Analysis of runtime re-configuration systems

Utthaman Thirunavukkarasu
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds
Part of the Higher Education Commons

Repository Citation
https://digitalscholarship.unlv.edu/rtds/1988

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
ANALYSIS OF RUNTIME RE-CONFIGURATION SYSTEMS

by

Utthaman Thirunavukkarasu

Bachelor of Computer Science & Engineering
Madras University
1999

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Electrical and Computer Engineering
Department of Electrical and Computer Engineering
Howard R. Hughes College of Engineering

Graduate College
University of Nevada Las Vegas
May 2006
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI Microform 1436801
Copyright 2006 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The Thesis prepared by

Utthaman Thirunavukkarasu

Entitled

"Analysis of Runtime Reconfiguration Systems"

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
ABSTRACT

Analysis of Runtime Re-Configuration Systems

by

Utthaman Thirunavukkarasu

Dr. Henry Selvaraj, Examination Committee Chair
Professor of Electrical and Computer Engineering
University of Nevada, Las Vegas

In recent years Programmable Logic Devices (PLD) and in particular Field Programmable Gate Arrays (FPGAs) have seen a tremendous increase in sales and applications in the area of embedded systems. The main advantage of FPGAs is the flexibility that they offer a designer in reconfiguring the hardware. The flexibility achieved through re-configuration of FPGAs usually incurs an overhead of extra execution time, data memory and also power dissipation.

FPGAs provide an ideal template for run-time reconfigurable (RTR) designs. Only recently have RTR enabling design tools that bypass the traditional synthesis and bitstream generation process for FPGAs become available, JBits is one of them. With run-time reconfiguration of FPGAs, we can perform partial reconfiguration, which allows reconfiguration of a part of an FPGA while the other part is executing some functional computation. The partial reconfiguration of a function can be performed earlier than the time when the function is really needed. Such configuration pre-fetch can hide the reconfiguration overhead more effectively.
This thesis will implement a reconfigurable system and study the effect of runtime reconfiguration using VERILOG and a new Java based tool JBITS. This work will provide pointers to high level synthesis tools targeting runtime re-configuration.
TABLE OF CONTENTS

ABSTRACT ......................................................................................................................................... iii

LIST OF FIGURES ........................................................................................................................... vii

LIST OF TABLES ............................................................................................................................. vii

ACKNOWLEDGEMENTS ................................................................................................................ ix

CHAPTER 1 INTRODUCTION .................................................................................................... 1
  1.1 Field Programmable Gate Arrays and Reconfigurable Computing ....................... 2
  1.2 Run-Time Reconfigurable Computing ......................................................................... 3
  1.3 JBits ....................................................................................................................................... 5
  1.4 JPEG 2000 .......................................................................................................................... 5
  1.5 FPGAs and Image Processing ......................................................................................... 8
  1.6 Motivation and Contribution of the thesis ................................................................... 9
  1.7 Organization ....................................................................................................................... 10

CHAPTER 2 BACKGROUND ................................................................................................... 11
  2.1 Run Time Reconfiguration ............................................................................................ 11
  2.2 Stages in the JPEG2000 Algorithm ............................................................................. 16
  2.3 MQ Encoder .................................................................................................................... 19

CHAPTER 3 TOOLS USED ........................................................................................................ 29
  3.1 VERILOG Language ........................................................................................................ 29
  3.2 Memec Insight Virtex-II MB ....................................................................................... 30
  3.3 Xilinx ISE 5.0 ................................................................................................................ 31
  3.4 JBits 3.0 .......................................................................................................................... 33

CHAPTER 4 METHODOLOGY ................................................................................................. 43
  4.1 VERILOG Implementation ........................................................................................... 43
  4.2 Synthesis and Power Estimation of the Design ......................................................... 48
  4.3 JBits and XHWIF ............................................................................................................. 51

CHAPTER 5 RESULTS AND DISCUSSIONS ..................................................................... 55

CHAPTER 6 CONCLUSION ..................................................................................................... 64

REFERENCES ............................................................................................................................. 65
LIST OF FIGURES

Figure 1.1 Traditional FPGA Design Flow ................................................................. 3
Figure 1.2 JBits Design Flow .................................................................................... 6
Figure 2.1 Stages in JPEG2000 Encoding Process .................................................. 17
Figure 2.2 Basic structure of a generic Q-coder ....................................................... 20
Figure 2.3 Pseudo code of Q-coder ........................................................................ 21
Figure 2.4 The interval in the unit interval [0, 1] divides into LPS subinterval and MPS subinterval respectively ................................................. 22
Figure 2.5 Probability Estimation ......................................................................... 24
Figure 2.6 An example of probability estimation for an LPS followed by a sequence of MPS Symbols .................................................. 25
Figure 2.7 Probability Estimation Table ................................................................ 26
Figure 3.1 Block diagram of Memec Insight Virtex-II MB Development Kit Board .............................................................. 31
Figure 3.2 JBits Environment ............................................................................... 33
Figure 3.3 Static Design Flow ............................................................................. 35
Figure 3.4 RTR Design Flow ............................................................................... 36
Figure 3.5 RTR Execution Model ....................................................................... 37
Figure 3.6 Snapshot of the BoardScope tool ....................................................... 42
Figure 4.1 Design Flow for Power calculation and FPGA configuration .......... 50
Figure 5.1 Simulation waveforms of the verilog_file_1.v code ......................... 56
Figure 5.2 Floorplan for verilog_file_1 design on XC2V1000 FPGA ............... 58
Figure 5.3 Routing Congestion for verilog_file_1.v design on xc2v1000 FPGA ... 59
Figure 5.4 Floorplan of the verilog_file_1.v design on xc2v1000 FPGA ......... 60
Figure 5.5 Routing Congestion for verilog_file_1.v design on xcv1000 FPGA .. 61
Figure 5.6 Design view of verilog_file_1.v on xcv1000 FPGA using BoardScope ............................................................................. 62
Figure 5.7 Design view of verilog_file_2.v on xcv1000 FPGA using BoardScope ............................................................................. 62
LIST OF TABLES

Table 4.1  Arithmetic Encoder Main Data Inputs/Outputs .........................................45
Table 4.2  Arithmetic Encoder Tasks..............................................................................45
Table 4.3  Main Data path Registers C............................................................................46
Table 4.4  Outputs of the Arithmetic Encoder VERILOG Entity......................................48
Table 5.1  Resource Utilization of the designs on
            XC2V1000 FPGA............................................................................................57
Table 5.2  Dynamic Power consumption of the designs on
            XC2V1000 FPGA............................................................................................58
Table 5.3  Resource Utilization of the designs on XCV1000 FPGA.........................59
Table 5.4  Dynamic Power consumption of the designs on
            XCV1000 FPGA...............................................................................................60
ACKNOWLEDGEMENTS

I would like to acknowledge the immense assistance and moral support provided by my advisor, Dr. Henry Selvaraj during the course of my masters program at the University of Nevada, Las Vegas. The guidance provided by him in steering this project from concept to completion has been invaluable.

I would like to thank Dr. Muthukumar Venkatesan for all his direct and indirect support throughout this investigation. It is also important to mention the moral support of my immediate family including my parents and my brother who were available for me at all times. But for their constant motivation, it would have been impossible for me to get this far in life.

I would like to acknowledge the help of Gopinath Balakrishnan, Balaji Anandsagar and all my friends for their help and advice which has always been an invaluable source of motivation for me.

Finally I would like to thank the Department of Electrical and Computer Engineering at the University of Nevada, Las Vegas for giving me an opportunity to pursue my masters’ degree.
CHAPTER 1

INTRODUCTION

Decades ago, electrical engineers had to use multiple dedicated chips to perform logic functions. Later, these chips were consolidated into general devices that could be customized for specific Boolean operations. As transistors became smaller, these devices became more powerful and their range of operations became greater. FPGAs represent the most recent and most powerful of these customizable parts and their complexity have risen to exceed that of regular computer processors. However, computer scientists are yet to recognize the capabilities of FPGA design, which remains the sole province of electrical engineers. FPGAs provide a number of unique advantages, one such advantages involves Run Time Reconfigurable processing. To appreciate these qualities, we need to see how FPGAs fit into a computational spectrum that ranges from fixed-structure to flexible-structure processing, and from programmable to configurable when compared to Digital Signal Processors (DSP). The Digital Signal Processors are designed to perform high speed math and give more flexibility to the user. However, this flexibility is achieved at a cost of higher complexity and programming difficulty. All general purpose processors have a fixed instruction set and no matter how flexible they are, the processors have to work within that instruction set (i.e. you can customize these processors, but you can't change the instruction set in any way). Since the instruction set is determined by the
processor's internal structure, these processors can be called fixed-structure processors and the customization part of this processor is called programming. But FPGAs are different from these processors since with an FPGA, you have the option of using processors with flexible structures which can be reconfigured.

1.1 Field Programmable Gate Arrays and Reconfigurable Computing

FPGA is a short for *Field-Programmable Gate Array*, a type of logic chip that can be reconfigured. Generally speaking it is a piece of hardware that can be rewired to perform any logic. FPGAs consist of thousands of separate, independent computational elements, called logic elements. The process of implementing a design on FPGA is called configuration. When you configure an FPGA, you're changing the structure and behavior of the logic element's internal connections and setting parameters that control its logical functions. An FPGA's logic elements are too small to be useful by themselves, but by configuring and connecting them, you can use the FPGA to perform a fully customized computation. Of course, configuring hundreds of logic elements individually is prohibitively difficult but the presence of modern CAD tools makes it easy for users to configure an FPGA. Most digital designers create designs using one of two languages – VHDL and Verilog. Then, they use specialized software to convert these designs into bit streams. This conversion process is called *synthesis*, which extracts the logical characteristics from a design, and *implementation*, which builds a bitstream according to these characteristics. This bits ream file is then pushed onto an FPGA which in turn is the hardware implementation of the logic function. Any FPGA processing can be called
reconfigurable computing because you can always reset an FPGA back to its blank state and configure it with a new bitstream. For example, you can initially configure a Xilinx

![FPGA Design Flow Diagram]

Figure 1.1: Traditional FPGA Design Flow

Virtex-II FPGA to perform GPS tracking in a cell phone. Then, once the position has been acquired, you can reset the FPGA and reconfigure it to function as a voice decoder. Regular FPGA development is a straightforward process and many sources can be found that explains it in greater depth.

1.2 Run-Time Reconfigurable Computing

Run-time or dynamic reconfiguration of circuits has recently become viable with the introduction of SRAM based dynamically reconfigurable Field Programmable Gate Arrays (FPGAs). Run-time reconfiguration (RTR) or Dynamic reconfiguration (DR)
also called, on-the-fly reconfiguration or in-circuit reconfiguration is a type of reconfiguration which allows modifications of a system configuration during its normal operation without resetting the FPGA. Most traditional FPGAs are configured with full bit streams. That is, after the device has been reset, the bitstream sets each connection and parameter within each Logic Element. However, with new generation FPGAs like the Xilinx Virtex family FPGAs, you can perform partial reconfiguration, in which the partial bitstream affects part of the device. In this case, you don’t need to reset the FPGA to load the other part of the bitstream since it can be done during run-time. This is a very important capability since configuring during runtime allows you to gradually adjust an FPGA’s structure to better reach a goal. For example in signal processing, you can dynamically alter the coefficients of a digital filter to improve signal-to-noise ratio which in turn improves the signal quality. While dynamic reconfiguration has always been possible in all Xilinx SRAM-based parts, very little has been done to provide software support for this capability. In general, the design flow has been limited to static circuit design tools, with schematic capture or Hardware Description Language (HDL) front-ends. In addition, the method used to produce configuration data from these circuits was based on automatic placement and routing technology developed originally for production of printed circuit boards. This approach relied on the solving of known NP-complete problems and was necessarily slow and non-deterministic. The placement algorithms usually provided a physical implementation of the circuit which bore little resemblance to the logical circuit. This made the task of locating items for reconfiguration difficult. To support dynamic reconfiguration of an SRAM based FPGA we require a tool which is as fast as possible, and provide physical information about the
circuit for reconfiguration. One such tool is JBits. A brief description of JBits is given in the next section.

1.3 JBits

The JBits™ software is a set of Java™ classes which provide an Application Programming Interface (API) to access the Xilinx FPGA bitstream. The interface operates on either bit streams generated by Xilinx design tools, or on bit streams read back from actual hardware. This permits all configurable resources like Look-up tables, routing and the flip-flops in the FPGA to be individually configured under software control. The API has been used to construct complete circuits and to modify existing circuits. In addition, the object-oriented support in the Java programming language has permitted a small library of parameterisable, object oriented macro circuits or Cores to be implemented. Finally, this API may be used as a base to construct other tools. This includes traditional design tools for performing tasks such as circuit placement and routing, as well as application specific tools to perform more narrowly defined tasks.

1.4 JPEG 2000

The Joint Photographic Experts Group (JPEG) standard has been in use for more than a decade now. It has proved a valuable tool during all these years, but it cannot fulfill the advanced requirements of today. Today’s digital imagery is extremely demanding, not only from the quality point of view, but also from the image size aspect. Current image size covers orders of magnitude, ranging from web logos of size of less than 100 Kbits to
high quality scanned images of approximate size of 40Gbits. With the continual expansion of multimedia and Internet applications, the needs and requirements of the

Figure 1.2: JBits Design Flow

technologies used, grown and evolved and this in turn led to the development of a new image compression standard called JPEG 2000. The JPEG 2000 standard, finalized in 2001, defines a new image-coding scheme using state-of-the-art compression techniques based on wavelet technology.

The JPEG 2000 international standard represents advances in image compression technology where the image coding system is optimized not only for efficiency, but also for scalability and interoperability in network and mobile environments. Digital imaging has become an integral part of the Internet, and JPEG 2000 is a powerful new tool that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
provides power capabilities for designers and users of networked image applications. The JPEG 2000 standard provides a set of features that are of importance to many high-end and emerging applications by taking advantage of new technologies. The major difference between JPEG and JPEG2000 is that the later uses Discrete Wavelet Transform instead of Discrete Cosine Transform used in JPEG standard. The markets and applications better served by the JPEG 2000 standard are Internet, color facsimile, printing, scanning (consumer and prepress), digital photography, remote sensing, mobile, medical imagery, digital libraries/archives, and E-commerce. Each application area imposes some requirements that the standard, up to a certain degree, should fulfill.

The JPEG2000 algorithm is large and complex. Given the limited scope and time available for this thesis, implementing most of the standard in hardware is not a realistic goal so it was decided to implement a section of JPEG 2000 on hardware. Selecting the section of JPEG2000 to implement on hardware is therefore an important choice to be made. The selection was made so that the greatest possible performance gain is obtained, given that only a subset of the algorithm processing will be carried out in hardware. After going through the previous works and understanding the working of JPEG 2000, the Arithmetic Entropy Encoder stage of the JPEG2000 encoding standard was chosen for hardware implementation. In this thesis the Arithmetic Encoder stage of JPEG2000 is implemented on FPGA and is reconfigured during runtime to explore the advantages and overheads incurred during runtime reconfiguration over static reconfiguration.
1.5 FPGAs and Image Processing

The use of reconfigurable field-programmable gate arrays (FPGAs) for imaging applications show considerable promise to fill the gap that often occurs when Digital Signal Processor (DSP) chips fail to meet performance specifications. Although DSP chips can process data at high-speeds, their architectures can inhibit overall system performance in real-time imaging. The rate of operations can be increased when they are performed in dedicated hardware, such as special-purpose imaging devices or FPGAs, which provides the architecture necessary to implement real-time image processing products successfully and cost-effectively. For many fixed applications, non-SRAM-based (antifuse or flash-based) FPGAs provide the raw speed to accomplish standard high-speed functions. However, in applications where algorithms are continuously changing and compute operations must be modified, only SRAM-based FPGAs give enough flexibility. The addition of reconfigurable FPGAs as a flexible hardware facility enables DSP chips to perform optimally. The benefits primarily stem from optimizing the hardware for the algorithms or the use of reconfigurable hardware to enhance the product architecture. Since the use of JBits for RTR requires a separate “host” processor and Java Virtual Machine (JVM) to execute the Java classes that perform bitstream reconfigurations. It was desirable, therefore, to have the image processing application benefit from the required PC-FPGA shared processing environment. Image processing applications have been shown to benefit from shared processing environments, in which the FPGA is utilized as a co-processor [16, 17]. This concept can be extended to utilize RTR for the core, in which the “host” process defines a specialized circuit coprocessor instance to accelerate computation of the current image-processing task. Designing
specialized circuitry is useful in image encoding applications, where the performance of the encoding algorithm is dependent on the image itself.

1.6 Motivation and Contribution of the thesis

For the design of an RTR application to be justifiable, it should exhibit clear advantages over a similar, static circuit. Although a number of architectures for static ASIC-based Arithmetic Encoder architectures have been explored, they offer little in the way of operation customization. Advantages that can be exploited through RTR include circuit speed increases through decreased latency or increased clock frequency, and decreased resource consumption when compared to the static implementation counterpart. The purpose of this thesis is to explore the advantages of RTR, estimate the overheads incurred during RTR in terms of power and resource and try to suggest methods to minimize it.

The main contribution of this thesis is study and implementation RTR when applied to Arithmetic Encoder of JPEG2000. This thesis presents a working Verilog implementation of the arithmetic encoder stage of the algorithm. The code has been tested using test bench to check whether it confirmed with the JPEG2000 standard set by the Final Draft Committee of JPEG 2000 standard. The Verilog code was simulated and the performance of the design was estimated. The Verilog code successfully passed all simulation tests as a working implementation of the JPEG2000 arithmetic encoding stage. Since JBits 3.0 supports Virtex-II – FPGAs, the Memec Insight Virtex-II MB Development Kit was eventually identified as an appropriate product available that fulfilled the necessary requirements since JBits 3.0 supports Virtex-II FPGA and with a
million gates Virtex-II FPGAs gives us more flexibility for partially reconfiguring the FPGA and eventually is very well suited for RTR. At this point, the Verilog modules were synthesized and programmed into the FPGA. Once the FPGA was configured JBits was used to reconfigure the hardware in run time and the results were analysed, which is discussed in the results chapter of the thesis.

1.7 Organization

As explained in this introductory chapter, this thesis presents the results of RTR when applied to the Arithmetic Encoder of JPEG 2000. As background, Chapter 2 discusses previous work that is related to this thesis. It discusses the previous works on Runtime Reconfiguration and exiting JPEG 2000 image processing algorithm. Chapter 3 discusses the tools selected for RTR and then tries to justify the selection of Virtex-II FPGA for the experiment. Chapter 4 explains the complete experiment process, the Verilog code, simulation waveforms, synthesis and implementation of the code, and the JBits environment in which RTR of the FPGA is carried out.

Chapter 5 of the thesis discusses the results and Chapter 6 concludes the thesis with consideration given to future developments that could arise out of the work presented in this thesis.
CHAPTER 2

BACKGROUND

2.1 Run Time Reconfiguration

2.1.1 Introduction

Nowadays many emerging applications in communication, computation and consumer electronics demand that their functionality stays flexible after the system has been manufactured. Such flexibility is required in order to cope with the changing user requirements, improvements in system features, changing protocol and data-coding standards, demands for support of a variety of different user applications, etc. Until recently, FPGA has only been used in prototyping of ASIC designs and low-volume production, mostly because of its low speed, high cost per unit and high power consumption. However, thanks to the improvements of FPGA technology, soaring non-recurring engineering (NRE) cost and shortening time-to-market requirements, there is an increasing interest in using FPGAs instead of ASICS for embedded systems design [1]. Most applications running on FPGA-based systems are implemented using a single configuration per FPGA [10]. This means that the functionality of the circuit does not change while the application is running. Such an application can be referred to as being Compile-Time Reconfigurable or Static-Time Reconfigurable, because the entire
configuration is determined at the compile-time and does not change throughout system operation. Another implementation strategy is to implement an application with multiple configurations per FPGA [12], [11], [8]. In this scenario the application is divided into time exclusive operations that need not, or cannot, operate concurrently. Each operation is implemented as a distinct configuration which can be downloaded into the FPGA as necessary at run-time during application operation. This approach is referred to as Run-Time Reconfiguration, RTR or Dynamic Reconfiguration. Dynamic Reconfiguration can be achieved into two different ways: dynamic external reconfiguration and embedded reconfiguration. Dynamic external reconfiguration implies that an active array may be partially reconfigured by an external device such as a Personal Computer, while ensuring the correct operation of those active circuits that are not being changed. Embedded reconfiguration extends the concept of dynamic reconfigurability assuming that specific circuits on the array are used to control the reconfiguration of other parts of the FPGA. Clearly the integrity of the control circuits must be guaranteed during reconfiguration, so by definition embedded reconfiguration is a specialized form of dynamic reconfiguration [13]. As a programmable platform, a dynamically reconfigurable architecture only makes sense when it provides a better solution than other alternatives e.g., superscalar processor and DSP, in terms of performance, cost, power and development efforts. Considering ever-increasing performances of processors, a good design methodology is essential to the success of this approach. A new class of cores called run-time parameterisable (RTP) has been introduced in [9]. RTP cores allow a single core to be computed and customized at run-time. For example, an adder core can be produced, and then parameterized at run-time for different operand widths. An innovation of this approach consists in considering
the RTP cores as a specific example of a reconfigurable core, placed on the programmable device in a dynamic manner to respond to the changing computational demands of the application. The problem of this methodology is that the RTP are targeted only to a single device family and there is no information about the communication channel between RTP and about how they solve the physical reconfiguration problem.

There is lot of research being conducted on Design Methodology and Environment for Runtime Reconfiguration. The most common requirement for the set of tools and associated methodologies addressing the following issues are:

- Automatic or manual partitioning of a conventional design,
- Specification of the dynamic constraints,
- Verification of the dynamic implementation through dynamic simulations at major steps of the design flow,
- Automatic generation of the configuration controller core for HDL implementation,
- Dynamic floor planning management and guidelines for modular back-end implementation.

2.1.2 Design Partitioning

One of the most important tasks in Dynamic reconfiguration is Design Partitioning. There are basically two types of partitioning 1) Temporal and 2) Spatial partitioning. If a function is partitioned in to set of operations that are executed sequentially in time it is known as temporal partitioning. As the name suggests partitioning a function in space is called spatial partitioning. General purpose microprocessors provide a silicon medium that can be configured to solve any computation task and it is mostly a temporal computation or in other words serial computation. Reconfigurable devices compute a
function by configuring functional units and interconnecting them in space. This provides parallelism to computation. Superscalar and VLIW micro-processors exploit some low level parallelism, but not like the reconfigurable logic devices. Though reconfigurable logic provides parallelism in computing due to the lack of serial computation they exhibit scalability problems i.e. an application larger than the capacity of a reconfigurable computer can not be mapped without scaling the available hardware resources [22], but a judicious temporal partitioning can avoid an over sizing of the resources needed.

Run-Time reconfiguration tries to take advantage of the serialism and parallelism of a design by splitting larger designs into temporally exclusive collections, so-called configurations of smaller sub problems that are loaded onto FPGAs dynamically during the application’s run-time. The dynamically reconfigurable computing consists of successive execution of a sequence of algorithms on the same device. The objective is to swap different algorithms on the same hardware structure, by reconfiguring the FPGA array in hardware several times in a constrained time and with a defined partitioning and scheduling [42, 43]. So when we are taking about RTR we are usually concerned about both temporal and spatial partitioning of the design, i.e. a step that partitions design into time-exclusive spatial segments and groups these sub-functions or hardware objects into FPGA configurations [19, 20]. Several research efforts have been made to partition and map a computational task onto spatially interconnected reconfigurable processing elements [17] [18]. The objective is to exploit the parallelism in the application by mapping it to the spatially interconnected elements. Splash [17], PAM [18] and other computing environments have practically demonstrated the performance gains equivalent to super computers. One other important issue in run time reconfiguration are the control
of FPGA reconfiguration and the generation of communication channels between hardware objects in arbitrary configurations [21]. This is obtained by proper sequencing of the partitions. One common way of sequencing the configuration is the use of DFG. As we know an application consists of some potential for spatial computation and some restrictions that require temporal computation. If an application can be represented as a data flow graph (DFG) with sub-tasks as nodes and their dependencies as edges, all the nodes with same level in the graph constitute potential candidates for spatial computation i.e., they exhibit spatial flexibility. The nodes connected by edges should be executed sequentially i.e., temporal computation.

2.1.3 Design Issues and JBits

Most FPGA designs follow the traditional ASIC design flow, confining the reconfigurability to load time. However, run-time or dynamic reconfiguration is of special interest among the research community because it provides a performance/cost advantage over load-time configuration [2]. In the past years, there has been a little research addressing the design issues of dynamically reconfigurable architectures. Hauser et al. presented the Garp architecture [3] and its compiler [4]. The Garp architecture combines configurable hardware with a standard MIPS processor on the same die. The specially designed features allow hardware to be reconfigured in microseconds. The Garp compiler can find instruction-level parallelism (ILP) from C code, and directly compile selected loops to the reconfigurable array. However, if only ILP is to be exploited, the system is unlikely to outperform VLIW and superscalar processors regarding speed. Kaul et al. proposed a SPARCS framework [5]. In SPARCS, a high-level synthesis tool is employed to estimate resources and latency. An Integer Linear Programming (ILP) model
is formulated to solve spatial and temporal partitioning problems. This flow is complete and well-defined. But it is still based on traditional hardware design, and complexity is even increased due to dynamic reconfiguration. Hutchings et al. proposed a JHDL tool to handle run-time reconfiguration design [6]. It provides high-level language support to express dynamic reconfiguration and a dual simulation/execution environment. Nevertheless, JHDL suffers from a structural design approach, which makes it too low-level to be used in large designs. JBits is another tool that supports run time reconfiguration Xilinx’s JBits Software Development Kit (SDK) [23][24] gives system designers the ability to directly configure and reconfigure the Virtex™ family of FPGAs using standard Java software development tools. While JBits does provide access to the FPGA hardware at the lowest levels, it is possible to use JBits to design circuits at a higher level of abstraction using the Run-Time Parameterizable (RTP) Core library [26]. Recent enhancements to the RTP Core specification provide support for high-level abstractions for placement, routing and variable granularity. In addition, output of static net lists to formats such as EDIF is supported [25]

2.2 Stages in the JPEG2000 Algorithm

This section provides a very basic explanation of the internal stages of the JPEG2000 algorithm. The JPEG2000 algorithm will be presented from the point of view of the encoding process. Unless otherwise specified, the information in this section has been sourced from [28] and [29]. The JPEG2000 Final Draft Committee (FCD) divides the algorithm into six sections. This is shown in Figure 2.1.
An image input to JPEG2000 for compression first undergoes some basic preprocessing. Here the input image samples are level-shifted so that they have a "nominal dynamic range that is approximately centered about zero" [28]. A digital image can contain multiple components. For example, a color photo is often specified in terms of its red, green and blue parts. Each of these is considered a separate image component. JPEG2000 optionally allows for an inter-component transform to be applied to the image after it has been level-shifted. This transform helps de-correlate the separate components for multi-component images. No other part of the algorithm relates different components to one another. At this point in the process the image undergoes a Discrete Wavelet Transform (DWT). In contrast to the Discrete Cosine Transform (DCT), used by JPEG, the DWT is the source of a number of the superior features of JPEG2000. For example, the DWT deals with image discontinuities far better than the DCT [28]. As a result, JPEG2000 displays none of the artifacts that were present in JPEG versions of images with sharp discontinuities. Additionally, different variations of the DWT can be applied
depending on whether lossy or lossless compression is desired. The output of the DWT process is a set of transform coefficients. After the DWT has been applied, the transform coefficients undergo quantization. For lossy compression, quantization is one of the parts of the algorithm where information is lost [28]. Once the transform coefficients have been quantized, JPEG2000 specifies a coefficient encoding procedure, referred to here as “coefficient bit modeling”. The quantized values are arranged into rectangular arrays called code-blocks. The coefficient data in these code-blocks is considered to be a series of bit-planes. (A bit-plane refers to “all the bits of the same magnitude in all coefficients or samples” [29]). Each code-block is encoded one bit-plane at a time, with three consecutive coding passes being used per bit-plane. These three coding passes are called the ‘significance propagation’, ‘magnitude refinement’ and ‘cleanup’ passes respectively [29]. Each bit-plane encoding pass generates a series of output binary symbols. The JPEG2000 standard allows these sets of symbols to be passed through an arithmetic entropy encoder. This encoder compresses the symbols to further reduce the amount of data to be placed in the output file. The specific arithmetic encoder used in this stage is known as an “MQ Encoder”. The MQ encoder in JPEG2000 is also compatible with the arithmetic encoder used in the JBIG2 compression standard for bi-level (e.g. black and white) images [38]. After bit-plane and arithmetic encoding, the coding pass data is “packaged into data units called packets, in a process referred to as packetization” [39]. This data ordering stage of the algorithm arranges the final JPEG2000 compressed output, referred to as the “code stream”. The standard allows for different arrangement orders of packets within the code stream. The order in which packets are arranged affects the way in which the image can be progressively recovered. For example, one ordering
arrangement allows the compressed image to be progressively decoded with increasing fidelity. A different ordering arrangement allows progressive decoding with increasing resolution. The resulting code stream containing the JPEG2000 compressed version of the original input image can optionally be wrapped in the JP2 file format, also specified in the standard. The file format allows extra information about the image and its interpretation to be included with the data. For example, the code stream itself does not specify the color system used by a multiple-component image (e.g. RGB). That information can be specified in the JP2 file format.

2.3 MQ Encoder

2.3.1 Algorithm of Q coder

A new adaptive binary arithmetic coding system, the Q-coder, was developed in IBM research. It is characterized by multiplier free approximation, renormalization-driven probability estimation and bit stuffing. Coding conventions are used in optimal hardware and optimal software implementation. It also incorporates a new probability-estimation technique which provides an extremely simple and robust mechanism for adaptive estimation of probabilities during the coding process.

This section presents an algorithm and convention of Q coder. First, a discussion of the coding conventions leads to optimal hardware and software implementation. Second, Section 2.3.2 covers fixed precision operation, multiplier-free approximation, bit-stuff, and the estimation of probabilities by a new technique which uses only the interval renormalization which is a necessary part of finite-precision arithmetic coding process. Dynamic probability estimation makes the Q coder an adaptive binary arithmetic coder.
2.3.2 Coding convention of Q-coder

The basic structure of a compression/decompression system is shown in Figure 2.2. The compression process is divided into three basic parts: a model which converts uncompressed data into binary decisions, a probability estimator, and arithmetic coder. The dash boxes enclose the parts of the system comprising the Q coder. The model is outside the scope of this section. The model in the encoder uses the uncompressed data to determine the state S (used to determine where the probability estimate Qe for that state is stored) and YN, the binary (yes/no) decision that is to be encoded. These are the inputs to the Q coder. This model is called statistical probability modeling. The compressed data are output one byte at a time and may be transmitted to a decoder immediately or

![Diagram of Q-coder structure](image)

Figure 2.2: Basic structure of a generic Q-coder
stored for suture use. The Q decoder is similar to encoder.

The code string approaches the final code from below. It is initialized to be zero and always points to the base of the arithmetic-coding interval. This interval is subdivided into LPS and MPS subinterval. LPS is beneath MPS. The relative size of each code. Let A denote the present interval on the number line. Let C, the code string, subinterval is determined by the estimated LPS probability Qe and the estimated MPS probability Pe, which is equal to 1 – Qe. The following is a simplified pseudo point to the base of that interval.

The coding process for a single symbol is as follows:

```
Receive YN
If MPS is encoded
  C = C + Qe
  A = Pe = A – Qe
Else (LPS is encoded)
  A = Qe
End
If A < 0.75
  Renormalize A and C;
  Update Qe
End
```

Figure 2.3: Pseudo code of Q-coder
According to the pseudo code, it is observed that renormalization is driven when A register drops below 0.75. Thus, the normalized range is maintained in the interval 0.75 to 1.5. This keeps A centered around 1 so that the arithmetic approximations are reasonably good. For ease of implementation in hardware, it is suggested that the test for renormalization be done on the most significant bit of A. As mentioned above, most of the symbols are MPS sense and fewer symbols are LPS. But renormalization occurs following both the MPS (occasionally) and the LPS (always). Renormalization following the LPS always occurs since Qe, which is always smaller than 0.5 assigns to A. If Qe becomes smaller, the sequence of leading zero becomes longer. So the computation on LPS sense focuses on shift operation without interval subtraction. The algorithm of Q-coder is suitable for hardware implementation because the interval subtraction and code string addition can be done in parallel.
2.3.3 The Characteristic of Q-coder

- **Fixed-precision:**

  Arithmetic coders usually avoid the increasing-precision problem by using a fixed precision arithmetic. Implementation in fixed-precision arithmetic requires that a choice be made for the fixed-precision representation of the interval. So, a renormalization rule must be devised which maintains the interval size. Both the code string and interval size must be renormalized identically; else the identification of the code string as a pointer to the current interval will be lost. Efficiency of the hardware and software implementations suggests that renormalization be done by using a shift-left logical operation.

- **Multiplier-Free Approximation:**

  One final practical problem needs to be resolved. In general, arithmetic coding requires a multiply operation to scale the interval after each coding operation. Generally, multiplication is a costly operation in both hardware and software implementations. An early implementation of adaptive binary arithmetic coding avoids multiplication [13]. However the Q-coder uses an even simpler approximation to avoid the multiply. If renormalizations are used to keep the current interval, A, in the range $0.75 \leq A < 1.5$, the multiplications required to subdivide the interval can be approximated as follows:

  \[
  A \times Q_e \approx Q_e
  \]

  \[
  Ax Pe = A (1 - Q_e) \approx A - Qe
  \]

- **Probability Estimation:**

  Adaptive arithmetic coding requires that the probability be re-estimated periodically. Dynamic probability estimation is a very important concept, it was developed in earlier arithmetic coding implementations [42] [43]. The probability estimation technique used in
the Q-coder differs from the earlier techniques in a way the estimates are revised only
during the interval renormalization that is required in the arithmetic coder. Estimations
only at renormalizations are very important for efficient software implementations. The
inner loop of the coder is then minimized. Since each renormalization produces at least
one compressed-data bit, the instructions cycle spent on the estimation process are related
to the compressed-data code string length.

The dynamic probability estimation can be defined as a finite-state machine, that is, a
table of Q values and associated next states for each type of renormalization and MPS
exchanging flag for MPS and LPS reversing.

Figure 2.5: Probability Estimation

Figure 2.5 shows the actual finite-state machine used to estimate the probabilities.
The leftmost section illustrates the exchange of MPS and LPS definitions at \( Q_e \approx 0.5 \). As
the LPS goes from \( K_{ex} \) to \( K_{ex} - 1 \), the LPS and MPS senses are reversed, as indicated by
the asterisk. The center section shows region where the finite-state machine changes from
a single-state jump on LPS to a double-state jump. Some parts of the finite-state machine
require a jump of more than one state in order to balance the movement to larger or
smaller Qe indices following the renormalization. The rightmost section shows the
diagram for the smallest values of Qe.

Figure 2.6: An example of probability estimation for an LPS followed
by a sequence of MPS Symbols

Figure 2.6 illustrates the sequencing of the probability estimator for a LPS followed
by a sequence of MPSs. The ordinate shows interval (A-register) value, and the abscissa
shows the discrete allowed values of Qe. Solid lines indicate changes to the interval
resulting from coding operations: dashed lines represent changes resulting from
renormalizations. The initial LPS renormalization (marked with an asterisk) causes a
transition to a known A-register value and a known state in the finite-state machine. As
MPSs are coded, the interval (A-register) drops below 0.75. At that point a transition is
made towards a smaller Qe. And the interval is renormalized by doubling until it is
greater than 0.75. In most cases only one doubling is needed. But the pair of doublings shown at \( Qe = 0.32831 \) is the exception rather than the rule.

The \( Qe \) values for each \( Qe \) index are chosen to have mostly bit values of 0, except that the Tbit is always a 1. This strategy simplifies the hardware implementation in the sense that each bit value of 1 (except the last) requires additional wiring and circuits.

<table>
<thead>
<tr>
<th>( Qe )</th>
<th>( \Delta Qe )</th>
<th>LPS</th>
<th>MPS</th>
<th>Sense</th>
<th>( Qe )</th>
<th>( Qe ) Value</th>
<th>Inter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0463</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0646</td>
<td>0.0829</td>
<td></td>
</tr>
<tr>
<td>0.0906</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.1189</td>
<td>0.1472</td>
<td></td>
</tr>
<tr>
<td>0.1224</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.1561</td>
<td>0.1844</td>
<td></td>
</tr>
<tr>
<td>0.1702</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.1984</td>
<td>0.2266</td>
<td></td>
</tr>
<tr>
<td>0.1984</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.2266</td>
<td>0.2548</td>
<td></td>
</tr>
<tr>
<td>0.2266</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.2548</td>
<td>0.2830</td>
<td></td>
</tr>
<tr>
<td>0.2548</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.2830</td>
<td>0.3112</td>
<td></td>
</tr>
<tr>
<td>0.3112</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.3112</td>
<td>0.3394</td>
<td></td>
</tr>
<tr>
<td>0.3394</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.3394</td>
<td>0.3676</td>
<td></td>
</tr>
<tr>
<td>0.3676</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.3676</td>
<td>0.3958</td>
<td></td>
</tr>
<tr>
<td>0.3958</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.3958</td>
<td>0.4240</td>
<td></td>
</tr>
<tr>
<td>0.4240</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.4240</td>
<td>0.4523</td>
<td></td>
</tr>
<tr>
<td>0.4523</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.4523</td>
<td>0.4805</td>
<td></td>
</tr>
<tr>
<td>0.4805</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.4805</td>
<td>0.5087</td>
<td></td>
</tr>
<tr>
<td>0.5087</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.5087</td>
<td>0.5369</td>
<td></td>
</tr>
<tr>
<td>0.5369</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.5369</td>
<td>0.5651</td>
<td></td>
</tr>
<tr>
<td>0.5651</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.5651</td>
<td>0.5933</td>
<td></td>
</tr>
<tr>
<td>0.5933</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.5933</td>
<td>0.6215</td>
<td></td>
</tr>
<tr>
<td>0.6215</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.6215</td>
<td>0.6497</td>
<td></td>
</tr>
<tr>
<td>0.6497</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.6497</td>
<td>0.6779</td>
<td></td>
</tr>
<tr>
<td>0.6779</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.6779</td>
<td>0.7061</td>
<td></td>
</tr>
<tr>
<td>0.7061</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.7061</td>
<td>0.7343</td>
<td></td>
</tr>
<tr>
<td>0.7343</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.7343</td>
<td>0.7625</td>
<td></td>
</tr>
<tr>
<td>0.7625</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.7625</td>
<td>0.7907</td>
<td></td>
</tr>
<tr>
<td>0.7907</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.7907</td>
<td>0.8189</td>
<td></td>
</tr>
<tr>
<td>0.8189</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.8189</td>
<td>0.8471</td>
<td></td>
</tr>
<tr>
<td>0.8471</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.8471</td>
<td>0.8753</td>
<td></td>
</tr>
<tr>
<td>0.8753</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.8753</td>
<td>0.9035</td>
<td></td>
</tr>
<tr>
<td>0.9035</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.9035</td>
<td>0.9317</td>
<td></td>
</tr>
<tr>
<td>0.9317</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.9317</td>
<td>0.9600</td>
<td></td>
</tr>
<tr>
<td>0.9600</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.9600</td>
<td>0.9882</td>
<td></td>
</tr>
<tr>
<td>0.9882</td>
<td>0.0283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.9882</td>
<td>0.9999</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.7: Probability Estimation Table

Figure 2.7 gives the information necessary to perform the probability estimation. For each \( Qe \) value in the table, a LPS sense causes the \( Qe \) index to be decremented by the amount indicated in the “Deer LPS” column. MPS renormalizations increment the \( Qe \) index by 1 unless the \( Qe \) index had reached the maximum value, as shown in the “Inter
MPS” column. An LPS from the top entry causes an exchange in the sense of the MPS and LPS, as indicated by the “MPSexch flag”. The Qe values as binary bits are followed by the decimal form. The decimal fraction 0.75 as chosen to correspond to X’1000’: this determines the scaling in converting the first column to the last column.

- Bit stuffing in the Q coder

Another problem to be resolved in the fixed-precision arithmetic is a carry propagation problem. It appears on binary arithmetic coding. It is possible to generate a code string with a consecutive sequence of 1-bit of arbitrary length. If a bit is added to the least significant bit of this sequence, a carry will propagate until a 0 is encountered. Langdom and Rissanen [42] resolved this problem by “bit stuffing”. If a sequence of 1-bit of a predefined maximum length is detected, an extra 0-bit is stuffed into the code string. The stuffed 0-bit acts as a carry trap for any carry from future coding operations. The decoder, after detecting this same sequence, removes the stuffed bit, adding any carry contained in it to the code string remainder. The Q-coder follows this general scheme, but with the additional constraints that the string of 1-bits is eight bits in length and is byte-aligned.

The general principle of bit stuffing is reviewed in [42]. Carry propagation can only occur through a sequence consecutive of 1-bits. In principle, the sequence of 1-bits can as long as the code strings itself. To avoid arbitrary propagation of the carry, sequences of 1s are interrupted periodically by inserting or stuffing 0-bits. In the Q-coder the bit stuffing is done only on byte boundaries. One zero is stuffed into the high order bit of the byte immediately following any byte which is all 1s(X’FF’), this 0-bit is the receiver for any carry which might subsequently occur. The diagram below shows the 12 “x” bits to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
the right of the binary point aligned with the fractional bits in A, the four “s” spacer bits, and the 8 “b” bits which will output as a byte when the flag bit which starts in the bit position 24 is shifted out of the register. A carry, which will sometimes be shifted into bit 24, is added to preceding byte. If the 8 bits which will be output are X ‘FF’, the stuffed bit which is inserted is following the high order seven “b” bits to prevent the carry bit from next compressed data.
3.1 VERILOG Language

The implementation of the JPEG2000 arithmetic encoder was written in VERILOG. The code written was required to conform to the JPEG2000 standard for the arithmetic entropy encoding stage. Verilog HDL is a Hardware Description Language (HDL). Verilog is one of the major Hardware Description Languages (HDL) used by hardware designers in industry and academia. The Verilog language provides the digital designer with a means of describing a digital system at a wide range of levels of abstraction, and at the same time, provides access to computer-aided design tools to aid in the design process at these levels. One of the main reasons for the popularity of Verilog is that it is very similar to computer language C which make it easy for engineers to understand since they usually have prior experience working with C. VERILOG is widely used for creating designs that can be programmed into FPGA devices. It is a general and versatile language that can be used to design a digital system and is widely compatible with industry tools. Using VERILOG allows as much of the design as possible to be portable to other synthesis tools, other FPGA cards or even other FPGA architectures. Previous experience had already been obtained in the use of VERILOG for programming FPGA
devices. This experience was of invaluable help for the implementation section of this project.

3.2 Memec Insight Virtex-II MB

The Memec Insight Virtex-II MB Development Kit provides a complete solution for developing designs and applications based on the Xilinx® Virtex-II FPGA family. The Virtex-II MB system board utilizes more than one-million gates Xilinx Virtex-II device (XC2V1000-4FG456C) in the 456 fine-pitch ball grid array package. The high gate density and large number of user I/Os allows complete system solutions to be implemented in the high performance FPGA. The system board includes a 2 M x 16 Double Data Rate (DDR) memory, two clock sources, RS-232 port, and additional user support circuits. A Low Voltage Differential Signaling (LVDS) interface is provided with 16-bit transmit and 16-bit receive ports, plus clock, status and control signals for each. The Virtex-II FPGA family has the advanced features needed to fit the most demanding, high performance applications. The Memec Design Virtex-II MB Development Kit provides an excellent platform to explore these features. The board provides an ideal platform for reconfigurable computing applications with its usage of high-density FPGA devices that provide a template for reprogrammable logic circuits. It also provides a unique test platform since a host computer can interact with the Programmable Elements through onboard memory. One important reason for selecting this board is that JBits 3.0 is compatible with xilinx virtex-II FPGAs and can be used to program the XC2V1000-4FG456C chip. The XHWIF interface which comes with the
JBits 3.0 SDK provided by xilinx supports the XC2V1000 chip and can be used to communicate with a FPGA board.

Figure 3.1: Block diagram of Memec Insight Virtex-II MB Development Kit Board

3.3 Xilinx ISE 5.0

Synthesis is one of the most essential steps in the design methodology. It takes the conceptual Hardware Description Language (HDL) design definition and generates the logical or physical representation for the targeted silicon device. Design implementation is the process of translating, mapping; placing, routing and generating a BIT file for the design.
Synthesis tools are required to produce highly optimized results with a fast compile and quick turnaround time. To meet this requirement, the synthesis engine needs to be tightly integrated with the physical implementation tool and have the ability to proactively meet the design timing requirements by driving the placement within the physical device. In addition, cross probing between the physical design report and the HDL design code will further enhance the turnaround time. Xilinx’s Integrated Simulation Environment (ISE) provides support for today’s most popular methods of design capture including HDL and schematic entry. ISE even allows designs that contain a mixture of VHDL and Verilog, integration of IP cores as well as robust support for reuse of your own IP. The complex design process is streamlined and completed in less time, reducing the overall project costs, with easy-to-use graphical interfaces and powerful problem-solving technology such as Architecture Wizards, PACE floor planning, Project Navigator and more. With these rich features, plus the mixture of Design Entry capabilities, ISE provides the easy to use programmable logic design tools available today for logic design.

Once synthesis is done configuring the programmable logic device is the last step in the design methodology. A bit stream is generated from the physical place and route information and is transferred through cables to the target device. IMPACT, included with all ISE configurations, is a robust configuration tool that automatically takes care of everything from bit stream generation to the device download. A single interface provides support for both parallel port JTAG cables, and USB and RS-232 serial port MultiLINX cables. In this thesis we use Xilinx ISE 5 to synthesize and configure the
FPGA. There is no specific reason for choosing this even though availability of the tool and its compatibility with the Virtex – II FPGA played a significant role in its selection.

3.4 JBits 3.0

The JBits API is a set of Java classes that provide an Application Program Interface (API) into the Xilinx Virtex II FPGA family bit stream. This API may be used to construct digital designs and parameterisable cores that can be executed on Xilinx Virtex II FPGA devices. The API provides the lowest level interface to the Virtex II architecture and thus it can also serve as a base to construct traditional circuit placement and routing, as well as application specific tools to perform more narrowly defined tasks. Figure 3.1 shows a component view of the JBits environment. The following sections discuss each component in greater detail.

![Figure 3.2: JBits Environment](image)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
JBits operates on either bit streams generated by Xilinx ISE design tools, or on bit streams read back from actual hardware. This provides the capability of designing, modifying and dynamically modifying circuits for Xilinx Virtex II series FPGA devices. This capability is achieved by providing access to all the resources of a Virtex II device. The API gives access to the Look up Tables (LUT) inside a Configurable Logic Block (CLB) and to the routing resources of the Virtex II device. The device architecture is represented as a two-dimensional array of Configurable Logic Blocks (CLB) in JBits. Each CLB is referenced by a row and column, and all configurable resources in the selected CLB may be set or probed. Control of all routing resources adjacent to the selected CLB is made available in the API. JBits code is written in Java. JBits API provides a High level language approach to develop reconfigurable systems including runtime reconfiguration. Though JBits API is an enabling technology for RTR, the API can also be used to produce traditional static design bit stream files for Virtex II FPGAs. The design flow followed is different from that of the traditional CAD tools. Unlike the conventional design flow, that uses HDL or schematic design entry, a design is specified in a Java program using the JBits API. The application takes a bit stream file as an input and extracts the device configuration data. This data is modified according to the design specified using the JBits API and is output to a bit stream file that will be used to configure the Virtex II hardware. Once downloaded to the Virtex II hardware, the design can be debugged using the XHWIF interface. The execution model for this flow is illustrated below.
The real power of JBits is its use in the development of Java Run Time Reconfiguration (RTR) applications. In this flow, the circuits can be configured on the fly by executing a Java application that communicates with the circuit board containing the Virtex II device. This is made possible by using the JBits to specify the design and use the XHWIF API to download the design within the same Java application. This design flow is illustrated below. Again, this bit stream input to the Java Application can be a null bit stream or a bit stream for an existing design.
The execution model for this flow is illustrated below. In this case, the host computer executing the JBits application uses the XHWIF interface API to communicate with the Virtex II reconfigurable hardware. For example, in a typical PC computing environment, the host microprocessor executes the JBits application and configures the Virtex II reconfigurable hardware located in the PCI slot or connected by a JTAG cable, using the XHWIF API. This enables run-time configuration and reconfiguration of the Virtex II device.

3.4.1 Accessing the CLB resources:

A resource is defined as the configurable element inside FPGA. This may include the CLB inputs (F LUT inputs, G LUT inputs, BX, BY...), Routing (singles, Hexes,...), Clocks, etc. The JBits API gives the user the ability to configure CLB directly. The CLB functional elements are in classes in the Java package which is given below.
For example, consider the Java class `com.xilinx.JBits.Wires.Center.BX0`. This class abstracts the Slice 0 BX input of CLB.

Consider the following code: Let's select Slice 0 BY to be Slice 0 BX input of CLB (0,0).

```java

JBits jbits = new JBits(Device.getDevice(deviceType));
jbits.read("myBitstream.bit"); int clbRow = 0; // Row 0
int clbCol = 0; // Column 0

jbits.setCLBBits( clbRow, clbCol, BX0.BX0, BX0.BY0 );
```
LUTs in CLBs can be used to implement logic functions. In JBits, Virtex CLB LUTs are defined in the \texttt{com.xilinx.JBits.Virtex.Bits.Logic.Center.LUT} class. The configuration data can be read from and written to the LUTs using the \texttt{set CLB Bits()} and \texttt{get CLB Bits()} methods respectively. The CONTENTS field can be considered a 2D array indexed first by the required slice (0, 1, 2, 3) and then by the F or G constants defined in the class. Thus to access the F LUT of slice 1, you would use \texttt{LUT CONTENTS[1][LUT.F]}.

3.4.2 JBits Wire Database:

JBits has a set of classes that provide an API for obtaining connectivity information. The information about the connectivity is collectively known as the JBits wire database. These classes contain information about connectivity within a tile, between tiles, and bits associated with each connection. Full support for every tile and resource in the Virtex-II FPGA is available.

The JBits wire database represents the connectivity of a XilinxFPGA with Java objects. The database is represented in a device generic manner and can be used for any member of the Virtex-II family of FPGAs. Information can be obtained about sources, sinks, and bit stream settings using the API. The database uses a tile based approach which uses one object to tell the connectivity of a single tile type (intra-tile routing graph) and a method to create the connectivity between tiles (inter-tile routing graph). This leads to a single representation of any device in a family of FPGAs. Some important definitions are:

\textit{Wire} – A template of a physical wire in a tile. It includes multiplexer input and output connectivity information (intra-tile routing graph). Also includes information about the
location of bits in the bit stream that are set to modify the connectivity.

*Pin* – An instantiation of a Wire template that is specific to a tile in the device.

*Segment* – A collection of Pins that are directly connected together across tiles (inter-tile routing graph). Some important packages of the wire database are

package *com.xilinx.JBits.ArchIndependent*: This package includes architecture independent implementations of the following classes: Wire, Segment, Pin, Lookup table.

package *com.xilinx.JBits.Virtex2*: This package contains the architecture specific classes that provide a main entry into the database.

### 3.4.3 XHWIF:

The XHWIF API provides various methods to describe an FPGA-based board and to send data on and off the board. It includes methods for reading and writing bit streams to FPGAs, and methods for describing the kind and number of FPGAs on the board. Also included are methods for incrementing the on-board clock, and for reading from and writing to on-board memories, if they are available. The interface standardizes the way the applications communicate with hardware, so that using the same interface can communicate with a variety of boards. All of the hardware specific information is hidden inside of a class that implements the *XHWIF* interface. Using the Java programming language's *Native Methods*, XHWIF can communicate with hardware directly, through libraries or through a device driver. The advantage of this approach is that new hardware can be quickly and easily supported. All that is required is that an *XHWIF* interface for the new hardware be implemented. Once this interface exists, all software which uses the *XHWIF* interface will run on the new hardware, without modification and without re-compilation. The XHWIF interface can be used to communicate with a FPGA board,
once the XHWEF API is ported to that particular board. Once the API is ported to a particular board various tools and applications that use the XHWEF API to perform various hardware board operations can run on the newly supported piece of hardware without any re-compilation. Finally, XHWEF package also provides a client/server program. This can be used to connect to hardware’s placed remotely on a network. However, application that uses the XHWEF API can run on a local and remote hardware without modifications. The XHWEF API provides a generic set of methods to interface with FPGA hardware. These methods calls can be used to communicate with a board once the board interface is obtained. The first and foremost step in communicating with a board using the XHWEF API is to obtain the interface specific to that board. The XHWEF.Get() method is used for this purpose. The example code shows how-to obtain the board interface for a fictitious hardware named, "MyBoard". The board name can be passed as a command line parameter but it is hard coded here for illustration purposes. Only after connecting to the board, other operations are performed.

```
String boardName = "MyBoard";
XHWIF board = XHWIF.Get(boardName);

/* Get the remote host name (if we have one) */
String remoteHostName = XHWIF.GetRemoteHostName(boardName);

/* Get the remote port number (if we have one) */
int port = XHWIF.GetPort(boardName);

/* Connect to the hardware */
result = board.connect(remoteHostName, port);
```
Once connected, information about the type and number of devices on the board can be obtained. This information can be used to instantiate an appropriate JBits object and read in the bit stream to be downloaded on to the board. Once connected XHWIF provides methods to Resetting and configuring the FPGA device, clocking the Design, Reading back the Bit stream accessing the on-board memory.

3.4.4 Board scope

Board scope is a graphical FPGA debugging environment that operates at the bit stream level. Communication with hardware takes place using an underlining XHWIF layer. This allows BoardScope to use any hardware platform that has been ported to XHWIF. The BoardScope environment displays FPGA read-back information in a graphical context. This information includes CLB flip-flop states, LUT configurations, BRAM data, and IOB register states. The debugging process is started with a connection to a supported FPGA hardware platform, either locally, or remotely using a network connection. After connecting to desired hardware, bit streams can be loaded on the device. Debugging in BoardScope allows the user to switch between hardware platforms; Figure 3.4 provides a screen shot of the BoardScope debugging environment, along with captions that explain features. The debugging environment features different graphical views in which the operation of the FPGA hardware is shown in different contexts. Possible views include State, Core, Power, and Routing Density. As an example, the State view provides a graphical display of read back state information. The main grid representing the CLB layout shows the state information of all four flip-flops within a single grid square. Clicking on a CLB grid square causes the look-up table configuration
to be displayed in the graphical CLB viewer. These views create a robust debugging environment for RTR applications.

Fig. 3.6: Snapshot of the BoardScope tool
CHAPTER 4

METHODOLOGY

4.1 VERILOG Implementation

This section discusses the implementation of the arithmetic entropy encoder in the JPEG2000 standard in VERILOG. Once written in VERILOG, the encoder design can be synthesized and programmed into the FPGA on the Memec Insight Virtex-II MB board. The background issues related to the implementation are discussed first. Following this, the design methodology for the VERILOG code is presented. The implementation process involves several steps. These included an examination of both the JPEG2000 standard, the design of the encoder data path and the design of an overall controller for the encoder.

4.1.1 Arithmetic Entropy Encoder

As outlined in Section 2.2, the coefficient bit modelling stage of JPEG2000 compression involves encoding the quantized coefficients produced by the wavelet transform. Coefficient bit modelling involves a series of ‘passes’ for each code block bit-plane – the significance, refinement and cleanup passes. Each of these produces a series of output binary symbols. It is these symbols which can then be compressed further by passing them to the arithmetic entropy encoder (also just referred to as the arithmetic encoder). The standard additionally allows an arithmetic-coding
bypass mode (also known as 'lazy' mode [28]), whereby the arithmetic encoding stage is skipped for some coding passes [28] [29]. The bypass mode can be used to reduce the computational complexity of the compression process [28]. This feature of the standard reinforces the profile results which show that the arithmetic encoding stage is computationally expensive and therefore well-suited to hardware processing. The specific type of arithmetic encoder used in the JPEG2000 standard is known as ‘MQ coder’ [28].

4.1.2 Examination of the JPEG2000 Specification

The first step in the development process was to analyze thoroughly the JPEG2000 specification for the arithmetic entropy encoder. Analyzing the standard led to a good knowledge of the internal workings of the arithmetic encoder as well as to initial ideas about how a hardware implementation could be designed. The JPEG2000 standard explains basic data inputs and outputs of the encoder, as shown in Table 4.1. This table gives a good description of the information passed to and from the encoder. The specific binary inputs and outputs used to transmit the information will be covered later.

4.1.3 Data path Design

After examining JPEG 2000 encoder implementation, work on the VERILOG implementation began. Initial task for an implementation was to design of the data path for the encoder. Firstly, the key registers in the data path were identified. Nearly all of these registers corresponded to variables in the JPEG2000 specification. These main registers are shown in Table 4.3.
Table 4.1: Arithmetic Encoder Main Data Inputs/Outputs

<table>
<thead>
<tr>
<th>Data Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Bit</td>
<td>The symbols produced by the coefficient bit modeling stage are encoded one bit at a time.</td>
</tr>
<tr>
<td>Context</td>
<td>Extra information passed in addition to the actual input bits which is required for compression is known a ‘context’. A context can be specified with two pieces of information:</td>
</tr>
<tr>
<td></td>
<td><em>An Index.</em> This index points to an entry in a large look-up table of constants used in the compression process. This table is Table C-2 in the JPEG2000 FCD [29], also referred to in the VERILOG code as the ‘Qe-value table’.</td>
</tr>
<tr>
<td></td>
<td><em>A More-Probable Symbol (MPS).</em> At any stage, the encoder considers one of the binary digits 0 or 1 to be the more probable ‘symbol’ to be encoded. The complementary binary symbol (0 or 1) is consequently the Less-Probable Symbol (LPS).</td>
</tr>
<tr>
<td>Output Byte</td>
<td>Stream of compressed bytes of data produced by the entropy encoder at its output.</td>
</tr>
</tbody>
</table>

Table 4.2: Arithmetic Encoder Tasks

<table>
<thead>
<tr>
<th>Encoder Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Clears the internal state of the encoder so that it is ready to start encoding bits.</td>
</tr>
<tr>
<td>Set context</td>
<td>Sets the current encoder context that will be used to encode subsequent bits. The current context is part of the encoder’s internal state.</td>
</tr>
<tr>
<td>Encode bits</td>
<td>Bit Encode the next bit of input data. An Encode Bit task is usually, but not always, preceded by a Set Context task. A consequence of the Encode Bit task may be that the encoder outputs another byte of compressed data. However, this generally happens only after multiple Encode Bit tasks.</td>
</tr>
<tr>
<td>Flush</td>
<td>Signals the end of a stream of input bits. The Flush task generally causes the encoder to output multiple concluding bytes of compressed data. JPEG2000 supports two variations of termination for the Flush task.</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The arithmetic encoding algorithm often involves operating on these data registers and placing the results back into the same registers. All the arithmetic operations to be performed on the registers are listed after completely examining the algorithm. Some registers, such as the K and MPS registers, could be updated by only a few different operations. Others, in particular the C register, could be updated by any one of a substantial set of arithmetic calculations. Once registers and the operations which acted on them had been identified, the VERILOG code for the data path was written to implement this computational functionality.

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>32</td>
<td>32 Code register. Compressed data bytes are initially generated in certain bits of this register.</td>
</tr>
<tr>
<td>A</td>
<td>32</td>
<td>Interval registers for arithmetic encoding.</td>
</tr>
<tr>
<td>CT</td>
<td>5</td>
<td>Count register. Used to count the number of times A and C registers a bit-shifted under certain conditions.</td>
</tr>
<tr>
<td>MPS</td>
<td>1</td>
<td>Current More Probable Symbol</td>
</tr>
<tr>
<td>Index</td>
<td>6</td>
<td>Current index into look-up table.</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>Output byte buffer. The next compressed byte to be output comes from this register.</td>
</tr>
<tr>
<td>Temp C</td>
<td>32</td>
<td>Temporary C register.</td>
</tr>
<tr>
<td>K</td>
<td>5</td>
<td>Used for the predictable termination variant of the Flush task.</td>
</tr>
</tbody>
</table>

Control signals were created to select the operation that was performed on a given register on the next rising clock edge. During synthesis, large multiplexers would be generated at the inputs of registers that could be updated by a great number of different arithmetic operations. As the multiplexer size increases, so does its propagation delay. Of all the registers in the encoder, the C register is associated with the largest number of operations. As mentioned briefly in Section 2.3, the arithmetic entropy encoder also
requires the use of a large look-up table containing constants. These constants relate to a probability estimation process that the encoder performs [29], any change in the look-up table results in change of compression ratio. A grey scale image can be compressed more efficiently if we used a different look-table since we are dealing only with 256 shades of grey. At this point it is made clear that there were two different arithmetic encoder design created verilog_file_1.v and verilog_file_2.v. These tow file differ only in their probability look up table.

4.1.4 Designing the Controller

Once the data path was in place, the controller for the VERILOG arithmetic entropy encoder was designed. The controller was responsible for translating the four encoder tasks in Table 4.2 into a series of operations on the data path registers. The JPEG2000 specification clearly states all the operations that must be performed on each registers in order to produce a correct compressed data. The JPEG2000 standard makes heavy use of flowcharts to specify this internal behavior. A significant advantage of the hardware implementation over the software implementation is the ability of the hardware to perform parallel processing. Operating on two or more registers in the same clock cycle allows faster performance to be achieved than if only one register can be modified at a time. Designing the controller involved analyzing the flowcharts in the standard to determine which operations could be performed in parallel and which could be safely executed only in sequence. Additionally, it was required that the VERILOG design be written as synthesizable code. The primary requirement of the VERILOG implementation was one of the compliance. A synthesis tool such as Xilinx ISE had to be able to compile the VERILOG arithmetic encoder so that Virtex-II FPGA could be programmed with the
design. Once the code for the arithmetic encoder was written, a test-bench was written (also in the VERILOG language) to test the VERILOG encoder with test data. The test-bench was simulated in using Aldec’s Active-HDL tool. This test was passed successfully. The output produced by the VERILOG implementation matched exactly the expected output stream of bytes.

Table 4.4: Outputs of the Arithmetic Encoder VERILOG Entity

<table>
<thead>
<tr>
<th>Output</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boutreg</td>
<td>8</td>
<td>Bytes of compressed data are output from this port.</td>
</tr>
<tr>
<td>encrd</td>
<td>1</td>
<td>Equals ‘1’ when the arithmetic encoder is idle and ready to perform another task.</td>
</tr>
<tr>
<td>bready</td>
<td>1</td>
<td>Usually ‘0’, this signal is set to ‘1’ for one clock cycle when a new byte of compressed data appears on boutreg.</td>
</tr>
<tr>
<td>cout</td>
<td>32</td>
<td>These five signals are provided as outputs in order that the internal state of the arithmetic encoder can be observed at any point in time. These outputs are directly connected to the associated registers in the encoder data path.</td>
</tr>
<tr>
<td>aout</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>ctout</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>mpsout</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Indout</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

4.2 Synthesis and Power Estimation of the Design

The Xilinx Integrated Software Environment (ISE) 5 along with Xilinx Power (XPower) is used for design synthesis, configuration and estimation of power. The Xilinx ISE 5 uses the steps shown in Figure 4.1 to generate the files required for the XPower to compute power. Once tested for its correctness the code had to be synthesized. Xilinx’s ISE 5.0 synthesis and implementation tool was used to synthesize the code. During synthesis, we create a new project and specify the device as Virtex II xc2v1000 with package as FG456 where FG stands for the package type Fine Grid and 456 stands for the number of package pins.
The first step is design entry; here we include the Verilog code describing the encoder. Once the design has been entered it is synthesized, during which the Verilog code is converted to a net list file which is passed on as input to the next step. After synthesis, you run design implementation, which converts the logical design into a physical file format that can be downloaded to the selected target device. Implementation process consists of three steps Translating, Mapping, Placing and Routing. The Translate process merges all of the input net lists and design constraints and outputs a Xilinx native generic database (NGD) file, which describes the logical design reduced to Xilinx primitives. The Map process, maps the logic defined by an NGD file into FPGA elements, such as CLBs and IOBs. The output design is a native circuit description (NCD) file that physically represents the design mapped to the components in the Xilinx FPGA. An ASCII Physical Constraints File (PCF) is also produced in MAP. The PCF contains timing constraints that are used by power estimation tool XPower to identify clock nets. It also provides temperature and voltage information, if these constraints have been set in the User Constraints File (UCF). The Place and Route process takes a mapped NCD file, places and routes the design, and produces an NCD file that is used as input for bit stream generation. The next step is Generate Programming File process which produces a bit stream for Xilinx device configuration.
After the design is completely routed, you must configure the device so it can execute the desired function. This bit stream is used to configure the FPGA using IMPACT. The design was verified for syntax error and the post implementation verification was done using floor plan. The design passed all the verifications. Once the design has been synthesized and implemented we calculate the power using the Xilinx’s Xpower tool.

The timing simulations done in Active-HDL are verified using the waveform editor. The waveforms are exported as Value Change Dump (VCD) files. In the last step, the XPower uses the files generated in the previous steps to estimate power consumed by the design. XPower calculates the power in the design by summing up the power consumed
by each element. The power consumed by each switching element in the design is given by equation

\[ P = C \cdot V^2 \cdot E \cdot F, \] (4.1)

where \( P \) represents the power in mW, \( C \) represents the capacitance in Farads, \( V \) represents the voltage in Volts, \( E \) represents the switching activity (average number of transitions per clock cycle), and \( F \) represents the frequency in Hz.

The capacitance is determined from the NCD file. The voltage is a fixed value for a specific device and it is set by default in the XPower interface. \( F \cdot E \) gives the activity rate of the signal in the design. The activity rate is defined as the rate at which a net or logic element switches. For dynamic power calculation activity rates are expressed as frequency, which is the most variable element in Equation 4.1. The activity rates are set in the VCD file generated from the timing simulation in Active-HDL.

4.3 JBits and XHWIF

Once we have a working model of the arithmetic encoder synthesized and pushed on to an FPGA. JBits was used to reconfigure the FPGA dynamically. With not much documentation available on the latest version of JBits we had to work hard on the API to understand its utilities. The first and major step in using JBits is the porting of Xilinx Hardware Interface XHWIF on to the board. Most challenging aspect of porting to new hardware in the XHWIF environment is the use of the Java programming language. The current method used by Java to provide support for non-Java implementations is called the Native method. The Java Native Method is simply a function defined in a Java program, which is tagged with the reserved keyword native. This means that no Java code
will be supplied for this function. Instead, Java will search some user-supplied library for the code, which implements this "native" function. This approach is the technique used by Java to interface to other languages such as C, which in turn is used to interface to device drivers, xc2v1000 hardware or other non-Java software.

Porting XHWIF can be basically divided in two sections, the first section deals with \textit{board description} code, which tells \textit{XHWIF} which FPGA devices and packages are used by the hardware and the second part deals with Java Native methods and the library creations. So as said above we created a \textit{xc2v1000.java} a board specific Java file. This file contains the basic information about the board; the FPGA device on the board and their package type of the FPGA in our case it was FG456 and their description. Once \textit{xc2v1000.java} is written it is compiled using the \textit{Java xc2v1000.java} command to get the class file. This class file is stored in the JBits3.0/com/xilinx/XHWIF/Boards directory.

Once these constants are defined, you may compile this file to produce a Java \textit{class} file. Note that your compiler may require that other \textit{XHWIF} classes. See your compiler documentation if it complains about missing classes. Once you have produced this class file, it should be moved to the \textit{XHWIF/Boards} directory.

Once the board has been described and the \textit{XHWIF} interface compiled into a class file, the Java Native Methods must be implemented. These native methods are the low-level and system dependent functions which provide access to the hardware. The following native methods must be implemented to provide full \textit{XHWIF} support:

\begin{verbatim}
public native int xc2v1000Connect();
public native int xc2v1000Disconnect();
public native int xc2v1000GetSystemInfo(int data[], int length);
\end{verbatim}
public native int xc2v1000Reset();

public native int xc2v1000SetClockFrequency(float frequency);

public native int xc2v1000ClockOn();

public native int xc2v1000ClockOff();

public native int xc2v1000ClockStep(int count);

public native int xc2v1000GetConfiguration(int device, byte data[], int length);

public native int xc2v1000SetConfiguration(int device, byte data[], int length);

public native int xc2v1000GetRAM(int address, byte data[], int length);

public native int xc2v1000SetRAM(int address, byte data[], int length);

Using Java xc2v1000 command, we produce xc2v1000.h "C" header files, which describe the "C" functions that are called by the Java Native Methods. Once these header files are produced, we wrote xc2v1000.c a "C" code to implement these functions. The actual code to implement the hardware functionality will be placed in a single "C" file, which will have "generic" functions which will be referenced by the JNI. Files generated:

- `xc2v1000.h` The xc2v1000.h file is simply a small file containing the function prototypes for the xc2v1000.c file
- `xc2v1000.c` This file contains all of the functions which must be implemented to produce the "C" library
- `xc2v1000JNI.h` This is the header file that is automatically generated by javah utility from the Sun JDK.
- `xc2v1000JNI.c` This provides the bridge between the automatically generated interface and the more generic functions in xc2v1000.c This file can easily be produced by taking the function prototypes for xc2v1000JNI.h
All the files are copied on to the com/xilinx/XHWIF/Boards directory where they all are complied to produce the libraries required by the Java Native Methods. The xc2v1000.dll library file produced is copied and place in the java 1.4.2/bin directory so that it can be accessed by any JBits program related to the board. Therefore the four important files creating a subclass of the com.xilinx.XHWIF package. This is a subclass that describes the details of the new hardware platform. In the examples below, this class will be called "xc2v1000". Compile this interface, called xc2v1000.java into an xc2v1000.class file and place this class file in the "XHWIF/Boards/" directory. Produce the Java Native Method interface for the xc2v1000 class by running Sun's Javah. Implement the Java Native Method interfaces in "C" and compile into a library / libraries. Place these libraries in an appropriate directory (typically one in the search path). Once the XHWIF was ported a code was written using JBits API to read the bit stream from the board and load a new modified bit stream on the board. During the load stage we make sure that the clock has been stopped. We can also load the new bit streams using BoardScope.
As mentioned earlier, the arithmetic entropy encoder used in JPEG2000 is known as an ‘MQ encoder’ and is also used in the JBIG2 compression standard. While the JPEG2000 Final Committee Draft (FCD) [29] contains no test data for the arithmetic encoder, such data is found in the JBIG2 Final Committee Draft [29]. As the arithmetic encoders used in JPEG2000 and JBIG2 are compatible, the test data in the JBIG2 FCD can be used to test the VERILOG encoder developed in this project. The test data sequence from JBIG2 consists of a series of input bits that are fed to the encoder, as well as the sequence of output bytes that a correct implementation should produce. One small difference between the arithmetic encoder specifications in the two standards should be noted. Since the ‘Flush’ task in the JBIG2 FCD results in the constant bytes 0xFF and 0xAC being output at the conclusion of the ‘Flush’ procedure and the JPEG2000 specification does not result in these concluding two bytes being produced. Consequently, when testing the VERILOG encoder implementation the sequence of expected output from the JBIG2 FCD was adjusted to remove the trailing 0xFF and 0xAC bytes. This was the only modification made to the test data. A test-bench was written (also in the VERILOG language) to test the VERILOG encoder with the JBIG2 test data. The test-bench was simulated in the Active-HDL programmed. This test was
passed successfully. The output produced by the VERILOG implementation matched exactly the expected output stream of bytes obtained from the JBIG2 FCD. Both the codes have been tested and the simulation waveforms are near same.

Figure 5.1: Simulation waveforms of the verilog_file_1.v code

Once the codes were tested and simulated we exported the .vcd file of for power calculation. The Verilog implementations of the arithmetic encoder were synthesized successfully using the Xilinx ISE 5 software. The place and route procedure was also performed, in order to generate timing information. For place and route, the FPGA targeted was a Virtex II xc2v1000 device of speed grade -5, as found on the board. The
maximum clock frequency of the Verilog_file_1 encoder was reported by Xilinx ISE 5 to be 73.43MHz. and 81.3 MHz for the Verilog_file_2 encoder design. Once the designs were placed and routed we verified the design using floorplan which give us a better picture of the resource utilized by the designs and all the input output ports used.

Table 5.1: Resource Utilization of the designs on XC2V1000 FPGA

<table>
<thead>
<tr>
<th></th>
<th>Verilog_file_1</th>
<th>Verilog_file_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slices</td>
<td>490</td>
<td>444</td>
</tr>
<tr>
<td>Number of Slice Flip Flops</td>
<td>235</td>
<td>214</td>
</tr>
<tr>
<td>Number of 4 input LUTs</td>
<td>861</td>
<td>823</td>
</tr>
<tr>
<td>Number of Board IOB</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

The floorplan also gives us an idea about the routing resources used; we see that all the wires and CLBs used are concentrated in a single location giving us an idea of modularity. We also see that the design occupies less than half of the resources available. Since both the codes are more or less the same except for the look up table we have not provided the floor plan for the second code. The similarity of the designs can be understood once we see the synthesis reports of the designs. The synthesis report for both the codes are tabulated below, we see that there is a little difference in the resource and power consumptions of the designs. The design verilog_file_1.v consumes more resources and more dynamic power than verilog_file_2.v since the look table used in this design has more entries than the one used in verilog_file_2.v.

Once the resource and power data are tabulated the JBits XHWIF was created for the Virtex II xc2v1000 board, although the library files were successfully created, unfortunately the interface couldn't be ported to the board since Xilinx Hardware
Interface for JBits 3.0 had some missing directory files for some boards and were in process of development.

Table 5.2: Dynamic Power consumption of the designs on XC2V1000 FPGA

<table>
<thead>
<tr>
<th></th>
<th>Verilog_file_1</th>
<th>Verilog_file_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Power mWatts</td>
<td>30.12</td>
<td>26.67</td>
</tr>
<tr>
<td>Current mAmp</td>
<td>12.5</td>
<td>10.51</td>
</tr>
</tbody>
</table>

Figure 5.2: Floorplan for verilog_file_1 design on XC2V1000 FPGA

Since we were unable to connect to the board using XHWIF on JBits 3.0 we reverted to the previous version of JBits i.e. JBits 2.8 since it has Boardscope GUI and Virtex Device Simulator which acts as a virtual board. JBits 2.8 is not compatible with Virtex –II FPGA.
but is compatible with Virtex FPGA so we select xcv1000 board for our test since it has nearly the same number of gates as the xc2v1000.

![Routing Congestion for verilog_file_l.v design on xc2v1000 FPGA](image)

Figure 5.3: Routing Congestion for verilog_file_l.v design on xc2v1000 FPGA

<table>
<thead>
<tr>
<th></th>
<th>Verilog_file_1</th>
<th>Verilog_file_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slices</td>
<td>486</td>
<td>449</td>
</tr>
<tr>
<td>Number of Slice Flip Flops</td>
<td>235</td>
<td>214</td>
</tr>
<tr>
<td>Number of 4 input LUTs</td>
<td>844</td>
<td>819</td>
</tr>
<tr>
<td>Number of Board IOB</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.3: Resource Utilization of the designs on XCV1000 FPGA

Since the board has been changed we conducted all the synthesis, power analysis test and floorplan generation for the new Xilinx board. The data and simulation results are given in table 5.3 and 5.4.
Table 5.4: Dynamic Power consumption of the designs on XCV1000 FPGA

<table>
<thead>
<tr>
<th></th>
<th>Verilog_file_1</th>
<th>Verilog_file_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Power (mWatts)</td>
<td>61.51</td>
<td>65.67</td>
</tr>
<tr>
<td>Current (mAmp)</td>
<td>24.61</td>
<td>26.67</td>
</tr>
</tbody>
</table>

Figure 5.4: Floorplan of the verilog_file_1.v design on xcv1000 FPGA

When compared to the Virtex-II board the Virtex board consumes more power and the maximum clock frequency at which it operates is less than the clock frequency at
which the Virtex-II board operates. Since our main aim is to dynamically reconfigure the device we used Boardscope to connect to the new board using Virtex device simulator; Virtex Device Simulator acts as a virtual board. The bitstreams were loaded onto the virtual board using boardscope and the routing density was captured using the Routing Density Viewer. We see that they are similar to the floorplan that we got through Xilinx ISE 5.

![Routing congestion for the design verilog_file_1.v on xcv1000 FPGA](image)

Figure 5.5: Routing congestion for the design verilog_file_1.v on xcv1000 FPGA
Figure 5.6: Design view of verilog_file_1.v on xcv1000 FPGA using Boardscope

Figure 5.7: Design view of verilog_file_2.v on xcv1000 FPGA using Boardscope
Since the board is virtual we couldn’t get the power results as planned for the thesis though it is possible to get the approximate power consumption using Boardscope power viewer it is a exhaustive process where we have to probe all the CLBs present in the FPGA to get the number of flip flops toggling every clock cycle and manually calculate the power by multiplying the toggle rate by a fixed a value and adding these power values to get the total power but since the power value is not accurate which we would expect after such a tedious job probing nearly 250 flipflops ,the power calculation was not carried out. The written API core to swap the design couldn’t be tested since the lack of board but with the use of Virtex Device simulator in conjunction with Boardscope we could simulate the loading of the bitstreams. We first loaded the bitstreams related to the first design on to the virtual hardware and then without resetting the device we replaced it with second design and saw the result using the routing density viewer .We can clearly see in the screenshots that when we replace an existing design by another one the design gets placed in a different location on not where the first one existed. This clearly shows that we do incur a area overhead during runtime reconfiguration of the device. Even though this is not dynamic reconfiguration as we know it is very close to one since we are only missing the hardware in the whole environment .A successful porting of the XHWIF interface on to the board should make it easier for us to reconfigure the board at runtime.
CHAPTER 6

CONCLUSION

An implementation of the arithmetic encoder was written as a Verilog module. This module was extensively tested in simulation, using test bench. The Verilog encoder correctly passed all tests as a working implementation of the JPEG2000 standard. The module was successfully synthesized and used to configure a Virtex – II FPGA. All timing information resource and power utilization results were calculated successfully.

A new tool for Run time reconfiguration was tested and used to reconfigure a virtual Virtex FPGA dynamically; even though this was a complete replacement of the design on a virtual hardware we did this without resetting the FPGA which is dynamic in nature. We couldn’t connect to the board since the Xilinx’s Hardware Interface couldn’t be ported to the specified board. Even though JBits provides access to all the resources in a FPGA it is useful and more effective only when a separate RTP core is written for a design rather than using a bitstream generated by traditional HDL tools, since in a RTP core we know the resources used and their location on the FPGA as user defined. This leads us to suggest some future work where an RTP core for arithmetic encoder can be created and reconfigured using JBits and can be used as a co-processor in JPEG 2000 image processing.
REFERENCES


[14] Design Section Head MBDA France Gerard Habay Technical Manager Deltatec, Alain Rachet Senior ASIC/FPGA Designer MBDA France Managing Partial Dynamic Reconfiguration in Virtex-II Pro FPGAs ASIC/FPGA


[25] JBits Design Abstractions Cameron Patterson and Steven A. Guccione Xilinx, Inc., 2100 Logic Drive, San Jose, CA, 95124 (USA)

{Cameron.Patterson,Steven.Guccione}@xilinx.com


[27] **Partitioning in Time: A Paradigm for Reconfigurable Computing** Karthikeya M. GajjalaPurna and Dinesh Bhatia Design Automation Laboratory University of Cincinnati Cincinnati, OH 45221 {0030 fkgajjala,dineshg@ececs.uc.edu


[31] **An FPGA-based Run-time Reconfigurable 2-D Discrete Wavelet Transform Core** by Jonathan B. Ballagh, Virginia Polytechnic Institute and State University

[32] **EBCOT coprocessing architecture for JPEG2000** Huakai Zhang, Jason Fritts, Dept. of Computer Science and Engineering, Washington University, One Brookings Drive, St. Louis, MO, USA 63130

[33] **JBits: Java based interface for reconfigurable computing** by Steve Guccione, Delon Levi and Prasanna Sundararajan

[34] **The hoplite guide to run-time reconfigurable computing** written by Matthew Scarpino Revision 1.0 – 8.13.04

[36] JRoute: A Run-Time Routing API for FPGA Hardware Eric Keller


[40] http://www.xilinx.com

VITA

Graduate College
University of Nevada, Las Vegas

Local Address:
4223 Grove circle, Apt # 3
Las Vegas, Nevada 89119

Degrees:
Bachelor of Engineering, Education, 1999
University of Madras, Chennai, India

Master of Science in Electrical Engineering, 2005
University of Nevada, Las Vegas

Special Honors and Awards:
Member of Tau Beta Pi

Thesis title: Analysis of Runtime Re-Configuration in FPGAs

Thesis Examination Committee:
Chairperson, Dr. Henry Selvaraj, Ph. D.
Committee Member, Dr. Muthukumar Venkatesan, Ph. D.
Committee Member, Dr. Sharam Latifi, Ph. D.
Graduate Faculty Representative, Dr. Laxmi Gaewali, Ph. D.