Report Number: WUCSE-2003-4

2003-05-01

# An FPGA-Based High-Speed Search Engine for Off-the-Shelf Hard Drives

Benjamin M. West

This thesis presents a Field Programmable Gate Array (FPGA) based, high-speed search system that is intended to perform simple data mining operations on the data streaming from an off-the-shelf hard drive. This system includes the search engine itself and a device that snoops the traffic on an ATAPI/IDE peripheral bus, capturing data transmitted by a hard drive attached to the bus and forwarding that data to the search engine. The search engine, which is an adaption of the Smith-Waterman local sequence alignment algorithm, can process search data at a rate of 100 MB/sec, with a query string up to... Read complete abstract on page 2.

# An FPGA-Based High-Speed Search Engine for Off-the-Shelf Hard Drives

Benjamin M. West

Complete Abstract:

This thesis presents a Field Programmable Gate Array (FPGA) based, high-speed search system that is intended to perform simple data mining operations on the data streaming from an off-the-shelf hard drive. This system includes the search engine itself and a device that snoops the traffic on an ATAPI/IDE peripheral bus, capturing data transmitted by a hard drive attached to the bus and forwarding that data to the search engine. The search engine, which is an adaption of the Smith-Waterman local sequence alignment algorithm, can process search data at a rate of 100 MB/sec, with a query string up to 38 bytes long. The motivation for developing this system is to move most of the processing burden in data mining applications from the CPU to a level closer to the hard drive, while at the same time achieving search throughput gains by taking advantage of massive parallelism possible in FPGA-based implementations. To demonstrate the magnitude of performance gain that is possible, this thesis also includes the results of simple performance tests that compare this system to traditional, CPU-based search applications like the UNIX tool "grep." The search engine and related components were developed and implemented on the Field Programmable Port Extender (FPX), an FPGA-based component of the Washington University Gigabit Switch (WUGS).

# An FPGA-Based High-Speed Search Engine for Off-the-Shelf Hard Drives

**Benjamin M. West**

Benjamin West, "An FPGA-Based High-Speed Search Engine for Off-the-Shelf Hard Drives," Master's Thesis, Technical Report WUCSE-2003-4, Department of Computer Science and Engineering, Washington University, Saint Louis, MO, 2003.

Computer and Communications Research Center
Washington University
Campus Box 1115
One Brookings Dr.
St. Louis, MO 63130-4899

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

---

# AN FPGA-BASED, HIGH-SPEED SEARCH ENGINE FOR OFF-THE-SHELF HARD DRIVES

by

Benjamin M. West, B.S.

Prepared under the direction of Professor Roger Chamberlain

---

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2003

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

---

ABSTRACT

---

# AN FPGA-BASED, HIGH-SPEED SEARCH ENGINE FOR OFF-THE-SHELF HARD DRIVES

by Benjamin M. West

---

ADVISOR: Professor Roger Chamberlain

---

May, 2003
Saint Louis, Missouri

---

This thesis presents a Field Programmable Gate Array (FPGA) based, high-speed search system that is intended to perform simple data mining operations on the data streaming from an off-the-shelf hard drive. This system includes the search engine itself and a device that snoops the traffic on an ATAPI/IDE peripheral bus, capturing data transmitted by a hard drive attached to the bus and forwarding that data to the search engine. The search engine, which is an adaptation of the Smith-Waterman local sequence alignment algorithm, can process search data at a rate of 100 MB/sec, with a query string up to 38 bytes long. The motivation for developing this system is to move most of the processing burden in data mining applications from the CPU to a level closer to the hard drive, while at the same time achieving

search throughput gains by taking advantage of the massive parallelism possible in FPGA-based implementations. To demonstrate the magnitude of performance gain that is possible, this thesis also includes the results of simple performance tests that compare this system to traditional, CPU-based search applications like the UNIX tool "grep." The search engine and related components were developed and implemented on the Field Programmable Port Extender (FPX), an FPGA-based component of the Washington University Gigabit Switch (WUGS).

# Contents

# List of Tables

# List of Figures

# Acknowledgments

This thesis could not have happened by any measure without the generous help of my fellow students, the faculty of the Computer Science Department, Computer Engineering Program, and Electrical Engineering Department, and the ARL staff. Specifically, I wish to thank Dr. Roger Chamberlain for somehow enduring me as an advisee during my research and consequent thesis composition. His optimism and energy enabled me to meet milestones with which I otherwise would still be struggling. I wish to thank Drs. Ronald Indeck, Mark Franklin and Ronald Cytron, for the original idea upon which this thesis is based was theirs and Dr. Chamberlain's. I would like to thank Maggie Qiong Zhang and Brian Bruggeman, who worked with me in course projects that ultimately evolved into the research for this thesis. I also would like to thank Dr. John Lockwood for allowing me to hi-jack an FPX and WUGS for the purpose of this thesis' research, especially considering the demands my work placed on his hardware. In addition, I wish to thank Dr. Lockwood's students Todd Sproull, Sarang Dharmapurikar, and David Lim, who suffered an unending barrage of questions and calls for assistance from me, but still always offered their help. I would like to thank Dr. Jeremy Buhler for providing his clear, concise outline of the Smith-Waterman algorithm that made my implementation possible, and for being on my thesis committee. Finally, I would like to thank my parents for providing me sincere encouragement and support during my time at Washington University, and for perhaps suspending their disbelief and agreeing whenever I said, yes, I would graduate.

Benjamin M. West

*Washington University in Saint Louis*
*May 2003*

# Chapter 1

# Introduction

## 1.1 Overview

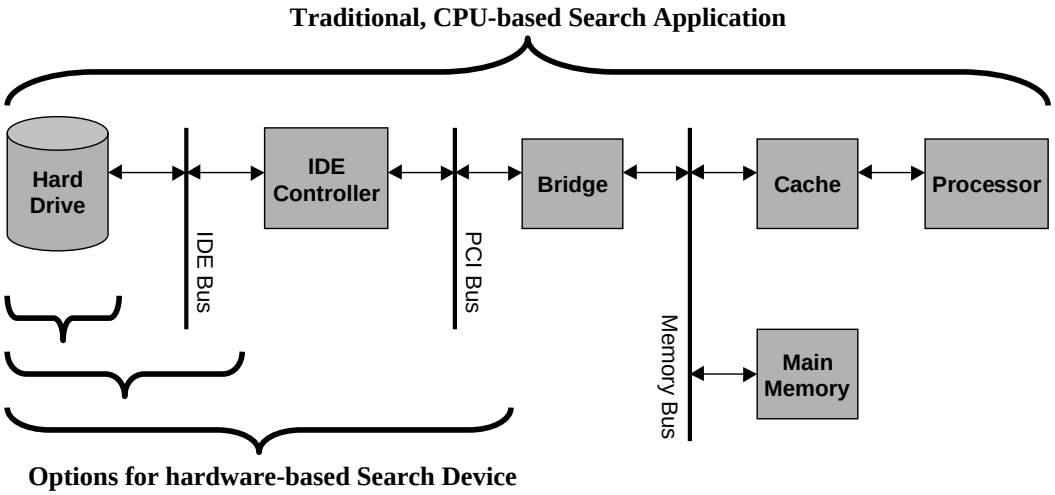


Figure 1.1: Traditional Data-path between Hard Drive and Processor

The continued, explosive growth of hard drive capacity, which outstrips the growth rate predicted by Moore's Law, has engendered the desire to fill this new-found space with meaningful data of a volume hitherto unimagined. This includes large database projects such as the Human Genome Project, intensified information retrieval activities by Intelligence services, digitization of printed paper-based libraries, and so on. The non-trivial nature of these data naturally leads to the desire to search them thoroughly, and as quickly as possible. However, the capability to quickly transport such data from secondary storage to a processor for searching

has not grown at nearly the same rate as hard drive capacity, and indeed lags significantly behind the potential state of the art. Figure 1.1 conceptually illustrates the traditional arrangement for attaching a hard drive to a processor for any sort of data-mining application. No less than four levels of interconnect exist between the data on the drive and the processor.

1. IDE peripheral bus (hard drive -> drive controller)
2. PCI bus (controller -> memory bus bridge)
3. Memory bus (bridge -> processor cache)
4. Cache bus (cache -> processor)

Each level adds its own overhead, i.e., control or wait states, latency in transmission across the interconnect because of framing requirements, arbitration among multiple bus nodes, and synchronization across clock domains. These elements of overhead compound to create a bottleneck significant enough that a large majority of the time required for any search application is spent while the processor waits for data from the hard drive. Indeed, improvements are being made to all levels of interconnect illustrated in Figure 1.1, but yet the heterogenous nature of this data-path still leads to a throughput that is less than ideal, as the data-path is only as fast as its slowest segment.

A straightforward solution to this troublesome bottleneck is to bring the processing element closer to the hard drive, i.e., to eliminate some of the levels of interconnect and make the path between the data and the processor shorter and faster. This idea is not new, as it dates back to the experimental database machines of the 70's [29]. More recently, this concept finds implementation in the Smart Disks developed by the University of Maryland [40], the University of California at Santa Barbara [20] [40], and Carnegie-Mellon University [31]. What differs between these various implementations is the amount of processing power placed near the disks, ranging from small embedded microprocessors that are simply replicated as often as necessary to provide the necessary computational power through parallelism, to higher-class processors only a generation or two behind state of the art, which offload a significant portion of the data-mining code from the CPU. Even outside academia in the commercial sector, this concept is manifesting itself in the form of Network Attached Storage Devices (NASDs) [31], where hard drives feature an on-board thin network server, so that they can be attached directly to a Local Area Network (LAN).

However, none of these implementations to date capitalizes on recent advances in high speed Field-Programmable Gate Arrays (FPGAs), such that whatever computing power that is placed near the disk is typically in the form of microprocessors or Application-Specific Integrated Circuits (ASICs) and some amount of RAM. This approach, while perhaps allowing the fast execution of a limited number of tasks, places a restriction on the storage device's versatility for on-board data reduction, thus constraining its overall usefulness. In contrast, the prototype search system presented in this thesis is comprised of large (millions of gates) Xilinx FPGAs, which may be programmed to perform arbitrary operations on the data at 32 bits wide, and at clock speeds approaching 100 MHz. Furthermore, the decentralized architecture of the FPGAs makes them ideal for operating on the data in a streaming fashion, e.g., with a parallel bit-slice-based (or byte-slice-based) design, to provide a data throughput high enough to handle the rates at which bits flow from hard drive read heads.

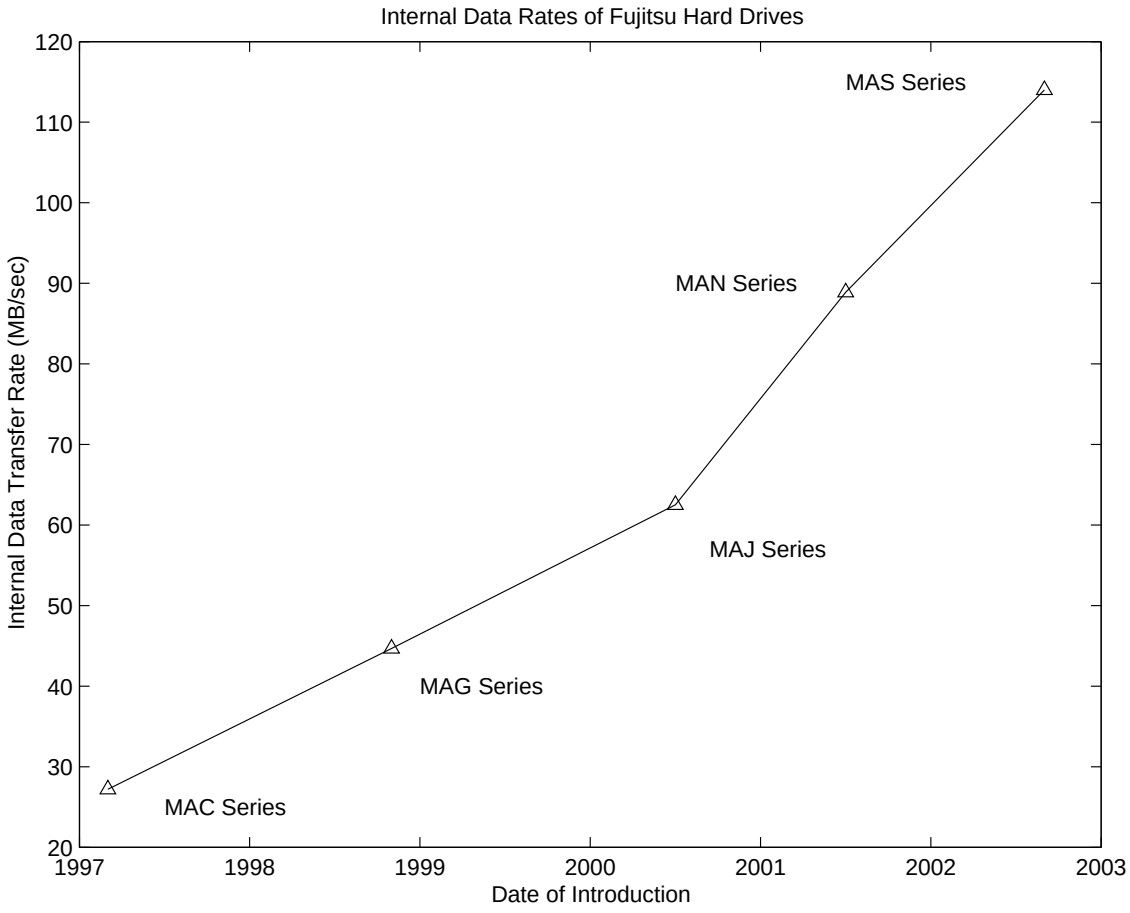


Figure 1.2: Internal Data Transfer Rates for Fujitsu Hard Drives [10]

As a demonstration of the trends for increasing hard drive speeds, Figure 1.2 plots the internal data transfer rates of Fujitsu hard drives against their introduction dates. The internal data transfer rate is the speed at which the hard drive's internal components can deliver data to the chip that relays those data onto the peripheral bus (i.e., the ATAPI/IDE or SCSI controller chip on the hard drive). This rate is dependent on the rotational speed of the drive's platters and on the speed of the electronic components that process the bits streaming from the magnetic read head. The trend apparent for the three most recent models of Fujitsu hard drives show a yearly increase in internal data rate of approximately 30 MB/sec. This rate does not follow a growth trend consistent with Moore's Law, but it is still approaching the aggregate throughput limit of the data-path illustrated in Figure 1.1. Furthermore, the heterogeneous nature of the data-path in Figure 1.1 means the growth of its effective throughput rate is difficult to characterize, although that growth frequently falls well below Moore's Law. By comparison, FPGA speeds have steadily followed Moore's Law over the past decade [8], making FGPAs quite suitable for processing data in such an internal data-path of a hard drive.

Although the multiple levels of interconnect for the data-path shown in Figure 1.1 lead to a compounded overhead that drags down the effective throughput of that data-path, those levels also provide multiple locations where an FPGA-based search device could be placed, which each level still enjoying a throughput advantage over CPU-based search applications. This thesis has so far discussed the possibility of placing such a device inside the hard drive, as close to the storage medium as possible, and Chapter 3 will present a search device that sits on the hard drive's peripheral bus (i.e., an ATAPI/IDE bus). However, other possible locations for a search device, such as the PCI interconnection bus, or even the peripheral bus connecting multiple hard drives in a Redundant Array of Inexpensive Drives (RAID), do exist, but they are not discussed in this thesis.

For the prototype of the hardware-based search system described above, the author made heavy use of Dr. John Lockwood's Field-Programmable Port Extender (FPX) [26], an FPGA-based device that sits between the switch fabric on the Washington University Gigabit Switch (WUGS) [39] and a line-card. The FPX was originally devised to allow arbitrary, user-programmable transformation of ATM cells that traverse the WUGS, but for the purpose of this thesis, it simply provides a

generic FPGA prototyping platform, with a 1.2 Gbit/sec[1] bidirectional data-path (i.e., the WUGS switch fabric) to other FPXs. Thus, the prototype could be easily specialized or reconfigured to suit nearly any application, and multiple FPXs could be used in parallel if the gates required to implement the search logic exceeded the number of CLBs on a single FPX.

## 1.2 Contributions

The top-level goal of the work presented in this thesis was to fabricate a functioning FPGA-based search system out of the FPX/WUGS infrastructure available to the author. This goal was decomposed into the following sub-tasks:

- Inject data from a hard drive into the WUGS switch fabric, for forwarding onto an FPX
- Process the hard drive's data in a fashion that exhibits parallelism impossible with traditional, CPU-based search applications
- Develop and consolidate paths for control and search result reporting
- Demonstrate search performance gain of this search system

The first two tasks were realized as FPX modules, each programmed onto its own FPX. The third was realized by building atop the extant control software for the FPX and WUGS, to provide a streamlined interface to control functions specific to the author's implementation. The fourth goal was realized by measuring execution times of traditional CPU-based search applications, both with timing functions internal to the host workstation running the search applications, and with an external device that monitors IDE bus traffic. These execution times, measured at two points in the data-path illustrated in Figure 1.1, were then folded together to quantitatively show where limitations in such a traditional hard drive-CPU data-path impede search throughput.

The specific contributions made during the work of this thesis are listed below:

- IDE Bus Snooper FPX Module

[1]This thesis uses the following convention for quantifying data: GB = $10^9$ bytes, MB = $10^6$ bytes, KB = $10^3$ bytes, Gb = $10^9$ bits, Mb = $10^6$ bits, and Kb = $10^3$ bits.

  - Deciphered the ATAPI/IDE Protocol [27] to design a state machine that recognizes data bursts initiated by the hard drive
  - Designed and built a custom PCB with voltage translation buffers to handle voltage incompatibility between the FPX and the IDE peripheral bus
  - Separated the Bus Snooper into two clock domains, to allow sampling of the IDE bus signals at a higher frequency (increase sampling accuracy)
  - Developed a control path, both for the hard drive being snooped and for the Bus Snooper Module, that allowed data retrieved from hard drive over both paths (i.e., through the IDE host controller and through the Snooper) to be viewed side-by-side in real time

- Biological Computation ("BioComp") FPX Module
  - Developed a systolic array-based implementation of the Smith-Waterman local sequence alignment algorithm [35] that was scalable up to the size of the FPGA
  - Implemented pipelining and parallelism to allow the BioComp Module to accept search data at the full width of the FPX's data-path
  - Devised a scheme for extracting snapshots of the state of the systolic array that doesn't involve a fan-in arrangement that scales with the size of the array
  - Developed a Web-based interface to the BioComp module that allowed users to submit runtime parameters and search queries, and then to view the search results, all in real time

- Performance tests of traditional CPU-based search applications
  - Developed a method for reliably measuring CPU execution time down to sub-microsecond accuracy
  - Devised an experiment method that avoids caching effects in the host workstation's secondary storage
  - Devised a scheme for aligning the CPU execution time measurements made on the host workstation with IDE bus activity measurements made by an external device
  - Implemented the core of the BioComp FPX Module as-is in software to provide a direct point of performance comparison

## 1.3 Organization of This Thesis

Chapter 2 presents relevant background material to provide a helpful context to the information contained in subsequent chapters. Specifically, this chapter outlines the following topics:

- The Smith-Waterman local sequence alignment algorithm [35], and its typical implementation in parallel hardware
- Recent work in hardware-based sequence alignment
- Recent work in hardware-based text searching
- The FPX/WUGS infrastructure and associated design environment

Chapter 3 details the author's use of an FPX to snoop the traffic on an AT-API/IDE [27] bus between a hard drive and its host controller, extracting the contents of data bursts originating from the drive. This IDE Bus Snooper then repackages those captured data into ATM cells, and injects them into the WUGS switch fabric for forwarding onto a second FPX for processing and data reduction. The IDE Bus Snooper is the first of two components of the high-speed search system presented in this thesis.

Chapter 4 details the author's use of a second FPX to perform simple search applications on the data stream extracted from the IDE bus by the IDE Bus Snooper FPX. The reprogrammable nature of the FPX allows the implementation of nearly any stream-based search operation. For this thesis, the author implemented the Smith-Waterman algorithm [35], a dynamic programming algorithm for local sequence alignment. This algorithm is traditionally used for alignment of genomic and protein sequences, i.e., biological computation. However, the author's implementation, called the BioComp Module, operates on data at the byte level, allowing this FPX module to also perform inexact string matching directly on ASCII text. The BioComp Module was designed with high search throughput in mind, and thus, by taking generous advantage of the massive parallelism possible in FPGA-based implementations, it can process search data at 100 MB/sec. The BioComp module is the second component of the search system presented in this thesis.

Chapter 5 details search performance tests conducted on a traditional workstation like that illustrated in Figure 1.1, measuring the required time to complete

searches of artificially generated datasets of varying size. The actual search application chosen for these tests was the GNU string-matching program "grep," both for its widespread use among UNIX users, and for its simple-to-characterize, state machine-based kernel. Also chosen was the author's software implementation of the Smith-Waterman algorithm [35], to provide more direct comparison with the search performance of the BioComp FPX Module. The execution time for both search applications was measured, both in terms of CPU cycles elapsed and in terms of disk activity observed on the hard drive's peripheral bus. These timing measurements are decomposed in Chapter 5 to illustrate where each search application suffered an increase in execution time because it was idle waiting for data from the hard drive, or because it relied on serialized execution on the workstation's processor.

Chapter 6 then summarizes this thesis, restating the points presented in the preceeding chapters, and the contributions outlined above. This Chapter also explains avenues for future research and especially for possible improvement upon the work presented in this thesis, both for the near future, and for versions of this search system that migrate beyond the FPX/WUGS infrastructure.

# Chapter 2

# Background

## 2.1 Smith-Waterman Local Alignment Algorithm

The classic Smith-Waterman local alignment algorithm [35] is a dynamic programming method used in biological computation which finds the best possible alignment between two strings of characters, the pattern string $p$ and the target string $t$. The score of the alignment is judged by the gaps and mismatched characters that must be tolerated to make the alignment. The pattern string $p$ is understood to be the search query argument, and the target string $t$ the database upon which the search is performed, which is expected to be orders of magnitude larger than $p$. The range of characters found in both strings depends on the nature of alignment that is sought; for genome alignments, for example, the range would be the four DNA bases "ATCG" (along with whatever wildcard characters are allowed), and for proteomic alignments, the range would the be the approximately 20 amino acids (and wildcard characters). Indeed, this alignment method can also find application with regular text searching, where the range of characters could be those in the 8-bit ASCII character table or any other arbitrary character set. Furthermore, given the ability of the Smith-Waterman algorithm to find alignments that include gaps or mismatched characters, it lends itself quite well to performing inexact searches on text, e.g., searches that account for variations in spelling of the sought pattern string $p$.

Indeed, numerous algorithms and methods exist that are meant primarily for inexact text string matching, most notably those algorithms that involve Regular Expression matching. The decision to implement the the Smith-Waterman algorithm so it could handle tasks in biological computation, i.e., DNA sequence alignment, and generic text-string matching, was made since this would allow the implementation's

use in a range of potential search applications broader than those possible just for inexact text-string matching.

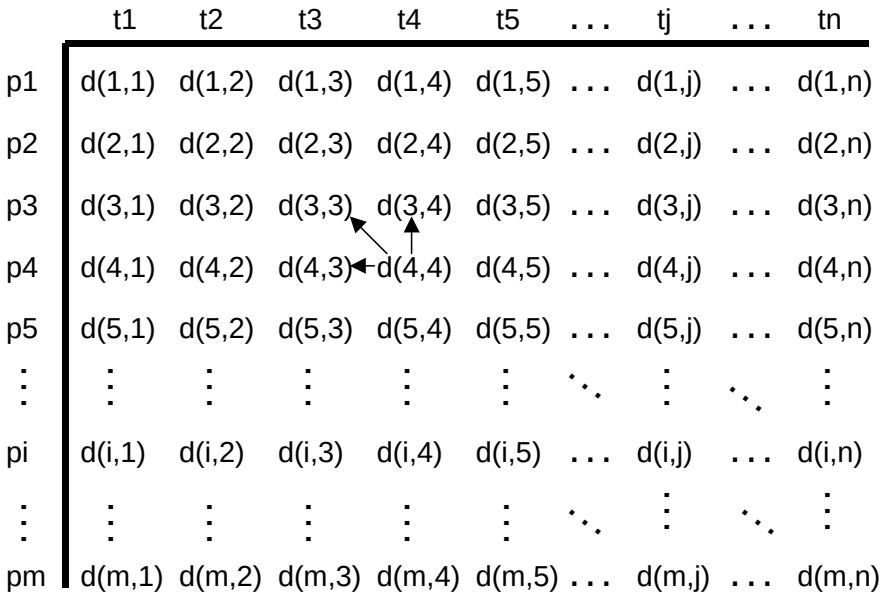


Figure 2.1: Dynamic Programming (DP) Matrix

The Smith-Waterman local alignment algorithm finds the best possible alignment between $p$ and $t$ by arranging both strings along an axis of a 2-dimensional matrix of size $m \times n$, shown in Figure 2.1, where $m$ is the length of the pattern string $p$ and $n$ the length of the target string $t$. Each element $d(i,j)$ in this matrix (called the Dynamic Programming (DP) matrix) represents the score for the $ith$ character of $p$ aligned with the $jth$ character of $t$. This score is determined by the base cases and recursion shown in Equations 2.1 through 2.4.

$$d(1,1) = B(1,1) \tag{2.1}$$

$$d(i,1) = max[A; d(i-1,1) + A; B(i,1)] \tag{2.2}$$

$$d(1,j) = max[d(1,j-1) + A; A; B(1,j)] \tag{2.3}$$

$$d(i,j) = max[d(i,j-1) + A; d(i-1,j) + A; d(i-1,j-1) + B(i,j)] \tag{2.4}$$

$A$ in Equations 2.1 through 2.4 is a user-defined constant representing the single-character gap penalty (usually negative), and $B(i,j)$ the scoring function dependent on the characters $p_i$ and $t_j$. In the implementation used for this thesis,

the scoring function $B(i, j)$ simply returns a user-defined constant (usually positive) when the characters $p_i$ and $t_j$ match, and another constant (usually negative), when the characters don't match. Thus, these two constants, which shall be called $B_{match}$ and $B_{nomatch}$, respectively, represent the single-character match score and mismatch penalty that are used in calculating the overall alignment scores for $p$ and $t$.

It should be noted the recurrence defined in Equations 2.1 through 2.4 is not local with respect to both the target and pattern strings. Rather, the alignment is local with respect to the target string, but global with respect to the pattern. That is, the alignment scores computed by this recursion represent an alignment of the pattern with the entire target string. By comparison, a recursion that is local for both the pattern and the target strings could compute scores representing an alignment of the pattern with a subset of the target string.

| | | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 | t10 | t11 | t12 | t13 | t14 | t15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | a | a | c | g | g | a | t | g | a | g | g | g | t | a | t |
| **p1** | a | 9 | 9 | -9 | -9 | -9 | 9 | -9 | -9 | 9 | -9 | -9 | -9 | -9 | 9 | -9 |
| **p2** | t | -9 | 0 | 0 | -18 | -18 | -9 | 18 | 0 | -9 | 0 | -18 | -18 | 0 | -9 | 18 |
| **p3** | c | -9 | -18 | 9 | -9 | -27 | -27 | 0 | 9 | -9 | -18 | -9 | -27 | -18 | -9 | 0 |
| **p4** | g | -9 | -18 | -9 | 18 | 0 | -18 | -18 | 9 | 0 | 0 | -9 | 0 | -18 | -27 | -18 |
| **p5** | g | -9 | -18 | -27 | 0 | 27 | 9 | -9 | -9 | 0 | 9 | 9 | 0 | -9 | -27 | -36 |
| **p6** | a | 9 | 0 | -18 | -18 | 9 | 36 | 18 | 0 | 0 | -9 | 0 | 0 | -9 | 0 | -18 |
| **p7** | a | 9 | 18 | 0 | -18 | -9 | 18 | 27 | 9 | 9 | -9 | -18 | -9 | -9 | 0 | -9 |
| **p8** | t | -9 | 0 | 9 | -9 | -27 | 0 | 27 | 18 | 0 | 0 | -18 | -27 | 0 | -18 | 9 |
| **p9** | g | -9 | -18 | -9 | 18 | 0 | -18 | 9 | 36 | 18 | 9 | 9 | -9 | -18 | -9 | -9 |
| **p10** | g | -9 | -18 | -27 | 0 | 27 | 9 | -9 | 18 | 27 | 27 | 18 | 18 | 0 | -18 | -18 |
| **p11** | g | -9 | -18 | -27 | -18 | 9 | 18 | 0 | 0 | 9 | 36 | 36 | 27 | 9 | -9 | -27 |
| **p12** | t | -9 | -18 | -27 | -36 | -9 | 0 | 27 | 9 | -9 | 18 | 27 | 27 | 36 | 18 | 0 |
| **p13** | a | 9 | 0 | -18 | -36 | -27 | 0 | 9 | 18 | 18 | 0 | 9 | 18 | 18 | 45 | 27 |
| **p14** | t | -9 | 0 | -9 | -27 | -45 | -18 | 9 | 0 | 9 | 9 | -9 | 0 | 27 | 27 | 54 |

**A = -18; B_match = 9; B_nomatch = -9**

**Alignment:** T: **AACGGA-TGAGGGTAT**
P: **ATCGGAATGG-G-TAT**

Figure 2.2: Alignment Extraction by Tracing Back Through DP Matrix

An alignment occurs in the DP matrix when a particular element $d(i, j)$ exceeds a user-defined constant, $Threshold$. This threshold can be used to specify the number of characters and gaps (as both determined by the constants $A$, $B_{match}$, and $B_{nomatch}$) that the desired alignment should have. The alignment can then be derived by following the pointers between adjacent $d(i, j)$ elements from the element that exceeded the threshold, back up to the first row, $p_1$, or back to the first column, $t_1$, of the DP matrix. A pointer for a particular $d(i, j)$ element points back to the element west,

northwest, or north of it, that was selected by the max() function in Equation 2.4 in calculating $d(i, j)$. Figure 2.2 illustrates such a path taken along these pointers, to find the alignment that resulted in the element $d(14, 15) = 54$. Each diagonal arrow pointed toward an element $d(i, j)$ represents an exact alignment of the characters $p_i$ and $t_j$ in the final alignment. Each vertical arrow represents a gap inserted before the character $t_j$ in the target $t$ to maximize alignment, and each horizontal arrow a gap inserted before the character $p_i$ in the pattern $p$. Indeed, there may be multiple possible paths of pointers to follow back up to the first row or column (as there are in Figure 2.2), and this would represent multiple alignments that result in the same value for the particular $d(i, j)$ that exceeded threshold.

As shown in Equation 2.4, and in the arrows radiating from $d(4, 4)$ in Figure 2.1, the value of the DP matrix element $d(i, j)$ depends on only the matrix elements to the west, northwest, and north of it (along with the characters $p_i$ and $t_j$ and the constants $A$, $B_{match}$, and $B_{nomatch}$). This is an important property of the Smith-Waterman algorithm, since it enables a straightforward implementation in parallel hardware, in that the matrix may be implemented as a systolic array of atomic processing elements (PEs), where each PE is responsible for a single $d(i, j)$ element. However, because the length of the target string $t$, i.e., the dimension $n$, is expected to be significantly larger than the length of the pattern string $p$, i.e., the dimension $m$, implementing the entire DP matrix would likely require an impractical amount of hardware. Thus, existing implementations typically involve some form of partitioning along the $t$ axis (and possibly also the $p$ axis), such that only a portion of the matrix is computed at a time.

A simple form of such partitioning, similar in spirit to that described later in this thesis, is to compute one column of the DP matrix at a time. Thus, only $m$ PEs are required, along with a register for each PE that stores the value of the matrix element in the previous column (i.e., the element $d(i, j - 1)$, if the current PE is responsible for the element $d(i, j)$). Then, once the PE has computed the value of $d(i, j)$, it stores that value in its associated register (overwriting the value $d(i, j - 1)$), to then be used in the computation of $d(i, j + 1)$ in the next column). Figure 2.3 illustrates both the arrangement of PEs and associated registers, and the 2-part data-flow for the computation. The left half of Figure 2.3 depicts the dependencies for each PE, i.e., the values which each PE must take as input to compute the value $d(i, j)$. Note that each PE depends on the value computed by the PE above it (i.e., the value $d(i - 1, j)$), and thus each PE must wait until the one above it has completed its

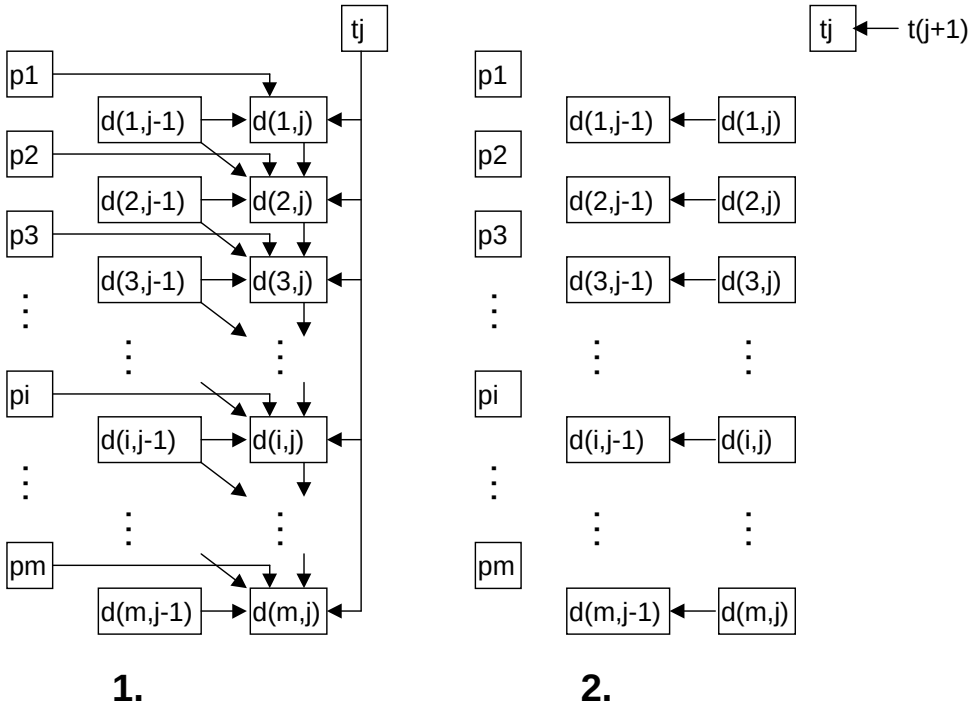


Figure 2.3: 2-Step Computation of DP Matrix Column, 1: Calculation (with data dependencies shown), 2: Data moving

computation before starting its own computation. The right half then illustrates the data movement that takes place once all PEs have finished computation, namely how the $d(i, j)$ value calculated by each PE is stored in its associated register (overwriting the value $d(i, j-1)$), and how the next target string character to be processed, $t_{j+1}$ (i.e., the next column of the DP matrix), is then shifted into the register at the top. Indeed, because each PE is dependent on the result calculated by the PE above, an optimized implementation of this form of partitioning would create a pipeline of the PEs, $m$ stages deep, to ensure completely parallel computation of the matrix column. A match (i.e., a $d(i, j)$ value that exceeds the user-defined $Threshold$) can be found by adding a comparator to each PE, with the comparison result used to toggle a match flag. For clarity, however, the components required for signalling matches are not shown in Figure 2.3.

Another form of partitioning, perhaps more efficient than the one described above, would have a single bottom-left to top-right diagonal of the DP matrix calculated at each time. This method eliminates the need for the pipeline between the PEs, since each PE would then not be dependent on the result calculated by the PE above it. The reasons why this method was not chosen over the one in the preceeding

paragraph are explained later in Chapter 4, which details the design of the biological computation ("BioComp") FPX module.

Such a hardware implementation as those described above is useful for performing a first pass at finding alignments of the pattern $p$ in an exceedingly large target $t$, since its running time is essentially just $O(n)$, compared to $O(m \times n)$ for a software implementation on a uniprocessor machine. However, because this implementation only stores the values for two columns of the DP matrix at a time, it is not possible to extract alignment information directly from the hardware. Rather, a workstation observing the hardware can extract snapshots of the current column being calculated, at periodic intervals, or when the match flag is set, reconstruct in software the DP matrix between the current snapshot and the previous snapshot, and then extract the actual alignment from that reconstructed portion of the matrix. If the number of alignments of the pattern $p$ in target $t$ is expected to be quite small (i.e., significantly less than the ratio $n/m$), then the overhead required for such alignment extraction shouldn't outweigh the inherent speed advantage of the hardware implementation.

## 2.2 Recent Work in Sequence Alignment

Recent work in implementing sequence alignment on parallel hardware has all involved some variation or refinement of the Smith-Waterman algorithm [35] described above, and all involved systolic arrays of parallel processing elements that compute elements in the DP matrix. The target application for these implementations has been typically DNA or protein sequence alignment, although adaptation of this algorithm for generic text searching is possible. In fact, the next section describes two such implementations generalized for text-searching, as does Chapter 4 of this thesis. The hardware platforms chosen for these sequence alignment implementations varied; [2], [5], [11], [12], [14], and [15] present FPGA-based implementations, [13], [22], and [34] present VLSI ASIC implementations, and [41] presents an implementation on a proprietary reconfigurable hardware platform. However, all these implementations shared similar choices of design and optimization, which divide them into two general categories: those which had single-character edit costs (the values $A$, $B_{match}$, and $B_{nomatch}$ described in the section above) hard-wired into their control circuitry, and those which didn't, letting the user specify the values as runtime parameters. The implementations which fall in the first category are [2], [5], [12], [13], [14], [15], [22], and [41]. Those which fall in the second category are [11] and [34].

This is an important distinction, since the choice of using fixed single-character edit costs derives from the original description of the Smith-Waterman algorithm in [35]. Fixing these values means both the values of individual DP matrix elements, and the difference in value between adjacent elements are bounded in range, which allows significant optimization. Specifically, hardware required to store values of DP matrix elements, as well as the hardware required for communication between adjacent elements, may be reduced dramatically through minimal bit encoding. For example, [14] only uses one bit to store edit distances in each matrix element. This optimization allows designers to fit more processing elements per die or per FPGA, and thus to achieve higher computational density. However, a drawback to this optimization is that the fixed single-character edit costs constrain the usability of these sequence alignment implementations. That is, the fixed edit costs chosen by designers of sequence alignment hardware may be not be useful in all variations of sequence alignment that molecular biologists wish to perform. Or, because the minimal bit encoding scheme chosen for storing values of DP matrix elements constrains the range of edit costs, there is an indirect limit on the length of sequences that may be aligned, before overflow in edit cost calculation occurs.

The two implementations which let the user specify single-character edit costs as runtime parameters, [11] and [34], realize this ability for runtime parameters differently. [34], which is an ASIC implementation, stores the single-character edit costs in on-chip SRAM. While making itself useful to a wider range of sequence alignment applications in biological computation, this choice by [34] ultimately constrains its search throughput to less than what the hardware could otherwise allow, since the time required for edit cost computation is dominated by relatively slow SRAM accesses. [11], on the other hand, takes an approach very similar to that used in the FPX BioComp module presented in Chapter 4. [11] is an implementation done on the same Xilinx FPGA device as that used on the implementation in Chapter 4, and it takes advantage of a proprietary feature the Xilinx device to modify the FPGA-programmed circuit at runtime. That is, runtime parameters submitted by the user modify the FPGA logic used for computing edit costs in [11], thereby enabling runtime parametrization of single-character edit costs. The implementation in Chapter 4, on the other hand, doesn't involve any runtime reconfiguration of the FPGA, but it does still offer the same capability by storing the single-character edit costs in registers which the user may set at runtime.

## 2.3 Recent Work in Text-Searching

Recent work in implementing text searching on parallel hardware has followed two general approaches: dynamic programming sequence alignment algorithms adapted for generic string matching, and a hash-based approach which matches against a finite dictionary of keywords (or encodings thereof) stored in RAM. Both approaches lend themselves to different applications in text searching; the dynamic programming implementations lend themselves readily to searching for a single, arbitrary keyword in large text databases, e.g., searching email records, while the hash-based implementations lend themselves to applications that look for a finite set of keywords in text databases, e.g., spell-checking.

Recent work in adapting dynamic programming algorithms for text searching includes [3] and [33]. Both articles describe implementations of very similar sequence alignment algorithms: in [3] the Smith-Waterman algorithm [35], and in [33] the "Wagner and Fisher" algorithm. Both implementations are done with a systolic array of custom 8-bit processing elements on an ASIC. The 8-bit data-path allows both implementations to compute alignment scores (also called edit distances) between arbitrary strings of 8-bit characters, e.g., ASCII text characters. As with most of the implementations of sequence alignment hardware described in the previous section, [3] and [33] hard-code the single-character edit costs into the control circuitry, simplifying the logic of the 8-bit processors, but limiting user configurability.

Recent work in implementing hashing functions in hardware includes [2] and [5], which both describe an implementation of a keyword-searching engine on the FPGA-based SPLASH 2 platform. However, instead of computing edit costs to find inexact matches of the keywords, this implementation has its processing elements compute hashes of each word in the input text. The hashes are then translated into memory addresses and compared to a dictionary of keywords stored in RAM. If the memory address is valid, then the word of input text that created the hash exists in the dictionary, and a bit at that memory location is flipped. Indeed, this method of keyword-searching differs quite dramatically in spirit from other methods based on sequence alignment described this section, in that it can't accommodate inexact keyword matches, or at least if it can, in a very limited fashion.

A final note goes to [19], where two different keyword-searching methods (neither of which fall into the two general approaches mentioned above) were implemented on the same FPGA to compare the search throughput achieved with each method.

The first method was the Knuth, Morris, and Pratt (KMP) keyword-matching algorithm, which guarantees the minimum number of single-character comparisons when comparing two strings. The second method was a simple, "brute force, parallel comparator" that allowed multiple simultaneous character comparisons. Because of an FPGA's capacity for parallel computation, the authors of [19] found the simpler, brute force comparator provided greater search throughput than the KMP keyword-matching algorithm, which is optimized for execution on uniprocessor machines. This article is mentioned as a demonstration of precedent, since Chapter 4 of this thesis presents an FPGA implementation of the Smith-Waterman algorithm [35], that can offer similar throughput gains in generic text and genome searching, over implementations optimized for uniprocessor machines.

## 2.4 FPX, WUGS and NCHARGE Overview

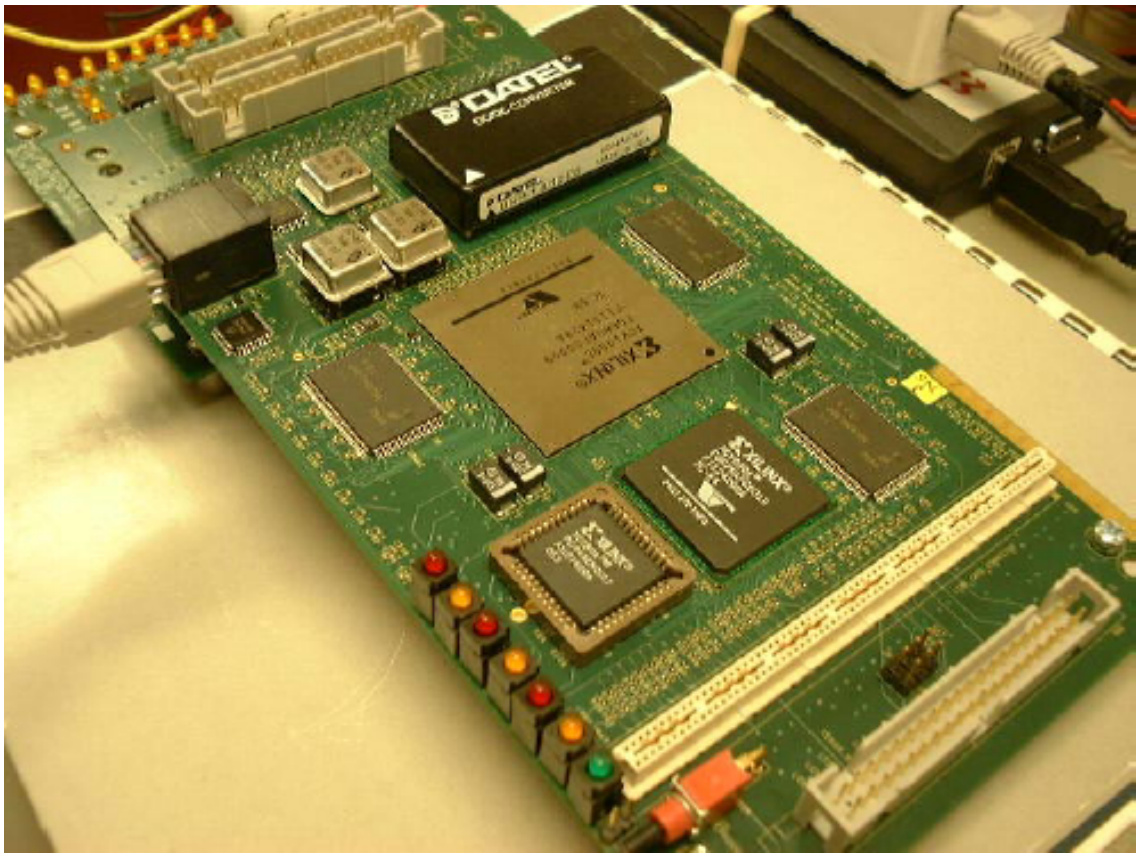


Figure 2.4: Field Programmable Port Extender (FPX) [24]

The Field Programmable Port Extender (FPX) [24] [25] [26], shown in Figure 2.4, is an FPGA-based circuit board designed for use with the Washington University Gigabit Switch (WUGS) [39]. The WUGS is an experimental Asynchronous Transfer Mode (ATM) switch developed at the Department of Computer Science in Washington University. The FPX, which sits on a port on the WUGS switch fabric, provides a reconfigurable hardware platform to perform arbitrary functions on ATM cells passing through that port. While FPX was originally intended to perform networking applications on the traffic traversing the WUGS switch fabric (i.e., Fast IP Lookup, JPEG encoding, and DES encryption), this thesis focuses on using the FPX as a development platform for hardware-based searching of streaming data. That is, the ATM traffic traversing the WUGS switch fabric is similar in speed (1.2 Gb/sec) to the data that streams from the magnetic read head on a hard drive. Thus, stream-searching applications developed on the FPX could easily be ported to a conventional hard drive that has been augmented to send the data streaming off its read head through reconfigurable hardware. However, this thesis only details stream-searching applications, as developed on the FPX, and leaves the details of porting such applications to an FPGA-augmented hard drive to further research.

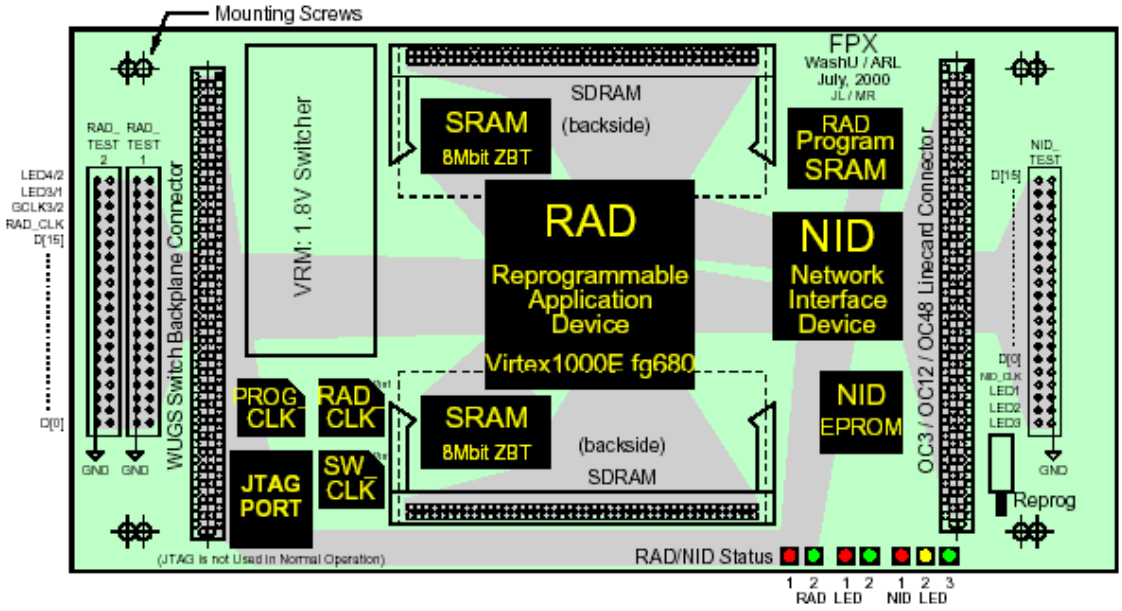


Figure 2.5: FPX Printed Circuit Board[24]

As depicted in Figure 2.5, the FPX features two FPGAs, the Reprogrammable Application Device (RAD), which is available to the user to program as he wishes,

and the Network Interface Device (NID), which is responsible for routing ATM traffic to and from the RAD and for reprogramming the RAD. The features of the FPX and even the RAD are quite manifold, but only those relevant to this thesis will be discussed in detail, namely software development for RAD, the RADTEST interface, and the NCHARGE control software, which manages the FPX from a remote workstation.

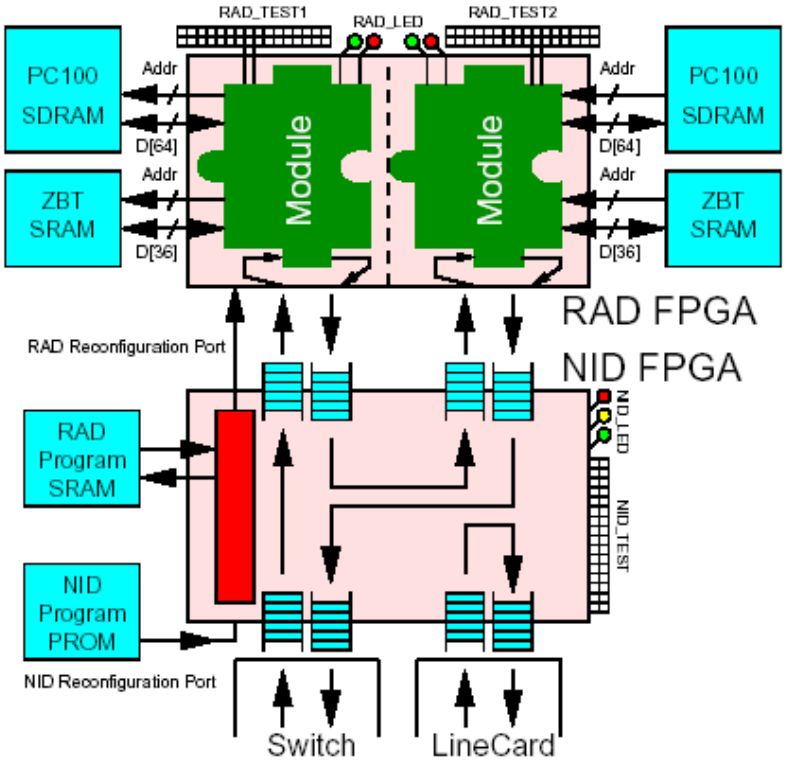


Figure 2.6: Logical Arrangement of FPX Components [24]

The RAD is a large Xilinx FPGA with (at last count) a 2-million logic gate capacity, and is entirely available to the user to program as desired. Figure 2.6 illustrates the various interfaces between the RAD and other components on the FPX. Of importance are the four data interfaces between the RAD and the NID, each 32 bits wide, and the RADTEST1 and RADTEST2 interfaces, which combined provide a 32-bit interface between the RAD and an external device. The architecture of the FPX, primarily the four data paths between the RAD and NIC, dictate that user-developed applications programmed onto the RAD be contained in one of two independent modules, where each module is responsible for a data path to and from the NID. While this scheme allows for two independent applications to run on the RAD at

a time, the nature of stream-searching applications developed for this thesis, where search data would be submitted to the RAD from a remote source, and then search results returned back to that remote source, rendered the second module unnecessary. Thus, all FPX modules discussed in this thesis were developed under the assumption that the second module slot would either be empty, or would contain a benign place holder that requires negligible chip resources.

The basis of any RAD module is a finite state machine that handles data passing over the two 32-bit interfaces between it and the NID. Each of these interfaces is a UTOPIA interface, with data formatted into ATM cells, such that this fundamental state machine is simply one that understands the ATM protocol. Each raw ATM cell contains two 32-bit header words for every 12 32-bit data words, and so this state machine spends most of the time forwarding ATM data between the two NID interfaces (during which time, of course, a custom developed module would be doing something meaningful with the data). If one treats the time and computation resources dedicated to handling the ATM headers as negligible overhead, then one can see how the RAD module is essentially confronted with continuous, streaming data, 32 bits per clock tick. At 62.5 MHz clock frequency, the maximum supported by the NID, this leads to a data throughput of 250 MB/sec.[1] For comparison, SCSI hard drives currently support bus transfer speeds up to 80 MB/sec, allowing one to immediately see the applicability of a stream-searching application developed on the RAD FPGA, when ported over to an FPGA-augmented hard drive.

The RADTEST interface, shown at the top of Figure 2.6 as two 40-pin connectors, allows the user to attach up to 32 pins of the RAD to an external device. As originally envisioned, this interface lets the user route arbitrary signals in the RAD module to external pins for observation on an oscilloscope or logic analyzer. However, the reconfigurable nature of the RAD means these 32 pins may also be used as additional inputs to a RAD module. This is quite fortunate since the ATAPI/IDE protocol defines 32 meaningful signals for the bus between IDE hard drives and their hosts [27]. In fact, Chapter 3 details how the RADTEST interface was hi-jacked to let the FPX snoop an IDE bus, and thus observe data outputted by the hard drive.

The final note on the FPX goes to NCHARGE [24] [36], the control software which runs on a Linux or NetBSD workstation attached to a port of a WUGS

[1]The WUGS switch fabric supports a port-to-port speed of 1.2 Gb/sec. However, a NID at a particular port could see a data throughput greater than 1.2 Gb/sec when more than one port is sending traffic to it.

populated with one or more FPXs. This software is responsible for downloading FPGA bit files to to the FPX to program the RAD, setting up routing parameters for the NID, and observing user-defined status messages from modules on the RAD. Additional functionality, such as parsing test ATM cells to send to the FPX and observing the results, is also available from NCHARGE, but the interface between the workstation and the WUGS is constrained to 150 Mb/sec, limiting the usefulness of NCHARGE for routing large volumes of data between it and an FPX. Thus, use of NCHARGE for the FPX modules described in this thesis is limited to testing functionality, setting up runtime parameters (e.g., the search query for modules that perform stream-searching), and viewing search results.

# Chapter 3

# IDE Bus Snooper

## 3.1 Overview

The IDE Bus Snooper is an FPX whose RAD FPGA has been programmed to interpret the ATAPI/IDE protocol [27], so that it can recognize data bursts initiated by the hard drive [1] attached to the IDE bus, capture the contents of those data bursts, and then inject the data (encapsulated within ATM cells) into the WUGS switch fabric for further processing. The FPX and WUGS were chosen as a development platform for this device because they offer an environment with an array of 2-million gate capacity FPGAs coupled together by a Gb/sec interconnection network. That is, an FPX programmed as an IDE Bus Snooper can send its captured data over the WUGS switch fabric to one or more FPXs, which have been programmed to search or process the captured IDE data. Thus, the WUGS and FPX infrastructure lends itself quite well to developing hardware-based search modules that are ultimately destined for integration directly into a hard drive's controller logic.

Figure 3.1 illustrates how this connection between the IDE bus and the RAD FPGA is made using 32 test pins available on the RAD FPGA, to observe the 16 IDE data lines and the 15 relevant IDE control lines. Unfortunately, voltage incompatibility between the RAD FPGA, which has a 3.3V supply voltage, and the 5V IDE bus required that a custom-built PCB with voltage translation buffers sit between the FPGA and the IDE bus. A limitation of this voltage translation stage was that the IDE Snooper could only passively observe events on the IDE bus, and not initiate bus transactions itself. Although this did lead to a far simpler (and faster) state machine

[1] The drive used in these tests was a Seagate ST320414A 20 GB ATAPI/IDE hard drive, formatted with an ext2 file-system.

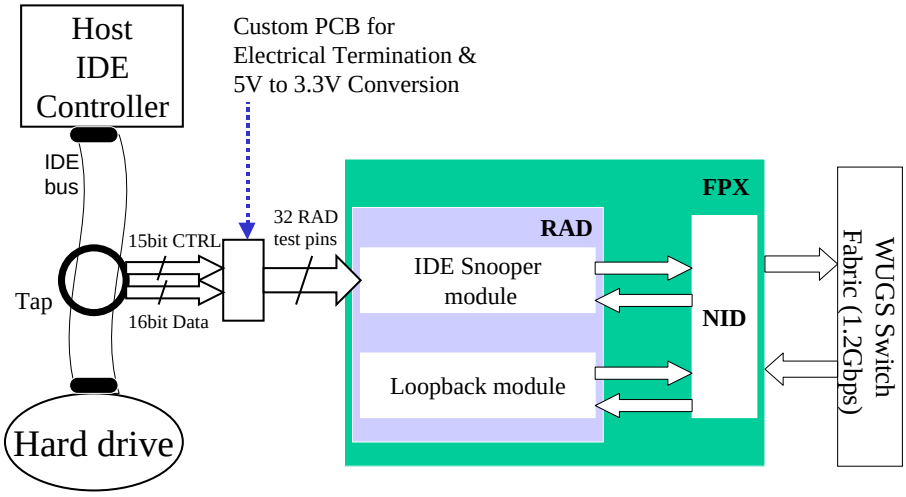


Figure 3.1: Block Diagram of IDE Snooper as FPX Module

in the IDE Snooper, it required that any control commands to the hard drive being snooped be issued over a path separate from the IDE Bus Snooper. In context of the development platform illustrated in this section, this control path was achieved by having the host workstation (containing the IDE bus being snooped) mount a file-system on that drive in normal fashion, and perform traditional read/write tasks on that file-system.

## 3.2 RAD FPGA Module Design

A simplified, conceptual block diagram of the IDE Bus Snooper RAD Module is shown in Figure 3.2, depicting the module's major components and Finite State Machines (FSMs). Of particular importance is the dual port RAM in the center of the diagram, as this component comprises most of the interface between the IDE bus being snooped and the UTOPIA bus to the NID. That is, the captured data from the IDE bus are written into to one port of this RAM, and then read out the other port as words in an ATM cell payload, with the RAM functioning as a ring buffer. The RAM is, however, a ring buffer with asymmetrical ports, since the data words captured from the IDE bus are 16 bits wide, while the data words written out to the NID as ATM payload must be 32 bits wide. The three FSMs also shown in Figure 3.2 are responsible for managing both ports of the RAM, to ensure no captured IDE data are lost during their trip to the NID. The two counters shown in Figure 3.2 generate the read and

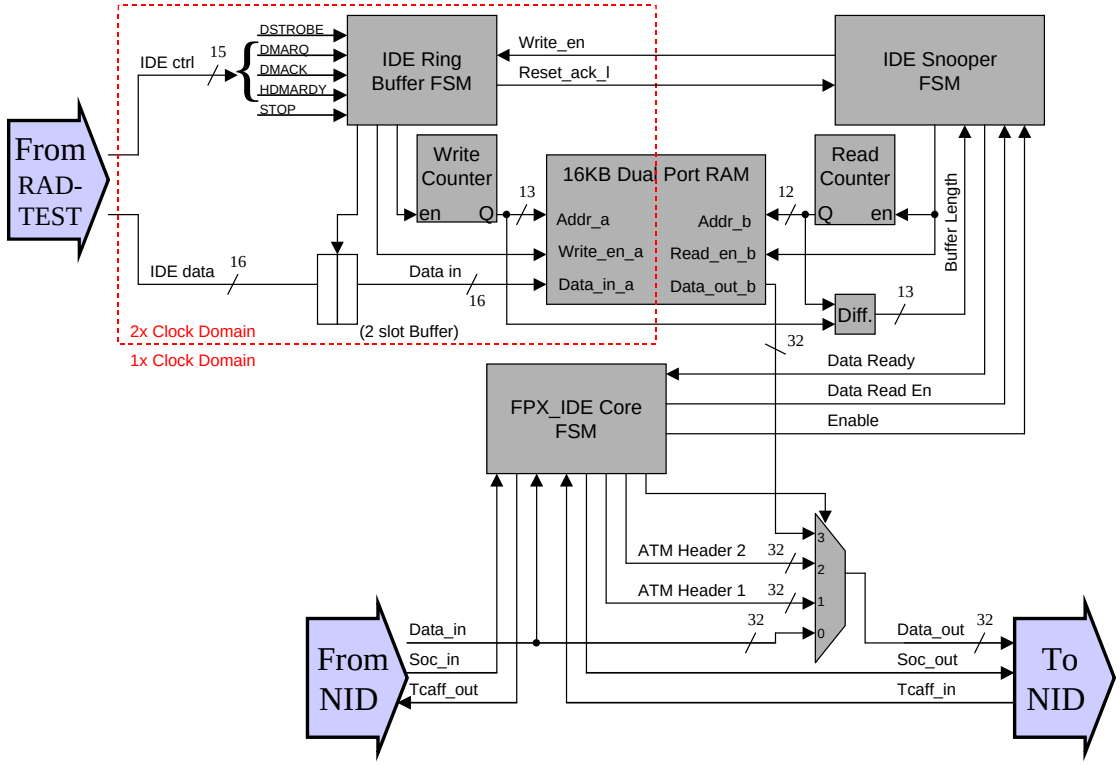


Figure 3.2: Block Diagram for IDE Bus Snooper RAD Module

write addresses for the dual port RAM, with the IDE Snooper FSM tracking the difference between those addresses to monitor the ring buffer length.

The IDE Bus Snooper RAD Module operates with two clock domains, as illustrated with the dashed line in Figure 3.2. This is done so that the write port of the ring buffer and the IDE Ring Buffer FSM controlling it may run at a higher frequency, i.e., double that of the rest of the module. This allows the data and control signals sampled from the IDE bus to be over-sampled to increase accuracy. At the time of this writing, the primary clock frequency for the RAD was 62.5 MHz, making the doubled frequency 125 MHz. As per Nyquist Sampling Theorem, this is at least theoretically enough to reliably sample IDE bus signals up to 62.5 MHz, above the 50 MHz specified in [27].

### 3.2.1 FPX IDE Core Finite State Machine

The IDE Core Finite State Machine is responsible primarily for the UTOPIA interface between the RAD and NID. This includes managing the following tasks:

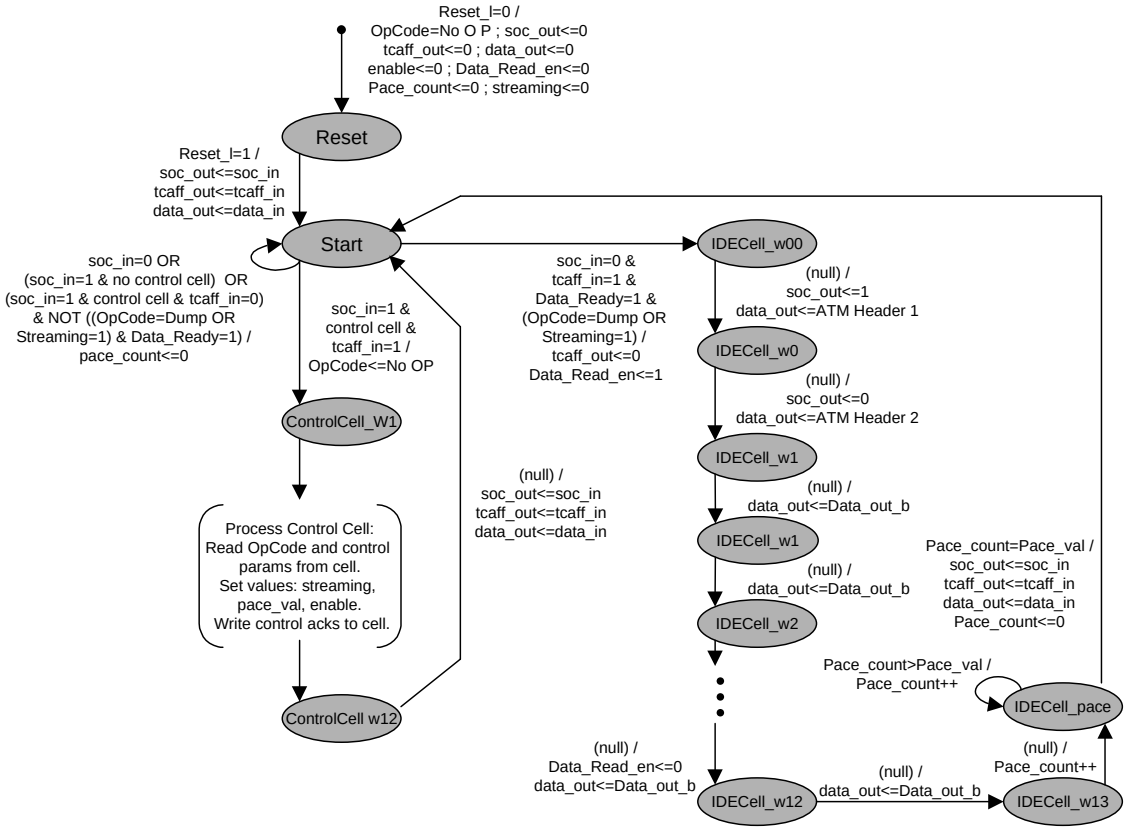


Figure 3.3: Abbreviated State Diagram for FPX IDE Core FSM

- Accept incoming ATM control cells from the NID, read relevant control parameters from their payload, and write appropriate responses into the payload of the control response cells sent back to the NID.
- Encapsulate IDE data from the ring buffer into ATM cells and send them out to the NID.
- Forward incoming ATM cells not destined for the RAD through the IDE Snooper Module, back to the NID.

Figure 3.3 shows the abbreviated state diagram of the IDE Core FSM. This diagram shows two separate circuits, which each handle one of the first two tasks listed above. The left circuit, which has been abbreviated for clarity, handles incoming control cells from the NID, as well as the control responses sent back to the NID. This task includes extracting runtime parameters from the control cell payload, enabling or disabling the IDE Snooper FSM if the control cell requests it, and, if the control cell

is querying a runtime parameter, writing that value into the payload of the outgoing control cell response. The right circuit in the state diagram in Figure 3.3 handles the control of the IDE Snooper FSM and the forwarding of IDE data from the ring buffer out to the NID. The right circuit also features a state, "IDECell pace," for inserting arbitrary wait periods between outgoing cells of IDE data. Pacing between outgoing cells is necessary, since although the NID and the WUGS switch fabric can accept cells as fast as the RAD can send them out, the device at the destination port for these cells possibly could not. Specifically, the device that forwards ATM cells from the WUGS switch fabric to a workstation for observation and debugging could only accept cells at a small fraction of the switch fabric bandwidth.

More detail on the format and handling of control cells, as well as methods for generating such cells and sending them to the FPX, are available in extant literature on the NCHARGE control software in [4], [24], [25], and [36]. Table 3.1 gives a synopsis of the control functions the IDE Core FSM provides to NCHARGE.

### 3.2.2 FPX IDE Snooper Finite State Machine

The IDE Snooper finite state machine is responsible for managing the read port of the ring buffer shown in Figure 3.2, and for relaying enable/disable commands from the IDE Core FSM to the IDE Ring Buffer FSM. Figure 3.4 depicts a simplified state diagram for the IDE Snooper FSM, showing, primarily, the progression of states involved in outputting 12 32-bit words of IDE data from the ring buffer to fill an ATM cell. Much of the complexity of this state diagram stems from the logic that guarantees the ring buffer only output IDE data, once it has at least enough to fill an ATM cell. This logic can also be seen in the "Buffer Length" and "Data Ready" signals in Figures 3.2 and 3.4, which are used to notify the IDE Snooper and IDE Core FSMs, respectively, when enough IDE has been captured in the ring buffer.

### 3.2.3 FPX IDE Ring Buffer Finite State Machine

The final state machine in the IDE Bus Snooper RAD Module is the Ring Buffer FSM, which monitors control signals on the IDE bus, and then enables the write port of the ring buffer when an IDE data burst is detected. Figure 3.5 depicts a simplified state diagram for the Ring Buffer FSM, showing the progression of states involved in recognizing when a data burst on the IDE bus begins, and then in capturing each 16-bit data word once the burst is under way. A short note should be made about

Table 3.1: NCHARGE Commands for IDE Bus Snooper Control Module

| |
|---|
| Enable and disable the IDE Snooper FSM (makes IDE Ring Buffer FSM start or stop monitoring IDE bus events) |
| Tell the ring buffer (via the IDE Snooper FSM) to dump its contents out to the NID |
| Set or query following runtime parameters:<br>• ATM Headers to place on cells with captured IDE data (specifies destination in switch fabric)<br>• Pacing value, i.e., wait period to insert between each outgoing IDE data cell<br>• Streaming mode, i.e., if the ring buffer should dump its contents as soon it has enough IDE data to fill an ATM cell, or if it should wait for a dump command |
| Query the following values:<br>• Current value of IDE control signals DSTROBE, HDMARDY, DMARQ, DMACK, STOP<br>• Current ring buffer size<br>• Current write and read addresses<br>• Total number of ATM cells of IDE data sent out |

this FSM's interaction with other components in the IDE Bus Snooper module, since many of those components are in a different clock domain. Specifically, all signals shown in Figure 3.2 which cross the dashed line delimiting the two clock domains must first pass through a stage of flops upon entering the new domain. This is done to ensure that any logic in the IDE Bus Snooper module only operates on signals synchronous to its respective clock, and, thus, that timing integrity is preserved.

Although the IDE bus has 15 control signals, the Ring Buffer FSM only needs to monitor 5 signals to recognize a data burst that originates from the hard drive: DMARQ, DMACK, HDMARDY, DSTROBE, and STOP. Of these 5 signals, the most crucial control signal is DSTROBE, which acts as a double-edged data clock for the duration of the burst. Figures 3.6 through 3.9 are timing diagrams extracted from

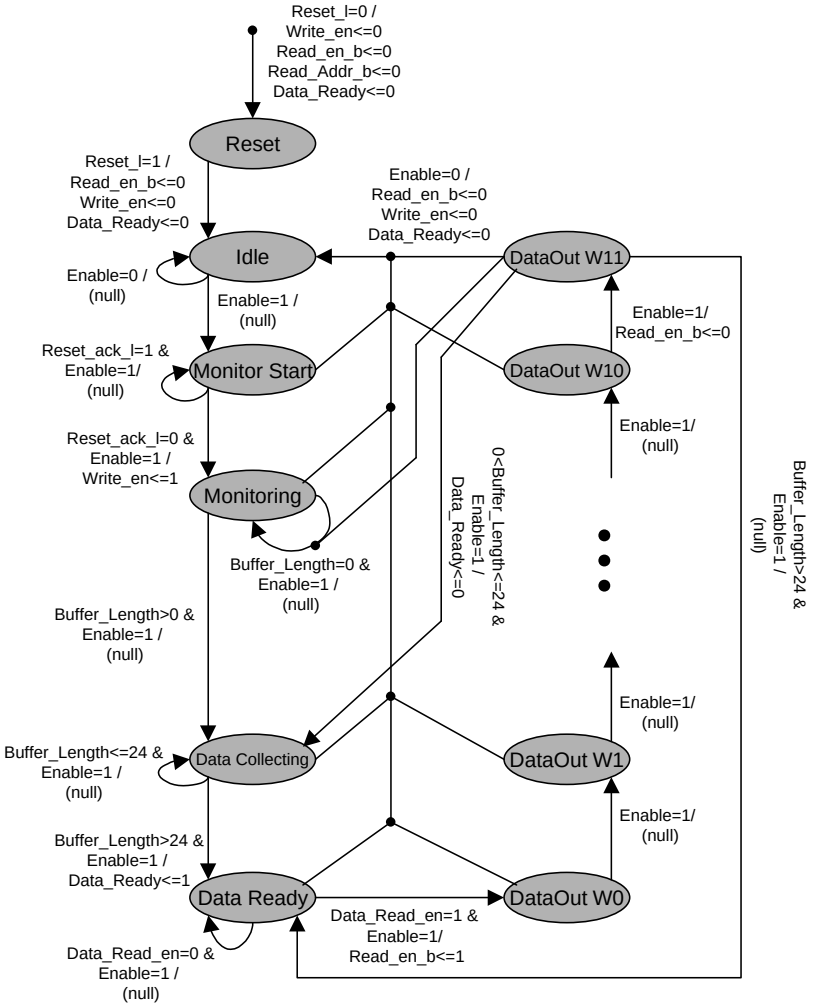


Figure 3.4: State Diagram for IDE Snooper FSM

[27], which illustrate these signals during an IDE data burst. Most of the complexity in the state diagram in Figure 3.5 is involved in finding the rising and falling edges on DSTROBE once the burst is under way, and then enabling the write port of the ring buffer to capture the value of the 16 IDE data lines at that moment. Because the IDE bus is not synchronized to the FPX in any way, this actually proves to be a difficult process when setup and hold times given in [27] are taken into account. To boost the sampling accuracy of this process, the captured IDE data first pass through a 2-stage buffer, shown in Figure 3.2. This buffer lets the Ring Buffer FSM select from three versions of the sampled IDE data value, i.e., from the current clock cycle and the two previous, depending on which best aligns with the edge on DSTROBE.

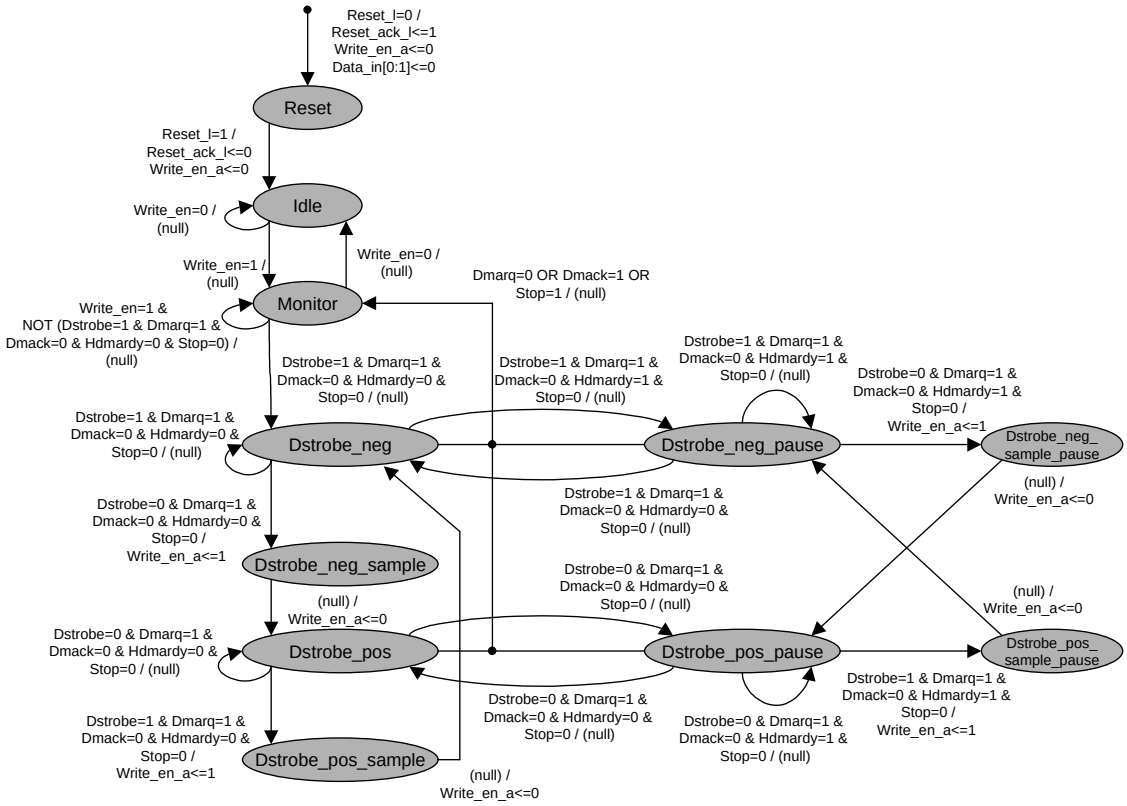


Figure 3.5: State Diagram for IDE Ring Buffer FSM

An ideal version of this component of the Snooper would have included an asynchronous circuit driven just by the DSTROBE signal, thus guaranteeing synchronization with the IDE data bursts, but as the architecture of the RAD FPGA does not allow logic to be triggered by a dual-edged clock such as DSTROBE, this ideal version was not implementable. As explained below, this detail unfortunately ended up preventing the Snooper from actually reaching its maximum theoretical transfer rate.

## 3.3 Physical Design

The physical connection between the hard drive being snooped and the RAD occurs over the two 40-pin RADTEST connectors on the FPX, which are shown in Figure 2.5. These two connectors, which together provide 32 signal lines to the RAD, were originally intended by the FPX designers to provide external taps to arbitrary signals inside the RAD, for debugging with a logic analyzer. However, the since the

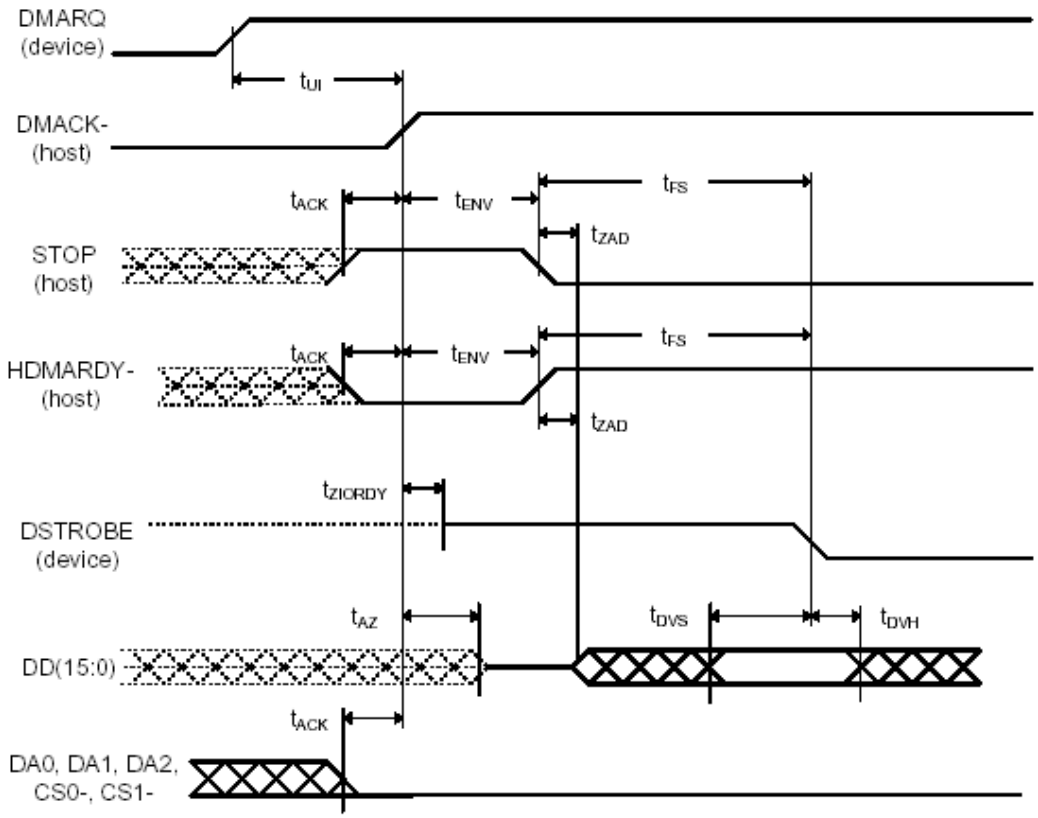


Figure 3.6: Timing Diagram for Data Burst Start from IDE Hard Drive [27]

I/O buffers in the RAD FPGA can be reconfigured to operate as output, input, or bidirectional buffers, these 32 signal lines happened to provide enough inputs to the RAD to let it completely sample the IDE bus. Thus, the FPX board could essentially be used as-is to tap into an IDE bus and snoop its traffic.

However, a non-trivial detail that arose during the design of the Bus Snooper was the voltage incompatibility between the 3.3V input buffers on the RAD and the 5V IDE bus. This incompatibility led to the design and fabrication of a custom PCB to hold voltage translation buffers, which would sit between the IDE bus and the FPX. Figure 3.10 shows the annotated mask for this custom PCB. The red colored regions represent areas on the top layer of the PCB where metal was left, the blue regions where metal was left on the bottom layer. The 40-pin connector in the center of the board is for the IDE bus ribbon cable, with a breakout pattern for the 31 relevant IDE signals shown in the traces that fan out on each side of this connector. Because all signals tapped from the IDE bus are sampled passively, all signals pass through the same series of components on the voltage translation board, allowing

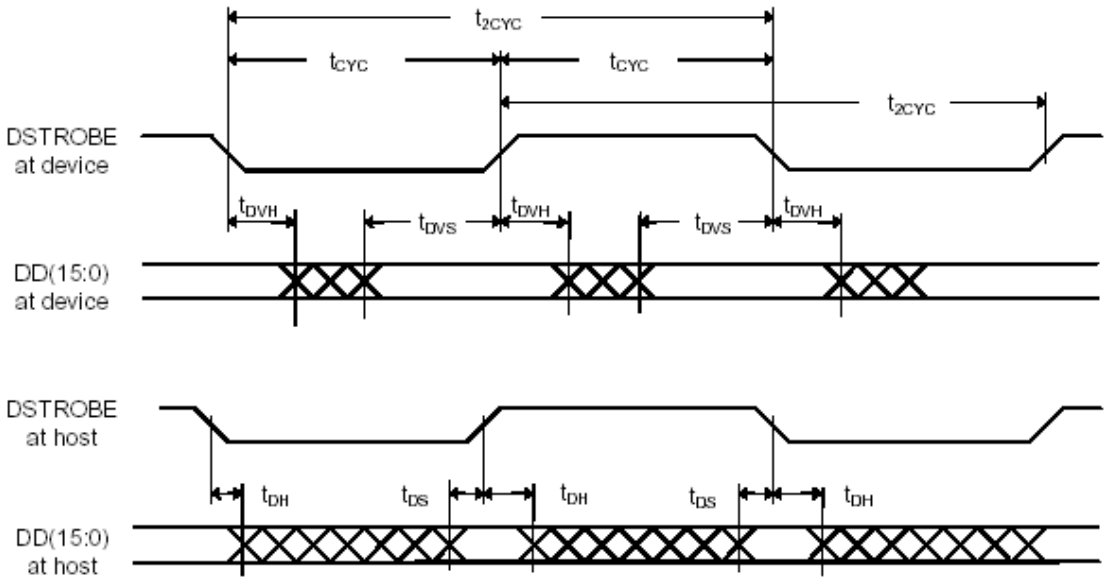


Figure 3.7: Timing Diagram for Data Burst Under Way from IDE Hard Drive [27]

it to be divided into roughly symmetrical quadrants, with 8 or 7 IDE signals each. The two lower quadrants handle the 16 IDE data lines, and two upper quadrants the 15 relevant IDE control lines. The components which the IDE signals pass through for voltage conversion (shown in a left-right progression if viewed on the right half of the board in Figure 3.10) are a 16-pin DIP for in-line termination resistors, a 10-pin strip for pull-up/pull-down resistors, the 5V-3.3V voltage translation buffers themselves (20-pin DIPs), and finally one of the two 40-pin RADTEST connectors. The termination and pull-up/pull-down resistors were ultimately removed or shorted, since they proved to be unnecessary. That is, although these resistors are explicity required in the ATAPI/IDE specification [27], this PCB was not intended for actually initiating any IDE bus transactions (meaning it would never drive current into the IDE bus). Thus, the high input impedance of the voltage translation buffers turned out to be enough to electrically isolate the PCB from the IDE bus. Indeed, this electrical isolation has been empirically verified for IDE bus speeds up to 50 MHz, which was the upper speed limit defined in [27] as of this writing.

Unfortunately, limitations of the hardware used to implement the Bus Snooper, especially issues with synchronization with the IDE bus, prevent it from working perfectly. Specifically, the FPGA architecture of the RAD only allows logic sensitive to single-edged clocks, meaning an asynchronous circuit driven by dual-edge data clock DSTROBE couldn't be implemented. Thus, the Snooper has yet to work reliably

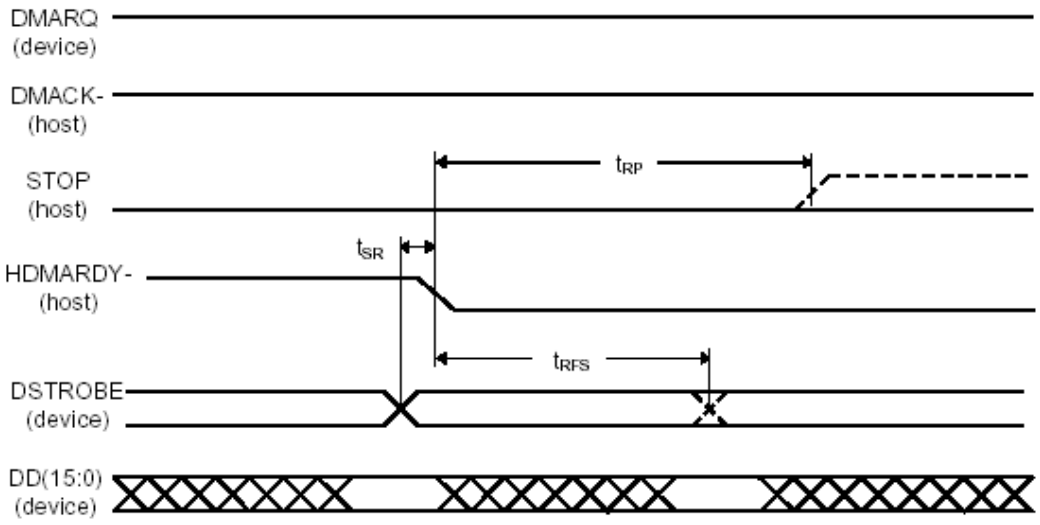


Figure 3.8: Timing Diagram for Data Burst Pause from IDE Hard Drive [27]

at the 50 MHz IDE bus speed mentioned above. In addition, the hard drive itself proved unable to sustain a data throughput above 40.5 MB/sec. Nevertheless, the Snooper hardware in its current state does work reliably at lower IDE bus speeds. Since the Snooper was constructed for experimentation and proof of concept, these limitations imposed by the development and testing environment are acceptable to the author. Later versions of this hardware, of course, will overcome these imperfections, but that is outside the scope of this thesis.

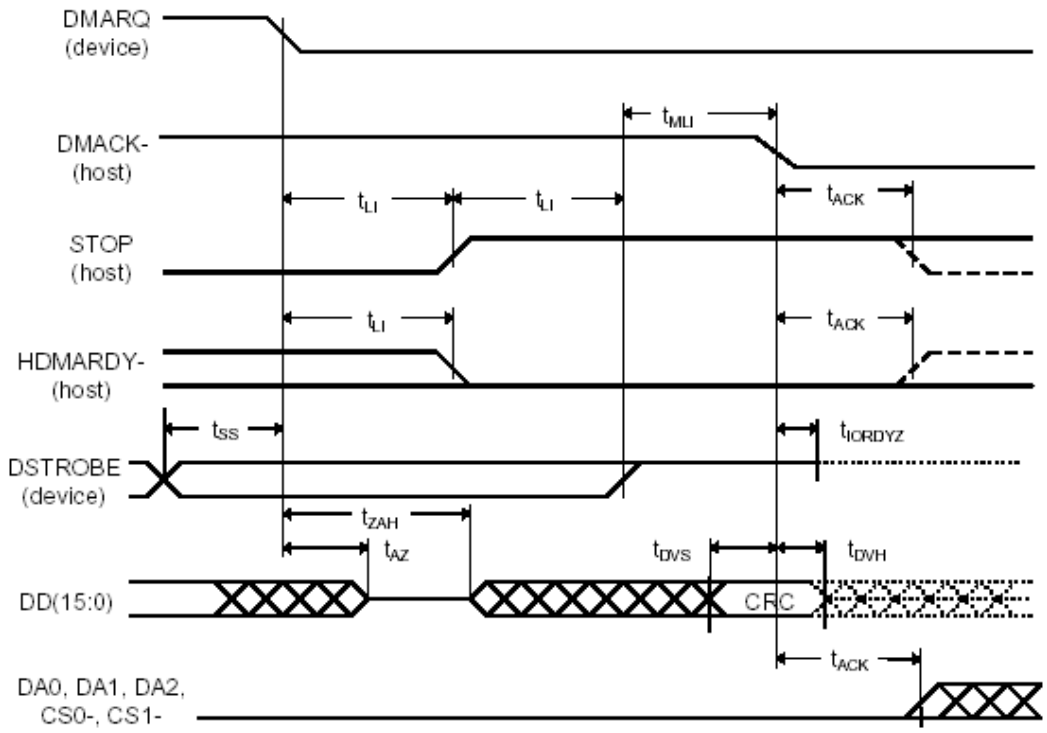


Figure 3.9: Timing Diagram for Data Burst End from IDE Hard Drive [27]

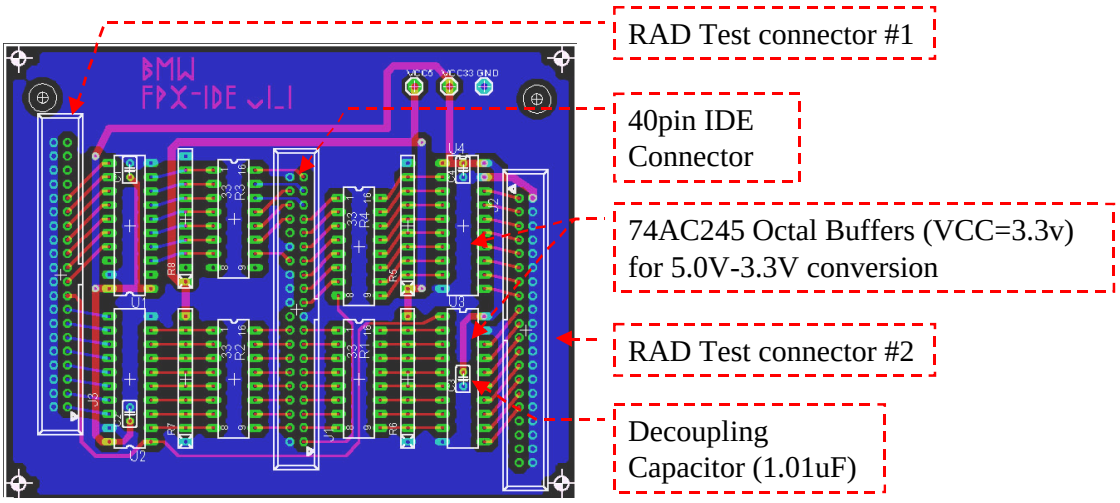


Figure 3.10: Mask for Custom Voltage Translation PCB of IDE Bus Snooper

# Chapter 4

# Sequence Alignment on the FPX

## 4.1 Overview

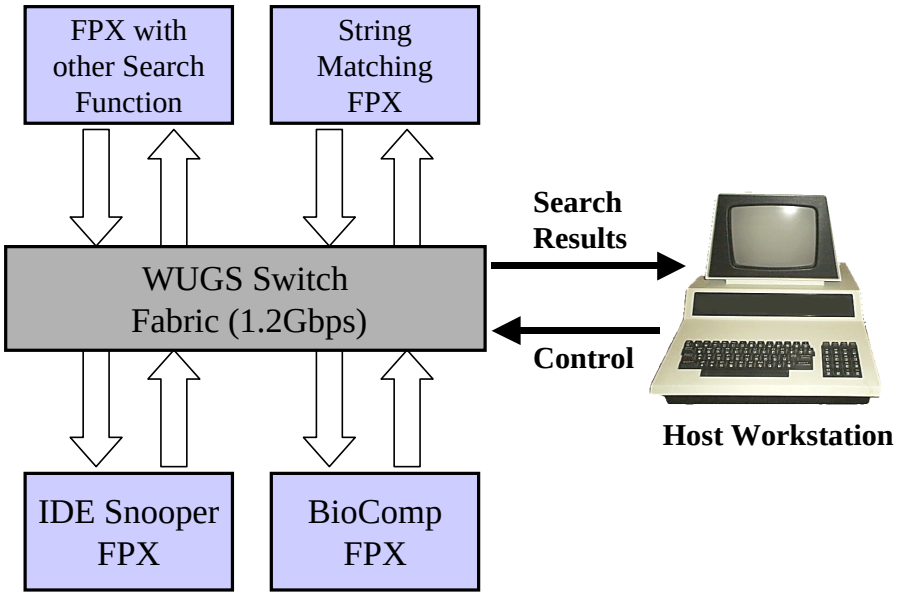


Figure 4.1: FPX as Testbed for Hardware-based Searching

As explained in Chapter 2, the FPX [26] and WUGS [39] infrastructure provides an excellent testbed for developing and implementing stream-searching applications in hardware. The Smith-Waterman local alignment algorithm [35], which was also explained in Chapter 2, was chosen as an example search application to illustrate the advantages of using the FPX for such development, because the algorithm lends itself quite well to optimization through parallelism and pipelining. This chapter will detail the design and implementation of an FPX module, the BioComp Module, that

is based on the Smith-Waterman algorithm. The following sections will present the architecture of the BioComp module, detailing its internal components, with a special emphasis on the systolic array in the module's core that computes the DP matrix elements. A final section will also present a simple, Web-based user interface for the BioComp module that runs on top of the FPX control software NCHARGE [24].

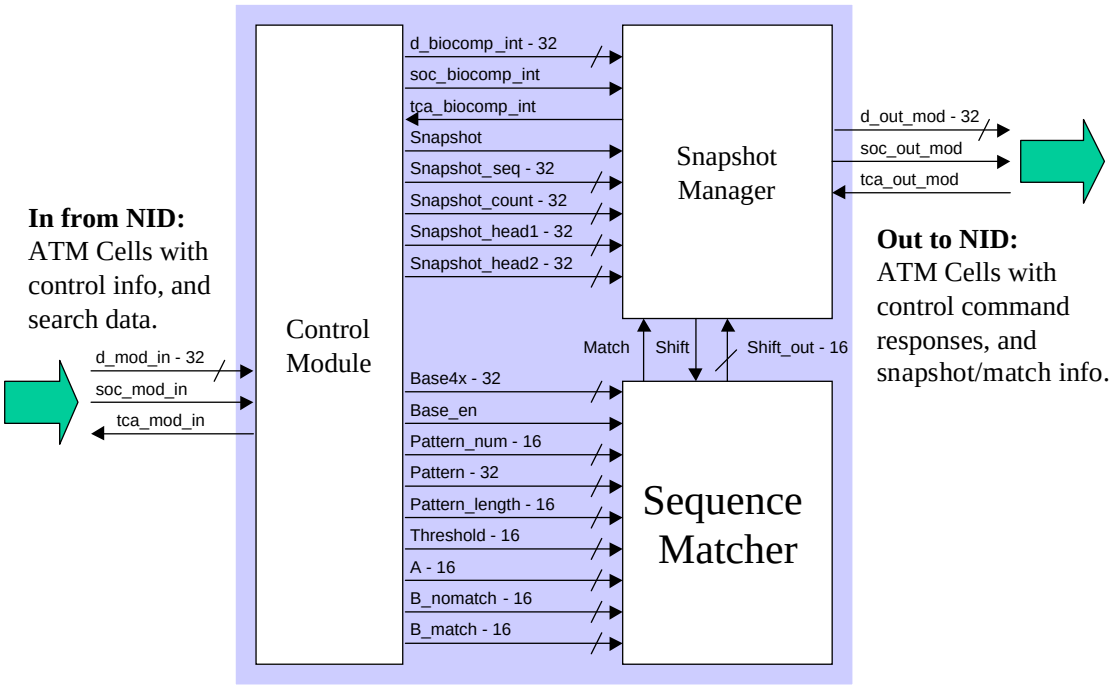


Figure 4.2: Diagram of BioComp Internal Components

Figure 4.2 shows a block diagram of the three internal components of the BioComp FPX module: the Sequence Matcher containing the systolic array, the Control Module, and the Snapshot Manager.[1] The Control Module at left receives incoming cells from the NID, processes control cells to extract runtime parameters, and routes the payload of target data cells to the Sequence Matcher. In addition, the Control Module oversees the operation of the Sequence Matcher and Snapshot Manager, supplying each with runtime parameters extracted from control cells. The Snapshot Manager at top right handles the extraction of snapshots of the systolic array and match results from the Sequence Matcher, and then packages those data into ATM cells for transmission out to the NID. The Sequence Matcher at bottom

[1]The Control Module and Snapshot Manager are derived from the designs of Maggie Qiong Zhang and Brian Bruggeman, respectively, done in context of the Spring 2002 Course EE563.

right contains the systolic array of processing elements which compute DP matrix values and evaluate potential matches.

A unique feature of this implementation of the Smith-Waterman algorithm, compared to those mentioned in Chapter 2, is that the BioComp Module accepts parameters like the gap penalty $A$, single-character match/mismatch weightings $B_{match}$ and $B_{nomatch}$, and the search pattern itself, as runtime parameters that are stored in registers among the computation logic. Many of the other implementations mentioned Chapter 2 either have these parameters hard-coded into their computation logic, limiting those implementations' applicability, or store them in banks of RAM, causing memory bandwidth limits to constrain the implementations' speed of operation.

The data-path across the BioComp module progresses from left to right as seen in Figure 4.2. Incoming ATM cells from the NID enter the Control Module over the 32-bit signal "d_mod_in." If the cells are target data cells, the Control Module routes the cells' 32-bit payload words into the Sequence Matcher over the signal "base_4x," and enables the Matcher's array by asserting "base_en". Match results are reported back to the user by the Snapshot Manager, which sends a snapshot of the current state of the systolic array, along with a match flag, out to the NID over the signal "d_out_mod," for eventual routing onto the user's host workstation. A snapshot consists of the values in the systolic array's fourth column, which is shifted out of the Sequence Matcher row by row over the signal "shift_out." The match flag indicates whether the Sequence Matcher found a match in the target data it has processed since the last snapshot was made. Snapshots would need to extracted from the array and transmitted back to the host workstation in regular intervals, to permit the user to track the progress of the BioComp Module across incoming target data. To obtain the exact alignment that triggered the match flag, the user must take the array columns outputted in the snapshots before and after that match occurred and regenerate the DP matrix between those columns. From there, the user can follow the alignment extraction procedure outlined in Chapter 2.

At the time of this writing, the synthesized BioComp module occupies 99 percent of the gates on the RAD, a Xilinx XCV2000E-6 device. This is with the systolic array instantiated out to 38 rows. The maximum simulated clock frequency of the synthesized module is 27 MHz, and the module has been tested to 25 MHz in actual hardware. Nevertheless, at 25 MHz the module can sustain a search throughput of 100 million target characters per second, or 100 MB/s. This throughput exceeds the 40.5 MB/sec at which IDE Snooper module, described in Chapter 3, could potentially

capture IDE data and forward them onto the WUGS switch fabric. However, this limitation is actually imposed by the hard drive attached to the IDE bus, as the ATAPI/IDE specification [27] lists the maximum throughput of the IDE bus itself as 100 MB/s.

## 4.2 Sequence Matcher Core

The BioComp module was designed to accept incoming target data at 32 bits per clock cycle, i.e., the full width of the data-path between the RAD and the NID, meaning the Sequence Matcher must accept data 4 bytes at a time. The example hardware implementation of the Smith-Waterman algorithm shown in Figure 2.3, by comparison, would only accept target data one character at a time. Thus, the Matcher's systolic array must compute 4 columns of the DP matrix at a time, as depicted in Figure 4.3. This is roughly equivalent to 4 instances of the column in Figure 2.3 running in parallel, except that computation of each row in the Sequence Matcher's array is pipelined to reduce the number of logic stages to traverse in each clock cycle.

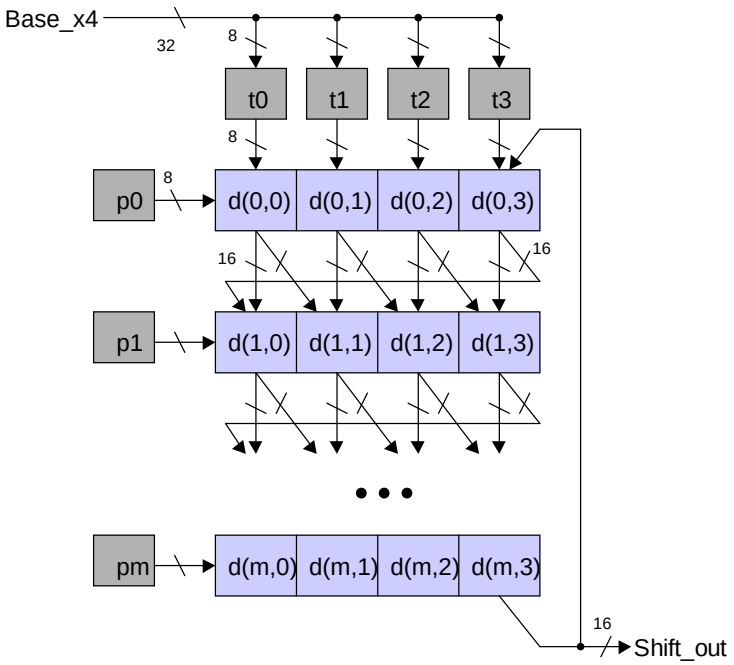


Figure 4.3: Block Diagram of Systolic Array

Figure 4.3 shows a conceptual block diagram of the systolic array contained within the Sequence Matcher. At top of the array is a 32-bit register which accepts a group of 4 target characters from the Control Module on each clock cycle. To the left of the array's lowest order column are 8-bit registers holding the pattern characters, with each character aligned to a row of the array. The longest pattern which the array can accept is simply that with the same number of characters as the array has rows, i.e., a pattern $m$ characters long. As explained below, this value $m$ is actually parametrized, making the Sequence Matcher's systolic array scalable up to the gate capacity of the RAD FPGA.

A width of 16 bits was chosen for all elements in the systolic array, including for the parameters $A$, $B_{match}$, $B_{nomatch}$, and the match threshold. Since the target and pattern characters only affect the value of $B(i, j)$, i.e., whether it is $B_{match}$ or $B_{nomatch}$, the width of the target and pattern characters is independent of the width of elements in the array. To allow the BioComp Module to operate directly on bytes, e.g., 8-bit ASCII characters, a width of 8 bits was thus chosen for the target and pattern characters. Although the BioComp Module is intended to operate on streaming target data, i.e., such that the length of the target string can be effectively infinite, the signed 16-bit arithmetic of systolic array can indirectly impose a limit. That is, given a long enough target string sparsely populated by pattern characters, successive additions of the negative values $B_{nomatch}$ or $A$ can lead to underflow in an element of the the array, resulting in a false match result. The converse is also true, i.e., given a long enough target string densely populated by characters in the pattern, where the pattern is comprised of the same character, successive additions of the positive value $B_{match}$ can lead to overflow, causing a false non-match result. The first case is possible when the character set of the target is largely disjoint from that of the pattern, and the second case is considered trivial.

The 16-bit signals between each row illustrate the data dependencies between elements in adjacent rows, namely that computation of each element's value depends on the values of the element's North and Northwest neighbors in the row above. These would correspond to the values $d(i-1, j)$ and $d(i-1, j-1)$ from Equation 2.4, respectively. Because of these dependencies, the rows of the array must be computed in pipelined fashion, with each group of 4 target characters cascading down the array, one row per clock cycle. Note that each row actually has is own 32-bit register for holding target characters; Figure 4.3 only depicts that of the top row for clarity.

Also note the Northwest signal must wrap around at the end columns, as the row-wise pipelining means the Northwest neighbor of the lowest-order row element is the highest-order element in the row above.

Snapshot data are extracted from the array via the "shift_out" signal to the right of the array's highest-order column, with the signal's branch that extends up to the top row turning that column into a large rotate register. This arrangement lets the array return to its original state once the snapshot extraction is complete, and thus resume computation where it left off.

An unusual feature of this implementation of the Smith-Waterman algorithm is that it computes the DP matrix in columns rather than in the more customary diagonals. This design decision was made because of the requirement that the array accept 4 target characters per clock cycle, meaning each row must perform a calculation that is essentially Equation 2.4 nested to 4 levels deep. The column-wise organization, along with the row-wise pipelining, allows the logic necessary for this calculation to be contained completely within each row, simplifying the array's scalability to the extent that one need only add additional rows to accommodate larger patterns. Indeed, in the VHDL source code for the BioComp module, the length of the array is parametrized, with only the gate count of the RAD FPGA limiting the number of rows that may be instantiated. Patterns longer than the limit imposed by the size of the FPGA could be accommodated by vertically chaining multiple instances of the array together, i.e., by chaining several FPXs together, and partitioning the pattern string across the chain. This, however, is beyond the scope of this thesis and is left to future research.

### 4.2.1 Pipelined Systolic Array Row

Figure 4.4 shows a single, 4-element row of the Sequence Matcher's systolic array in greater detail, enumerating the different logic blocks that perform the alignment score computation. As with Figure 4.3, the 32-bit registers necessary for cascading the target characters from row to row are not shown for clarity. A prominent feature of Figure 4.4 is the three pipeline stages above the 16-bit registers that store each row element's value. This pipeline is actually in addition to the row-wise pipelining mentioned in the previous section, and is necessary because data dependencies between adjacent elements make the task of computing a row of 4 elements in parallel

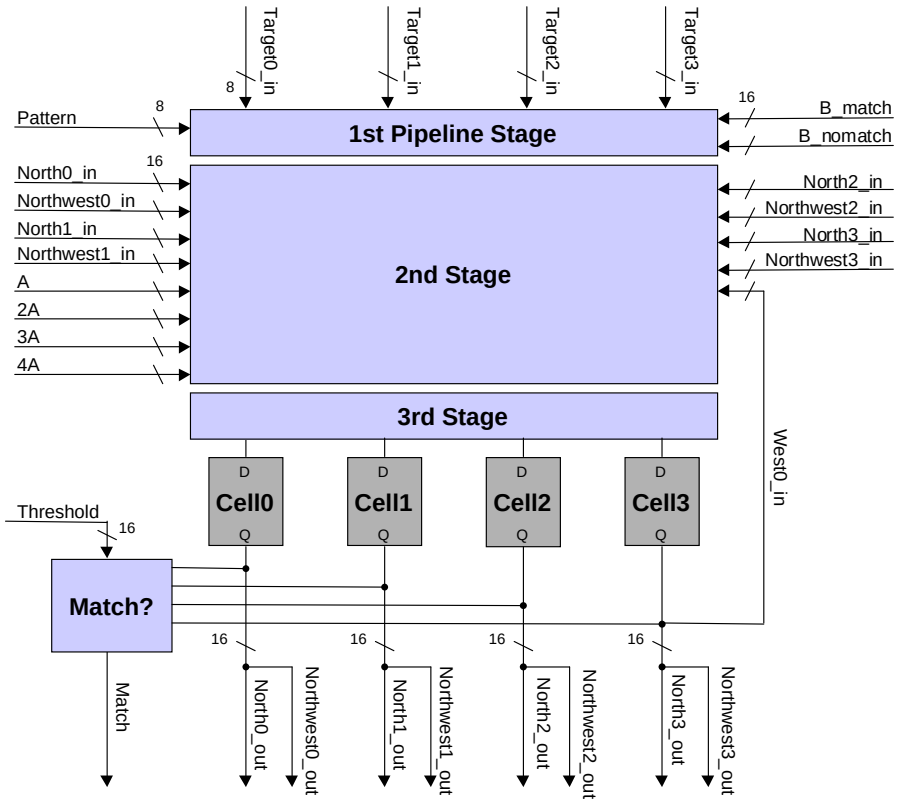


Figure 4.4: Block Diagram of Pipelined Array Row

non-trivial. Indeed, the logic required to do this computation in parallel is quite similar in spirit to that found in carry-lookahead adders, in that redundant logic must be added to avoid propagation delays that increase with the input width.

$$d(i,0) = max[d(i-1,0)+A; d(i,-1)+A; B(i,0)+d(i-1,-1)] \quad (4.1)$$

$$\begin{aligned} d(i,1) &= max[d(i-1,1)+A; d(i-1,0)+2A; d(i,-1)+2A; \\ &\quad B(i,0)+d(i-1,-1)+A; B(i,1)+d(i-1,0)] \end{aligned} \quad (4.2)$$

$$\begin{aligned} d(i,2) &= max[d(i-1,2)+A; d(i-1,1)+2A; d(i-1,0)+3A; \\ &\quad d(i,-1)+3A; B(i,0)+d(i-1,-1)+2A; \\ &\quad B(i-1,1)+d(i-1,0)+A; \\ &\quad B(i-1,2)+d(i-1,1)] \end{aligned} \quad (4.3)$$

$$\begin{aligned} d(i,3) &= max[d(i-1,3)+A; d(i-1,2)+2A; d(i-1,1)+2A; \\ &\quad d(i-1,0)+3A; d(i,-1)+4A; B(i,0)+d(i-1,-1)+3A; \end{aligned}$$

$$B(i-1,1)+d(i-1,0)+2A; B(i-1,2)+d(i-1,1)+A;$$
$$B(i-1,3)+d(i-1,2)] \quad (4.4)$$

Equations 4.1 through 4.4 above show the 4-level nested calculation based on Equation 2.4 for row $i$, with the additive term $A$ grouped and with the nested maximum operations flattened. Note that the term $d(i-1,-1)$, i.e., the Northwest input for the lowest-order row element, would actually be $d(i-1,3)$, with the wrap-around of that signal between rows shown in Figure 4.3. Also note the term $d(i,-1)$, i.e., the West input, would actually be $d(i,3)$, with the wrap-around shown for the signal "West0_in" in Figure 4.4. Of particular interest is the increasing number of inputs to the maximum function with the higher order row elements. For example, the calculation for $d(i,3)$ would require the maximum to be computed across 9 terms, which themselves are sums of yet more terms.

Implementing equations 4.1 through 4.4 as-is in hardware would produce a speed optimal circuit for computing the row elements' values. Operations with more than 2 inputs, e.g., the 9-input maximum, would be best implemented with a balanced binary tree of 2-input maximum operations on an FPGA, because of its architectural constraints. This would yield a maximum operation 4-stages deep for the term $d(i,3)$, plus an additional 2 stages for the additions, assuming no 3-input adders are available, and assuming the values $B(i,0)$, $B(i,1)$, $B(i,2)$, $B(i,3)$, $2A$, $3A$, and $4A$ are pre-computed. However, because of the space requirements of this speed optimal circuit, it was not implemented as-is in the systolic array. Rather, a compromise between speed and space optimization was sought, as explained below.

$$d(i,0) = max[\overbrace{max[d(i-1,0);d(i,-1)]}^{F}+A;\overbrace{B(i,0)+d(i-1,-1)}^{G}] \quad (4.5)$$
$$d(i,1) = max[\overbrace{max[F+2A;G+A;d(i-1)+A]}^{H};\overbrace{B(i,1)+d(i-1,0)}^{I}] \quad (4.6)$$
$$d(i,2) = max[\overbrace{max[H+A;I+A;d(i-1,2)+A]}^{K};\overbrace{B(i,2)+d(i-1,1)}^{J}] \quad (4.7)$$
$$d(i,3) = max[max[K+A;J+A;d(i-1,3)+A];B(i,3)+d(i-1,2)] \quad (4.8)$$

Equations 4.5 through 4.8 now show the expressions 4.1 through 4.4 re-organized to pick out common factors, and to reduce the final max operation to

2 terms. Of particular interest are the terms $F$, $H$, and $K$, which are all results of 2 or 3-input maximum operations. Because these terms occur more than once, and each time only with a constant $A$, $2A$, or $3A$ added, the result of each maximum operation (in terms of which of the inputs is selected as the maximum) would be the same across each equation. That is, if the result of the operation $F = max[d(i-1,0); d(i-1,-1)]$ is that the first of the two terms is greater, then the result of $F + A = max[d(i-1,0) + A; d(i-1,-1) + A]$ would be the same. Thus, when implementing the circuit that performs these maximum operations, one may save space by implementing all occurrences of $F$, $F + A$, $F + 2A$, etc. with a single comparator and then several multiplexers. Indeed, this is precisely the optimization implemented in the computational logic of each row in the systolic array, specifically for the terms $F$, $H$, and $K$. The tradeoff for this grouping of common factors is that more stages of logic are required than for the speed optimal circuit described above, increasing propagation delay. For the development of the BioComp module, both speed and space constraints were considered with equal weight, leading to a design with both speed and space optimizations blended.

A final note on the Sequence Matcher goes to the 3-stage pipelining of the the per-row calculation illustrated in Figure 4.4. As alluded to earlier, the terms $2A$, $3A$, and $4A$ are pre-computed, although not in the pipeline shown above since the parameter $A$ is loaded into the Sequence Matcher several cycles before the systolic array would be enabled by the Control Module. The first stage of the pipeline accepts the 4 incoming target characters $t_0$ through $t_3$ and compares them with the pattern character $p_i$ to calculate $B(i,0)$, $B(i,1)$, $B(i,2)$, and $B(i,3)$, which are then stored in registers between the first and second pipeline stages. The second stage then performs all calculation to reduce the remaining input values to 8 16-bit terms, namely the inputs to the 4 maximum operations shown in Equations 4.1 through 4.4, and stores the terms in additional pipeline registers. The third stage performs these remaining 4 maximum operations to obtain the final values for the row elements, and stores these values in the registers labeled "Cell0" through "Cell3" in Figure 4.4.

The critical path which dictates the maximum clock frequency of the BioComp Module is in the second pipeline stage shown in Figure 4.4. This path follows the logic in that stage which computes the inputs to the 4 maximum operations shown in Equations 4.1 through 4.4. The differential between the propagation delay along this path and that along any other path in the BioComp Module is great enough that small modifications made elsewhere in the module have no effect on the overall

critical path. For example, reducing the width of the target and pattern inputs of the systolic array to 2 bits per character, i.e., for an optimized encoding of DNA bases, would indeed save gates in the RAD. However, since the second pipeline stage only involves 16-bit arithmetic, regardless of the width of the target and pattern inputs, it would not improve the critical path.

## 4.3 Control and Match Reporting

### 4.3.1 BioComp Control Module

The primary task of the BioComp Control Module is to accept and process ATM control cells sent to the FPX by the NCHARGE software running on the user's host workstation. This entails extracting runtime parameters from those cells' payload, replacing the parameters with control acknowledgements, and controlling the Sequence Matcher and Snapshot Manager based on commands embedded in the control cells. More detail on the format and handling of control cells, as well as methods for generating such cells and sending them to the FPX, are available in extant literature on the NCHARGE control software in [4], [24], [25], and [36]. Table 4.1 gives a synopsis of the control functions the Control Module provides to NCHARGE.

Because of the scalability of of the Sequence Matcher's systolic array, the Pattern which the user submits via NCHARGE control cells may be of varying length. Since the NCHARGE control cell format only allows fixed-width fields, the Pattern is submitted to the Control Module in 32-bit segments, with an additional field specifying the segments' order. The Pattern length specification is necessary when the user submits a Pattern that is shorter than the length of the systolic array. That is, for short Patterns, the Sequence Matcher should only monitor match results from a subset of its array's rows to prevent reporting possible false matches.

The Snapshot Period specifies the number of target characters that pass before the Control Module tells the Snapshot Manager to output a snapshot of the systolic array. Since the Sequence Matcher must halt computation for a snapshot to be extracted from it, thereby adding overhead and decreasing the BioComp module's target data throughput, ideal values for the Snapshot Period would be quite large, e.g., of the order $10^6$ or greater.

Table 4.1: NCHARGE Commands for BioComp Control Module

| |
|---|
| Set or query following runtime parameters:<br>• Match Threshold<br>• Gap Penalty $A$<br>• Single-character match weighting $B_{match}$<br>• Single-character non-match weighting $B_{nomatch}$<br>• Length of submitted Pattern<br>• 32-bit segment of Pattern and segment number<br>• ATM VCI on incoming target data cells<br>• ATM Headers to place on outgoing cells with snapshot data (specifies destination in switch fabric)<br>• Snapshot Period |
| Tell the Control Module to begin listening for incoming target data cells |
| Tell the Control Module to ignore incoming target data cells, i.e., pass them through |
| Reset the systolic array and all runtime parameters |

Figure 4.5 shows an abbreviated state diagram for the Control Module. Two circuits make up the the diagram, one for processing incoming control cells and another for routing target data from incoming data cells to the Sequence Matcher. Each state in the data cell circuit monitors a count of target characters that have passed, comparing the count to the Snapshot Period. Once that period has elapsed, the Control Module tells the Snapshot Manager to begin extracting a snapshot. As the incoming data can not stop immediately because of limitations of the data-path between the RAD and NID, a waiting period must elapse before it is safe to disable the systolic array. This is explained in more detail in the following section.

Figure 4.6 shows the format of incoming target data cells. In the interest of optimizing data throughput, the target cells have as bare a format as the ATM specification will allow, i.e., two 32-bit headers and twelve 32-bit payload words. This

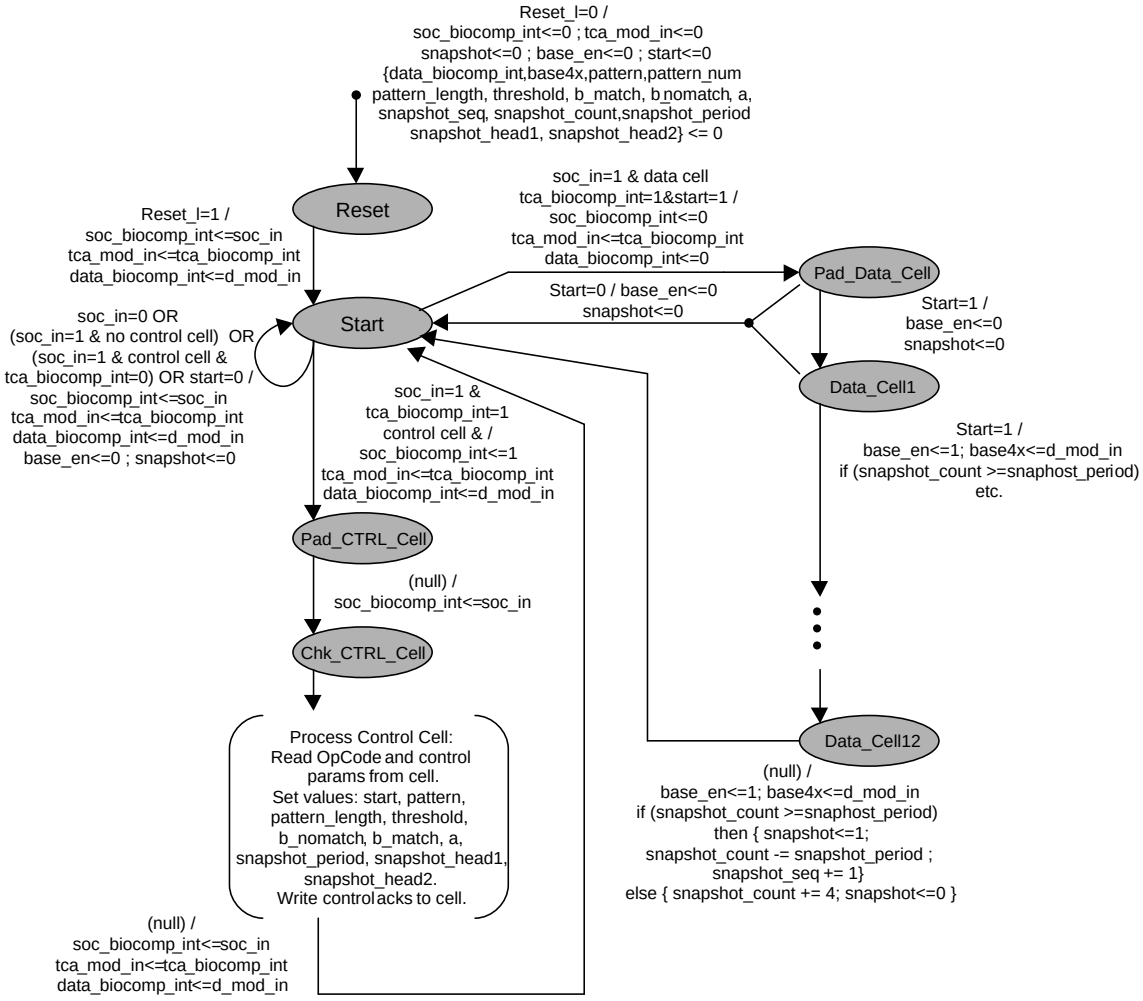


Figure 4.5: State Diagram for Control Module

still leaves a 14 percent overhead, although that is imposed solely by the switch fabric of the WUGS itself.

### 4.3.2 BioComp Snapshot Manager

The Snapshot Manager handles the extraction of snapshot data and match results from the Sequence Matcher, and packages those data into ATM cells to send back to the user's host workstation. Figure 4.7 shows the state diagram for the Snapshot Manager. Much of the complexity of the state diagram results from the variable length of the systolic array, and that the snapshot data must be formatted into fixed-length ATM cells for transmission back to the host workstation. If a single

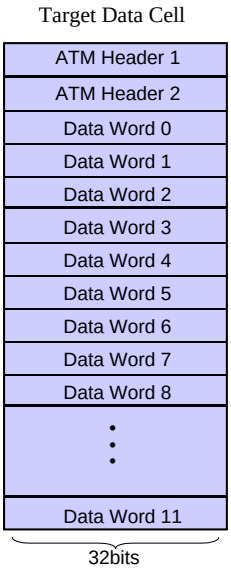


Figure 4.6: BioComp Target Data ATM Cell

snapshot doesn't fit in a single ATM cell, then the Snapshot Manager must output a group of consecutive ATM cells for each snapshot, with those cells' format depicted in Figure 4.8.

A non-trivial detail of the format of the snapshot ATM cells shown in Figure 4.8 is that the row element values of the systolic array are given in descending order, which is an artifact of the arrangement of the "shift_out" signal of the Sequence Matcher shown in Figure 4.3. That is, the first snapshot cell will contain the row element values starting with the highest-order row, i.e., Row $m-1$ for a systolic array instantiated out to $m$ rows, then Row $m-1$, and so on until Row 0. Besides the 16-bit element values extracted as a snapshot from the systolic array, the Snapshot Manager also outputs the following values in the first ATM cell of snapshot data:

- Snapshot sequence number, indicating which snapshot period this is
- Delta value
- Match indicator

The Delta value measures the number of target characters that still enter the Sequence Matcher after the Control Module has instructed the Snapshot Manager to make a snapshot. That is, once the Control Module observes that a snapshot period has elapsed and activates the Snapshot Manager, the Snapshot Manager halts

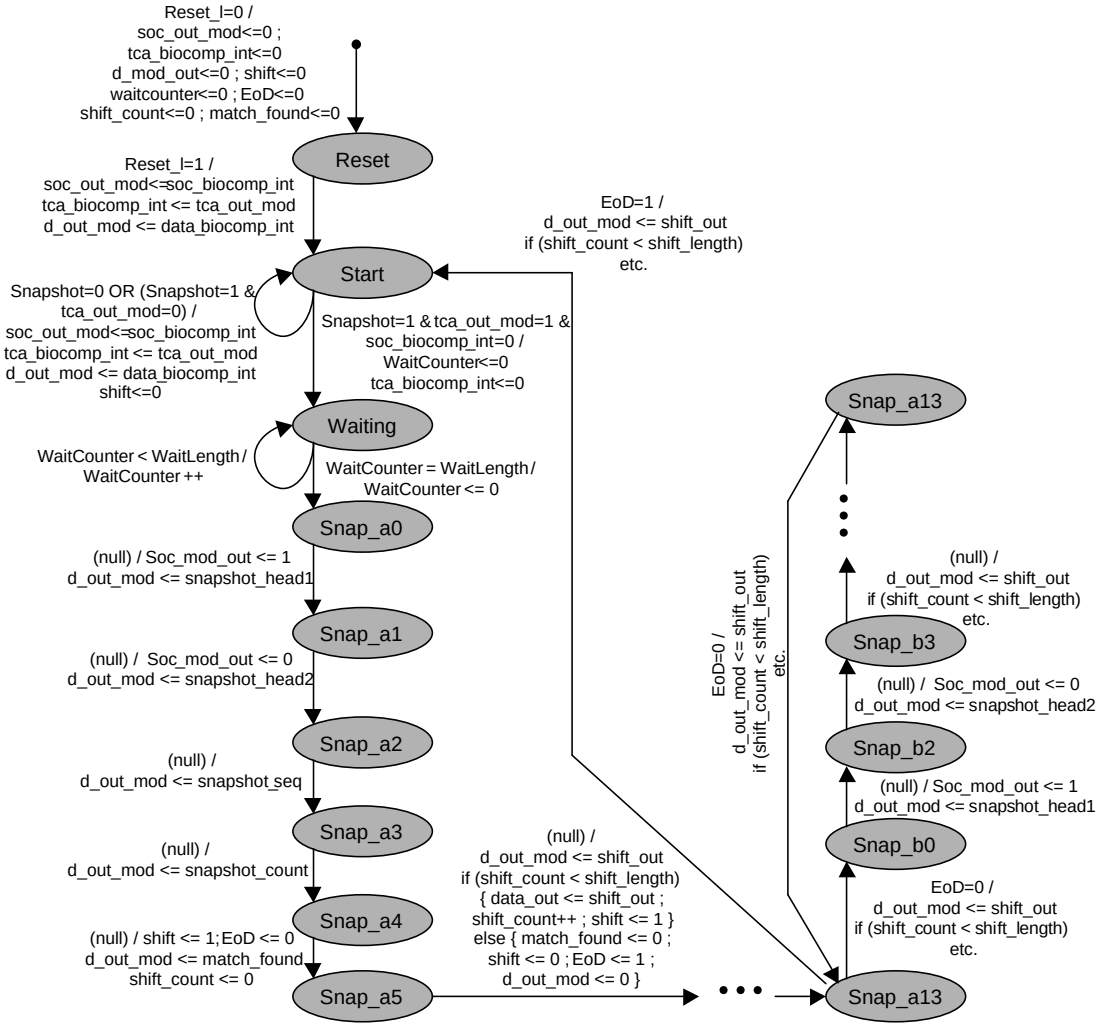


Figure 4.7: State Diagram for Snapshot Manager

incoming target data cells by asserting the back-pressure signal "tca_biocomp_int," shown in Figure 4.2, which then propagates through the Control Module back to the NID. Since ATM cells moving between the RAD and NID must be transmitted whole, the effect of the back-pressure signal would not be seen until the current incoming target data cell, and potentially another, had completely passed. Thus the Delta value, which is simply the value of the Control Module's target character counter once incoming data has stopped, would be needed by the user to correctly reconstruct the DP matrix from snapshot data. That is, the user would calculate the column number, $j$, of the snapshot data as shown in Equation 4.9, which may then be

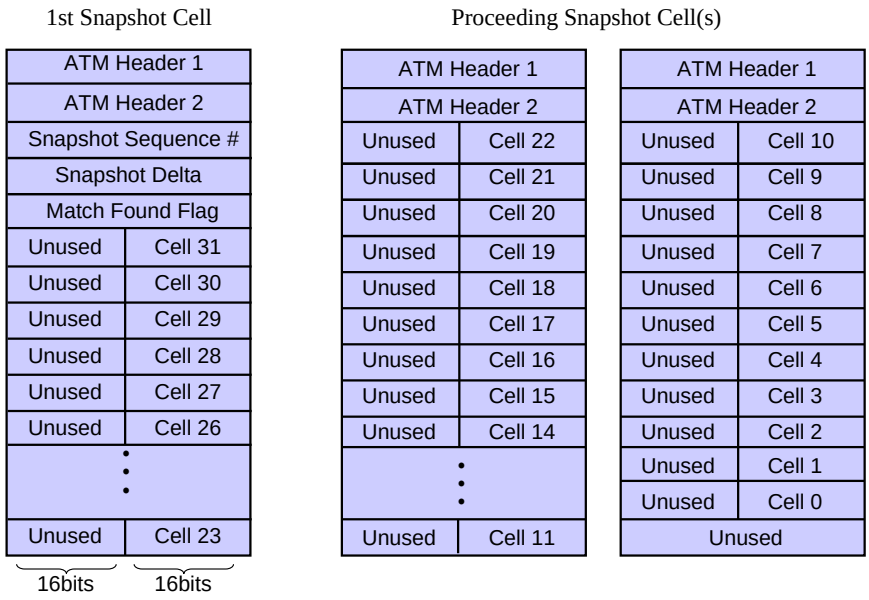


Figure 4.8: BioComp Snapshot ATM Cell Format, for a 32-row Array

substituted for the value $j$ shown in Equation 2.4 for reconstructing the DP matrix.

$$j = Snapshot\ Sequence\ Number \times SnapshotPeriod + Delta \tag{4.9}$$

To minimize the complexity of the Snapshot Manager's state machine, it simply the waits the maximum number of cycles that may elapse before the the back-pressure signal "tca_biocomp_int" would take effect, 17 cycles. In addition, because the Snapshot Manager can not begin extracting from the Sequence Matcher until all pipeline stages of the of the systolic array are idle, an additional wait period of $m + 3$ cycles is required, where $m$ is the number of array rows. This makes the wait time after a snapshot period has elapsed $m + 20$ cycles. Furthermore, the time required to then extract the snapshot data and transmit them back to the NID would be $\lceil \frac{m+3}{12} \rceil \times 14 + 6$ cycles, where the additional 6 cycles is the time required for the de-asserted back-pressure signal "d_biocomp_int" to reach the NID and take effect. This makes the total overhead introduced by each snapshot $\lceil \frac{m+3}{12} \rceil \times 14 + m + 26$ clock cycles, hence the suggestion above to make the Snapshot Period parameter quite large. A larger Snapshot Period value does, however, increase the off-line computation required to reconstruct the DP matrix between consecutive snapshots for alignment extraction. Nevertheless, if the hit rate of the Pattern being sought is expected to be low, this

off-line computation would be negligible compared to the prospect of constructing the DP matrix for the entire Target string off-line.

### 4.3.3 Web-based Interface to BioComp

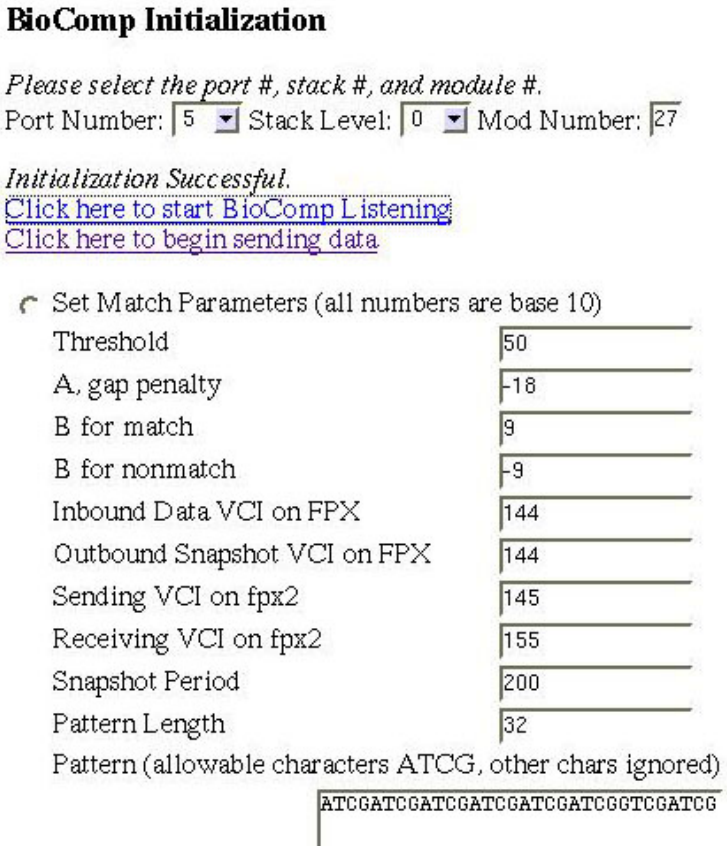


Figure 4.9: BioComp Web Interface, Initialization

Figures 4.9 and 4.10 are screen-shots made of a Web-based interface designed to run on top of the NCHARGE FPX control software, to provide live demonstrations of BioComp Module's operation. HTML forms allow the user to submit runtime parameters to the BioComp module (Figure 4.9) from a Web browser, and to submit target data and then view the outputted snapshots (Figure 4.10).[2] This interface makes extensive use of the existing NCHARGE software (which indeed already has a Web interface of its own), and the motivation for its development was to dramatically tighten the development cycle for the BioComp module.

[2]The software which parses raw snapshot data into the tabular, human-readable form shown in Figure 4.10 was written by Brian Bruggeman in the context of the Spring 2002 course EE563.

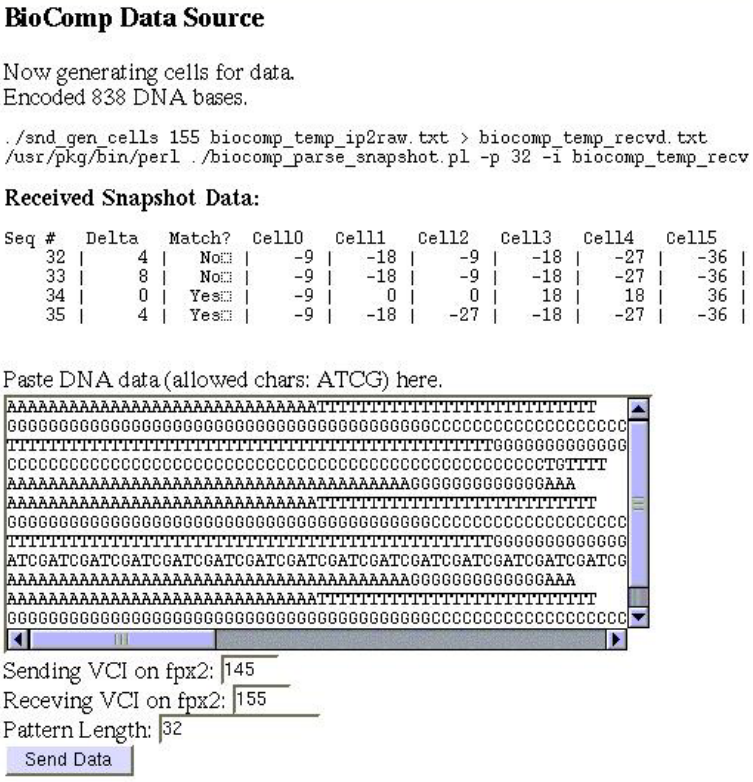


Figure 4.10: BioComp Web Interface, Search Results

# Chapter 5

# Performance Comparisons

## 5.1 Overview

A crucial part of any presentation of high-speed search system is a direct, side-by-side comparison with the traditional search methods this search system is meant to replace. Thus, this chapter presents performance data collected for traditional CPU-based search applications, and points out specifically where the the performance of these applications suffers from constraints not present in the search system discussed in this thesis. As mentioned in the thesis outline in Chapter 1, the two CPU-based search applications selected for these performance test are as follows.

- GNU string-matching tool "grep," version 2.4.2, freely available at http://gnu.org
- The author's C++ implementation of the Smith-Waterman algorithm, meant to imitate input and output of the Sequence Matcher core of the BioComp FPX Module (Chapter 4)

The relevant configuration of the host workstation on which the tests were performed is as follows:

- SMP workstation with two Intel Pentium-III 933 MHz processors [1]
- 512 MB RDRAM
- Redhat Linux version 7.2 operating system

[1] "grep" and the Smith-Waterman implementation were not compiled for multi-processing, thus their execution time on this machine would be comparable to that on a uniprocessor machine.

- Promise TX-2 ATAPI/IDE controller
- Seagate ST320414A 20 GB ATAPI/IDE hard drive, formatted with an ext2 file-system
- IDE bus also fitted with an Innotec Design, Inc. ID620a bus analyzer

The two major hurdles to collecting performance data on this workstation were the precision of execution time measurements derived from the workstation's real-time clock, and the lack of synchronization of such execution time measurements with the IDE bus activity measurements gleaned by the bus analyzer. Both hurdles were overcome with relatively straightforward work-arounds. Specifically, the precision of execution time measurements, which would have been limited to milliseconds if derived from the workstation's real-time clock, were instead derived from the RDTSC cycle counter proprietary to the Pentium-series processors [18]. This enabled timing measurements with sub-microsecond precision. The synchronization between these measurements and the bus analyzer data (made all the more important by the dramatically increased precision) was then achieved by framing each individual experiment run with a 2-second delay. This allowed the portions of the bus analyzer data relevant to each experiment run to be easily located, and then grouped with that run's execution time measurement.

The experiment runs themselves all used sets of artificially generated search data files of increasing size. The use of separate files, as opposed to consecutive executions with the same input file, was necessary to prevent caching operations in the operating system and the hard drive itself from affecting the performance measurements. That is, the data files in each experiment run were read after any cached copies of them stored on the hard drive itself or in RAM had been cleared, and no data file was read more than once during a run. Thus, the data files came streaming directly off the hard drive's platters, preventing read cache, pre-fetching, or similar operations from affecting the measured execution time. The increasing size of the input files also allowed the execution time measurements to be plotted against file sizes, with the derivative of the resulting trend line yielding a sec/byte search throughput value. This throughput value could then be compared directly against those of other CPU-based applications, or against that of the BioComp FPX Module to obtain a quantitative performance gain. An ideal search throughput value, where processing time of the search application did not affect the throughput, but where the inherent limitations of the hard drive-CPU data-path shown in Figure 1.1 did

have an effect, was obtained with the Linux tool "hdparm." This ideal throughput value was shown by the bus analyzer to be 40.5 MB/s, which is ideal because there was actually no searching done. That is, "hdparm" simply read an arbitrary portion of the hard drive's contents to test its throughput.

The input files used in all performance tests were ASCII text files containing 128-byte strings of characters delimited by a new-line character. The portions of the input files that were not matches planted by the author were populated with a single character. The files ranged in size from approximately 10 KB to 200 MB, although performance tests involving large numbers of planted matches started with larger input files to accommodate one planted match per line. The input files were generated and written to an ext2 file-system stored on the hard drive being tested. During each performance test, the only activity between the IDE controller and the hard drive was that relevant to mounting the file-system and reading the input files. Nothing else was written to or read from the hard drive during the tests. Furthermore, the input files were generated and written to the hard drive in the same order in which they would be read during a performance test. This improved the probability that the input files for a particular test would be written to a roughly contiguous area on the hard drive's platters, and thus minimize variation in the hard drive's access time.

## 5.2 Grep Performance Tests

The UNIX tool "grep" is a popular string-matching application whose character-matching kernel is implemented as a state machine to optimize its execution time, i.e. by streamlining operations such as register copying, single-character comparisons, and conditional branching. As a result, "grep" can exhibit $O(n)$ execution time, where $n$ is the size of the input data to be searched. Nevertheless, "grep" is still an application designed for uniprocessor machines, meaning it will introduce overhead in processing time. Since "grep" only reports which lines of the input data contain the specified search query, "grep" can skip portions of the input data that don't need to be processed. For example, given a query string $m$ characters long, if the $mth$ to last character in a line of input does not match the first character of the query, "grep" will skip ahead to the next line. The author, however, sought to prevent such optimization, as it takes advantage of the capability for random access to the input data, and would invalidate any performance comparisons of "grep" with stream-based hardware search devices like the BioComp Module.

Although "grep" can perform inexact string-matching, e.g., with wildcards and regular expressions, the performance tests conducted for this thesis only covered the less complex operation of exact string-matching. This means the search throughput results shown in this section represent an upper limit on "grep's" performance, and that "grep's" optimized state machine should only need to perform a minimal number of signal-character comparisons, i.e., $O(n)$ for an input file $n$ bytes large.

### 5.2.1 Execution Time vs. Number of Matches

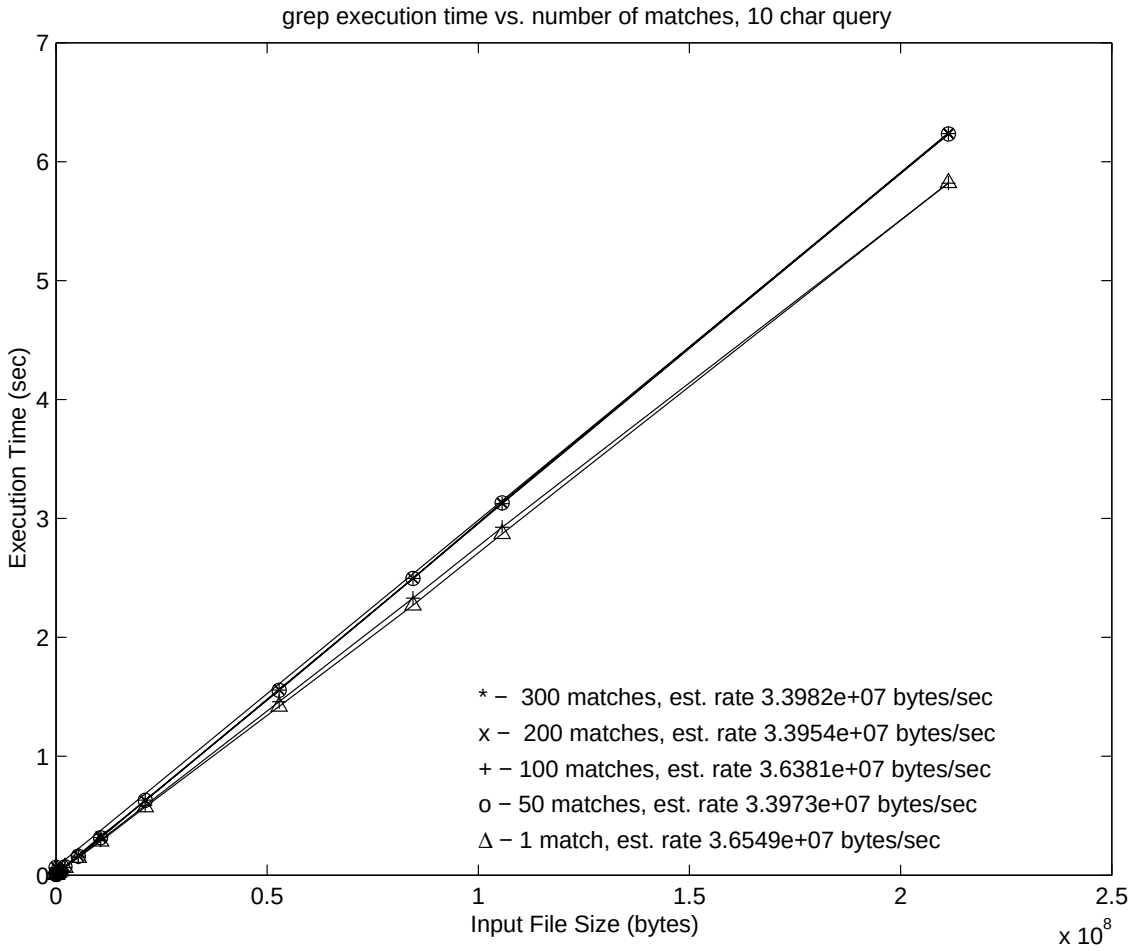


Figure 5.1: "grep" execution times vs. number matches

Figure 5.1 shows "grep's" execution time on an array of input files plotted against the files' size. The multiple lines represent performance tests with different numbers of matches to the search query planted in each file. To ensure that grep's state machine only encountered matches or partial matches at the points where the

author planted matches, the remaining portions of the input files were populated with a single character not equal to the first character of the search query. Since grep only seeks the first match on each line of the input file, the matches were planted one to a line, positioned at the end of the line, thus ensuring that "grep" processed every byte of the input files. For each performance test an approximation of "grep's" byte/sec throughput was calculated using linear regression. Those throughput values are listed in Figure 5.1.

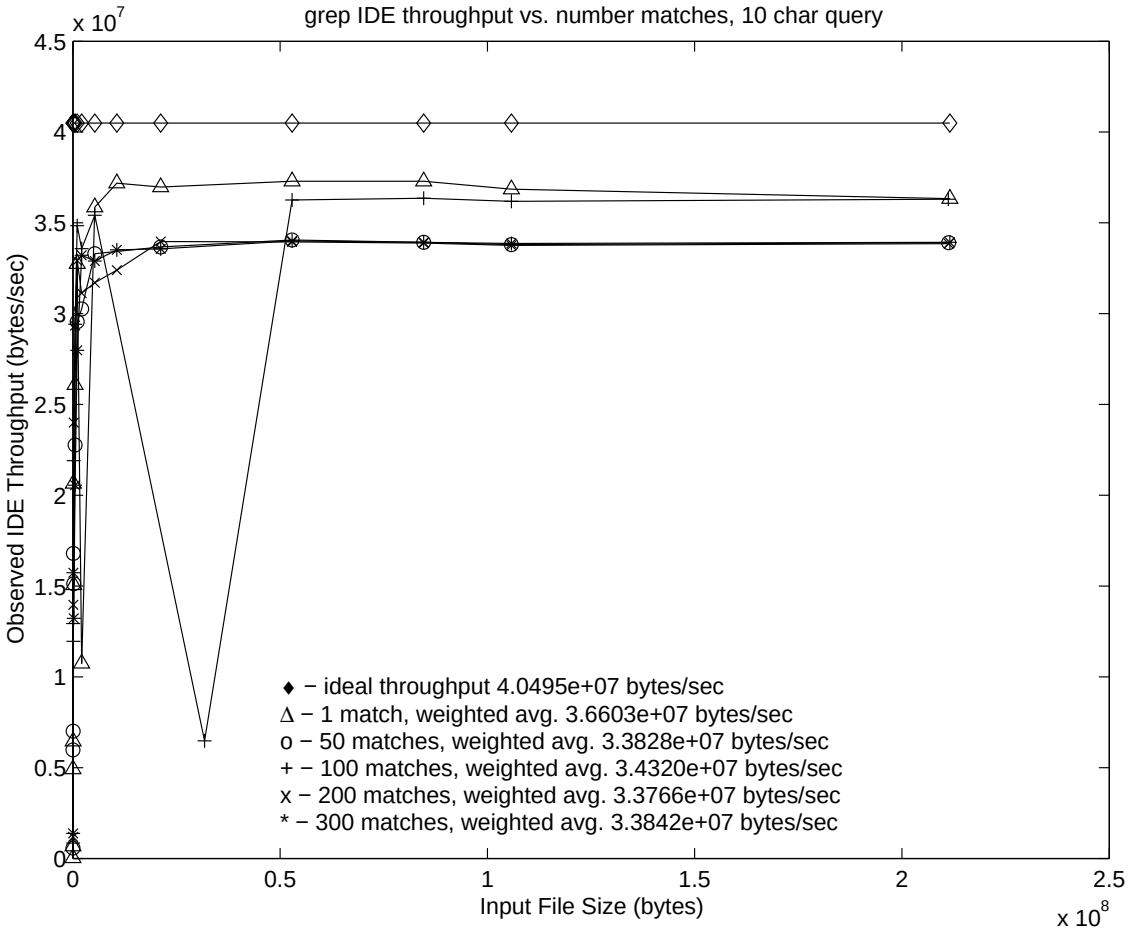


Figure 5.2: "grep" IDE throughput vs. number matches

Figure 5.2[2] shows IDE bus throughput measurements made by the bus analyzer during the performance tests shown in Figure 5.1. The throughput for each data point was calculated by dividing the number bytes that traversed the IDE bus in each "grep" execution by the total time duration of the IDE activity observed during that execution. Figure 5.2 also lists the weighted average throughput values for each performance test, where the input files' sizes were used for weighting. These throughput values correspond roughly with the estimated throughput values listed in Figure 5.1. The correspondence, however, is not exact due to the lack of perfect synchronization between the IDE bus events monitored by the bus analyzer and the "grep" execution time measurements. Furthermore, block granularity of the input data, i.e., at the file-system's level and at the physical level on the hard drive's platters, ensured that the amount of input data that traversed the IDE bus during each "grep" execution was usually greater than the size of the input file. This artifact has a minimal effect on the average throughput values because of the weightings used, but it is still present.

The variation in search throughput values listed in Figures 5.1 and 5.2 does not follow the variation in the number of matches planted in the input files. Furthermore, since this variation between the individual performance tests is consistent in both figures, i.e., the performance tests fall into the same groupings in each figure, the variation is mostly likely that of the hard drive's access time.

### 5.2.2 Execution Time vs. Query Length

Figure 5.3 shows "grep's" execution time on an array of input files plotted against the files' size. The multiple lines represent performance tests with query strings of different lengths. The query strings were composed of non-repeating characters, with the first character being unequal to the single character that populated the rest of the input files. The estimated throughput values were computed in the same fashion as in Figure 5.1, and the planted matches were positioned in the input files in the same fashion as for the performances tests shown in Figures 5.1 and 5.2. The "false matches" mentioned in the title of Figure 5.3 indicate that the planted

[2]The sharp drop in throughput seen in the curve for the performance test with 100 matches planted in each input file is a result of the fact the author did not have complete control of the IDE bus. That is, bus events unrelated to the performance tests could occur during the tests themselves, obscuring the bus throughput data the author sought to extract. That is what happened with this particular data point, but since the input file affected is relatively small (30 MB), the impact of this artifact is small.

matches actually do not match the query string exactly. Rather, they differ at the last character. This arrangement causes "grep" to not skip any bytes in the input files through the optimization mentioned above, while at the same time preventing any match-reporting subroutines which "grep" would otherwise run from affecting the execution time measurements.

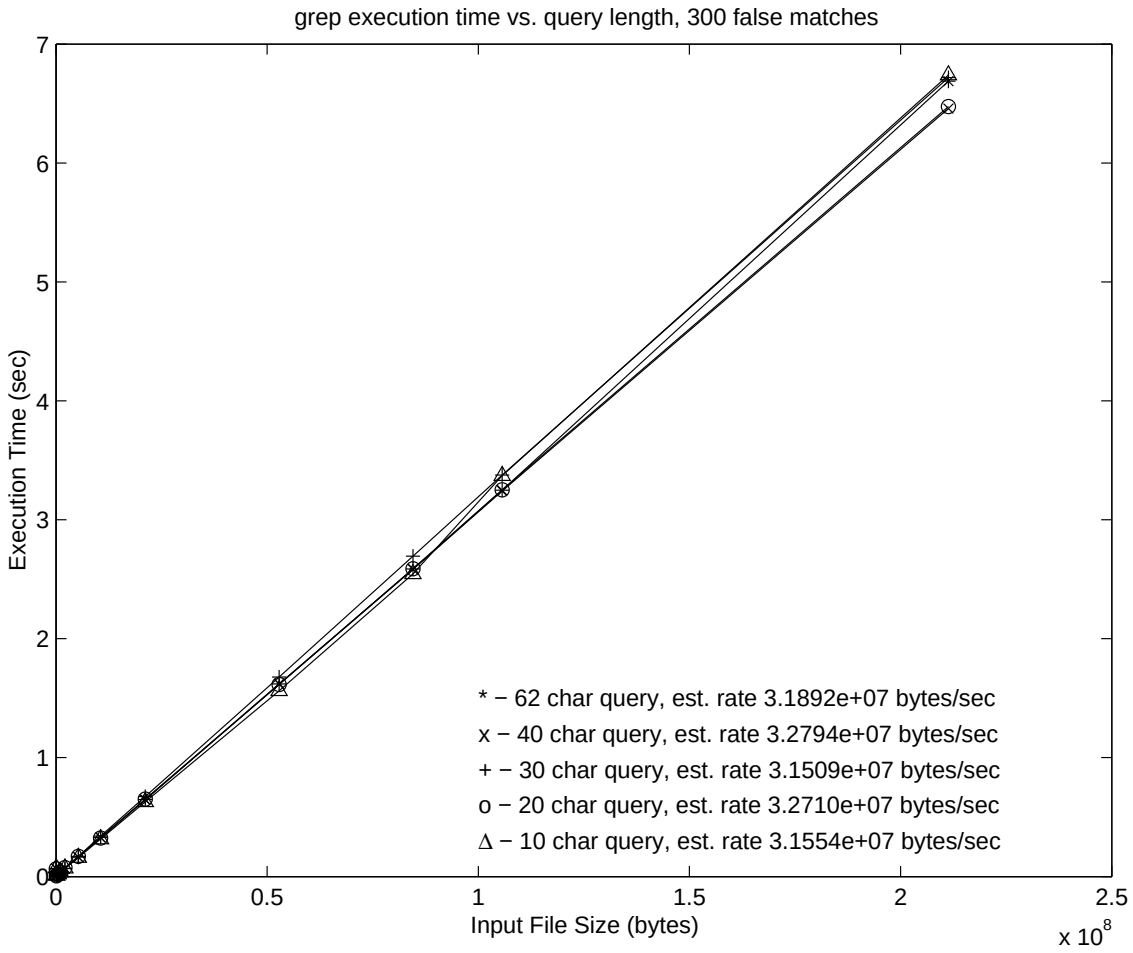


Figure 5.3: "grep" execution times vs. query length

Figure 5.4 shows the IDE throughput measured by the bus analyzer during the performance tests shown in Figure 5.3. The weighted average throughput values were computed in the same fashion as in Figure 5.2.

As with Figures 5.1 and 5.2, the variation in execution time and IDE bus throughput evident between individual performance test in Figures 5.3 and 5.4 is not consistent with the varying query string length. This implies that query string length, like the number of matches, does not have a noticeable effect on "grep's"

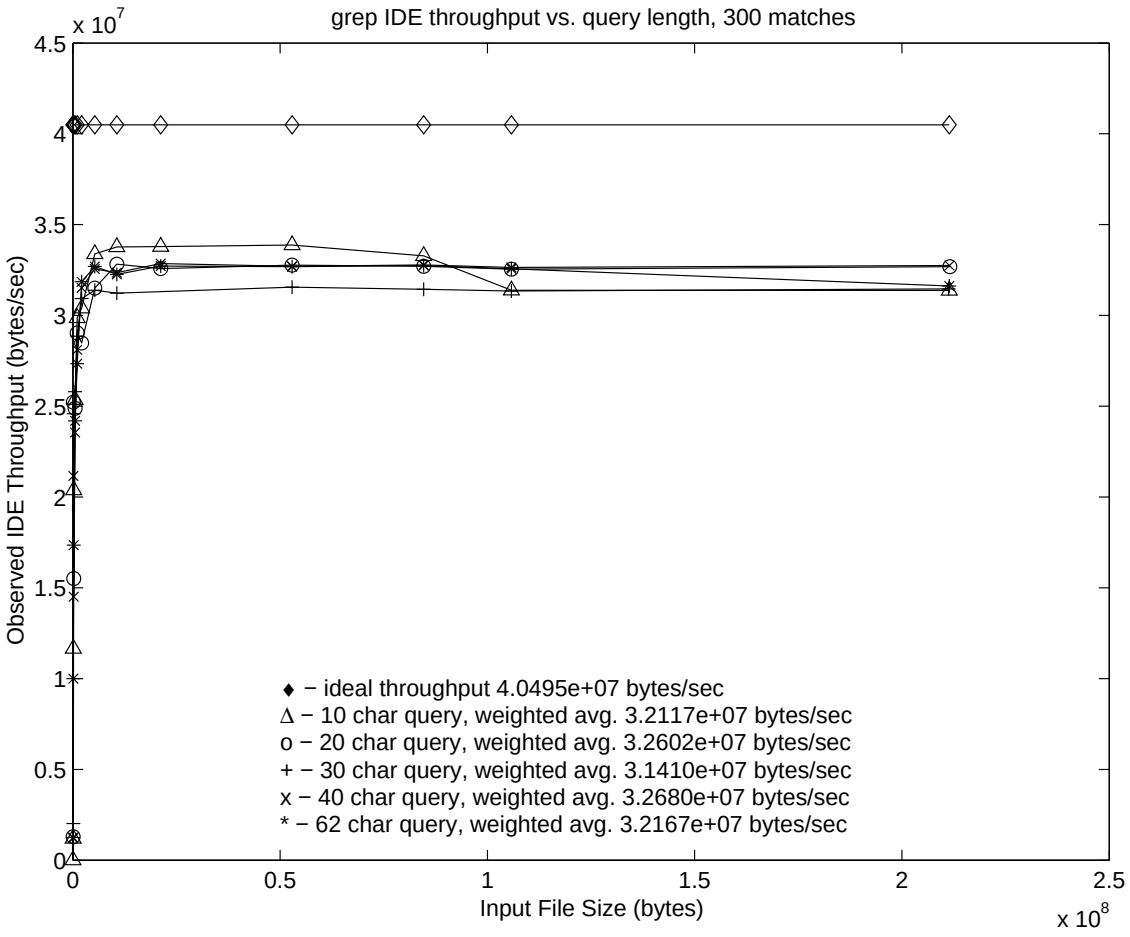


Figure 5.4: "grep" IDE throughput vs. query length

execution time. This is to be expected for an optimized string-matching application which minimizes the number of single-character comparisons to $O(n)$ for an input file of $O(n)$ characters. Furthermore, the bifurcation of data points is consistent in both Figures 5.3 and 5.4, once again suggesting this variation in execution time and IDE throughput results from the varying access time of the hard drive.

What is also evident in Figures 5.2 and 5.4 is that even with an optimized number of single-character comparisons, "grep" still can not reach the ideal IDE bus throughput observed with the "hdparm" utility. This implies "grep's" search throughput is indeed CPU-limited, and likely representative of the upper limit on exact string-matching throughput possible on a traditional workstation like that illustrated in Figure 1.1.

## 5.3 Smith-Waterman Performance Tests

This section presents the results of performance tests run on the author's C++ implementation of the Smith-Waterman algorithm, as a function of pattern length (and, thus, systolic array length). The Smith-Waterman implementation, dubbed "biocomp," is trivial, in that the algorithm has been implemented as-is, i.e. with $O(m \times n)$ execution time for a target of size $n$ and a pattern of size $m$, and it mimics the input and output characteristics of the BioComp FPX Module. Nevertheless, these performance results illustrate quantitatively the performance gain the BioComp Module enjoys through parallelism. "biocomp" was implemented such that the composition of the target and pattern string (and the resulting number of matches) would not alter any conditional branching during execution, thus making the specified pattern string and composition of the input file irrelevant to the performance results. For consistency, however, the Snapshot Period was set to 1 MB for all tests, making the overhead processing time required to parse each snapshot negligible compared to the total execution time.

Figure 5.5 shows "biocomp's" execution time on an array of input files plotted against the files' size. The multiple lines represent performance tests with patterns strings of different lengths. The estimated throughput values were computed in the same fashion as in Figure 5.1.

Figure 5.6 shows the IDE throughput measured by the bus analyzer during the performance tests shown in Figure 5.5. The weighted average throughput values were computed in the same fashion as in Figure 5.2.

What is strikingly evident in Figures 5.5 and 5.6 is the dramatic dependency of "biocomp's" execution time on the pattern string length. This is hardly a surprise, as the $O(m \times n)$ execution time makes the two values inversely related.

Computing a regression line over the estimated search throughput values from Figure 5.5 versus the inverse of the pattern string lengths yields a slope of $1.716 \times 10^7$ [(target bytes/sec) $\times$ (pattern bytes)], with the intercept $3.572 \times 10^5$ (target bytes/sec). This regression line is plotted in Figure 5.7. By this metric, the software implementation "biocomp" would exhibit a search throughput of $8.088 \times 10^5$ target bytes/sec for a 38-character pattern string (i.e., the maximum pattern length for the BioComp FPX Module). This gives the BioComp Module, which has a constant search throughput of 100 MB/sec, a performance gain of 124 over its software implementation. (This gain is neglecting the 40.5 MB/sec limitation imposed by the hard

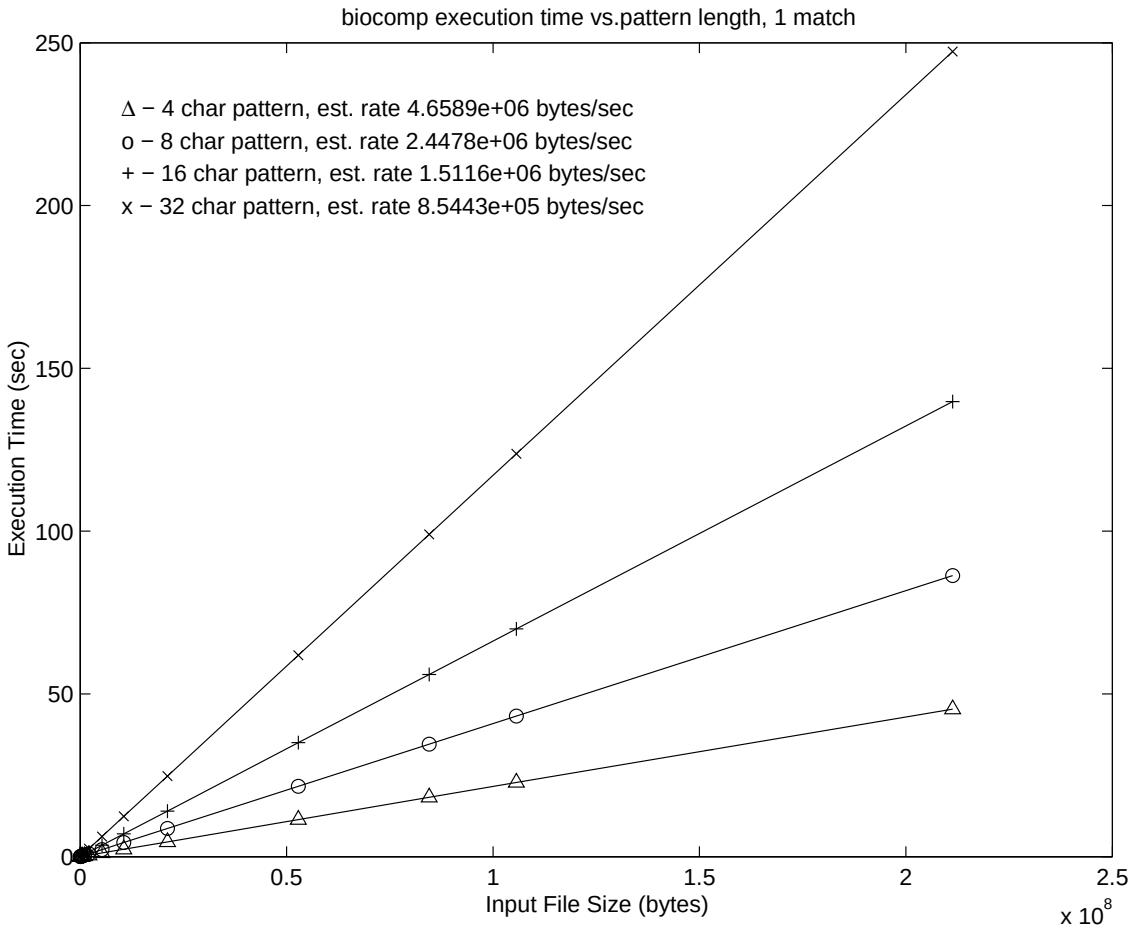


Figure 5.5: Smith-Waterman execution times vs. pattern length

drive itself, mentioned in Chapter 3, which would reduce the gain to 50.1.) Indeed, given the inverse relationship of "biocomp's" execution time to the pattern string length, this gain would only increase with larger pattern strings.

## 5.4 Conclusion

This section has presented performance results for two string-matching applications intended for traditional, uniprocessor workstations. The two applications were the string-matching tool "grep" and the author's direct implementation of the Smith-Waterman algorithm, "biocomp." Both applications, especially "biocomp," exhibited a CPU-dependent bottleneck on search throughput that limited their search throughput. "grep," for example, reached approximately 90 percent (Figure 5.1) of the ideal

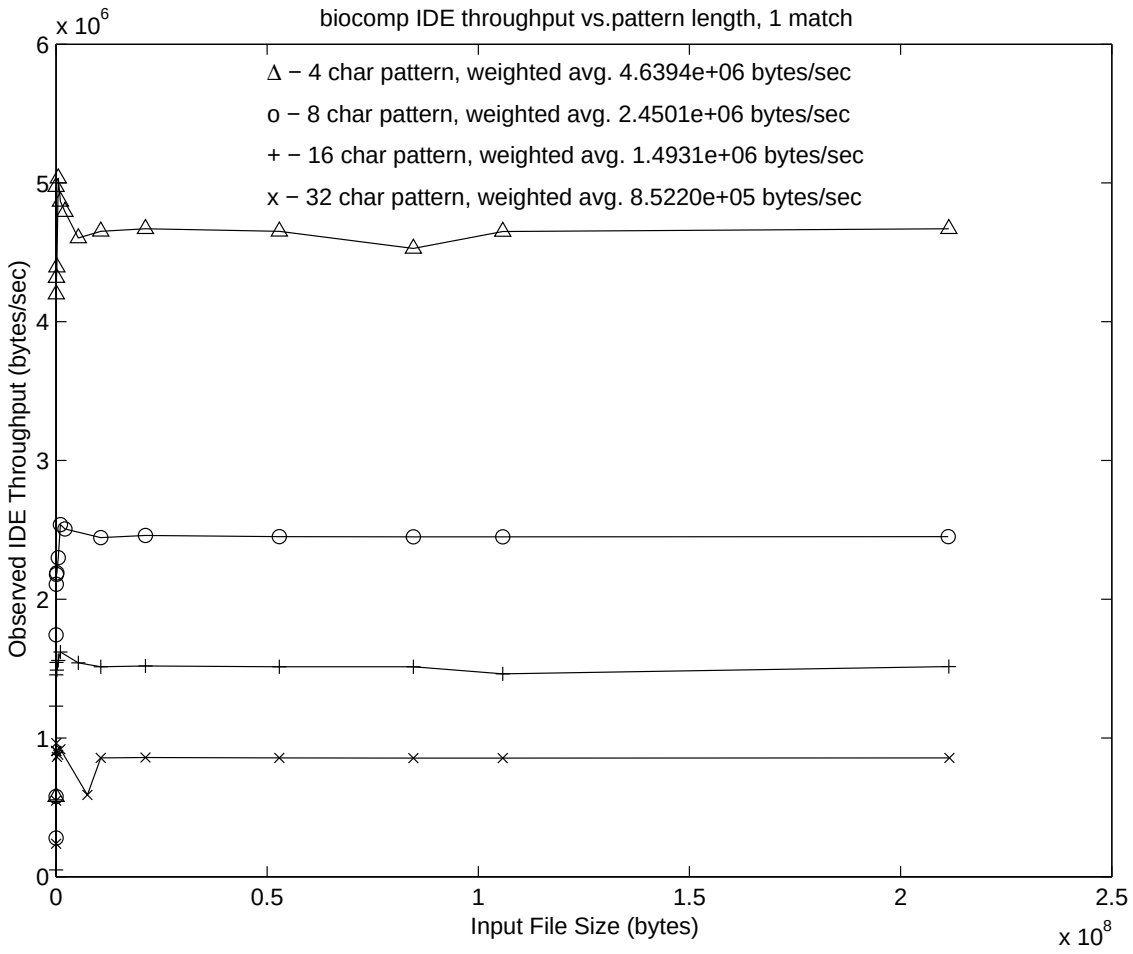


Figure 5.6: Smith-Waterman IDE bus throughput vs. pattern length

throughput obtained with "hdparm." "biocomp," on the other hand, ended up being out-performed by its hardware twin by up to a factor of 124.

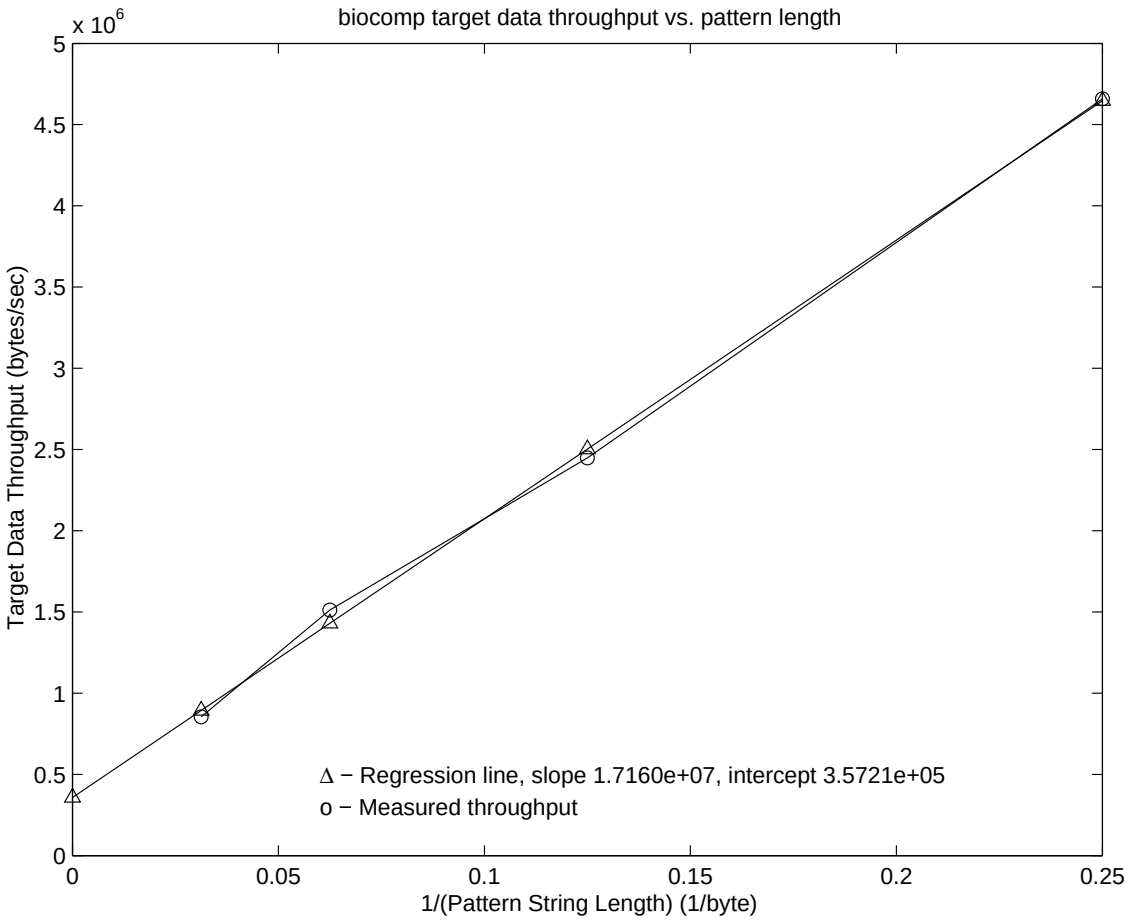


Figure 5.7: Smith-Waterman target data throughput vs. pattern length

# Chapter 6

# Conclusion and Future Work

## 6.1 Overview

This thesis has presented the design and performance results of an FPGA-based, high-speed search system that implements the Smith-Waterman local sequence alignment algorithm [35]. The motivation for designing and building this system was to demonstrate the magnitude of search performance gain possible by implementing simple, stream-searching algorithms on an FPGA, and then by placing that FPGA at a level much closer than the CPU to the hard drive storing the target data. This system was implemented on the FPX [26], an FPGA-based component of the WUGS [39], because the FPX and WUGS provided a platform where large FPGAs could access a shared data-path over a gigabit/sec interconnection network.

## 6.2 Contributions

The top-level goal of the work presented in this thesis was to fabricate a functioning FPGA-based search system out of the FPX/WUGS infrastructure available to the author. This goal was decomposed into the following sub-tasks:

- Inject data from a hard drive into the WUGS switch fabric, for forwarding onto an FPX
- Process the hard drive's data in a fashion that exhibits parallelism impossible with traditional, CPU-based search applications
- Develop and consolidate paths for control and search result reporting

- Demonstrate search performance gain of this search system

The first two tasks were realized as FPX modules, each programmed onto its own FPX. The third was realized by building atop the extant control software for the FPX and WUGS, to provide a streamlined interface to control functions specific to the author's implementation. The fourth goal was realized by measuring execution times of traditional CPU-based search applications, both with timing functions internal to the host workstation running the search applications, and with an external device that monitors IDE bus traffic. These execution times, measured at two points in the data-path illustrated in Figure 1.1, were then folded together to quantitatively show where limitations in such a traditional hard drive-CPU data-path impede search throughput.

The specific contributions made during the work of this thesis are listed below:

- IDE Bus Snooper FPX Module
  - Deciphered the ATAPI/IDE Protocol [27] to design a state machine that recognizes data bursts initiated by the hard drive
  - Designed and built a custom PCB with voltage translation buffers to handle voltage incompatibility between the FPX and the IDE peripheral bus
  - Separated the Bus Snooper into two clock domains, to allow sampling of the IDE bus signals at a higher frequency (increase sampling accuracy)
  - Developed a control path, both of for the hard drive being snooped and for the Bus Snooper Module, that allowed data retrieved from hard drive over both paths (i.e., through the IDE host controller and through the Snooper) to be viewed side-by-side in real time
- Biological Computation ("BioComp") FPX Module
  - Developed a systolic array-based implementation of the Smith-Waterman local sequence alignment algorithm [35] that was scalable up to the size of the FPGA
  - Implemented pipelining and parallelism to allow the BioComp Module to accept search data at the full width of the FPX's data-path
  - Devised a scheme for extracting snapshots of the state of the systolic array that doesn't involve a fan-in arrangement that scales with the size of the array

  - Developed a Web-based interface to the BioComp module that allowed users to submit runtime parameters and search queries, and then to view the search results, all in real time

- Performance tests of traditional CPU-based search applications

  - Developed a method for reliably measuring CPU execution time down to sub-microsecond accuracy
  - Devised an experiment method that avoids caching effects in the host workstation's secondary storage
  - Devised a scheme for aligning the CPU execution time measurements made on the host workstation with IDE bus activity measurements made by an external device
  - Implemented the core of the BioComp FPX Module as-is in software to provide a direct point of performance comparison

## 6.3 Summary

The search system includes two primary components, the IDE Bus Snooper and the BioComp Module, each of which were programmed onto an FPX's RAD, a Xilinx XCV2000E-6 device with a 2-million gate capacity. The IDE Bus Snooper monitors the traffic of an IDE bus connected to the RAD's test pins by deciphering the AT-API/IDE protocol [27] to extract the contents of data bursts from an IDE hard drive. A second FPX with the BioComp Module then receives the captured IDE data over the WUGS's interconnection network and performs Smith-Waterman local sequence alignment, searching for patterns up to 38 characters long. The BioComp Module was simulated at a maximum clock frequency of 27 MHz, resulting in a target data throughput of 108 MB/sec, and it was tested in hardware at 25 MHz, with a data throughput of 100 MB/s. The IDE Snooper was simulated and tested in hardware at the clock frequency 62.5 MHz, with an on-chip clock doubler in the RAD allowing the Snooper to sample the IDE bus at 125 MHz. Although this would theoretically allow the Snooper to sample the IDE bus at its maximum speed of 50 MHz (100 MB/s data transfer rate), limitations imposed by the architecture of the RAD prevent this transfer rate from actually being attained. Furthermore, the IDE hard drive itself is unable to output data faster than 40.5 MB/sec. Nevertheless, performance

tests conducted for this thesis show that the search system would out-perform a direct software implementation of the Smith-Waterman algorithm by a factor of 50.1. Neglecting the 40.5 MB/sec limitation of the hard drive and assuming data can be delivered to the BioComp Module at its maximum search throughput of 100 MB/sec, this gain becomes 124. These gains are possible because the search system takes advantage of the massive parallelism possible in FPGA designs, and because it sits at a level much closer to the hard drive. Thus, this system serves as a worthwhile testbed for data-searching devices meant to bypass the I/O bottleneck between the CPU and secondary storage.

## 6.4 Future Work

Avenues do exist for further improving the performance of this FPGA-based, high-speed search system, even within the limitations of the FPX/WUGS infrastructure. For example, the custom, hand-assembled PCB used by the IDE Bus Snooper for voltage translation could be replaced with a professionally fabricated device (with better impedance control) that synchronizes the IDE bus signals to the RAD's clock. The computation logic in the rows of the BioComp Module's systolic array could be more efficiently balanced across its pipeline stages to increase the module's maximum clock frequency. In addition, the BioComp Module's state machines could be modified to allow multiple FPX's programmed with the module to operate in parallel, and thus, accommodate search patterns longer than 38 characters. Future versions of this search system that migrate beyond the FPX/WUGS architecture would promise even greater search performance gain. That is, the FPGA programmed with the BioComp module (or any other search application) could be placed within the hard drive itself, in the data-path between the drive's read head and its interface to the host controller. Besides eliminating inefficient interfaces like the voltage translation stage of the IDE Bus Snooper, this arrangement would introduce an additional dimension of parallelism, in that the search system could be replicated for each read head in the hard drive.

# References

[1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] Jeffrey M. Arnold, Duncan A. Buell, and Elaine G. Davis. Splash 2. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–322, 1992.

[3] H.-M. Bluethgen and T. G. Noll. A programmable processor for approximate string matching with high throughput rate. In *Proceedings of Application-Specific Systems, Architectures, and Processors (ASAP)*, 2000.

[4] Florian Braun, John W. Lockwood, and Marcel Waldvogel. Layered protocol wrappers for internet protocol processing in reconfigurable hardware. Technical Report WUCS-01-10, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, July 2001.

[5] Duncan A. Buell, Jeffrey M. Arnold, and Walter J. Kleinfelder, editors. *SPLASH 2: FPGA's in a Custom Computing Machine*. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[6] Roger D. Chamberlain, Ron K. Cytron, Mark A. Franklin, and Ronald S. Indeck. Exploiting truly fast hardware in data mining. Technical Report WUCS-2002-23, Department of Computer Science and Engineering, Washington University in Saint Louis, August 2002.

[7] David J. DeWitt and Paula B. Hawthorn. A performance evaluation of database machine architectures. In *Proceedings of VLDB Conference*, 7th, pages 199–213, 1981.

[8] Chris Dick. Field programmable logic enabling new software radio design. In *International Symposium on Advanced Radio Technologies*, Boulder, CO, USA, March 2002. Institute for Telecommunication Sciences (ITS).

[9] D. M. Dahle et al. Kestrel: Design of an 8bit SIMD parallel processor. In *Proceedings of the Conference on Advanced Research in VLSI*, September 1997.

[10] Inc. Fujitsu. Fujitsu press releases, 2002. `http://pr.fujitsu.com/`.

[11] Steven A. Guccione and Eric Keller. Gene matching using JBits. In *12th International Field-Programmable Logic and Applications Conference (FPL)*, Monpellier, France, September 2002.

[12] P. Guerdoux-Jamet and D. Lavenier. Systolic filter for fast DNA similarity search. In *Proceedings of Application-Specific Systems, Architectures, and Processors (ASAP)*, 1995.

[13] Jeffrey D. Hirschberg, Richard Hughley, and Kevin Karplus. Kestrel: A programmable array for sequence analysis. In *Proceedings of Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 23–34. Application-Specific Systems, Architectures, and Processors (ASAP), August 1996.

[14] Dzung T. Hoang. A systolic array for the sequence alignment problem. Technical Report CS92-23, Department of Computer Science, Brown University, April 1992.

[15] Dzung T. Hoang. Searching genetic databases on splash 2. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[16] Richard Hughley. Massively parallel biosequence analysis. Technical Report UCSC-CRL-93-14, University of California, Santa Cruz, April 1993.

[17] Richard Hughley. Parallel hardware for sequence comparison and alignment. In *Proceedings of CABIOS*, pages 473–476. University of California, Santa Cruz, December 1996.

[18] Intel, Inc. *Using the RDTSC Instruction for Performance Monitoring*, 1997. Available on-line at `http://developer.intel.com/drg/pentiumII/appnotes-/RDTSCPM1.HTM`.

[19] Jack S. N. Jean, Guozhu Dong, Hwa Zhang, Xinzhong Guo, and Baifeng Zhang. Query processing with an FPGA coprocessor board. In *Proceedings of The First International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2001.

[20] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). In *Proceedings of the ACM SIGMOD Intenational Conference on Management of Data*, pages 42–52. ACM, June 1998.

[21] Dominique Lavenier. Dedicated hardware for biological sequence comparison. *Journal of Universal Computer Science (JUCS)*, 2(2), 1996.

[22] Dominque Lavenier. Speeding up genome computations with a systolic array. *SIAM News*, 31(8):1–7, October 1998.

[23] R. J. Lipton and D. Lopresti. A systolic array for string comparison. In *Chapel Hill Conference on VLSI*, pages 363–376. Computer Science Press, 1985.

[24] John W. Lockwood. *Field Programmable Port Extender (FPX) User Guide: Version 2.2.* Department of Computer Science, Washington University, http://www.arl.wustl.edu/arl/projects/fpx, June 2002.

[25] John W. Lockwood. Field programmable port extender homepage, 2002. `http://www.arl.wustl.edu/arl/projects/fpx/`.

[26] John W. Lockwood, Jon S. Turner, and David E. Taylor. Field programmable port extender (FPX) for distributed routing and queuing. In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, pages 137–144, Monterey, CA, February 2000.

[27] Peter T. McLean. *Information Technology - AT Attachment with Packet Interface - 5 (ATA/ATAPI-5), Revision 3.* T13 Technical Committee, February 2000.

[28] C. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48(3):443–53, 1970.

[29] E. A. Ozkarahan, S. Shuster, S. A. Smith, and K. C. Smith. RAP - associative processor for database management. In *AFIPS Conference*, 1975.

[30] E. A. Ozkarahan, S. A. Shuster, and K. C. Sevcik. Performance evaluation of a relational associative processor. *ACM Transactions on Database Systems*, 2(2):175–195, June 1977.

[31] Erik Riedel. *Active Disks - Remote Execution for Network-Attached Storage*. PhD thesis, Carnegie Mellon University, November 1999.

[32] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th VLDB Conference*, 1998.

[33] Raghu Sastry, N. Ranganathan, and Klinton Remedios. CASM: A VLSI chip for approximate string matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(9):824–30, August 1995.

[34] R. K. Singh, W. D. Dettloff, V. L. Chi, D. L. Hoffman, S. G. Tell, C. T. White, S. F. Altshul, and B. W. Erickson. BioSCAN: A dynamically reconfigurable systolic array for biosequence analysis. In *Proceedings of CERCS96*. National Science Foundation, June 1996.

[35] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 141(1):195–7, 1981.

[36] Todd Sproull, John W. Lockwood, and David Taylor. Control and configuration software for a reconfigurable networking hardware platform. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[37] David E. Taylor, John W. Lockwood, and Sarang Dharmapurikar. Generalized RAD module interface specification of the field-programmable port extender (FPX). Technical Report WUCS-TM-01-15, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, July 2001. Available on-line at `http://www.arl.wustl.edu/arl/projects/fpx/wugs.ps`.

[38] David E. Taylor, John W. Lockwood, and Naji Naufel. RAD module infrastructure of the field-programmable port extender (FPX). Technical Report WUCS-TM-01-16, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, July 2001.

[39] J. Turner, T. Chaney, A. Fingerhut, and M. Flucke. Design of a Gigabit ATM Switch. In *Proceedings of Infocom 97*, March 1997.

[40] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Evaluation of active disks for decision support databases. In *Proceedings of HPCA*, 2000.

[41] Bin Wang. Implementation of a dynamic programming algorithm for DNA sequence alignment on the Cell Matrix architecture. Master's thesis, Utah State University, 2002.

# Vita

Benjamin M. West

**Date of Birth** January 6, 1977

**Place of Birth** Louisville, KY

**Degrees** B.S. Cum Laude, Electrical Engineering, August 2000
B.S. Cum Laude, Computer Engineering, August 2000
B.A. Magna Cum Laude, Germanic Languages and Literature, August 2000

May 2003