FPGA-based Hardware Acceleration for a Key-Value Store Database

by

Kevin D. Hsiue

S.B., EECS, Massachusetts Institute of Technology (2013)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
May 23, 2014

Certified by........................ Signature redacted ......
Srini Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by... Signature redacted ....
Wesley Chen
Platform Software Manager, (NetApp, Inc.)
Thesis Supervisor

Accepted by ..... Signature redacted ....
Albert R. Meyer
Professor of Electrical Engineering and Computer Science
Chairman, Department Committee on Graduate Theses

FPGA-based Hardware Acceleration for a Key-Value Store Database

by

Kevin D. Hsiue

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering
and Computer Science

Abstract

As the modern world processes larger amounts of data and demand increases for better transaction performance, a growing number of innovative solutions in the field of database technologies have been discovered and implemented. In software and systems design, the development and deployment of the NoSQL (Not Only SQL) database challenged yesterday's relational database and answered the demand for exceedingly higher volumes, accesses, and types of data. However, one less investigated route to bolster current database performance is the use of dedicated hardware to effectively complement or replace software to 'accelerate' the overall system. This thesis investigates the use of a Field-Programmable Gate Array (FPGA) as a hardware accelerator for a key-value database. Utilized as a platform of reconfigurable logic, the FPGA offers massively parallel usability at a much faster pace than a traditional software-enabled database system. This project implements a key-value store database hardware accelerator in order to investigate the potential improvements in performance. Furthermore, as new technologies in materials science and computer architecture arise, a revision in database design welcomes the use of hardware for maximizing key-value database performance.

Thesis Supervisor: Srini Devadas
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Wesley Chen
Title: Platform Software Manager, (NetApp, Inc.)

Acknowledgments

There are many people who deserve recognition and thanks for contributing to the completion of this thesis project.

My parents, Eric and Jennifer Hsiue, and my brother, Peter, who have always supported my aspirations and endeavors in life.

Professor Srini Devadas, both my undergraduate and thesis supervisor, for providing much-needed advice and inspiration during my academic career at MIT.

Wesley Chen, my NetApp manager, for making my VI-A experience exceed my highest expectations and teaching me about various aspects of the technical industry.

Joshua Silberman, for his invaluable and patient tutelage during the duration of this project; much of the thesis could not have been done without his guidance.

Steve Miller and Kathy Sullivan, for giving me the opportunity to participate in the VI-A Thesis Program with NetApp, facilitating the entire process from beginning to end.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

The recent phenomenon of big data has had irreversible effects on how society captures, processes, and stores data. As a result, the data storage industry has become both extremely competitive and lucrative, demanding constant improvement and innovation.

Very recently, the rise of big data computing has once again shifted the landscape of data storage. Various models inspired by low cost, high redundancy, and network parallelism have risen to handle massive amounts of data. With this new design paradigm, multiple challenges arose in computer engineering principles such as atomicity, consistency, isolation, and durability (ACID) properties. While a lone computer can run into issues of the ACID properties alone based on memory access behavior, an entire tribe of computers can suffer from even larger issues, and require a new level of abstraction and complexity to operate smoothly.

The data centers of today too often suffer from underutilization, penalizing the industry with various costs of environmental, fiscal, and efficiency impact. Therefore, the realm of data storage technology is often revolutionized by advances in both software design and hardware implementation; there is great incentive to populate data centers with machines built with scalable and flexible hardware architectures that maximize the utility of the entire platform.

The what and the why of the problem at hand are obvious. The underlying objective is to store an electrical charge or lack of to represent a binary 1 or 0 as data that can be encoded to represent text, audio, video or program code. The how is the more interesting aspect. A dire need for efficient, lightweight solutions for massive and parallel database technology is present in the status quo. Even more importantly, the design must anticipate near-future improvements in storage technologies and be abstracted from the particular mechanisms of memory access; the database should be modular and act as an agent between the memory pool and the controller that dispatches commands.

Recent trends in the past several decades have generally gravitated around the dependency on a faster processor. Once physical limitations became an upper bound on speed, the industry shifted towards developing platforms based on multi-core processors. Most recently, there has been great interest and research in the realm of hardware accelerators, which seek to yield better performance with the current hardware platforms that already exist.

This thesis aims to improve key-value store database performance via a Field-Programmable Gate Array (FPGA) implementation. In doing so, a novel approach to both optimizing database transaction throughput and parallel behavior will be designed, implemented, and examined.

1.1 Objectives

The primary objective of this project is to design and implement an effective key-value store database on an FPGA. Involved in this process is the exploration and understanding of the strengths and challenges of hardware acceleration as well as the various database designs that are readily used; the intersection of these two concepts produces challenging yet interesting problems to solve. A simple and concise list of goals are to:

- Explore the design of current database implementations in software.
- Develop working HDL code to emulate or replace said software, and fully test in simulation.

- Optimize the implementation to leverage its hardware strengths, rather than mimic software weaknesses.

- Perform timing analysis and produce estimated performance metrics.

- Deploy code as a module in an overall environment and obtain actual performance metrics.

Interestingly enough, much of this project focuses on designing the code so that it is reasonably implemented in hardware, liberated from a strict software mindset. To elaborate, should a programmer who is familiar with Python decide to learn C, various alterations are made to the programming sandbox, such as the allocation of memory or passing arguments by value. Translating a software concept from Python to C is similar to jumping from rock to rock in a small stream. Translating code from C to Verilog is akin to jumping from across said stream from shore to shore. An entirely new level of complexity is introduced when doing so; these challenges will be discussed in a later chapter.

The project is attractive due to its research flexibility; several factors at each stage mentioned above can be manipulated to yield interesting results. The depth of either software or hardware implementation is determined by experimental results, and different combinations of either component could produce subtle performance gains. Even the test-bench is left up for design and different performance behaviors.

1.2 Road Map

The structure of this thesis follows a logical pattern of presenting the information necessary to make well-informed design choices, the implementation and structure of the prototype, and its resulting architecture and performance.

Chapter 2 presents the background knowledge that is relevant to the large span of this thesis. Given the subject matter of the project and its realm of computer architecture, it touches upon both computer science and electrical engineering, allowing for an educationally rich span of knowledge.

Chapter 3 briefly lays out the challenges that are tackled in this thesis, which translate well to tasks that need to be accomplished in order to implement a solution.

Chapter 4 examines previous experimentation that inspired and provided guidance for this thesis.

Chapter 5 gives an overview of the different designs in both software and hardware that were explored and implemented.

Chapter 6 lays out a description of the hardware architecture and consequent design choices made in the thesis.

Chapter 7 discusses the resulting timing and resource utilization, as well as an interpretation of the results and how it applies to solving the problem introduced at the beginning of the thesis.

Chapter 8 concludes the thesis and forecasts the future expansion of this project and its implications on the data storage industry.

Chapter 2

Background

This thesis covers a wide breadth of technical detail; the following topics provide relevant foundation for understanding various aspects of the research project.

2.1 NetApp, Inc.

The vast majority of the research in this thesis was done on-site at the company headquarters of NetApp Inc., in Sunnyvale, CA, a world leader in the data storage and management industry. The company's flagship product since its inception is its family of innovative and market-defining data storage and management systems. Founded in spirit as a software company, NetApp deploys its own operating system Data ONTAP, as well as its own file system, Write Anywhere File Layout (WAFL), which both can be run on readily available off-the-shelf hardware. NetApp offered both the hardware and platform resources necessary to implement and test the utility of a hardware enabled key-value store.

The junction of this project and NetApp is at the use of an FPGA chip design as a vehicle to develop and test the proposed key-value store database hardware implementation. While the specific purpose of the chip in the NetApp architecture is interesting and a universe of complexity of its own, this thesis uses the chip as a compartmentalized platform, which allows for a certain degree of modularity and independence from the overall system.

2.2 Hardware Acceleration

The concept of utilizing computer hardware for faster performance is nothing new; specialized designs will more often than not run faster than a general-purpose central processing unit (CPU). The specialized logic in a hardware accelerator is optimized for specific purposes and therefore more efficient than a general-purpose CPU.

For example, any graphics card is inherently a hardware accelerator by offloading expensive and complex operations. Such external hardware components allow the overall system to run at a significantly faster rate. Furthermore, it is simple to upgrade or improve the hardware accelerator module.

Again, using the graphics card as an example, a user could simply purchase another, more recently produced graphics card to enjoy the updated benefits without having to worry about the underlying processor architecture. The modular aspect of hardware acceleration makes the overall concept extremely attractive in a time where system customization on the fly is in high demand.

2.3 Field-Programmable Gate Array

The Field-Programmable Gate Array (FPGA) is an integrated circuit that is reconfigurable due to its programmable-read-only-memory (PROM) and programmable-logic-device (PLD) technologies. Due to its spatially-programmed architecture, varying levels of design abstraction exist when designing the behavior of the FPGA; the most common being at the register-transfer level (RTL) via hardware description languages (HDL) such as Verilog or VHDL. The fundamental ingredients for an FPGA are base logic blocks, interconnects, and input/output ports, as seen in Figure 2.3.

2.3.1 Development History

The origin of the FPGA can be traced to Steve Casselman's work with the Naval Surface Warfare Department in the late 1980s in an attempt to develop a computer that would implement up to 600,000 reprogrammable gates; Casselman's success was

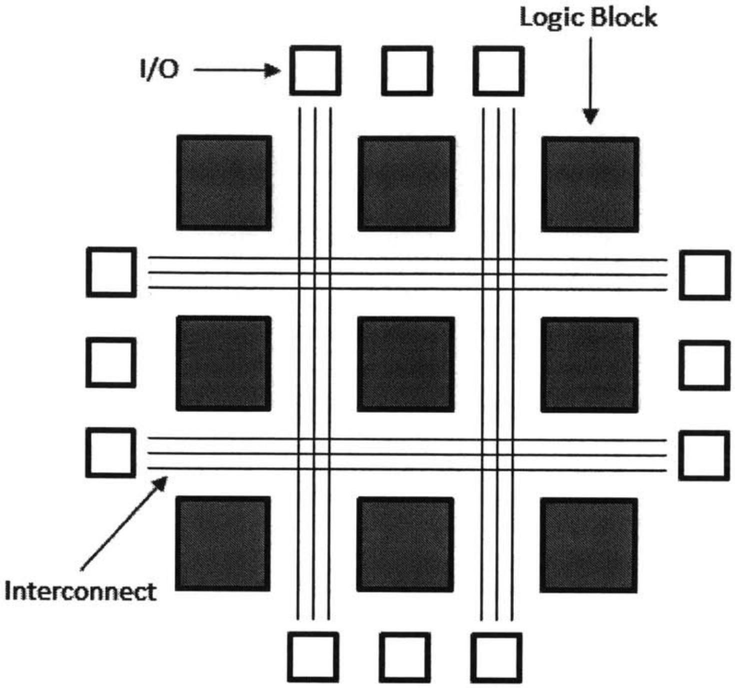


Figure 2-1: Basic representation of an FPGA

awarded with a patent in 1992. However, the first commercially-viable system, , the XC2064, was invented by Xilinx co-founders Ross Freeman and Bernard Vonderschmitt in 1985. The industry grew rapidly and by the early 2000s, FPGAs were widely utilized in both industrial and consumer applications.

2.3.2 Comparison to Peer Technologies

The FPGA resides in a happy-medium between a general-purpose central-processing unit (CPU) - high programmability, low speed - and application-specific integrated circuit (ASIC) - low (non-existent) programmability, high speed. An FPGA offers both programmability and speed. Compared to an ASIC, an FPGA has lower cost of development, particularly when considering modern process geometries. Compared to a CPU, an FPGA is capable of performing more complex operations per clock cycle.

Another technology worth mentioning is complex programmable logic devices (CPLDs). The primary distinction between a CPLD and FPGA is that the former uses a sea-of-gates and the latter uses look-up-tables (LUTs) to implement logic functions. As a result, the FPGA offers more flexibility that is not necessarily restricted to a sum-of-products logic array and is much larger in terms of various computing and memory resources.

It is important to recognize potential shortcomings of FPGAs. Due to its fine-grained capabilities, FPGAs require a degree of technical ability to both design and program, and are still slower and less energy efficient than their ASIC counterparts. Regardless, FPGAs are still widely used in the technical industry.

2.3.3 Functional Utility

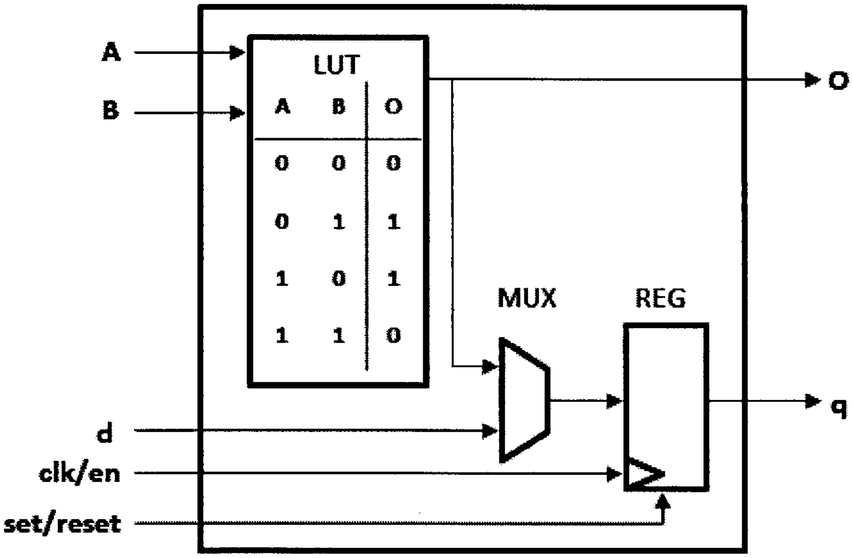


Figure 2-2: Simple logic block implementing XOR function

Essentially, an FPGA is capable of emulating digital circuitry and serving as a logic-grained black box given the proper RTL abstraction through hardware description language (HDL) code. While most designers imagine this and design their code at the digital-gate level, the actual technology behind FPGA configuration lies in a vast network of programmable logic blocks that contain LUTs, which are most similar to multiplexing blocks that produce a given output given a finite combination of inputs [3]. Different vendors provide varying designs or units of LUTs, but fundamentally the basic block of programming starts at this piece.

A toy example is demonstrated in Figure 2-2. For simplicity only two inputs are present in the LUT - A and B. Using this basic LUT building block, a design can be concise and readable in Verilog, but infinitely complex in the implementation, generating gate schematics that no human could realistically implement and test individually in the course of a lifetime.

2.3.4 Development Process

The initial development process of an FPGA design is not unlike any other software engineering process; the developer designs a block of HDL code based on specifications and tests the code using functional simulation tools. Similar to a C compiler for systems development, there is a compilation-like process of synthesis, which verifies that the code can be represented using the FPGA.

The introduction of a new dimension of complexity is the actual mapping of the HDL code to the FPGA architecture known as placing and routing, which results in a crashing of software optimism and hardware reality. Testing and timing simulation take a significant amount of effort due to various real-world constraints of deploying the previously written code to the available hardware.

At this point, the design is only a few steps from seeing sunlight for the first time as physical hardware. Logic synthesis, physical layout, and the configuration of the host device are those final steps.

2.4 Non-Volatile Random Access Memory

Non-volatile memory has the unique capability to retain data when there is no power supplied. A well known example of non-volatile memory is flash memory, which is seen in a plethora of today's industrial and consumer applications. Distinguished from random-access memory (RAM) due to its data persistence, non-volatile memory also differs from read-only memory (ROM) due to its read-write behavior.

On the NetApp systems, non-volatile flash memory is coupled with RAM to create

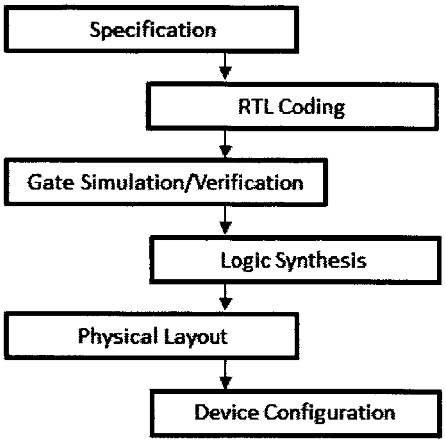


Figure 2-3: Downwards process flow of FPGA development

non-volatile random access memory (NVRAM). NVRAM is most often used for write logging. The physical component exists as a Peripheral Component Interconnect Express (PCIe) card with various chips and peripherals. A write is first logged and eventually flushed out of NVRAM memory. NVRAM is split into two halves, one which is receiving incoming write commands and the other flushing stored write commands to disk; a finite time limit forces the halves to trade roles to ensure that write commands are eventually flushed to disk [20].

As a result, NVRAM essentially acts as a journal backup for uncommitted write commands. Benefits of this approach include faster acknowledgments to user writes as well as logging data transactions for reliability. In order to perform these duties, the NVRAM card houses an FPGA, which serves as the platform of this thesis project [23].

2.5 Not Only SQL (NoSQL)

A relational database stores data in an organizational model known as the relational model. Each item corresponds to a row and may (or may not) have defined attributes for each of the column characteristics. Each table schema must identify a column that represents a primary key to uniquely identify each row; other tables can reference these rows with a foreign key, therefore establishing a relationship between the two data items [19].

Relational databases have been a dominant technology for the last few decades. However, with the rise of big (massive volume) data, weaknesses (such as impedance mismatch) of relational databases have encouraged the industry to seek out alternative database paradigms to properly deal with the oncoming storm of big data. Sometimes this data was inconsistent, without a well-defined set of characteristics that would fit nicely in a relational database.

Demand for a change in data storage was driven by the need to support large volumes of data running on multiple clusters. The practice of employing a cluster of smaller machines was very common and allowed for optimizations of storage and retrieval that relational databases simply could not efficiently carry out. Therefore, the rise of the vague term NoSQL came about, since there was no definitive NoSQL architecture or database. However, a few common characteristics of such databases include:

- Most obviously, no relational model.
- Efficient performance on clusters.
- Open source.

One major change in the mindset of database development for NoSQL was the departure of guaranteed ACID transactions (Atomic, Consistent, Isolated and Durable). Interesting paradigms begin to emerge as each one of the ACID properties is balanced and tweaked; some characteristics give, some take, but all in all, NoSQL by definition does not necessarily promise all four qualities in a given database.

2.6 Key-Value Store

Key-value stores are a powerful and robust database architecture that serve as an alternative to relational paradigms. Essentially, the database operates with a bucket of keys that each individually correspond to a defined data value. Each key should be unique. Different data structures exist that allow efficient algorithms to retrieve the associated data. The qualification for key-value store is its ability to allow for application developers to store data with schema-less data [19].

The three primitive operations that a key-value store should implement, given a valid key, value, or both are briefly described in Table 2.1.

Operation	**Description**
`Get(key)`	Retrieve the key with the associated value.
`Put(key, value)`	Store the key and the value.
`Delete(key)`	Delete the key and its associated value.

Table 2.1: Key-Value Store Database Primitives

The implementation and specifications of these three primitives will be explored more thoroughly in a following section.

Chapter 3

Challenges

The specific challenges and skill sets required for this thesis represent a significant amount of electrical engineering and computer science. Such relevant tasks include:

- Understanding and reverse engineering a large-scale C software package.
- Analyzing the critical path of a software package in various development environments.
- Using a hardware description language to implement the C software package for an FPGA.
- Establishing a cooperative link between FPGA and user interface.
- Writing required firmware for experimental analysis.
- Designing and gathering results using an appropriate test bench.

The above tasks have further implications and sub-tasks that will be expanded throughout the thesis. As a general trend, the thesis follows this cyclical research pattern: research and comprehend a software module, dissect and reverse engineer said software module, evaluate its proper representation in hardware, implement hardware model, and finally test and compare results between both self-produced software and hardware implementations.

3.1 Parallelism

When designing systems that involve interactions between software and hardware, it is important to be cognizant of the different rates at which either entity operates. More specifically, the different clock rates and latency of each particular component must be considered.

It is generally expected that hardware completes specified complex operations faster than a CPU; an ASIC is often magnitudes of order faster than software. However, engineering hardware that takes advantage of its greater speed is not a trivial task. Software latency is introduced in the delay in communication between a single processor and a separate co-processor. For a particular system, optimizing the latency in hardware is relatively useless when the magnitude of software delay is much larger; overall the system sees very little improvement. For this project, the hardware, an FPGA, sits on a PCIe bus in order to communicate with the CPU.

One such solution to achieve this goal is to enable parallel behavior, therefore increasing the throughput by allowing for multiple transactions to occur at the same rate.

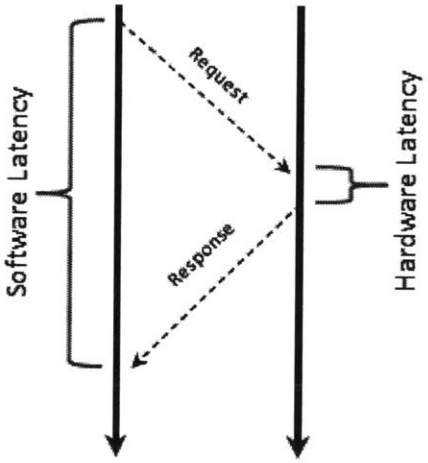


Figure 3-1: Software delay >> hardware delay

For this particular thesis, the rate at which the FPGA could perform the key-value operations was without surprise significantly faster than the software was capable of processing. Therefore, it was necessary to design a parallel architecture that could take advantage of the increased clock rate and deliver batches of transaction results. In theory this seems simple, but parallel accesses to a database introduces the possibility of multiple issues, all of which could result in erroneous data.

This is a well explored and documented issue with multiple effective solutions. Parallel operations are often to be viewed as the activity of unique threads that go about reading and writing data; mechanisms to ensure fair play mostly revolve around variations of resource locking to guarantee that collisions in data retrieval are performed serially. This approach is robust and actually can allow for much higher performance output based on the behavior of the operations. Since reading does not effectively change the data stored, multiple threads can operate and read the same data without fear of getting stale data. Writing or removing data in the database is where additional challenges come into play.

Another dimension of parallelism is introduced at the hardware and software boundary in a producer-consumer relationship. While the base Verilog key-value store implementation simply reads one set of shared registers in RAM to initiate operations in the FPGA, a parallel-enabled design could perform multiple operations at once, reading from the entire RAM itself, which is organized in such a protocol that enables software to write to a certain area, hardware to write to a certain area, and both software and hardware to read from the memory. This enables a very specific way of control; for example, software written data requests and hardware written acknowledgment and completion flags.

Parallelism is a powerful tool for improving system performance. However, as detailed in the later sections of this thesis, this improvement comes with a price in implementation design and flexibility.

3.2 Language Plurality

This thesis project requires a variety of programming languages in order to comprehend, dissect, and implement a key-value store. Each language is appropriately used at each step to maximize understanding and efficiency. The programming languages, and their use in this project, are listed below:

- Python: For implementing simple examples of various algorithms and data structures; used for simple verification and debugging.
- C/C++: Mostly used in developing models of key-value stores based on open source packages.
- Verilog: Exclusively used for building hardware key-value store.

Going from one language to the next highlighted several interesting facets about each language and the abstraction layer each one provides.

3.2.1 Python to C

The most obvious jump from Python to C is the relative simplicity, offset by the costly programming requirements set forth by Python. A primary difference in these languages is Python's built-in high-level data types and dynamic typing. There is no need to declare types of variables, arrays are augmented with list and dictionary types, and Python code is compiled at run-time. Also, Python is built around the concept of object-oriented programming, and allows for a much less constrained approach when dealing with various algorithms and data structures.

C on the other hand requires a structured programming practice that uses a static type system. The programmer is essentially tasked with and expected to correctly allocate and free memory locations. Pointers are a foundational tool in C, which allow for indirect addressing to data types or even indexing for arrays. Objects exist as grouped data types collected under structures, which then allow for shared attributes to be viewed and manipulated [8].

3.2.2 C to Verilog

Some simple differences exist between C and Verilog. Obviously, Verilog is tasked with developing the RTL logic to be implemented in the FPGA, while C exists as code that is compiled to run on a CPU. In a sense, the Verilog code is also compiled, but instead as a physical configuration that would operate as a circuit. This immediately implies an important difference: Verilog program execution can be parallel, as opposed to the serial instruction procedure of C. While parallelism does exist in the C universe (or even Python), Verilog, or hardware-description languages as a whole, can truly be parallel since the languages explicitly describe the layout of a physical device [3].

Two very interesting analogies can be drawn from adapting C code to Verilog code: dereferencing pointers or attributes and developing functions. To better illustrate these nuances, a simple C example is provided below:

```
int main() {
  int x = 0
  int *xp = &x;
  while (1) {
    *xp = *xp + 1;
  }
}
```

In this simple C program, x is incremented forever. The following mappings to Verilog can be made, should an engineer desire to create an endless counter (which is more interesting and challenging than it may seem).

- The integer variable x would be an entry to a RAM instance with a specific address.

- The integer pointer xp would also be an individual register or another RAM entry containing the memory address of integer variable x.

- The loop would be translated as a condition that triggers on a positive edge of a clock.

- A state machine would represent the different stages of the code: dereferencing the pointer (reading the value of xp), loading the variable (given its address, reading the value of x), incrementing the value x (storing the new register value), and then repeating the process (revisiting the original state and starting over).

While the mapping is not perfect, it is accurate enough to provide a methodology to at least start building hardware based on software designed databases.

Chapter 4

Previous Work

As part of the MIT Electrical Engineering and Computer Science Masters of Engineering Thesis Program, this thesis project was completed in the span of three distinct segments:

1. First summer internship - Researched performance of open source key-value store.

2. Second summer internship - Developed framework for thesis project and defined parameters of research goals.

3. Fall cooperative internship - Implemented key-value store in FPGA and tested results.

The vast majority of this thesis primarily discusses and explores the work done in the latter two portions of research. This chapter explains the preceding investigation (the first summer internship) that motivated later research.

4.1 Performance of a Key-Value Store Database

This initial project explored the performance for a new database architecture for the NVRAM card. The NVRAM functionally served as a transaction log that allowed for users to dispatch a request and continue. NVRAM collected transactions and periodically dumped them to system memory. The question was whether or not the implementation of a key-value store could perform better than an existing transaction log [20].

The goal was to investigate the performance of a new interface for NVRAM utilizing a readily-available open source key-value store database: *Kyoto Cabinet.*

4.2 *Kyoto Cabinet*

Kyoto Cabinet is a lightweight database management library that is stored in a file of key/value pairs. Each key must be unique to that database; records are organized in either hash tables or B+ trees.

Kyoto Cabinet and its predecessor were both written by FAL Labs, a personal business by Hatsuki Hirabayashi and Mikio Hirabayashi. *Kyoto Cabinet* is open source software licensed under the GNU General Public License [6].

4.3 Design Approach and Environment

The software implementation was divided into three primary tasks:

1. Research and develop a model workload.

2. Write a program that calls *Kyoto Cabinet* methods.

3. Implement timing mechanisms.

All programming was done with C++ in a Linux environment. Specifications included an Intel E31235 CPU @ 3.20 GHz, 8.00 GB of RAM, and 64-bit operating system.

4.4 Implementation of *Kyoto Tester*

Kyoto Tester was a simple C++ testing program for gauging the effectiveness of the *Kyoto Cabinet* database management library.

The pseudo-random workload was generated based on Standard Performance Evaluation Corporation Software Functional Specification (SPEC SFS) distributions. Each action specified by SPEC SFS that was applicable in the NVRAM context (Write,Create,Remove, and Setattr) was composed of primitive operations (put and replace). The primitive operations represented the abstraction layer before directly calling *Kyoto Cabinet's* database set and remove methods.

These primitive operations, put and replace, were represented as structs and contained within an operations array. Once the workload generator selected one of the four actions to perform, the corresponding collection of operation structs were defined with the correct attributes. Each struct operation was aware of its own key value, the type of operation that it was, the scheduled times for the associated primitive operations, and the size of each operation (ranging from 256 B to 64 KB for different operations). All of these attributes were specified by the predicted behavior of NVRAM.

The four macro operations, Write,Create,Remove and SetAttr were used to allocate and remove entries in the database. Table 4.1 shows the sequence of puts and removes for each macro.

Write	Put(1 – 64KB)	Put(16KB)	Put(256B)	Put(16KB)
Create	Put(256B)	Put(4KB)	Put(256B)	X
Remove	Put(4KB)	Put(256B)	Remove(4KB)	Remove(256B)
SetAttr	Put(256B)	X	X	X

Table 4.1: Puts and removes for macro operations

Once the workload was created, *Kyoto Tester* iterated through the array of all operation structs and executes each one, calling *Kyoto Cabinet's* API when appropriate. Timestamp counters were used to measure the beginning time, the end time, and the elapsed time. All of the timing figures were measured in machine cycles.

The length of the array which contained all the operation structs was manipulated for experimentation. *Kyoto Tester* returned the average latency of all the operations that took place.

4.5 Results

A shell script was written to automate test runs from a span of 10,000 to 1,000,000 operations, stepping by 10,000 operations per testing run. The operations were generated randomly but adhered to the correct percentages specified by SPEC SFS. The results are displayed in Figure 4-1. It is important to remember that the operation workload was generated by a probabilistic distribution model of the four previously mentioned macro operations. The results display a somewhat linear relationship between number of key-value store operations versus machine cycles.

While the specific details of *Kyoto Cabinet* are better documented in a variety of other sources, this initial experiment served to provide some justification for using key-value stores for database transactions. This set the stage for answering the question: can hardware perform these operations more quickly?

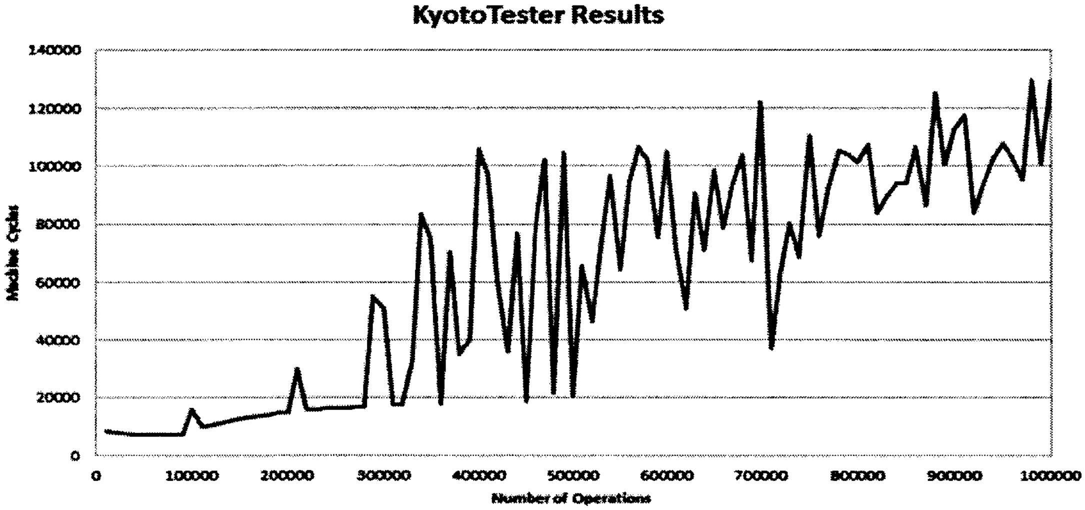


Figure 4-1: Average latency (machine cycles) as a function of number of operations

The individual average operation times are also displayed for 1,000,000 operations in Table 4.2. Note that the measuring was done in machine cycles, so at its fastest it can be assumed that these cycles were occurring at the same speed as the processor, or in this case, 3.2 GHz.

Therefore, the total number of entries that existed at some point in the table was the sum of the Write, Create, Remove and SetAttr operations, which equaled

Operation	Occurrences	Machine Cycles	Operations/Second
Write	666376	107394.74	29796
Create	49614	78862.4333	40576
Remove	66532	6423.361	498181
SetAttr	217478	27746.875	115328

Table 4.2: Individual operation timing results

the number of puts per macro operations (from Table 4.1) times the number of occurrences of each macro (from Table 4.2): $4 * 666376 + 3 * 49614 + 2 * 66532 + 1 * 217478 = 3164888$ entries. While this is not entirely accurate, it is able to serve as a basis for comparison later on in the thesis project.

An interesting observation that arose from this testing framework was that the speed of the *Kyoto Cabinet* experiment was limited by operations per second, rather than amount of data being written. For instance, multiplying the number of put operations for each macro operation in 4.1 by the operations per second of the corresponding operation in 4.2 equaled roughly 120,000 puts per second, regardless of the size of the data that was being written. Because the rate of puts per second seemed to be independent of size of data, it can be assumed that the limiting factor was indeed operations per second.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Design Considerations

In order to fully explore the breadth of hardware acceleration for database transactions, various design choices were made at different points of the project. This chapter explains the engineering thought process behind several of the monumental decisions that steered the final outcome.

5.1 Key-Value Store Databases

With the ongoing trend of massively growing data, architectural decisions have great impact on the performance of any set of database transactions. For the purpose of this thesis project, key-value store databases were chosen as the experimental model. Key-value store databases are straight-forward to understand and implement. A key of some established data type is associated with a given value. Operations on the database access the data value by using the associated key. Fundamental operations include `Get(key)`, `Put(key,value)`, and `Delete(key)`.

While some specifications can vary, these three operations are implemented at the base level of any key-value store. Small differences in behavior (such as the result of two successive `Put(key,value)` function calls) can vary between different implementations. Search trees and hash tables, two differing data structures most often utilized for key-value stores, are explored in the subsequent sections. To appropriately investigate and implement the database, software implementations of each data structure were written and tested before producing HDL code.

Table 5.1 displays the time complexity of three primitive operations: search, insert, and delete, for a variety of data structures [18].

Data Structure	**Search**	**Insert**	**Delete**
Singly Linked List	$O(n)$	$O(n)$	$O(n)$
Average Case Binary Search Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
Worst Case Binary Search Tree	$O(n)$	$O(n)$	$O(n)$
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
B-Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
Average Case Hash Table	$O(1)$	$O(1)$	$O(1)$
Worst Case Hash Table	$O(n)$	$O(n)$	$O(n)$

Table 5.1: Data structure time complexities

5.2 Search Trees

Search trees are a fundamental data structure in computer science. In its most basic form, a search tree is a binary tree whose node values are stored in an ordered set. The structure of the tree is then determined by a comparative heuristic, allowing for organized traversal and manipulation.

Several flavors of search trees with varying complexity were implemented in this project. Each of the following trees were first drafted in Python and then written in C. Only the primitive binary search tree was implemented in Verilog.

5.2.1 Binary Search Tree

A binary search tree is the base member of the search tree family. It consists of a node-based binary tree data structure with the following properties:

- A node consists of an identifying key value and an associated data value.
- A non-empty tree must have one and only one root node which has no parent node.

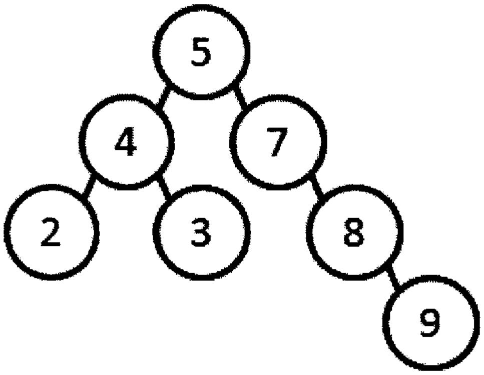


Figure 5-1: A simple unbalanced binary search tree

- Each node has at most two children nodes; a node with no children is known as a leaf.
- Each node that is not the root has exactly one parent.
- The left subtree of a node contains only nodes with keys less than the node key.
- The right subtree of a node contains only nodes with keys greater than the node key.

To provide a concrete example for the node and following tree data objects, the C struct definitions can be seen below:

```
typedef struct Node Node;
struct Node
{ int key;
  Node* left;
  Node* right;
  Node* parent;
};
typedef struct Tree Tree;
struct Tree { Node* root; };
```

The algorithm for searching for a key in the tree is very simple. In fact, the fundamental search algorithm is often used as a vehicle to illustrate recursive programming due to its simplicity; describing the process in pseudo-code essentially produces working code.

In order to find a target node n, start at the root node root and compare the keys of both nodes, n → key and root → key. If n → key is less than root → key, then the comparison is repeated with the left child of the root node key, leftChild → key, and n → key. On the contrary, if n → key is greater than root → key, the comparison is made using rightChild → key. The algorithm continues comparing each successive key until either the comparison returns an equality or a child node does not exist on the desired path (is NULL).

This search function can be succinctly described in the below C code:

```
Node* search(Tree* tree, int key) {
  Node* start = tree->root;
  while ((start != NULL) && (key != start->key)) {
    if (key < start->key) {
      if (start->left == NULL) return NULL;
      start = start->left;
    }
    else {
      if (start->right == NULL) return NULL;
      start = start->right;
    }
  }
  return start;
}
```

To put a new node in the tree, a search operation is first performed. Assuming the key does not exist in the tree, an unsorted implementation of the tree would simply walk along the tree as described above, and at its termination (reaching a node that is NULL), create a new node that points to that leaf of the tree as a parent, with the leaf now pointing to the new node as a child. It is up to the programmer to decide what should happen if the node's key already exists. By definition, there should not be duplicate keys in the database, so it is common to return a failure should the key already reside in the tree.

The insert function is shown below; a new blank node is returned by the createNode(key) function call and given attributes according to the location it is inserted in:

```
void insert(Tree* tree, int key) {
  Node* insertNodePtr = createNode(key);
  Node* trackingPtr = NULL;
  Node* currentPtr = tree->root;
  while (currentPtr != NULL) {
    trackingPtr = currentPtr;
    if (insertNodePtr->key < currentPtr->key) {
      currentPtr = currentPtr->left;
    }
    else {
      currentPtr = currentPtr->right;
    }
  }
  insertNodePtr->parent = trackingPtr;
  if (insertNodePtr->key < trackingPtr->key) {
    trackingPtr->left = insertNodePtr;
  }
  else {
    trackingPtr->right = insertNodePtr;
  }
}
```

Removing a node from the data structure is comparatively the most difficult function, but is still very simple in an unsorted search tree. Assuming the key exists in the tree, a successful delete should update both the parent node and possible children nodes with the correct pointers. Because the location and lineage of the specific node introduces various possible outcomes, a helper function, `transplant(old,new)`, is useful for reorganizing the tree in a sensible manner. This helper function is responsible for aligning the proper nodes of the tree in order to maintain its sorted order. The C code is shown below to quickly demonstrate the algorithm:

```
void transplant(Tree* tree, Node* old, Node* new) {
  if (old->parent == NULL) tree->root = new;
  else if (old == old->parent->left) old->parent->left == new;
  else old->parent->right = new;
  if (new) new->parent = old->parent;
}

void delete(Tree* tree, int key) {
  Node* deleteNode = search(tree, key);
  if (deleteNode) {
    if (deleteNode->left == NULL)
      transplant(tree, deleteNode, deleteNode->right);
    else if (deleteNode->right == NULL)
      transplant(tree, deleteNode, deleteNode->left);
    else {
      Node* current = findMin(deleteNode->right);
      if (current->parent != deleteNode) {
        transplant(tree, current, current->right);
        current->right = deleteNode->right;
        current->right->parent = current;
      }
      transplant(tree, deleteNode, current);
      current->left = deleteNode->left;
      current->left->parent = current;
    }
  }
}
```

Both inserting and deleting have a time complexity of $O(\log n)$, assuming a balanced tree. There is no active reordering in the unbalanced binary search tree; inserting an array of increasing integers would cause a binary search tree to deprecate to a singly linked list, resulting in a worst-case search time of $O(n)$.

5.2.2 Red-Black Tree

A red-black tree is a slightly more complicated binary search tree. Red-black trees are capable of self-balancing by painting each node one of two colors, red or black. Using this additional bit of coloring information, red-black trees maintain the following properties, in addition to the requirements of a binary search tree:

- A node can be either red or black.

- The root and leaves are always black.
- Every red node must have two black nodes (can be `NULL` nodes).
- Each path from any descendant leaf to the root must have the same number of black nodes.

A correctly constructed red-black tree will always maintain a path from the root to the furthest leaf that is no more than double the path from the root to the nearest leaf. As a result, the tree is height-balanced, and will maintain a time complexity of $O(\log n)$.

Searching through a red-black tree is identical to searching in a binary search tree. While the benefit of a balanced tree is obvious in terms of performance time, the associated algorithm to balance the tree is cumbersome to implement in hardware. Balancing occurs anytime a node is either inserted or removed from the set. A traversal of the tree orders and rotates the various branches by using the node colors as a heuristic.

Figure 5-2 is an example of a simple red-black tree with the same keys as the tree in Figure 5-1. Note that the red-black tree has a height of 3, which is less than the previously illustrated binary search tree.

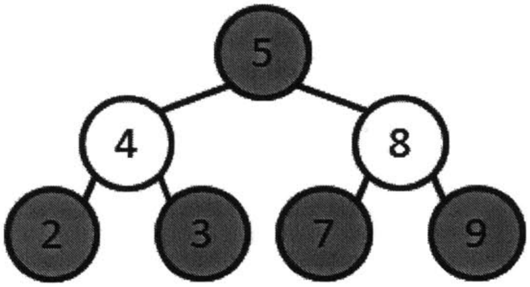


Figure 5-2: A red-black (white-gray) tree

5.2.3 B-Tree

The final form of the search tree explored in this thesis is the B-tree, or more specifically, the 2-3 tree. This format optimizes performance for databases that read and write large blocks of data, taking advantage of spatial and temporal locality during data manipulation.

The properties of 2-3 tree are listed below, differing slightly from the previously mentioned trees:

- Nodes can have more than two child nodes; the 2-3 tree allows each internal node to have either two or three.
- Each node stores up to two ordered keys; every internal node can either have two children and one key, or three children and two keys.
- Leaf nodes have zero children and either one or two key values.

By allowing each node to store more units of data, the height of the tree decreases and the number of node accesses is reduced; accessed nodes are capable of delivering information on multiple keys [11].

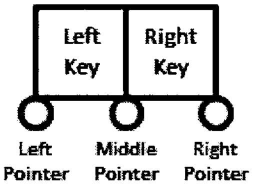


Figure 5-3: Anatomy of a 2-3 tree node

Within the node, the keys are ordered - the lesser of the two keys is on the left, the greater on the right. The associated child pointers on the left, middle, and right correspond to node values that are less than the left key, between the two keys, and greater than the right key, respectively. The same rules for children keys from binary search trees apply: the subtree pointed to by the left pointer contains values less than the left key, the middle pointer points to a subtree with keys in between the left key and right key, and the right pointer points to a subtree that has keys greater than the right key.

Searching is very similar to searching in a binary search tree. Starting at the root, comparisons are made recursively from top to bottom, deciding at each level

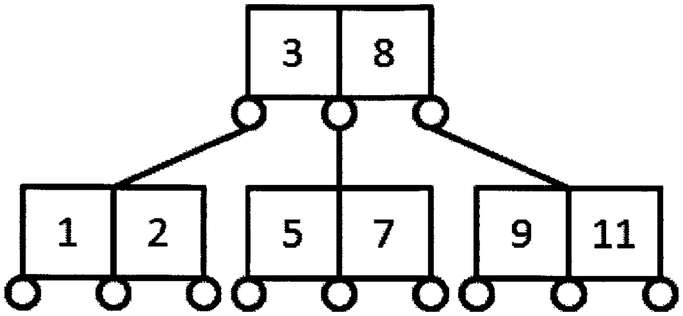


Figure 5-4: Sample 2-3 tree

how the desired key compares to the keys in the node and then which child pointer to traverse. Each memory access produces one node with multiple keys to iterate through, therefore amortizing the latency penalty of reading memory.

Insertion into the tree identifies which node the target key should be in and checks to see if there is less than the maximum amount of keys allowed - a 2-3 tree checks if there is either one or zero keys in the node. If there is room, the key is inserted in the correct order. Otherwise, the median value of the two original keys and the target key is sent up the tree, creating three nodes: new node with the median value, with a node containing the lesser value as the left pointer, and a node containing the greater value as the middle pointer. A pictorial demonstration is shown in Figure 5-5.

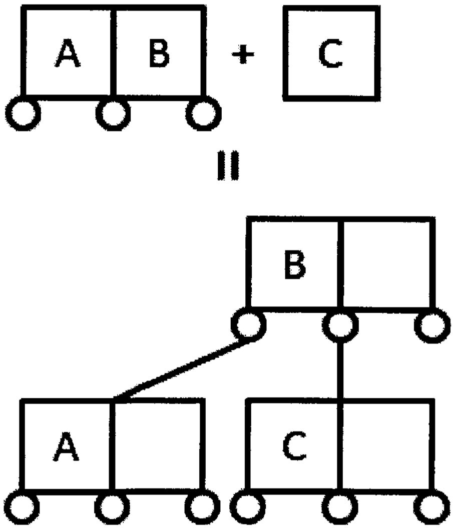


Figure 5-5: 2-3 tree insertion of key C (assuming A<B<C)

Deletion from the tree finds the target key and removes the item, restructuring the tree to maintain its invariants. Whereas insertion splits and creates new nodes, deletion merges and deletes existing nodes to maintain the tree structure.

As for performance, all of searching, insertion, and deletion have a time complexity of $O(\log n)$, similar to the red-black tree. The key difference is the practical ability to load multiple keys into one node, which can represent various units of physical memory structures; for example, a node can store an entire page in a cache, and therefore rely on locality to minimize memory accesses.

5.3 Hash Tables

A hash table maps keys to values using a hash function to calculate an index for an array of buckets. Ideally, the index is always unique and there is an infinite amount of buckets, but this is not always the case. More often than not, the entire set of keys is unknown and memory space is finite. Hash collisions occur when more than one key maps to the same index as another key. When this occurs, the colliding key is appended to the linked list that corresponds to that bucket [10].

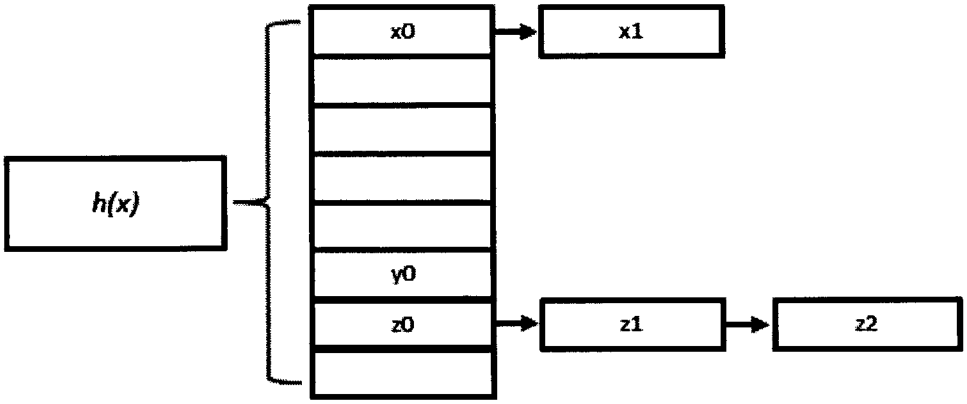


Figure 5-6: Hash function and data array with key collisions

During a search, the same hash function is used to repeat the calculation of the index of the correct bucket. If the key is found, then the search is successful. If it not found, and there is more than one key in the bucket, then the linked list is traversed until either the key is found or there are no more keys to check. Deletion of a key also relies on the hash function to compute the index. Deleting a key that is present in the bucket requires updating both preceding and successive key pointers.

It is interesting to note that the data structure rooted at each index bucket can be virtually any form of a linked list or a search tree. While hash collisions as described earlier with a linked list have a time complexity of $O(n)$ a desirable time complexity of $O(\log n)$ can be achieved by using either a red-black tree or a 2-3 tree.

Several hash functions were experimented with and implemented in various languages for this thesis. Below, brief explanations for each of these hash functions are provided.

5.3.1 Most-Significant-Bit Mask

A simple hashing function (used initially to develop the hash table) is the omission of a certain number of the most-significant-bits.

For example, the bitwise exclusive-or of key byte `0xAA` with mask `0x0F` would result in the hash index of `0x0A`. Essentially, the higher nibble is discarded and the lower nibble becomes the index. The same hash index would be calculated from `0x8A`, `0xFA`, or `0x0A`; a total of up to sixteen keys would collide and be inserted into the `0x0A` bucket.

For the sake of testing, this method was extremely useful when manipulating randomly generated multi-byte keys. This method is not attractive as a hash function since the resulting index is largely dependent on only a portion of the original key.

5.3.2 Division Method

The division method is another simple hashing method. Given a key x and a divisor M (usually a prime number), the function is $h(x) = x \textit{ modulo } M$. The *modulo* operator, or %, returns the remainder of x divided by M.

For example, given a hash table with eight buckets and divisor 8, a key of `0xAA` (decimal 170) would map to key index `0x02` (decimal 2), since `0xAA` divided by `0x08` yields a whole number quotient of 21 with a remainder of 2.

Similar to the most-significant-bit mask, the division method is simple to compute but not optimal due to its dependence on only a portion of the bit representation of x.

5.3.3 Jenkins One-At-A-Time

The Jenkins One-At-A-Time hash function was written by computer scientist Robert John Jenkins. The hash is a good example of the preferred avalanche behavior, where one minute change in the input results in significant changes in the output. Furthermore, the function takes various bits of the input to make successive, dependent changes on the final hash index.

The code (programmed in C) for the Jenkins One-At-A-Time hash function is shown below [7]. Notice how each bit of the key is first used to shift and change the resulting hash.

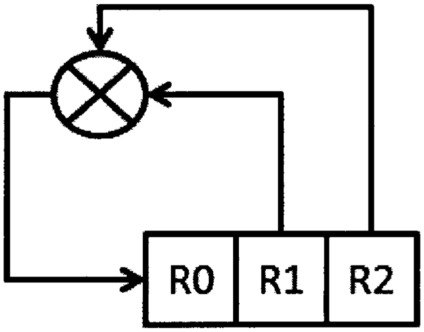
```c
uint32_t jenkinsoat(char* key, size_t len)
{
  uint32_t hash;
  uint32_t count;

  for (hash=count=0; count<len; ++count)
  {
    hash += key[count];
    hash += (hash << 10);
    hash ^= (hash >> 6);
  }

  hash += (hash << 3);
  hash ^= (hash >> 11);
  hash += (hash << 15);
  return hash;
}
```

5.3.4 Cyclic Redundancy Check

A cyclic redundancy check (CRC) is a coding method that detects errors in data transmission. Its strength lies in its relative simplicity. Blocks of data to be transferred are extended with a short check value - the remainder of a polynomial division of their contents. On the receiving end, the calculation is repeated to ensure that the data had not suffered from corruption during transmission [22].

The use of cyclic error-correcting codes was first proposed by W. Wesley Peterson during 1961 in order to detect errors in communication networks. Implementation and verification of the code was simple due to its systematic calculation and its resulting fixed-length check value.

The CRC requires a unique generator polynomial, which represents the divisor in a long division function. Once the algebra is performed, the quotient is ignored and the remainder becomes the check value. To complement the design of modern computer architecture, the Galois field of two elements - *GF(2)* - is used. To implement the CRC, a linear feedback shift register is utilized as a base building block.



Step	R0	R1	R2
0	0	0	1
1	0	0	0
2	0	1	0
3	1	0	1
4	1	1	0
5	1	1	1
6	0	1	1

Figure 5-7: Simple three-bit LFSR

The linear feedback shift register (LFSR) is a design where the output of a standard shift register is logically manipulated and then fed back into its input to produce a finite cycle of output patterns. This is achieved by maintaining an array of shift registers with taps that represent feedback functions (typically exclusive-ors). For every polynomial, there is a combination of taps that yield a maximal length of output sequence.

The values transmitted from a CRC are fed into an LFSR, which holds a checksum that is dependent on every bit in the data stream. Both sender and receiver have the same CRC calculator, and therefore are capable of verifying that the transmitted data is correct.

The CRC has several attractive features as a hashing function for this implementation; it is a reasonable compromise between primitive hash functions and complexity. The resulting hash is created using the entire length of the key, and has the previously mentioned avalanche effect. Compared to other hashing schemes (SHA1 or MD5), CRC is computationally much less complex in both design and implementation in embedded hardware.

Chapter 6

Implementation

This chapter explores the implementation of the key-value store database in hardware. While the previous chapter of design considerations produced Python or C programs, the end result of this chapter is entirely written in the Verilog hardware description language. The sequence of topics builds the design from the ground up, starting from the bare fundamental data type definitions and ending with the final implementation with revised optimizations.

6.1 Verilog Hardware Description Language

The Verilog Hardware Description Language is one of the most commonly used languages for FPGA development. One of the first modern hardware description languages to be invented, Verilog was created by Prabhu Goel and Phil Moorby and eventually purchased by Cadence Design Systems in 1990. This prologue section examines an example snippet of code to explain some Verilog syntax and conventions used in the rest of the chapter [3].

```
module simple_register #( parameter DBITS = 32 )
(
input clk, reset, enable,
input [DBITS-1:0] value_in,
output [DBITS-1:0] value_out_wire,
output reg [DBITS-1:0] value_out_reg
)

assign value_out_wire = value_in;

always @ ( posedge clk or posedge reset )
  if ( reset ) value_out <= {DBITS{1'b0}};
  else begin
    if ( enable )
      value_out <= value_in;
  end
```

Verilog syntax is very similar to the C programming language syntax. It is case sensitive and has near identical control flow keywords and operator precedence. Its differences lie within its mechanisms for describing physical hardware constraints.

A Verilog design is built from a hierarchy of modules, each of which communicate through a declared set of inputs and outputs. As seen in the example, `simple_register` is the module name. The # denotes parameters for the module, which can be configured before compiling to give variable lengths for instantiation. In this case, the parameter `DBITS` is set to be 32, which determines the width of `value_in`, `value_out_wire`, and `value_out_reg`. As seen above, there are two outputs: `value_out_wire`, which is simply a wire, and `value_out_reg`, which is a register.

Verilog uses the `always` key word to express a block of code that is triggered on a specified condition. In `simple_register`, the positive edges of both `clk` (clock) and `reset` allow for execution of the block code. Wires can be assigned values using the `assign` key word as shown above for `value_out_wire`.

Information is passed through either registers or wires. Registers maintain the stored value, wires do not. Values can be assigned to registers with non-blocking (<=) statements, which execute in parallel without the needing temporary storage variables, or blocking assignments (=), which execute sequentially [17].

For example, to swap the values between registers A and B, the non-blocking example below would be successful, whereas the blocking example incorrectly results in both registers storing the original value of B.

```
\\ Non-blocking
A <= B;
B <= A;
\\ Blocking
A = B;
B = A;
```

The simple_register module uses a non-blocking statement to assign the input wire value_in to the output register value_out_reg if the input wire enable is asserted on each positive edge of the clock. If a reset is asserted, then the output register is loaded with the value 0. While value_out_wire is simply a pass-through of value_in, value_out_reg retains the value set when enabled until it is either updated or cleared by a reset.

Though the simple_register code is useful for demonstrating the basic concepts, it does not illustrate more complex Verilog programming paradigms. A design style frequently employed in the implementation was encoding the next state transition as combinational logic. At each state, the next state was determined in that cycle and then registered to be active on the next clock edge. While the result was the same as writing transitions based on the last state environment, combinational logic for the next state allowed for easier to understand code. For example, when successive states read from RAM, checking the next state and loading the resulting address during the current state allowed the desired data to be active on the next clock cycle. The alternative would be advancing to the next state, checking the correct address, and then reading the data in the following state, one cycle late.

6.2 Basic Building Blocks

The implementation of the key-value store database utilized two basic building blocks: random-access-memories (RAM) and First-in, First-out (FIFO) buffers.

RAM

RAM was the primary representation of internal memory for the key-value store database. It could be automatically inferred by the compilation process, allowing for efficient allocation of FPGA resources to accommodate the desired amount of memory.

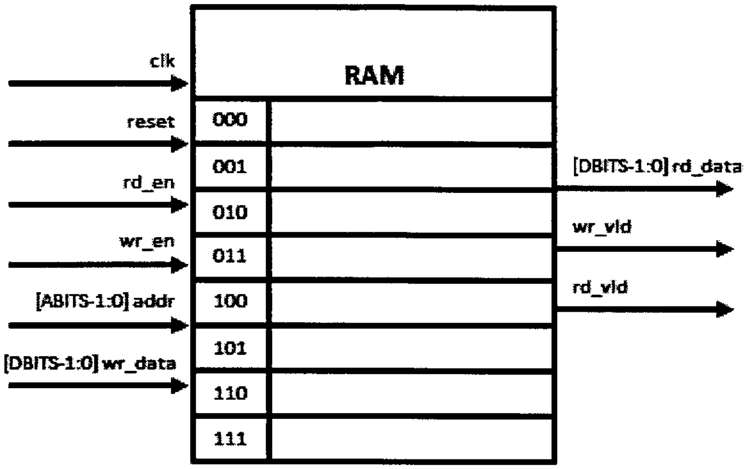


Figure 6-1: Block representation and I/O of RAM

Figure 6-1 illustrates both the input and output controls as well as the block representation of RAM. Even though the specific implementation varied from design to design, the fundamental mechanics were identical. Given an address and an enable signal, RAM either read a value or wrote a value on the next clock cycle after the enable was asserted.

RAM can be classified by its number of ports; each port allowed independent concurrent access to RAM . A dual-port RAM would allow for two different addresses to be accessed simultaneously. However, design complexity increases with number of ports; there must be a way to resolve conflicts with reading and writing to the same port.

FIFO

FIFO queue buffers were used to store data when the input rate was different from the output rate.

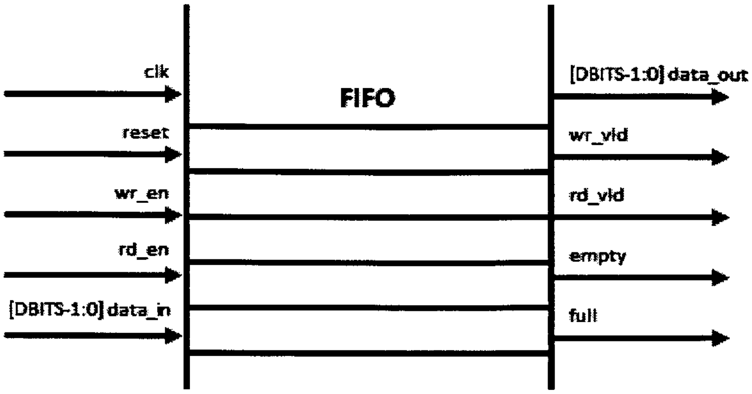


Figure 6-2: Block representation and I/O of FIFO

Figure 6-2 shows the block diagram representation and input/output wires of a FIFO buffer. The implementation was very similar to a RAM and differed only in the determination of the read and write addresses. Internal read and write pointers gave the next address for dequeuing or queuing; the FIFO was either empty or full if the pointers were equal at zero or the max index, respectively. A dequeue incremented the read pointer and a queue incremented the write pointer.

An optimization to the FIFO was the ability to peek at the output value, eliminating the need to spend an additional cycle registering the output value.

6.3 System Parameters

The basic units in the system were the key, value, and operation. While the actual width in bits of each element changed throughout the course of the project, the final implementation supported a key of 32 B and a value of 16 B, with an operation encoded as 1 B with extra bits to spare for future functionality. While a 16 B value is small compared to normal workloads, this thesis assumed that 16 B was sufficient to indirectly reference an address somewhere else that stored a much larger value.

The key-value store accepted inputs of packets consisting of a key, a value, and an operation. Get(key) either returned the desired key and value if it existed in the database or returned a failure. `Put(key,value)` either successfully stored the key and value into the database or returned a failure if the key already existed. `Delete(key)` either successfully removed the key and value from the database or returned a failure if the key did not exist.

Note that the clock and reset signals are included for all designs but not described explicitly. The clock signal was the global clock and the reset was written as a asynchronous global reset. To clarify, the Verilog snippet of an asynchronous reset shown below triggers on both clock edge and reset edge, allowing reset to occur independent of the clock.

```
always @ ( posedge clk or posedge reset )
  if ( reset ) ...
  else ....
```

6.4 Interface to External Memory

The implementation described in this thesis project utilized external memory in its final form. In the beginning, simple memory models were written in Verilog as instances of RAM, but later on, the entire data structure was stored in external memory. Given the properties of external memory - high bandwidth and high latency penalty for access - operation parallelism for the key-value store was extremely desirable.

Operations were dispatched with a unique context that stored their information. The final implementation used these contexts to service multiple operations at once to hide the latency of access to external memory, thereby increasing the throughput of the entire system.

6.5 Binary Search Tree

The binary search tree (BST) was implemented first as an exercise in converting C to Verilog. Due to its role as an exploratory vehicle, this implementation was not self-balancing and only maintained keys; the BST was functional but left incomplete due to preference for the hash table.

6.5.1 Inputs, Outputs, and Parameters

The inputs and outputs are shown in Figure 6-3 and described in Table 6.1.

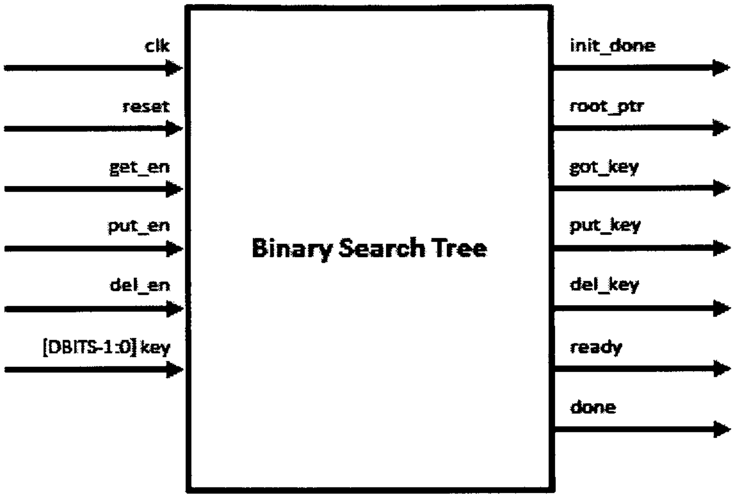


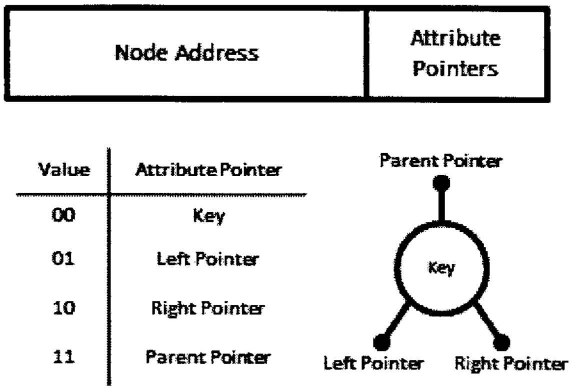
Figure 6-3: I/O of BST block

Type	Name	Description
Parameter	DBITS	Width of data bits corresponding to key
Input	get_en	Enables get operation
Input	put_en	Enables put operation
Input	del_en	Enables delete operation
Input	key	Key of width DBITS for operation
Output reg	init_done	Set if memory initialization is complete
Output reg	root_ptr	Address storing BST root pointer
Output reg	got_key	Successful get operation
Output reg	put_key	Successful put operation
Output reg	del_key	Successful delete operation
Output reg	ready	Set if BST is ready for a new operation
Output reg	Done	Set if BST completed an operation

Table 6.1: Descriptions of BST block I/O

6.5.2 Node Anatomy

Each node was represented as four entries in RAM, the two least-significant-bits of a given address represented the left child, right child, or parent of the node. The representation of node can be seen in Figure 6-4 below.



Value	Attribute Pointer
00	Key
01	Left Pointer
10	Right Pointer
11	Parent Pointer

Figure 6-4: BST node representation

6.5.3 Node Allocation

The same data structure used to store the search tree was used also to maintain control over which nodes were blank and available for writing. Instead of storing a parent pointer, a blank node stored a pointer to the next blank node. Before a node was inserted, the blank node list was consulted; the root index was written to and added to the tree with the new root being the child of the previous root. When a node was deleted, the tail of the linked list, which originally had a `NULL` next pointer, would point to the recently deleted node. This method of node allocation had little overhead, but was cumbersome to implement, especially with the edge cases of having either empty or full total memory.

6.5.4 State Representation

While the primary states for the BST are shown in Figure 6-5, many more intermediate states were involved in the manipulation of keys, such as the series of `transplant` states that served as a helper function for the `delete` operation. Incremental steps of the primary stages are concatenated in the figure; for example READ was actually written as five unique states (READ0, READ1, ..., READ5) to account for reading data from the unique address of the key, left child, right child, and parent of the node.

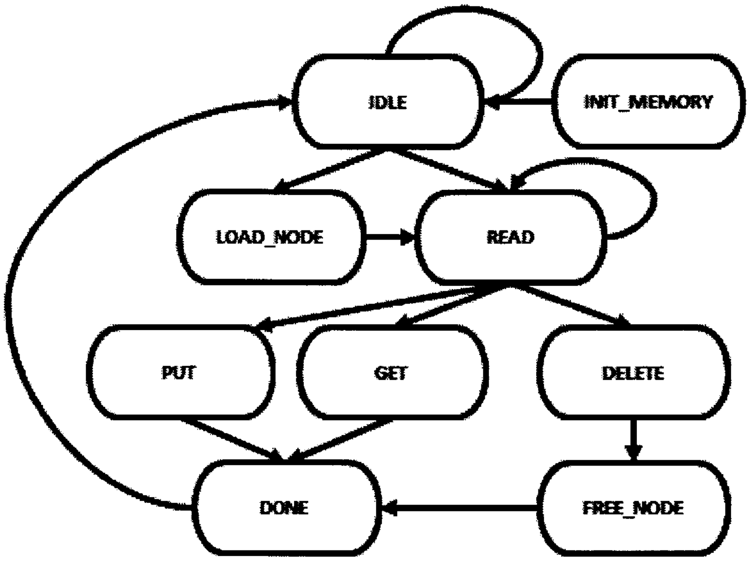


Figure 6-5: BST state diagram

The search tree instantiated a RAM instance and upon initialization, would walk through and set all the values to a NULL value (MSB set to 1) in the INIT state. Once this was completed, init_done was asserted and the module waited in the IDLE state until one of the operation enable lines was asserted.

Getting a key would require transitions through the READ state to find the correct key. If the traversal led to a NULL dead-end and then state DONE, the done signal was held high without a got_key high signal. This represented a failure. The same failure results applied to both signals put_key and del_key , given different failure conditions. If a key already existed for a put, it was a failure. If no key existed for a delete, it was a failure. Each of these failures would end up in the IDLE state.

Deleting a key would also start with the READ state, but after a successful delete would end in the FREE_NODE state. Deleting is perhaps the most involved operation, since there are a multitude of options to restructure the tree in order to to maintain its invariants.

Putting a key started with the LOAD_NODE state to guarantee that there was room in RAM. Once a node was loaded, the correct location to place the node was found by the READ states, and the new node was inserted.

On a successful operation, the corresponding success signal is asserted as well as the done signal.

6.6 Hash Table

It was decided that the hash table would be a more feasible implementation for the overall system for several reasons. Primarily, there was no need to reorganize the data structure - once a key was removed the memory location was simply marked as available for writing. Accessing the memory (without collision) was independent of the number of objects in the database, therefore allowing for a constant access time.

The initial system design is depicted in Figure 6-6. Keys were represented in 32 bytes and values in 16 bytes. Parallel operations were supported by storing each operation in its own context in a Context RAM. The Context RAM indexes were passed throughout the system and upon completion, were available to service a new operation. Parallelism was achieved by processing new operations while a memory access occurred; each read or write to memory to traverse the data structure was a discrete step of an operation.

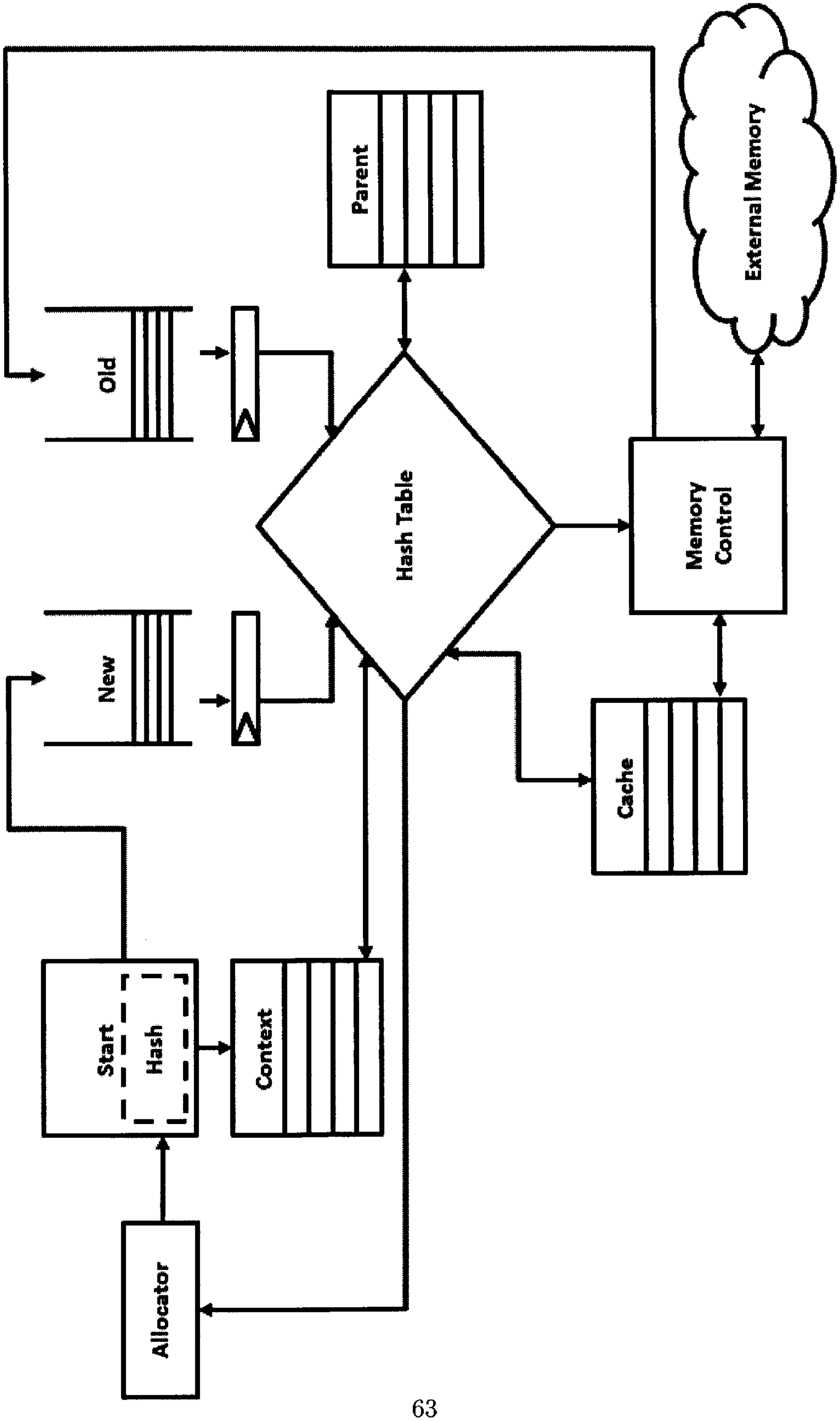
6.6.1 Parameters

The parameters of the hash table are shown below in Table 6.2. These parameters were used throughout the entire design and instantiated from the top module of the key-value store database.

6.6.2 Start

Figure 6-7 describes the inputs, outputs, and state transitions of the Start module. Upon a positive edge on the request input (assuming the ready signal was asserted), Start exited the IDLE state and requested a blank context index from the Allocator

Figure 6-6: Initial System Design



Parameter	Description
OP_W	Operation encoding
KEY_W	Key length
VAL_W	Value length
HASH_W	Hash buckets
PTR_W	Memory address pointer
MEM_W	Memory address
OPS_W	Parallel operations
CTX_A_W	Context address
CTX_D_W	Context data
FIFO_A_W	FIFO address
FIFO_D_W	FIFO data

Table 6.2: Hash table parameters

in the CTX state and wrote to Context RAM in the WRITE states.

At the same time, Start stored the incoming operation in internal key and context FIFO buffers. The Hash Function dequeued the key FIFO as soon as it was non-empty and a hash index was computed from the key. The context FIFO held the context index and operation. Upon completion, the Hash Function dequeued the context FIFO to obtain the associated index and operation, and finally queued the index, operation, and hashed memory pointer to the New FIFO in the DONE state.

The ready signal was then held high if there was vacant memory and if the Allocator had available context indexes to distribute. If either of these conditions were false, then Start would no longer be capable of accepting input operations.

6.6.3 Allocator

The Allocator module was implemented as a FIFO buffer that stored the indexes of available locations in the Context RAM. Initialization queued the indexes to the internal FIFO and once the FIFO was full (all indexes were queued), the rd_ready and wr_ready signals were set, given that the FIFO was neither empty nor full.

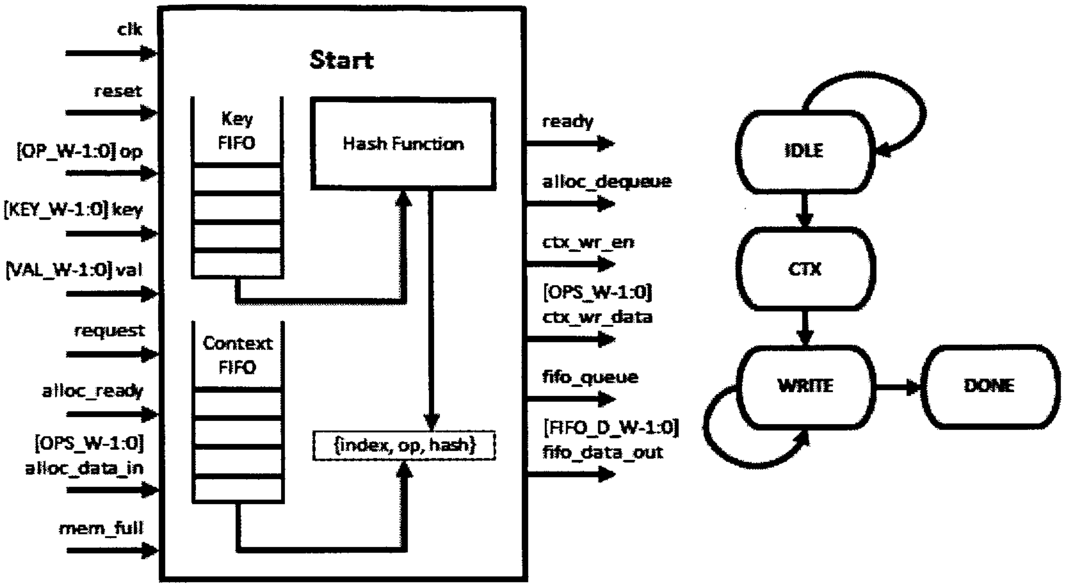


Figure 6-7: Start block and state diagrams

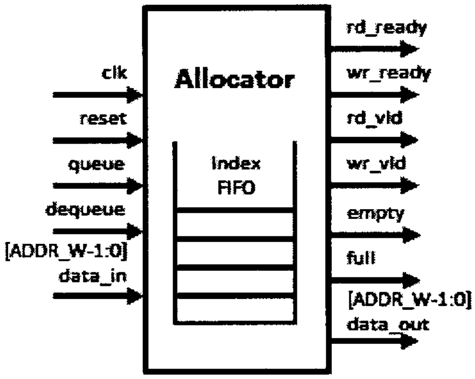


Figure 6-8: Allocator block diagram

6.6.4 Context RAM

The `Context RAM` was responsible for storing the key and value. Instead of having to pass the entire key and value between each module of the key-value store database, the context index, operation, and hash index was all that necessary as an identifying tag.

Operation	Context Index	Hash Index

Figure 6-9: Anatomy of an operation tag

Context RAM had two ports that were controlled by the Start module and Hash Table module. Start was capable of only writing (to update keys and values) and Hash Table was capable of only reading (to load keys and values).

6.6.5 New FIFO and Old FIFO

In order to queue operations before entering the Hash Table, New FIFO was used to schedule new input operations. Whenever New FIFO was non-empty, Hash Table would dequeue New FIFO for the next operation to process.

Old FIFO was used to handle successful memory accesses, and stored the same parameters as the New FIFO: context index, operation, and hash index. When a memory access completed after successfully updating the Cache RAM, the Memory Control module queued the relevant information onto the Old FIFO, which was then read by the Hash Table. However, in order to ensure parallelism, Hash Table always checked and dequeued Old FIFO to complete ongoing operations before servicing new operations in New FIFO.

One non-trivial addition to the common FIFO structure was the ability to peek forward for both New and Old FIFO buffers. Additional control logic and registers were used to display the to-be-dequeued data.

This was required to ensure consistency in the database. Whenever a Put(key, value) or Delete(key) was queued in New FIFO, Hash Table completed all the operations currently in the system before dequeuing New FIFO. Whenever a Put(key, value) or Delete(key) was queued in Old FIFO, Hash Table dequeued the Old FIFO to ensure that operations that could manipulate the data structure were completed serially. Though forcing the system to complete putting and deleting serially was a brute force solution to ensuring data consistency, the frequency of either operations was dependent on the workload, which in this model, was relatively low.

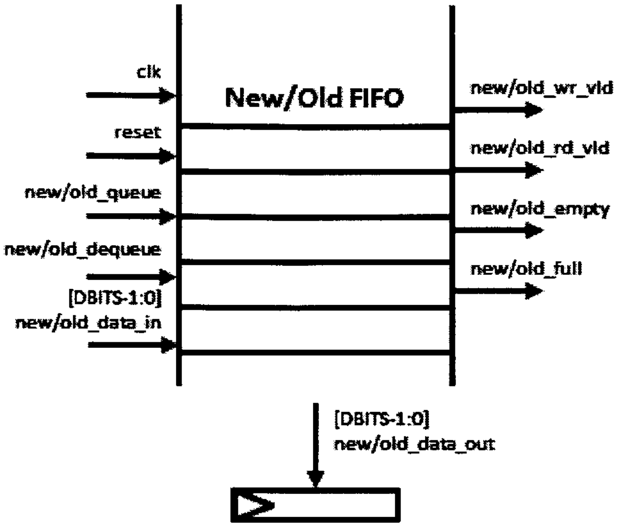


Figure 6-10: New/Old FIFO block diagram

6.6.6 Hash Table

The Hash Table module was the core of the key-value store database and was responsible for receiving operations, processing memory responses, and dispatching memory requests.

Operations were received from either the New or Old FIFO. As mentioned previously, if a new operation altered the data structure by inserting or removing an entry, then the Hash Table ensured that all current operations were completed to guarantee consistency by favoring the Old FIFO over the New FIFO until all entries in the Context RAM were accounted for. Once all operations were completed, the insert or delete proceeded and the Hash Table transitioned to only dequeuing from the Old FIFO, which contained only the current insert or delete operation.

If a new operation tag (as shown in Figure 6-9) were loaded into the Hash Table, a memory request would be dispatched to the Memory Control module to load the associated data stored in memory. A memory access for a new operation would return the root pointer stored in that hash bucket; if no node existed at that location then a NULL value was returned.

However, if the operation tag came from the Old FIFO, then the memory had

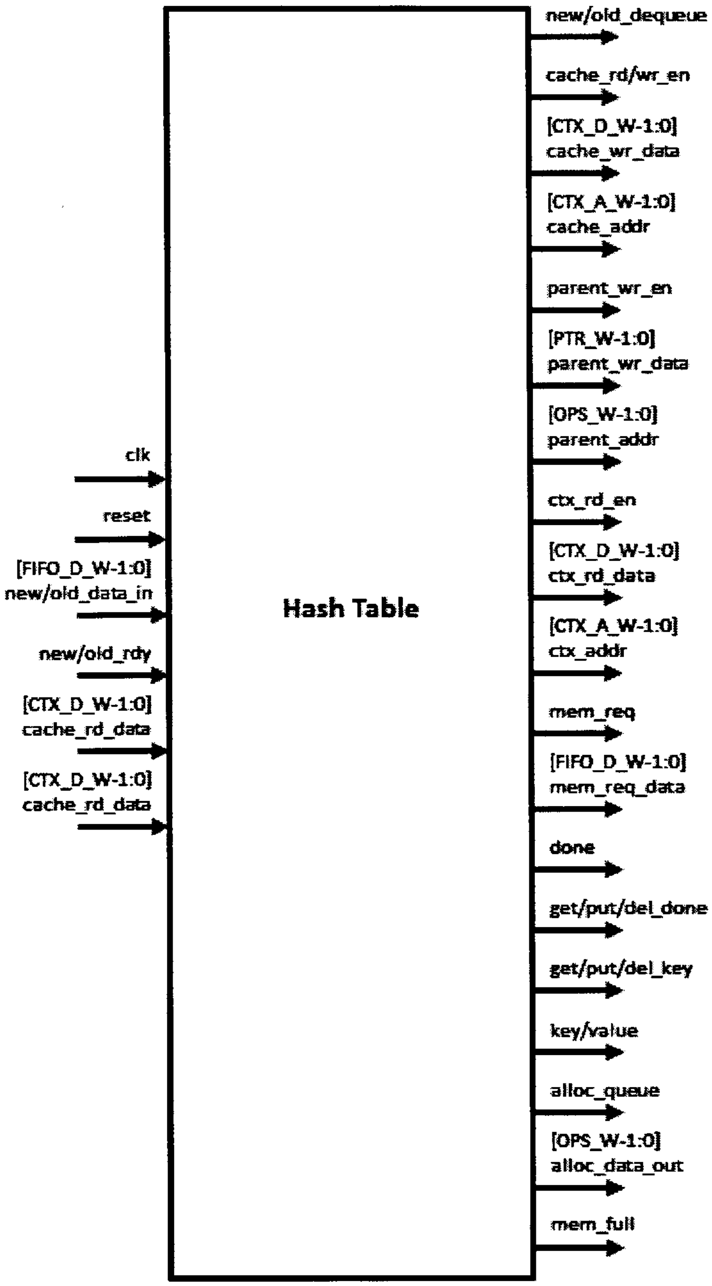


Figure 6-11: Hash Table I/O

already been loaded in a previous cycle and was stored in the Cache RAM. The context index was processed to obtain the attributes associated with the operation in a series of READ states. Once the Context RAM data was loaded into registers, a series of compare (CMP) states checked the key values between the memory data loaded in the

Cache RAM and the operation data stored in Context RAM. If the key matched, then Hash Table would transition to the appropriate operation state. If the key did not match, then a memory request for the next key in the hash bucket was dispatched. If no pointer existed, then the key did not reside in the database and a failure was returned. Parallel accesses were thus broken up at each modular memory access for a given key value and traversal down the linked list of each hash bucket.

Once the correct key was found (if it existed), then the associated operation dictated the state transition to the correct state for getting, putting, or deleting.

Figure 6-12 illustrates the state transitions of the Hash Table module. Note that not all transitions and states are shown, the general flow is depicted for simplicity.

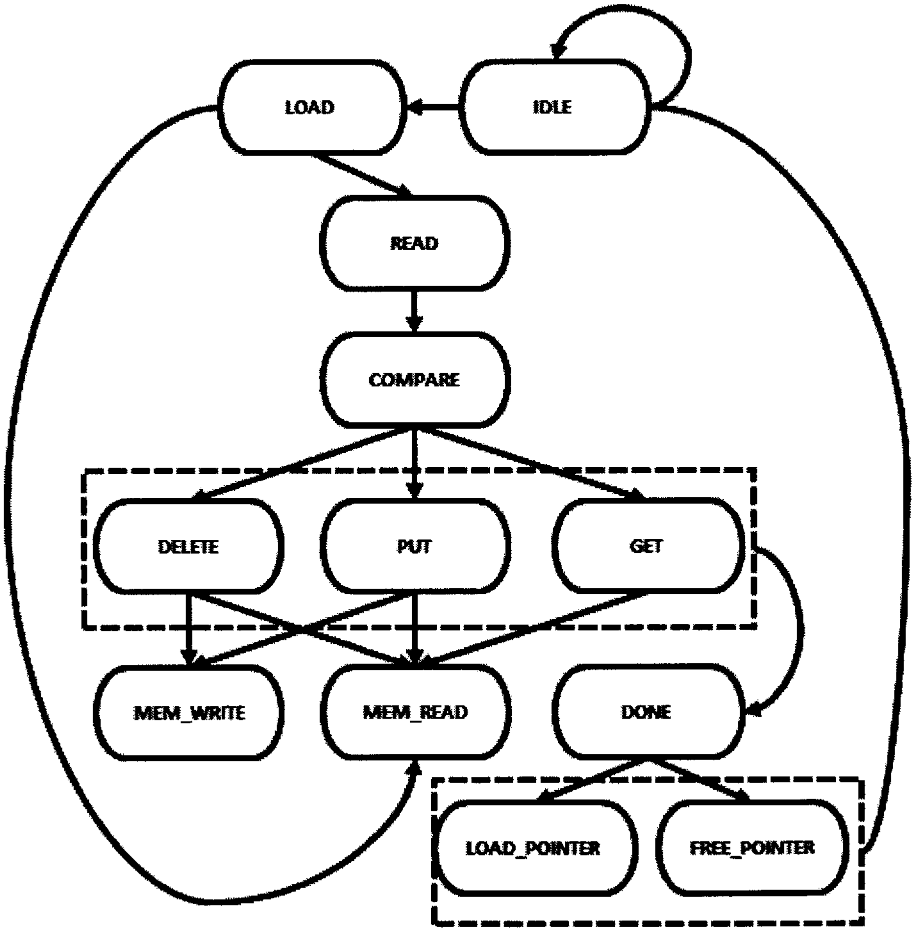


Figure 6-12: Simplified Hash Table state diagram

GET would terminate on either the correct key or a NULL pointer. Because GET did not alter the data structure, the Hash Table could support an entire Context RAM of GET operations.

PUT would check if the key already existed and if not, traverse the linked list until a NULL pointer was found (which could be at the root of the hash bucket) and inserted the key and updated the parent or root pointer information if necessary.

DEL was slightly more complicated. If traversal reached a NULL pointer, then a failure was returned. Several actions were possible if a value was found. If the key was at the root of the hash bucket, then the key was deleted and the root was marked NULL. If the target key was not the root then the parent key also had to be updated to point to the child of the target key, which could have either been another key or NULL. Figure 6-13 an example of deleting an intermediate key.

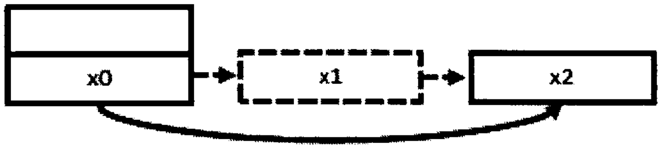


Figure 6-13: Deleting intermediate X1 key

Either PUT or DEL could write to memory in MEM_WRITE. Furthermore, these two operations, if successfully completed, would enter either the LOAD_POINTER or FREE_POINTER states to properly manage locations of available memory.

To take advantage of locality in terms of potential manipulation of a parent key, hash index and context index of parent keys were cached at each cycle of the Hash Table; deleting or putting keys into the database would often require altering the parent key information.

Once an operation completed, the done signal was asserted. If the operation were successful, then an accompanying get_key, put_key or del_key signal would confirm success; the key and value outputs would mirror the associated data as an output of the key-value store. The context index would then be queued back to the Allocator module to allow for a new operation to enter the system.

6.6.7 Parent RAM

The Parent RAM stored the information for associated parent nodes. Both deleting and inserting required modifications to the parent key, altering the parent pointer to point to either the deleted child's next pointer or a NULL value.

Parent RAM was exclusively controlled by Hash Table and Memory Control in order to allow for proper updating of parent pointers. The Hash Table was responsible for updating the parent key during traversal, and the Memory Control read from the Parent RAM in order to load the proper key for updating.

6.6.8 Cache RAM

In order to store accesses and modifications to memory, the Cache RAM served as an intermediate buffer between the Hash Table and system memory. This allowed for the implementation to treat nodes as one contiguous chunk of memory and isolated individual memory access requests.

Memory loads were stored in the Cache and read by the system, and any modifications were written to the Cache and eventually flushed to memory. In order to mirror the full capacity of the Context RAM, the dimensions of the Cache RAM had to be the same as Context RAM; the address widths were equal.

6.6.9 Memory Control

While this thesis project served as an initial foray into the utility of a hardware-based key-value store database, it was important to enable future improvements and actual use. Therefore, instead of simply instantiating smaller-scale memory modules on the FPGA for testing, a significant effort was put into interfacing the key-value store with actual memory components. In this system, 8 GB of double data rate synchronous dynamic random-access memory (DDR SDRAM) was used. Note that this module in particular was heavily influenced by the design of the NetApp infrastructure, so some explicit detail will be omitted from the description.

Interactions between memory and the key-value store database were managed by a Memory Control module. This module accepted memory requests from the Hash Table, dispatched the requests to DDR SDRAM, and handled the memory responses.

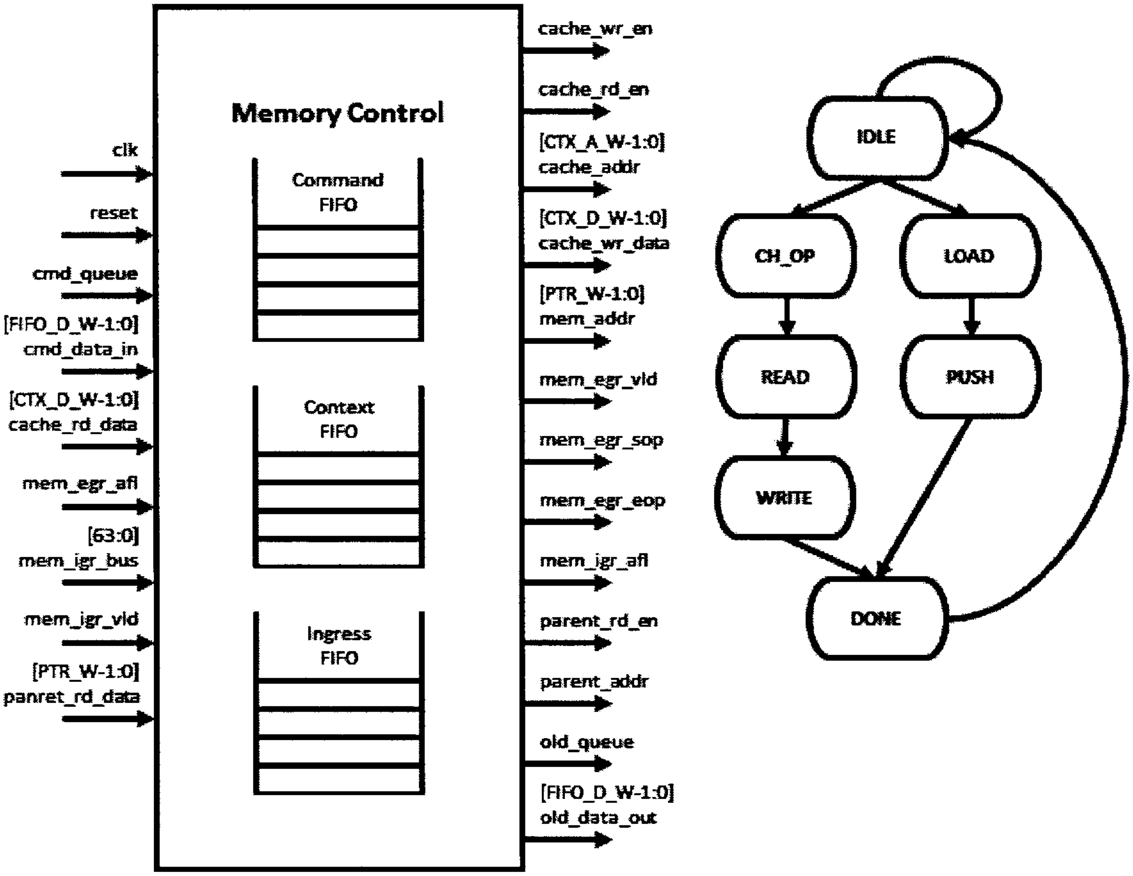


Figure 6-14: Memory Control I/O and state diagram

On reset, the Memory Control initialized the memory and then entered an IDLE state. Memory Control either handled a memory response or an incoming command, both of which were buffered by FIFO buffers. If the memory FIFO was dequeued, the following series of states in LD (load) wrote the incoming data into the Cache RAM, while dequeuing the command FIFO would result in assembling a packet to dispatch to memory.

The operation tag that was passed between the Old FIFO, New FIFO, and Hash Table determined how the memory was accessed and written to the Cache RAM. In certain situations, a single operation required multiple memory accesses, such as altering the pointer of a parent key. As a result, additional encodings were used to signal these unique operations. For example, deleting a key with both a parent and a child (such as in Figure 6-13) required updating the parent pointer to point to the child. The operation tag had an encoding that represented this specific procedure and both `Hash Table` and `Memory Control` correctly traded off responsibilities to update context and memory accordingly.

Transactions between memory were handled by packets of incoming or outgoing data. Two lines for ingress and egress controlled the interaction between the `Memory Control` module and system memory. When a response was registered, the incoming ingress data was buffered in a FIFO and then processed in the loading states, as described previously. When an operation completed and relinquished its context index, the corresponding entries in `Cache RAM` were written to memory via the egress bus.

6.7 Software Interface

The interface to the key-value store database was designed using much of the existing infrastructure implemented by NetApp engineers. This section gives a brief explanation of the interface for context, but does not describe the details or source code in depth.

The software side of the junction was implemented in C and allowed for a simulation program to control and communicate with the Verilog code. The key to the design was to share a data structure between C and Verilog. The data structure was writable in some areas by only software and others by only hardware but entirely readable by both, therefore allowing for a hand-shake protocol of requests and responses.

The C program maintained an internal data structure to keep track of the inputs and outputs of the database. For example, a key was stored in C as an array of four `UINT_64t` types, and processed and packaged by the top Verilog module as a 256 bit value. In the same way a key was stored, a value or a specific operation could also be stored. A request bit in the shared data region was then set by software.

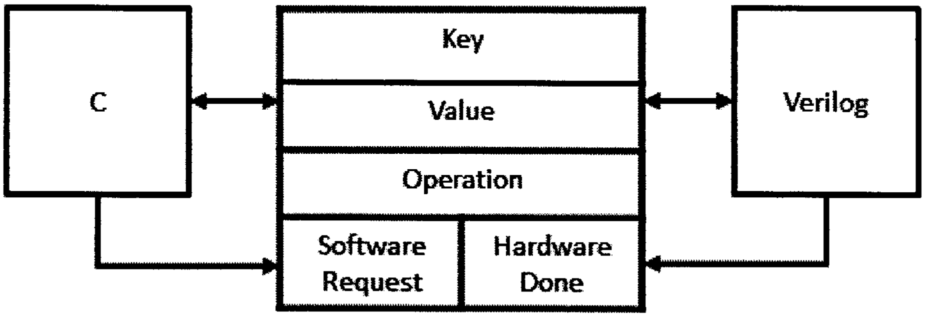


Figure 6-15: Sample data structure shared by software and hardware

On the Verilog side, the top module constantly polled if a request was made. Once a software request was asserted, the top module fed the correct inputs to the key-value store and asserted a `request` signal. When an operation completed, the `done` signal wrote the returned keys and values and a completion bit in the shared data registers. The C program constantly polled this bit and once asserted, would return the associated values of the requested operation.

This initial mapping allowed for great flexibility in both development and testing. The C programming language gave more leeway for designing and employing robust test cases, and was controllable from the command line. Unfortunately since the software waited until a bit was set by hardware, the entire system was at first forced to operate in a serial manner, since only one operation was acknowledged by both software and hardware at a time; the solution to this problem is described in the next section.

6.8 Optimizations

Once the initial prototype was created, further development allowed for debugging and a more intuitive interface. During this brief testing period, multiple emergent issues inspired the effort to introduce several optimizations to the existing infrastructure.

6.8.1 Adding CRC Hash

The original implementation relied on a painfully primitive hash function that only took a number of the least-significant bits as a direct hash index. While this approach was appropriate for prototyping purposes, the scale and size of real-world workloads required something more robust in hashing.

Therefore, a CRC module, as described earlier, was integrated into the system. This was relatively simple, since the `Start` block contained a `Hash Function` module and buffered both its inputs and outputs, allowing for modular switching with virtually no effect on the surrounding infrastructure.

6.8.2 Shared Context Memory

The primary obstacle to the desired parallel behavior of the key-value store emerged at the software and hardware junction. Because the C program initially had no way of distinguishing between different contexts and spun in a loop until a `done` signal was asserted, only one operation was dispatched at a time.

To combat this emergent issue, the system design was subtly changed. The final Verilog implementation no longer maintained an internal `Context RAM`. Instead, the `Context RAM` module was moved up to the software-hardware junction as a general `Context` array. The allocation of memory had to reside somewhere, and no apparent penalty was incurred by moving it out of hardware

Essentially, what was once before a shared data structure for one operation grew into an entire array of data structures that corresponded to parallel operations. The same bits to poll requests and completions were used, but this time both hardware and software iterated through an array of contexts to check the bits and process eligible contexts. The `Context` stored an array of either free or busy contexts. The software handled user inputs and searched for a free context. When a free context was found, the software populated the memory with attributes of the operation and asserted the `request` signal for that context.

The hardware simultaneously polled the same `Context` array and upon receiving a request, unpacked the information and delivered the hash index and operation to the key-value store database. There was no longer any need for a Context RAM or Allocator in hardware; software now had the responsibility of allocating and storing the context information. Instead, an Input FIFO buffered the inputs of context index and operation to the Hash module, which directly then fed into the New FIFO of the initial implementation. Upon completion, the return key and value were written to Context and external memory; the completion bit for that specific context was set. An updated block diagram of the entire key-value store is shown in Figure 6-16.

This optimization allowed for the scheduling of multiple operations since software no longer required a done signal to proceed; instead it could take in as many operations as the Context could hold, thereby hiding the latency of the software component of the system.

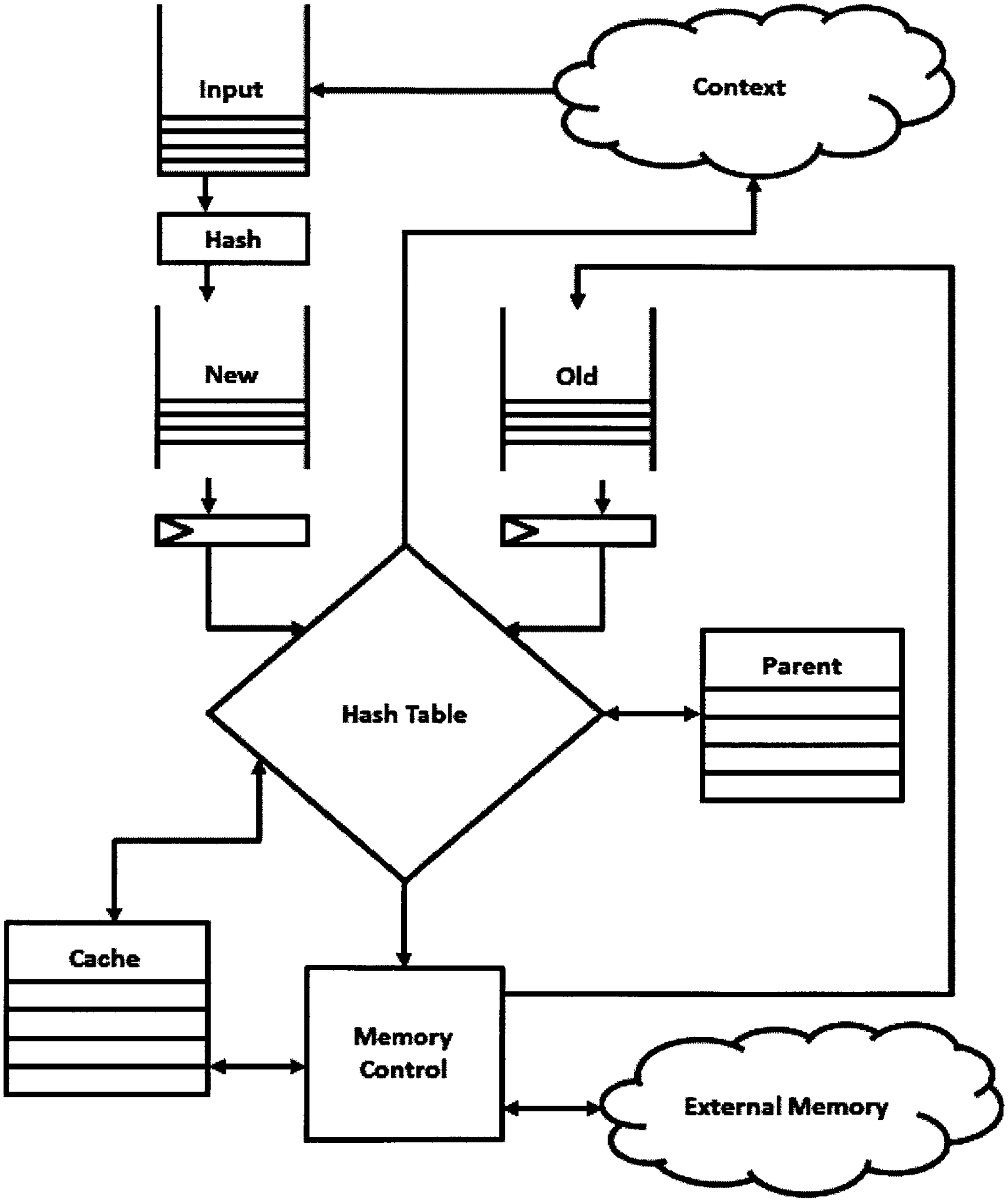


Figure 6-16: System design with external context

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Results

The hardware implementation of the key-value store database was exhaustively tested to ensure correctness, and also measured in simulation to give conclusive performance results. This chapter examines the testing environment and proceeds to discuss the results in both resource utilization and simulation timing.

7.1 Testing Environment

Testing was implemented as a deterministic comparison between the hardware key-value store and a C-implemented singly-linked list using randomly generated keys, values, and operations.

Given the programmable C interface, a robust set of test cases could be deployed to guarantee the correct implementation of the key-value store database. An example of the parallel testing for Get operations is shown below:

```
int testGet(sharedContext* sc, testList* list, int key) {
  int c, v;
  c = find(list, key);  \\ Operation on singly-linked list
  v = get(kvr, key);    \\ Operation on hardware implementation
  if ( c != v ) return 0;
  else return 1;
}
```

Running a suite of tests which dispatched the primitive test operations and comparing the results was suitable for debugging and obtaining data for the initial (internal `Contex RAM`) system design. A loop dispatched a specified number of randomized requests, keys, and values to simulate database behavior. At the time of completion, test runs of up millions of operations completed successfully without discrepancy.

More interestingly enough, a side-to-side comparison with *Kyoto Cabinet* allowed for further verification on end results. The *Kyoto Cabinet* methods, once included from the header source file, were called in the same way as the singly-linked list. While this confirmed correct end result behavior, this did not allow for a proper timing comparison, which will be discussed in a following section.

The final system implementation (shared `Context`), given its out-of-order completion, was individually tested but not in parallel with a software model for verification. This was justifiable since there was no change to the underlying mechanics, only the location at which `Context` resides in memory.

7.2 Resource Utilization

The primary attractiveness of hardware acceleration is the trade of optimal performance for a subset of common operations at minimal hardware cost. The result of this thesis project was no exception from this desirable exchange. Even more appealing is its extremely small footprint on the existing framework of the current NetApp FPGA designs. Both the length of the code and physical resource consumption of the key-value hardware store are dwarfed by the current existing design.

While the length of Verilog code often has little to no correlation to physical resource consumption, it is worth noting that the entire thesis project was concise enough to fit within an order of magnitude of a thousand lines. Even more interesting is that the light-weight implementation could exist as multiple instances in an overall system, allowing for multi-database integration for common applications. Various levels of abstraction can be employed in a larger network, where keys in one database can refer to identifying tags for entire databases themselves, and so forth.

When compiled, synthesized, placed, and fitted to an Altera Stratix IV FPGA, the overall logic utilization of the key-value store database was 1% of the total resources. This was the summation of the combinational ALUTs, memory ALUTs, and dedicated logic registers, which are reported in Table 7.1 below.

Resource	Amount	Percentage
Combinational ALUTs	879/140,600	<1%
Memory ALUTs	64/70,300	<1%
Dedicated Logic Registers	1074/140,600	<1%
Block Memory Bits	27,776/11,704,320	<1%

Table 7.1: Logic utilization

The logic utilization consumption is minimal on the FPGA, less than 1% for each dedicated resource. This confirms that the key-value hardware implementation is affordable and a feasible alternative.

7.3 Timing

The results gathered for timing analysis were derived from running the entire test suite in simulation and measuring the output waveforms. Though not fully confirmed in actual FPGA testing, the simulation was sufficient to give a general sense of the timing scale.

Using the timing simulation program, the maximum frequency for the system was 244.56 MHz. The simulation timing for each operation in a single context (no parallelism) is shown in Table 7.2. Compared to the previous results presented in Chapter 4, these results are several orders of magnitude larger than their respective functions in the software model.

Operation	Time (ps)	Operations/Second
`Get(key)`	188223	5,312,846
`Put(key,value)`	234509	4,264,228
`Delete(key)`	211855	4,720,209

Table 7.2: Operation timing

As with the results for resource consumption, the speed of the, allowing for up to roughly five million operations per second, which is certainly a significant increase in performance than the status quo. Note that the timing is for an individual operation, so in theory parallel accesses would multiply the throughput by the number of possible parallel operations. The underlying point is that this implementation is capable of supporting enough parallel contexts to hide the memory latency by taking advantage of its bandwidth. For example, if a generic DDR3 memory offered an estimated 6 GB/s of usable memory, a maximum of 96 million operations could occur per second. Whether or not this upper bound was reached was determined by the throughput of the core logic, which attempted to achieve the maximum bound through parallel operations.

7.4 Comparisons

Even though the results generated in this thesis project are very optimistic about the performance of the hardware implementation of a key-value store database, it is necessary to ground expectations by maintaining context of what the perceived speed actually means.

According to the *Kyoto Cabinet* website, the hash database is capable of storing one million records every 0.9 seconds given an overhead of 16 B per record [6]. While the dimensions are different and can certainly change the exact comparison, it is evident that the hardware implementation is capable of greater throughput in its initial state. The extent of that difference is not immediately clear, since the performance of the hardware design will undoubtedly scale with the size of the database, similar to *Kyoto Cabinet*.

As discussed in Chapter 4, the preliminary investigation showed an average serial latency for different macro operations in the *Kyoto Cabinet* testing framework. The comparison between this hardware implementation and *Kyoto Cabinet* is not necessarily one-to-one. The Verilog simulation timing results do not take into account the latency of the software used to dispatch and receive the operations to the hardware.

So it can be expected that the overall system latency is indeed higher than the reported results. However, this latency penalty is mitigated by the fact that multiple contexts can be in flight, increasing throughput of the overall system. For example, assume that the additional system latency were 1 `ms` and the dominant latency. Then, at most one million operations could be completed in a second. The previously discussed optimization of maintaining a shared array of contexts between hardware and software allowed for this latency to be hidden by dispatching multiple contexts, therefore increasing throughput. This optimization would have been significantly more difficult to implement in the software-based key-value store database.

Furthermore it is is important to remain cognizant of the differences in operating power, size, and scaling. An FPGA consumed a minute fraction of the power used by a desktop computer running *Kyoto Cabinet*. Also, the FPGA had one memory controller, as opposed to the multiple memory controllers for a desktop computer. These differences require further investigation to fully develop a case that may be compared on a one-to-one basis. Regardless, this foray into the realm of key-value store databases is still indicative that an FPGA-based hardware implementation is both feasible and attractive.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 8

Conclusion

The results of this thesis project were ultimately satisfactory; a combination of minimal space requirement, high throughput, and parallel capability in the hardware implementation of a key-value store database encouraged future development.

8.1 Predicted Application

One of the primary challenges of extreme-scale (thousand-core) computing is energy efficiency. Recent research has indicated that coordinating multiple hardware and software optimizations results in a Pareto optimal efficiency in terms of power consumption. The hardware implementation of the key-value store database serves as one such hardware solution.

The use of the design can be readily predicted in both the scope of NetApp systems as well as modern computing at large. The ability to rapidly identify keys that correspond to values is required at various levels of the software and hardware stack. As the recent shift has gone from increasing parallel processing power to satisfy application requirements, this is one step in answering a too seldom asked question: can current hardware capabilities be leveraged to obtain better performance?

This implementation could serve as a substitute for a processor transaction log, a back-end server handler for a web page, or even the central node for database coordination in warehouse-scale computing. The versatile application of the implementation is both exciting and promising for increased system performance.

8.2 Future Work

Given the scope of this project, the prototype design was sufficient to gather conclusive results and direction. However, there are several areas of improvement that could be pursued to maximize the utility of the hardware key-value store database, all of which are described below.

8.2.1 Hybrid Data Structure

A potential optimization of the internal hash table data structure would be the inclusion of some type of tree to handle hash collisions, as opposed to a simple singly-linked list.

While collisions would be resolved at a time complexity of $O(n)$, any tree with resolution complexity of logarithmic behavior would be a more attractive alternative, especially since the associated cost in this system is spent primarily on the latency of memory requests. Even if half of these accesses could be omitted, this would be a significant performance improvement. The resulting structure would be a hash table which mapped each bucket to the root of a search tree, which would improve both latency and maximum operations per second.

Of course, this assumes that collisions do occur. Assuming that the hashing is done in hardware, and that the volume of keys processed is orders of magnitude larger than the capability of different buckets in the hash table, the hybrid data structure would offer a significant decrease in latency due to minimal memory references per query.

8.2.2 Intelligent System Locking

In this implementation, an alteration of the data structure required freezing the entire system. First, all current operations are completed, and then the data-altering op-

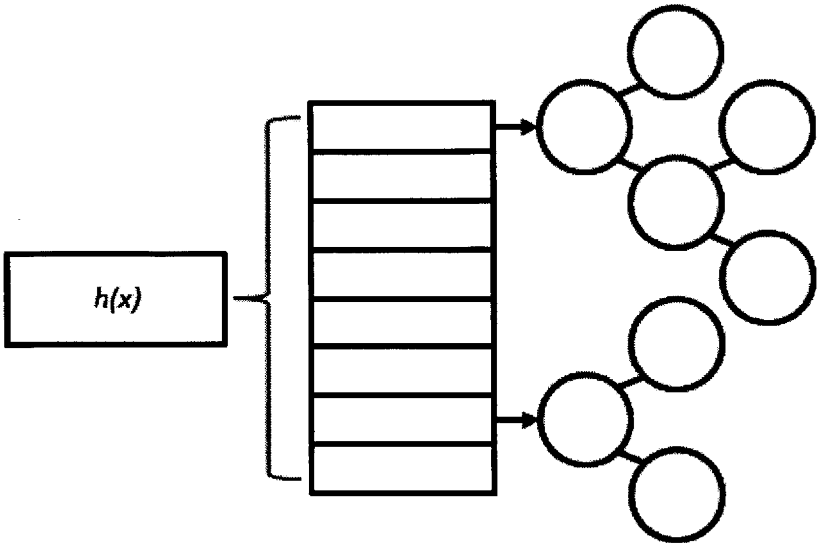


Figure 8-1: Hybrid hash-table and tree data structure

eration processed. The system resumed once the data-altering operation completed. While this was justified by the predicted low frequency of such operations (inserting and deleting) for application use, it is apparent that different applications will have varying workloads that could have a higher percentage of inserts and deletes.

The system-freezing is a brute force approach. Refined solutions exist and could be implemented in the future. For example, even the actual step that alters the data structure could be delayed since the location or key still has to be found, given that successive accesses to that key are delayed until the alteration completes. Or, relationships can be mapped for each bucket and appropriately locked if an operation altered that specific hash bucket, as opposed to locking the entire hash table.

Given the right workload, this could lead to high returns in performance efficiency, since parallel operations could still exist for inserting and deleting, instead of forcing serial behavior upon operations that can change the data structure.

8.3 Final Word

The goal of this thesis project was to implement and investigate the performance of an FPGA-based hardware accelerator for a key-value database. The final design was successful in that regard. Besides optimizations to the system design, increased strenuous testing with more realistic workloads and eventually actual hardware would illuminate the full extent of performance improvement. Though only a prototype was implemented as a proof-of-concept, the results were sufficient to warrant further investigation.

Bibliography

[1] Peter J. Ashenden. *Digital Design: An Embedded Systems Aproach Using Verilog.* Morgan Kaufmann, Burlington, Massachusetts, September 2007.

[2] Jon Bentley. *Programming Pearls.* Addison-Wesley, Crawfordsville, Indiana, second edition, September 2013.

[3] Ken Coffman. *Real World FPGA Design with Verilog.* Prentice Hall PTR, Upper Saddle River, New Jersey, first edition, January 1999.

[4] Abhijeet Gole and Naveem Bali. Efficient use of NVRAM during takeover in a node cluster (US 7730153 B1), June 2010. http://www.freepatentsonline.com/7730153.html.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Waltham, Massachusetts, fifth edition, September 2011.

[6] Hatsuki Hirabayashi and Mikio Hirabayashi. Kyoto Cabinet: a straightforward implementation of DBM, March 2011. http://fallabs.com/kyotocabinet/.

[7] Robert John Jenkins. A hash function for Hash Table lookup, 2006. http://www.burtleburtle.net/bob/hash/doobs.html.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall PTR, Upper Saddle River, New Jersey, second edition, January 2012.

[9] Steve Kilts. *Advanced FPGA Design.* Wiley-IEEE Press, Hoboken, New Jersey, June 2007.

[10] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, Westford, Massachusetts, third edition, January 2012.

[11] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Westford, Massachusetts, third edition, January 2012.

[12] Kyle Louden. *Mastering Algorithms with C.* O'Reilly Media, Sebastopol, California, August 1999.

[13] Clive 'Max' Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows.* Newnes, Burlington, Massachusetts, April 2004.

[14] Steve Miller. A Peek Under the Hood of the FAS6200, May 2012. https://communities.netapp.com/docs/DOC-9948.

[15] Samir Palnitkar. *Verilog HDL.* Prentice Hall PTR, Upper Saddle River, New Jersey, second edition, March 2003.

[16] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* McGraw-Hill, New York, New York, third edition, August 2002.

[17] Blaine C. Readler. *Verilog By Example.* Full Arc Press, first edition, 2011.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, third edition, July 2009.

[19] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley, Upper Saddle River, New Jersey, first edition, August 2012.

[20] Mark Woods, Amit Shah, and Paul Updike. Intelligent Caching and NetApp Flash Cache, July 2010. https://communities.netapp.com/docs/DOC-6508.

[21] Donald Thomas and Philip Moorby. *The Verilog Hardware Description Language.* Springer, New York, New York, fifth edition, October 2008.

[22] Henry S. Warren. *Hacker's Delight.* Addison-Wesley, Boston, Massachusetts, second edition, October 2012.

[23] Jay White and Carlos Alvarez. Back to Basics: RAID-DP, October 2011. https://communities.netapp.com/docs/DOC-12850.

[24] Mark Woods. Optimizing Storage Performance and Cost with Intelligent Caching, August 2010. http://www.netapp.com/uk/media/wp-7107.pdf.