I, Kalyan Bemalkhedkar, hereby submit this work as part of the requirements for the degree of:
Master of Science
in:
Computer Science

It is entitled:
"An Object Oriented Framework for Bitstream Generation for Multi-Technology FPGA Chips"

This work and its defense approved by:

Chair: Prof. Karen Tomko
Prof. Fred Beyette
Prof. Carla Purdy
An Object Oriented Framework For Bitstream Generation For Multi-Technology FPGA Chips

A thesis submitted to the

Division of Research and Advanced 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

April 2005

By

Kalyan Bemalkhedkar
B.E. (CS), Shri G. S. Institute of Tech. and Sc., Indore, India, June 2002

Thesis Advisor and Committee Chair: Dr. Karen A. Tomko
Abstract

Technologies in FPGA (Field Programmable Gate Array) chips have rapidly evolved over the past few decades. The architectures of these FPGA chips are constantly evolving. To increase their applicability, newer analog, digital and mixed signal components are been integrated. The software tool chain used to program application specific functionality onto these chips also needs to evolve simultaneously to support these new components and architectures.

This software tool chain starts with high level design entry and is completed by the generation of bitstream for programming the chip. In this thesis, we present a novel object oriented framework approach for bitstream generation: XBits. XBits takes in a placed and routed circuit description and generates the bitstream for an FPGA chip whose architecture is specified in our custom Architecture Description File Format. This format allows us to specify varying FPGA architectures and components which can be easily integrated into rest of the framework.
Acknowledgements

This thesis is a result of a joint effort of more than one person. One of the foremost of the people behind my work is Prof. Karen Tomko, my graduate advisor. She has always been the driving force and inspiration for my thesis and without her support and guidance I would not have been able to achieve what I have.

Secondly, I would like to extend my thanks to my friend and project partner, Vishal Sapre, for his excellent collaborative effort in the development of “XBits”. I have been working with him for the past one and half years and have found him to be an awesome team player, moral support and above all a great friend. His insights in the hardware and design aspects of the MT-FPGA, went a long way to shape our team effort. I would like to wish him all the best with his thesis and future.

I am also indebted to the department of Electrical, Computer Engineering and Computer Science at University of Cincinnati. The lab and other resources provided by them to us have been invaluable towards our thesis work.

A big gratitude goes to my lab-mates Sachin Gandhi, Amol Deshmukh, Amr Ahmadain, Anurag Tiwari and Liu Qingyuan for their support and helpful suggestions.

And finally, I would like to thank my parents and my brother without whom I would not have been in the place I am.
# TABLE OF CONTENTS

## 1. INTRODUCTION

## 2. BACKGROUND

2.1. FPGA Bitstream Generation ................................................................. 6
2.2. Overview of Bitstream Generation for MT-FPGA .......................... 9
2.3. MT-FPGA Overview ................................................................. 10
2.4. Related Work ................................................................. 11

## 3. DESIGN

3.1. Circuit Description Generation ......................................................... 19
3.2. Circuit Object Hierarchy ............................................................. 19
3.3. Architecture Hierarchy Traverser .................................................. 20
3.4. Architecture Hierarchy Traverser ↔ Circuit Object Hierarchy Interface .... 22
3.5. Bit Stream Generator ............................................................... 23
3.6. Advantages of the Framework Approach .................................. 23
3.7. Architecture Description File Format .......................................... 26
3.8. Summary ................................................................. 29

## 4. IMPLEMENTATION

4.1. Generation of Bits ................................................................. 30
4.2. XBits Framework Implementation ............................................. 31
   4.2.1. Execution Flow ............................................................. 31
   4.2.2. DOM (Document Object Model) ...................................... 33
   4.2.3. genBits() ................................................................. 34
   4.2.4. Example Run-Through for genBits() ................................ 38
   4.2.5. processComponent() Interface .................................... 46
   4.2.6. Bitstream Generator .................................................. 50
4.3. Utility Features of the Architecture Description File Format ............ 52
   4.3.1. Ranging ................................................................. 53
   4.3.2. Offsetting ............................................................... 54
   4.3.3. Referencing ............................................................ 54
4.4. Integrating New Chip Components .............................................. 56
4.5. Summary ................................................................. 58

## 5. RESULTS AND FUTURE WORK

5.1. Test Environment ................................................................. 60
5.2. Architecture Description Files .................................................. 60
5.3. Circuit Description (BLIF) files .................................................. 61
5.4. VPR Output ................................................................. 63
5.5. XBits Output (Bitstream) ......................................................... 65
5.6. Performance Comparison ......................................................... 67
5.7. Future Work ................................................................. 70
5.8. Summary ................................................................. 72

## 6. CONCLUSION

## 7. REFERENCES
TABLE OF FIGURES

FIGURE 2-1 DESIGN FLOW FOR FPGA PROGRAMMING .......................................................................................... 7
FIGURE 2-2 LOGICAL BITSTREAM GENERATION FLOW ...................................................................................... 9
FIGURE 2-3 HIERARCHIAL VIEW OF THE MTL (ADAPTED FROM [39]) .................................................................. 11
FIGURE 3-1 XBits Framework Components Overview .................................................................................... 18
FIGURE 3-2 Architecture Description File Format ............................................................................................ 27
FIGURE 3-3 MT-FPGA Architecture Description File .................................................................................... 28
FIGURE 4-1 genBits() Flowchart .................................................................................................................. 37
FIGURE 4-2 Example Run-Through DOM Tree ................................................................................................. 39
FIGURE 4-3 Example Run-Through Architecture Description File .................................................................. 41
FIGURE 4-4 ProcessComponent() (FPGA) ...................................................................................................... 49
FIGURE 4-5 Architecture Description File (without using ranging) .............................................................. 53
FIGURE 4-6 Architecture Description File (using ranging) .......................................................................... 53
FIGURE 4-7 Example (Offsetting) .................................................................................................................. 54
FIGURE 4-8 Example (Referencing) ................................................................................................................ 55
FIGURE 4-9 DOM tree structure showing referenced CLB Node .................................................................. 56
FIGURE 5-1 b1.blif taken from the MCNC benchmark .................................................................................... 61
FIGURE 5-2 b1replicated.blif (Replicated Version of b1.blif) ........................................................................ 63
FIGURE 5-3 VPR Graphical Output for b1.blif mapped onto a 3 x 3 Architecture ............................................ 64
FIGURE 5-4 VPR Graphical Output for b1replicated.blif mapped onto a 5 x 5 Architecture ............................. 64
FIGURE 5-5 Generated bitstream for b1.blif mapped onto a 3 x 3 Architecture ............................................. 65
FIGURE 5-6 Generated bitstream for b1replicated.blif mapped onto a 5 x 5 Architecture ............................. 66
FIGURE 5-7 Execution Delay for the whole XBits Framework for the test cases .............................................. 68
FIGURE 5-8 Heap Allocation for the 3 x 3 Architecture (using b1.blif) ............................................................ 69
FIGURE 5-9 Heap Allocation for the 5 x 5 Architecture (using b1replicated.blif) ............................................ 69
FIGURE 5-10 Performance Comparison Summary between 3 x 3 Architecture and 5 x 5 Architecture Experiment Cases .............................................................................................................. 70
1. Introduction

Traditional design flow for contemporary FPGA design broadly consists of the following phases: the Front End Design, Synthesis and Programming. Front End Design primarily deals with conversion of higher level specifications of the circuit (HDL etc.) to logic gate arrays. The Synthesis phase is mainly involved in mapping, packing, placement and routing of these logic gate arrays into physically feasible component-layouts and designs. And last, but not the least, the Programming phase is responsible for storing the higher level circuit description into the layouts. While there has been a lot of focus on the first two phases, Programming (Bit Stream Generation) has not had significant research interest due to proprietary nature of the information involved and less computational complexity. Also since the available systems in this area are also proprietary, they are build for specific chips and do not support multiple architectures. In building the XBits framework, we attempt a different approach at bit stream generation which aims to eliminate this rigidity associated with bit stream generation process and attempt to create a framework which is generic enough to support changes to the chip architecture.

XBits has evolved from an application specific tool for generating bit streams for a particular chip to more generic base infrastructure for such kind of tools. One of our main design goals behind XBits has been to create a framework which would allow easy extension and integration of newer components with varied behaviors and characteristics. It is indeed a daunting task to integrate functionality for generating bit streams for all these components and account for newer components in one single
specific tool. Instead, what we provide is a framework which has substantial functionality built into it but in a way such that it is flexible enough to accommodate newer components by providing different implementation to specific parts dealing with bit generation process of the new component. This does not in any way mean that XBits is not a standalone tool in itself; rather it is a generic toolkit which can be used as is or extended to support required components with specific behavior.

One of the most prominent features of XBits is that it is an Object Oriented framework. A large number of existing of tools, that cater to the FPGA Configuration Design Flow (especially bitstream generation) have been based on procedural and sequential programming methodologies. This approach has restricted the potential area in which the code base of the tools can be applied to a related problem, architecture or signal domain. Making the code of these tools adapt to newer capabilities of the FPGA chips has been a very complex and involving task. Most of the time, it has been much more feasible to start a new tool from scratch rather than attempting to reuse the legacy code base of these existing tools. Also the inherent nature associated with the use of procedural language has led to a complex code for these tools which is hard to debug or modify in case of bugs or updates. To eliminate these potential disadvantages for the future the XBits framework was built using an Object Oriented Programming Language (JAVA more specifically). This approach has lent itself well to make the design and implementation of XBits more reusable and also platform independent. We have been able to leverage these benefits by constructing an object oriented framework from the ground-up which allows seamless re-use of existing component classes and infrastructure. In this way newer
components can be easily integrated into the XBits framework without the need for re-hauling the whole code base.

This thesis has been organized into the following chapters:

In Chapter 2, we provide some necessary background information required before leading into the further chapters on design and implementation. This information includes an overview of SRAM cell based FPGA chips and the design flow used for these chips, bitstream generation process in general and in the context of the MT-FPGA. We describe the physical structure of the MT-FPGA itself which is used as the target chip for the XBits framework. We also give a brief overview of some work done in related areas like bitstream generation (a different approach), bitstream compression and bitstream caching.

In Chapter 3, we will take a look some of the key design goals, used in the development of XBits framework. We will examine the various core components and analyze the interactions between these framework components. We would be referring to these broad level software components as framework components from here on, to prevent ambiguity with components on the chip (which will be called chip components). These components and the interaction between them cover most the bitstream generation process (once circuit description has been specified). We will also describe the XML Architecture Description File format that we have developed specifically for this framework to represent architecture description information of reconfigurable chips. We shall show how this format is capable of describing arbitrary chip architectures succinctly and precisely.
This discussion on the overview and design of the tool leads to the next chapter on the internals and implementation details. In Chapter 4, we will dwell into the details of how we went about implementing the major functionality of the bit stream generation process while adhering to our design guidelines. We have been able to reuse some legacy code-base from the JBits Interface [32]) into our XBits framework. The object oriented design of XBits has helped us to carry out this integration seamlessly. After this an in-depth view of the Architecture Hierarchy Traverser component is presented. We will also thoroughly look at the Circuit Object Hierarchy of chip components and see how it closely resembles the actual physical chip. Covering more details, we describe the `processComponent()` interface, a major architectural element of the framework. This will be followed by a discussion on how to integrate new components into the XBit implementation. All these implementation details give us a picture of the interactions and control transfer between framework components which build up a good understanding to the extensibility points of the framework for customizing in future.

We wrap up the discussion in earlier chapters by doing a broad level generic analysis of the XBits framework based on various factors like performance, memory usage etc. This analysis should prove helpful in making XBits a better tool in its future versions. Based on our analysis, we also suggest a few improvements and additional features that can be taken up as future work.

XBits framework can be viewed as a learning tool for understanding the process of bit stream generation in general as well as an instance of application for use with a specific chip. As said earlier, building all the functionality need to completely account
for any updates or changes to chip structure, is a difficult task to cover in the scope of an application specific effort. XBit does indeed have its limitations, but it opens itself to a more general API for bit stream generation by building upon independent framework components, loosely defining chip parameters, providing an extensive and flexible Architecture Description File Format and placing less constraints on internal and external interactions. I hope to see XBits framework been applied in digital, analog, mixed signal as well as other domains.
2. Background

In this chapter we present background information to preface the discussions on design and implementation of XBits framework. This also serves as a good foundation to the actual process of bitstream generation from the circuit description. We will briefly describe the standard design flow for configuring FPGA chips. In process, we will be looking at some existing tools and environments available for various stages in the design flow in both industry and academia. This will be followed by a general overview of the architecture of the MT-FPGA, the chip used as a target FPGA platform in the current version of XBits. Also, we will present some related work done in the field of FPGA bitstream generation.

2.1. FPGA Bitstream Generation

The FPGA bit stream generation design flow consists of a step by step process to go from the design and implementation of a high level circuit description to a form (bitstream) which can be used to actually configure an FPGA chip. In general, the FPGA design flow normally consists of the following phases (carried out in a sequential order) as shown in Figure 2-1. Note that this is a simplified view of the design flow. There may be other phases involved depending upon the implementation.
The design flow can be divided into three broad phases: Front End Design, Synthesis and Programming. Front End Design is more application specific and starts with the creation the logical design of the circuit in a high level description language like HDL, VHDL, Verilog etc. After checking for logical correctness of the new design using Simulation and Design Verification the development moves to the Synthesis phase. In the first step the design entry is converted into actual gates/blocks needed for physically implementing the design. This gate/block level description is represented using a standard net list format in the FPGA design flow. One of such formats is .blif (Berkeley Logic Interchange Format) [1]. The gate level description is usually flattened before passing to the next phase using a tool like FlowMap [2]. On
the commercial front, Xilinx Integrated Software Environment (ISE) [3] is a powerful and complex set of tools which besides providing strong support for front end design, covers the entire design flow.

Once the design has been mapped to logic gates, it is pipelined down to FPGA Technology Mapping stage. In this stage these gates are packed into logical blocks and placement and routing is done to select the optimal position for these blocks and minimize the length of interconnections on the device. A number of tools are available for this purpose which are both Commercial (Industry based) and Non-Commercial (Academia or Research based). The commercial popular tools include Xilinx ISE, Altera Quartus II etc. In the academic and research domain there are lot more options like TRACER, SEGA, LocusRoute, VPR/T-VPack etc. [4, 5, 6, 7].

For XBits, we have used to VPR/T-VPack suite as the tool for the Placement and Routing phase. VPR takes in .blif file as the input file and generates .p and .r files. The .p contains the placement information for the blocks in the design and the .r file contains the routing information for the interconnections. The T-VPack tool converts the .blif file into a .net file which represents the packed net-list for the circuit. For more information on these file formats, please refer to the [8].

Once the circuit has been packed, placed and routed, its time to generate the bitstream which will be used to configure the FPGA. This bitstream is a stream of 0s and 1s which is sent to the chip in a specific order. The 0s and 1s get stored in the SRAM cells which are scattered uniformly over the chip area. These SRAM cells are connected to the configurable elements like LUT entry bits, Switch Block switches,
MUX Selects Bits etc. Thus these bit values that get stored in the SRAM cells are used to program the configurable elements of the chip. This is the main focus of the XBits framework. Let’s take a look at a more detailed overview of this phase and also see how it fits with the others stages in the design flow.

2.2. **Overview of Bitstream Generation for MT-FPGA**

As described in the previous section, XBits uses the outputs from the VPR/T-VPack as inputs for generating the bitstream for configuring the MT-FPGA. Figure 2-2 shows the logical view of various inputs and outputs of the XBits framework.

![Logical Bitstream Generation Flow](image)

**Figure 2-2 Logical Bitstream Generation Flow**

As you can see, Figure 2-2 presents the inputs grouped by information type. The .blif file contains the LUT bits obtained from the technologically mapped files. This file is usually generated from the Mapping phase in the design flow. The .p and .r file contain the placement and routing information (as discussed in the previous section).
The positional details of the SRAM cells are obtained from an Architecture Description File (.xml format [9]). The format of the Architecture Description File is specific to XBits and has been designed as part of the framework. For more information on this format please refer to 3.7. Each of the files listed above are parsed by separate internal parsers. This parsed information is then stored in appropriate data structures inside XBits, where it is combined and processed to generate the bitstream to configure the MT-FPGA. We will be describing this process in more detail in the forthcoming chapters.

2.3. **MT-FPGA Overview**

A basic understanding of the architecture of the MT-FPGA chip (the target chip for XBits) is vital to understanding the process of bitstream generation for it. So this section presents an overview of the chip from a structural point of view. For more detailed description of the MT-FPGA, readers are suggested to refer to [39]. The MT-FPGA is a mixed-signal field programmable device that incorporates traditional FPGA technology. In addition to the usual digital components like LUT, MUX etc., it may also have non digital sensor based nodes. The digital components of MT-FPGA are programmable in nature and are similar to typical SRAM cell based FPGA [10] in their setup procedures. Configuring a specific circuit description onto the MT-FPGA is done by downloading a Bit stream where each bit is used to program a certain SRAM cell to either a ‘0’ or ‘1’. Any design implementation on the MT-FPGA thus needs to make a stream of bits, which maps the design to the programmable fabric. The current version of the MT-FPGA chip has 8 programmable MTLC (Multi Technology Logic Cluster), which are also referred to as CLB (Configurable Logic
Each of these MTLCs has 4 PLBs (Programmable Logic Block) which in turn contain a 4-LUT and D Flip-Flop in them. Interconnection among the 8 MTLC, is achieved by employing interconnection buses, Connection Blocks (CB) and Switch Blocks (SB). XBits and the internal class hierarchy closely resemble the chip structure of this MT-FPGA, and so the bit stream generation process is tailored toward configuring this particular chip. But this class hierarchy can be easily modified as per our needs to handle other kinds of FPGA chip architectures. A hierarchical view of the structure of the MT-FPGA chip is depicted in Figure 2-3.

![Hierarchial View of the MTLC](image)

**Figure 2-3 Hierarchial View of the MTLC (Adapted from [39])**

### 2.4. Related Work

A number of different approaches have been followed for generation of bitstreams for configuration of FPGA chips. Xilinx has developed JBits [11] for generating and manipulating bitstreams for their Xilinx Virtex series of chips. JBits is a proprietary tool which provides support for bitstream generation through a set of Java APIs. Another approach (JPG) which builds on top of JBits tool and adds in the capability of partial reconfiguration of the bitstream is described in [12]. JPG allows a user to
selectively configure a part of the whole bitstream generated by JBits by making use of information derived from other tools in the standard design flow. A novel FPGA configuration bitstream generation has been discussed by K.Siozios et al. in their paper “A Novel FPGA Configuration Bitstream Generation Algorithm and Tool Development” [13]. In this paper, they describe a bit stream generation tool for generating bitstreams for a custom FPGA platform. This custom FPGA platform has been developed under the AMDREL project [14]. A lot of the work done under this project has involved the design flow chain for FPGAs. DAGGER, their bitstream generation tool is a part of this chain. XBits matches DAGGER in terms of functionality and intent. Both tools were developed to support a particular target FPGA chip but can act as standalone tools given standard inputs and compatible target FPGA architecture. The approach followed in DAGGER is different though from the one we have used. DAGGER treats the SRAM cell grid as a collection of patterned grid sub-blocks. The sub-blocks are further grouped based on the SRAM cell pattern they contain and classified as top, bottom, left, right IO Block and internal CLB (CLB and Switch Block combined) Block. As you can infer, this is a more of a bottom up approach to bit stream generation, whereas XBits follows a top-down hierarchical method where an FPGA object (representing an entity on chip) defines and controls its sub objects. Also, representing the SRAM cell grid array are patterned sub-blocks may not be suitable or even feasible if the chip structure is irregular. So in view of maintaining more flexibility and providing support for irregular chip structures, we decided to not to adopt the approach followed in [13].
Another area where a lot of work has been focused is that of reducing bitstream configuration overhead. This helps in reducing the bitstream download time to chips thus reducing reconfiguration delays. The three major techniques for doing this are configuration pre-fetching [15], bitstream caching [16] and compression of bitstreams for FPGA chips [17]. [17] presents a general overview of the compression of bitstreams for FPGA using various standard compressions algorithms like Lempel-Ziv [18], Huffman [19] and Arithmetic Coding [20]. It describes the limitations of using these compression algorithms directly without regard to features of the bitstream. It proposes methods of performing regularity analysis and choosing optimal symbol length to increase the compression ratios of these algorithms specifically for FPGA bitstreams. It also presents a nice comparative of the performance of these different algorithms on selected benchmarks. Although these techniques have been demonstrated in context of the Xilinx Vertex series chips, they can be adapted to any generic FPGA architecture. An approach to improving the efficiency of compression by don’t care discovery has been proposed in a previous paper by the same authors [21]. In this, they attempt to increase the compression ratios of one of their own algorithms by finding out don’t-care bits in the bitstream and using this information in their modified compression algorithm. Other algorithms can likewise benefit from using don’t-care bit information.

A lot of the foregoing techniques and methodologies can be very easily integrated into the XBits framework. The Object Oriented and Component Based architecture of the framework should make it a lot easier to do so. These can prove to be very useful improvements that may be implemented as future work.
3. Design

In this chapter, we present the various design aspects associated with the development of XBits as generic framework for constructing bit stream generators for varying architectures and tool flow chains. There were different approaches that could have been adopted to making of a software tool for bit stream generation for configuration of a specific Multi–Technology FPGA chip. We will consider these approaches that lead up to the current design and then discuss in detail the framework components and the interactions between them. We will also summarize the benefits we were able to leverage from the approach we followed.

The architecture of the XBits framework is based on a collection of core framework components and the interaction between these components. The XBits tool did not originate as a framework based architecture. This has slowly evolved over the course of development and various design iterations. Some of decisions were taken for specific scenarios and some were made to augment the functionality of XBits as a whole.

There were a number of key factors and criteria that were to be considered in the design of such kind of tool, catering to a specific chip. The first and foremost, is the nature and the area of application of this tool. Since this project originated as a support infrastructure development for the MT-FPGA chip (2.3), the initial design plans suggested development streamlined and tailored toward the chip architecture. Although this design would have led to a product which could generate the required outputs for this particular chip in the the most efficient way, but it would have been
limited to one prototype device. For instance, any change in the chip architecture or the method of configuration would require a change to the code, which in the worst case, may have a global cascading effect. Also hard-coding the parametric information about the chip would restrict the user from experimenting with the bit generation process. This kind of application specific development has its benefits in terms of the development effort required and the amount of complexity to be dealt with. But during the design phase it was argued that placing such a restriction would seriously impede the user’s ability to apply this tool and that development effort required to add in generic functionally was considered to be worth it.

The second important reason that affirmed the above argument was that the architecture of the MT-FPGA chip was in a process of evolvement and refinements. A lot of structural improvements were made to the chip from its initial design and newer improvements were being suggested. As a result of this, having a specific tool bound to only one type of architecture would have led to a lot of rework, each time a modification was made to the chip. To account for this changing architecture, the need was felt to develop a backend infrastructure which would allow the front end of the tool to be customized for each new chip with relative ease and less overhauling of the whole tool. This meant that the tool had to capture and handle the parametric as well as structural complexities of the chip architecture in a generic way which would be extensible, flexible and customizable.

On account of above, a framework-component based approach for XBits was considered as a better alternative and development was steered towards building up a generic XBits Framework for Bitstream Generation. Some basic objectives and
design guidelines were defined for this framework. These served the purpose of keeping the project scope and development streamlined, rather than acting as strict rules. Here’s a list of these basic objectives and design guidelines that we followed:

1. XBits, although a complete tool in itself, would also be used as a framework for building other tools on top of it or in design flow with it.
2. The architecture of the XBits framework would be component based.
3. These components would isolate as much of the internal implementation from other components as possible.
4. The top level functionality of the tools would be based on interactions between these components. These interactions would be defined as clearly as possible and kept simple.
5. The component architecture would support plugging in of newer components in place of older ones with minimal effort required.
6. The components internally would be designed using Object Oriented Programming Techniques to leverage code reuse and extensibility using Java.
7. Where possible, preexisting code base and modules would be integrated and used (for example JBits Interface 1.0 [32])
8. The output and input formats would be chosen to be standard and industry/academia accepted and would feature easy extensibility and concise descriptions.

Adhering to the above guidelines, kept the overall architecture of XBits framework together and helped in making it more modular. Also separation of the whole framework into distinct components has reduced, to a large extent, the complexity
involved in the process of bit stream generation from higher level circuit descriptions. Since there were more than one programmers working on the development framework simultaneously, the modularity achieved through the component based framework allowed individuals to focus on the complexity of modules assigned to them thus easing division of work. All of this resulted in a better software development process which lead to a scalable and customizable design tool.

This basic architecture overview which comes directly out of the above guidelines is represented in Figure 3-1. As you can see, the XBits framework can be seen as a collaboration of distinct framework components. This diagram shows a deliberate emphasis on displaying the component interaction after the circuit description data has been generated corresponding to emphasis in my individual thesis. For a more detailed discussion of the circuit description generation components, refer to [41].
Figure 3-1 XBits Framework Components Overview

The following major components are illustrated in the Framework Diagram:

1. Circuit Description data generation component
2. Circuit Object Hierarchy
3. Architecture Description File Parser
4. Architecture Hierarchy Traverser
5. Bit Stream Generator
In the following paragraphs, we will take a look at the functionality associated with each of these components and describe the various types of interactions (shown by block arrows) between them. In this chapter we will consider the foregoing more from a design point of view. In Chapter 4, we will revisit this design and show the implementation aspects associated with it.

### 3.1. Circuit Description Generation

We will briefly touch upon the Circuit Description Generation process to start the description of components. Circuit Description Generator is responsible for converting the users circuit description which is represented by higher level input files like .blif, .r, .p and .net (2.2) into an internal memory representation inside XBits. This framework component consists of various parsers like BLIF parser, Routing parser, Placement parser and Net Parser which read in the corresponding input files, instantiate the Circuit Object Hierarchy data structures and store their information in these data structures at appropriate places. We will see that this “place” is actually fields inside various objects of Circuit Object Hierarchy. The interaction between this component and the Circuit Object Hierarchy is in form of transfer of SRAM bit “value” data generated from the circuit description. The SRAM bit locations are filled in at a later stage.

### 3.2. Circuit Object Hierarchy

The Circuit Object Hierarchy forms the central data model for the circuit description and architecture description information. It also plays a very important role in generation of bits needed to configure the chip. The Circuit Object Hierarchy from a design point of view consists of a representation of the whole chip in a hierarchical,
tree-based memory data structure. This data structure is comprised of nested instance objects of various classes having a one to one correspondence with the hierarchy of the actual physical chip architecture. The chip component objects are generally aggregated from other chip sub component objects. For example, a CLB (Configurable Logic Block) object will be aggregated from a collection of 4 BLE (Basic Logic Element, same as PLB) objects, 1 CB (Connection Block) object) etc. The leaf objects of the Circuit Object Hierarchy are always instances of the SRAM Bit class storing the generated bits. The circuit description data that is aggregated in this tree structure maintains both the location of the configurable bits as well as their values. These two different aspects of the bits may be filled in at different times. As mentioned before the bit value data is primarily filled in by the Circuit Description Generator. The other aspect, the location of these bits is transferred to the SRAM bit objects of the Circuit Object Hierarchy through its interaction with the Architecture Hierarchy Traverser component (shown by the two way arrow between them (Figure 3-1)). This is a two way interaction, the Circuit Object Hierarchy exchanges the FPGA context information in return for the bit location information. Since this interaction involves the Architecture Hierarchy Traverser component heavily, we will defer talking about it until after the paragraph on the Architecture Hierarchy Traverser.

### 3.3. Architecture Hierarchy Traverser

The Architecture Hierarchy Traverser is the framework component which is responsible for transferring the SRAM bit location data from the architecture description to the Circuit Object Hierarchy. Although the name Architecture
Hierarchy Traverser suggests that it is performing the parse of the Architecture Description File (which is an instance of an XML document [26]), this task is actually performed by the Architecture Description File Parser that parses the physical file to an XML tree model in memory. This model of the XML document in memory is parsed and traversed by the Architecture Hierarchy Traverser to perform its functions, hence the name Architecture Hierarchy Traverser. The Architecture Hierarchy Traverser is responsible for interacting with the Circuit Object Hierarchy and pulling and pushing information from it. Thus in the process of bit generation, the Architecture Hierarchy Traverser acts as the active components whereas the Circuit Object Hierarchy acts as the passive one. The bulk of the computation is performed inside the methods of the Architecture Hierarchy Traverser where the XML data model information is sent over to the Circuit Object Hierarchy and corresponding FPGA object references are returned back. One of the main goals behind keeping the XML tree traversal, SRAM bit offset computation and XML node referencing code in the Architecture Hierarchy Traverser component (outside of the Circuit Object Hierarchy) was to keep the Circuit Object Hierarchy interface as simple as possible. So that any legacy code base and data model for circuit description could be substituted in place of the current Circuit Object Hierarchy with minimal modifications to its code. Also adding in newer chip components to the existing Circuit Object Hierarchy is much simpler if the chip component object added does not have to worry about handling tree traversal and other things.
3.4. *Architecture Hierarchy Traverser ↔ Circuit Object Hierarchy Interface*

Now that we have an overview of the Circuit Object Hierarchy and the Architecture Hierarchy Traverser, let’s discuss the inter-component interactions between these two. This interaction maintains a layer of separation between these components and makes them completely independent of each other. Since this interaction is performed through out the traversal of the XML tree, the overall computation time of the process can be heavily affected by it. Therefore, the design of this interaction was done so that the interaction itself requires as little computation as possible. A thin interface was used to achieve this goal. This thin interface simply consists of a single node from of the XML tree containing the Architecture Description Information on Architecture Hierarchy Traverser side and an object reference describing the FPGA circuit context information on the Circuit Object Hierarchy side. In Figure 3-1, this has been represented by the two way arrow connecting these components. The interface itself is shown as a `processComponent()` block. This is the only interface method that was designed to be a part of the thin interface. This simple interface, with only one method and two encapsulated parameters ensures a clear separation between the two components and provides a lot of flexibility in terms of changing or even completely substituting the internal implementations of the components. We will be looking in detail at the mechanism which leads to this flexibility as well as examine the `processComponent()` interface method in section 4.2.5. Bit generation for the configuration of the FPGA is completed as a result of the foregoing series of interactions. Once the location information has been transferred to the appropriate FPGA objects in the object hierarchy, we have sufficient data to configure the chip.
This data comprised of the value and location of the SRAM bits is used to generate the bit stream by the Bit Stream Generator Component.

3.5. **Bit Stream Generator**

The purpose of the Bit Stream Generator is to convert the value and location information of the SRAM bits to a stream of bits which can be sent directly to the chip through appropriate means to configure the chip according to the circuit description. For the sake of simplicity this component does not cater to the actual process of configuring the chip, as this can be conveniently handled by a simulation test bench for VHDL simulation or as part of a tool like LabView® used to configure a chip mounted on a test fixture. The input data provided to the Bit Stream Generator is in form of collection of the SRAM Bit objects. It serializes the information in these objects to a stream of 0s and 1s based on a configurable streaming policy. A number of different streaming policies can be supported, like row major, column major etc. Also a Default Bit Policy can also be selected to decide on the values to be filled into unspecified bits on the chip.

3.6. **Advantages of the Framework approach**

Now that we have looked into the design of the XBits framework, we are ready to identify and study as to why we followed the approach of creating a framework oriented bit stream generation system rather than a task specific tool. We will discuss briefly, in subsequent paragraphs, the major advantages to our approach:

1. Extensibility
2. Compatibility with Legacy
3. Flexibility

I assume the biggest motivating factor behind taking a framework oriented approach was that of the immense benefits gained in terms of extensibility. Extensibility is achieved in multiple domains. By not restricting our tool to handle the architecture of a specific chip, we have ensured that it can be used for different kinds of prototype devices. These devices can vary in different aspects like scale, structure and component implementation. On account of the loose coupling between the framework components of the XBits framework, a user can easily incorporate device descriptions which may have a much larger grid of logic blocks. The utility features like referencing and ranging provided in the XML Architecture Description Format ensure that the architecture description of these huge circuits can be handled in a memory efficient manner. Changes in the structure of chip components can also be handled elegantly by sub-classing the corresponding objects in the Circuit Object Hierarchy and overriding the `processComponent()` interface to handle the modifications. Last but the not the least, the XBits framework can also be extended by incorporating a completely new component which does not pre-exist in the Circuit Object Hierarchy. This capability may allow XBits to be used for bit stream generation of newer chips containing digital, analog and mixed domain components. As long as the behavior of these components and there reconfigurable parts can be captured in a circuit description object model, integrating these components into XBits framework should be relatively easy. Again, the extensible Architecture Description File Format would help a lot in process of capturing the architecture information of this new component.
A large domain of tools in today’s FPGA tool chain, are huge software systems that contain large bodies of legacy code. These systems may cover multiple layers in the tool chain or be specific to handling only certain functionality. In any case, these systems have their own internal representation of the circuit description data and the chip architecture. This internal representation may be constructed using as varied data structures as arrays, lists, hash tables, trees or even specific object graphs. XBits framework provides us maximum convenience in integrating the data model of XBits with these kinds of legacy systems by minimizing the constraints on the interface and having a clear separation between the interface and the implementation (as seen before). Since there are minimal amount of things that need to be taken care for interfacing different framework components (or layers), we found it relatively easy to incorporate legacy code. A good example of this is built into the XBits framework already. As mentioned before, we have adopted the JBits Interface 1.0 and augmented it to integrate into our framework. JBits Interface software component consists of various modules of legacy code for building and representing circuit description model. By using the `processComponent()` interface and base classing the circuit description model components from `FPGAEntity` abstract class, we were able easily able to reuse a large amount of functionality provided by JBits Interface. The same methodology may be applied to other legacy code base systems, thus expanding the ways and areas XBits framework can be applied to in future.

Besides being customizable with external components, XBits framework has an internal architecture that is highly flexible too. The various framework components have been implemented using Object Oriented Programming Techniques to achieve
maximum encapsulation and code reuse. All the classes in these components can be derived from to modify selected behavior. For instance, methods in Architecture Hierarchy Traverser can be very easily overridden to provide a different mechanism for referencing nodes. So in effect internal software units such as the DOM tree traversal modes, search algorithms, class handling of XML nodes are all extremely customizable. Also the use of XML format for architecture description allows for great amount of flexibility to the way different chip structures can be represented and handled by the XBits framework without needing any modifications.

3.7. Architecture Description File Format

The Architecture Description File Format used by XBits is based on the XML (eXtensible Markup Language) document format. XML is a language for representing hierarchical information in a text based format [22]. XML is a W3C standard [23] and widely accepted for information exchange over the internet. It is completely platform independent. For more information on the XML format, its features and applications please refer to [24]. The inherent hierarchical nature of the structure of the MT-FPGA (or any other standard FPGA chip for that matter) makes it convenient to represent succinctly and accurately in an XML based document. This was one of the main motivations behind selecting the XML format. The Architecture Description File Format in the current version of XBits mainly specifies the SRAM bit locations for configurable bits contained in various chip components. It also provides the information about the bit patterns corresponding to the various lines in the MUXs used in the chip. The format allows the file to specify optional user defined attributes to chip components which may contain any other architectural parameters. The
method for specifying the chip components is pretty straightforward. They are specified as nested block element tags (<B> ... </B>). This nesting exactly matches the physical composition on chip. The general format for the block element tag is as follows:

```xml
<B CLASS="Component Class" X="X Offset" Y="Y offset">
  [Other optional attributes]
</B>
```

Figure 3-2 Architecture Description File Format

Each block element tag represents a physical entity on the chip. These tags can be nested to as many levels needed to specify the SRAM bits for all chip components. The CLASS attribute of the element tag (CLASS="Component Class") represents the type of the chip component represented by the element tag. Common component classes are CHIP (top level element), LB (Configurable Logic Block), LE (Logic Element, equivalent to PLB), MUX, LUT, CB (Connection Block), SB (Switch Block) etc. The SRAM bit is represented by a block element tag with no CLASS attribute specified as shown above in Figure 2-2. As you can see, each block element tag has X and Y attributes (X="X Offset" Y="Y offset"). These specify the top level SRAM bit location offsets for the chip components corresponding to the element tag. This is used in the offsetting mechanism to generate the SRAM bit locations. We will be describing this in more detail in section 4.3.2. Beside the class and the bit offsets, you can also
include any user defined optional attributes. The interpretation of these attributes is left to the \texttt{processComponent()} method of the corresponding object in the Circuit Object Hierarchy to which the element tag is passed after been parsed. This enables us to provide specific or auxiliary information about chip components to the Circuit Object Hierarchy making the format more flexible and extensible. Some of the common optional attributes used are \texttt{ID}, \texttt{ROW}, and \texttt{COLUMN} etc. Generally, these optional attributes are used to convey logical information about the block elements. Here’s an example extracted from the Architecture Description File for the MT-FPGA:

```
<B CLASS="CHIP" ROW="3" COL="3" X="0" Y="0" BIT_ARRAY_ROWS="61" BIT_ARRAY_COLS="99">
  <B CLASS="LB" ROW="3" COL="1" X="7" Y="8" NAME="DEF_MTLC">
    <B CLASS="LE" ID="0">
      <B CLASS="LUT">
        <B ID="0" X="15" Y="0"/>
        <B ID="1" X="15" Y="1"/>
        <B ID="2" X="15" Y="2"/>
        <B ID="3" X="15" Y="3"/>
        <B ID="4" X="15" Y="4"/>
        <B ID="5" X="15" Y="5"/>
        <B ID="6" X="15" Y="6"/>
        <B ID="7" X="15" Y="7"/>
        <B ID="8" X="8" Y="0"/>
        <B ID="9" X="8" Y="1"/>
        <B ID="10" X="8" Y="2"/>
        <B ID="11" X="8" Y="3"/>
        <B ID="12" X="8" Y="4"/>
        <B ID="13" X="8" Y="5"/>
        <B ID="14" X="8" Y="6"/>
        <B ID="15" X="8" Y="7"/>
    </B>
  </B>
  <B CLASS="MUX" ID="0" LINES="8">
    <L SEL="000" TRK="4"/>  <!-- RC1-->
    <L SEL="001" TRK="00"/>  <!-- RC2-->
    <L SEL="010" TRK="00"/>  <!-- D0-->
    <L SEL="011" TRK="11"/>  <!-- A1-->
    <L SEL="100" TRK="LUT1"/>
    <L SEL="101" TRK="LUT2"/>
    <L SEL="110" TRK="LUT3"/>
    <L SEL="111" TRK="LUT4"/>
  </B>
</B>
```

Figure 3-3 MT-FPGA Architecture Description File
In Figure 3-3, block element tag for the \texttt{MUX} class contains line element tags besides other block element tags. These line element tags (\texttt{<L> ... </L>}) are used to specify the bit pattern needed in the Select-Lines of the MUX to select a particular input line. The combination of nested block element tags and line element tags are used to completely specify the SRAM bit information for the whole chip.

\section*{3.8. Summary}

In this chapter we looked at how and why we made some design decisions during the development of the XBits framework. We saw a top level view of the framework and its major components. We also looked at the Architecture Description File Format which we will be using extensively in the next chapters.
4. Implementation

The XBits framework is divided into the following components: Circuit Description data generation component, Circuit Object Hierarchy, Architecture Description File Parser, Architecture Hierarchy Traverser, and Bitstream Generator. In Chapter 3 we looked at these framework components from a design point of view. In this chapter we will look at their implementation details. Our main focus will be the Architecture Hierarchy Traverser and the Circuit Object Hierarchy.

4.1. Generation of Bits

Before describing the implementation of the process of generation of bitstreams, we need to define what generation of bits (or bit stream) exactly signifies in context of XBits. This will be used throughout the discussions in further paragraphs.

MT-FPGA chip uses a two dimensional SRAM cell grid array for configuring its chip components (2.3). These SRAM cells can store a binary value of 0 or 1, which can have different meaning based on where and how the cell is used. These SRAM cell objects may belong to other higher-level FPGA objects which are part of the Circuit Object Hierarchy (3.2). For instance, a LUT object contains 16 SRAM cell objects, which denote the 16 different values corresponding to each of the 16 LUT entries. Based on these two facts, the SRAM cells have been represented internally using a simple object (SwitchEntity) with three field variables: the X and Y fields (denoting the location in the SRAM array) and the value field, storing a 0 or 1 value in the bit. Thus an SRAM cell object can be thought of as having two different aspects associated with it, positional \((x, y)\) and the value aspect.
An SRAM cell object (or just bit) can be said as generated (completely) only when both these aspects have been appropriately filled in. But this process may span several framework components or layers, where one component would generate one aspect and other(s) another. So it is very difficult to demarcate as to where exactly the SRAM cell bits are generated. To make it easier, we will be referring to the process of generation of any one of the aspects of the switch bit objects, simply as “generation of bits”. The particular aspect being generated or filled in should become clear from the context of the discussion. Otherwise, the generated aspect will be stated explicitly.

The important thing to keep in mind is that generation of values for any of the field variables of the switch object is a progressive step towards “generation of the bit”. So, generation of the complete bit object can be thought of as analogous to the generation of an aspect of the bit.

### 4.2. XBits Framework Implementation

#### 4.2.1. Execution Flow

The entry point class of the XBits framework is the `XBitsCoordinator` class. This class is responsible for coordinating the execution order of the various components of the XBits framework and managing their inputs and outputs (Figure 2-2). It accepts the names of BLIF file (.blif), placement (.p), routing (.r), net-list (.net) and Architecture Description files to be used for bitstream generation from the command line. Refer to section 2.1 for a description of these files. The execution flow continues from this point as following:

1. Information from the BLIF file is parsed and stored in internal data structures.

   This is done by the `Blif_Parser` class.
2. Information from the net-list is parsed using the Net Parser (Net_Parser).

3. Next, the placement file is parsed and the block placement info is read in (Placement_Parser).

4. After this routing information about the interconnections is read and parsed into internal data structures (Routing_Parser).

5. At this point all the circuit description information needed has been parsed into data structures in the XBits framework or in other words the whole Circuit Object Hierarchy (3.2) corresponding to circuit described in the input files has been allocated and a majority of the SRAM cell bit values have been filled in. Note the Circuit Object Hierarchy still does not contain SRAM cell bit location information. This is stored in the Architecture Description File. So the next step is to parse it. This task is performed by XMLParser class.

6. XMLParser uses javax.xml.parsers.DocumentBuilder (available as part of JAXP library [25]) class to parse the physical Architecture Description File into a DOM tree (4.2.2) in memory.

7. Once the tree has been built, the root XML Node and root Circuit Object Hierarchy node is passed to XMLParser.genBits() method. This method ensures that the bit locations and values are filled for all the SRAM cell objects in the Circuit Object Hierarchy. We will look at the internals of this method in detail in section 4.2.3.

8. The BitStreamArray class then generates and outputs the bit values from the SRAM cell objects to a bitstream on standard output in form of 0s and 1s.
4.2.2. **DOM (Document Object Model)**

The first step performed by the xmlParser class is to parse the Architecture Description File located on a physical disk medium. This work of parsing in the XML file into a memory data structure was delegated to standard Java Libraries. This was one of the motivating factors behind the use of universal XML format. Many different open source libraries are available for conversion of XML to memory data structures. Some examples are SAX (Sequential API for XML) [26], JAXB (Java API for XML Binding) [28] etc. For our purpose, we chose to use DOM (Document Object Model) [27] which is part of the JAXP (Java API for XML Processing) [25]. DOM parses the XML document into a tree-based model of the XML document consisting of DOM tree nodes. This tree represents the exact structure of the XML file in a memory data structure with attributes and sub elements available using class methods. For convenience we will be referring to both element blocks in the XML file on disk and DOM tree nodes in memory as simply XML nodes since they are just different representations of the same data.

This generic node based tree allows for easy traversal and manipulation of its nodes. Having a generic node object representing an element in the XML format eliminates the need to handle each type of element tag representing a different part of the chip differently. For referencing, offsetting (4.3) etc. the same code can be used regardless of the type of node. Also once the tree model is created; traversing of a XML DOM tree is much faster than SAX and can proceed in any arbitrary direction. The only major drawback of the DOM is it usage of memory. DOM uses considerably larger memory than SAX and a little more on an average than JAXB. Since the use of
referencing and offsetting features of the Architecture Hierarchy Traverser ensures that most of the XML nodes are virtual and do not need to exist in memory, the memory usage factor becomes less significant. This is especially true if the chip structure is regular.

4.2.3. genBits()

genBits() is a method of the XMLParser class (the main class of the Architecture Hierarchy Traverser component). The genBits() method contains the core algorithm which co-ordinates between the Circuit Object Hierarchy and the XML DOM tree to generate the bit offsets for the SRAM cells. It is also responsible for computing and maintaining the absolute bit offsets from relative ones for the current node in the tree traversal and passing on this information along with the node itself to appropriate context in the Circuit Object Hierarchy. Before we delve more into genBits(), lets touch upon some parameters that we will come across. Here’s the signature of the genBits() method:

```
    genBits(FPGAContext, n, bitRowOffset, bitColOffset)
```

As you can see, the following information is passed (at every step in recursion):

1. FPGA Context (FPGAContext) – Current object in the FPGA Object Hierachy
2. XML Node (n) – Current XML Node in the DOM tree
3. Bit Row Offset (bitRowOffset) – Relative Offset for Bit Row Location
4. Bit Column Offset (bitColumnOffset) – Relative Offset for Bit Column Location

During the traversal of the XML tree, genBits() has to maintain coordination between the XML nodes and the current object in the Circuit Object Hierarchy. FPGAContext
defines the current object for which the bit locations are being generated. It defines
the object in the Circuit Object Hierarchy corresponding to a particular node in DOM
tree traversal. It can represent anything from the whole MT-FPGA chip, an IO Block,
a Switch Block to a simple SRAM Switch. FPGAContext just supports a common
interface that allows the Architecture Hierarchy Traverser to look at any FPGA object
in the same way irrespective of its type. The second parameter to the genBits() routine
is the current node (specified here as N). This is the current node in the XML DOM
tree. The last two parameters specify the current bit location offsets (row and
column). These location offsets specify the relative origin in the SRAM cell grid for
all the child objects of the FPGAContext. This is used to leverage relative offsetting as
described in section 4.3.2.

Figure 4-1 shows the basic flow chart diagram for the genBits() method. As you can
see from the diagram the algorithm is top down recursive. The dotted line represents
the recursive calls to genBits(). On entry, the first thing that is checked for is the tree
traversal terminal condition. It is important to analyze the significance of this terminal
condition and how it affects the tree traversal. The assertion for the terminal condition
is: either FPGAContext is null or XML node N is not valid. In anyone of these cases we
gracefully exit and stop traversing the current branch of the tree. The assert condition
can evaluate to true if FPGAContext, returned previously from a higher level of
recursion, is null. For instance, a particular object corresponding to an SRAM cell
may return a null to notify it has no more sub objects that need to be processed.

FPGAContext can also be null, if it is referring to an object which was never allocated
during the Circuit Description Generation. This way memory can be selectively
allocated for only those chip components that are actually to be used. Another way the tree traversal can complete is if the node of the DOM tree of the architecture description file evaluates to a null. Although this condition is never reached in the current Architecture Hierarchy Traverser, but this check is included so that it may be possible to selectively choose which components to generate bits for, by setting other DOM tree nodes to null.

If the terminal condition evaluates to a false, we process the context information passed to us as parameters and proceed with generating bits for all the sub components of the current FPGAContext. The DOM tree node (which is an object of class org.w3c.dom.Node [29]) provides access to various attributes like logical row & columns, bit location offsets, class etc through class methods. Before the bits are generated, utility features like ranging and offsetting (4.3) are applied to the bit location offset of this node. The attributes values may change due to the application of these utility features. And these attributes need to be saved before applying the utility features. We will see the reason for doing so below.
Is FPGAContext Null?

Yes

Save the contents of the node before manipulating it for ranging, offsets etc.

Calculate new offsets, range limits and perform the following steps repeatedly for the range specified

 Exit genBits() 

No

Is reference to another node?

Yes

Find the XML node which is referenced and use it as the new current node

No

Compute all the child nodes of the current node and perform processComponent() on each of them returning new FPGAContext

Call genBits() recursively for all new child nodes, using the new FPGAContext and bit offset values

Restore the contents of the node

Figure 4-1 genBits() Flowchart
Once the utility features have been applied to the attributes of the XML node, the new bit location offsets (bitRowOffset, bitColOffset) are computed from the \(X\) and \(Y\) attributes of the current XML nodes (4.2.4, 4.3.2). Next, the XML node is passed to the `processComponent()` method of the current object from the Circuit Object Hierarchy pointed to by `FPGAContext`. This method examines the contents and attributes of the XML node passed to it and based on it returns the corresponding object to the XML node from the Circuit Object Hierarchy. We will be discussing the `processComponent()` interface method in detail in section 4.2.5. The returned object is set as the new `FPGAContext`. After this the child node list `childNodes[]` of the current XML node in the DOM tree is computed. The new `FPGAContext`, child XML node (`childNodes[i]`) and the new bit location offsets are used as parameters to the next lower level call to `genBits()` for each child node (as shown by the dotted line in the Figure 4-1). In this way, both the XML DOM tree and the FPGA class hierarchy are traversed in co-ordination. The SRAM bit locations are actually generated at the leaf nodes (SRAM cells) of this tree traversal (which are represented by the `SwitchEnity` object in the Circuit Object Hierarchy and node with no `CLASS` attribute in the XML DOM tree (4.2.4)). It is important to restore the saved attributes of the current node, after the recursive call returns. This is done so that any further node, referencing this node, gets the correct (original) attributes and not the changed (possibly offset) ones.

**4.2.4. Example Run-Through for `genBits()`**

To illustrate the process of bit generation more clearly, let’s go through an example tree traversal. Describing the complete traversal of the Circuit Object Hierarchy and DOM tree for the whole MT-FPGA might not be feasible here in terms of space and
scope. So to make things simple we will try to show the tree traversal for a part of the hierarchy under a CLB (or MTLC) node. Also we will be ignoring the utility features and attribute saving/restoring for the same sake. Figure 4-2 shows the class composition diagram for the part of the Circuit Object Hierarchy under a single CLB (or MTLC).

Figure 4-2 Example Run-Through DOM Tree

As you can see the partial Circuit Object Hierarchy diagram exactly matches the actual CLB (MTLC) in the MT-FPGA chip structure. A CLB object consists of 4 BLE

S = SwitchEntity (object representing SRAM cell)
(only two are shown here). A **BLE** in turn has 1 **Lut**, 4 input **MUXs** and 1 output **MUX**. The **Lut** is composed of 16 **SwitchEntity** objects (SRAM cells). Each of input **MUXs** are of 8-to-1 type and therefore contain 3 **SwitchEntity** bits for selecting inputs. The output **MUX** is used as a 1-to-2 de**MUX**. Even though the input and output **MUXs** are of different type, in the Circuit Object Hierarchy they are both instances of the **MUX** class (as shown in the Figure 4-2). The part of the Circuit Object Hierarchy that we will be traversing in this example run-through has been enclosed within a dashed line region in Figure 4-2. We will specifically be showing how starting from the **CLB** object the bit offsets are filled into the “shaded” **SwitchEntity** object.

Figure 4-3 shows the fragment of the XML Architecture Description File corresponding to the above Circuit Object Hierarchy. The XML nodes corresponding to the part of tree enclosed in the dotted line area in Figure 4-2, have been bold faced. As described in section 3.7, the **CLASS** attribute defines the type of XML node and its omission denotes that the node is representing an SRAM cell. The `<L>`...`</L>` blocks under the **MUX** node do not correspond directly to any object in the Circuit Object Hierarchy. They only specify the bit pattern corresponding to each line in the MUX. So they will not be considered in this run-through. (Note: **CLASS** attributes for **CLB** and **BLE** object have been abbreviated as “**LB**” and “**LE**” respectively)

```
<B CLASS="LB" X="7" Y="8" ROW="3" COL="1" NAME="DEF_MTLC">
  <B CLASS="LE" ID="0">
    <B CLASS="LUT">
      <B ID="0" X="15" Y="0"/>
      <B ID="1" X="15" Y="1"/>
      <B ID="2" X="15" Y="2"/>
      <B ID="3" X="15" Y="3"/>
      <B ID="4" X="15" Y="4"/>
      <B ID="5" X="15" Y="5"/>
      <B ID="6" X="15" Y="6"/>
      <B ID="7" X="15" Y="7"/>
      <B ID="8" X="8" Y="0"/>
      <B ID="9" X="8" Y="1"/>
    </B CLASS="LUT">
  </B CLASS="LE" ID="0">
</B CLASS="LB" X="7" Y="8" ROW="3" COL="1" NAME="DEF_MTLC">
```
Figure 4-3 Example Run-Through Architecture Description File
Let us assume the tree traversal (genBits() method) is initially in a state where the current object in the Circuit Object Hierarchy (FPGAContext) is the top level CLB and the current XML node (N) been parsed is the XML node with CLASS="LE" (a child of <B CLASS="LB" ...> (Figure 4-3)). The bit location offsets (bitColumnOffset, bitRowOffset) are calculated incrementally top down by adding their values from the previous recursive level to the current x and y attributes. These offsets are added to the x and y attributes of the current XML Node before passing it to processComponent(). Note in this case the offsets are the same as attribute values as the previous recursive level location offsets are 0, 0.

So we have:

- FPGAContext → CLB (parent of BLE class)
- N → BLE XML Node (<B CLASS="LE" ID="0" ...
- bitColumnOffset → 7 (<B CLASS="LB" X="7" Y="8"...)
- bitRowOffset → 8 (<B CLASS="LB" X="7" Y="8"...)

First the new FPGAContext is computed by making a call to processComponent() method of the current FPGAContext (which at current level is referring to a CLB object). The actual call to the method is:

```java
newContext = FPGAContext.processComponent(N);
```

This call transfers the program execution automatically to the CLB.processComponent() method of the CLB object (through Polymorphism [31]). The method is listed below:
Since the context parameter passed to this method is equal to node N above, the method returns back BLE child object with ID="0" (lines 2 and 3) from its BLE array which becomes the newContext.

Now all the children of the XML node N are found out. In this example this would be list of the following nodes:

```
<B CLASS="LUT">...</B>
<B CLASS="MUX:IN" ID="0" LINES="8">...</B>
<B CLASS="MUX:IN" ID="1" LINES="8">...</B>
<B CLASS="MUX:IN" ID="2" LINES="8">...</B>
<B CLASS="MUX:OUT">...</B>
```

As you can see, these are the second level XML nodes in Figure 4-3. Now each of these child XML nodes (say childNode[i]) is passed as a parameter to a recursive call of genBits() as following:

```
genBits(newContext, childNode[i], bitRowOffset, bitColOffset);
```

The bit location offset parameters (bitRowOffset, bitColOffset) are automatically passed from one recursive level to the next to maintain the relative offsets. These offsets are added to the x and y attributes of the XML Node N before passing the node as parameter to the processComponent() method.

Lets consider the above call to genBits() for the first child node childNode[0] ( 

```
<B CLASS="LUT" ID="0">...</B>
```

).
So in this level of recursion we have:

- FPGAContext → BLE
- N → LUT XML Node (<B CLASS="LUT" ID="0"> …</B>)
- bitColumnOffset → 7 (<B CLASS="LUT" ID="0">)
- bitRowOffset → 8 (<B CLASS="LUT" ID="0">)

The BLE.processComponent() method returns the Lut object as the newContext.

The children of the LUT XML Node are as follows (third level in the Figure 4-3):

```
<B ID="0" X="15" Y="0"/>
<B ID="1" X="15" Y="1"/>
<B ID="2" X="15" Y="2"/>
<B ID="3" X="15" Y="3"/>
<B ID="4" X="15" Y="4"/>
<B ID="5" X="15" Y="5"/>
<B ID="6" X="15" Y="6"/>
<B ID="7" X="15" Y="7"/>
<B ID="8" X="8" Y="0"/>
<B ID="9" X="8" Y="1"/>
<B ID="10" X="8" Y="2"/>
<B ID="11" X="8" Y="3"/>
<B ID="12" X="8" Y="4"/>
<B ID="13" X="8" Y="5"/>
<B ID="14" X="8" Y="6"/>
<B ID="15" X="8" Y="7"/>
```

These are the 16 XML Nodes representing the 16 SRAM cells in a 4-Input LUT.

Again here each of these XML nodes are passed one by one as parameters to

genBits() as before.

Since there are no x and y attributes specified for the LUT XML node (N), the bit
location offsets remain unchanged.

For the next level of recursion we consider the call with the first child node as parameter.
So now in the next level of recursion we have:

- FPGAContext → LUT

- \( N \rightarrow \text{SRAM cell Node} \) \(<\text{B ID="0" X="15" Y="0"/>}\)

- \( \text{bitColumnOffset} \rightarrow 7 + 15 = 22 \) \(<\text{B ID="0" X="15" Y="0"/>}\) (offsetting)

- \( \text{bitRowOffset} \rightarrow 8 + 0 = 8 \) \(<\text{B ID="0" X="15" Y="0"/>}\) (offsetting)

The \( \text{Lut.processComponent(N)} \) method returns the \( \text{SwitchEntity} \) object corresponding to \( N \).

\( \text{SwitchEntity} \) is the class corresponding to an SRAM cell in the Circuit Object Hierarchy. This \( \text{SwitchEntity} \) object corresponds to the “shaded” SRAM cell object in Figure 4-2.

Since the XML node \( N \) has no children (as it as node representing SRAM cell), it is itself passed as XML Node parameter to the next recursive \( \text{genBits()} \) call.

Note the previous level offsets have been added to the current location offsets.

At the next level of recursion we have:

- FPGAContext → SwitchEntity

- \( N \rightarrow \text{SRAM cell Node} \) \(<\text{B ID="0" X="15" Y="0"/>}\)

- \( \text{bitColumnOffset} \rightarrow 22 \) (null)

- \( \text{bitRowOffset} \rightarrow 8 \) (null)

\( \text{SwitchEntity.processComponent(N)} \) sets the SRAM bit location value as as follows:

```java
setX(Integer.parseInt(context.getAttribute(XMLTags.A_X)));
setY(Integer.parseInt(context.getAttribute(XMLTags.A_Y)));
```
The calls to `setX()` and `setY()` method of the `SwitchEntity` object set the location of the “shaded” SRAM cell (Figure 4-2) in the SRAM cell grid.

`SwitchEntity.processComponent(N)` returns `null` as the new `FPGAContext`. So the tree traversal terminates at the next level of recursion (Figure 4-1). Thus starting from CLB and traversing down to a child SRAM cell, the location of the SRAM cell is filled in making it ready to be serialized into the bitstream.

The same method is followed for other paths in the tree traversal leading up to the allocated SRAM cell objects in the Circuit Object Hierarchy as leaf nodes to fill up the location values in all of them.

4.2.5. `processComponent()` Interface

The `processComponent()` interface (or interface method to be more precise) is the point of communication between the Architecture Hierarchy Traverser and the Circuit Object Hierarchy keeping them independent of each other. Since it’s just an interface, it allows one framework component to interact with others without worrying about the implementation. That is to say, the client component implementation can be completely changed without affecting the execution flow of the Architecture Hierarchy Traverser, if both of them adhere to this interface. This kind of architecture provides immense benefits in terms of extending and modifying the functionality of the framework with ease.

Let’s take a look at the interface definition in more detail. The Java definition of the `FPGAContext` interface is shown below:

```
interface FPGAContext {
```

As you can see, this is a really simple interface definition with just one method. The FPGAContext interface defines one public method, `processComponent()` as part of its interface. This means that any class implementing this interface has to at least provide the implementation for this one method. What that implementation actually is, is not specified by the interface. So there is clear separation between implementation and interface. The `processComponent()` method has one argument `context` of type `org.w3c.dom.Node`. This is a parameter which provides context information from the architecture XML file in form of a DOM tree XML node. This single node parameter which encapsulates all the attribute and context information is what makes this interface simple.

As the return value, this method returns an object which again implements the FPGAContext interface. This return value represents the FPGAContext corresponding to the given parameter XML Node `context`. So in essence, all this interface does externally is that it takes in a node from the DOM tree domain and returns the corresponding FPGAContext from the Circuit Object Hierarchy. The relation between the input and output parameter, or how the conversion is done, is completely the responsibility of the implementation. Delegating this work to the implementation is a decision that was consciously made to make XBits framework much more flexible and to easily integrate with higher level tools.

In case of XBits framework, the direct implementation to the FPGAContext interface is provided by the FPGAEntity abstract base class. This class is the base class for all the
component classes of the FPGA chip. All other classes extend this class to add functionality to it (i.e. override the `processComponent()` method to do something useful).

So in effect every entity object of the chip, no matter how large or small it is, is an `FPGAEntity`. This includes the whole FPGA chip class (`FPGA`), the SRAM Switch class (`SwitchEntity`) and also every other class in Circuit Object Hierarchy for which bit generation is carried out. Since every object derived from `FPGAEntity` implements the `FPGAContext` interface, it can be very easily referred to in the Architecture Hierarchy Traverser for traversing the Circuit Object Hierarchy. The benefits of having this class as the base for all the chip component classes are two fold. First, since this class implements `FPGAContext`, we can provide a base implementation for the `processComponent()` method which can be called in the derived classes using `super.processComponent()`. This can prove really helpful if some common processing needs to be performed on all XML nodes. Secondly, this base class allows us to factor out common attributes and functionality required for all the chip components and reuse it in derived classes. It is important to note here that though we are using a base class for all the chip components to make it more convenient to process them, it is not necessary to do so. As long as we adhere to the `FPGAContext` interface, the Architecture Hierarchy Traverser should be able to interact with the components easily. This is especially notable in case where we want to integrate chip component class hierarchies of pre-existing circuit description generation tools.

So what does `processComponent()` do? In brief, we can say that it is responsible for processing the node given as parameter to it and based on the node attributes return a new `FPGAContext` which can be used by the Architecture Hierarchy Traverser to
traverse further. This is specified functionality as defined by the FPGAEntity interface and XBits framework. But, in addition to the designated role, it may generate the location aspect of the bits, the value aspect, both or even none. It may also indirectly help in generation of bits for sub components. Further more, it is free to perform any additional operations which may affect the whole chip structure or a specific part of it. Lets take a look at an example case where processComponent() has been overridden to perform specific functions pertaining to the FPGA object in the Circuit Object Hierarchy. This will not only give us a sneak preview at the inner workings of XBits framework but also serve as groundwork knowledge before moving onto the section on Integrating New Chip Components.

```java
//
// public FPGAEntity processComponent(org.w3c.dom.Element context) {
//  generate the routing bits for components with FPGA object
//  genRoutingBits();
//  // logic blocks (MTLCs)
//  //
//  if (context.getAttribute(XMLTags.A_CLASS).equals("LB"))
//    return (getCLB(Integer.parseInt(context.getAttribute(XMLTags.A_COL)),
//                      Integer.parseInt(context.getAttribute(XMLTags.A_ROW))));
//  // switch blocks (SBs)
//  //
//  if (context.getAttribute(XMLTags.A_CLASS).equals("SB"))
//    return (getSB(Integer.parseInt(context.getAttribute(XMLTags.A_COL)),
//                  Integer.parseInt(context.getAttribute(XMLTags.A_ROW))));
//  // IO Blocks
//  //
//  if (context.getAttribute(XMLTags.A_CLASS).equals("IOB"))
//    return (getIOB(Integer.parseInt(context.getAttribute(XMLTags.A_COL)),
//                    Integer.parseInt(context.getAttribute(XMLTags.A_ROW))));
//  return (null);
```

Figure 4-4 processComponent() (FPGA)

This FPGA object corresponds to the XML Node with “<B CLASS="CHIP"... ”, which, as expected, is the root node in the DOM tree. The node that the processComponent() of
FPGA object actually receives as an input parameter is supposed to be a second level sub component of the whole chip. It is the responsibility of processComponent() to return the appropriate FPGAContext corresponding to the parameter node. But before everything else, a call is made to genRoutingBits() [41], which generates the values for various SRAM cell bits pertaining to routing, for the whole FPGA chip. This reason that it is included in processComponent() of FPGA object is because FPGA.processComponent() has access to all the subcomponents needed for generating the routing bits for the entire chip and it is called right after all the circuit description information has been read from the corresponding files into memory structure. The locations for these routing bits are generated during the further traversal of the tree. The rest of the method demonstrates the typical use of processComponent() where a selection of the appropriate FPGAContext is done based on the CLASS and other attributes of XML node parameter. In this particular case, second level components like CLB (MTLC), IO Blocks (IOB), Switch Blocks (SB) etc. are chosen and returned corresponding to the input node. These returned values are used by the Architecture Hierarchy Traverser to choose which branch of the tree to process. If the node parameter does not match the criteria for any of the appropriate FPGAContext at this level, a null value is returned to indicate to the genBits() method to stop traversing along this branch of the tree. This can be very useful in case of generation of bits for selective components or area on the chip.

4.2.6. Bitstream Generator

Bit stream generation is not complete until the bit values have been stored in the output bit stream array. This function is performed by the Bit-Stream Generator
framework component. We have defined generation of bits as the process of storing in the field values of the various SwitchEntity objects (representing SRAM cells on chip) belonging to chip components. Once all the SwitchEntity objects have been generated, logically bit generation is complete. But, to transfer these bits to the MT-FPGA chip, they have to be converted to a form, which is understandable to a circuit configurator. This form is usually just a sequential stream of bits. The BitStreamArray class performs this conversion of collection of SwitchEntity objects to a sequential stream.

Before we describe this conversion, let’s see where this collection of switch bit objects comes from. As we saw in section 4.2.4, the processComponent() method of SwitchEntity object itself sets the X and Y fields (location aspect) and higher level objects (from the Circuit Object Hierarchy) or framework components (Circuit Description Parsers) set the bit value field of the SwitchEntity object. These SwitchEntity objects are constructed on a per need basis by higher level chip components. The collection of these SwitchEntity objects is implicitly formed upon their construction. That is, the constructor method automatically adds each new instance to a globally maintained collection. This approach eliminates the need for other components to worry about allocation and organization of SwitchEntity objects into a collection.

The BitStreamArray class comes into action once the Architecture Hierarchy Traverser has finished traversal of the DOM tree and the Circuit Object Hierarchy has “completely” generated bits for the whole chip. At this point, the automatically constructed collection of the SRAM bit objects is passed to it. It first performs the
conversion of this collection to a two dimensional array of boolean bits. The collection is iterated, the location (X, Y) and bit value fields are used to set value into the Boolean bits of the two dimensional array. In this way the grid of bits needed to configure the chip is formed.

This grid still needs to be serialized and sent to the chip through a circuit configurator. The bit stream passed to it is emitted by the serialize() method. The serialize() method converts the grid to a serial stream of bits based on a user defined streaming policy. The current version of Xbits framework supports the following policies constants, which may be passed as a argument to the method:

- **ROW_MAJOR**: traverses the grid row by row and emits bits values as they are traversed
- **COLUMN_MAJOR**: traverses the grid column by column

In addition to these policies, the serialize() method can be easily overridden to provide for custom serialization for specific types of chips.

### 4.3. Utility features of the Architecture Description File Format

Now that we have an understanding of the overall genBits() algorithm, let's discuss some of the utility features supported by the Architecture Hierarchy Traverser component to make the Architecture Description Format more concise and easier to create. The utility features for the Architecture Hierarchy Traverser help make the format of the XML architecture specification file concise and manageable (both for the human user as well as for the XBits Framework), even though they do not provide
any special advantages for bit generation from the higher level circuit description. These features are applied to the attributes of the XML node just before passing it to the `processComponent()` method.

4.3.1. Ranging

Ranging allows the user to automatically specify a node over a range of values for the bit offsets, by declaring the node once and specifying the bit offset attributes in form of a range. For example, consider the following extract from Figure 4-3 from section 4.2.4:

```
<B CLASS="LUT">
  <B ID="0" X="15" Y="0"/>
  <B ID="1" X="15" Y="1"/>
  <B ID="2" X="15" Y="2"/>
  <B ID="3" X="15" Y="3"/>
  <B ID="4" X="15" Y="4"/>
  <B ID="5" X="15" Y="5"/>
  <B ID="6" X="15" Y="6"/>
  <B ID="7" X="15" Y="7"/>
  <B ID="8" X="8" Y="0"/>
  <B ID="9" X="8" Y="1"/>
  <B ID="10" X="8" Y="2"/>
  <B ID="11" X="8" Y="3"/>
  <B ID="12" X="8" Y="4"/>
  <B ID="13" X="8" Y="5"/>
  <B ID="14" X="8" Y="6"/>
  <B ID="15" X="8" Y="7"/>
</B>
```

Figure 4-5 Architecture Description File (without using ranging)

The above can be represented, using ranging, as:

```
<B CLASS="LUT">
  <B ID="0..7" X="15" Y="0...7"/>
  <B ID="8..15" X="8" Y="0...7"/>
</B>
```

Figure 4-6 Architecture Description File (using ranging)

Notice how the use of ranges can considerably reduce the number of XML Nodes required. This ".." can be used with any numerical attribute in any block (`<B>`) XML Node. The Architecture Hierarchy Traverser handles Ranging by iterating over the
range specified and substituting the value of the current iteration into the node attribute.

### 4.3.2. Offsetting

Offsetting is a feature which allows us to specify a top level offset to a numerical attribute of XML nodes which is automatically added to the value of same attribute in its child nodes. Consider again, the following extract from Figure 4-3:

```xml
<B CLASS="LB" X="7" Y="8" ROW="3" COL="1" NAME="DEF_MTLC">
  <B CLASS="LE" ID="0">
    <B CLASS="LUT">
      <B ID="0" X="15" Y="0"/>
      <B ID="1" X="15" Y="1"/>
      <B ID="2" X="15" Y="2"/>
      <B ID="3" X="15" Y="3"/>
    </B CLASS="LUT">
  </B CLASS="LE" ID="0">
</B CLASS="LB" X="7" Y="8" ROW="3" COL="1"
```

**Figure 4-7 Example (Offsetting)**

The XML Node corresponding to the CLB (`<B CLASS="LB" X="7" Y="8" ROW="3" COL="1" NAME="DEF_MTLC">`) specifies 7 and 8 and the X and Y bit location offsets respectively. These are added to bit location offsets of all its child nodes. So for instance, these values are added to the bit locations of the XML Node corresponding to the first SRAM Cell of the LUT (`<B ID="0" X="15" Y="0"/>`) before it is passed to the `processComponent()` method. So the actual offsets of this SRAM Cell XML Node become `X="22"` and `Y="8"`. The advantage of offsetting will become more apparent when we look at referencing in the next section.

### 4.3.3. Referencing

Referencing allows the user to substitute (replicate) an already defined XML Node at current location in the Architecture Description File, by referring to its NAME in the REF attribute (`REF="NAME"`). This substitution (replication) is virtual, in the sense that the referred node is not physically copied. Only a reference is maintained to it. This
reference is used during the DOM tree traversal to jump to the referenced node, effectively substituting the referenced node in current position. It should be noted here that the attributes of the original ("referencing") node are not replaced. This allows the user to for instance specify new bit location offsets for the substituted node. We demonstrate the application of referencing here with a simple example. Let’s say we want to define another CLB (MTLC) in the Architecture description file which has the exact same structure as the one shown in Figure 4-3 but is located in a different area on the chip resulting in different (offset) locations for its SRAM cells. One method for this would be to physically replicate all the lines pertaining to this CLB and edit all the bit location offsets in the copied lines manually. A more concise equivalent using referencing is displayed in Figure 4-8.

```xml
<B CLASS="LB" X="7" Y="8" ROW="3" COL="1" NAME="DEF_MTLC">
  <B CLASS="LE" ID="0">
    <B CLASS="LUT">
      <B ID="0" X="15" Y="0"/>
      <B ID="1" X="15" Y="1"/>
      <B ID="2" X="15" Y="2"/>
      <B ID="3" X="15" Y="3"/>
      <B ID="4" X="15" Y="4"/>
    </B>
  </B>
</B>
<B CLASS="LB" X="50" Y="108" ROW="4" COL="5" REF="DEF_MTLC"/>
```

**Figure 4-8 Example (Referencing)**

The above code defines two instances of the CLB, one with bit location offsets $X=7$ and $Y=8$ and the other with $X=50$ and $Y=108$. As you can see, in the second instance you do not need to specify the child nodes since they are automatically substituted from the first XML node defining the CLB ($<B CLASS="LB" X="7" Y="8" ROW="3" COL="1" NAME="DEF_MTLC"> ... </B>$). Figure 4-9 shows the DOM tree structure corresponding to
Figure 4-8. Notice the tree needs to be defined for only one CLB (MTLC) node, thus considerably saving memory.

Figure 4-9 DOM tree structure showing referenced CLB Node

4.4. Integrating new chip components

One of the key areas of extending XBits is by integrating new chip components into the Circuit Object Hierarchy. Integrating new chip components into the XBits framework is a three-step process. The first step of adding a new type of component to the Architecture Description File involves inserting a new block element tag (<B>) and specifying the new class name as the CLASS attribute (the class name can be chosen arbitrarily). The block element corresponding to this new chip component may be comprised of the other block elements which the framework already knows how to handle. Note that no change to the XML Format is needed.
The second step is providing the class code implementation for handling the integration of the component into the Circuit Object Hierarchy. This is where you create a class based data structure model for storing the generated circuit description information for the new chip component. You may choose to derive this new data model as a class from FPGAEntity or simply choose to implement the FPGAContext interface if you are using legacy code. You can also derive from an existing class in the Circuit Object Hierarchy to reuse its behavior. You can also use the existing classes from the Circuit Object Hierarchy to “compose” the new component’s data model. This model needs to be filled in by higher level layers or framework components in the tool chain (Note: covering the changes needed to higher level layers like Mapping, Placement and Routing etc. is out of scope of this discussion).

Once the data model is constructed, we need to implement the interface (processComponent()) between the Architecture Hierarchy Traverser and the Circuit Object Hierarchy. You will also need to modify the parent component class (probably an existing class of the Circuit Object Hierarchy) to handle this new type of child object. Here we present a simple example of integrating a 3-Input LUT. In general the following steps will be involved in integrating a new component. Notice the reuse of the existing 4-Input LUT class to implement the 3-Input LUT:

**Step 1: Specifying the structure and bit location offsets of the component in the Architecture Description File**

```
<B CLASS="LUT_3" >
  <B ID="0" X=".." Y=".."> <!--  LUT bits -->
  <B ID="1" X=".." Y="..">
  <B ID="2" X=".." Y="..">
  <B ID="3" X=".." Y="..">
  <B ID="4" X=".." Y="..">
  <B ID="5" X=".." Y="..">
```
Step 2: Re-use the base `Lut` class (4 -Input) and extend it to support 3-Input LUT

```
public class LUT_3 extends Lut implements FPGAContext {
    private LUT_3 lut = new LUT_3();
    FPGAContext processComponent(org.w3c.dom.Node context) {
        if (context.getAttribute(XMLTags.A_CLASS).equals(""))
            // restrict to 3-Input LUT
        if (Integer.parseInt(context.getAttribute(XMLTags.A_ID) < 7))
            return (super.processComponent(context));
    }
}
```

Step 3: Modify the parent class to handle the new component

```
public class BLE implements FPGAContext {
    private LUT_3 lut = new LUT_3();
    FPGAContext processComponent(org.w3c.dom.Node context) {
        if (context.getAttribute(XMLTags.A_CLASS).equals("LUT_3"))
            return (lut[lut.getAttribute(XMLTags.A_ID)]);
    }
}
```

Although this is a trivial example, it shows how new components can be plugged into XBits without affecting the rest of the framework. It must be noted here that this section describes the changes needed in context of the XBits tool. Of course, appropriate modifications are also needed to be made to the higher level tools in the FPGA design flow (VPR/T-VPack [7]) to support these new components.

4.5. Summary

In this chapter, we looked into the various implementation aspects of XBits framework. These implementation details, though not directly necessary to use the framework, provide groundwork knowledge for extending it and customizing it
provide a solution for a particular scenario. We looked at the details of the genBits() and processComponent() methods, the two main methods of the XBits framework. Also we described the functioning of the BitStreamArray class in serializing the bitstream.

We also saw the ways in which XBits framework can be extended by integrating new components.
5. Results and Future Work

We now present some results which were obtained by testing the XBits framework on a few benchmark and custom circuit descriptions. In the following sections, we will be describing the test environment used, the input files used and the output bitstream generated. Based on these results we suggest some areas of enhancements and modifications to the XBits framework which can be taken up as future work.

5.1. Test Environment

The test cases were performed on Intel Pentium 4 platform with Microsoft Windows XP® as the operating system. Sun Microsystems Java 1.4.2® was used to compile and execute XBits. NetBeans 3.6 IDE was used with JFluid plug-in [33] module as the integrated test environment. VPR and T-VPack 4.30 [8] were used to pack, route and place the benchmark circuit given in BLIF net list format.

5.2. Architecture Description Files

As we have already described, one of the input files to the XBits framework (in the process of bitstream generation) is the Architecture Description File (2.2). We created two different versions of these for our testing purposes. The Architecture Description Files were created manually using a simple text editor. For the first run we created a file which represented the architecture of the current version of the MT-FPGA chip [40]. The current version of MT-FPGA has a 3 x 3 CLB (MTLC) array architecture. Using our custom Architecture Description File Format (3.7), we were able to specify the configurable SRAM bits for all of the Digital Components of the chip; in the current chip the analog components are not programmable hence have no
corresponding SRAM bits. For the second test, we extended the current architecture to a hypothetical 5 x 5 CLB array. We were able to leverage the benefits of the referencing and offsetting utility features (4.3.3, 4.3.2) to reuse the existing 3 x 3 Architecture Description File to a large extent. On account of referencing, the CLB (MTLC) array was easily extended by referencing the existing CLB XML node with proper bit location offsets. A similar method was used in the case of other the components like Switch Blocks, IO Blocks etc.

5.3. **Circuit Description (BLIF) files**

For the BLIF files, instead of creating them manually (which is not a trivial task), we used predefined benchmark circuits for mapping onto the above two architectures. The standard circuit b1.blif [34] from the MCNC benchmark set was chosen as the primary example (Figure 5-1). b1.blif defines a simple Boolean expression on the Boolean inputs \(a, b, c\). It describes the circuit at the gate level. For more detailed information of the BLIF format, please refer to [1].

```
.model b1
<inputs a b c
.outputs d e f g
.names n e
 0 1
.names p f
 0 1
.names c g
 0 1
.names a b n
 11 1
.names a b c p
 01-1
10-1
1-1 1
-11 1
0-0 1
-00 1
.names c d
 1 1
.end
```

*Figure 5-1 b1.blif taken from the MCNC benchmark*
The above defined file was used as circuit description for mapping onto the 3 x 3 architecture. For the purpose of the 5 x 5 architecture, we used a modified version of b1.blif. We replicated the logical circuit defined in b1.blif to 4 times to generate a bigger circuit description. This file called b1replicated.blif is shown below:

```
.model b1
.inputs a b c aa bb cc aaa bbb ccc
.outputs d e f g dd ee ff gg ddd eee fff ggg dddd eeee fffff gggg
.names n e
 0 1
.names p f
 0 1
.names c g
 0 1
.names a b n
 11 1
 00 1
.names a b c p
 01- 1
 10- 1
 1-1 1
-11 1
 0-0 1
-00 1
.names c d
 1 1
.names nn ee
 0 1
.names pp ff
 0 1
.names cc gg
 0 1
.names aa bb nn
 11 1
 00 1
.names aa bb cc pp
 01- 1
 10- 1
 1-1 1
-11 1
 0-0 1
-00 1
.names cc dd
 1 1
.names nnn eee
 0 1
.names ppp fff
 0 1
.names ccc ggg
 0 1
.names aaa bbb nnn
 11 1
 00 1
.names aaa bbb ccc pp
 01- 1
 10- 1
 1-1 1
-11 1
 0-0 1
-00 1
.names ccc ddd
 1 1
```
Figure 5-2 b1replicated.blif (Replicated Version of b1.blif)

These input BLIF files were fed to T-VPack to be technology mapped to 4-LUT CLBs. The resulting net list files are b1.net and b1replicated.net.

5.4. VPR Output

The above Net-List files were passed as input respectively to VPR to generate the corresponding Placement and Routing information files (2.2). This information is generated in the form of .p and .r files. Figure 5-3 and Figure 5-4 show the graphical output of the placed and routed circuits generated for the two different architectures. As you can see, the figures show the grid of CLBs (MTLCs) surrounded by the IO blocks. They also display the routing nets as lines connecting the components. The Switch blocks are not shown since this is only a logical representation of the nets. The used components are shaded in gray and labeled with the placed logical element name from the corresponding BLIF file.
Figure 5-3 VPR Graphical Output for b1.blif mapped onto a 3 x 3 Architecture

Figure 5-4 VPR Graphical Output for b1replicated.blif mapped onto a 5 x 5 Architecture
5.5. XBits Output (Bitstream)

Once the circuits defined by the BLIF files were placed and routed, the outputs generated along with the original BLIF and Net-List files were fed as inputs to our XBits framework (as described in 2.2). XBits by default generates the bitstream as a single sequential stream of 0s and 1s printed out to the standard output. For the sake of better understanding and clarity, we present these generated bitstreams grouped by rows, where each row is comprised of the number of bits in a row of the actual SRAM bit array on chip. This helps in better correlation between the graphical output generated by VPR and the bitstreams. Figure 2-2 depicts the generated bitstream for the 3 x 3 CLB architecture.

Figure 5-5 Generated bitstream for b1.blif mapped onto a 3 x 3 architecture
Figure 5-6 shows the bitstream for the 5 x 5 architecture.

As you can see from Figure 5-5 and Figure 5-6, the areas with generated bits correspond closely to the placement of the components shown in Figure 5-3 and...
Figure 5-4 respectively. The bitstream size for the 3 x 3 architecture is 99 x 61 bits, whereas that for the 5 x 5 architecture is 190 x 90 bits. The places in the bitstreams above where a 0 or 1 is shown are those bits which are actually generated as a result of the input circuit. The places marked by dots (.) are those bits which are not affected by the circuit description (and hence can have a default value) or correspond to grid location which don’t have associated SRAM cells. These generated bitstreams can be fed to a circuit configurator which actually downloads them onto the FPGA.

5.6. Performance Comparison

In this section we compare the above defined two test cases in terms of execution delays and memory usage. Figure 5-7 (next page) shows the execution delay recorded over several runs for the whole XBits framework. The execution times for the 5 x 5 architecture are higher than the 3 x 3 architecture as expected. The mean execution time for the 5 x 5 architecture is almost twice that for the 3 x 3 architecture. As you would notice from the figure above, there is considerable variation between execution times in consecutive runs. The main reason behind this is the non-deterministic nature of Garbage Collector (GC, handles automatic memory allocation and de-allocation) which is a part of the Java Virtual Machine (JVM) [36, 37]. This variation effect is even more pronounced in the case of the 5 x 5 architecture as there is a lot more object allocation done for the Circuit Object Hierarchy as well as XML DOM tree (see Figure 5-8 and Figure 5-9). The allocation/de-allocation policies of the Garbage Collector are dependent on the prevailing execution environment as well as the particular JVM implementation.
Figure 5-7 Execution Delay for the whole XBits Framework for the test cases

Figure 5-8 and Figure 5-9 show the heap allocation for the entire course of execution of the XBits framework in case of the 3 x 3 and 5 x 5 architectures respectively. These graphs have been generated using the JFluid Profiling Toolkit for Netbeans IDE. It must be noted here that JFluid API introduces its own overhead into the application execution which is substantial compared to the normal execution times of XBits. As you can clearly infer from a comparison of these two figures, there is a lot more memory allocation and de-allocation (irregularities in Used Heap curve) in case of the 5 x 5 architecture. These irregular points, which denote the execution of GC to re-organize heap, are the major cause for the introduction of variation in the execution times. Some techniques of reducing these irregularities are suggested in the next section as future work.
Figure 5-8 Heap Allocation for the 3 x 3 Architecture (using b1.blif)

Figure 5-9 Heap Allocation for the 5 x 5 Architecture (using b1replicated.blif)

Figure 5-10 summarizes the performance results for the 3 x 3 and 5 x 5 architecture experiment cases, also displaying the average heap usage for DOM tree and Circuit Description. Determining the memory allocated by an object is not a trivial task in
Java, especially if the object is composed of other objects (as is the case for trees).

Therefore, we manually inserted code statements which took snapshots of the global heap around relevant “sections” of the code to compute the best-possible memory usage figures. These figures may not be accurate but give us a good comparative view, since the same mechanism was used to determine them in both experiment cases. One important observation that can be made from Figure 5-10 is that the memory usage for the DOM tree (which stores the parsed information from the Architecture Description File) grows only marginally (less than 3%) from the 3 x 3 architecture to the 5 x 5 architecture. This is on account of the referencing mechanism (4.3.3) which eliminates the need to re-specify the bit location information for pre-defined component every time the component is replicated at an offset.

<table>
<thead>
<tr>
<th></th>
<th>3 x 3 Architecture</th>
<th>5 x 5 Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean execution time</td>
<td>1856 ms</td>
<td>3940 ms</td>
</tr>
<tr>
<td>Maximum available (allotted) heap</td>
<td>2 MB</td>
<td>2 MB</td>
</tr>
<tr>
<td>Maximum used heap</td>
<td>1.2 MB</td>
<td>1.5 MB</td>
</tr>
<tr>
<td>Average heap used for Circuit Description</td>
<td>65 KB</td>
<td>320 KB</td>
</tr>
<tr>
<td>Average heap used for DOM tree</td>
<td>93 KB</td>
<td>96 KB</td>
</tr>
</tbody>
</table>

Figure 5-10 Performance comparison summary between 3 x 3 architecture and 5 x 5 architecture experiment cases

5.7. Future Work

One of the major goals behind the creation of this framework was to be able to apply and use it not for a specific chip but any generic architecture. Although we have tested XBits with relatively simple and small test circuits, we believe the XBits framework can be a very promising option for generating bit streams for larger
circuits. It would be interesting to create Architecture Description Files for chip architectures larger than 5 x 5 CLB (MTLC) grid array and map larger circuit descriptions onto them. An automated system for generation of these Architecture Description Files (specifying structure and SRAM bits of the FPGA chip) would greatly benefit the process of studying the use of XBits for these architectures.

The current version of XBits does not take into account the behaviour of Java Garbage Collector mechanism. A lot of improvements can be made to the current methods of object allocation to optimize the memory allocation performance with respect to the GC implementation. A number of techniques for doing so have been show in [37]. These techniques would also help in stabilizing the Adaptive Heap Expansion for larger circuits.

Another aspect where XBits can be very easily extended is the support for new components of varied nature (like analog, digital etc.). As we saw in 4.4, this process can be performed incrementally by extending the Circuit Object Hierarchy and adding to the Architecture Description File. The inherent object oriented design of XBits minimizes the effort required in doing so. We expect that a major portion of future work will be done in the direction of adding new component classes for newer chips. Although the current version of the XBits supports analog components, this support is limited by the ability of higher levels tools to generate circuit description information for these components. A lot of work needs to be performed on these higher level tools for supporting these newer components. This would also include defining the circuit description inputs and outputs pertaining to these components for
the higher level tools. Currently the Multi-Technology Components are treated as I/O to the digital portion of the device.

The Architecture Description File format already supports a number of utility features like offsetting, referencing etc. (4.3) which make the task of creating it much simpler. The current design of XBits allows adding of new utility features very easily without affecting the any other framework components besides the Architecture Hierarchy Traverser. A lot of new utility features like include files, independent component element tags (which can be used as building blocks) etc. can be included to make the format more modular and usable.

### 5.8. Summary

In this chapter we looked at the results generated for the two test cases (3 x 3 CLB with `b1.blif` and 5 x 5 architecture with `b1replicated.blif`) and compared them on various grounds. We also looked at some enhancements that can be made to the XBits framework and saw how these enhancements would benefit us.
6. Conclusion

XBits is a bitstream generation tool for configuring Multi Technology FPGA chips. It features a novel object oriented framework built on technologies like Java and XML. The Architecture Description File Format specified by XBits can represent FPGA chips which have digital, analog and mixed signal components. XBits provides the base infrastructure for building complex tools incorporating more functionality for handling these components.
7. References


3. Integrated Software Environment (ISE), Xilinx Corp.,


8. V. Betz et al., VPR and T-VPack, March 2000,
   http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html


10. Circuit Cellar, “The FPGA Tour”, November 1999,


18. J. Ziv, A. Lempel, “A Universal Algorithm for Sequential data Compression
20. I. H. Witten, R. M. Neal, J. G. Cleary, “Arithmetic Coding for Data Compression
21. Z. Li, S. Hauck, “Don’t Care Discovery for FPGA Configuration Compression”,
   ACM/SIGDA International Symposium on Field -Programmable Gate Arrays, pp.
   91-100, 1999.
24. XML Applications, W3Schools,
       http://www.w3schools.com/xml/xml_applications.asp
25. Java API for XML Processing (JAXP), Sun Microsystems, 2005,
       http://java.sun.com/xml/jaxp/index.jsp
27. org.w3c.dom Javadoc Reference, Sun Microsystems,
       http://java.sun.com/j2se/1.4.2/docs/api/org/w3c/dom/package-summary.html
28. Java Architecture for XML Binding (JAXB), Ed Ort and Bhakti Mehta, March
   2003,
       http://java.sun.com/developer/technicalArticles/WebServices/jaxb/index.html
29. org.w3c.Node Javadoc Reference, Sun Microsystems,
       http://java.sun.com/j2se/1.4.2/docs/api/org/w3c/dom/Node.html


32. Weifeng Xu et al., VPR for Virtex + JBits Interface, http://www-unix.ecs.umass.edu/~wxu/jbits/


34. “b1.blif”, http://www.ift.cs.uni-potsdam.de/~alexej/FtS_Praktikum/b1.blif


