Accelerating Next Generation Genome Reassembly in FPGAs: Alignment Using Dynamic Programming Algorithms

Maria Kim

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering

University of Washington

2011

Program Authorized to Offer Degree:
Department of Electrical Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a master's thesis by

Maria Kim

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final examining committee have been made.

Committee Members:

_____

Scott A. Hauck

_____

Carl Ebeling

Date: _____

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Washington, I agree that the Library shall make its copies freely availab le for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes consistent with "fair use" as prescribed by the U.S. Copyright Law. A ny other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature _____

Date _____

University of Washington

Abstract

Accelerating Next Generation Genome Reassembly in FPGAs: Alignment Using Dynamic Programming Algorithms

Maria Kim

Chair of the Supervisory Committee:
Professor Scott A. Hauck
Electrical Engineering

DNA sequencing has proven to be advantageous in a multitude of applications, but challenges in computation has hindered its widespread growth in various fields. Although innovative sequencing machines have been able to process millions of DNA segments in parallel, reassembling these short read pieces has become a bottleneck in the computation. We use a hardware platform and various algorithms to significantly accelerate current software systems by utilizing a FPGA as a coprocessor. Our reassembly process requires two principle steps: 1) the Matcher, which implements the BFAST algorithm and 2) the Aligner, which manages multiple alignment computations. This thesis covers the Aligner, which uses a hybrid sequence alignment dynamic programming algorithm to obtain the best alignment for the short reads. The algorithm, design, and results of this thesis describe the implementation, as well as the resulting improvements in computation speed.

TABLE OF CONTENTS

LIST OF FIGURES iii

LIST OF TABLES v

ACKNOWLEDGEMENTS vi

1. Introduction 1

2. Background 3

2.1 DNA Sequencing 3

2.2 Issues with Reassembly 9

2.3. Hardware Platform 12

3. Algorithms 17

3.1 Dynamic Programming Algorithms 17

3.2 Smith-Waterman vs. Needleman-Wunsch Algorithm 22

3.3 Computational Burden 26

3.4 The BFAST Algorithm 27

4. System Design 29

4.1 The Matcher 30

4.2 The Aligner 36

5. Design Specifications 38

5.1 Top Level 38

5.2 RAM Controller 40

5.3 Computation Array 44

5.4 Controller 56

5.5 Track 61

6. Results 65

6.1 Score Computation 65

6.2. System Runtime 67

6.3. Aligner Runtime 69

6.4. Resource Utilization 72

7. Conclusions 73

7.1. Conclusions 73

7.2. Future Design Considerations 73

References.. 78

LIST OF FIGURES

Figure 1: Structure of DNA [1] 3
Figure 2: Decomposed DNA Strand 4
Figure 3: Aligning 200 million Illumina short reads to a 3 billion base pair reference sequence using a string matching technique. 8
Figure 4: Single Nucleotide Polymorphisms (SNPs) 10
Figure 5: Sequence Errors 11
Figure 6: Difference and Error-free Alignment 12
Figure 7: FPGA Structure 13
Figure 8: Co-processing 14
Figure 9: Dynamic Programming Score Matrix 17
Figure 10: Initialized Matrix 18
Figure 11: Derived Scores 19
Figure 12: Smith-Waterman Example [21] 20
Figure 13: Smith-Waterman Alignment [21] 21
Figure 14: Global Alignment [21] 23
Figure 15: Needleman-Wunsch Initialization 24
Figure 16: Hybrid Initialization 25
Figure 17: CALs of a Read 27
Figure 18: Overall System 29
Figure 19: Generating Seeds 30
Figure 20: Creation of the RIT 31
Figure 21: Using the RIT 32
Figure 22: Seed Division 33
Figure 23: Hash Table and Index Table 33
Figure 24: System with Filter 34
Figure 25: Data Flow of System with Matcher Highlighted 35
Figure 26: Data Flow of System with Aligner Highlighted 36
Figure 27: Top Level Module 40
Figure 28: RAM Controller Control Logic 41
Figure 29: Reference in Memory 42
Figure 30: Aligned CAL in Memory 43
Figure 31: Computation Array 45
Figure 32: Multiple Reads 46
Figure 33: Section of the Score Matrix with Derived Scores 48
Figure 34: Score Computation 50
Figure 35: Clock Cycles in Computation Array 52
Figure 36: Inputs for a Computation Unit 53
Figure 37: RefReady and ReadReady 56
Figure 38: Controller Diagram 58
Figure 39: Controller Flow Control 60

Figure 40: Track Diagram 62
Figure 41: CAL, Position, and Location 63
Figure 42: Aligner Modules 64
Figure 43: Computation Array II 65
Figure 44: System Runtime on Chromosome 21 68
Figure 45: Graph of Multiple Computation Arrays 70
Figure 46: Resource Utilization in the Aligner 72

LIST OF TABLES

Table 1: Dynamic Programming Algorithm for Sequence Alignment, Steps 20
Table 2: Hybrid Algorithm 23
Table 3: BFAST Actual Runtime 28
Table 4: Aligned Reference 44
Table 5: Unaligned Reference 44
Table 6: Score Computation 49
Table 7: Computation Array II, Speed 66
Table 8: Computation Array II, Area 66
Table 9: Computation Arrays, Throughput in Alignments/slices-ns 67
Table 10: System in Hardware vs. BFAST 69
Table 11: Bottleneck Runtimes 71
Table 12: System vs. Aligner Throughputs 71

ACKNOWLEDGEMENTS

First of all, I would like to thank Pico Computing (Seattle, WA) and the Washington Technology Center (Seattle, WA) for providing funding to make the completion of this research project possible. I'd also like to give many thanks to Scott Hauck, who advised me every step through this journey; Carl Ebeling and Larry Russo for sharing their knowledge; Corey Olson for creating the other half of this design and providing me with much needed assistance; and Boris Kogon and Cooper Clauson for building a foundation for the design. Last, but most definitely not least, I thank my loving family and friends for their unfailing encouragement; I couldn't have done anything without their support.

1. Introduction

Since the completion of the Human Genome Project in 2003 [6], DNA sequencing has reached revolutionary speeds in the computation of the human genome. Next Generation Sequencing machines massively parallelize DNA segments and output millions of short read data to be analyzed. However, a large part of DNA data analysis involves reassembly, and current reassembly software systems cannot keep up with these new sequencing machines.

By utilizing parallelism, Field Programmable Gate Arrays (FPGAs) have the ability to ease the computational burden. While software systems are somewhat limited in their ability to parallelize programs, FPGAs have the advantage of an intrinsic parallel structure to handle large amounts of genomic data. The execution speed of hardware coprocessors has proven to achieve significantly higher performance. Because reassembly involves independent threads of computation, it is able to take advantage of this massive parallelism.

Because of the ability to significantly accelerate the entire DNA sequencing process, researchers from a wide variety of fields will be able to take advantage of what is stored in genomic data. DNA data contains valuable information which can help reveal various processes of biological occurrences. This data may assist in fields ranging from medicine to forensics, revolutionizing research in numerous areas.

We implemented a short read reassembly system in the design, which contains two main parts, the Matcher and the Aligner. The Matcher adapts an existing algorithm to hardware in order to cut down the number of computations required in alignment. The Aligner receives data from the Matcher and runs multiple processes in parallel to accelerate the computation.

This thesis covers the Alignment portion of the design, and the following chapters are documented:

- **Chapter 2: Background** gives background information on DNA sequencing, reassembly, alignment, and FPGAs.
- **Chapter 3: Algorithms** describes the algorithms used in the design.
- **Chapter 4: System Design** provides information on the short read reassembly design in hardware.
- **Chapter 5: Design Specifications** breaks down each component of the Aligner.
- **Chapter 6: Results** presents the outcome of our designs.
- **Chapter 7: Conclusion** provides information on possible additions to the design, as well as an overall conclusion to the thesis.

2. Background

2.1 DNA Sequencing

Deoxyribonucleic acid (DNA) is arranged in a double helix structure, in which two parallel sugar phosphate backbone strands are linked with four different nucleotide bases: adenine, thymine, guanine, and cytosine. Figure 1 illustrates the structure of DNA and the nucleotide bases that hold this bonded, paired structure.

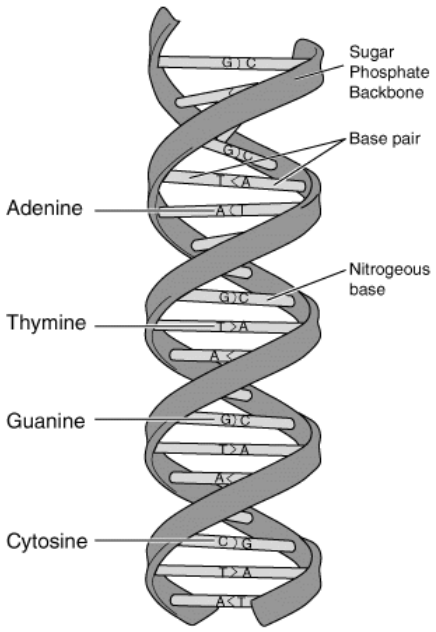

Figure 1: Structure of DNA [1]

The process of DNA sequencing determines the arrangement of these nucleotide bases in a molecule of DNA, and this arrangement contains genetic information. To sequence DNA, the identification of only one of a pair of bases is required, since "A" always pairs with "T" and "C" always pairs with "G". Consequently, by sequencing one side of the DNA strand, the

other side is automatically known. Imagine unraveling the structure of Figure 1 into the one in Figure 2.

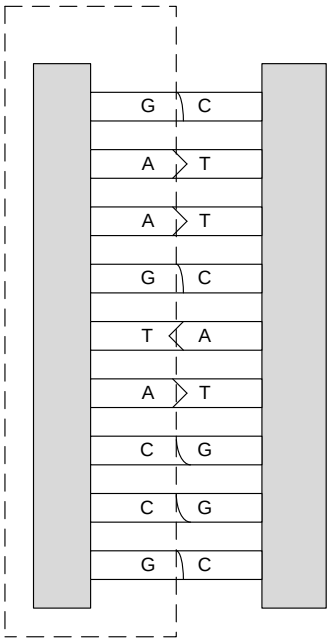

Figure 2: Decomposed DNA Strand

From Figure 2, we can "read" or sequence the bases on the left, as the string "GAAGTACCG". With this string identified, its pair is recognized as "CGGTACTTC".

A sequenced strand of DNA has the string of nucleotide bases identified, and locating these nucleotide bases in the genome has proven to be useful in various applications. The following sections describe the applications that utilize DNA sequencing, as well as the history and process behind it.

2.1.1 Applications

DNA sequencing has become a valuable asset to a wide range of applications, including medicine, biological research, and forensics. Personalized medicine has been of great interest in recent years, and genetic information allows the discovery of disease susceptibility and

genetic risk factors [2]. In the future, many medical specialists hope to locate the cause of genetic diseases, such as certain cancers, diabetes, and hypertension, and exploit individual genetic data for prevention purposes. In addition to protective measures, identifying a malignant gene may be useful in the discovery of treatments and remedies.

With the ability to access sequencing technology, researchers could make thorough studies of biological organisms. There are an estimated 30 million different species in the world [19], providing a massive amount of raw data to work with. If DNA sequencing was more readily available, an expansion of evolutionary knowledge is within reach. Genetic data can allow biologists to compare differing organisms, as well as similar individuals, since the genomic structure of each DNA strand varies.

In forensics, DNA sequencing assists the identification process in challenging cases. Most criminals inevitably leave behind a trace of DNA, and DNA sequencing tools could use this evidence to help reveal the suspect(s) involved in the crime. This will result in less time, effort, and money spent during the investigation. An accurate identification process will not only ease current forensic issues, but it may also result in deterrence of future crimes.

These initial applications provide a motivation for this study, and have proven that DNA sequencing can be applied to a range of different fields. Many more applications and benefits are expected to arise as the technology evolves.

2.1.2 Human Genome Project and Sequencing Goals

Due to the benefits of DNA sequencing, people became interested in sequencing the first human genome and began the Human Genome Project. This project was an arduous, time-

consuming, and costly task. The human genome is composed of three billion nucleotide base pairs (bp), whereas smaller bacterial genomes such as E. coli have merely 5 million bp [5], a difference of 600 times. Because of the sheer number of nucleotide bases in the human DNA, the Human Genome Project was a process with an official time span of 13 years, from 1990-2003, and a price tag of $3 billion [6]. From this project, the first human genome was sequenced.

Despite these hindrances, many prospective opportunities arose from this project. Various groups were able to initiate sequencing goals for many more genomes. The 1000 Genome Project (http://www.1000genomes.org) [7] is determined to catalog the genomic sequence of a thousand different humans. With newer sequencing technologies, researchers expect to bring the $3 billion dollar price tag down to $1000 per genome [8]. All these ambitious goals will allow a range of applications to be within reach, and have become a primary task for many research groups.

2.1.3 Sequencers

Since the first DNA sequencing publication by Sanger in 1977 [3], computational biologists have put much effort in generating new, more efficient sequencing tools. As a result, in 2004 Next Generation Sequencing machines have changed the outlook for DNA research [4]. Although sequencing DNA has accelerated significantly, the analysis of the resulting data is unable to keep up. In fact, sources [2] have indicated that the computation of DNA sequencing analysis has fallen behind the processing capabilities of modern technology.

Next Generation Sequencing machines randomly slice DNA strands into “short read” components and sequence millions of bases in parallel. The lengths of the short reads have

decreased considerably from the 650-800 bp reads of the original Sanger sequencing technique; current Illumina/Solexa and Roche 454 machines generate short reads ranging from 35 bp to 250 bp [1]. The shorter reads result in massive parallelization, and therefore an efficient method of sequencing DNA. However, these shorter reads also lose valuable information for reassembly, which may prove to be significantly more costly and demanding [4].

2.1.4 Reassembly and Alignment

As described in the previous sections, the process of sequencing a human genome has reached incredible speeds; Next Generation Sequencing machines have the ability to sequence many nucleotide bases in parallel. However, another part to the sequencing process exists, which involves assembling these short read pieces back to its 3 billion base pair strand. There are two primary methods of performing this task: alignment and de Novo.

Alignment is the process rearranging the short reads into its 3 billion base pair form by using a string matching technique. An already sequenced and assembled "reference" genome is compared against the short reads. This concept is displayed in Figure 3 with a DNA sequencing machine from Illumina/Solexa, which outputs approximately 200 million short reads for one human genome [4].

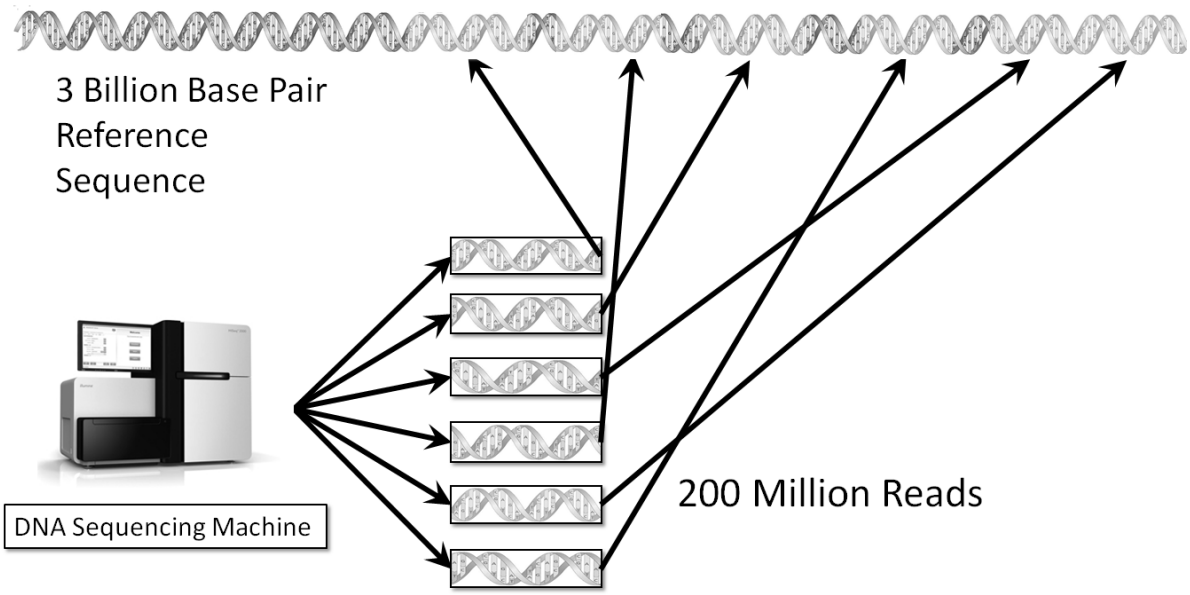

Figure 3: Aligning 200 million Illumina short reads to a 3 billion base pair reference sequence using a string matching technique.

These 200 million 75-100 bp reads are then mapped to a 3 billion bp reference sequence via a string matching procedure. Note that the reference genome and the produced short reads must have an appropriate evolutionary distance for accurate alignments [9]; the two sequences must be similar enough for a base by base comparison. Because a sequenced human genome exists from the Human Genome Project, and the similarity between two humans is found to be approximately 99.5% [10], alignment is a suitable process for reassembling human short read data.

The other known method for reassembly is called de Novo. This method does not use a reference sequence, and is categorized in a class of NP-hard problems, where no efficient exact solution is known [11]. Because our design utilizes alignment for reassembly, no further discussion of de Novo will be given. However, more information can be found in Paszkiewicz and Studholme [12].

2.2 Issues with Reassembly

Reassembly is a difficult task, partially due to the massive amount of computation required. This amount of computation is necessary because of the various conflicts that occur in the genome, including the repetition of bases, differences between the compared genomes, and machine errors. All these issues will be discussed in the following sections.

2.2.1 Repetition

The human genome contains many repetitive regions, which are more likely to create alignment problems for shorter read lengths. With the shorter read lengths in Next Generation Sequencing, repetition becomes a more frequent occurrence. Shorter read lengths will cause more identical segments of nucleotide bases, which results in ambiguous alignments. This issue is treated in various ways, depending on the program used. Some algorithms will use the first matching short read for the alignment, while others disregard repetitive areas completely; neither method solves the problem. However, an increase in the short read length will decrease the probability of repetitions from occurring. Current sequencing machines are working to increase the length of the short reads, while maintaining its advantageous performance.

2.2.2 Biological Differences

Biological differences between the read and reference occur at a rate of approximately 0.5% [10]. One of the most common forms of these variations is called Single Nucleotide Polymorphisms (SNPs), which are found where the read and reference differ by a single nucleotide base. SNPs occur due to the diversity in human populations. For example, a

disease may occur in one person, but not in another because of SNPs. An example of a SNP is displayed in Figure 4.

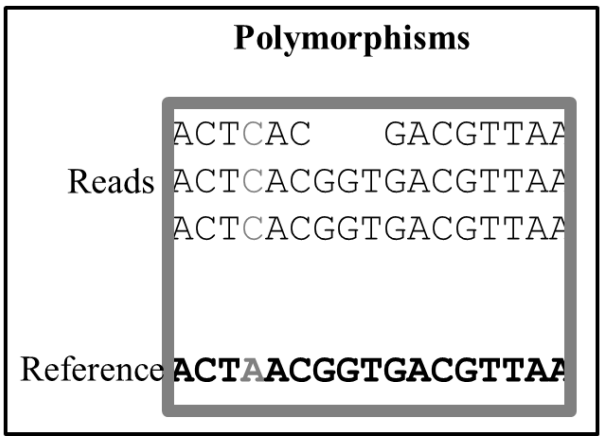

Figure 4: Single Nucleotide Polymorphisms (SNPs)

In gray, Figure 4 shows that the nucleotide base "A" does not match with the "C"s in the read, which is an indication of a SNP. Note that there are multiple layers of short read sequences in alignment; this feature becomes particularly useful when errors or differences occur in the short read because the depth of the reads allow for researchers to identify SNPs.

Other biological variations that occur in genomic sequences include insertions and deletions (indels), in which a segment of the read differs from the reference due to an insertion or deletion of one or more bases. Indels appear to shift bases in different locations of the genome in alignments. They also create gaps in either the read or the reference.

2.2.3 Machine Errors

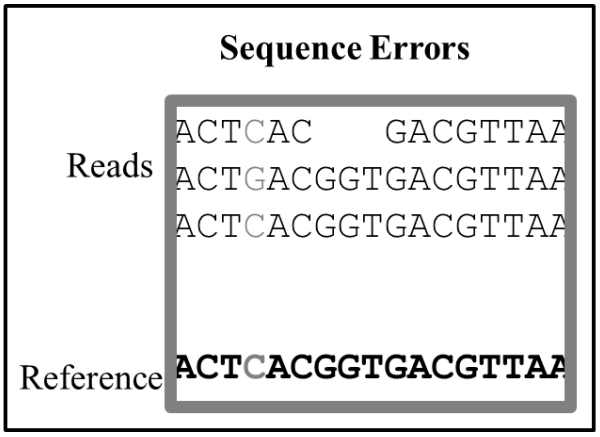

Figure 5: Sequence Errors

A machine error occurs when the DNA sequencing machine incorrectly calls a base. For the example in Figure 5, a "G" is reported in the genome where a "C" actually occurs. This can be differentiated from the SNP in Figure 4, because a sequencing machine error generally occurs in a single row of reads. In this way, the depth of the short reads helps distinguish a SNP or indel from a machine error.

2.2.4. Handling Reassembly Issues

If no differences occurred between the reads and the reference, alignments can be made quickly and accurately. An error-free and difference-free alignment is shown in Figure 6.

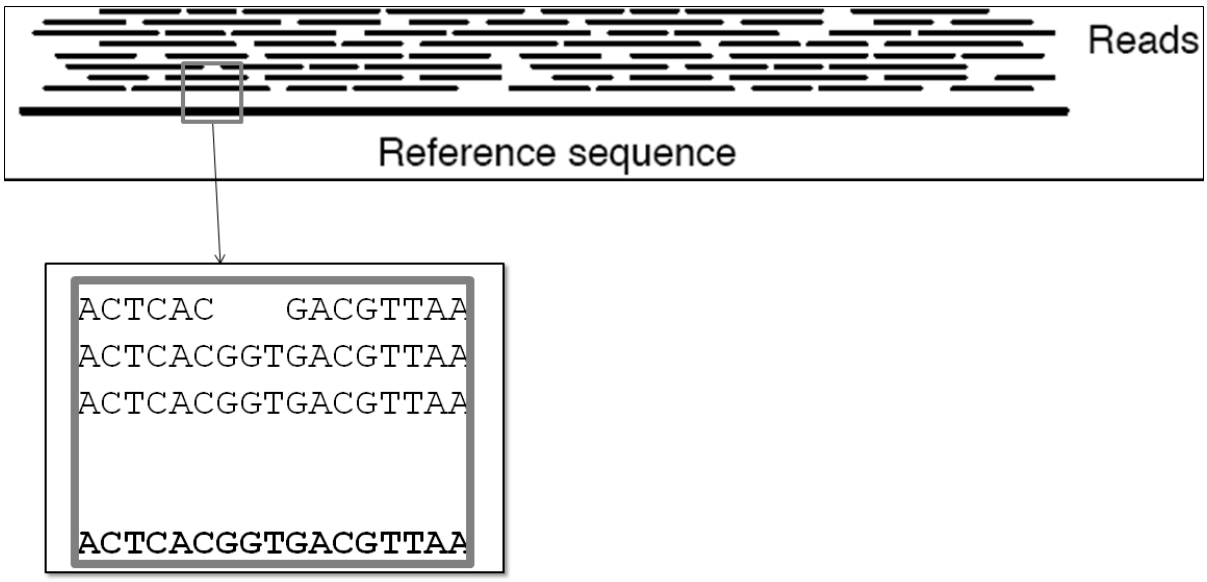

Figure 6: Difference and Error-free Alignment

In Figure 6, a quick string comparison is made to align the sequenced short reads to the reference. However, because biological differences and DNA sequencing machine errors are inevitable, an accurate string matching algorithm must be used for alignments.

Sequence Alignment dynamic programming algorithms have been shown to give accurate alignments by considering these issues. However, the performance of the dynamic programming algorithms has proven to be a setback. This issue can be partially resolved by using a hardware platform to compute the dynamic programming solution.

2.3. Hardware Platform

Current alignment software programs, such as BFAST [13] and MAQ [14] produce a relatively accurate consensus genome by utilizing a sequence alignment dynamic programming algorithm. However, computational speeds have become a major obstacle. We can accelerate these computation times significantly by using the same algorithms on a

hardware platform. Furthermore, Field-Programmable Gate Arrays (FPGAs) have proven to be an appropriate platform candidate due to its parallel structure and flexibility.

2.3.1 The FPGA Structure

An FPGA will not meet current CPU processing speeds in terms of frequency, but it is able to perform many computations in parallel. The structure of a typical Xilinx FPGA is shown in Figure 7.

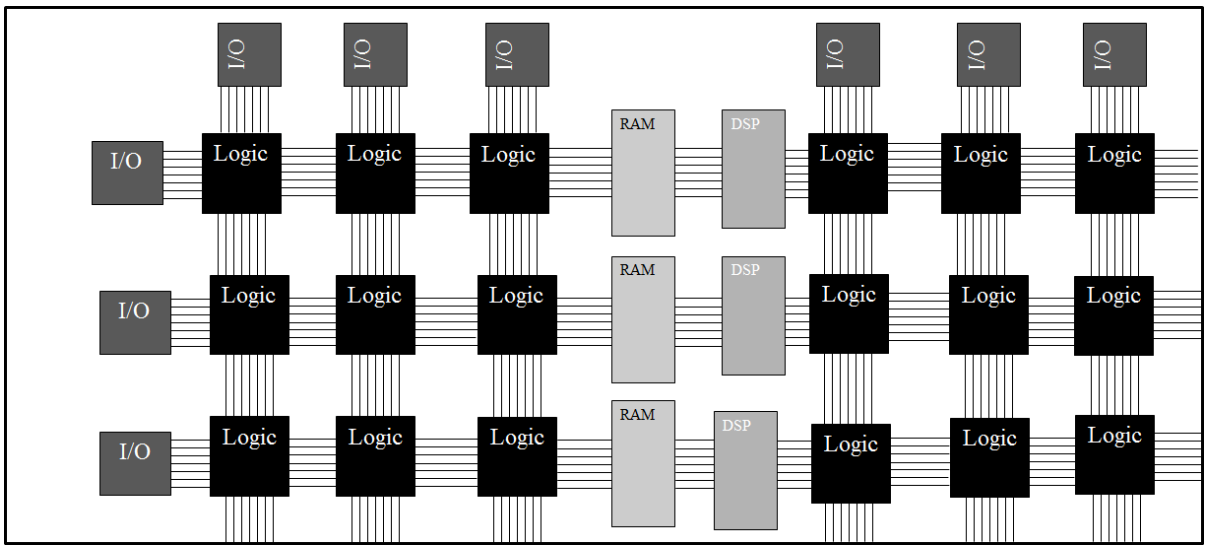

Figure 7: FPGA Structure

Each logic block of the FPGA is composed of logic elements and registers. The logic elements have lookup tables (LUTs) to perform combinational logic, represented using a truth table. The registers implement sequential logic to handle state-holding operations. The LUTs and registers are used together to execute the programmable logic. [16]

In addition to the numerous logic blocks, FPGAs provide DSP blocks for efficient multiplication, internal memories for data access, multiple clock speeds for user flexibility, and input/output ports for external communication. Because of these features, FPGAs are

commonly used as a coprocessor for systems with large amounts of computations. The first step of using a coprocessor involves streaming data from the host, as shown in Figure 8. Then the FPGA performs computations in parallel, using the received data. Once results have been computed, they are sent back to the host processor for further data analysis.

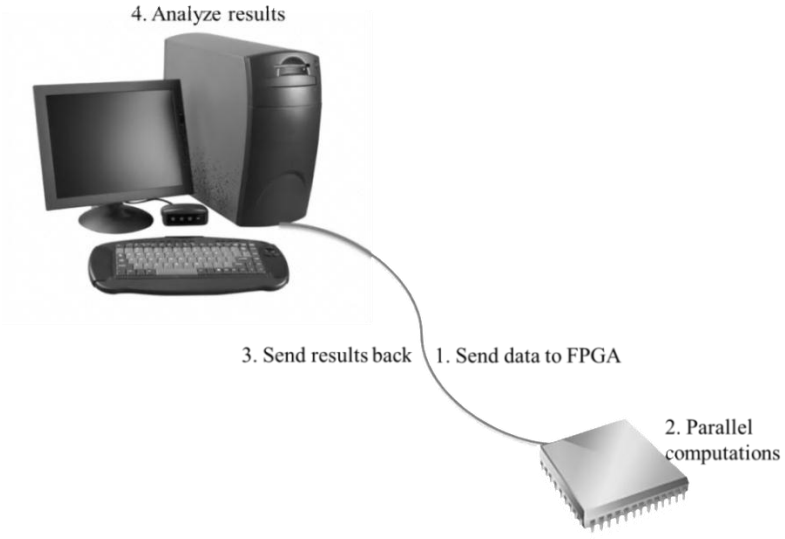

Figure 8: Co-processing

Although parallel processing appears to be universally effective, programs can take advantage of it only when the application permits. In order to utilize the FPGA, independent, repetitive computations must be inherent in the application. DNA reassembly is able to take advantage of the FPGA because reassembly entails massive amounts of base by base comparisons. The Design Specification (Chapter 5) of this thesis further examines this observation by describing the specific computations necessary for DNA reassembly.

2.3.2 Advantages of the FPGA

In addition to the FPGA, other accelerators exist for massive parallel computing. One example is the Graphics Processing Unit (GPU), which is commonly used to accelerate the

process of rendering three-dimensional (3-D) graphics. GPUs are composed of a large number of fine-grained parallel processors to assist in real-time, operation-heavy computations [20]. However, in cases where many random accesses are made to external memory, latency decreases the performance of GPUs significantly. This limitation applies to the DNA reassembly application because of the randomness of the short reads. The genomic data stored in external DRAM is read in an unpredictable order, which slows down the computation process on a GPU.

Alternatively, application-specific integrated circuits (ASICs) have the best performance and compute 3-4 times faster than FPGAs [17]. However, unlike GPUs and FPGAs, ASICs cannot be reprogrammed. This limiting factor results in an increase in implementation time as well as NRE costs.

FPGAs prove to be a useful middle ground between ASICs and CPUs. It is able to generate large computations in parallel and offers the flexibility of reprogramming. Accordingly, FPGAs appear to be the best hardware platform for a short read reassembly accelerator.

2.3.3 Pico Computing

Pico Computing provides hardware development systems that contain FPGAs, external memories, and communication ports. API libraries and examples are also offered to support the hardware system. This project uses Pico Computing's M-501 model. The M-501 uses Xilinx's Virtex-6 LX FPGA, x8 PCI Express Host Interface, and 512 MB DDR2. The PCI Express interface allows efficient bandwidth between the host software and the Xilinx FPGA. In addition, the 512 MB of DDR2 can be utilized to store large amounts of data.

Pico Computing's products ease the design of coprocessor systems by having a range of features for hardware designers. The provided memory controller uses a multi-port manager for projects with several memory streams. By querying each port in a round robin fashion, the controller is able to keep memory accesses from different modules independent. Furthermore, the software API allows C++ programs to transfer large quantities of data via PCI Express. Because 200 million 76 bp short reads must be streamed from the host, a large bandwidth is needed for this design.

3. Algorithms

3.1 Dynamic Programming Algorithms

Dynamic programming algorithms for DNA sequencing are commonly used as part of the reassembly process. These types of algorithms compare the sequence of two genomes using a string matching procedure. The score reveals the best alignment by rewarding matches and penalizing indels and mismatches. As a result, the best score comes from a perfectly matched, gapless sequence of bases.

3.1.1. Scoring

Because dynamic programming sequence alignment algorithms incorporate indels as penalties in the score, they are more accurate than a standard string comparison. The first step in these algorithms is to set one sequence to be compared on the left side of the matrix, while setting the other sequence across the top. For our design, the read sequence is on the left side of the matrix, while the reference sequence is displayed across the top, as shown in Figure 9.

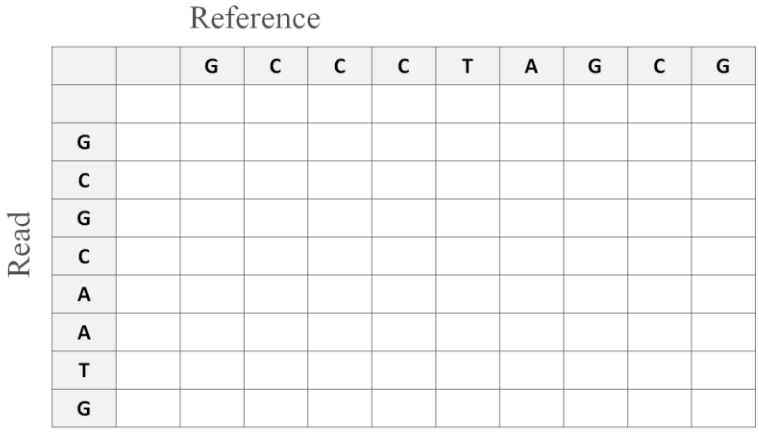

Figure 9: Dynamic Programming Score Matrix

The second step of this process involves initializing the matrix by setting the first row and first column of the score matrix to an initialized value. These initialized values depend on the algorithms used, which will be further discussed in the next section (3.2. Smith-Waterman vs. Needleman-Wunsch). The initialized matrix is shown in Figure 10.

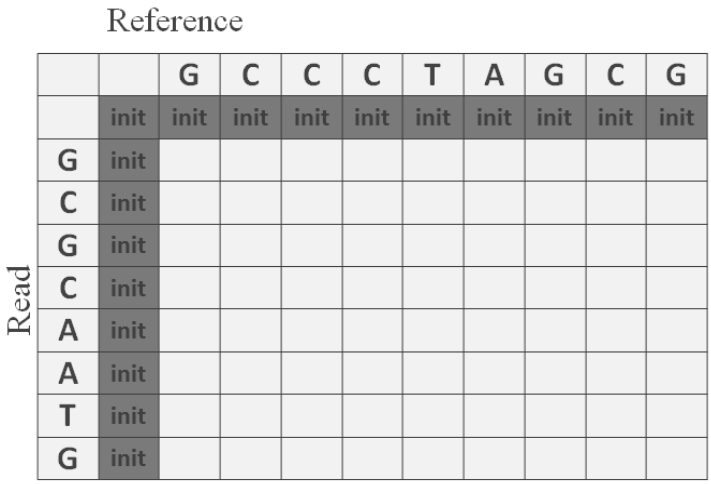

Figure 10: Initialized Matrix

After the second step, the third step requires the derivation of scores from neighboring elements in the matrix. These neighboring locations include the following three units: 1) the above left unit, 2) the left unit, and 3) the unit above, as illustrated in Figure 11.

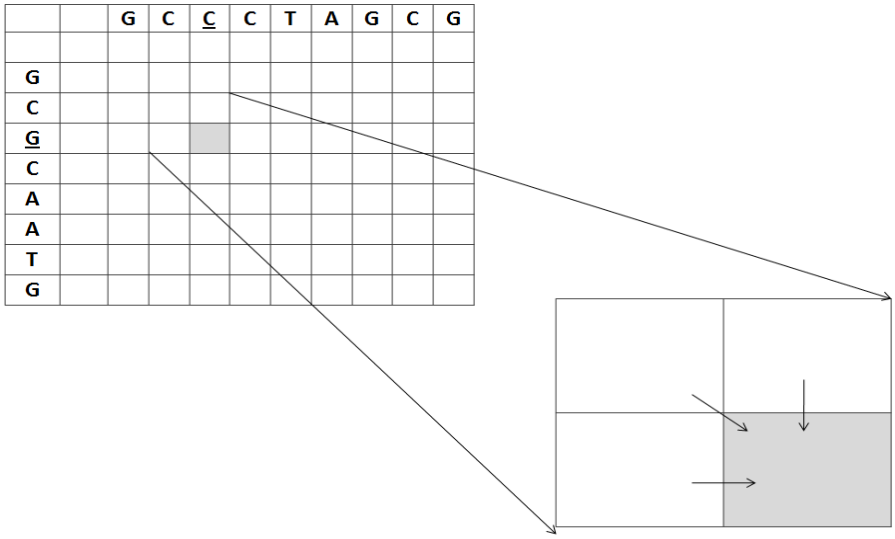

Figure 11: Derived Scores

Once these scores are derived, match bonuses, mismatch penalties, and gap penalties are computed on each score; this is the fourth step to the sequence alignment dynamic programming algorithm. A score derived from the upper left corner creates a contiguous alignment at that particular location in the alignment. For these types of scores, the aligning bases are compared to give a match bonus or mismatch penalty. For Figure 11, these bases are "G" for the read and "C" for the reference sequence. On the other hand, a score derived from the remaining above or left locations signifies a "gap" or a deletion in either the read or reference; these scores are given gap penalties. A score from the left indicates that there is a gap in the read, which is equivalent to an insertion in the reference. Conversely, a score from the above unit signifies a gap in the reference sequence or an insertion in the short read. Once these scores are computed, the maximum of the computed scores is the final score of that element in the matrix.

The fifth step of a sequence alignment dynamic programming algorithm involves searching through the matrix to find the highest score, and therefore the best alignment location for the

two alignments. The highest score is where the last base in the alignment occurs. A summary of the five steps are displayed in Table 1.

Table 1: Dynamic Programming Algorithm for Sequence Alignment, Steps

Step	Process
1.	Set the reference sequence across the top of the 2x2 matrix and the read sequence along the side.
2.	Initialize the first row and first column of the score matrix (values depend on algorithm)
3.	For each element, derive scores from neighboring above, above-left, and left units.
4.	For each element, compute match and mismatch scores on above-left score and gap score on above and left scores. Choose maximum of computed score as final score.
5.	Once all elements in matrix are filled, find the highest score, which is where the last base in the alignment occurs.

An example of the Smith-Waterman algorithm, one of the most commonly used sequence alignment dynamic programming algorithms, is given in the next section.

3.1.2. Smith-Waterman Algorithm Example

Reference

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	0	0	0	0	0	1	0	1
C	0	0	2	1	1	0	0	0	2	0
G	0	1	0	1	0	0	0	1	0	3
C	0	0	2	1	2	0	0	0	2	1
A	0	0	0	1	0	1	1	0	0	1
A	0	0	0	0	0	0	2	0	0	0
T	0	0	0	0	0	1	0	1	0	0
G	0	1	0	0	0	0	0	1	0	1

Read

Figure 12: Smith-Waterman Example [21]

In the example of Figure 12, the first step involves setting the reference sequence across the top and the read sequence along the side. Then, for the Smith-Waterman [23] algorithm, the first row and column is set to zeros. Once the score matrix is initialized, each element derives a score from its above, above-left, and left neighbors. For the dotted-lined element in Figure 12, all zeros are derived from the neighboring cells.

For this example, a match score of +1, a mismatch of -1, and a constant gap penalty of -1 are given. Thus, a score of -1 (gap), +1 (match), and -1 (gap) are computed from the above, above-left, and left cells, respectively. The highest score of the three is then the final score of that cell, which is +1. This method is then implemented on every cell of the matrix.

In addition, the Smith-Waterman algorithm finds the best alignment at the highest scoring matrix element. For the example in Figure 13, +3 occurs in the last "G" of the reference and the third "G" of the read sequence. The final alignment is therefore "GCG", as shown in Figure 13.

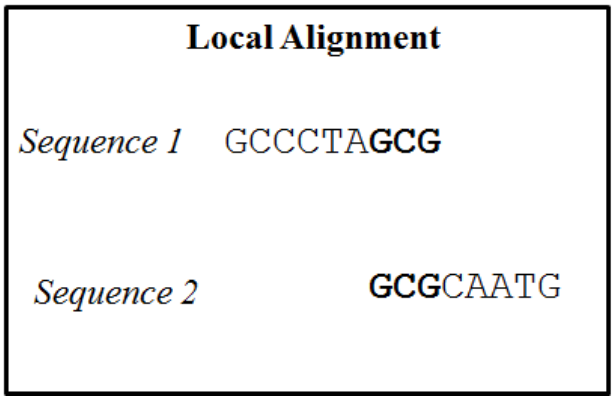

Figure 13: Smith-Waterman Alignment [21]

3.1.3. Affine Gap Scheme

There are two major methods to deal with scoring indels: a constant gap scoring scheme, and an affine gap scoring scheme. The constant gap scoring method uses a constant value to penalize each gap individually. This method was used in the Smith-Waterman algorithm example of the previous section.

In contrast, the affine gap scoring scheme is more complex because it penalizes a new gap, or an "open-gap", more than an "extended-gap". An open-gap has no other gaps directly preceding it, while an extended-gap prolongs an existing gap in the same direction. The affine gap scoring method is more accurate because a large insertion, as opposed to multiple single insertions, is considered to be a better alignment biologically; therefore, we implemented the affine gap scoring into our system.

3.2 Smith-Waterman vs. Needleman-Wunsch Algorithm

In addition to the Smith-Waterman algorithm, the Needleman-Wunsch [22] is another commonly used sequence alignment dynamic programming algorithms. The Smith-Waterman algorithm is a local sequence alignment algorithm, while the Needleman-Wunsch is a global sequence alignment algorithm. This section will discuss the difference between these two algorithms as well as their implementations.

3.2.1 Differences

The Smith-Waterman Algorithm is a local sequence alignment algorithm, which finds similar areas in two sequences. It does not penalize gaps in the beginning and end of a sequence, so it is most advantageous in finding sections of perfect alignment. However, because the

alignment is not constrained to the entire sequence, large areas of bases may be neglected, which was shown in the previous Smith-Waterman example of Figure 13.

Alternatively, the Needleman-Wunsch Algorithm is a global sequence alignment algorithm, which compares the entire sequence of two genomes, end-to-end, as illustrated in Figure 14.

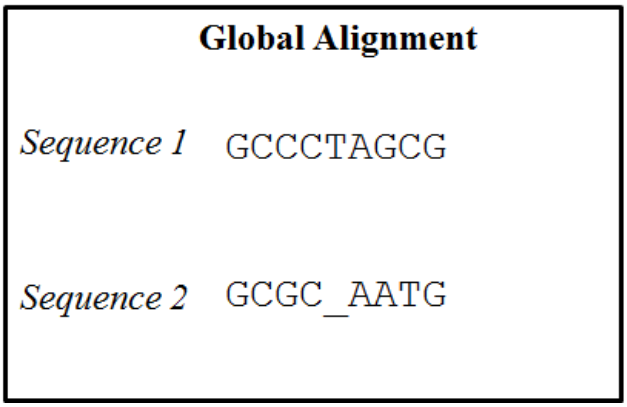

Figure 14: Global Alignment [21]

Because of this end-to-end comparison, the Needleman-Wunsch Algorithm is useful for utilizing an entire short read sequence. However, there are cases where the best alignment is needed for sections of the sequence, rather than the entire strand.

As a result, we chose to use a hybrid Smith-Waterman/Needleman-Wunsch algorithm. In our design, the short read and the reference use different algorithms. The short read sequence utilizes the Needleman-Wunsch algorithm while the reference sequence uses the Smith-Waterman algorithm. A summary of these observations is listed in Table 2.

Table 2: Hybrid Algorithm

Algorithm	Usage	Description
Smith-Waterman	Reference Sequence	Allows a portion of the sequence to be aligned
Needleman-Wunsch	Short Read Sequence	Constrained to use entire sequence for comparison

The implementation of this hybrid algorithm is accomplished in the initialization stage of the program, while the core of the dynamic programming algorithm remains the same.

3.2.2. Implementation

The main difference in implementing a local alignment and a global alignment lies in the initialization of the scoring matrix. Initializing a dynamic programming matrix requires assigning values to the first row and column. The Smith-Waterman algorithm sets both the first row and column to zeros, as shown in the previous example of Figure 12.

Alternatively, the Needleman-Wunsch algorithm uses a gap-score initialization. Only the top upper-left corner is assigned to a zero, while other scores in the first column and row are assigned to a gap score. Figure 15 shows how the initialization portion of the matrix would differ from the Smith-Waterman algorithm.

Reference

		G	C	C	C	T	A	G	C	G
	0	-2	-3	-4	-5	-6	-7	-8	-9	-10
G	-2									
C	-3									
G	-4									
C	-5									
A	-6									
A	-7									
T	-8									
G	-9									

Read

Figure 15: Needleman-Wunsch Initialization

Note that only the initialization scores are filled in for this figure. For our algorithm, -2 is used for the open gap penalty, while -1 is used as the extended gap penalty. In addition, +2 is

used as the match bonus and -2 is used the mismatch penalty. The first base in the first row or column receives an open-gap score because it aligns to a gap at the beginning of the sequence. The rest of the units in the row or column receive an extended-gap score, in addition to the preceding gap scores.

From these two initialization schemes, the hybrid Smith-Waterman/Needleman-Wunsch algorithm can be found, which is illustrated in Figure 16.

Reference

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	-2									
C	-3									
G	-4									
C	-5									
A	-6									
A	-7									
T	-8									
G	-9									

Read

Figure 16: Hybrid Initialization

As previously mentioned, the reference sequence uses the Smith-Waterman algorithm, and therefore, initializes its side of the matrix with zeros. The short read sequence uses the Needleman-Wunsch algorithm, and therefore, requires a gap-score initialization scheme.

3.3 Computational Burden

Although the dynamic programming algorithm proves to be an accurate sequence comparator, the computation time limits its efficiency. For a human genome, a 3 billion bp reference must be aligned to 200 million, 76 bp reads. This computation runs at $O(NM)$ where N is the number of bases in a reference and M is the number of total read bases. The number of clock cycles for this computation in shown in Equation 1.

$$N = 3 \times 10^9\ bp$$

$$M = 200 \times 10^6 * 76\ bp = 15.2\ \times 10^9\ bp$$

$$O(N * M) = 3 \times 10^9 * 15.2 \times 10^9 = 45.6 \times 10^{18}\ cells$$

Equation 1: Dynamic Programming Cycles

This equation assumes that one base-to-base computation takes twenty clock cycles in a CPU because 5 additions, 5 comparisons, as well as about 5 branches are made for each score. As a rough estimate, for a system running on 2.27GHz, with 16 cores, it would take 800 years to reassemble a human genome, if we assume that the dynamic programming algorithm is the only running program, as shown in Equation 2.

$$t = \frac{20 * 45.6 \times 10^{18} cycles}{16 * 2.27 \times 10^9 cycles/sec}$$

$$= 2.52 \times 10^{10} sec = 800\ years$$

Equation 2: Dynamic Programming Computation Time

Because of the unreasonable amount of time it takes to do the computation, another algorithm was implemented into our design.

3.4 The BFAST Algorithm

Current software alignment programs use various algorithms to ease the computational costs of sequence alignment. We adopted the algorithm from BFAST [13], because it has proven to be an accurate, commonly-used software alignment program.

BFAST addresses the performance drawbacks of sequence alignment dynamic programming algorithms by using a premade index to quickly identify candidate alignment locations (CALs) [13]. CALs are locations in the reference where the read is most likely to match. An illustration of a read and a couple of its CALs is shown in Figure 17.

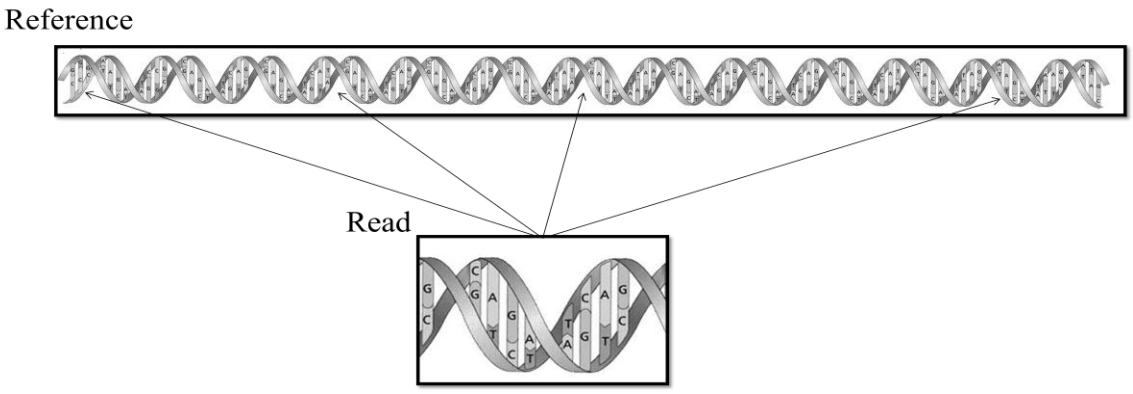

Figure 17: CALs of a Read

Once all possible CALs are found for a given short read, the dynamic programming algorithm runs on only those areas, which decreases the computation time significantly. The actual runtime of BFAST on the same 16 thread, 2.27 GHz machine is 6 hours, 38 minutes, and 15 seconds.

Table 3: BFAST Actual Runtime

Process	**Running Time**
BFAST's Algorithm + Dynamic Programming Algorithm	6 hours, 38 minutes and 15 seconds.
Dynamic Programming Algorithm alone	800 years

4. System Design

BFAST improves the alignment computation time of DNA reassembly by finding CALs and running the dynamic programming algorithm solely on these areas. Note that there are two main portions to the overall algorithm: 1) the Matcher, which finds the CALs, and 2) the Aligner, which runs the sequence alignment dynamic programming algorithm. This thesis primarily covers the Aligner, while [15] discusses the Matcher. Figure 18 illustrates the data flow between the Matcher and the Aligner.

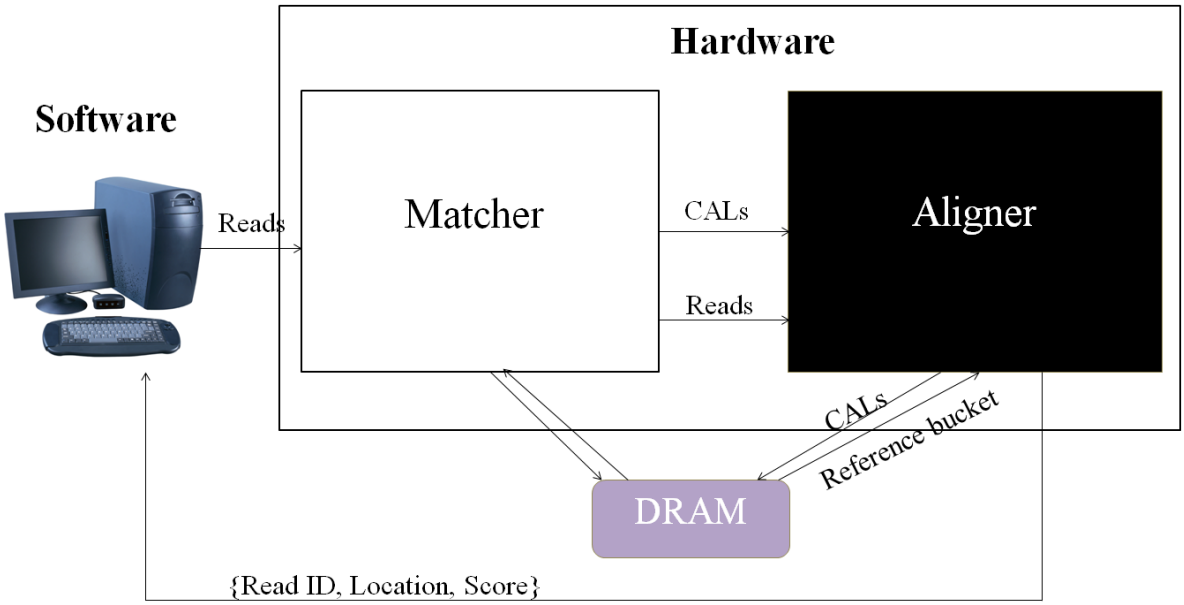

Figure 18: Overall System

The reference and other tables are precompiled in software, and then are stored in DRAM. The system begins when short read data is streamed into the hardware. The Matcher receives the reads first, and uses segments of the short read to look up CALs in memory. As the CALs are found, the Matcher passes them on to the Aligner. The Aligner is then able to score each CAL, and then pick the best alignment of the short read to the reference. The next sections discuss the various parts of the Matcher.

4.1 The Matcher

The Matcher's main purpose is to find all possible CALs for a short read. It accomplishes this task by using a precompiled index table of reference data, where short segments of the reference, called seeds, are used as the address. When a short read is streamed in, fixed length segments of the read are extracted and used as the address in the table. In this way, the short read and reference are quickly compared, and a simple look-up into the index obtains CALs for a read. The following sections describe these seeds and the look-up table.

4.1.1 Generating Seeds

The matcher obtains segments of the reference by sliding a window across the sequences. The mask and a segment of the reference or read performs a logical AND operator to extract seeds, as shown in Figure 19.

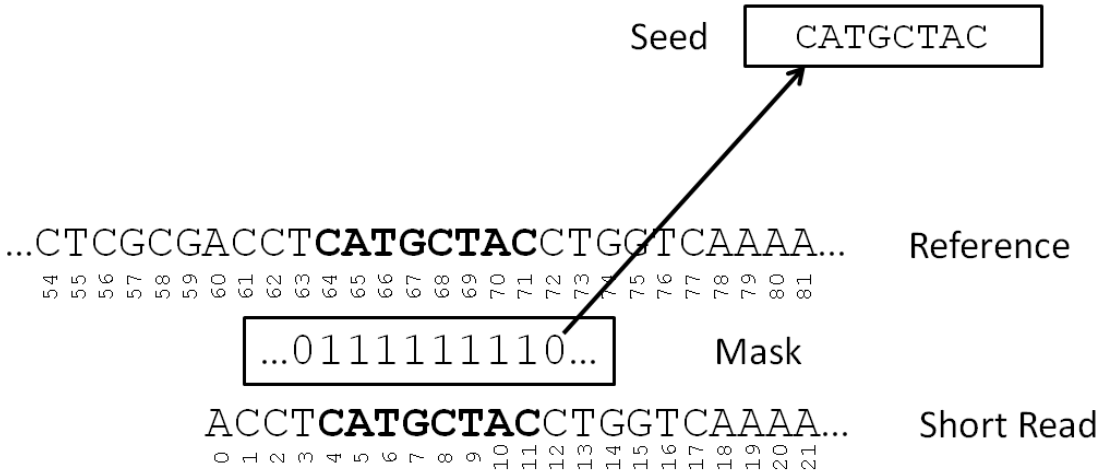

Figure 19: Generating Seeds

To create an index, the mask is slid across the entire reference sequence and obtains every pattern of nucleotide bases that exist in the reference. Our mask is composed of 22 ones, creating a seed length of 44 bits. The mask can contain 0's to create gaps in the seeds, but we

found the mask with all 1's sufficient for this application [13]. When processing short reads, the seeds are extracted and are used as the address in a pre-compiled look up table, the RIT.

4.1.2 The RIT

The Reference Index Table (RIT) stores every occurrence of a seed in the reference. To fill in the RIT, seeds are extracted from the reference sequence, and the locations of the seeds are recorded in the table.

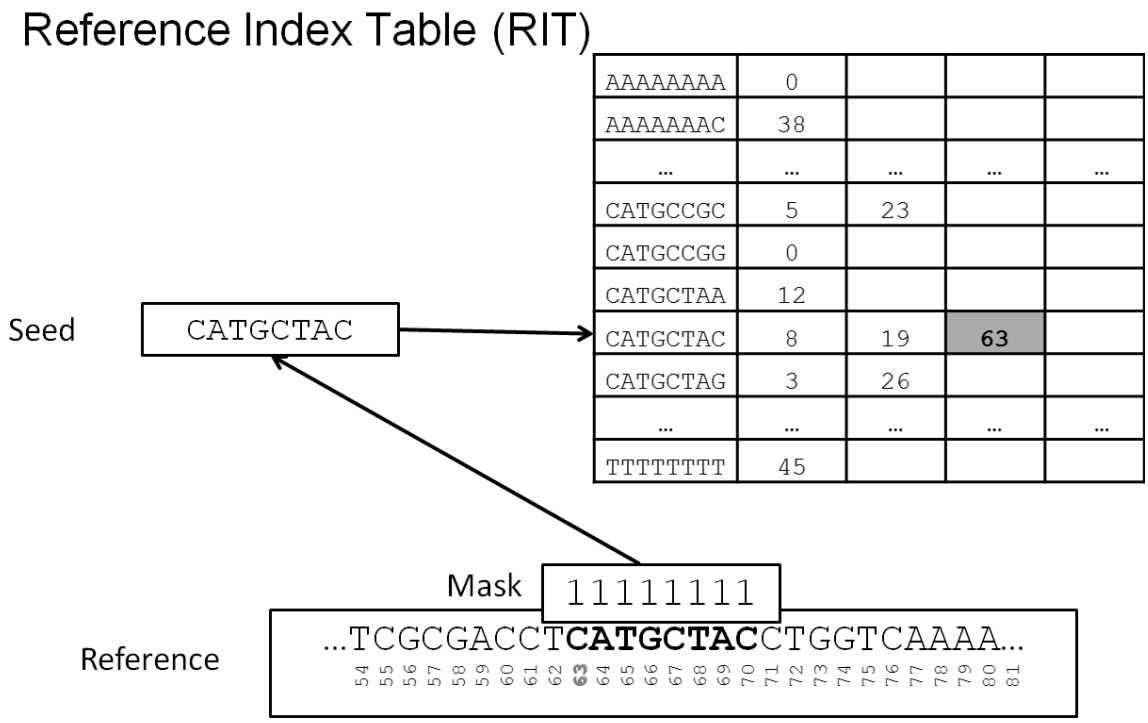

Figure 20: Creation of the RIT

In the example of Figure 2-, a seed, "CATGCTAC" is extracted from the reference at base 63. This seed is used as an address into the table, and location 63 is stored as data. This process continues along the entire reference to complete the RIT.

4.1.2 Using the RIT

During runtime, our system utilizes the prebuilt RIT to find CALs for each short read. The mask used for the reference sequence is utilized to extract seeds from the stream of short reads. Then, the system uses the generated seeds to look up the read's CALs, producing a quick CAL finder.

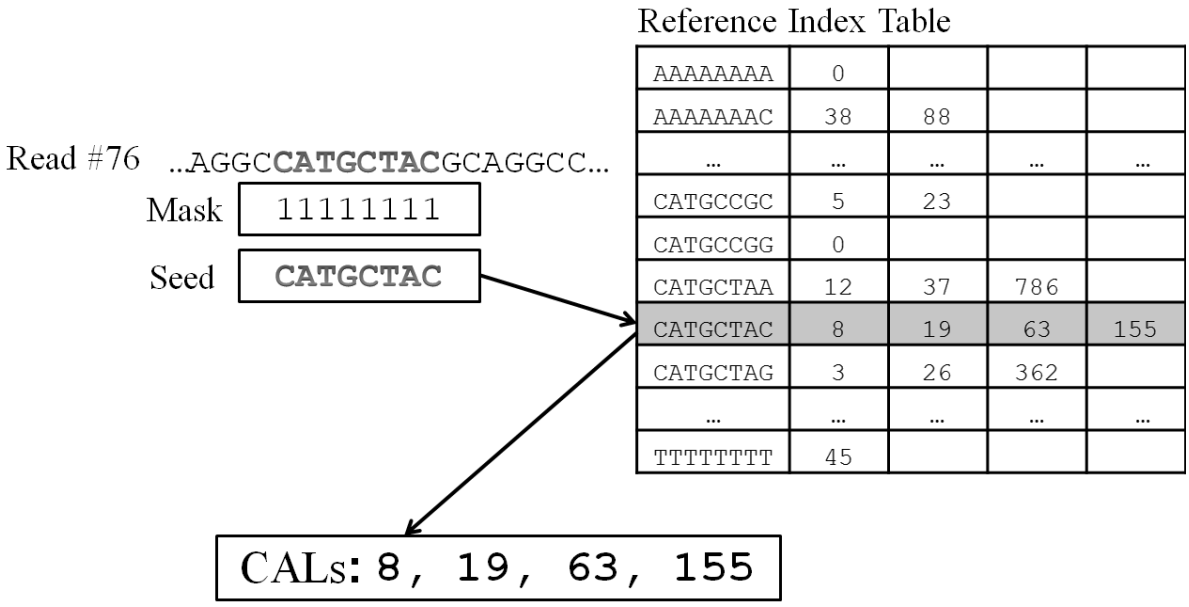

Figure 21: Using the RIT

In the example of Figure 21, the seed, "CATGCTAC" is extracted from Read #76. Read # 76 is the 76th read in the stream from the host software. The extracted seed is then used to look up the CALs: 8, 19, 63, and 155. In this manner, the RIT allows for a quick way to compare segments of the reference with the short reads.

4.1.3 Implementation in Hardware

In theory, the RIT is a single CAL table. However, in hardware, our system divides the RIT into two look up tables: the Pointer table and the CAL table. The Matcher utilizes the bits of the seed to look up into these two tables, as shown in Figure 22.

Figure 22: Seed Division

BFAST uses a default seed length of 22 bases, which is proven to be a reasonable length for sequence alignment [13]. Because a base requires 2 bits, the seed is composed of 44 bits. The seed is first hashed before it is subdivided into categories. More information about the hashing function can be found in [15]. The first 29 bits of the seed are the Address and the Tag, which are used for the Pointer table. The last 15 bits of the seed is called the Key, which is used in the CAL table. The two tables are shown in Figure 23.

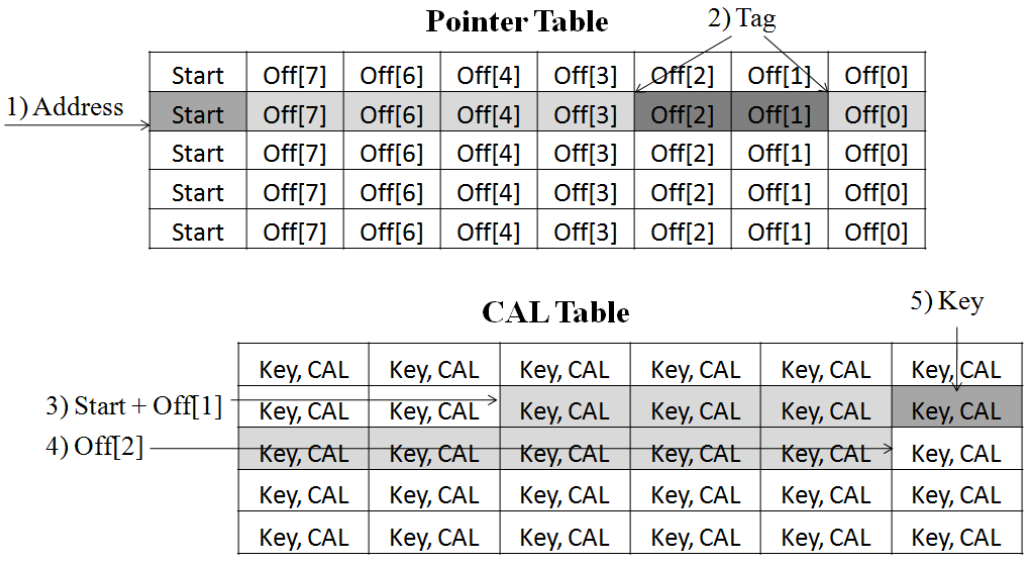

Figure 23: Hash Table and Index Table

The table look-up begins with the Address, which locates a row in the Pointer table. The first value in that row, Start, is saved for the CAL table look-up. Then, the Tag bits of the seed are used to find the correct offset values for the next step. These first two steps complete the Pointer table look-up.

The CAL table look-up begins with the addition of Start and Offset; this combined value identifies the beginning of an area in the CAL table. The end of that area is located using the second offset. Then, the Key bits of the seed are used to pass CALs that are associated with a matching Key within that area.

4.1.4 The CAL Filter

Because a sliding window is used to extract seeds, many seeds come from the same read and redundant CALs are inevitable. Therefore, our system incorporated a CAL Filter in the Matcher to improve the efficiency of the alignment computation.

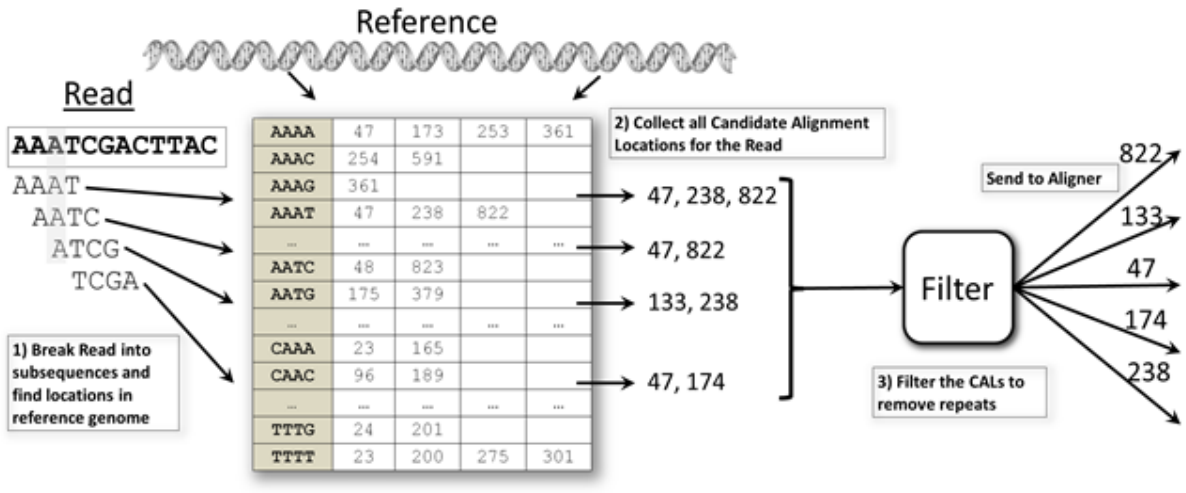

Figure 24: System with Filter

In Figure 24, the first step of the Matcher extracts seeds for the RIT. Then, the CALs for that read are sent to the input of the filter. The last step filters all the CALs for a read, so that each

CAL appears only once. Note that in this example, CALs 47, 238, and 822 show up multiple times before the filtering step.

4.1.4 CAL Finder and CAL Filter

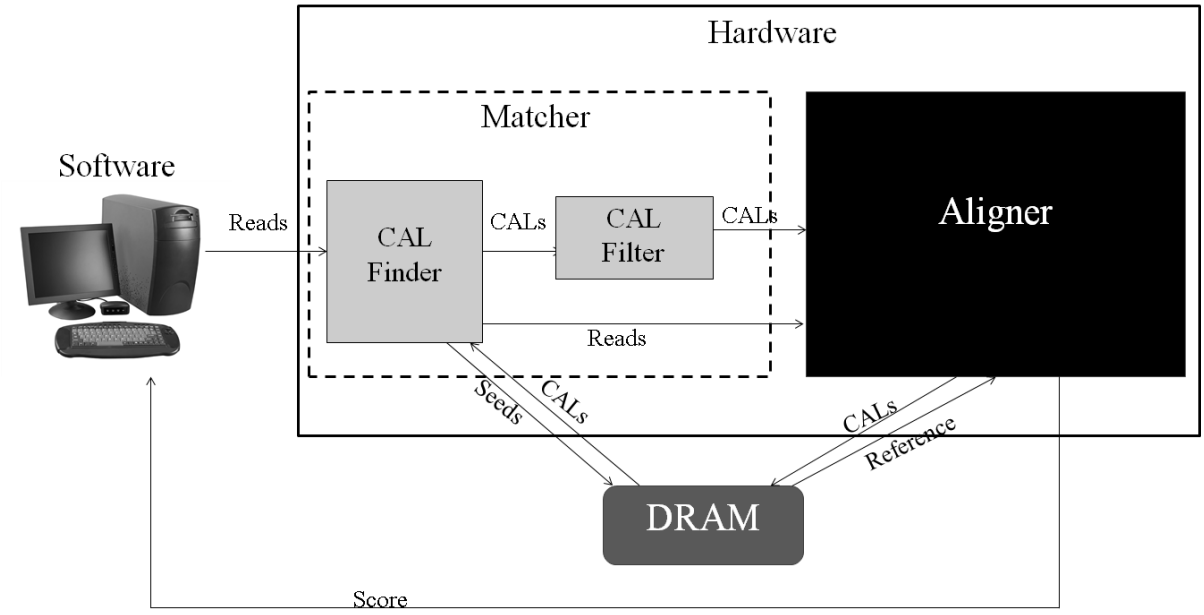

Figure 25: Data Flow of System with Matcher Highlighted

In Figure 25, the Matcher is broken down into the CAL Finder and the CAL Filter. The system starts running when reads are streamed from the software. The CAL Finder is the first component that receives reads. It extracts seeds to obtain CALs from the RIT in DRAM, and then sends them on to the CAL Filter. Once the CALs are filtered, they are forwarded to the Aligner, where they are used to acquire the reference from DRAM. With the reference sequence, the Aligner scores each CAL for a read, and outputs the Read ID, Location, and Score for the highest scored location.

4.2 The Aligner

The Aligner is the subsystem that compares the read and reference using the dynamic programming algorithm described in Chapter 3.1. It receives filtered CALs and short read sequences from the Matcher and finds the best CAL for a given read. Then, score information from the dynamic programming algorithm is sent back to the CPU.

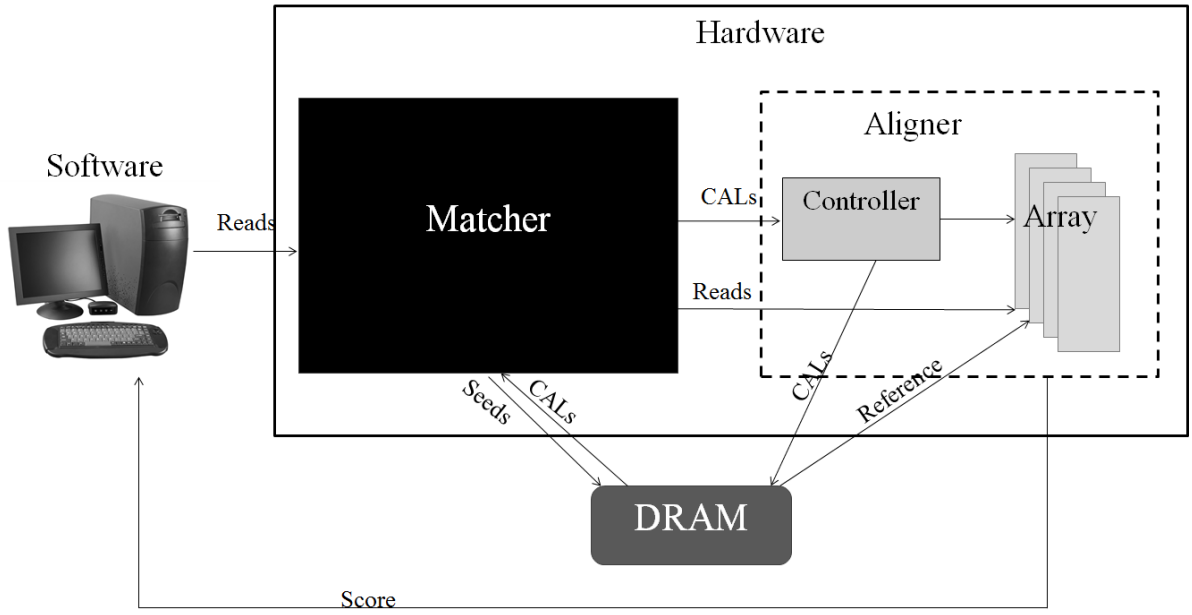

Figure 26: Data Flow of System with Aligner Highlighted

Figure 26 breaks down the Aligner into two main components: the Array and the Controller. The Array receives the read and reference sequences and outputs their alignment score using the dynamic programming algorithm. The dynamic programming algorithm is an independent process for each read and CAL pair, so multiple Arrays can run in parallel.

The task of the Controller is to handle the multiple array units and the memory accesses into the DRAM. The Controller uses a CAL, which is a numerical location value in the reference sequence, to obtain the nucleotide base sequences at that location from memory. The Array

and Controller work together to output the best alignment for a short read sequence in the reference. In the next section, we discuss the Aligner hardware.

5. Design Specifications

This chapter discusses the various modules of the Aligner subsystem and describes how each was implemented in hardware. There are five main sections in this chapter:

1) Top Level: interconnects the other four modules together and crosses clock domains.

2) RAM Controller: uses a CAL to read data from external DRAM.

3) Computation Array: finds the best score for a read and CAL pair using the sequence alignment dynamic programming algorithm.

4) Controller: handles the flow of data from the RAM Controller and the FIFOs to the multiple Computation Arrays.

5) Track: receives the output scores of the Computation Arrays and finds the best score for a read.

5.1 Top Level

The Top Level module instantiates the sub-modules of the Aligner, and wires signals together for flow control and data management. In addition, this module uses multiple asynchronous FIFOs to cross clock domains. This section will begin with a discussion of the various clocks and FIFOs in the Top Level module.

5.1.1 Clocks

There are three different clock speeds in the Aligner: the PicoClk, Ram_Clk, and the SWClk. These clocks, their uses, and their speeds are summarized in Table 4.

Table 4: Clocks

Clock Name	Use	Speed
PicoClk	Software/hardware communication, Matcher/Aligner communication	250 MHz
Ram_Clk	Pico's memory interface	267MHz
SWClk	Dynamic programming computation	125 MHz

The first clock listed in the table is the PicoClk, which runs the communication bus between the CPU and the FPGA, and between the Matcher and the Aligner. Similarly, the Ram_Clk is used to communicate with the external DRAM in Pico Computing's memory module. The SWClk runs the Computation Array and Controller, which runs on a slower clock due to the combinational logic involved in computing scores.

5.1.2 FIFOs

The Top Level module contains three FIFOs, including the Read FIFO, CAL FIFO, and the Output FIFO. The Read FIFO lies in between the CAL Finder and the Aligner to hold short read data. Similarly, the CAL FIFO sits in between the CAL Filter and the Aligner to store CAL data. Both the Read FIFO and the CAL FIFO cross from the PicoClk domain to the SWClk domain.

The last FIFO in the Top Level Module is called the Output FIFO, which collects the Score, Location, and ReadID for each short read. The Output FIFO crosses from the SWClk domain to the PicoClk domain in order to communicate with the software on the host processor. All FIFOs, clock boundaries, and sub-modules are illustrated in Figure 27.

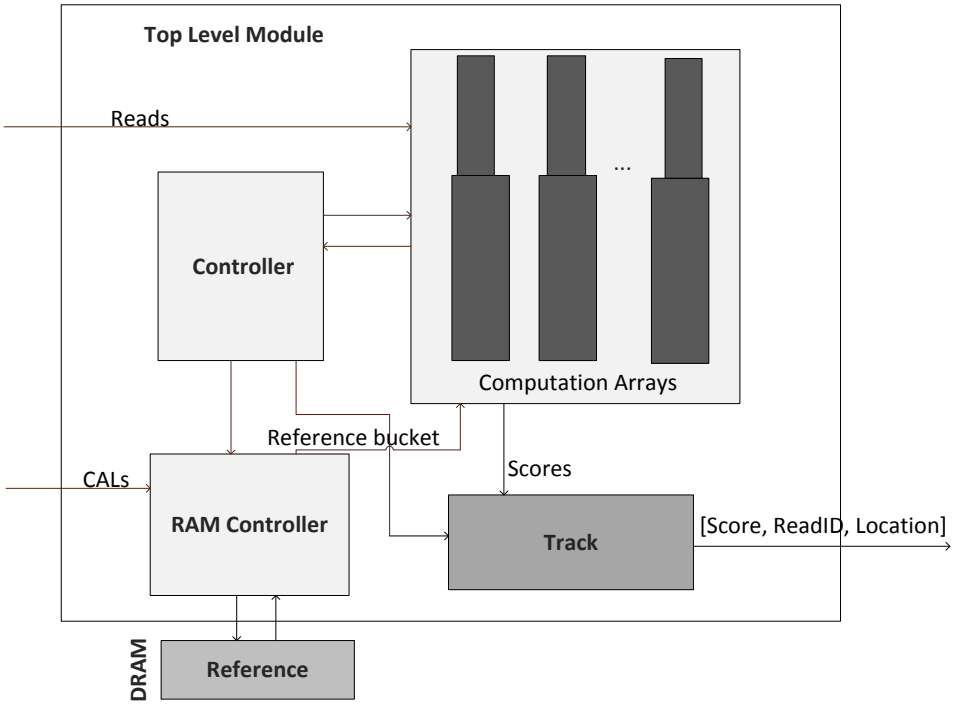

Figure 27: Top Level Module

5.2 RAM Controller

The Aligner uses an external memory to store reference sequence data. To read from the memory, the RAM Controller works with Pico Computing's memory module in addition to the Controller. It also contains internal FIFOs to cross between the SWClk domain and the Ram_Clk domain.

5.2.1 Memory Module

To obtain reference data from the DRAM, the RAM Controller uses the control logic illustrated in Figure 28.

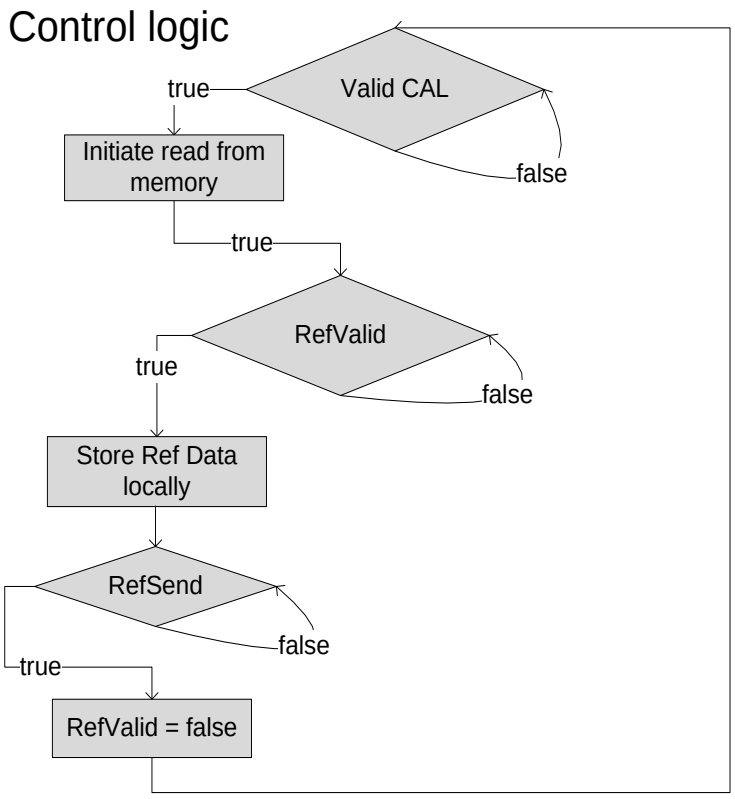

Figure 28: RAM Controller Control Logic

The RAM Controller's initial state begins when a valid CAL is available and ready to use. Once this occurs, the RAM Controller initiates a read from the DRAM uses a modified CAL as the address. When after approximately 40 cycles, when data is available, a "RefValid" signal is asserted by the RAM Controller to alert the Controller of valid reference data, which is stored locally in an asynchronous FIFO. As soon as the Controller is ready, it asserts RefSend to indicate the reference data was received, and then RefValid is deasserted by the RAM Controller.

5.2.2. DRAM Addressing

To read from the correct location in memory, several modifications must be made to the CAL. Pico Computing's memory module requires reading 32-bit addressable, 256-bit aligned

data. This implies that each 256-bit data is stored in addresses that increment by 8, as shown in Figure 29.

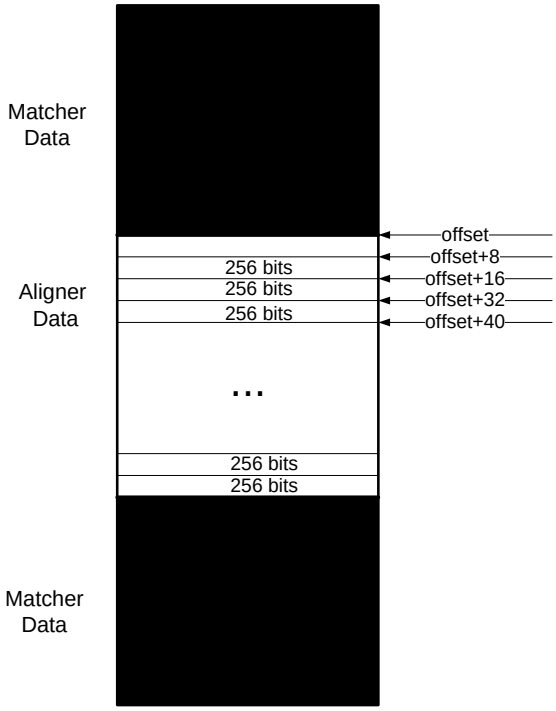

Figure 29: Reference in Memory

In addition, once a CAL is received from the Matcher, it must be converted from a base location to a bit location in the reference. The CAL value x represents the number of 2-bit bases from the start of the reference. Since the memory is word-addressed (32-bits) a division by 16, or a right shift by 4, is required to convert the CAL to an address in memory. The address modifications described in this section are summarized in Equation 3.

$$Addr = (CAL \gg 4) + MEM_{BASE}$$

Equation 3: Address Computation

In this equation, MEM_{BASE} is the base offset from the Matcher data at which the Aligner data is stored in memory.

5.2.3. Reference Sequence in Memory

Our current system uses 192 bases from the reference in the Computation Array. This is because our read base length is 76 bp, and the reference length must be at least 10bp longer on each side of the short read to handle indels; this requires a 96 bp reference. Pico Computing's memory module keeps data in 128-bit bursts in memory, but we must ensure that we get at least 96 bp (or 192 bits) for each CAL, no matter where it lies in memory. For example, in Figure 30, a CAL lies at the end of a memory burst. If we were to read just the first two bursts, the last 32 bp of the 96 bp reference chunk would not be loaded into the Aligner. Therefore, 3 bursts, or 192 bases, are needed for each reference in the Aligner.

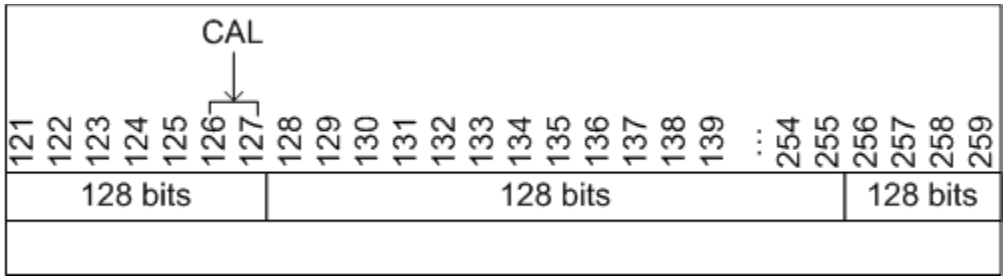

Figure 30: Aligned CAL in Memory

Because Pico Computing's memory module reads data in 128-bit, 256-bit aligned bursts, we must read 512 bits of data from memory for every reference sequence. Although we read 512-bits total, the reference only needs a 384-bit section of it. In addition, The Matcher sends a CAL as a 128-bit aligned location [15], so we must determine whether to take the MSB or the LSB 384-bits of the 512-bit data from memory. If the CAL is aligned to 256 bits, this indicates that its address is a multiple of $\log_2 256 = 8$, and we can take the LSB of the data, as shown in Table 4.

Table 4: Aligned Reference

Data Bits	[511:384]	[383:256]	[255:128]	[127:0]
Burst #	4	3	2	1

If the CAL is not a multiple of 8, the MSB data is taken instead, as shown in Table 5.

Table 5: Unaligned Reference

Data Bits	[511:384]	[383:256]	[255:128]	[127:0]
Burst #	4	3	2	1

5.3 Computation Array

The Computation Arrays are used to compute the dynamic programming algorithm score of the reference and read sequences. Recall that the score reveals where the read best aligns in the reference. In the array, the read sequence is stored in an internal register, while the reference sequence is shifted in each cycle by a Reference Shifter that feeds this array.

To replicate the behavior of the dynamic programming algorithm's score matrix, each read base meets with each reference base in the Computation Array, when the reference sequence is shifted in. This process is described in Figure 31. As soon as the entire reference sequence shifts through, the computation is finished for the two sequences, and a unit at the bottom of the Computation Array outputs the Score and Position of the alignment.

In addition to the dynamic programming score, various flow control signals are generated by the array. These signals interact with the Controller so that new sequences can be sent for computation. These features will be further discussed in the following sections.

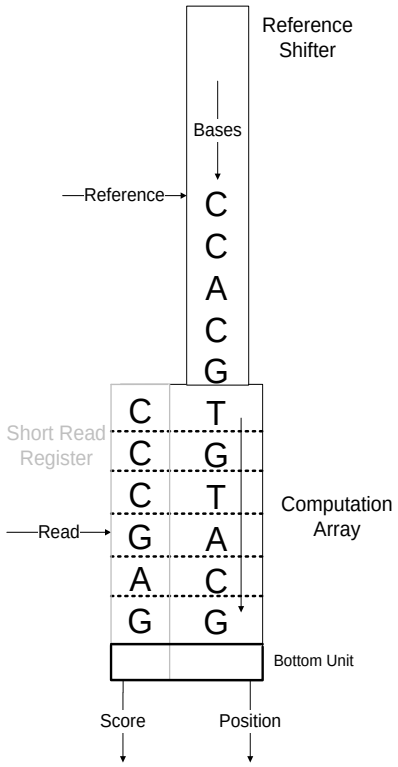

Figure 31: Computation Array

5.3.1 Reference Shifter

The Reference Shifter's purpose is to keep a reference sequence readily available for the Computation Array and convert from the parallel send of the reference from the memory to the serial stream the Computation Array needs. Because of this feature, the Computation Array has the ability to operate on two reference sequences at a time: one shifting out of the Reference Shifter and one shifting through the Computation Array. We chose to operate on two reference sequences at one time because there are multiple CALs for a majority of the short reads. This design choice creates a convenient and efficient way to pipeline the dynamic programming algorithm computation.

However, our design does not allow references from two different reads in a single Computation Array at one time. If we were to include this feature, the arrays could

continuously shift and compute, which would accelerate this part of the system. However, the Computation Array would also have two different reads simultaneously in the computation, and we would have to keep track of when and where to load a new read in, as shown in Figure 32.

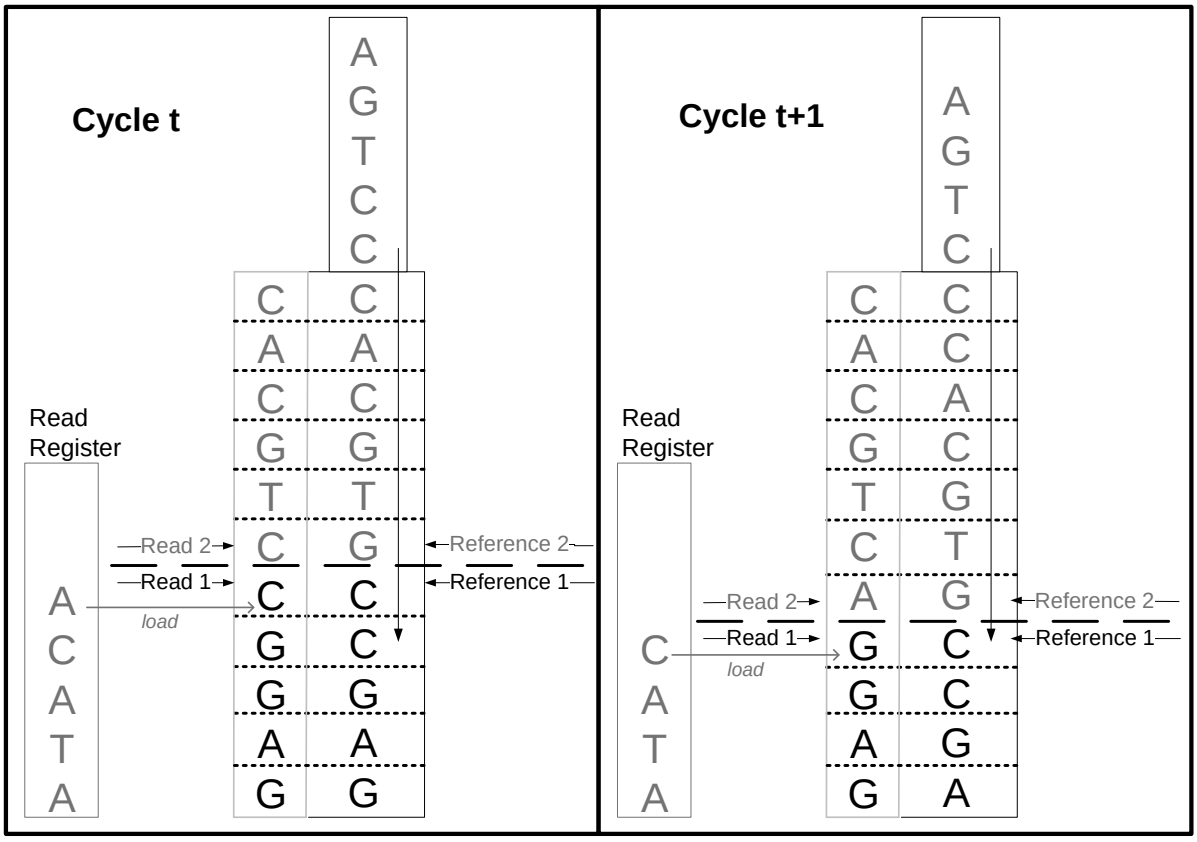

Figure 32: Multiple Reads

Cycle t is the clock cycle prior to Cycle t+1, and the change in the reference and read data is shown through these diagrams. A separate read register is required for this design, and the read in a cell would be loaded as soon as the previous read's computation is complete in this cell.

In addition, currently, the Computation Array is not the bottleneck of the system, so it is not necessary to accelerate this portion of the system. Shifting reference sequences from multiple reads would require more complex logic for no increase in efficiency of the overall system. Thus, for simplicity, this logic was not implemented our design.

5.3.2 Computation Units

The Computation Array is composed of multiple Computation Units, and there are as many units as there are bases in a short read sequence. Each unit represents an element in the score matrix and computes a score for one reference base and one read base each clock cycle. Three major steps are required to compute the score of one Computation Unit. First, each Computation Unit receives scores from neighboring units. Then, it uses those scores to compute its own score. The last step passes the Computation Unit's computed scores to its own neighboring elements.

The derived scores are from the following units in the score matrix: the Left unit, the Up (above) unit, and the Diag (above-left/diagonal) unit. Figure 33 displays how each of these five scores are derived.

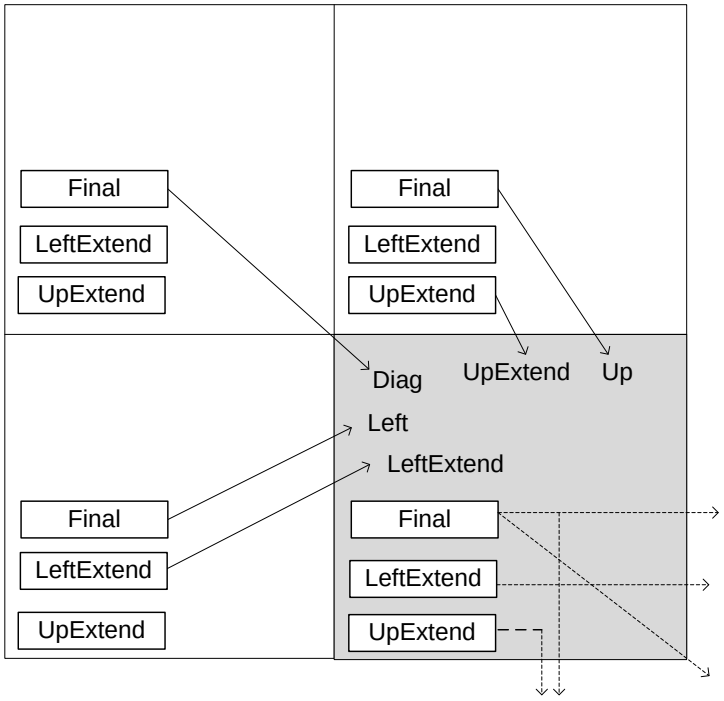

Figure 33: Section of the Score Matrix with Derived Scores

In this figure, the current unit is highlighted in gray, and its registers hold three values. These values include the Final score, the gap extended score from Up, and the gap extended score from the Left. The input scores are marked in solid arrows while the scores sent on by the current unit are marked in dotted arrows. There are five input scores total, two from each gap direction, and one from the diagonal.

The second step of the score finding process involves the computation of new scores by using the five derived values. This computation is explained in Table 6.

Table 6: Score Computation

Score Name	Description	Equation
LeftOpen	The open-gap score for a new gap from the left	LeftOpen = Left – OPEN_GAP[1]
LeftExtend	The extended-gap score for an extended gap from the left	LeftExtend = LeftExtend – EXTENDED_GAP[2]
UpOpen	The open-gap score for a new gap from above	UpOpen = Up – OPEN_GAP[1]
UpExtend	The extended-gap score for an extended gap from above	UpExtend = UpExtend – EXTENDED_GAP[2]
DiagScore	The diagonal score from the Diag unit with the match bonus or mismatch penalty	DiagScore= match? (Diag +MATCH_BONUS[3]): (Diag-MISMATCH_PENALTY[4])
In our design, the following values are assigned to penalties and bonuses: [1]OPEN_GAP = 2 [2]EXTENDED_GAP = 1 [3]MATCH_BONUS =2 [4]MISMATCH_PENALTY = 2		

The open gap scores are computed from the Final scores of neighboring elements, while the extended scores are computed from the extended scores of neighboring elements. In addition, DiagScore depends on the current unit's reference base and read base. If the bases match, a bonus score is given; otherwise, a penalty is taken away from the derived DiagScore.

The third step of the score computation involves finding the maximum of the computed values. In the sequence alignment dynamic programming algorithm we implemented, the best score is the highest score because mismatches and indels are penalties, while matches are bonuses. As a result, the best score is found from the maximum of the fives scores in Table 6, as shown in the segment of Verilog in Figure 34.

```
//Compute bonuses and penalties
always @(posedge clk) begin

       LeftExtend <= LeftExtendPrev - GAP_EXTEND_COST;
       LeftOpen <= Final - GAP_OPEN_COST;
       UpOpen <= Up - GAP_OPEN_COST;
       UpExtend <= UpExtendIn - GAP_EXTEND_COST;
       MatchScore <= Diag + MATCH_BONUS;
       MismatchScore <= Diag - MISMATCH_PENALTY;

end

//Final UpExtend score for unit, sent to bottom unit
assign UpExtendOut = (UpOpen < UpExtend)? UpExtend:UpOpen;

//Final LeftExtend score for unit, input to register
assign LeftExtendPrev = (LeftOpen < LeftExtend)? LeftExtend:LeftOpen;

//Maximum gap score
assign GapMax = (UpExtendOut < LeftExtendPrev)? LeftExtendPrev:UpExtendOut;

//Maximum match/mismatch score
assign DiagScore = (match) ? MatchScore:MismatchScore;

//Final Score
assign Final = (GapMax > DiagScore) ? GapMax:DiagScore;
```

Figure 34: Score Computation

In Figure 34, "GapMax" is the maximum of the gap extended values, LeftExtendPrev and UpExtendOut. The LeftExtendPrev and UpExtendOut scores take part in the computation of the final score, and are also saved in a register so that its values can be passed on to neighboring matrix elements. The Final score is found from the maximum of the score from DiagScore and GapMax.

In our system, the minimum positive score is -78, as shown in Equation 4.

$$-\big(\text{OPEN\_GAP} + (\text{READ\_LENGTH} \times \text{EXTENDED\_GAP})\big) =$$

$$-\big(2 + (76\text{bp} \times 1)\big) = -78$$

Equation 4: Minimum Score in Design

This would require 8 bits to implement a signed number. In addition, the maximum positive score is 152, as shown in Equation 5.

$$(\text{MATCH\_BONUS} \times \text{READ\_LENGTH}) =$$

$$2 \times 76 = 152$$

Equation 5: Maximum Score in Design

The maximum score in our design requires 9 bits for a signed number. Therefore, 9 bits was used for the scores in our system. The OPEN_GAP, EXTENDED_GAP, and MATCH_BONUS were listed in Table 6.

5.3.3 Score Matrix vs. Computation Array

The Computation Array utilizes the inherent parallelism in FPGAs by running multiple units simultaneously, as illustrated in Figure 35.

Reference

Read			G	C	C	C	T	A	G	C	G
		0	0	0	0	0	0	0	0	0	0
	G	-2	1	2	3	4	5	6	7	8	9
	C	-3	2	3	4	5	6	7	8	9	10
	G	-4	3	4	5	6	7	8	9	10	11
	C	-5	4	5	6	7	8	9	10	11	12
	A	-6	5	6	7	8	9	10	11	12	13
	A	-7	6	7	8	9	10	11	12	13	14
	T	-8	7	8	9	10	11	12	13	14	15
	G	-9	8	9	10	11	12	13	14	15	16

Figure 35: Clock Cycles in Computation Array

In Figure 35, the Computation Array starts at clock cycle 1, where the first reference base and the first read base use the dynamic programming algorithm to compute a score. At clock cycle 2, read base "G" is computed with reference base "C", while read base "C" is computed with reference base "G". In this way, several units are running at the same time in most clock cycles; only the first clock cycle and the last clock cycle have a single unit computing on its own.

Since the computation proceeds as a diagonal wave front through the dynamic programming array, we only need as many hardware units as the length of the longest diagonal. Since the read is shorter than the reference, we need as many units as the length of the read. Each unit is responsible for computing an entire row of the matrix, one cell at a time. This means that each unit receives one base of the read at the beginning of the computation, and the bases of the reference stream through the array, from the top to the bottom.

Because of this diagonal computation, the Left, LeftExtend, Up, and UpExtend of each unit is computed in the previous clock cycle, and Diag from two cycles ago. Left and LeftExtend come from the same unit, one cycle earlier. Up, UpExtend, and Diag come from the cell above the unit. The input score connections are shown in Figure 36.

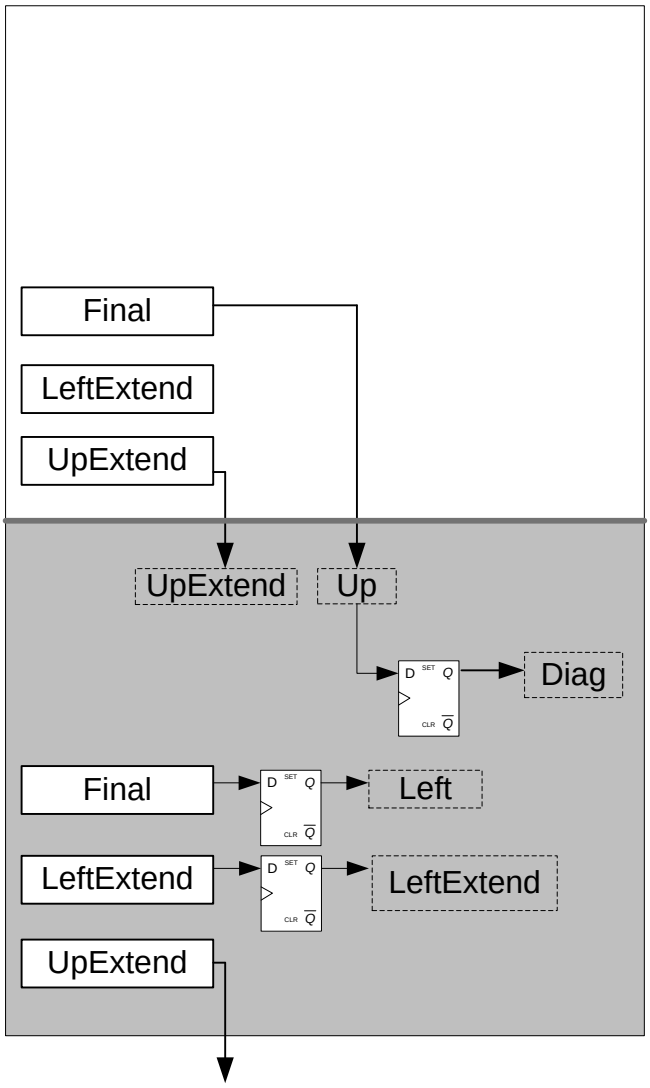

Figure 36: Inputs for a Computation Unit

Note that this figure does not display how each Final, LeftExtend, and UpExtend scores are computed, just how they are derived to be used for computation. Like the score matrix, the Computation Unit directly receives the Up and UpExtend scores from the unit above. On the

other hand, the Diag, Left, and LeftExtend scores are registered values because a clock cycle delay is equivalent to moving one unit to the left of the matrix (across the reference sequence). Therefore, the old Final value is used as Left, while the old LeftExtend value is used as the input LeftExtend value. In addition, the old Up value is used as the Diag value, which is equivalent to the above left value in the score matrix.

5.3.4 Bottom

Because the short read uses a global alignment dynamic programming algorithm, the highest score and position can be found by searching through the scores in the bottom row of the matrix [21]. The bottom unit of the Computation Array, shown in bold in Figure 31, receives scores from the last row of the matrix, computed by the last computation unit of the Computation Array. To find the best position and score of two sequences, the unit keeps track of the maximum score. Then, it holds onto the maximum score and position until the entire reference has shifted through the Computation Array. Once the end of the reference sequence arrives, the best score and location of that alignment is sent to the output.

5.3.5 Control Signals

Each Computation Array creates control signals for the Controller to initiate and end the computation process. Two primary signals are generated by the array in the flow control of this process: “RefReady” and “ReadReady”.

When the entire reference sequence is shifted out of the Reference Shifter, the RefReady signal is asserted. In this way, the RefReady signal implies that a new reference sequence can be loaded in the empty register, as long as it uses the same read. Recall that a new reference

is loaded only if a short read has multiple CALs to compute against, because only one short read is in the Computation Array at a time.

Conversely, the ReadReady signal implies that the Computation Array is available for a new short read. As soon as the reference sequence completely shifts out of the Computation Array, a Done bit is asserted by the bottom unit to indicate that the computation for one reference and read pair has been completed. If both the Done signal and the RefReady signal are asserted, the entire Computation Array, including the Reference Shifter, is empty. Once this occurs, the ReadReady signal is asserted.

From these observations, the last base of the reference sequence determines the generation of the RefReady and ReadReady signals. Figure 37 illustrates how the ready signals are generated in the system.

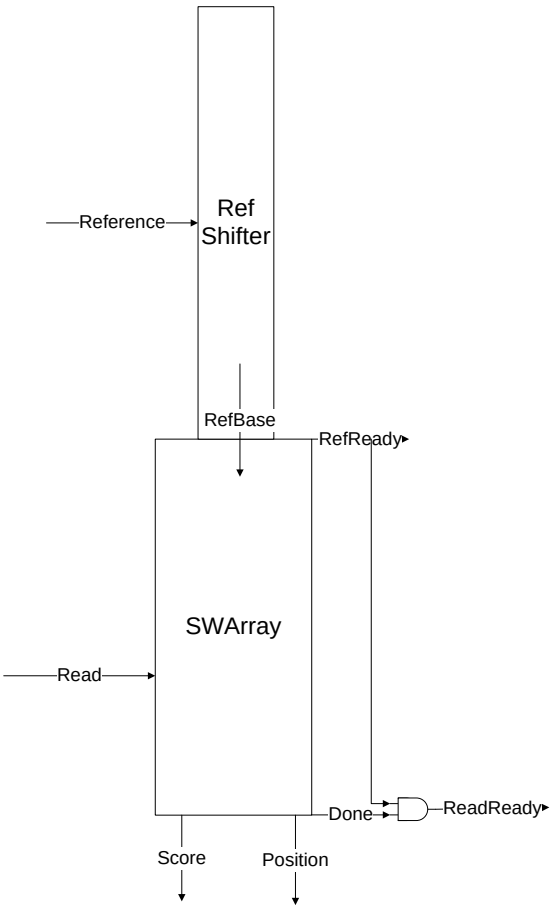

Figure 37: RefReady and ReadReady

RefReady is asserted as soon as the last reference base touches the end of the Reference Shifter, while the ReadReady signal is asserted as soon as the Done signal and the RefReady signal are true. These signals are then routed to the Controller, so that more sequence data can be received for computation.

5.4 Controller

The Controller handles the flow control between the various modules in the Aligner, and it has three main tasks. First, it reads from the CAL FIFO and Read FIFO, when data is available. Then, it initiates a read from DRAM using the CAL and waits for a valid reference sequence. The last step sends available read and reference signals to the multiple

Computation Arrays. Several strategies are used to accomplish these tasks, which will be described in the next few sections.

5.4.1 Target Computation Array

The multiple Computation Arrays are handled in a round robin fashion via a "target" counter. The target array receives the next available reference and read sequences as soon as it is ready for a new computation. Once the array receives data, the target value is incremented and the Controller moves on to the next Computation Array. This cycle continues as the Controller waits for available data and ready signals from the target array.

5.4.2 Sending Reads

For each computation, the CAL FIFO is dequeued and a reference sequence is sent to an available array. However, because there can be multiple CALs for a read, the short read sequence may not need to be sent to a Computation Array for every computation. Thus, the Controller must keep track of when to dequeue a read sequence from the Read FIFO, as well as when to send a new read sequence to a Computation Array. Note that these two tasks are not equivalent. A read is dequeued from the Read FIFO only once per read. We send a read to a Computation Array, depending on whether the Computation unit has received that short read previously. If only one Computation Array existed, then a read would be sent once each time a dequeue to the Read FIFO occurred.

To differentiate between a dequeue of a read versus a send, the Controller determines when to dequeue from the Read FIFO by using a "FirstCAL" bit from the CAL FIFO. The FirstCAL signal indicates that the next CAL is the first CAL for the new read; the FirstCAL

bit does not come with a CAL. Thus, the Controller must dequeue a new read when it sees an asserted FirstCAL bit.

A “FirstRound” bit is kept for each existing Computation Array, so that the Controller knows when to send a new read. An asserted FirstRound bit implies that a Computation Array has not received the current read yet. Therefore, for each computation, if the FirstRound bit is true for a Computation Array, a new read sequence is sent. Figure 38 illustrates these various signals in the Aligner.

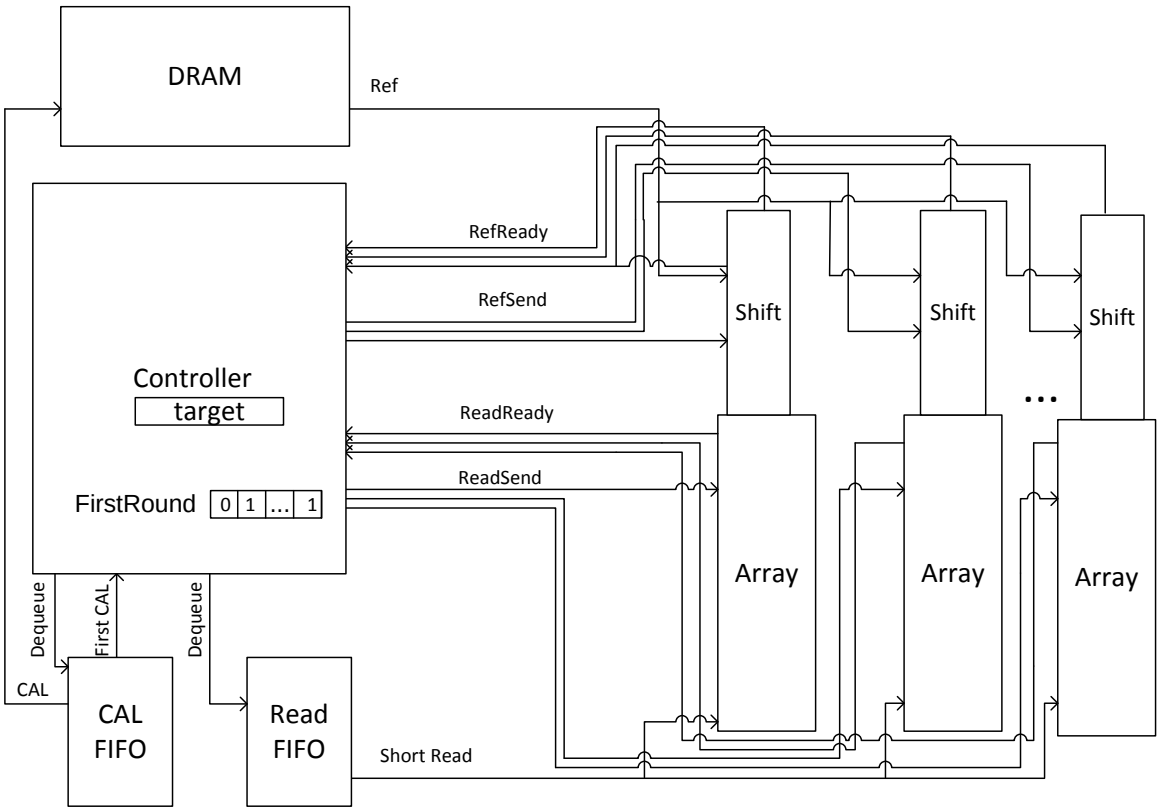

Figure 38: Controller Diagram

5.4.3 Controller Flow Chart

In Figure 39, the system flow chart begins with the CAL FIFO, which is dequeued as soon as data is available. This data contains a FirstCAL bit to indicate whether the ReadFIFO should

be advanced. If FirstCAL is true, the ReadFIFO is dequeued and all the First Round bits are asserted.

The next step checks the FirstRound signal, and if this signal is asserted for the target Computation Array, ReadReady is waited upon. Once the target Computation Array asserts its ReadReady signal, the read sequence is sent to a Computational Array. If FirstRound was not asserted, the RefReady signal is waited upon instead.

When the appropriate ready signal and RefValid is asserted, the Controller sends a reference sequence to the target Computation Array. Once this takes place, the cycle is completed and the target is updated for the next computation cycle.

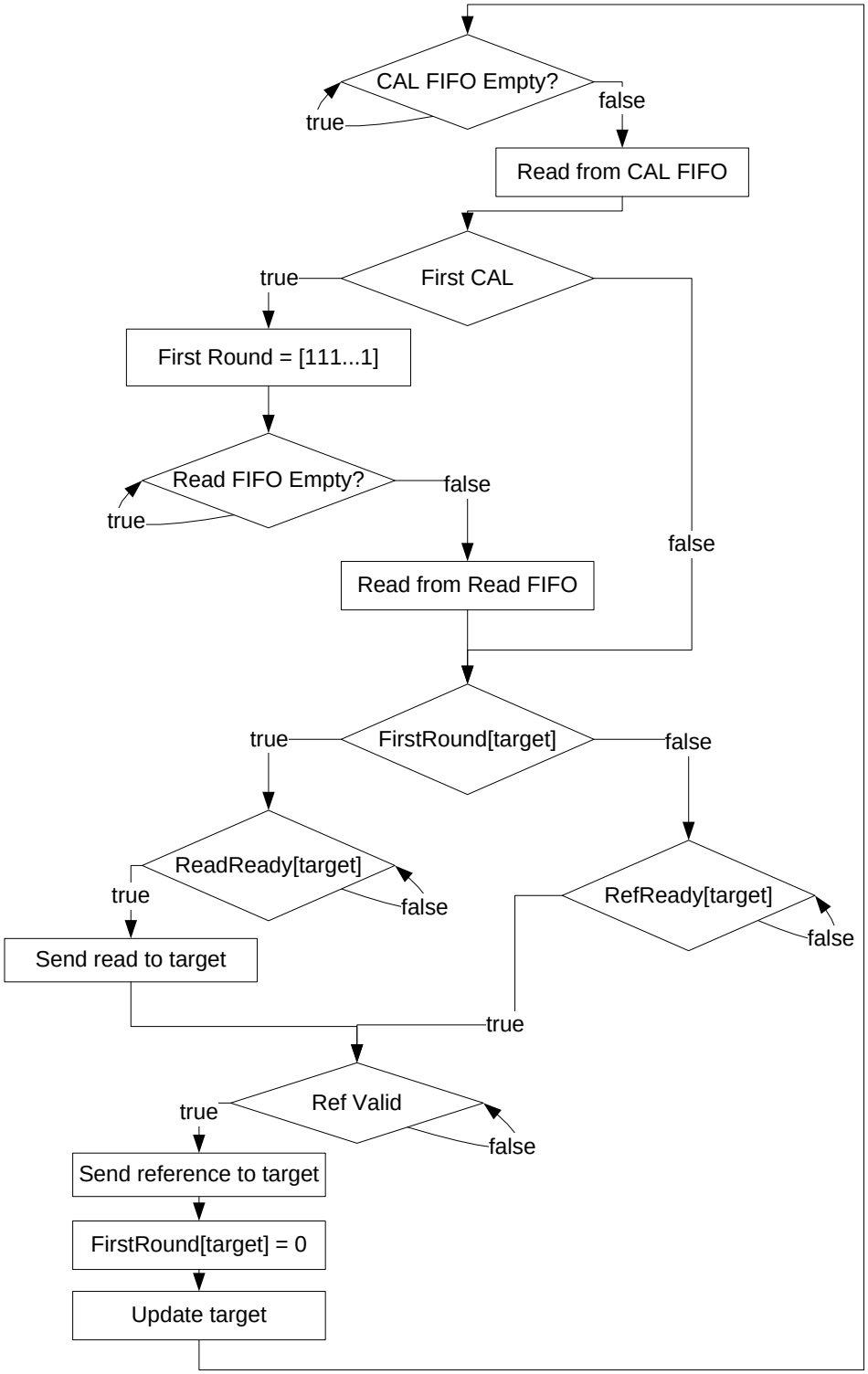

Figure 39: Controller Flow Control

The Controller uses the Computation Arrays via round robin because of simplicity, and because this design choice would not affect the computation time. In the current system, the Computation Arrays are not the bottleneck. Therefore, the next array would be waiting for the reference sequence from memory, and the next available Computation Array would be ready for the computation as soon as reference data is available.

5.5 Track

The Track module collects scores and positions from the Computation Arrays. It uses these values to find the best CAL for a given read. Once this is found, the Read ID and its best Position and Score are sent to the Output FIFO. The data flow and control flow for the Track module is shown in Figure 40.

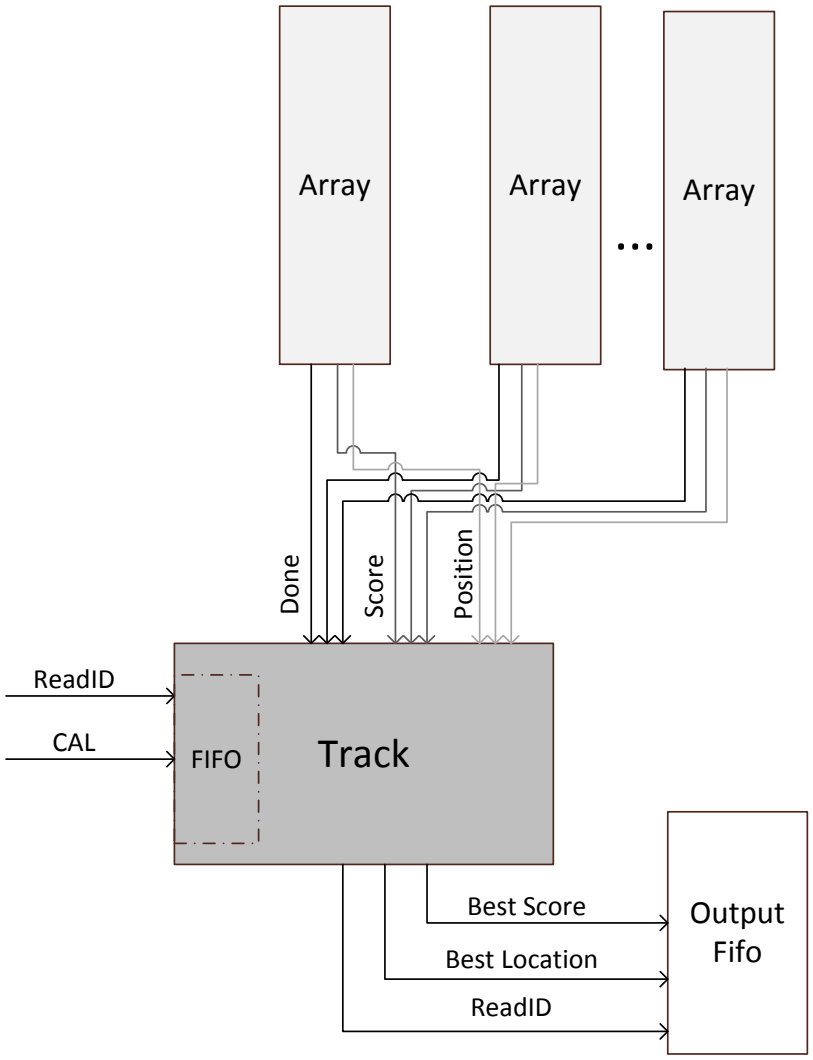

Figure 40: Track Diagram

5.5.1 Inputs

The CAL FIFO and Read FIFO send CALs and Read IDs to Track, which are stored in an internal FIFO. Track uses the CALs to find the best location of an alignment by combining the relative position output from the Computation Array with the CAL. The Position from the Computation Array is a location relative to the CAL, while a CAL is a place along the reference. The addition of these two values will give the exact base location of an alignment, as shown in Figure 41.

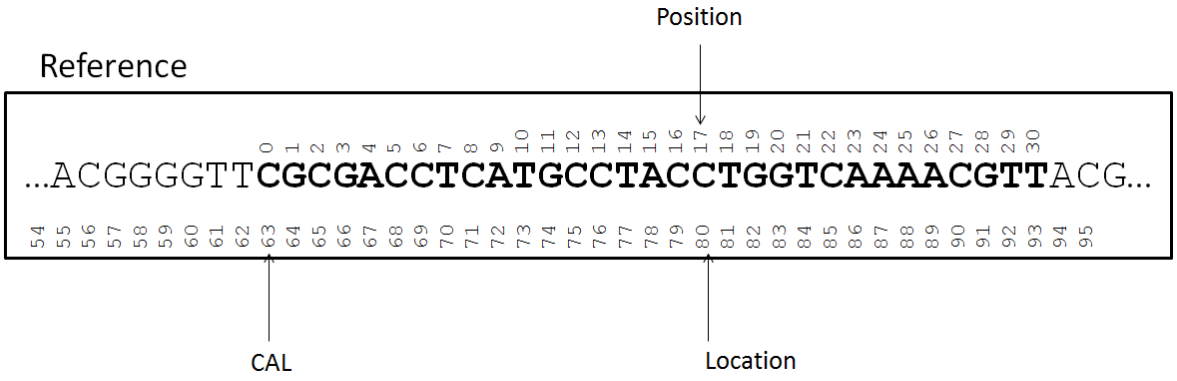

Figure 41: CAL, Position, and Location

5.5.2 Generating Outputs

Every time an array completes a computation, Track updates the highest score, along with the Position and ReadID. When the FirstCAL bit is seen, a new read is in the system, and the Score, Position, and ReadID of the old read are sent to the Output FIFO. This step finalizes the life cycle of a CAL and read pair through the Aligner. Figure 42 summarizes this chapter with a functional diagram.

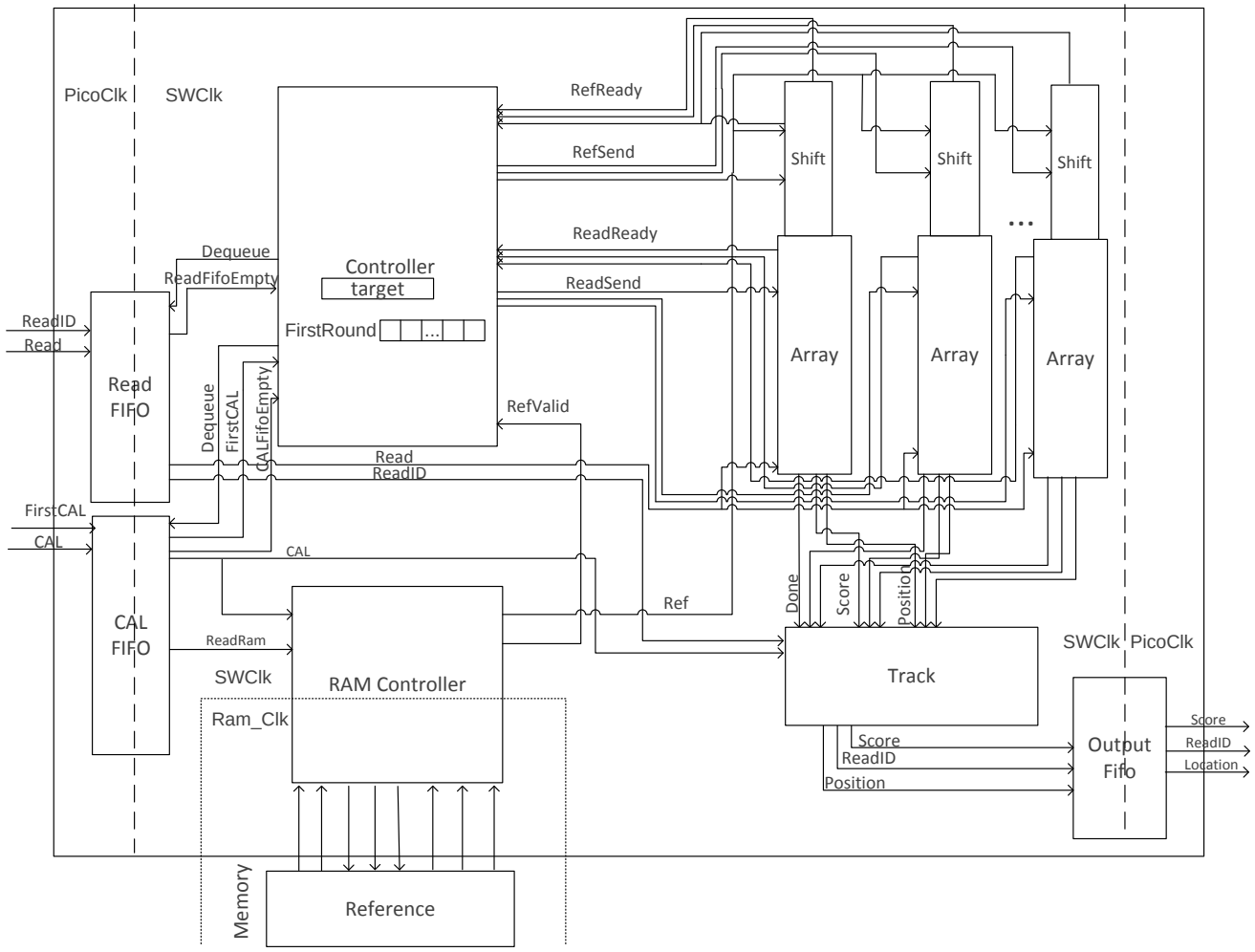

Figure 42: Aligner Modules

6. Results

6.1 Score Computation

As mentioned in the last chapter, the Computation Array uses a slow clock for the scoring process. In order to increase the clock frequency, a design was implemented in hardware for speculation. This design evaluates all the diagonal, open-gap, and extended gap scores at once by evaluating $\binom{n}{k} = \binom{5}{2} = 10$ comparisons simultaneously, as shown in Figure 43.

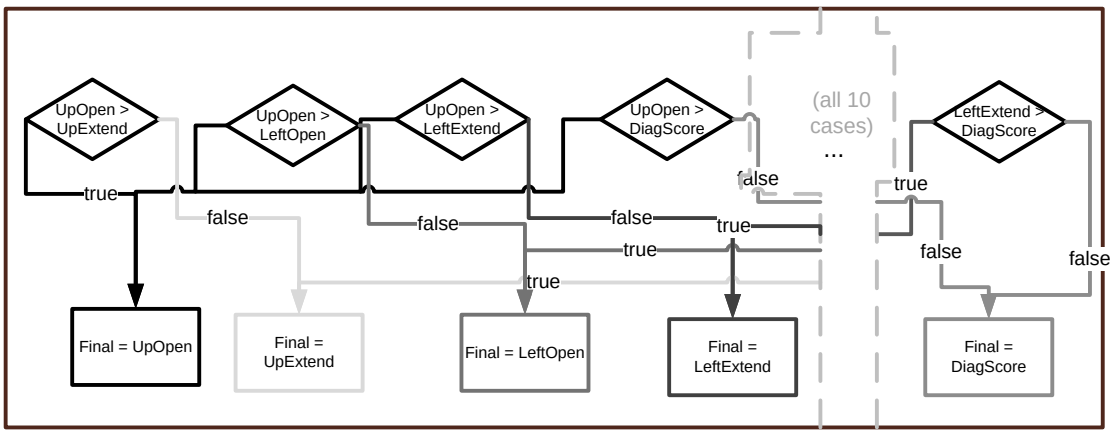

Figure 43: Computation Array II

As we expected, this version of the Computation Array was able to use a faster clock because it avoids doing comparisons in series, reducing the levels of logic. The period of the clock and the levels of logic of both systems are shown in Table 7.

Table 7: Computation Array II, Speed

Computation Array I	
Data Path Delay	6.879ns
Logic Levels	12
Computation Array II	
Data Path Delay	5.215ns
Logic Levels	7

Although a faster clock is one of the signs of an efficient system, that is not always the case. In hardware, a smaller system provides the ability to have more parallelism. This observation is true because more Computation Array units are able to fit in one FPGA. Thus, both the clock rate and the number of units must be computed to evaluate throughput. Table 8 lists the sizes of each Computation Array and the number of arrays that are able to fit in one FPGA, if it used strictly for Computation Arrays.

Table 8: Computation Array II, Area

Computation Array I					
		Slice Registers	**Slice LUTs**	**Block RAM/FIFO**	**Slices**
	Number of	4,414/301,440	10,044/150,720	0/416	2,830/37,680
	Percentage	1%	6%	0%	7%
	Number of Units per chip	68 units	15 units	infinity	13 units
Computation Array II					
		Slice Registers	**Slice LUTs**	**Block RAM/FIFO**	**Slices**
	Number of	4,413/301,440	13,073/150,720	0/416	4,066/37,680
	Percentage	1%	8%	0%	10%
	Number of Units per chip	68 units	11 units	infinity	9 units

In both versions of the Computation Array, the slices are the limiting factor because it restricts the number of units that are able to fit in each chip. Therefore, the throughput is calculated using the number of slices and the period of the Computation Arrays, as shown in Table 9.

Table 9: Computation Arrays, Throughput in Alignments/slices-ns

	Computation Array I	Computation Array II
Number of Slices	2884	4120
Period (ns)	6.879	5.215
Throughput (alignments/slices-ns)	5.04×10^4	4.65×10^4

From this table, we conclude that Computation Array I provides the highest throughput and is therefore, the most efficient design for our system, despite its lower clock speed. As a result, we kept the original computing algorithm of the system.

6.2. System Runtime

Our system was only able to fit the RIT and reference sequence of one chromosome in the human genome due to the memory limitations of the M-501. Figure 44 shows the system runtime on Chromosome 21, which is one of the 23 chromosome pairs in the human genome.

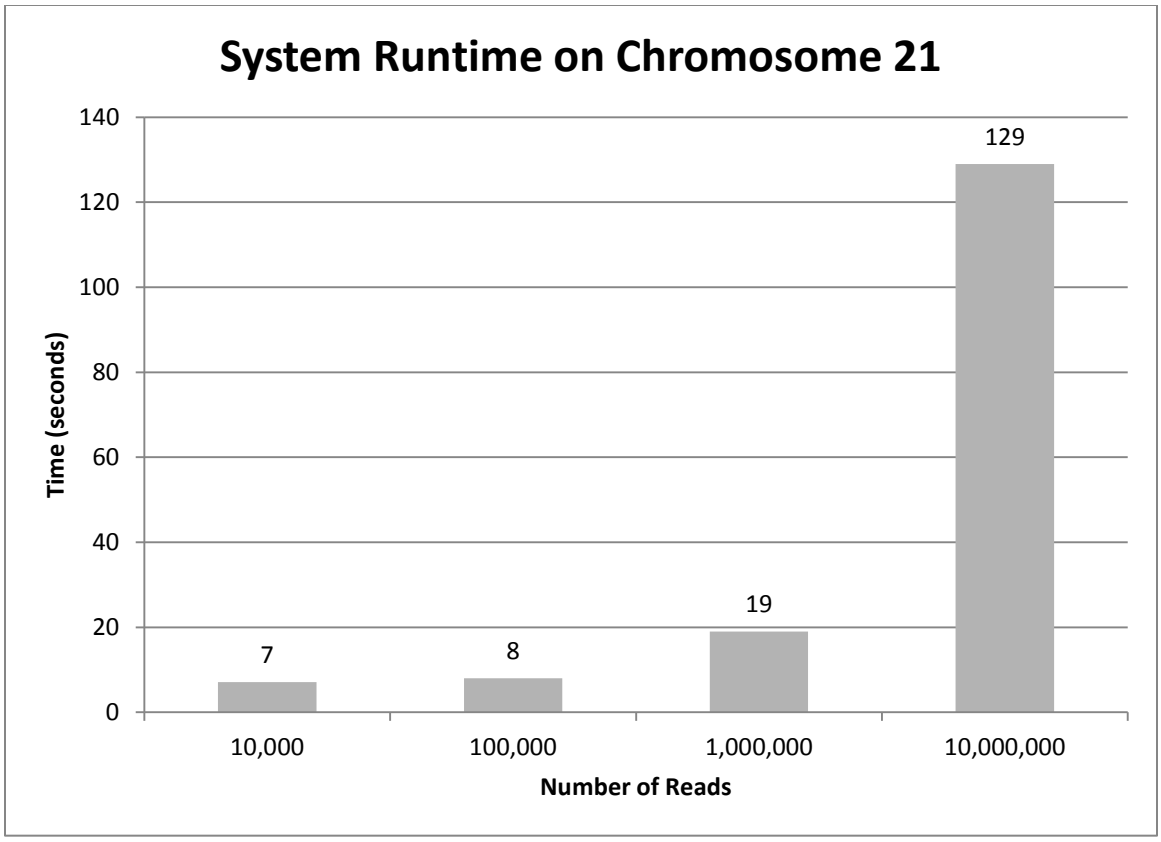

Figure 44: System Runtime on Chromosome 21

Corey Olson ran the system and included the load time of the DRAM, with a single Computation Array. The load time was included because BFAST includes its load time in the runtime, and we want to compare our results with it.

Running the system on Chromosome 21 proves that our system is applicable and accurate. However, because the reference sequence was generated using only one chromosome, less CALs are generated by the Matcher overall. This means that there are less dynamic programming algorithm alignments compared to a system with the entire human genome as the reference. Therefore, the runtime for the full human genome system is expected to be greater than the one gathered in this section.

We also ran BFAST on a 2.27 GHz, 16-core machine to compare against our results, as shown in Table 10.

Table 10: System in Hardware vs. BFAST

	System Runtime on 10,000,000 Reads
Hardware	122 seconds
BFAST	1,918 seconds

For Chromosome 21, our design is 15.7 times more efficient than the BFAST software program. We expect a more dramatic improvement in runtime on the full human genome, because of the efficiency of our dynamic programming algorithm. The human genome will require more computations in the aligning system, and this is where our system is most efficient.

6.3. Aligner Runtime

To analyze the performance of the Aligner system, we ran the program, without software streams or Matcher computations to interfere with the time measurements. The Aligner includes all the modules in Chapter 5.

We measured this runtime by generating reads and CALs in hardware, so that the PCIe stream time would not affect the accuracy of this time measurement. The range of the runtime was found for two different types of data sets on the system. The first type of data is one in which there is a single short read for all CALs. The second type of data is one in which every read has only one CAL. These two sets will give the full range of possible runtimes for the system. This is because the first data type is the most efficient that could occur in the system, while the latter is the most inefficient type of data for our

system; this is due to the way the Reference Shifter preloads reference sequences for the same short read, but does not do so for two different reads.

In this setup, we determined how multiple Computation Arrays affect the runtime of the Aligner system. Then, we compared this runtime to the overall system to analyze the possibility of the Aligner being the bottleneck.

6.3.1. Multiple Computation Arrays

To compare the effects of multiple Computation Arrays, we varied the number of Computation Arrays and measured the runtime of each system using the setup described in 6.3., with a large set of CALs. To collect accurate data, the design was run on a large time scale and consisted of approximately 1 billion alignments. The results from this experiment are graphed in Figure 45.

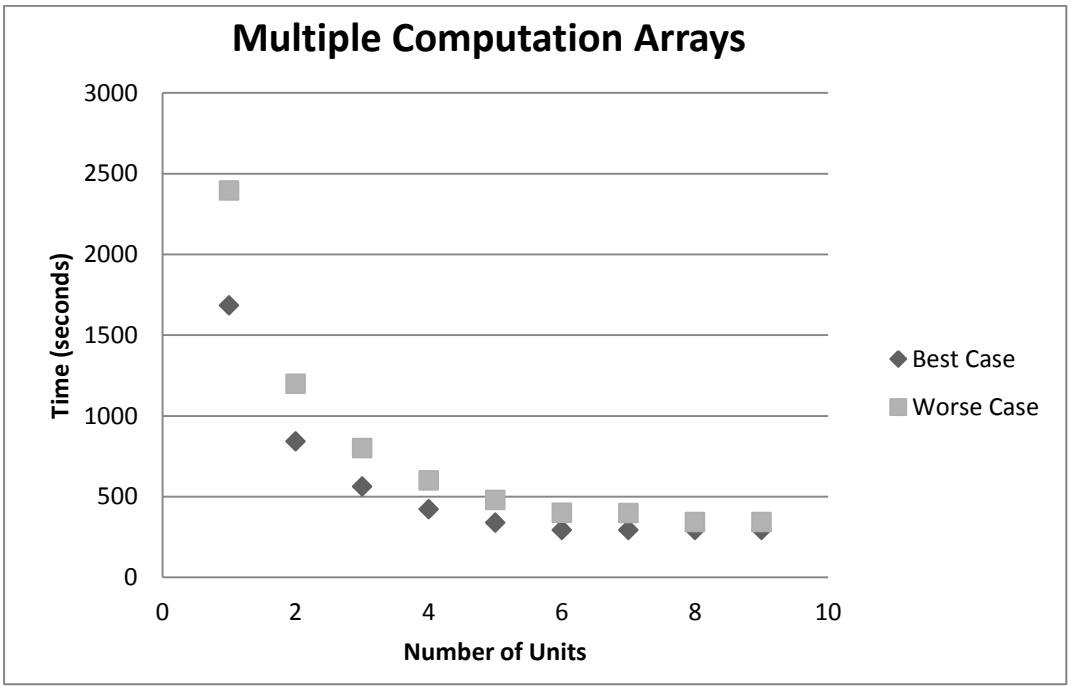

Figure 45: Graph of Multiple Computation Arrays

The runtime of the Aligner decreases with each additional Computation Array, but starts to level out at 6-8 arrays due to the memory bottleneck. Once the graph saturates, the arrays begin to stall as they wait for reference data. These results show the effectiveness of multiple Computation Arrays, as well as the limitations of the memory system.

6.3.2. Bottleneck

The Aligner becomes the bottleneck of the system if its throughput is less than the throughput of the entire system. Recall that this system generated CALs and short reads in hardware to negate the effects of the PCIe stream. This setup (described in 6.3.) resulted in a 28-40 minute runtime range for an Aligner with one Computation Array, as shown in Table 11.

Table 11: Bottleneck Runtimes

Type of data (2^{30}CALs)	Runtime (seconds)	Runtime
1 Read in system	1,684	28 minutes, 4 seconds
1 CAL per read	2,396	39 minutes, 56 seconds

The bottleneck runtimes allow us to estimate the throughput of the Aligner versus the entire system with one Computation Array, which was found in section 6.2. on Chromosome 21. This comparison is shown in Table 12.

Table 12: System vs. Aligner Throughputs

	Throughput (thousands of CALs per second)
Aligner	446-635
System	123

Table 12 shows that the throughput of the Aligner is greater than the system. The number of CALs for the system on Chromosome 21 was found to be 1,471,016 for 1,000,000 reads. This indicates that the Aligner is not the bottleneck of the system.

6.4. Resource Utilization

The system resource utilization, for the entire system, is plotted in Figure 46.

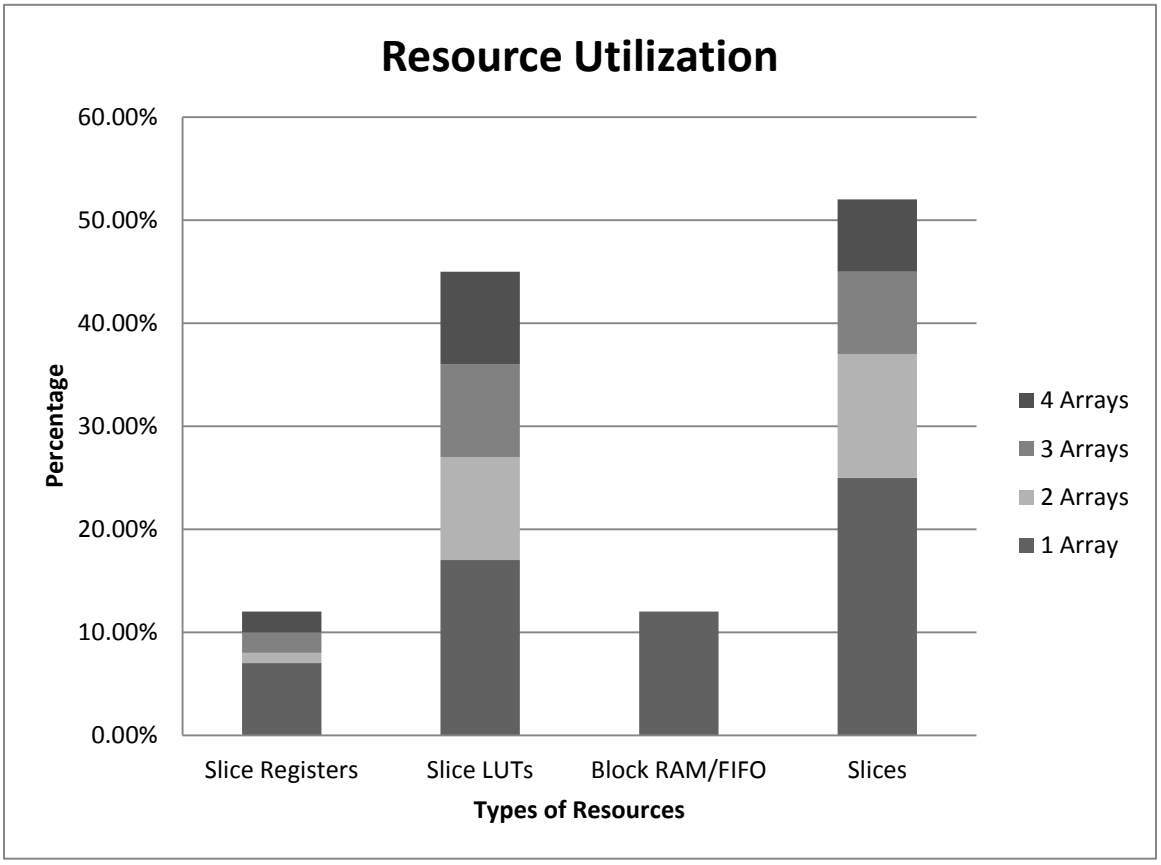

Figure 46: Resource Utilization in the Aligner

The limiting resource in our system is the slices, since it has the largest percentage utilization of the four resources. Each additional Computation Array increases in slices by about 9%. This means that if the system were not limited by memory accesses, our current system would be able to instantiate nine Computation Arrays for the best performance.

7. Conclusions

7.1. Conclusions

Our team designed a hardware system to align short read sequences for DNA reassembly and proved that it was achievable. We implemented the BFAST algorithm in FPGA hardware to accelerate the process by over 15 times. In addition, we expect this number to increase with the use of the entire human genome as the reference sequence. This will require a compute board with larger DRAM capacity.

The FPGA demonstrated to be valuable in balancing design flexibility with computation speed, and allowed massively parallel comparison of genomic sequences. With our new program, we anticipate more applications and advantages to arise from the field of DNA sequencing.

7.2. Future Design Considerations

This thesis has shown that the reassembly hardware accelerator is a useful tool due to the significant decrease in computation time. However, several additions must be incorporated in order to fully utilize the system for human genome mapping. In addition, other considerations are made in the following sections to improve upon the system.

7.2.1. Transferring to the M-503 Boards

Although the system was able to run on Chromosome 21, due to the limitations of memory size in our M-501s the reassembly process on the entire human genome was not processed. In addition to the M-501 boards, Pico Computing provides an M-503 with the

same Xilinx Virtex-6 LX240T. The M-503 also contains x8 Gen2 PCIe and two 4GB DDR3 SODIMM.

Because each M-503 board contains a total of 8GB of external memory, we would be able to fit our 30 GB human genome system into four boards. The transition from an M-501 to the M-503 is expected to be fairly smooth, and with the new platform, the entire human genome will be able to run in one system.

In addition, the M-503 has two different ports to external memory. This means that the memory bandwidth will increase by a factor of two, allowing more instantiations of Computation Arrays. We can estimate the approximate number of Computation Arrays needed to make full use of the memory system from Equation 6.

$$CA = \frac{sw\_compute\_time}{ref\_mem\_load\_time} = \frac{\#compute\_cycles * sw\_clk\_period}{\left(\frac{\#mem\_cycles * mem\_clk\_period}{mem\_ports}\right)}$$

$$= \frac{(192 + 76) * (1/125MHz)}{\left(\frac{40 * (1/267MHz)}{mem\_ports}\right)} = \frac{14.3}{mem\_ports}$$

Equation 6: Saturation of Computation Arrays

If we have a large number of Computation Arrays compared to the number of CALs per read, then we can approximate a Computation Array's execution time by the time it takes to finish a complete ref – read comparison. Since the references are 192 bases, and the reads 76 bases, it takes 268 clock cycles. Recall that memory loads take approximately 40 cycles, the memory clock is 267MHz, and the Computation Array's clock is 125MHz.

Thus, for each memory port on a Pico card, we can support at most 14.3 Computation Arrays.

On the M-501, which has one memory port, anything more than about 14 Computation Arrays will be wasted waiting for the memory to deliver data. However, an M-503 with two memory ports can support up to approximately 28 Computation Arrays.

Our current implementation of the Computation Arrays will fill the FPGAs in our system at about 9 units, which means we are compute-bound, not memory-bound. Thus, to take advantage of the extra memory bandwidth available on the M-503 (or even to improve the performance on our current M-501 system) we need to optimize the size of the Aligner system, improve the clock speed of the Computation Array, or use a larger FPGA. One possible mechanism is to use C-Slowing [24], which heavily pipelines circuits with feedback loops, but requires a circuit that can work on multiple independent tasks. Short-read alignment meets the requirements of C-Slowing well, since we have these independent data sets, though careful design to avoid over-using the storage resources of the device may be needed.

7.2.2. Memory Bottleneck

To improve upon the computation time of this design, the memory lookups should be accelerated because it is the bottleneck of the system. More specifically, the bottleneck of the design lies in the memory system of the CAL Finder because it requires two accesses to external DRAM for each seed, as opposed to just one for each read in the Aligner. This would be a ratio of approximately 110:1 memory lookups in the CAL Finder to the memory lookups in the Aligner, respectively. One possible solution to this problem lies

in transferring the Matcher into software, because memory accesses from software has greater bandwidth in the Pico system and is therefore more efficient for this application.

It is estimated that a high-performance CPU (for example, one with 8 cores and 2.27 GHz) is able to get 280 million random accesses per second. In addition, the system has two sockets, each with their own memory path, doubling the data access rate and enabling us to get 560 million accesses per second. Our current hardware system takes approximately 40 clock cycles to access one block of data at a 267 MHz clock rate. This is equivalent to 6.68 million accesses per second. This means that moving the Matcher system to software could increase its data access rate by approximately 84 times.

Although the memory accesses in the Matcher will no longer be the bottleneck of the system, the memory accesses will continue to be the bottleneck in the Aligner. However, because the Aligner will be able to use the DRAM without the Matcher, it will have full access to the memory controller. Therefore, more Computation Arrays will be able to run efficiently in the Aligner, decreasing the runtime of the system.

7.2.3. Reverse Complement

DNA sequencing machines output short reads from a DNA strand, but the side it originated from is unknown. This becomes a problem for the reassembly process because the short reads will not align correctly with the other side of the DNA strand. Therefore, the system must run for both sides of the strand by incorporating the reverse complement of the short reads.

However, incorporating the reverse complement would double the data size of the short reads, and therefore double the computation and/or memory data necessary. [15]

elaborates on the various ideas to include the reverse complement in the system design, while maintaining the current runtimes.

References

[1] Bruner, Bob. “DNA.gif”. Bob Bruner's Chemistry and Molecular Biology Resources. 03/30/2011. <http://bbruner.org/bitn/bitn_fig/dna.gif>.

[2] Mardis, Elaine R. “The impact of next-generation sequencing technology on genetics”. *Trends in Genetics* Vol.24 No.3. Washington University School of Medicine, Genome Sequencing Center. 2007.

[3] F. Sanger, S. Nicklen, and A. R. Coulson. 1977. “DNA sequencing with chain-terminating inhibitor”. 1977. *Proc Natl Acad Sci* 74: 5463-5468.

[4] McPherson, John D. “Next-generation gap”. *Nature Methods Supplement*, Next-generation sequencing data analysis. Volume 6, No.11s:S2-S5. 2009

[5] National Institute of Health. “E. Coli Genome Reported: Milestone of Modern Biology Emerges From Wisconsin Lab”.09/04/1997. <http://www.nih.gov/news/pr/sept97/nhgra-04.htm>

[6] U.S. Department of Energy Genome Programs. “Human Genome Project Information” <http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml> 10/2011.

[7] Daniel C. Koboldt, Li Ding, Elaine R. Mardis and Richard K.Wilson. “Challenges of sequencing human genomes”. *Briefings in Bioinformatics*. Vol 11. No 5. 484 - 498. 19th April 2010 .

[8] David Stephen Horner,Giulio Pavesi, Tiziana Castrignano', Paolo D’Onorio DeMeo, Sabino Liuni, Michael Sammeth, Ernesto Picardi and Graziano Pesole. “Bioinformatics approaches for genomics and post genomics applications of next-generation sequencing”. *Briefings in Bioinformatics*. Vol 11. No 2. 181-197. 6th September 2009

[9] Paul Flicek and Ewan Birney. “Sense from sequence reads: methods for alignment and assembly”. Natural Methods Supplements. Vol.6 No.11s. November 2009.

[10] Levy S, Sutton G, Ng PC, Feuk L, Halpern AL, et al. (2007) The Diploid Genome Sequence of an Individual Human. PLoS Biol 5(10): e254. doi:10.1371/journal.pbio.0050254.

[11] Mihai Pop and Steven L. Salzberg. “Bioinformatics challenges of new sequencing technology” Center for Bioinformatics and Computational Biology. Trends in Genetics Vol.24 No.3 144. 12/07/2007.

[12] Konrad Paszkiewicz and David J. Studholme “De novo assembly of short sequence reads” *Briefings in Bioinformatics*. Vol 11. No 5. 457-472. 7th May 2010

[13] Homer N, Merriman B, Nelson SF. "BFAST: an alignment tool for large scale genome resequencing". PLoSONE 2009; 4(11):e7767.

[14] Li H, Ruan J, Durbin R. Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Res 2008;18(11):1851–8.

[15] Olson, Corey. "An FPGA Acceleration of Short Read Human Genome Mapping". Department of Electrical Engineering, University of Washington. Master's Thesis, June 2011.

[16] S. Hauck, A. DeHon. "Reconfigurable computing: the theory and practice of FPGA-based computation". Morgan Kaufmann Publishers, Burlington, MA, 2008.

[17] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 26, no. 2, pp. 203-215, 2007

[18] Altera. "Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform", ver. 1. September 2007.

[19] Erwin, Terry L. "The Tropical Forest Canopy: The Heart of Biotic Diversity". E.O.Wilson, ed., Biodiversity, National Academy Press, Washington, D.C., pp.123. 1988.

[20] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. "GPU computing". Proceedings of the IEEE, 96(5), May 2008.

[21] Reiners, Paul D. "Dynamic programming and sequence alignment: Computer science aids molecular biology". IBM developerWorks. 11 May 2008.

[22] S. Needleman and C.Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". *J.Mo/.Biol*., 48:443-453, 1970.

[23] T. Smith and M. Waterman. "The identification of common molecular subsequences". *J. Mol. Biol*. 147 (1981), 195-197.

[24] C. E. Leiserson, J. B. Saxe, "Retiming Synchronous Circuitry", Algorithmica, Vol. 6, pp. 5-35, 1991.