

DESIGN AND IMPLEMENTATION OF A HARDWARE FILESYSTEM

by

Ashwin A. Mendon

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2008

Approved by:

Dr. Ronald R. Sass

Dr. James M. Conrad

Dr. Bharat Joshi

©2008
Ashwin A. Mendon
ALL RIGHTS RESERVED

ABSTRACT

ASHWIN MENDON. Design and Implementation of a Hardware Filesystem.
(Under the direction of DR. RON SASS)

The Reconfigurable Computing Cluster project is investigating the cost-effectiveness of using Platform FPGAs as the basic compute node in a petascale High-Performance Computing (HPC) cluster. Platform FPGAs are capable of hosting entire Linux-based systems including standard peripherals, integrated network interface cards and even disk controllers on a single chip. Filesystems, however, are typically implemented in software as part of the operating system. This presents a challenge as some applications are very sensitive to file I/O latency and Platform FPGA processor cores are clocked at relatively slow frequencies. This thesis describes the design and implementation of a filesystem in *hardware*. A hardware implementation offers several features that have potential of improving certain classes of applications.

The filesystem implemented is a simplified version of the well-known UNIX filesystem and specifically designed to handle a relatively low number of very large files. The design synthesizes but lacks a SATA host controller needed to test it. Instead, Modelsim was used to verify the functionality of four basic operations: open, read, write and remove. Synthesis results show that the core uses a modest 3% of the slices (and 3 BRAM blocks) of a Xilinx Virtex-4 FX60 device. By using a behavioral model of a SATA disk controller, sequential read bandwidth simulations achieved over 3 Gb/s for block sizes of 512 bytes. Higher bandwidths can be obtained using larger block sizes. Since physical disks are much slower, these results suggest that a hardware filesystem core offers several benefits with little cost and no loss of performance.

ACKNOWLEDGMENTS

I am deeply indebted to my adviser Dr. Ron Sass for his invaluable guidance, encouragement, understanding and help throughout the course of my masters thesis at The University of North Carolina at Charlotte.

I also would like to thank Dr. Bharat Joshi and Dr. James Conrad for agreeing to be on my thesis committee and for their time in reviewing my work.

I am grateful to people in the Reconfigurable Computing lab for spending time with me and answering all my technical questions related to my work in the lab.

Lastly I would thank my parents and family for their support.

TABLE OF CONTENTS

LIST OF TABLES	vii
LIST OF FIGURES	viii
CHAPTER 1: INTRODUCTION	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Organization	3
1.4 Contributions	4
CHAPTER 2: BACKGROUND	5
2.1 Filesystems	5
2.1.1 Filesystem Structure	6
2.1.2 Files	6
2.1.3 Block Allocation Methods	8
2.1.4 Disk Free Space Management	10
2.2 The UNIX Filesystem	10
2.2.1 File Structure	11
2.2.2 Disk Block Allocation	11
2.3 Software Reference Design	13
2.4 SATA disk interface	14
2.4.1 Features	16
2.4.2 Layers	16
2.4.3 SATA IP core interface	18
CHAPTER 3: DESIGN AND IMPLEMENTATION	20
3.1 SATA Behavioral Model	20
3.2 HWFS Core	24

	vi
3.3 Example Operations	25
3.3.1 Open File:	25
3.3.2 Read File:	27
3.3.3 Random Read(LSEEK):	30
3.3.4 Write File:	30
3.3.5 Remove File:	35
CHAPTER 4: EVALUATION	39
4.1 Experimental Setup	39
4.1.1 Synthesis	39
4.1.2 Simulation	40
4.2 Results and Analysis	40
4.2.1 Area	40
4.2.2 Performance	41
CHAPTER 5: CONCLUSION	48
REFERENCES	49
APPENDIX A: VHDL DESIGN ENTITIES	50

LIST OF TABLES

TABLE 4.1: Statistics for HWFS resource utilization with different blocksizes	41
TABLE 4.2: Subcomponent resource utilization statistics with a fixed blocksize	41
TABLE 4.3: HWFS Sequential Read Latency for different Block Sizes	43
TABLE 4.4: HWFS Sequential Write Latency for different Block Sizes	43

LIST OF FIGURES

FIGURE 1.1: Compound annual growth rate plot	3
FIGURE 2.1: Disk interface to operating system	7
FIGURE 2.2: Filesystem layout	11
FIGURE 2.3: UNIX inode structure	12
FIGURE 2.4: A Platform FPGA node in RCC cluster	15
FIGURE 2.5: SATA protocol layers	17
FIGURE 2.6: A SATA link layer frame	17
FIGURE 2.7: SATA IP core from ASICS world services	18
FIGURE 2.8: A cluster node showing SATA core interface to HWFS	19
FIGURE 3.1: Experimental Apparatus	21
FIGURE 3.2: Superblock structure	22
FIGURE 3.3: Empty disk	23
FIGURE 3.4: Sata Stub Behavioral	23
FIGURE 3.5: Hardware File System: State Machine and Components	24
FIGURE 3.6: Open File State Machine	26
FIGURE 3.7: HWFS Inode Structure	28
FIGURE 3.8: Read File State Machine	29
FIGURE 3.9: Random Read State Machine	30
FIGURE 3.10: Write File State Machine	33
FIGURE 3.11: Write File Example	34
FIGURE 3.12: Remove File State Machine	36
FIGURE 3.13: Remove File Example	37
FIGURE 4.1: Subcomponent slice utilization	42
FIGURE 4.2: Read Latency (in cycles) plotted against different file sizes	44

FIGURE 4.3: Write Latency (in cycles) plotted against different file sizes 45

FIGURE 4.4: File Read/Write Efficiency plotted against different file sizes 47

CHAPTER 1: INTRODUCTION

As Integrated Circuit (IC) technology advances, the programmable logic resources available on FPGA devices continue to grow as well. This allows, among other things, greater integration of computing systems. Indeed, it is now feasible to integrate network interface cards [8], disk controllers [13], and other conventional peripherals onto a single Platform FPGA device running Linux. These developments present an enormous potential for reconfigurable, high-performance computing — especially for out-of-core applications [7] and applications that need to stream very large data sets.

This thesis describes the implementation of a *hardware* filesystem. Filesystems are typically implemented in *software* as part of the operating system. The role of the filesystem is to organize the sequential fixed-size, block-addressable disk sectors into a collection of variable-sized, byte-addressable *files*. (Most modern filesystems also provide a hierarchy of directories to organize the files and many other features.) By moving this functionality into hardware, the proposed system gives computational accelerator cores implemented in the programmable logic of an FPGA direct access to the files on a disk. This has the potential of increasing the bandwidth from disk to core, lowering the latency, and reducing the computational load on the processor.

1.1 Motivation

For some scientific applications, the features mentioned above are extremely valuable. For example, in some cases, the resolution of experiment or simulation is limited by the main memory available to store the data structures. In order to increase the detail of the simulation, computational scientists are forced to code their algorithms so that data is explicitly moved between secondary storage and main memory. (These

so-called out-of-core applications are far more efficient than relying on the OS to swap virtual memory pages to disk.)

Alternatively, if part of the computation is performed by accelerators implemented in the programmable logic of an FPGA, then the data does not necessarily have to go through all of the traditional layers of an OS (device driver, filesystem interface) only to have the application inturn forward it to the core. Instead, the core can simply open the file and access it directly without buffering the data in main memory.

In a third case, the size of the dataset is growing faster than high-end computing speeds. Compare processor speeds and the size of bioinformatic databases. Suppose single processor performance continues to double every 18 months (a compound annual growth rate of 59%), the problem is that biological databases are growing even faster. Figure 1.1 shows both growth rates of between 1994 and 2004 on a semi-log graph. The nucleotide data points come from GenBank [6], a public collection sequenced genomes. A line fitted to this data shows a compound annual growth rate of 77%. Now consider the performance gains of I/O subsystems (disk and interface). Secondary storage is not keeping pace with processor speeds, let alone the growth rate of the biological databases. The most aggressive estimates [1] suggest a 10% compound annual growth rate in performance while others [4] suggest a more modest 6% growth rate. Regardless, the consequence is profound: the same question (e.g., *is this sequence similar to any known gene?*) will take longer and longer every year. In short, the problem size is growing so fast, the bottleneck is simply I/O bandwidth. A filesystem implemented in hardware will not directly address this issue. However, it is a first step towards multi-disk solutions are better handled in hardware.

1.2 Problem Statement

The central question we aim to answer in this thesis is whether it is feasible to implement a filesystem in hardware that improves disk to core bandwidth for data intensive compute cores. By feasibility, we mean the following questions are answered

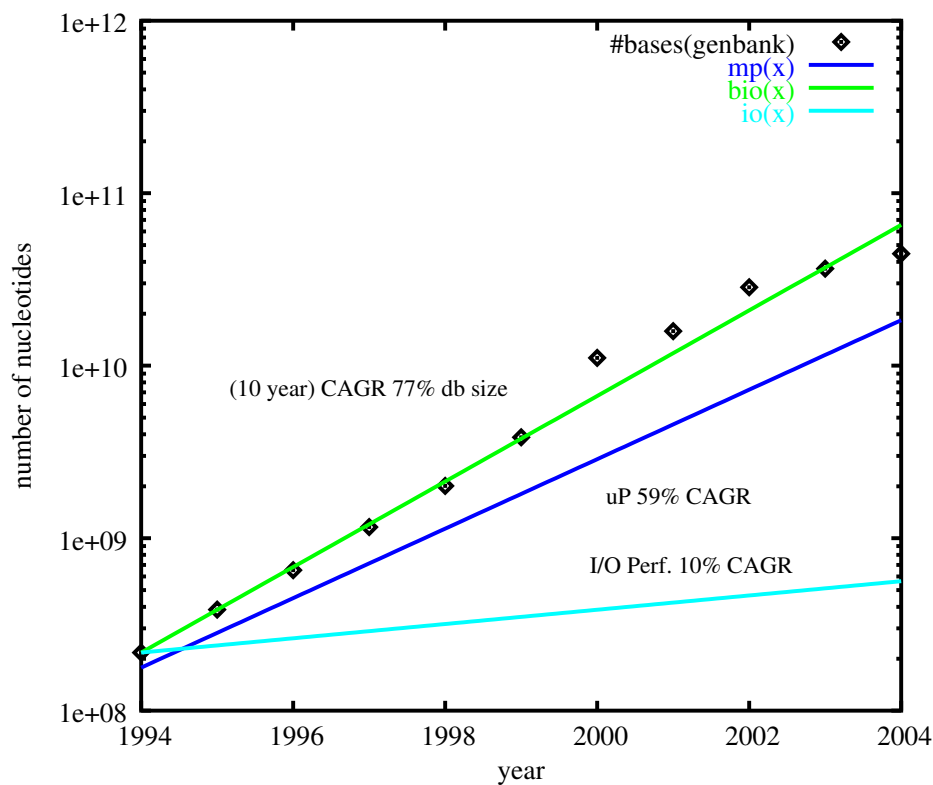


Figure 1.1: Compound annual growth rate plot

in the affirmative. a.) Does the hardware implementation provide the basic functionality of a traditional software filesystem? b.) Does the core give efficient run-time performance for large files? c.) Does the implementation consume reasonable FPGA resources?

1.3 Organization

The above question is answered by the following chapters. Chapter 2 describes the filesystem requirements and basic background technologies that are needed to implement a hardware filesystem. In Chapter 3, a set of design goals are enumerated followed by an overview of the proposed hardware filesystem. This design has been implemented in VHDL. Its functionality and performance have been tested with simulation and synthesis. These results are reported in Chapter 4. The thesis concludes with a brief summary and future directions.

1.4 Contributions

The specific contributions of this thesis are listed below.

- Developed a software reference design for the filesystem in C.
- Designed and implemented a synthesizable Hardware Filesystem core.
- Demonstrated functionality of basic file system operations: open, read, lseek, write and remove.
- Measured filesystem performance for block sizes ranging from 64 B to 512 B for 1 KB to 5 GB sized files.
- Quantified amount of FPGA resources needed for the design for block sizes ranging from 64 B to 4096 B.

CHAPTER 2: BACKGROUND

The chapter begins by describing the general structure of filesystems. It then moves on to compare different disk block allocation methods and free space management techniques used. Next, we focus on describing the UNIX file system which served as a reference for our design. Finally, the Serial ATA protocol and disk controller interface, which enables the filesystem to communicate with the hard disks, is explained.

2.1 Filesystems

The main purpose of a computer system is to create, manipulate, store, and retrieve data [11]. As such, filesystems have been central to all modern computing systems. Filesystems are responsible for managing and organizing files on a nonvolatile storage medium, such as a Winchester-type disk drive (commonly known as a hard disk or hard drive).

According to [11] the key functions of the file system are:

- Efficiently use the space available on the disk.
- Efficient run-time performance.
- Classify files for fast and easy retrieval.
- Perform basic file operations like create file, read, write and delete.

Of course many filesystems also provide more advanced features such as file editing, renaming, user access permission, and encryption.

Figure 2.1 illustrates the high-level relationships between an application and non-volatile storage. A filesystem serves as an abstraction between block devices such as

disks and applications which deal with character access. When an application program requests a file read, the Operating System asks the filesystem to open the file. The file system finds and reads the relevant block on the disk and delivers data to the operating system.

A hard disk consists of a set of spinning platters each divided into concentric circles called tracks. Each track is divided into arcs called sectors. A stack of tracks with equal diameters is a cylinder. The time needed for the drive to position the head to the proper cylinder is the *seek time*. The drive waits for the correct sector to move under the head called the *rotational latency*. The total time needed to get the data is the sum of the seek and the rotational delay called as *access time*.

2.1.1 Filesystem Structure

To allow data to be stored, located and retrieved conveniently from the disk, a file system involves designing two basic components: the user interface and the disk interface. For the user interface, the task involves creating an abstraction for the data to be stored called as a *file*, defining its attributes, operations allowed on a file and directory structure for organizing files. The logical file system is mapped onto the physical secondary storage device by means of algorithms and data structures.

The disk interface of a file system consists of the file data and metadata which is information about the file such as its location, type, size, and access permissions. The metadata is stored along with the file data on the hard drive.

The filesystem must be persistent in case of loss of power. Some information is cached in main memory to avoid expensive disk accesses. The cached copy must be kept consistent with the disk copy.

2.1.2 Files

Files are composed of bytes. The filesystem is responsible for implementing a byte-addressable file on a block addressable physical medium such as hard drive. The file

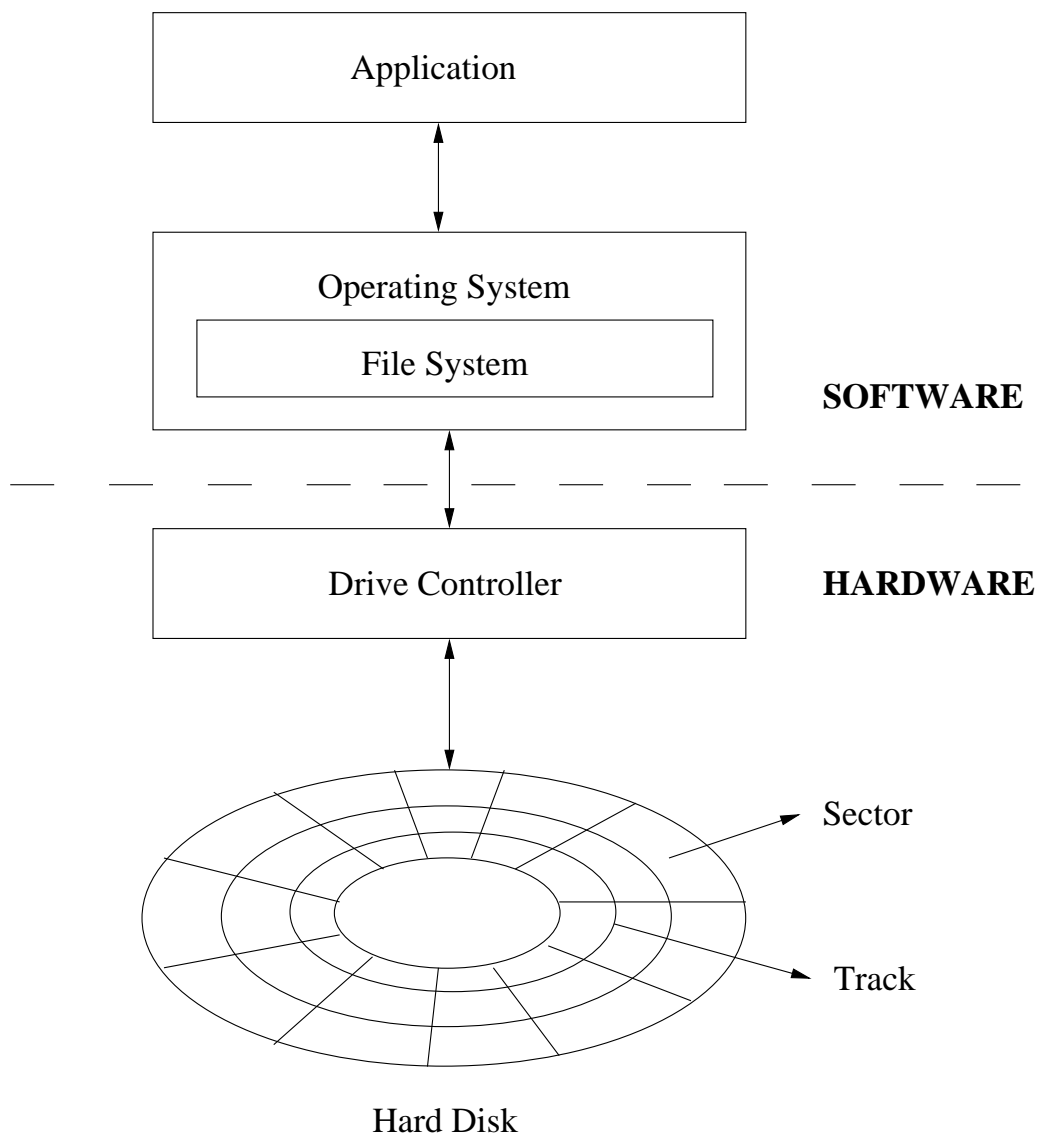


Figure 2.1: Disk interface to operating system

data may be text, numeric, executable programs, sound or graphics.

- **File contents:** Files are composed of one or more blocks stored on the disk. A block is a basic unit for data transfer between memory and disk. The size of each block is related to the sector size.
- **File meta-data:** Describes the characteristics of files such as file size, location of contents, last modified time and block of pointers to data. This information is kept in a directory or index block.

2.1.3 Block Allocation Methods

A filesystem should allocate space for the file making efficient utilization of disk space and providing quick access to the file. This manifests itself in the block allocation method. The major file allocation methods are continuous, linked, and indexed allocation [11]. Each method has its advantages and disadvantages.

Continuous allocation: Each file is a group of continuous blocks on the disk. A file of length n requires n continuous blocks. Hence, number of disk seeks needed to access a file is minimal. This method provides random access to blocks and only start of file and file length information are needed to access a file. Making space for a new file becomes difficult as the OS must search for n free continuous blocks for storing a file of length n . Such a strategy does not allow for simple expansion and contraction of files in the file system without running the risk of fragmenting free storage area on the disk. As files are allocated and deleted, the free disk space is broken into little pieces scattered over the disk. Enough free space is available to service a request but this is not continuous. Such a condition is called external fragmentation.

Linked allocation: Each file is a linked list of disk blocks which can be placed anywhere on the disk. Each block contains a pointer to the next block. A

file of four blocks starts at block 5, continues at block 10, then block 15 and finally block 26. . This strategy avoids external fragmentation since a free block located anywhere on the disk can be used to satisfy a request. While writing to a file the length need not be specified in advance. A file can continue to grow as long as there are free blocks available. The directory contains location of first and last blocks of the file. Only starting block is needed to access the file. Due to the linked list structure, only sequential access is possible. To find the n^{th} block of the file, disk seek must start at the beginning of file and follow the pointers until n^{th} block is reached. Another drawback is poor reliability. One bad sector can result in broken links and the file information will be lost permanently.

Modified Linked Allocation: This type of allocation is used in Windows-based machines. The disk keeps an area reserved for the File Allocation Table (FAT). The FAT table has a entry for every block on the disk. The table keeps track of which blocks are available and which are in use. Free blocks are marked as 0. For blocks belonging to a file, every entry in the FAT gives the entry for the next block in the file. The last block of every file contains an end marker.

Indexed Allocation: This strategy is a solution to the problems of both continuous and linked allocation. Each block of the file has a pointer to it. These pointers are grouped in an index block and stored on the disk. The directory contains the location of the index block. The i^{th} entry in the index block points to the i^{th} block (sector) of the file providing random access. To read the i^{th} block, the pointer in the i^{th} entry of the index block is read to find the desired sector. The disadvantage is that large files need bigger index blocks which adds to the file overhead.

2.1.4 Disk Free Space Management

Since disk space is limited, the file system needs to keep track of free space on the disk and reuse space from deleted files for new files. For this purpose it maintains a free space list which records all disk blocks that are not allocated to some file or directory. When a file is created, the blocks are taken off the free list and assigned to the file. On deleting a file, the free blocks are added back to the list.

Linked list of disk blocks: In this approach the free blocks are linked together by a pointer in each block. The pointer to the first block is kept at a special location on the disk. This method is inefficient as a disk operation is needed to allocate every new block.

Bit Vector: This scheme uses a string of bits to represent disk blocks. Free blocks are denoted by 1 and used blocks by 0. It is convenient to locate blocks at any position on the disk.

Linked list of address blocks: Another approach is to store the pointers to free blocks in an address block. Every address block has $(n - 1)$ free blocks, and a pointer to the next address block. Thus addresses of a large number of free blocks can be found quickly.

After researching the architecture of the existing filesystems, we chose to model our design by adapting the UNIX filesystem (UFS).

2.2 The UNIX Filesystem

The UNIX file system consists of a sequence of homogeneous logical blocks each containing 512, 1024, 2048, etc. (A logical block may consist of multiple disk sectors.) Using large logical blocks increases the data transfer rate between the disk and main memory. The file system layout is as shown in the figure [3].

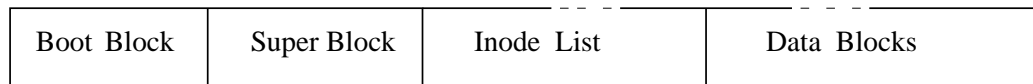


Figure 2.2: Filesystem layout

boot block: contains bootstrap code read into the machine to boot and initialize the operating system

super block: describes state of the filesystem such as blocksize, filesystem size, number of files stored and free space information

inode list: this is a list of indices to data blocks

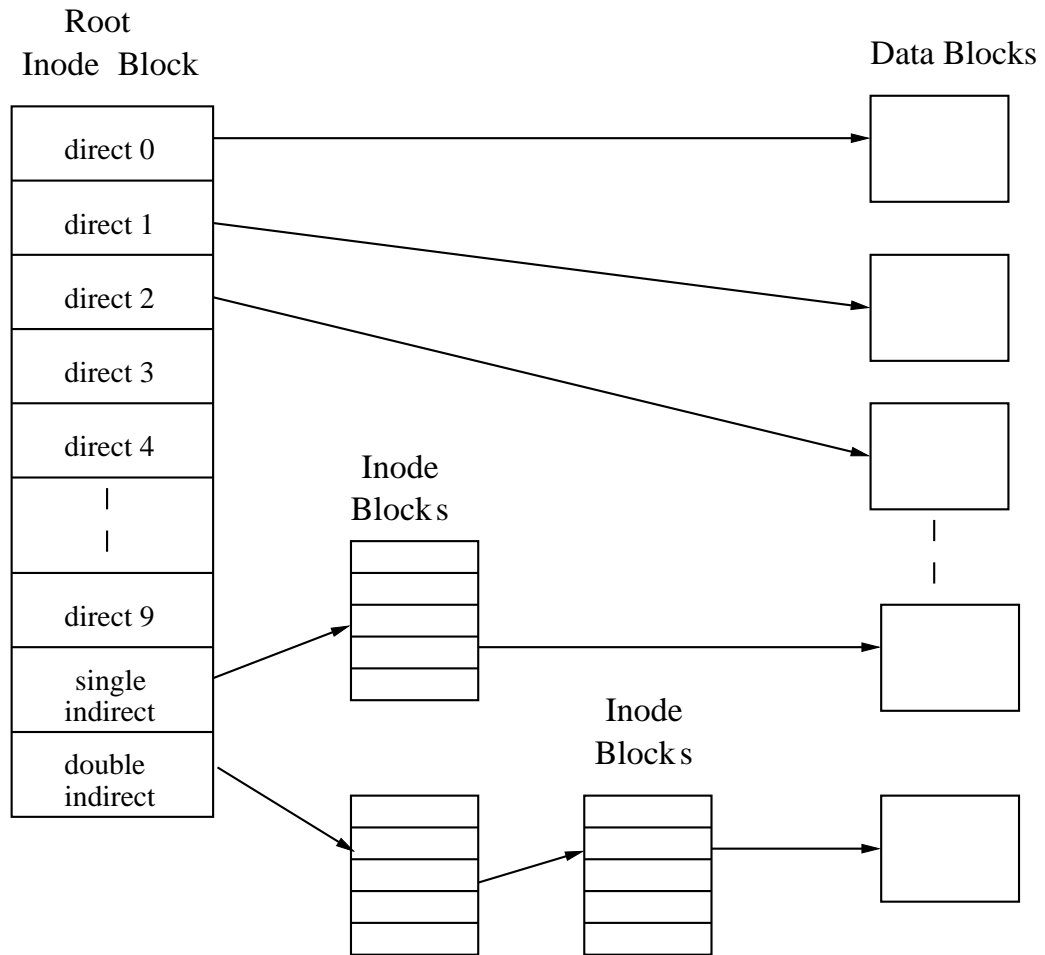
data blocks: contain actual file data and administrative data

2.2.1 File Structure

UNIX uses a hybrid multilevel indexed allocation scheme. The first i blocks in the file are from a direct index, next j blocks are from an indirect index and next k blocks are from a double indirect index. This keeps the inode structure small yet allows large files. The direct blocks contain block numbers of the data blocks. The indirect block points to a block of direct block numbers. Similarly the double indirect block contains a list of indirect block numbers.

2.2.2 Disk Block Allocation

UNIX filesystem keeps a linked list of address blocks on the disk as described in Subsection 2.1.4. Before writing a file, the disk needs to be searched for free blocks. However, frequent disk accesses are time consuming. Therefore, an array is maintained in memory that caches the block numbers of free disk blocks in the file system. The blocks of the filesystem are arranged in a linked list on the disk, such that each link of the list is an array of free disk block numbers. While writing data to disk, free blocks are assigned from the freeblock array in memory and used to store inodes and data blocks of a file. This block cannot be reallocated until it becomes



Direct and Indirect Inode Blocks

Figure 2.3: UNIX inode structure

free. The last freeblock in the memory array is treated as a pointer to a block that contains a list of free blocks. The memory list is refilled by free blocks from the disk. When a file is deleted, the disk blocks are returned to the memory array and stored on disk.

2.3 Software Reference Design

Before designing the filesystem in hardware, we created a software reference design, by adapting the UNIX filesystem to our requirements, keeping simplicity and brevity in mind for our prototype work. Our goals are specifically to support the RCC project [8]. The proposed filesystem core needs to support relatively few, very large files. Ownership, access control, and even the concept of subdirectories are not very important. Fast sequential access is essential and reasonably fast random read access is important. Our Filesystem only uses direct and single indirect pointers. However, the indirect logical blocks include a pointer to another indirect block — essentially reverting to a linked-list structure for very large files. The UNIX filesystem provides features such as file editing, renaming, copying, file compression, encryption, access permission etc. Our filesystem needs the basic functionality such as creating a file, opening and writing to it, reading from it and deleting a file. These are incorporated into the four modules `mkafs`, `writefile`, `readfile` and `removefile` which are explained below.

mkafs This program creates one empty UNIX file *virtdisk.bin* which represents a virtual disk with free blocks and initial metadata. It then forms a linked list of free disk blocks called as *freelist*. Each node in the list stores a block identification number which is the disk location of the block. The start of freelist is stored at disk block 0 in a structure called *superblock*. The file name to inode location mapping is also included in the superblock.

writefile Writefile begins by taking filename from the command line, filelength from

standard input and storing them in the superblock data structure in memory. The freelist is recreated by reading the start of freelist stored in the superblock. Free blocks are taken off the list and passed to the writeblock function to write data to the disk. On completion of the writing process, the freelist is updated by removing the nodes containing locations of used disk blocks. The new start of freelist is then written to the superblock and stored on the disk file.

readfile Readfile takes file name from the command line and uses linear search to find the inode location and filelength from the superblock. It then calculates the number of direct and indirect pointers needed for reading the file from the file length. The inode block is accessed next and the pointers are read and stored in a linked list. The linked list is traversed node by node to read the data blocks of the file.

removefile The purpose of removefile is release the space allocated for the file and invalidate the directory entry. The free space can be used for writing other files. The filename obtained from the command line is used to get the inode location information from the superblock. The filename is then deleted. The filelength read from the superblock is then used to compute the number of pointers needed to access the file. The start of freelist is read from the disk to recreate the freelist. The file's inodes are returned to the correct location in the freelist.

Normally, the filesystem is designed to be independent of the disk controller. For expediency, we have focused on the most common, commodity drives available today: Serial ATA (SATA).

2.4 SATA disk interface

The Reconfigurable Computing Cluster has 64 Xilinx ML410 boards [8]. Eight of the high-speed multi-gigabit transceivers [17], available on the ML-410 boards, are

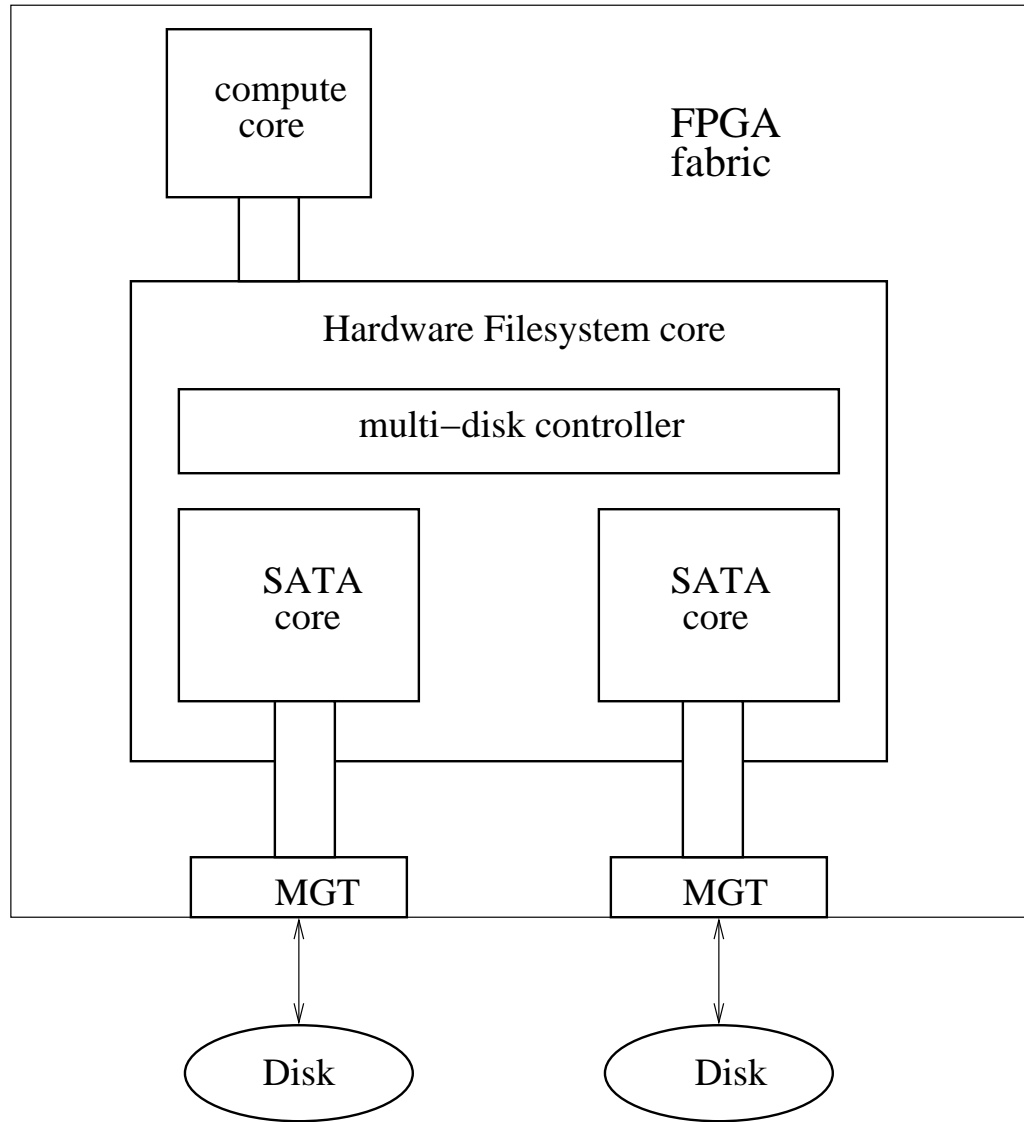


Figure 2.4: A Platform FPGA node in RCC cluster

dedicated for node-to-node communication. This leaves two MGTs to attach the high speed SATA disks. A commercial SATA IP core will be purchased at a later stage to drive the high speed RocketIO Multi-Gigabit Transceivers (MGTs) [14]. The MGTs can be configured to communicate via the SATA protocol at the physical layer. The Serial ATA bus is used for high speed data transfer between the MGTs and the disks.

Serial ATA was designed to overcome a number of limitations of parallel ATA. It has the following improved features [10, 9].

2.4.1 Features

SATA provides a 4-wire point-to-point configuration, supporting one device per controller connection. Each device gets a dedicated bandwidth and there are no master/slave configuration jumper issues as with parallel ATA drives. The pincount is reduced from 80 pins to 7 pins. Three ground lines are interspersed between four data lines to prevent crosstalk. Also, smaller cables result in less clutter, better routing and improved airflow.

The layers in the Serial ATA architecture are described below.

2.4.2 Layers

The Serial ATA function is divided into four layers as shown in Figure 2.5. The physical layer handles high speed serial communication between host and device. The Link and Transport layers control the overall operation. The application layer deals with ATA commands.

Physical Layer The Serial ATA achieves speeds of 1.5Gb/s and beyond by using a high speed clock (1500 MHz) and low voltage (250 mV) differential signaling. Each data signal is transmitted over two lines which carry equal and opposite versions of the signal. The receiver decodes the differential voltage between signals and eliminates crosstalk. The PHY layer provides Out of Band Signaling (OOB), incorporates serializer/deserializer and handles power-on sequencing and speed negotiations.

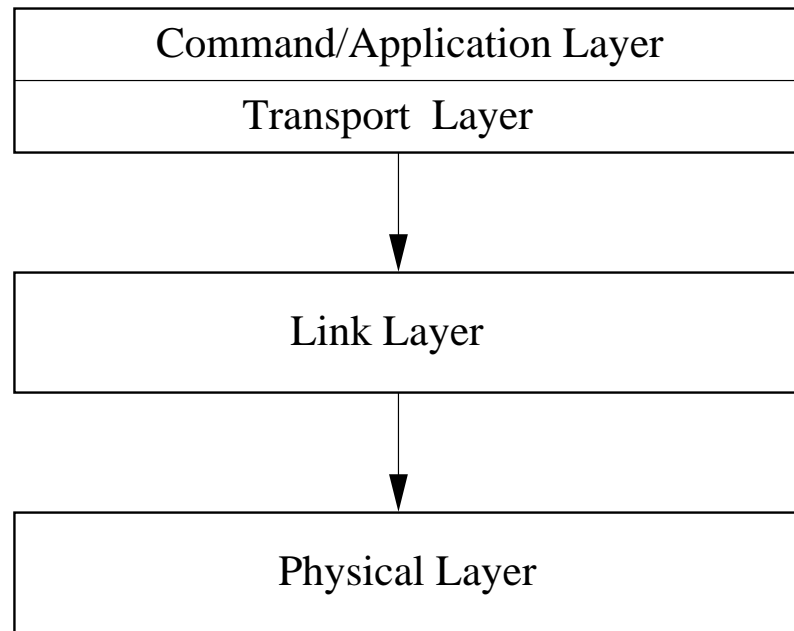


Figure 2.5: SATA protocol layers

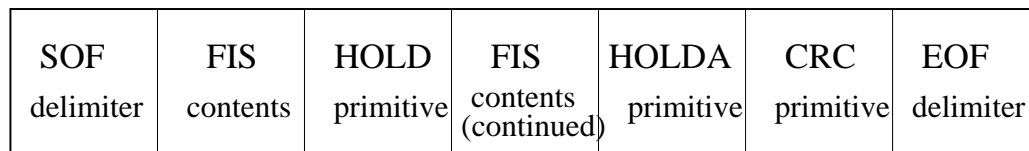


Figure 2.6: A SATA link layer frame

Link Layer The Link Layer is responsible for sending and receiving frames (Figure 2.6), control signal primitives and performing flow control. It takes the Frame Information Structure (FIS) from the transport layer and adds CRC, SOF and EOF delimiters, scrambles data and performs 8B/10B encoding before transmission. In the opposite direction, it checks the CRC and cleans the packet from primitives. It signals the good reception or link/PHY layer errors to transport and peer link layer.

Transport Layer The Transport Layer handles construction and deconstruction of the Frame Information Structure (FIS) at the application layer's command. It also manages FIFO or buffer memory for controlling data flow. In the transmission direction, data is filled in the FIFO through DMA transfer or PIO transfer and data FIS transmission is initiated. In the receiver direction, the data is delivered to the

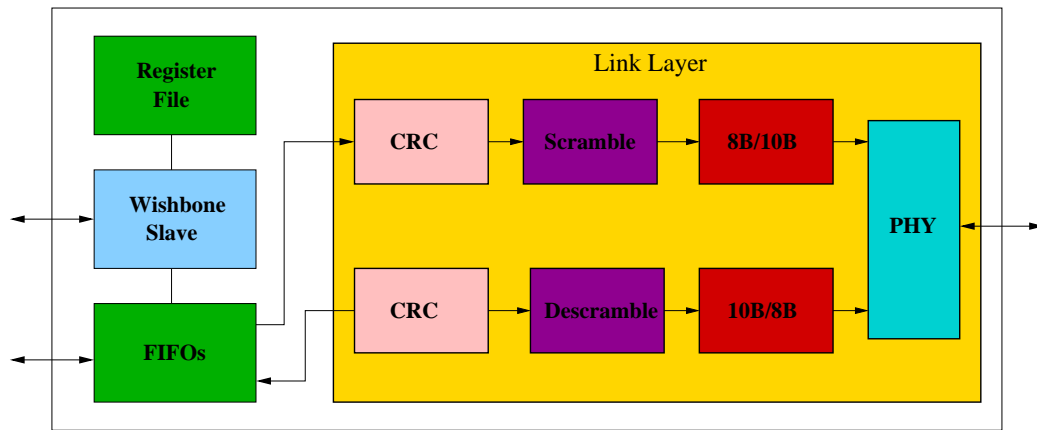


Figure 2.7: SATA IP core from ASICS world services

transport layer which detects FIS type and decomposes it. FIS type is indicated by the Frame Information type field located in byte 0 of first Dword of payload.

Application Layer The Application layer is responsible for generating the sequence of legacy ATA commands [12] and handles the status and control block of registers and the shadow register block.

2.4.3 SATA IP core interface

The Serial ATA host IP controller core from ASICS World Services [2] is shown in Figure 2.7.

This disk controller core will be interfaced with our hardware filesystem (described in the next chapter) for providing high speed data access to the compute cores as illustrated in Figure 2.8. The embedded IBM Power PC processor [15] in the FPGA chip will control the filesystem using the Device Control Register (DCR) bus. The filesystem issues commands to the shadow block registers in the SATA core using the Wishbone slave interface. These registers generate the ATA commands to communicate with the device controller core on the disk side. Data (reads and writes) is transmitted from the SATA core's data FIFO to locally-developed read/write FIFOs again using the Wishbone slave interface. These FIFOs directly interface with the compute core. Such a setup has the potential of providing the large datasets directly

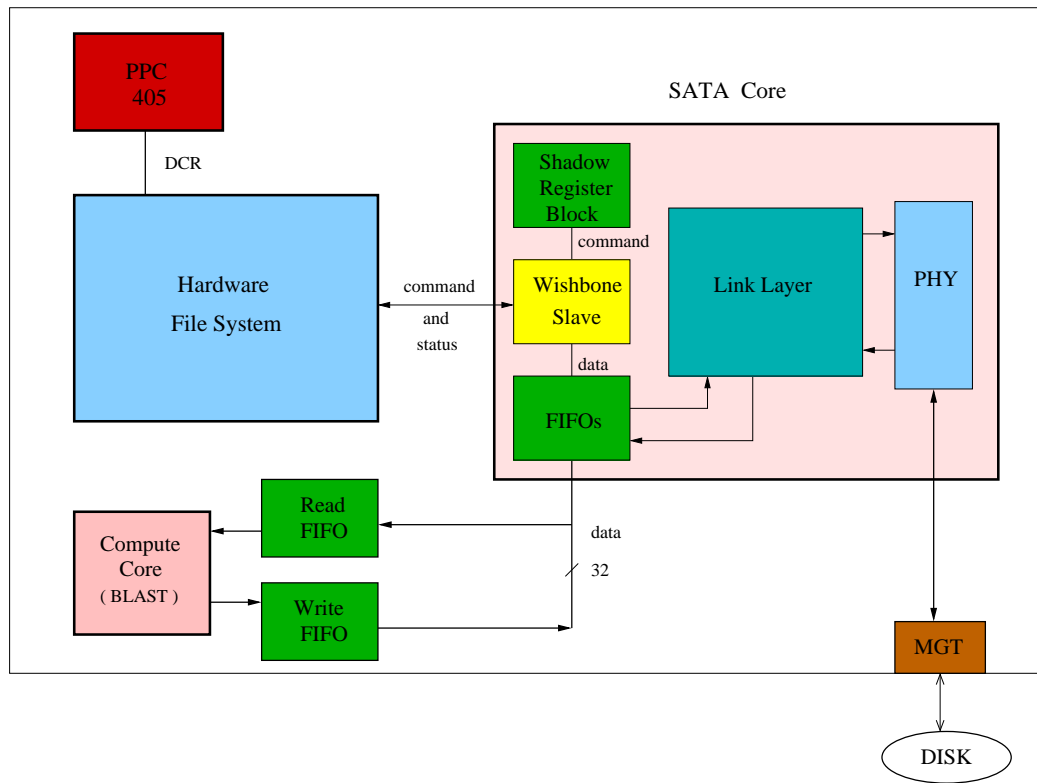


Figure 2.8: A cluster node showing SATA core interface to HWFS

to the compute cores.

CHAPTER 3: DESIGN AND IMPLEMENTATION

To evaluate the feasibility and performance of a hardware filesystem core, it was necessary to set up an experimental apparatus. Since we were designing the filesystem and a hardware implementation, we chose spiral development method to create a specification and manage risk. First, a software reference design was created. This provided a detailed specification of the filesystem layout and a reference for each operation. Next, a behavioral model of a SATA Host controller was created. Following that the HWFS was implemented along with a test bench. The last step was to modify the VHDL for synthesis. Figure 3.1 shows the three main components: a testbench to exercise the hardware filesystem, the hardware filesystem itself, and a behavioral model of a SATA host controller.

3.1 SATA Behavioral Model

Although SATA host controllers exist [13], the core is expensive and not required to test the feasibility of a hardware filesystem core. Thus our approach was to create a simple behavioral model of the SATA host controller, *satastub* (Figure 3.4), which interfaces the filesystem with the hard disk.

First, `mkafs.vhd` is run in a simulator to create an empty filesystem which is written to a local disk file on the simulation workstation (`virtdisk.bin`). This includes the *SuperBlock* metadata and an interleaved linked freelist of all the free disk blocks. The *SuperBlock* consists of the following fields:

- location of the first address block in the freelist,
- the index of the next free blockid in the freelist head,

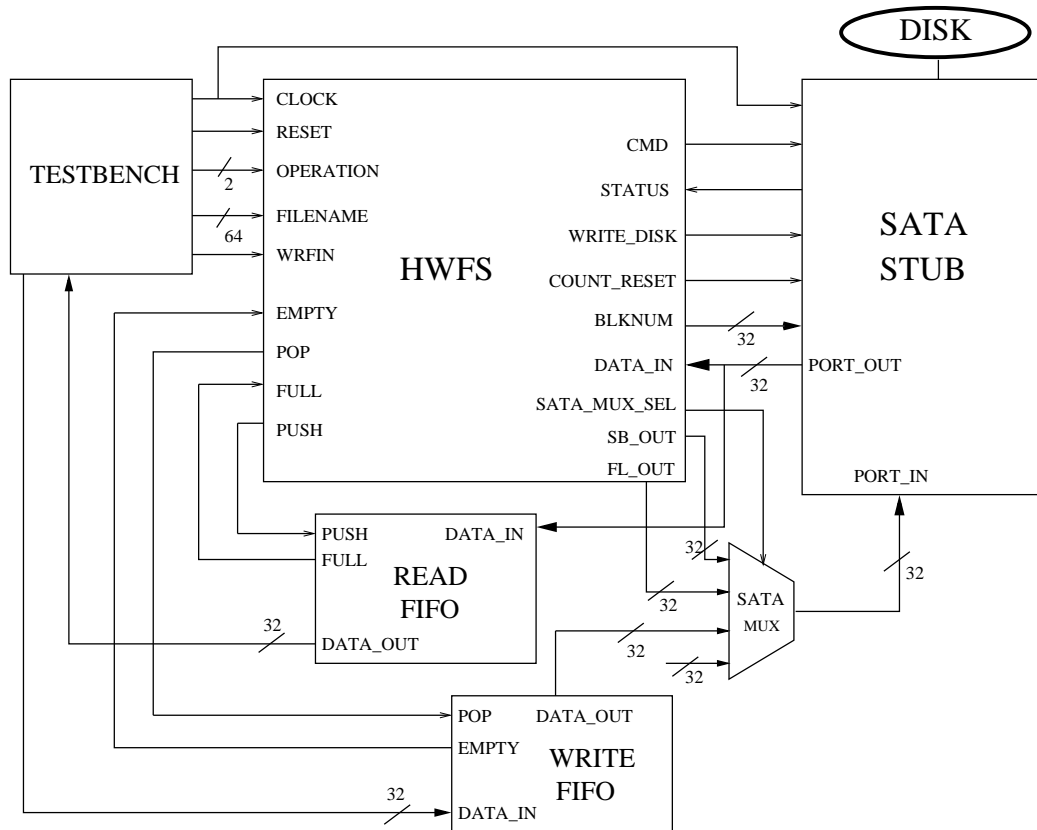


Figure 3.1: Experimental Apparatus

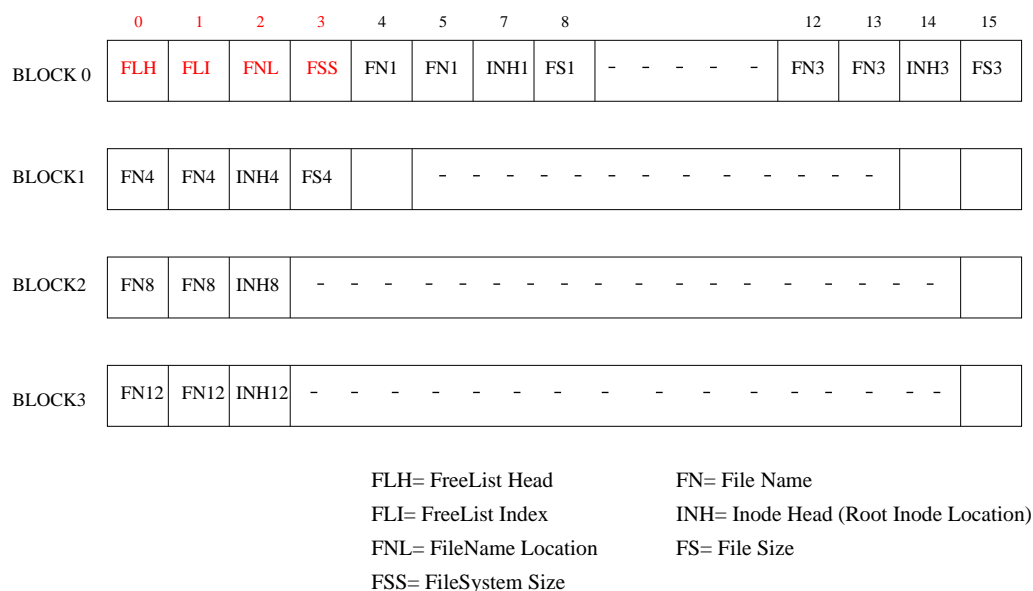


Figure 3.2: Superblock structure

- the location of the filename free slot in superblock,
- file system size,
- empty slots for storing filenames and their root inode locations.

In our current implementation, the *Superblock* is 4 blocks wide and can store upto 15 files (for a 64 B block). This is easily scaled up if the filesystem needs to accomodate more files.

The freelist holds the logical block numbers of all the freeblocks in the file system. These address blocks are interleaved among the free disk blocks in the form of a linked list. Figure 3.3 shows an empty disk having 100 blocks each 64 Bytes wide. The first four blocks (0 to 3) are reserved for the Super Block. The freelist head which is the first address block of the freelist is at block 4, the subsequent address blocks are located at offsets of integer multiples of 16 from the freelist head.

During simulation, the SATA stub opens the empty disk file `virtdisk.bin` and loads it into a 2D array in memory. The dimensions of the array are defined by the disk size and block size. The column indices are used as logical block numbers of the

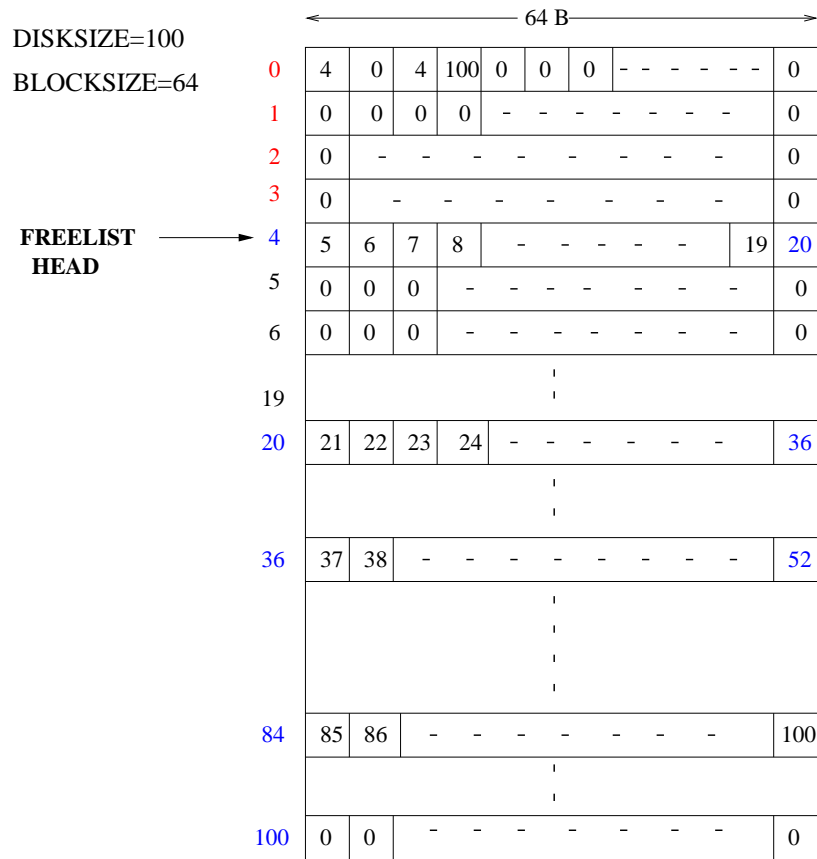


Figure 3.3: Empty disk

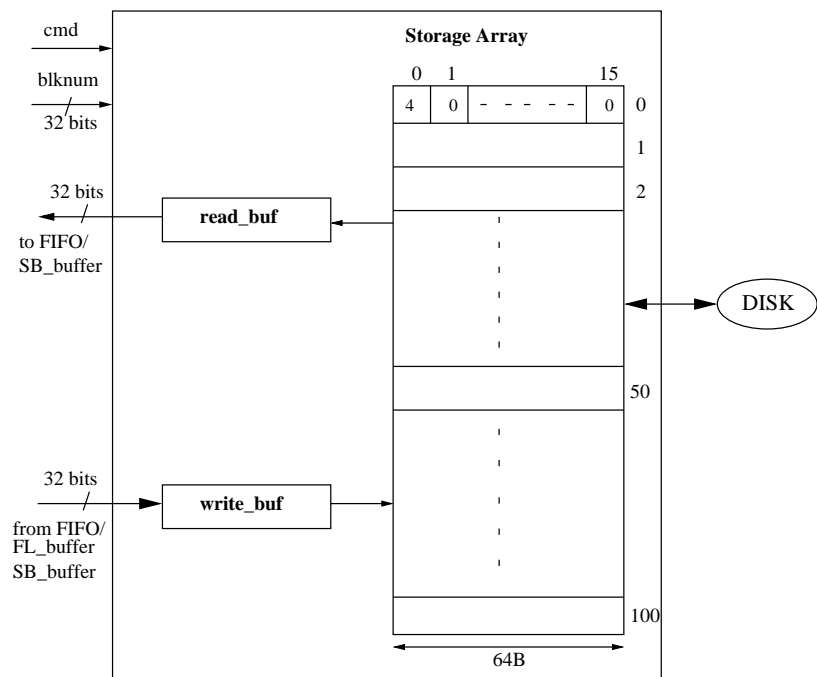


Figure 3.4: Sata Stub Behavioral

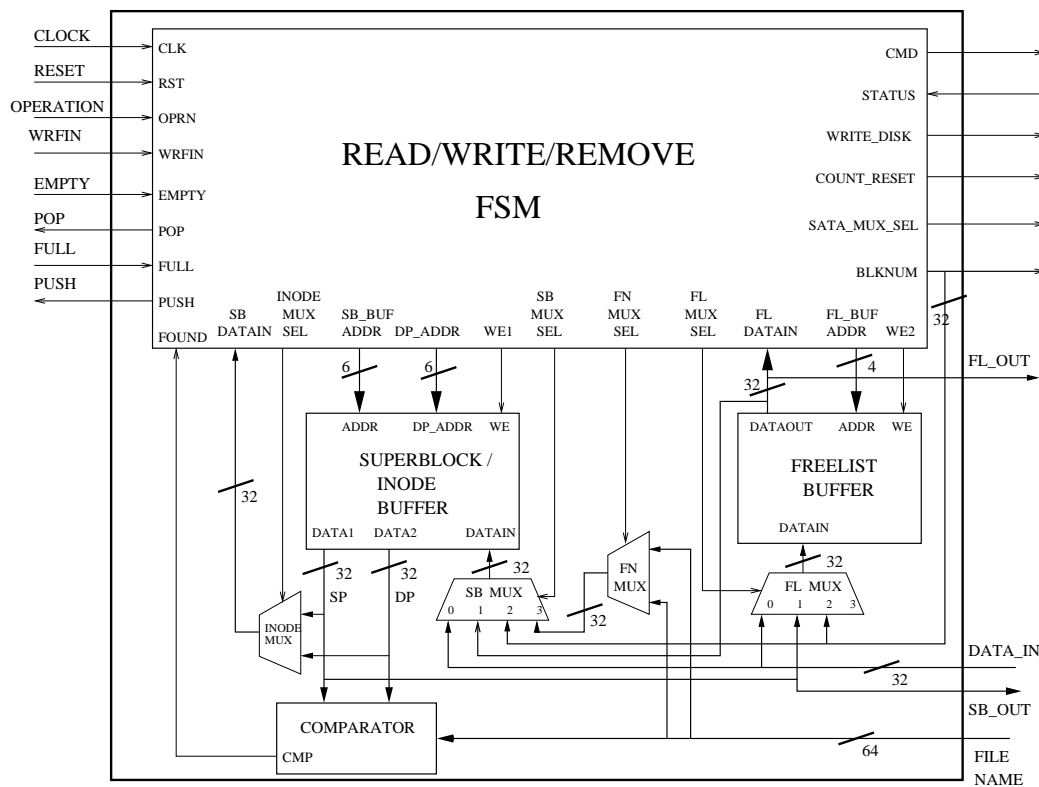


Figure 3.5: Hardware File System: State Machine and Components

disk. Depending on the command and block number received from the controller state machine, disk blocks are copied to and from read and write buffers. These buffers interface to the 32-bit input and output ports of the SATA stub. We did not attempt to model the actual characteristics of a disk drive; data is immediately available when the HWFS requests it.

3.2 HWFS Core

The Hardware FileSystem (HWFS) core consists of controlling State Machine and a Datapath as illustrated in Figure 3.5. The data path consists of two Block RAMs, one Comparator, three 32-bit 4:1 Multiplexers and two 32-bit 2:1 Multiplexers. It interfaces to the SATA host controller and, in our example, the testbench. The latter interface is simple enough that it can be directly connected to a computational accelerator or attached to a bus.

The state machine implements the Open, Read, Write and Remove file operations.

The 2-bit *operation* port is driven by a testbench to select from the 4 operations. The FSM issues an internal *command* signal and a 4-byte *blockid* to read from or write to the appropriate location in the *satastub* array. The *satastub* sends out the block 4-bytes at a time through *portout*. After completing a block transaction, it asserts the *status* signal. *WriteDisk* signals *satastub* to write the entire 2D array to disk file.

Block Rams are used as buffers for storing the *SuperBlock*, *InodeBlocks* and the *FreeListBlocks*. The superblock buffer, which is 4 blocks wide to accomodate a 4 block SuperBlock, is dual ported to allow for 8-byte filename comparison and also in checking for an end of file primitive. Blockids are transferred between the two buffers for the purpose of creating inode blocks in *writefile* and returning inodes to freelist in *removefile*.

The 64-bit equality comparator is used for finding a match between the testbench filename and the superblock filename.

The 32-bit 4:1 superblock mux is used to choose the input from either the sata stub, the freelist buffer, the blockid from FSM or the filename multiplexer.

The freelist mux takes in the input from either the sata stub, the superblock buffer or the blockid from FSM. The sata mux is used to select from either the superblock buffer, the freelist buffer or the write FIFO.

3.3 Example Operations

The Mealy-type controlling state machine was implemented to handle all four filesystem operations: *openfile*, *readfile*, *writefile*, and *removefile*. It has 17 states in all, some of which are common between read, write and remove operations.

The following sections will elaborate on the filesystem's functioning.

3.3.1 Open File:

Open File takes in a filename alongwith an open command and returns the file's root inode location.

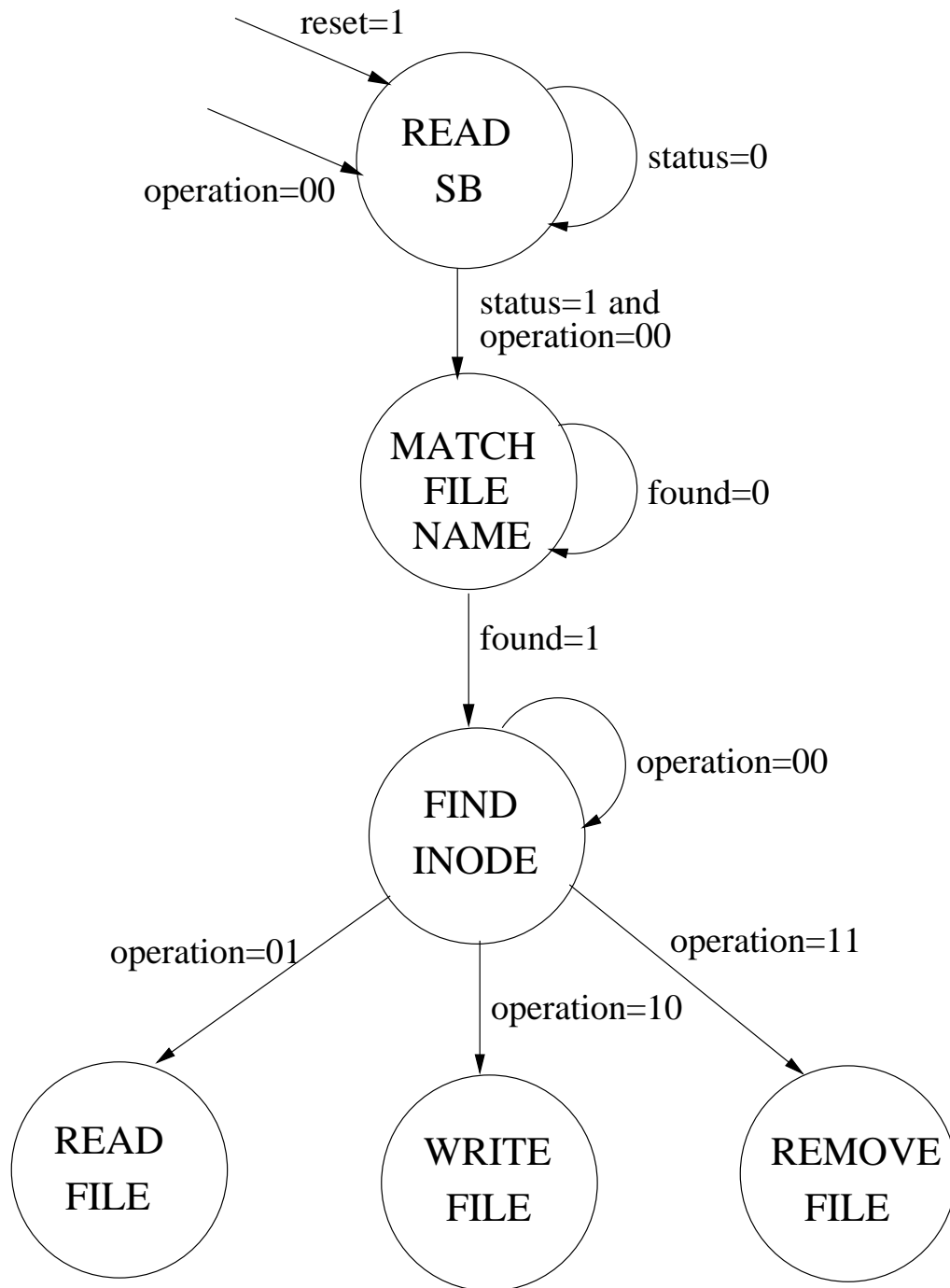


Figure 3.6: Open File State Machine

A file name and an open operation command (encoded as “00”) is sent to the controller state machine which cycles through the following states.

Read Super Block: The state machine starts from the *Read Super Block* state and reads blocks 0-3 from *satastub* into the *SuperBlock* buffer 4 bytes at a time. After reading 4 blocks, the state machine transitions to the *Match Filename State*.

Match File Name: Once the SuperBlock is read into the superblock buffer, match filename starts a linear search for the 8 byte filename. The FSM sequences through the superblock buffer, starting at the first file slot and reads the BRAM contents into a 64-bit equality comparator. If a match is found with the required filename, the comparator sends a *found* signal to the FSM which transitions to the *Find Inode* state. If the search is unsuccessful, the file does not exist in the filesystem superblock and a *flenotfound* signal is asserted.

Find Inode: In find inode state, the state machine gets the file’s root inode location from the filename-inode mapping in the superblock.

3.3.2 Read File:

Once the file is opened, Read File uses the file’s root inode block location to read in the file contents into the Read FIFO. Our Filesystem only uses direct and single indirect pointers as illustrated in Figure 3.7. However, the indirect logical blocks include a pointer to another indirect block — essentially reverting to a linked-list structure for very large files.

Read Inode: In *Read Inode* state, the root inode block of the file is first fetched from *satastub* into the SuperBlock/Inode buffer using the inode location number as a blockid.

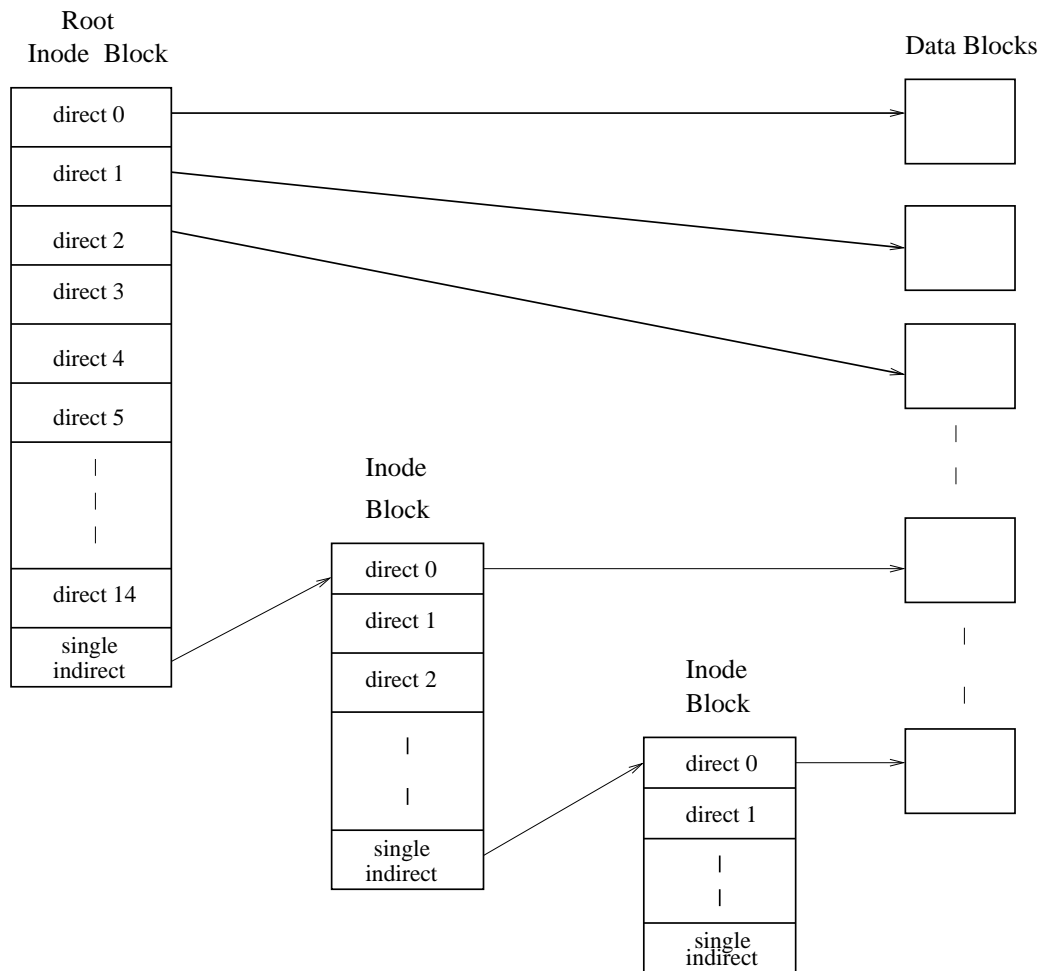


Figure 3.7: HWFS Inode Structure

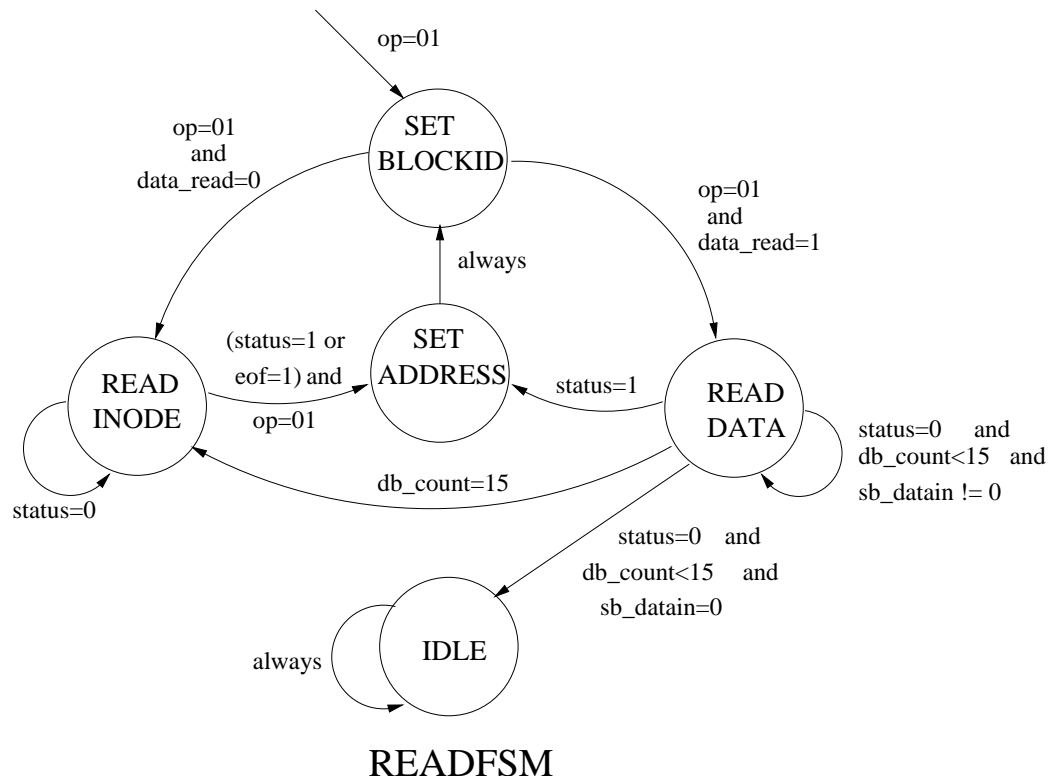


Figure 3.8: Read File State Machine

The state machine then transitions to the *Read Data* state via two intermediate states: *Set Address* and *Set Blockid* which are used to account for the delay between setting the BRAM address and reading the output blockid.

Read Data: In *Read Data* state, the inode block in the superblock buffer is read sequentially from the inode buffer, 4-bytes at a time. The output inodes are fed back to the state machine and used as blockids for fetching data blocks from the *satastub*.

The last inode in the root inode block links to the next inode block of the file. The FSM uses this link inode for fetching the next inode block of the file by cycling back to the read inode state.

The data blocks are read till an inode 0 is found which signals the end of file. The state machine then goes to the *idle* state.

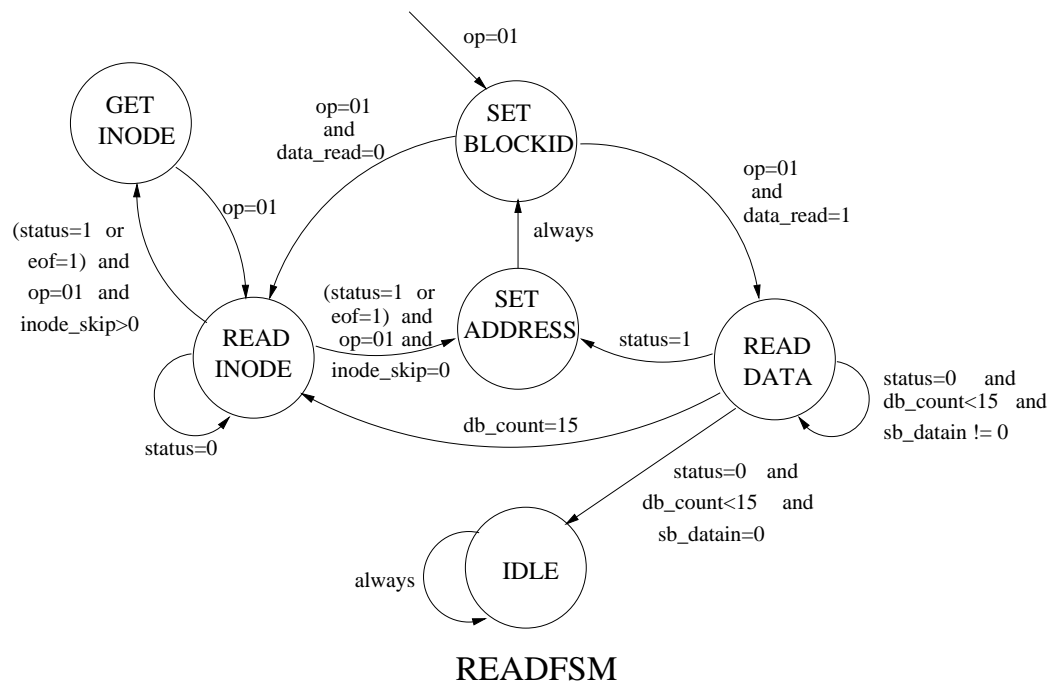


Figure 3.9: Random Read State Machine

3.3.3 Random Read(LSEEK):

In addition to the read command, a block offset is given to the state machine from the testbench. Using the root inode block as the starting point, the state machine seeks to the appropriate inode block of the file using an intermediate state: *Get Inode*. It then seeks to the calculated block offset of the file and starts reading the data blocks by cycling between *Read Data* and *Read Inode* states, till the end of file.

3.3.4 Write File:

Write File either creates a file if it does not exist on the disk or appends data to an existing file.

Read Address Block: In *Read Address Block* state, the *freelisthead* which points to the start of the freelist on disk, obtained from the superblock is used as blockid to fetch the first address block of the freelist into the freelist buffer. The freelist index points to the first available free blockid in the address block. If the file exists in the SuperBlock the FSM transitions to *Read Inode* state, else it goes

to *Write Data Block* state.

Read Inode Block: In *Read Inode*, the inode blocks of the file are read one by one till an inode 0 is reached, indicating the end of file. The FSM goes to *Write Data Block* state.

Write Data Block: File data is written to free blocks on the disk. The FSM picks up a free blockid from the freelist buffer and uses it as a blockid to write a block of data from the *Write FIFO* to *satastub*. The blockid is also simultaneously stored as an inode in the inode buffer. The data blocks are written continuously either till the address block is exhausted or the inode buffer is full, whichever happens earlier.

If the address block in freelist buffer is used up then the FSM goes back to the *Read Address Block* state to fetch the next block of the freelist using the last node in the freelist buffer as a blockid. If the inode buffer is full (filled upto the penultimate slot) the FSM transitions to the *Link Node* state to store the next free blockid in the last slot. This blockid is used to store the next inode block of the file on disk, forming a linked list of inode blocks. It then goes to *Write Inode Block*.

Write Inode Block: The inode buffer's contents are written to the *satastub* and a root inode block for the file is created. As the filesize increases, subsequent inode blocks of the file are created and added as a linked list to the root inode block. The file's inode blocks thus exist in the form of a linked list interspersed among the data blocks and freelist blocks.

The state machine cycles back and forth between *Write Data* and *Write Inode Block* states till the required amount of data is written to sata stub. Then it goes to write freelist state.

Write Freelist: In write freelist, the unused free blockids from the freelist buffer are written back to *satastub* and the FSM goes to *Read Super Block* state to load the superblock in the inode/superblock buffer.

Write Metadata: The filesystem metadata such as the location of the freelist head, the freelist index, the empty filename location and the file metadata which is the filename and its root inode location are written to the superblock buffer.

Write Super Block: Finally, the contents of the superblock are stored back to blocks 0 to 3 in sata stub.

Set Address and *Set BlockID* are used as intermediate states to account for BRAM read delay.

Figure 3.11 describes the process of writing a 1 kB file on an empty disk of hundred 64 byte blocks. The super block and the first two address blocks are shown in Figure 3.11(a). Sixteen free blocks, each 64 bytes wide are needed to write a kilobyte of data. Address block 4 gives the first 15 free block numbers for writing data. These block numbers are assigned to data blocks and simultaneously stored in an inode array. The last node (node 20) in address block 4 is a pointer to the next address block of the freelist. The first free block number from block 20 is used to store the inode array which becomes the root inode block of the file as shown in Figure 3.11(b). The second free node in block 20 (node22) is stored in the last slot of the root inode. This serves as a pointer to the next inode block of the file. Free block 23 serves as the last data block of the file and is stored in inode block 22. An end of file primitive, inode 0, is added in the last inode block of the file. The superblock is updated with the new freelist head (block 20), freelist index (3), the file name (12) to root inode mapping (block 21) and the filesize (16) in blocks. Figure 3.11(c) shows the modified superblock.

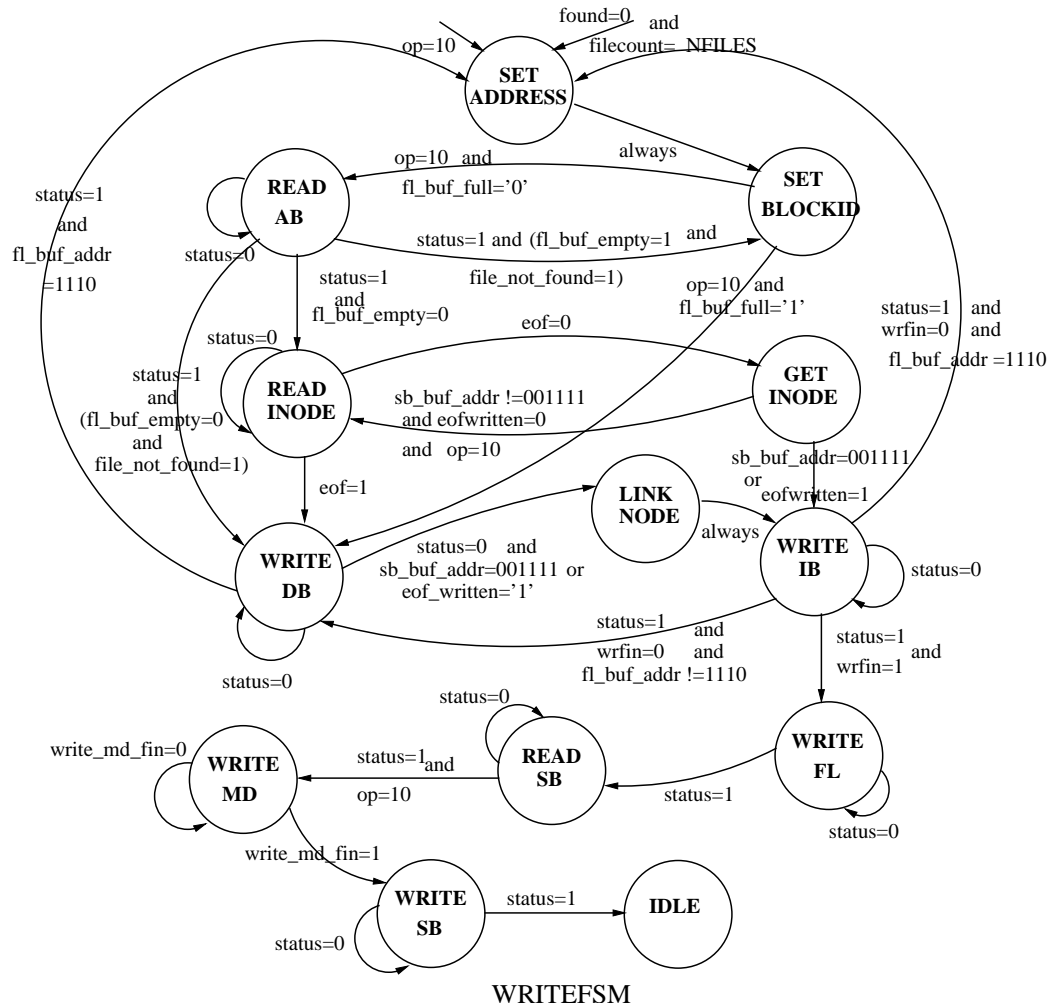


Figure 3.10: Write File State Machine

3.3.5 Remove File:

Read Address Block,Read Inode Block: The first address block from the freelist is read into freelist buffer and the root inode block is read into the inode buffer.

Return Inode: In the *Return Inode* state, the blockids from the inode buffer are transferred to the freelist buffer starting from the freelist index, till the freelist buffer is full of free blockids.

Write Freelist: The contents of the freelist buffer are sent to the disk and stored at the appropriate location.

If the inode buffer is empty, the subsequent inode blocks of the file are fetched in *Read Inode* state and returned to the freelist.

The state machine cycles between *Return Inode*, *Read Inode* and *Write Freelist* states till all the file's inode blocks are returned to the freelist.

Write Metadata,Write SuperBlock: The FSM reads the superblock to delete the filename and update the filesystem metadata.

The new freelist head and freelist index are written to the superblock buffer and then transferred to sata stub.

Consider the process of deleting a 1 kB file from the disk,as shown in the figure above. Figure 3.13(a) shows the root inode block(21) of the file "12" and the freelist head (20). The block numbers from the inode block are returned one by one to the freelist head, starting from one position to the left of the freelist index. The freelist head is written to the disk when full (Figure 3.13(b)) and it's block number is stored in the last slot of the next freelist block to serve as a link node. The return operation continues till the root inode in empty and the next inode block of the file is read from the disk. The last inode of the file, block 23, is transferred to the address

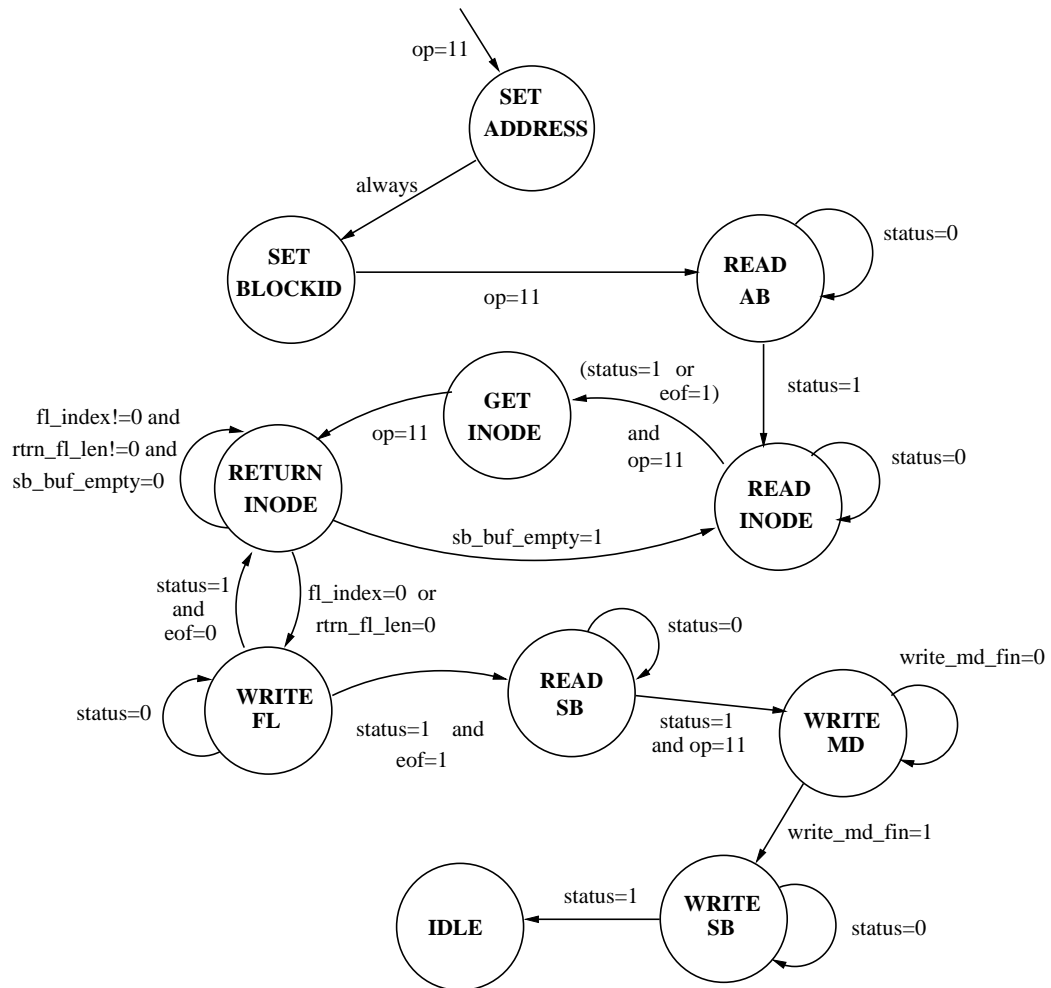


Figure 3.12: Remove File State Machine

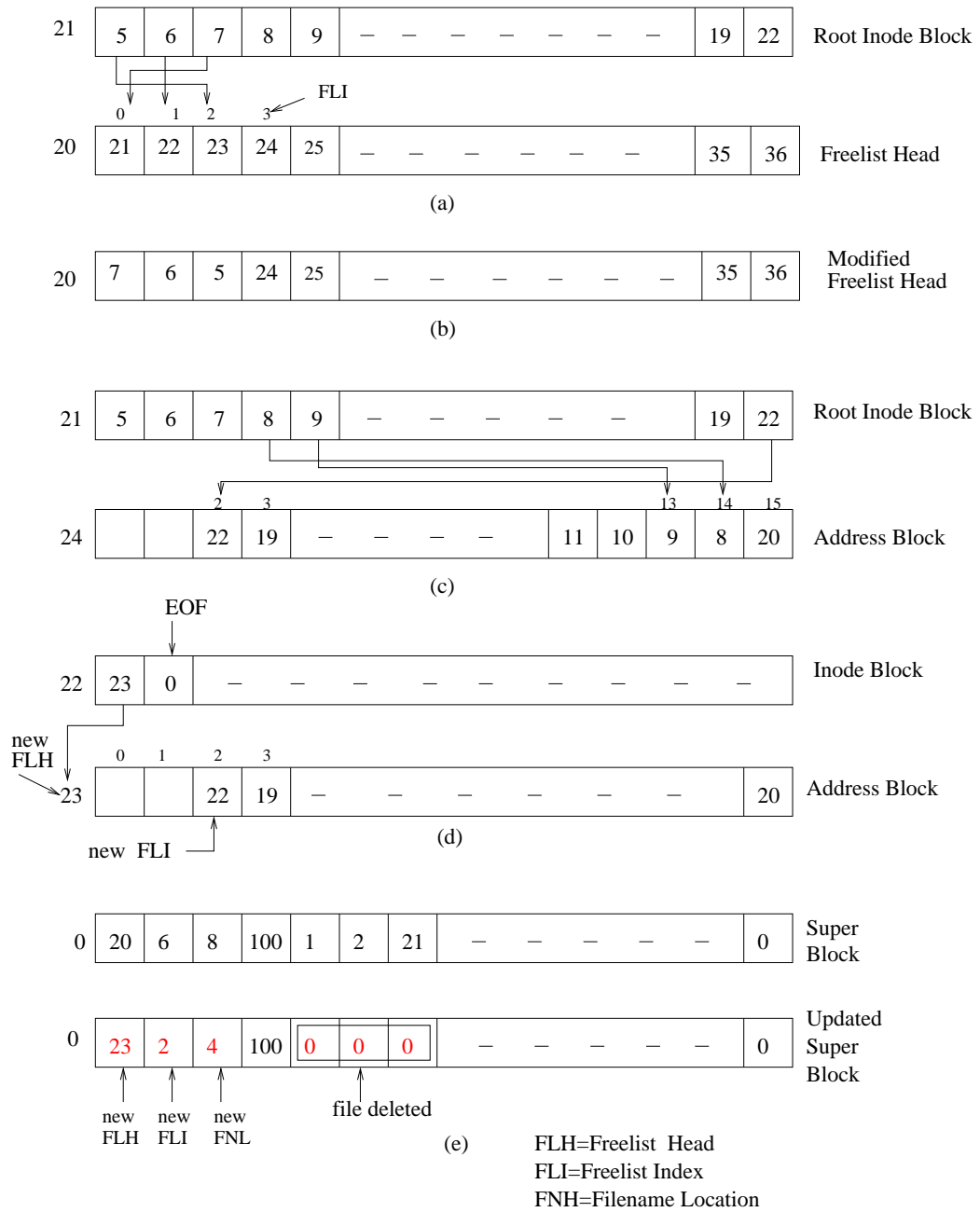


Figure 3.13: Remove File Example

block 24, terminating the return process. This now becomes the freelist head of the filesystem as illustrated in Figure 3.13(d). The superblock is updated with the new freelist head, freelist index and the file name and its root inode are deleted as shown in Figure 3.13(e).

CHAPTER 4: EVALUATION

To answer the central question of this thesis: Is it feasible to implement a filesystem in hardware that improves the disk to core bandwidth for data intensive compute cores, we simulated and synthesized the VHDL code for the HWFS described in 3. The experimental setup, results obtained and analysis follows.

4.1 Experimental Setup

To test the timing, area and performance metrics described in 1, we used ModelSim and Xilinx ISE for simulation and synthesis purposes.

4.1.1 Synthesis

The XST manual served as a guideline for writing synthesizable VHDL code. The controller state machine was decomposed into a two process Mealy model. Mealy state machines use fewer states than Moore models. This helps in reducing the number of flip-flops and decreases the latency of operations. Since our FSM is moderately sized (17 states) one hot encoding (OHE) was used to encode the state machine which gives us sufficient speed and reliability. It is more reliable than binary-encoded FSM since only two bits change during state transitions and uses fewer LUTs than gray encoding for encoding and decoding the states. Block RAMs were chosen in the design for the superblock/inode buffer and freelist buffer for efficient resource utilization. The buffer sizes vary with the disk block sizes of the filesystem. Hence, for large block sizes mapping this logic on BRAMs saves on slice resources. The state machine was suitably modified to read synchronously from BRAMs. A structural VHDL code instantiates and binds the synthesizable components in the design: the state machine, superblock buffer, freelist buffer, the comparator and the multiplexers.

The VHDL design description was synthesized using the Xilinx Synthesis Tool (XST) [19] available in the Xilinx ISE design suite, version 10.1 [16], for the target device XC4VFX60-11ff1152 from the Virtex-4 family [18] to generate the Xilinx specific NGC files. The HDL files are translated into gates and optimized for the architecture.

4.1.2 Simulation

For simulation purposes, the sata stub behavioral model and a VHDL testbench file was included to test the functionality of the design. The testbench instantiates the top level structural VHDL design module of the design. It then creates a 100 MHz master clock signal to synchronize the design and provides a reset pulse to initialize the state machine. Next, a test process generates an input test sequence to exercise the design. This includes a 64-bit filename for opening the required file from the disk and an 2-bit operation signal to select from one of the four operations: open, read, write and remove. On completing the read/write/remove file operations the state machine transitions to the idle state and asserts the stop-simulation signal. The testbench checks for this signal and reports a "Testbench Successful" message alongwith the iteration time.

ModelSim verification environment, version 6.3b, [5] running on a Linux Workstation was used for simulation and debugging.

4.2 Results and Analysis

4.2.1 Area

The design was synthesized for different block sizes and it was observed that the logic resource utilization is independent of the block size. The superblock and freelist memory buffers which vary with the block size are mapped onto BRAMs. Hence, the number of slices is not affected by increasing block sizes. For a 512 byte block, the design uses 783 slices, which takes up just 3 percent of the chip area. Table 4.1

Table 4.1: Statistics for HWFS resource utilization with different block sizes

Block Size	Slices	LUTs	F/Fs	BRAMs
64 B	759	1471	343	2
128 B	724	1369	345	2
256 B	749	1446	349	2
512 B	783	1502	353	2
1024 B	762	1463	356	3
4096 B	779	1476	364	10

Table 4.2: Subcomponent resource utilization statistics with a fixed block size

Component	Slices	LUTs	F/Fs	BRAMs
FSM	617	1169	358	0
sb buffer	0	0	0	2
fl buffer	0	0	0	1
comparator	17	33	0	0
sb mux	32	64	0	0
fl mux	32	64	0	0
sata mux	32	64	0	0
inode mux	18	32	0	0
fn mux	18	32	0	0

shows that two 18-Kbit Block RAMs are utilized for block sizes from 64 B to 512 B. This increases to 3 BRAMs for 1024 B blocks and goes up to 10 BRAMs for 4096 B blocks. The slice utilization varies between 724 to 783 slices. This is because, the logic resources used for decoding the BRAMs addresses varies with the block size and the Xilinx Synthesis Tool tries to optimize the design for speed.

The maximum frequency estimate of the design was found to be 144.35 MHz, which meets our target clock frequency of 100 MHz. The timing report also gave satisfactory delay estimates on the four netlist domains: register to register, input to register, register to output and input to output.

The pie chart derived from Table 4.2 shows the slice utilization for the subcomponents of the design for a fixed block size.

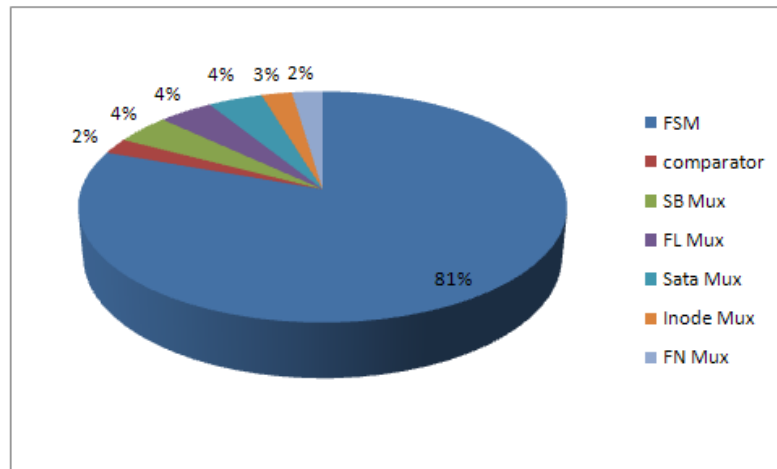


Figure 4.1: Subcomponent slice utilization

Table 4.3: HWFS Sequential Read Latency for different Block Sizes

FileSize(Bytes)	64 B	256 B	512 B
1 KB	4.24 us	5.32 us	7.81 us
10 KB	32.56 us	32.45 us	32.39 us
100 KB	299.2 us	268.6 us	263 us
1 MB	3.03 ms	2.71 ms	2.67 ms
10 MB	30.12 ms	26.8 ms	26.6 ms
100 MB	300.4 ms	266.4 ms	264.3 ms
1 GB	2.98 s	2.63 s	2.6 s
5 GB	14.6 s	13 s	12.9 s

Table 4.4: HWFS Sequential Write Latency for different Block Sizes

FileSize(Bytes)	64 B	256 B	512 B
1 KB	5.47 us	9.24 us	15.52 us
10 KB	33.02 us	34.26 us	36.6 us
100 KB	293.4 us	270.3 us	269.3 us
1 MB	2.96 ms	2.7 ms	2.65 ms
10 MB	29.6 ms	26.5 ms	26.5 ms
100 MB	295 ms	263 ms	262.5 ms
1 GB	2.96 s	2.65 s	2.63 s
5 GB	14.6 s	13.2 s	13 s

4.2.2 Performance

The filesystem performance for reads and writes was checked using file sizes ranging from 1 kilobytes to 5 gigabytes listed in Table 4.3 and Table 4.4 . Figure 4.2 shows a plot of the sequential read latency (including time taken to open a file and get its root inode) measured in cycles of 10 ns for 64 B, 256B and 512 B sized blocks. The number of cycles increases linearly with the filesize for a fixed block size.

We recorded the read and write latencies during simulation and computed the efficiency of the filesystem using the equation stated below

$$\text{eff} = \frac{\text{data latency}}{(\text{data} + \text{overhead}) \text{ latency}}$$

The data latency is the time taken to transfer raw data blocks of a file between the file system and the sata array. The read overhead includes the time taken to read a

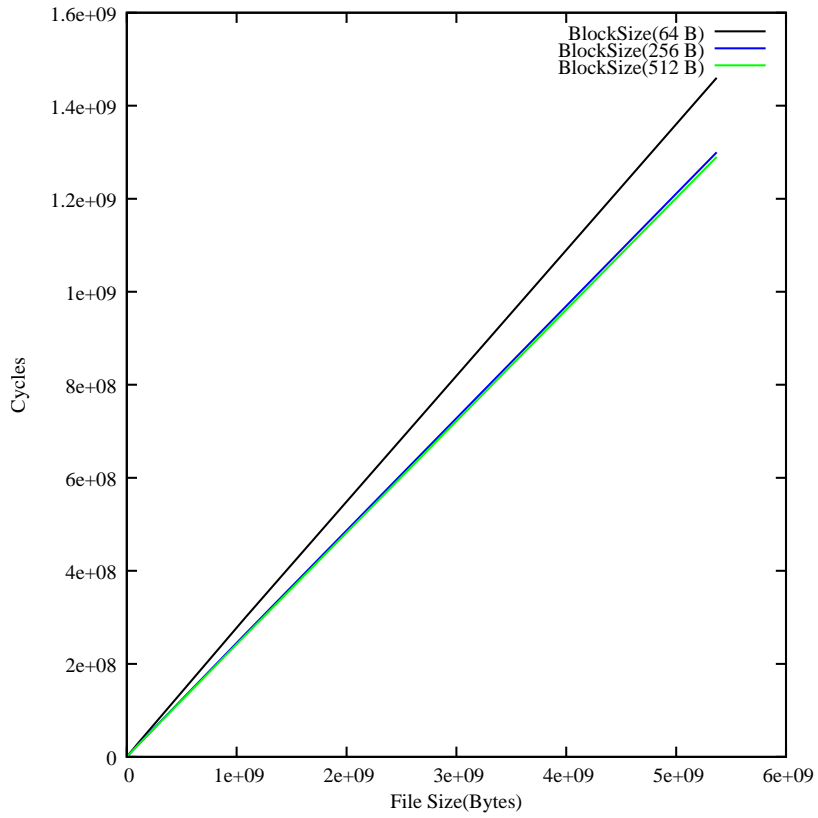


Figure 4.2: Read Latency (in cycles) plotted against different file sizes

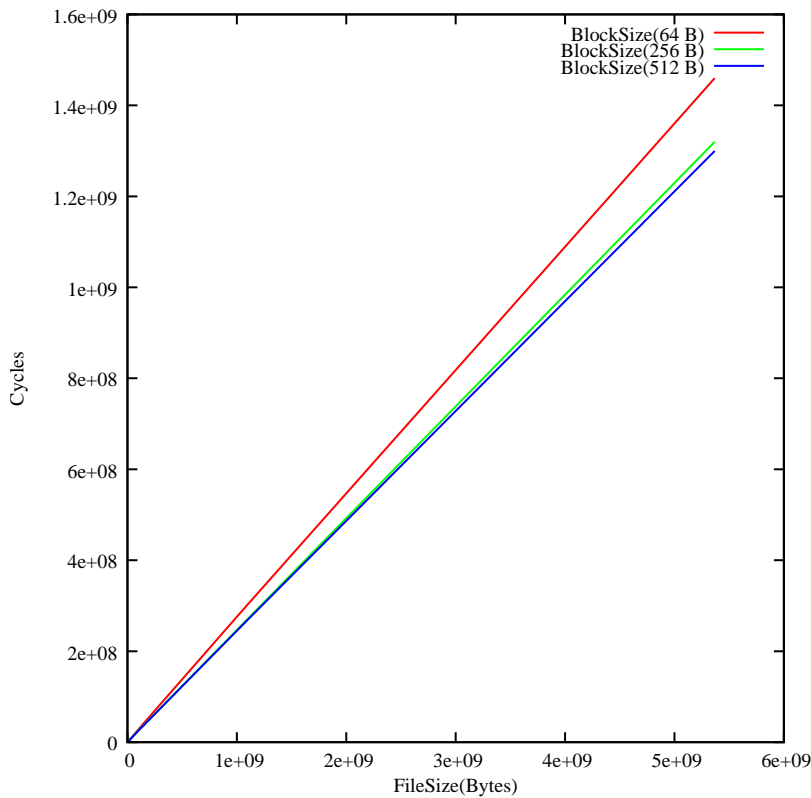


Figure 4.3: Write Latency (in cycles) plotted against different file sizes

superblock, find a filename match, get its root inode block (open file operation) and read the inode blocks of the file (read file operation). Figure 4.4 shows a plot of the sequential read and write efficiencies for 64 B, 256 B and 512 B sized blocks. It is observed that for small files (1 KB to 10 KB) the efficiency is below 80 % . It goes upto 95 % for 100 KB files and saturates for very large files (shown by a flattening of the plot for file sizes beyond 100 KB). This is because, the overheads have a lesser effect on the latencies for large files thereby achieving efficient run-time performance. Also, for a block size of 64 B the maximum efficiency achieved is 85 % . This goes upto 95 % for 512 B sized blocks, since more bytes of data and inode are delivered in each block transaction from the disk. For larger blocks, the figure shows that there is no substantial improvement in efficiency. Table 4.1 shows that using 512 B blocks consumes 2 BRAMs and for larger blocks the cost of BRAMs increases (10 BRAMs for 4 KB blocks). Hence, using 512 B blocks gives us the ideal tradeoff between area

and performance.

It is also observed from figure Figure 4.4, that the efficiency increases with the size of the block for the same filesize. This is because, using larger blocks improves the data transfer bandwidth. But having larger block sizes on the disk increases the block fragmentation, leaving large portions of the disk unused. Our file system will primarily deal with very large sized files existing in biological databases and hence fragmentation due to large block sizes will not be a critical issue. Also, from the read latency table it can be seen that the inode overhead involved in reading files decreases slightly with increasing file sizes.

Table 4.3 shows that reading a 1 GB file with 512 Byte block sizes takes 2.7 seconds. Hence, the sequential read bandwidth can be estimated as 3.01 Gb/s. We looked at a Xilinx application note for an Embedded Serial ATA Storage System on a Virtex-4 platform [13]. They have implemented the design using the ASICS SATA IP core on an ML405 Platform FPGA having a PowerPC 405 processor running on MontaVista Linux. The file system access performance listed states a sequential read bandwidth of 44,645 KB/s (349 Mb/s) for a WD360 Raptor Hard disk. Though we cannot provide a direct comparison with the bandwidth obtained from our implementation due to lack of a Serial ATA host controller IP core, we can estimate that a hardware filesystem implementation will provide substantial improvements in disk to core bandwidth.

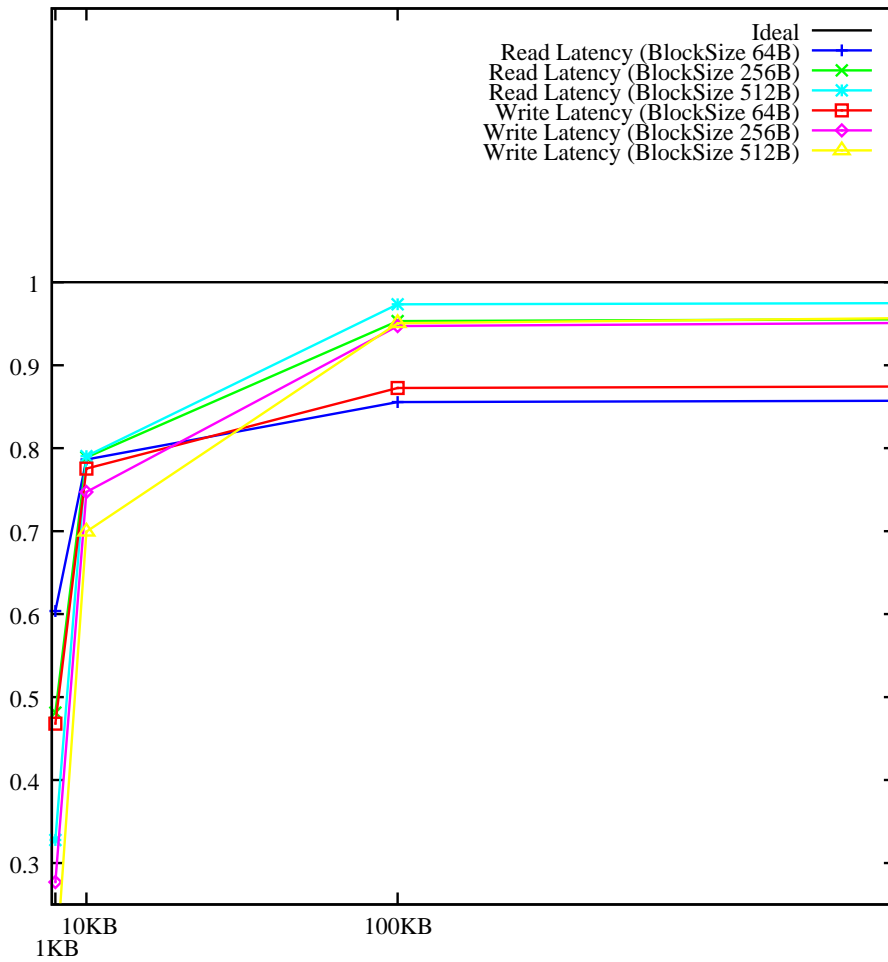


Figure 4.4: File Read/Write Efficiency plotted against different file sizes

CHAPTER 5: CONCLUSION

The goal of this work was to verify the feasibility of implementing a filesystem directly in hardware for high performance computing. The novel architecture proposed and implemented in this project avoids the sequential system software bottleneck by giving the computation cores on the FPGA the opportunity to bypass the operating system entirely. It enables the cores to get direct, high bandwidth access to large data sets. The design, correctly implements the four basic filesystem operations open, read, write and remove. The filesystem was synthesized for an ML410 Virtex-4 platform FPGA with a 3 % slice utilization. The read/write efficiencies measured during simulation, improve with larger disk block sizes due to higher data transfer rates. The filesystem also gives efficient run-time performance for larger files due to smaller overhead.

Once the SATA IP core is purchased, the filesystem can be interfaced to it for measuring the actual File I/O performance. The hardware file system can be replicated on each node of the 64-node cluster. This design will therefore, serve as an important first step to develop and evaluate a parallel hardware filesystem which will co-ordinate file access from multiple disks accross the cluster and present a single filesystem image for parallel applications.

REFERENCES

- [1] T. Agerwala. System trends and their impact on future microprocessor design. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. Invited keynote talk.
- [2] ASICS World Services. Serial ata host ip core, Dec. 2007. URL: www.asics.ws.
- [3] M. J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, September 1991.
- [4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.
- [5] Modelsim. URL: <http://www.model.com>.
- [6] NCBI user services. Genbank overview, Aug. 2005.
- [7] R. B. Ross. *Reactive Scheduling for Parallel I/O Systems*. PhD thesis, Clemson University, Dec. 2000.
- [8] R. Sass, W. V. Kritikos, A. G. Schmidt, S. Beeravolu, P. Beeraka, K. Datta, D. Andrews, R. Miller, and D. S. Jr. Reconfigurable computing cluster(icc) project:investigating the feasibility of fpga-based petascale computing. In *Proceedings of the 2007 International Symposium on Field Programmable Custom Computing Machines*, April 2007.
- [9] Serial ATA International Organization. Serial ata: A comparison with ultra ata technology.
- [10] Serial ATA International Organization. Serial ata specification, Oct. 2005.
- [11] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley and Sons, Inc., December 2004.
- [12] T13, Technical Committee of Accredited Standards Committee NCITS. Ata command set. URL: <http://www.seagate.com/support/disc/manuals/ata/d1153r17.pdf>.
- [13] S. Tam and L. Jones. Embedded serial ata storage system. Technical Report XAPP716(v1.0), Xilinx, Inc., oct 2006. URL: www.xilinx.com/support/documentation/application_notes/xapp716.pdf=20.
- [14] Xilinx Inc. Multi - giga bit serial i/o transceivers. URL: http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/rocketio.htm.
- [15] Xilinx Inc. Powerpc 405 processor. URL: http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/powerpc.htm.

- [16] Xilinx Inc. Xilinx ise design suite 10.1. URL: http://www.xilinx.com/products/design_resources/design_tool/\discretionary{-}{-}{-}index.htm.
- [17] Xilinx Inc. Xilinx ml410 board. URL: <http://www.xilinx.com/ml410-p/>.
- [18] Xilinx Inc. Xilinx virtex-4 fpga's. URL: http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/\discretionary{-}{-}{-}virtex4/index.htm.
- [19] Xilinx Inc. Xst user guide. URL: toolbox.xilinx.com/docsan/xilinx5/pdf/docs/xst/xst.pdf.

APPENDIX A: VHDL DESIGN ENTITIES

The synthesizable VHDL components of the Hardware Filesystem are listed below.

HWFS Structural

:

```

entity hwfs is
  generic
  (
    SB.BUF_ADDR.WIDTH: integer:= 6;
    FL.BUF_ADDR.WIDTH: integer:= 4
  );
  port
  (
    clk           : in std_logic;
    reset         : in std_logic;
    full          : in std_logic;
    empty         : in std_logic;
    operation     : in std_logic_vector(1 downto 0);
    fn1           : in integer;
    fn2           : in integer;
    write_data    : in integer;
    wrfin         : in std_logic;
    status        : in std_logic;
    datain        : in integer;
    cmd           : out std_logic;
    blknum        : inout integer;
    write_disk    : out std_logic;
    count_reset   : out std_logic;
    sata_mux_sel  : inout std_logic_vector(1 downto 0);
    dataout       : out integer;
  );

```

```

        push          : out std_logic;
        pop           : out std_logic
    );
end hwfs;

```

FSM Behavioral

```

:
```

```

entity fsm is
    generic(
        SB.BUF_ADDR.WIDTH: integer;
        FL.BUF_ADDR.WIDTH: integer;
        BLOCK_SIZE        : integer
    );
    port(
        clk                : in  std_logic;
        reset              : in  std_logic;
        --To select Read/Write/Remove operation
        operation          : in  std_logic_vector(1 downto 0);
        --output from FileName Comparator
        found              : in  std_logic;
        --from SataStub indicating that one DataBlock is read or written
        status             : in  std_logic;
        -- BlockID from SuperBlock Cache
        sb_datain          : in  integer;
        --BlockID from FreeList Cache
        fl_datain          : in  integer;
        -- from processor to stop writing data blocks to SataStub
        wrfin              : in  std_logic;
        full               : in  std_logic;
        empty              : in  std_logic;

```

— Output

```

push          : out std_logic;
pop           : out std_logic;
— To SataStub to select Read/Write operation
cmd           : out std_logic;
count_reset  : out std_logic;
—To SataStub for writing 2D array to disk file
write_disk   : out std_logic;
—Write Enable to SuperBlock Buffer
we1          : out std_logic;
—Write Enable to FreeList Buffer
we2          : out std_logic;
—BlockID sent to SataStub to fetch blocks from Array
blknum       : out integer;
sb_mux_sel   : out std_logic_vector(1 downto 0);
fl_mux_sel   : out std_logic_vector(1 downto 0);
sata_mux_sel : out std_logic_vector(1 downto 0);
inode_mux_sel : out std_logic;
fn_mux_sel   : out std_logic;

```

— IN/OUT

```

sb_buf_addr  : out std_logic_vector(SB.BUF_ADDR.WIDTH-1 downto 0);
dpaddr      : out std_logic_vector(SB.BUF_ADDR.WIDTH-1 downto 0);
fl_buf_addr  : out std_logic_vector(FL.BUF_ADDR.WIDTH-1 downto 0);
);
end fsm;

```

Super Block Buffer

:

```

entity sb_buffer is
    generic
    (
        SB.BUF_ADDR.WIDTH: integer
    );
    port
    (
        clk      : in std_logic;
        we       : in std_logic;
        addr     : in std_logic_vector (SB.BUF_ADDR.WIDTH-1 downto 0);
        dpraddr  : in std_logic_vector (SB.BUF_ADDR.WIDTH-1 downto 0);
        datain   : in integer;
        spdataout : out integer;
        dpdataout : out integer
    );
end sb_buffer;

```

Free List Buffer

:

```

entity fl_buffer is
    generic
    (
        FL.BUF_ADDR.WIDTH: integer
    );
    port
    (
        clk      : in std_logic;
        we       : in std_logic;
        addr     : in std_logic_vector (FL.BUF_ADDR.WIDTH-1 downto 0);
        datain   : in integer;
        dataout  : out integer
    );

```

```

    );
end fl_buffer;

```

4:1 Multiplexer : FreeList Mux, SuperBlock Mux and SATA Mux

:

```

entity mux_41 is
    port
    (
        a    : in  integer;
        b    : in  integer;
        c    : in  integer;
        d    : in  integer;
        sel  : in  std_logic_vector(1 downto 0);
        o    : out integer
    );
end mux_41;

```

2:1 Multiplexer : Inode Mux, FileName Mux

:

```

entity mux_21 is
    port
    (
        a    : in  integer;
        b    : in  integer;
        sel  : in  std_logic;
        o    : out integer
    );
end mux_21;

```