8
ILA_8_80	80	8
ILA_8_96	96	8

Fig 5.1 ILAs with different configurations used in the experiments

We have used a specific format to represent each of the ILA cores based on its configuration. The format used is as follows.

ILA_# of Data bits_# of Trigger bits

For instance an ILA core with an 8 bit Data signal and a 48 bit trigger signal can be represented using the above mentioned format as ILA_8_48. Each of these ILA cores utilize different amount of FPGA resources based on its configuration. In the experiments

we have used a single BRAM in the Virtex device to implement the data buffer in which the sampled values are saved on the occurrence of certain trigger condition. The table in Figure 5.2(a) gives the information of the resources used by each of the ILA cores based on its configuration when the ILA cores are added to the design using the bitstream modification techniques (BMOD) and the same is also depicted with the help of a bar graph in Fig 5.2(b). Similarly Fig 5.2(a) also shows the resources used by ILA cores with similar configuration when they are added into the VHDL source code before the commencement of PAR and a bar graph representing the same is shown in Fig 5.2(b). It is observed that the ILA cores add at the VHDL stage use less resource than the bitstream modification technique. This is due to the optimizations performed by the synthesis and Xilinx tools for ILAs added in VHDL which is not the case for ILA core added using the bitstream modification method.

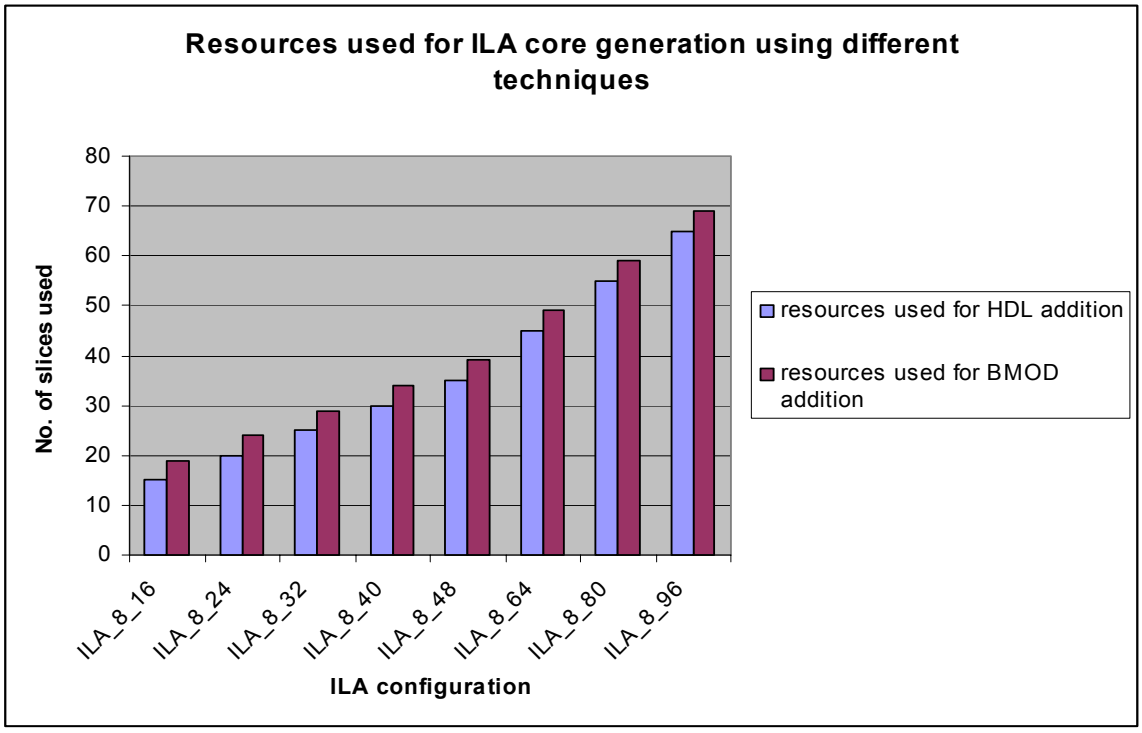
5.2.2 Benchmarks used to test the effectiveness of the technique:

The following is a brief description of each of the benchmarks that are used to test the technique developed. We have used the ITC 99' Benchmarks developed by the CAD group at Politecnico di Torino (I99T). It is considered that the characteristics of these circuits closely resemble those of the synthesized circuits [36]. We have implemented each of the Benchmarks on a Virtex XCV1000 chip. The ILA cores with the different configuration are added to each of these benchmarks after they have been synthesized, mapped, placed and routed. The benchmarks vary in size with 'b12' being the smallest one utilizing 1% of the FPGA slices and the largest being the 'b18'

benchmark that utilizes 72% of the slices in a Virtex™ XCV1000 FPGA. The table in Fig 5.3 gives a list of the FPGA resources used by each of the benchmarks. Using

Configuration of the ILA	BMOD		Added in HDL	
	No. of Slices used	No of BRAMs used	No. of Slices used	No of BRAMs used
ILA_8_16	19	1	15	1
ILA_8_24	24	1	20	1
ILA-8_32	29	1	25	1
ILA_8_40	34	1	30	1
ILA_8_48	39	1	35	1
ILA_8_64	49	1	45	1
ILA_8_80	59	1	55	1
ILA_8_96	69	1	65	1

(a)



(b)

Fig 5.2 Resources used by each ILA configuration

benchmarks with a large variation is size will help in analyzing the improvement in timing obtained by applying the technique developed inorder to add the ILA cores to the designs in this research and how the complexity of the designs governs this improvement?. An important aspect of these benchmarks lies in the fact that they were developed to test new techniques which help in improving the debugging capabilities of designs. Fig 5.4 gives a brief description of the 8 circuits from among the 22 available circuits that we used as benchmarks.

All these benchmarks are available in synthesizable RT Vhdl format or Edif netlist format. These benchmarks use only the IEEE standard library components which make it easy to synthesize them with any of the available synthesis tools. Some of these benchmarks are quite complex. For instance the b18 benchmark is reasonable more complex than the largest ISCAS'89 benchmarks. This is important to test the effect of time taken for ILA core addition as the complexity of designs change drastically. Another important aspect of these benchmarks is the fact that they use single clock which synchronizes all the flip flops in the design and makes it easy to test the capabilities of the ILA cores after they are added to the design. The number of primary input pins for each of these benchmarks varies form 1 to 37 and the number of output pins varies from 1 to 97. This variation in the number of I/O pins requires ILA cores with varying capabilities for each of the benchmarks.

5.2.3 Different methods used for adding the ILA cores to the design:

Inorder to test the degree of improvement that can be achieved by using the technique of bitstream modification as developed as a part of this thesis we have also

added the ILA cores to the benchmarks at the HDL stages of the design process so that the relative improvement achieved can be analyzed. We can add the ILA cores to the design directly in the HDL description of the design before the synthesis of the design is

Benchmarks	No of Slices used	% of Slices used	No. of Nets in the design
B12	185	1%	1289
B14	637	5%	4468
B15	1085	8%	7420
B17	3374	27%	22745
B18	8965	72%	60182
B20	1355	11%	9039
B21	1313	10%	8991
B22	1974	16%	13413

Fig 5.3 Resources used by each Benchmark

initiated. All the commercial as well as academic tools add the ILA cores before the commencement of the Xilinx CAD flow. For instance the Chipscope tool from Xilinx supports the addition of ILA core at the VHDL stage or after synthesis. The Signal Tap Logic Analyzer from Xilinx supports the ILA core addition in the logic database before commencing the process of placement and routing of the design. Similarly the JHDL

debugging tools also adds the ILA in the logical database or directly in the JHDL source code which is before the start of PAR. So we try to find the time taken for ILA core addition by repeating the process of PAR as each of these techniques discussed above atleast require the repetition of this process of PAR, we compare the PAR time with the time taken to add the ILA cores using our technique which adds the ILA at the bitstream stage without repeating the PAR step. Since we are considering only the time taken for the process of PAR it is the best case scenario for these other techniques as some of these techniques even require the repetition of logic synthesis as is the case with Chipscope ILA, which may result in this process of ILA core addition taking even more time. Once the ILA core is added to the design there are many options that are supported by the Xilinx Placement and Routing tools to place and route the designs like the Normal build, Guided PAR and fast build. In addition to these, the technique we developed is used to add the ILA cores directly at the bitstream stage which does not require the repetition of process of PAR each time modification are made to the ILA. We refer to this technique as Bitstream addition and modification of ILA (BMOD) .We use the Jbits and Jroute tools from Xilinx to modify the bitstream of the original design to add the ILA core and connecting the signals in the design to the ILA. Even though the routing algorithm used by Jroute is slow it was observed that the improvement in terms of time taken for ILA core addition was very less when compared to the other techniques. This is because using Jbits and Jroute the placement and routing of the original design is not disturbed and only the new resources in the FPGA for ILA core addition are configured and only the signals in the design to be monitored are routed using JRoute. Hence we are comparing the time taken to repeat the process of PAR for adding the ILA cores as is done by the other

methods against the time taken by our technique (BMOD) which utilizes the process of bitstream modification for ILA core addition and does not require the repetition of PAR. The following gives a brief description of each of the methods that have been employed to add the ILA cores to the designs and the time taken by each of these four techniques for ILA core addition is used to compare the effectiveness of each of these techniques.

Normal Build: In case of Normal Build the process of Placement and Routing (PAR) is performed by the Xilinx CAD tool with an average effort level for PAR. This is generally used when the design meets the timing goals with this average effort level. If the design does not meet the timing goals using this method then a high design effort is applied so that desired goals can be accomplished.

Fast Build: In the case of Fast Build the process of PAR which is most time consuming is performed with the least effort level. This method is used when the design being mapped easily meets the design goals like frequency, 100 % routing of the design, etc.

Guided PAR: In the case of Guided PAR the information from previously placed and routed design, without the ILA core added to it is used. The Native Circuit description file acts as the guide file to guide the process of PAR so that time can be saved during the process of mapping as well as placement and routing the design.

Bitstream modification of designs (BMOD) : As explained in chapter 4 after the design is paced and routed by the Xilinx CAD tools we take the user inputs such as the signals to be monitored in the logical view, the width of the trigger signal, the width of the data signal and the depth of the storage buffer, an ILA core is automatically generated and using the information in the symbol table which has the information of the Logical to

Physical design mapping modification are made to the design at the bitstream level using JBits and JRoute inorder to add the ILA core to the design and connect the signals to be monitored in the design to the corresponding ILA inputs.

Circuits	Description
B12	1-player game (guess a sequence)
B14	Viper processor (subset)
B15	80386 processor (subset)
B17	Three copies of b15
B18	Two copies of b14 and two of b17
B20	A copy of b14 and a modified version of b14
B21	Two copies of b14
B22	A copy of b14 and two modified versions of b14

Fig 5.4 Brief Description of each of he Benchmarks used

We have measured the time taken by each of the four techniques discussed above inorder to show that the fourth technique that we have developed is better in terms of time saved when adding the ILA cores as well as modifying them. We compare the time taken by the first three methods for mapping, placing and routing the design which is indicative of the time taken by the other tools in both academic and commercial domains which add the ILA cores before the start of PAR and comparing it with the time taken in modifying the bitstream to add the ILA cores using JBits and JRoute and using

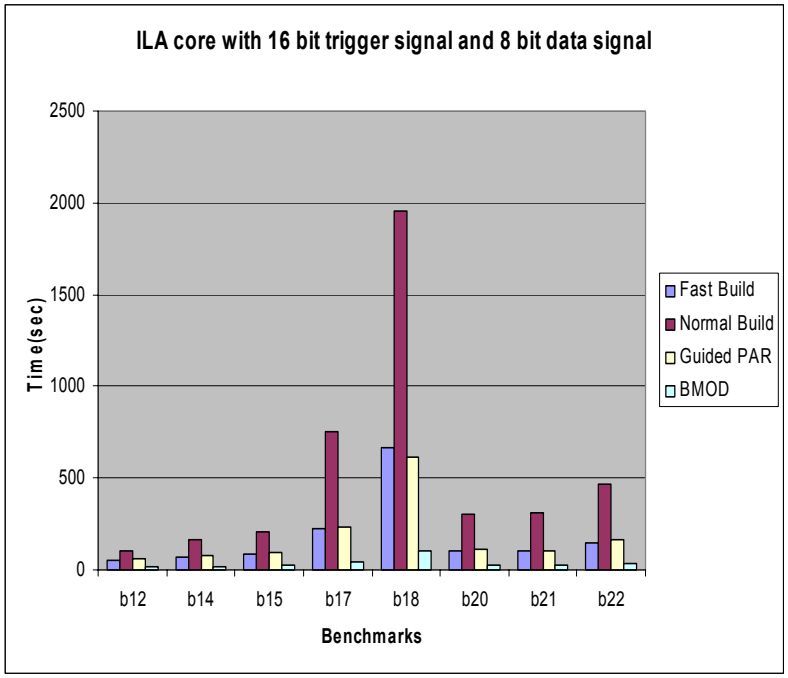
this as a metric to show the improvement in terms of time taken by each of these methods for ILA core addition.

5.3 Experiments and Results:

We have added eight ILA cores with varying configuration to each of the circuits. These different ILA cores added to each of the designs include ILA_8_16, ILA_8_24, ILA_8_32, ILA_8_40, ILA_8_48, ILA_8_64, ILA_8_80, ILA_8_96. We have performed these extensive experiments to gain an insight as to how the ILA addition time varies with respect to the complexity of the designs as well as the complexity of the ILA cores. Each of the figures from Fig 5.5 – Fig 5.12 gives a detailed information of the data that was gathered form the experiments. Each of these figures shows the time taken for the addition of all the an ILA core with a particular configuration each of the eight circuits. In each of these figures the table in Fig (a) gives the time taken to add the ILA core of particular configuration using different methods which include Normal Build, Fast Build, Guided PAR and the technique of ILA core addition using BMOD which utilizes Jbits and Jroute and developed as a part of this thesis. Fig(b) in each of these figures shows the percentage improvement in time taken using BMOD over each of the other techniques for ILA core addition which include Fast Build, Normal Build and Guided PAR. The graphs in Fig (c) shows in a visual format the time taken for adding ILA core to the design using all the four methods. Similarly the graphs in Fig(d) show a visual picture of the improvement in the ILA addition time using BMOD over the other techniques. With this data available we will try to perform a detailed analysis.

	Fast Build	Normal Build	Guided PAR	BMOD
B12	50	102	60	16.81
B14	72	165	79	21.5
B15	85	206	94	25.4
B17	221	755	234	46.72
B18	666	1956	612	100.44
B20	100	307	110	28.06
B21	102	311	108	28.12
B22	150	468	164	33.78

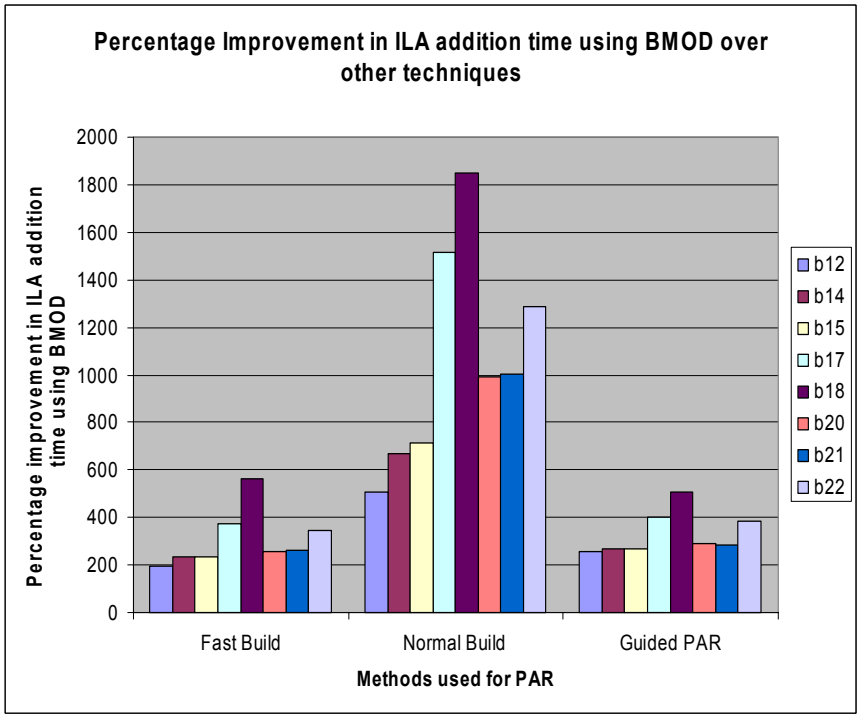
(a)



(b)

	Fast Build	Normal Build	Guided PAR
B12	197	506	256
B14	234	667	267
B15	234	711	270
B17	373	1516	400
B18	563	1847	509
B20	256	994	292
B21	262	1005	284
B22	344	1285	385

(c)

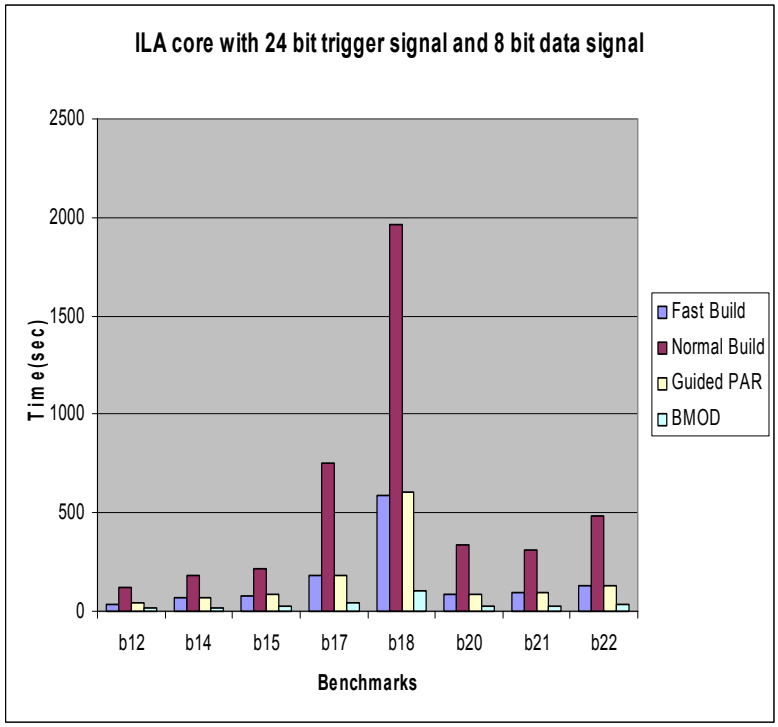


(d)

Fig 5.5 ILA core Addition time for ILA_8_16

	Fast Build	Normal Build	Guided PAR	BMOD
B12	37	117	44	16.54
B14	70	181	73	21.03
B15	74	213	84	25.86
B17	179	753	178	47
B18	585	1965	605	101.42
B20	84	336	83	27.93
B21	93	310	97	27.38
B22	128	488	133	33.76

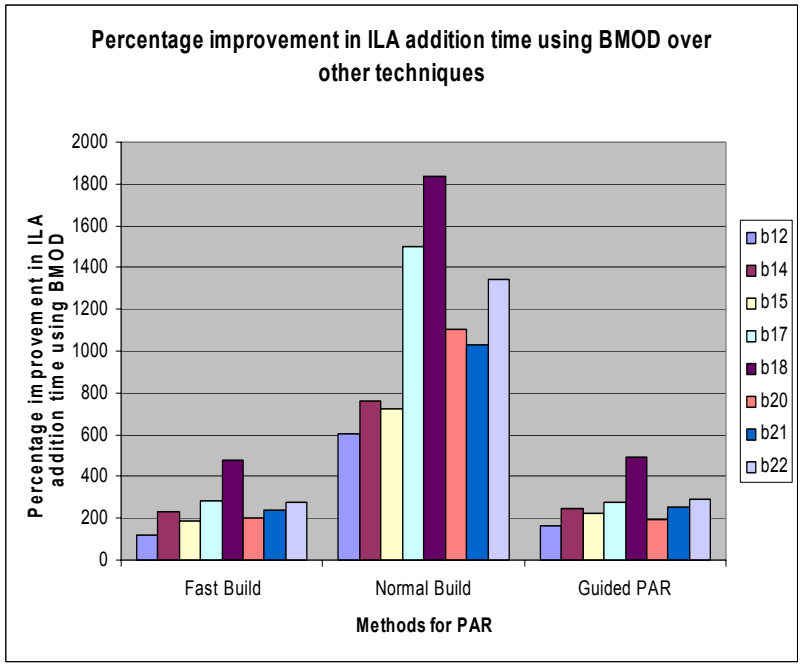
(a)



(b)

	Fast Build	Normal Build	Guided PAR
B12	123	607	166
B14	232	760	247
B15	186	723	224
B17	280	1502	278
B18	476	1837	496
B20	200	1103	197
B21	239	1032	254
B22	279	1345	293

(c)

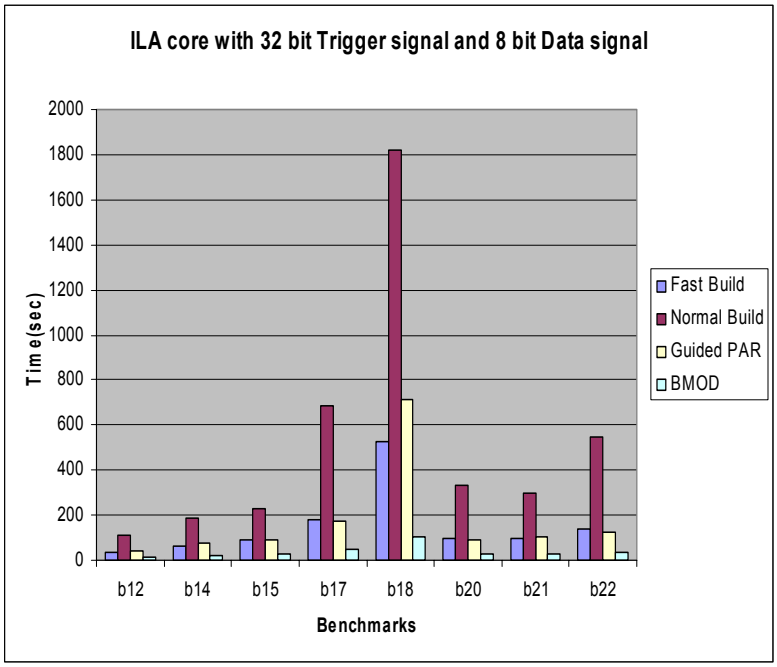


(d)

Fig 5.6 ILA core Addition time for ILA_8_24

	Fast Build	Normal Build	Guided PAR	BMOD
B12	35	109	43	17.08
B14	65	184	73	21.55
B15	87	227	87	25.8
B17	180	688	174	47.68
B18	529	1823	714	102.48
B20	97	331	91	28.13
B21	98	301	104	27.68
B22	138	545	125	34.18

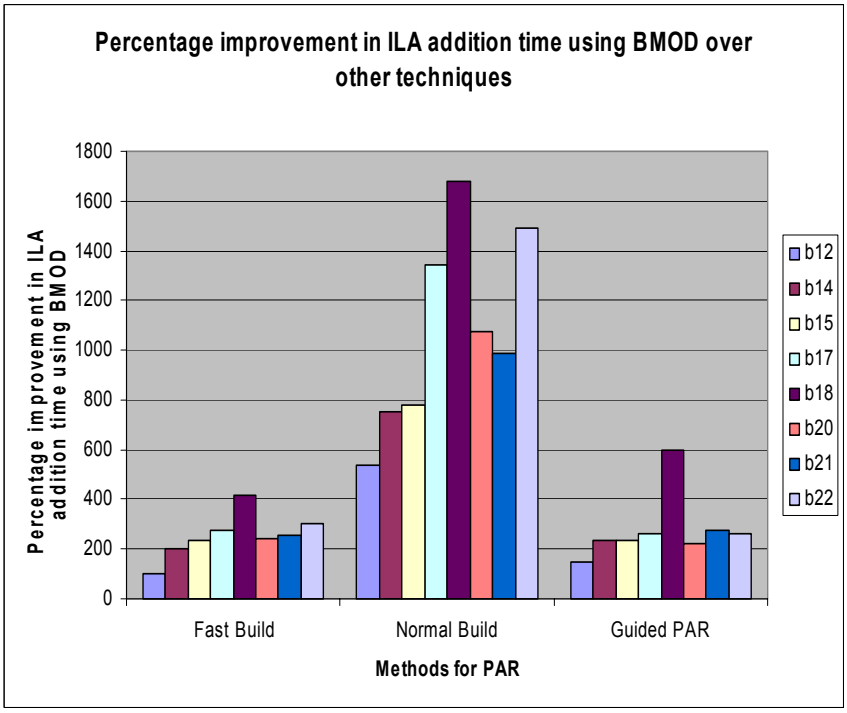
(a)



(b)

	Fast Build	Normal Build	Guided PAR
B12	104	538	151
B14	201	753	238
B15	237	779	237
B17	277	1342	264
B18	415	1678	596
B20	244	1076	223
B21	254	987	274
B22	303	1494	265

(c)

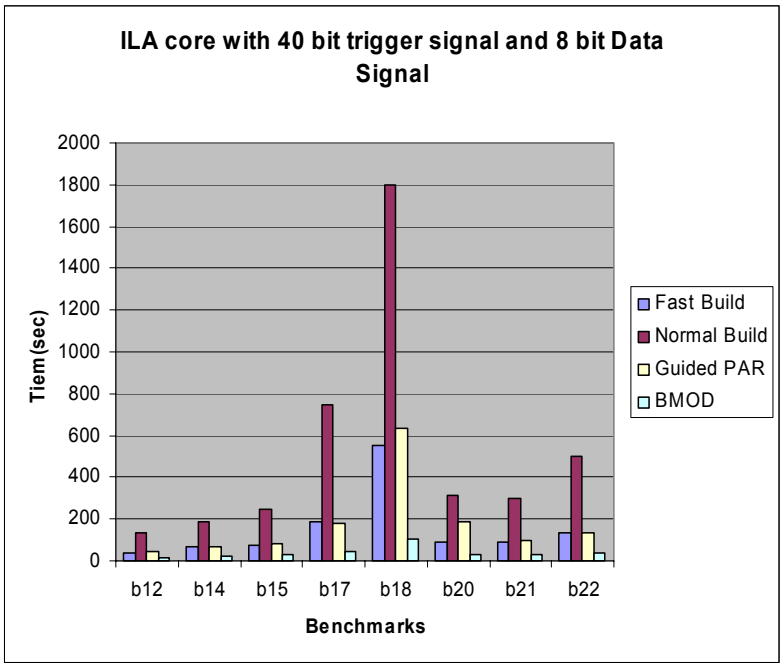


(d)

Fig 5.7 ILA core Addition time for ILA_8_32

	Fast Build	Normal Build	Guided PAR	BMOD
B12	38	136	44	17.48
B14	64	185	67	21.32
B15	73	247	83	26.28
B17	183	746	181	48.11
B18	554	1801	638	101.63
B20	93	315	185	28.38
B21	92	295	98	27.91
B22	131	500	134	34.45

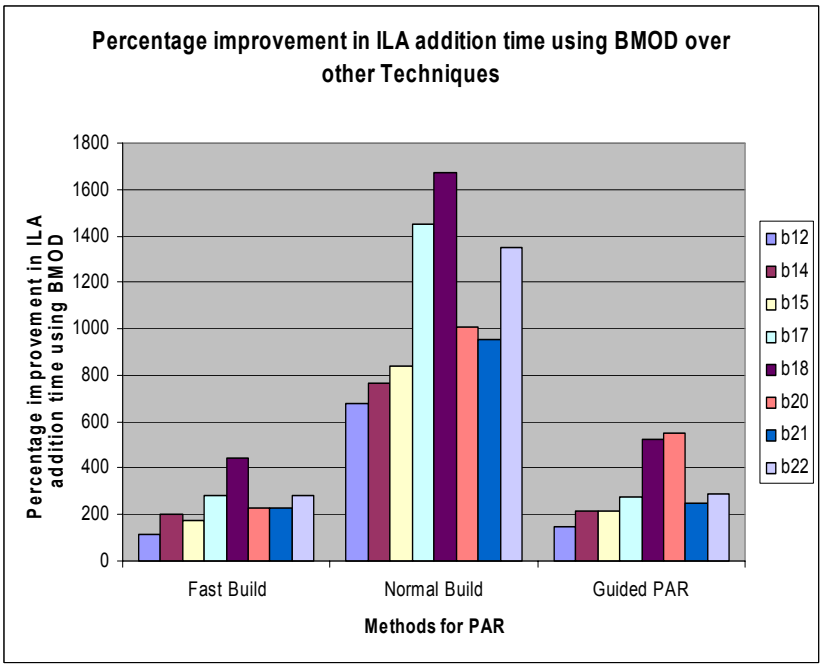
(a)



(b)

	Fast Build	Normal Build	Guided PAR
B12	117	678	151
B14	200	767	214
B15	177	839	215
B17	280	1450	276
B18	455	1672	527
B20	227	1009	551
B21	229	956	251
B22	279	1351	288

(c)

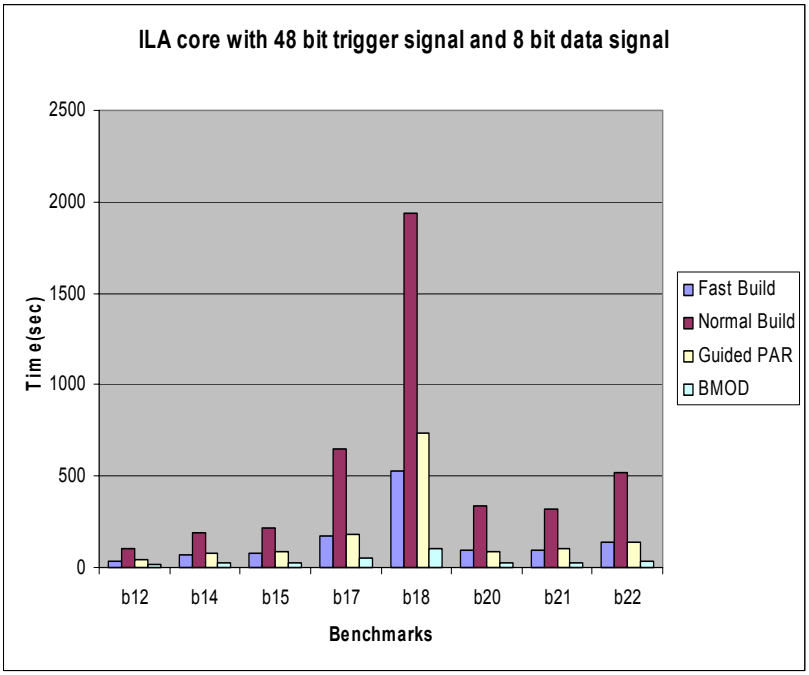


(d)

Fig 5.8 ILA core Addition time for ILA_8_40

	Fast Build	Normal Build	Guided PAR	BMOD
B12	35	106	47	17.64
B14	71	190	76	21.96
B15	82	219	88	26.48
B17	175	646	178	48.92
B18	529	1934	736	102.53
B20	95	341	85	29.48
B21	97	320	100	28.56
B22	136	521	136	34.6

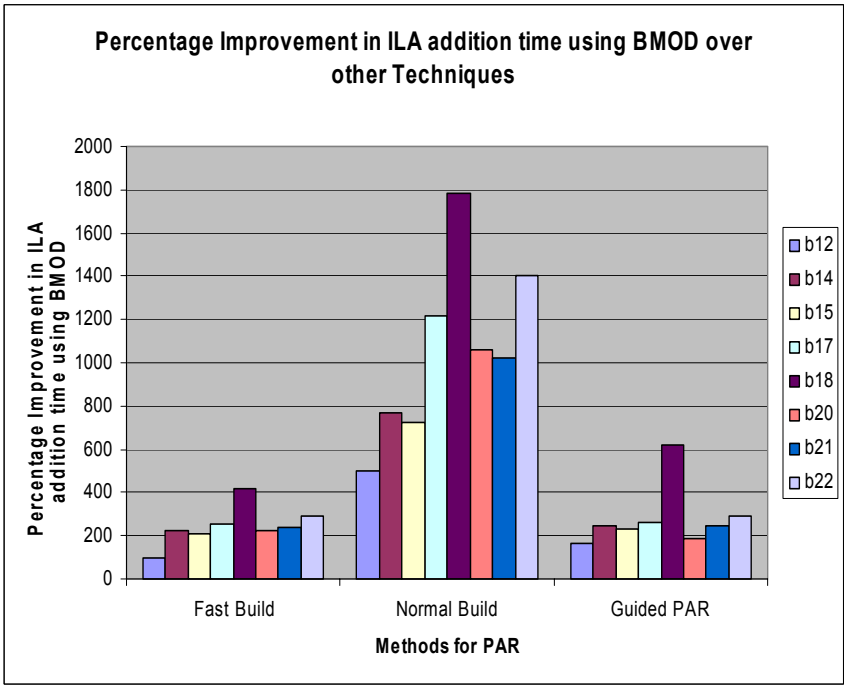
(a)



(b)

	Fast Build	Normal Build	Guided PAR
B12	98	500	166
B14	223	765	246
B15	209	727	232
B17	257	1220	263
B18	415	1786	617
B20	222	1056	188
B21	239	1024	250
B22	293	1405	293

(c)

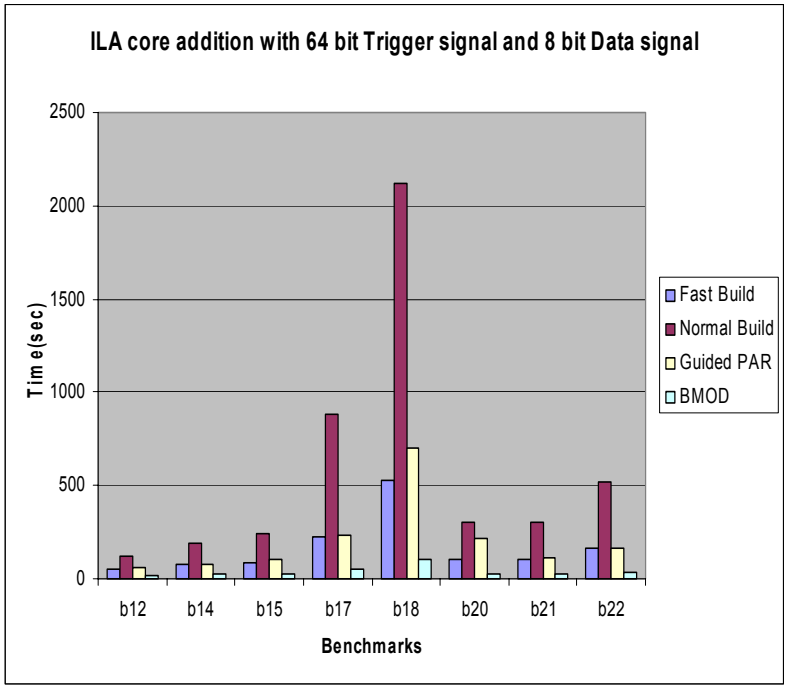


(d)

Fig 5.9 ILA core Addition time for ILA_8_48

	Fast Build	Normal Build	Guided PAR	BMOD
B12	51	118	61	17.97
B14	80	191	81	22.44
B15	90	239	101	26.57
B17	226	880	231	49.67
B18	525	2122	705	102.97
B20	104	307	213	29.05
B21	106	303	110	28.57
B22	163	515	164	35.18

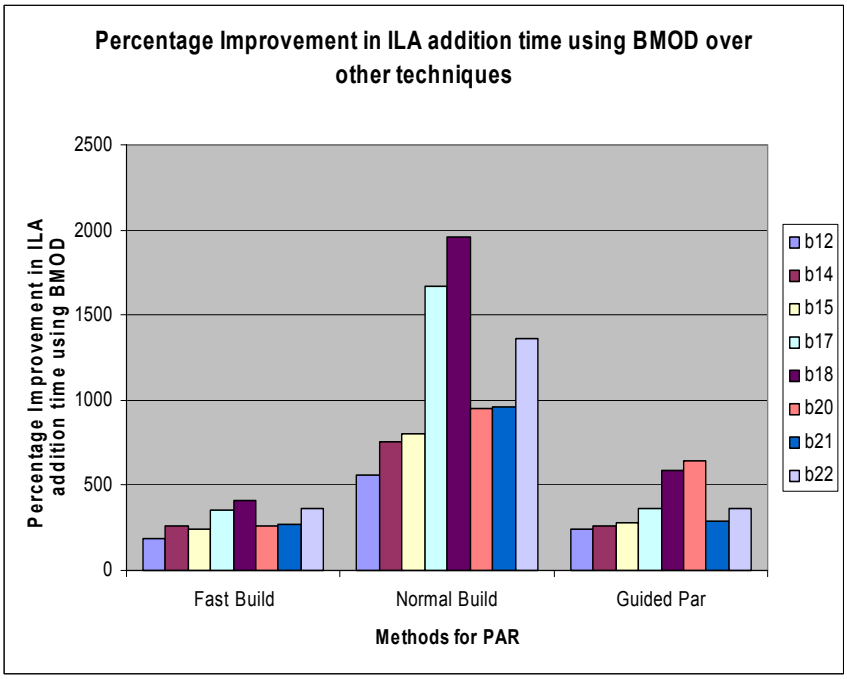
(a)



(b)

	Fast Build	Normal Build	Guided PAR
B12	183	556	239
B14	257	751	260
B15	238	799	280
B17	355	1671	365
B18	409	1960	584
B20	258	956	646
B21	271	960	285
B22	363	1363	366

(c)

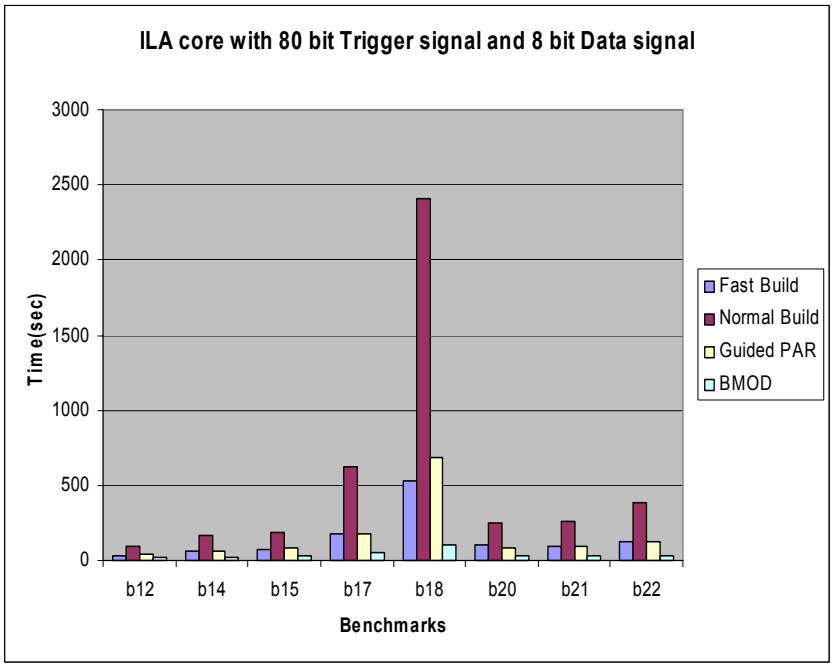


(d)

Fig 5.10 ILA core Addition time for ILA_8_64

	Fast Build	Normal Build	Guided PAR	BMOD
B12	35	97	45	18.69
B14	64	163	67	23.07
B15	75	190	84	26.94
B17	179	624	179	49.34
B18	530	2405	682	103.14
B20	99	251	84	29.31
B21	91	257	91	29.21
B22	123	388	120	35.93

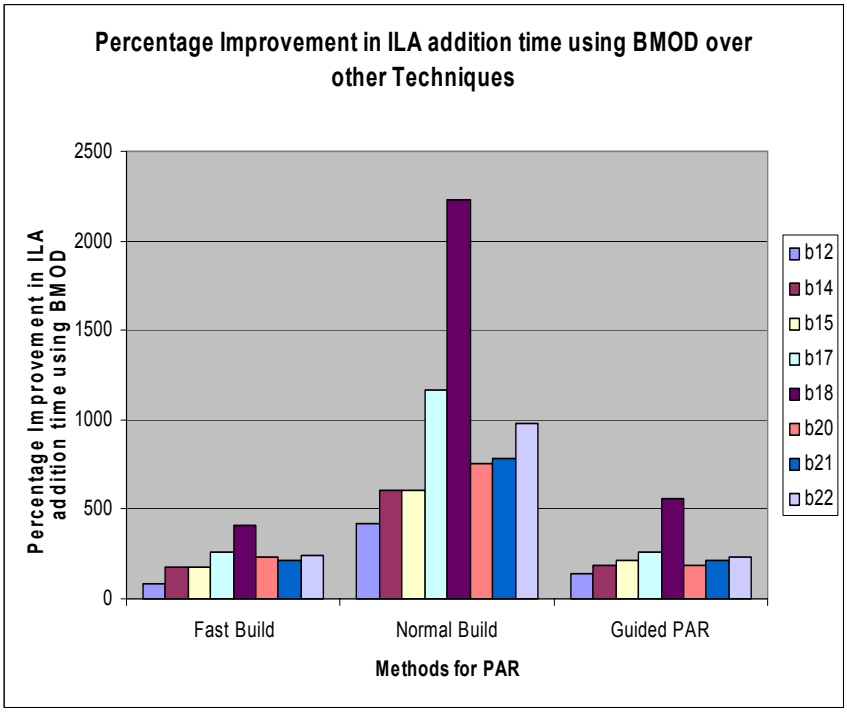
(a)



(b)

	Fast Build	Normal Build	Guided PAR
B12	87	418	140
B14	177	606	190
B15	178	605	211
B17	262	1164	262
B18	413	2231	561
B20	237	756	186
B21	211	779	211
B22	242	979	233

(c)

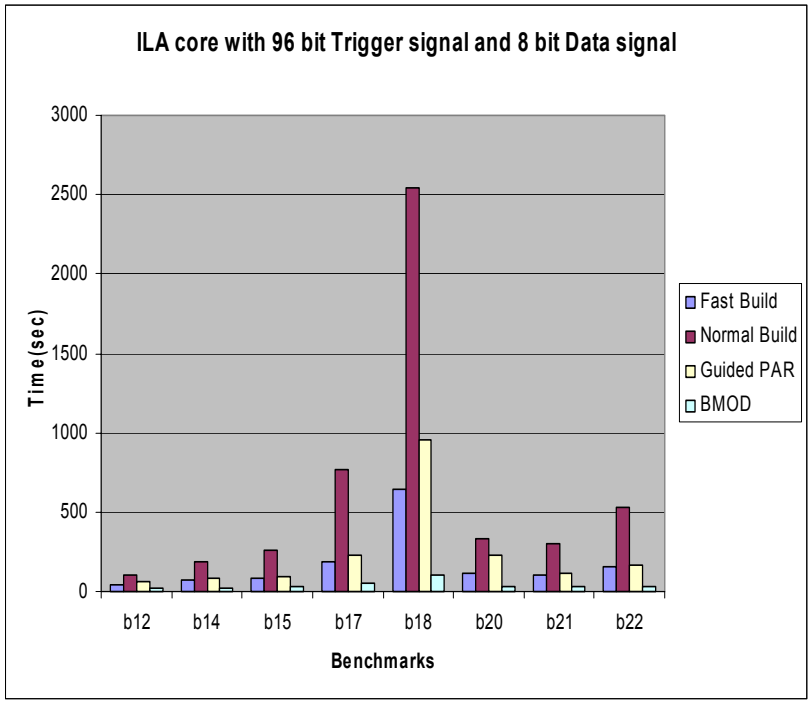


(d)

Fig 5.11 ILA core Addition time for ILA_8_80

	Fast Build	Normal Build	Guided PAR	BMOD
B12	40	108	63	19.59
B14	70	184	80	24.16
B15	82	259	97	27.77
B17	185	771	232	50.11
B18	647	2545	958	105.1
B20	111	331	225	30.29
B21	104	305	109	28.52
B22	156	525	168	36.15

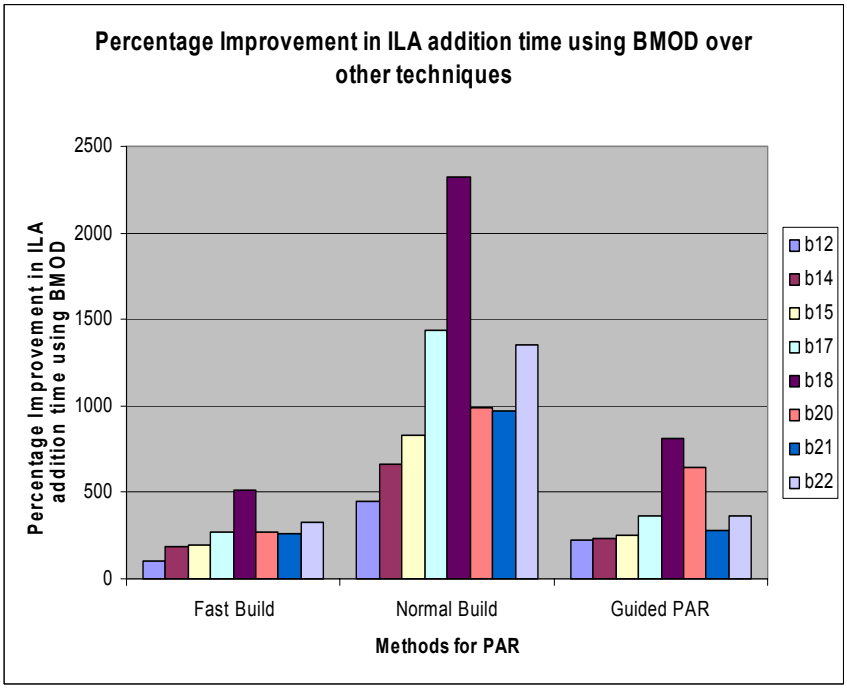
(a)



(b)

	Fast Build	Normal Build	Guided PAR
B12	104	451	221
B14	189	661	231
B15	195	832	249
B17	269	1438	362
B18	515	2321	811
B20	266	992	642
B21	264	969	282
B22	331	1352	364

(c)



(d)

Fig 5.12 ILA core Addition time for ILA_8_96

5.4 Analysis of the results: From the results it is observed that the time taken for the addition of the ILA cores using BMOD is quite a bit less than the time taken by the other three methods. It is also observed that as the complexity of the design increase the percentage improvement in the time taken also increases. It is observed form the graphs in Fig(d) for each of the ILA configurations, b18 which is the most complex design utilizing 72% of the FPGA resources has the highest percentage of improvement in terms of time taken fro ILA core addition using Jbtis and Jroute over other methods. This is highly desired as designs which are complex require more number of debugging iterations with different debugging capabilities during each iteration such as the change in the width of the trigger and data bus, the change in the buffer depth, the change in the signals being monitored, etc during the debugging process. Thus if the designer is able to make these desired changes in less time the debugging process can be made more efficient. Towards achieving this goal bitstream modification of designs for ILA core addition and modification can be an effective and desired technique for a Hardware Debugging environment.

Chapter 6
Conclusion and Future Work

6.1 Work done as a part of this thesis:

In this thesis we have developed debugging techniques that will result in improving the debugging capabilities of designs implemented on FPGA Based Systems. These techniques facilitate the process of verification and debugging and reduce the debugging time to achieve early time to market goals. Integrated Logic Analyzer Cores can be added to the FPGA based designs to improve their execution control. In this thesis we have developed a technique to enable addition of such cores at the last step of the FPGA design flow. This can ease and expedite the process of debugging designs implemented on FPGA based systems by improving their execution control, reducing the time taken to modify the debugging functionality and hence aid in achieving early time to market goals. The techniques developed in this thesis can be part of a comprehensive Debugging Environment which incorporates many other debugging capabilities in addition to the techniques developed as a part of this research.

The following subsections will summarize the techniques developed.

6.1.1 Symbol Table Creation:

As a part of this research the information of logical to physical design mapping is provided in the form of a symbol table. The symbol table contains information of all the logical instances in the design with their input and output ports, the nets connected to those ports and their detailed mapping to the physical components in the FPGA. This process of symbol table creation utilizes information from files generated by both the

synthesis tool and the Xilinx placement and routing tool. As a part of this research this process of symbol table creation has been automated. The information contained in this symbol table in utilized in the techniques developed in this thesis to improve the debugging capabilities of the designs.

6.1.2 ILA core addition through Bitstream Modification:

Inorder to improve the observibility and execution control of designs Integrated Logic Analyzer (ILA) cores which are a specialized form of logic are added at the bitstream stage using the bitstream modification tools called JBits and JRoute. The main components of an ILA core include trigger logic, control logic and data buffer. In the ILA created in this project, trigger signals from the design are monitored by the trigger logic and the value on the data line is saved into the data buffer on the occurrence of the user specified trigger condition. The width of the trigger, the width of the data signals and the depth of the data buffer can be changed as desired. The process of ILA core addition or modification also utilizes the information in the symbol table which is developed as a part of this research. Once the ILA cores are added to the design using JBits the designer selects the data and trigger signals in the logical view and the corresponding signals in the physical FPGA are connected to the ILA cores using JRoute. As a part of this research we have automated the process of ILA core generation. We have also developed and implemented a placement algorithm which is used to add the ILA cores into the design using the free resources in the FPGA. The main contribution of this research was the demonstration that these ILA cores can be added very late in the design process so

that a considerable amount of time can be saved when compared to performing the same task using other techniques.

The process of ILA core generation and addition was automated as a part of this research. From the experiments that were conducted it was observed that the time taken to add these ILA cores to the FPGA design was quite small. On average the technique developed took 8-10 times less time for ILA core addition when compared to the other techniques. In the case of complex designs the time saved was even more significant

6.2 Future Work: In this section we will try to give a brief description of the direction of future work.

6.2.1 Improving the features supported by the ILA: The ILA cores developed and tested in this research helps in improving the observibility and execution control of the design. They support a limited number of debugging capabilities like observing the state of various signals in the design and saving the sates of signal on the occurrence of certain trigger condition. ILA cores with more debugging features like clock control so that single and multi stepping of clock is supported, forcing the values of signals in the design, etc can be supported so that the controllability, observibility and execution control of the design can be improved.

6.2.2 Developing Compact and flexible ILA cores: The ILA cores generated can be made more compact and flexible by using the features supported by the Virtex FPGA. For instance an LUT can also act as a shift register. This feature can be used, so that trigger logic to support varying trigger conditions can easily be implemented [35]. In case

of complex designs which occupies almost the entire FPGA it is desirable to generate ILA cores which are compact, as more number of signals are to be monitored and less space is available to place these cores.

6.2.3 Developing better Algorithms for ILA core placement: The placement algorithm that is developed as a part of this research to place the ILA cores in the design can be improved in many ways. It is desired that the ILA cores be placed near to the signals being monitored. This is important to prevent the data or trigger signal connected to the ILA from becoming a part of the critical path. Also in the case of complex designs where there is a scarcity of free resources algorithms should be developed which can utilize these free resources in the best possible way while adding the ILA cores.

6.2.4 Developing a integrated Tool: In this thesis the process of Logical to physical design mapping was implemented in C using the Unix utilities Lex and Yacc while the addition of ILA cores is performed using JBits and JRoute which are Java based API. Due to the incompatibilities of these environments intermediate files are generated to pass the information from one environment to another. This results in wasted time and effort. So it is highly desirable to perform the task of symbol table creation and ILA core addition in a single program so that this overhead can be avoided.

6.2.5 Developing a GUI to ease the process of debugging: A GUI can be developed to ease the debugging process for the designer. The GUI can be used to provide the designer with a logical view of the design so the desired signals to be monitored can be selected

and observed. The designer can use this interface to enter the desired configuration of the ILA such as the depth of the storage buffer, the width of the data and trigger signals, etc. An interface can also be provided for observing the content of the storage buffer during design execution through partial readback as well.

Bibliography:

[1] Xilinx, “Virtex 2.5 V field programmable gate arrays: Module 2, detailed functional description”, Datasheet DS003-2, Xilinx, San Jose, CA, April 2001, v. 2.5.

[2] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software" ,in the proceedings of ACM Computing Surveys, June 2002, Vol. 34, No. 2. pages 171-210.

[3] Wo, D.; Forward, K.;”Compiling to the gate level for a reconfigurable co-processor”, in the proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, 1994, pages 147 -154.

[4] Xilinx, “Virtex FPGA series configuration and readback”, Application Note XAPP138, Xilinx, San Jose, CA, July 2002.

[5] Xilinx, “Virtex series configuration architecture user guide”, Application Note XAPP151, Xilinx, San Jose, CA, September 2000.

[6] S. A. Guccione, D. Levi, and P. Sundararajan, “JBits: A Java-based interface for reconfigurable computing”, in the proceedings of Second Annual conference on Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD), September 1999.

[7] Xilinx, “Virtex-II 1.5 V field programmable gate arrays: Module 2, detailed functional description”, Datasheet DS031-2, Xilinx, San Jose, CA, January 2001, v. 1.3.

[8] Altera Corporation, San Jose, CA, APEX 20K Programmable Logic Device Family Data Sheet, ver. 2.06 edition, March 2000.

[9] Altera Corporation, San Jose, CA, SignalTap User’s Guide, 1999.10 (revision 2) edition, November 1999.

[10] Altera Corporation, San Jose, CA, SignalTap Embedded Logic Analyzer Megafunction Data Sheet, ver. 1.01 edition, January 2000.

[11] Xilinx, San Jose, CA, ChipScope Software and ILA Cores User Manual, v. 4.1 edition, October 2001.

[12] Xilinx, San Jose, CA, ChipScope Software Tools tutorial, v. 4.1 edition, October 2001.

[13] B. K. Fross, R. L. Donaldson, and D. J. Palmer, “Pci-based WILDFIRE reconfigurable computing engines”, in the proceedings of SPIE—The International Society for Optical Engineering, Bellingham, WA, November 1996, vol. 2914, pages 170–179.

[14] L. Moll and M. Shand, “Systems performance measurement on PCI pamette”, in the proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, Apr. 1997, pages 125–133.

[15] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, “Building and using a highly parallel programmable logic array”, In the proceeding of IEEE transaction on Computer, Jan 1991, vol. 24, no. 1, pages 81–89.

[16] J. M. Arnold, D. A. Buell, and E. G. Davis, “Splash 2”, in the proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992, pages 316–324.

[17] R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider, “Teramac–configurable custom computing”, in Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, Apr.1995, pages 32–38.

[18] B. Schott, S. Crago, C. Chen, J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, “Reconfigurable architectures for systems level applications of adaptive computing”, In the proceedings of IEEE transaction on VLSI Design, 2000, vol. 10, no. 3, pages 265–279.

[19] P. Bellows and B. L. Hutchings, “JHDL—an HDL for reconfigurable systems”, in the proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, Apr. 1998, pp. 175–184.

[20] Xilinx, San Jose, CA, BoardScope User’s Guide, 2.8.1 edition, Oct 2001, Part of the HTML documentation provided with JBits 2.8.1.

[21] Hutchings, B.; Nelson, B.;”Developing and debugging FPGA applications in hardware with JHDL“, in the proceedings of Conference on Signals, Systems, and Computers, 1999. Volume: 1, pages 554 -558.

[22] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, “A CAD suite for high-performance FPGA design”, in the proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, April 1999, pages 12–24.

[23] Hutchings, B.L.; Nelson, B.E.;" Unifying simulation and execution in a design environment for FPGA systems", In the proceedings of IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Feb 2001, Volume: 9 Issue: 1 , pages 201-205.

[24] Graham, P.; Hutchings, B.; Nelson, B.; "Improving the FPGA design process through determining and applying logical-to-physical design mappings" In the proceedings of Conference on Field-Programmable Custom Computing Machines, 2000, pages 305 -306.

[25] Paul Graham," Logical Hardware Debuggers for FPGA-Based Systems", PhD Thesis, Brigham Young University, Electrical and Computer Engineering Department, December 2001

[26] T. Wheeler, "Improving design observability and controllability for circuit debugging in FPGAs using design-level scan techniques", Master's thesis, Brigham Young University, Provo, UT, 2001.

[27] Paul Graham, "Instrumenting Bitstreams for Debugging FPGA Circuits", in the Proceedings IEEE workshop on FPGA for Custom Computing Machines, 2001.

[28] Steven A. Guccione and Delon Levi, "Run-Time Parameterizable Cores", In the Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications, August / September 1999, pages 215-222.

[29] Scott P. McMillan, Brandon J. Blodget and Steven A. Guccione, "VirtexDS: A Device Simulator for Virtex", In the Proceedings of SPIE - The International Society for Optical Engineering, Bellingham, WA, November 2000, pages 50-56.

[30] Tim Price, Delon Levi and Steven A. Guccione, "Debug of Reconfigurable Systems", In the Proceedings of SPIE - The International Society for Optical Engineering, Bellingham, WA, November 2000, pages 181-187.

[31] Delon Levi and Steven A. Guccione," BoardScope: A Debug Tool for Reconfigurable Systems". In the proceedings of SPIE- The International Society for Optical Engineering, Bellingham, WA, November 1998, pages 239-246.

[32] S. Z. Hanono, "Innerview hardware debugger: A logic analysis tool for the virtual wires emulation system", Master's thesis, Massachusetts Institute of Technology, February 1995.

[33] Karen A. Tomko, Anurag Tiwari, "Design Techniques to Implement Reconfigurable Hardware Watch-Points for Hardware/Software Co-Debugging", In the Proceeding of conference on Engineering of Reconfigurable Systems and Algorithms, June 2001.

[34] Anurag Tiwari, "Hardware/software Co-debugging for Reconfigurable Computing Applications", M.S thesis, University of Cincinnati, Dept of ECECS, November 2001.

[35] A. Tiwari, K.A. Tomko, "Scan-chain Based Watch-points for Efficient RunTime Debugging and Verification of FPGA Designs", Proceeding of the ASPDAC, Jan 2003.

[36] Corno, F.; Reorda, M.S.; Squillero, G. ,"RT-level ITC'99 benchmarks and first ATPG results", in the IEEE Transaction on Design & Test of Computers , Volume: 17, Issue: 3, July-Sept. 2000, pp. 44 -53

[37] Keith D. Cooper, John Bennett, Linda Torczon, "Optimizing VHDL Intermediate Forms", Final Report, http://www.cs.rice.edu/~keith/VHDL/FinalReport.pdf

[38] Annapolis Micro Systems, Annapolis, MD, Addendum: WILDSTAR Reference Manual, Version 3.3 and STARFIRE Reference Manual Version 2.2, Topic: Readback, March 2000.

[39] S. Hauck, The Role of FPGAs in Reconfigurable Systems, Proceedings of the IEEE, Vol. 86, No. 4, April, 1998, pp. 615-638.

[40] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, P. Olsen, "Splash: A Reconfigurable Linear Logic Array", International Conference on Parallel Processing, 1990, pp. 526-532.

[41] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", IEEE Transactions on VLSI Systems, Vol. 4, No.1, March, 1996, pp 56-69.

[42] R. Tessier and W. Burleson, "Reconfigurable Computing and Digital Signal Processing: A Survey", In the Journal of VLSI Signal Processing, May/June 2001.

[43] S. Walters, "Computer-aided prototyping for asic-based systems", IEEE Design and Test of Computers, vol. 8, no. 2, June 1991, pp. 4–10.

[44] J. Varghese, M. Butts, and J. Batcheller, "An efficient logic emulation system", IEEE Transactions on Very Large Scale Integration Systems, vol. 1, no. 2, June1993, pp. 171–174.

[45] Krishnamachary, A.; Abraham, J.A.; Tupuri, R.S., "Timing verification and delay test generation for hierarchical designs", Fourteenth International Conference on VLSI Design, 2001, pp 157 -162.

[46] Lach, J.; Mangione-Smith, W.H.; Potkonjak, M. "Efficient error detection, localization, and correction for FPGA-based debugging**",** In Proceedings of the 37th Design Automation Conference, 2000, pp 207 -212.

[47] Miron Abramovici, Melvin A. Breuer, Arthur D. Friedman, "Digital Systems Testing and Testable Design", IEEE Press, 1990.

[48] Walters, S., **"**Reprogrammable hardware emulation for ASICs makes through design verification practical" , Proceedings of COMPCON Spring Conference, 1989, pp. 484 -486