University of Alberta

HARDWARE ACCELERATION OF BIO-SEQUENCE ALIGNMENT ALGORITHMS ON FPGAS

by

Kevin Cushon 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Spring 2008

Your file Votre référence
ISBN: 978-0-494-45801-3
Our file Notre référence
ISBN: 978-0-494-45801-3

Abstract

This thesis describes new FPGA-based architectures for computing biological sequence alignments using the Smith-Waterman algorithm and a simplified version of the BLAST algorithm. The Smith-Waterman system has been fully implemented, achieving maximum performance of 1.5 GCUPS (billions of cell updates per second) for amino acid sequences and 5.3 GCUPS for nucleotides. It uses a new design based on partitioning the alignment into multiple sections, which allows our system to support query sequence lengths of up to 8192 symbols for both nucleotide and amino acid alphabets. This is longer than any previously published FPGA-based system. We also present a new architecture for computing a subset of BLAST on amino acid sequences at a projected rate of 100 GCUPS. This system has been fully designed and partially implemented. We experimentally confirm that our simplified BLAST implementation is equivalent to full BLAST with minor post-processing performed by a host PC.

Acknowledgements

I would like to thank my supervisor Stephen Bates for his guidance, encouragement, help, and financial support throughout my studies, as well as Bruce Cockburn, my co-supervisor, and Lukasz Kurgan and Jon Schaeffer, who made up my examination committee. I would also like to thank Peter Meulemans and Tim Rollingson at Arithmatica Ltd. and Bjarne Knudsen at CLCbio for their contributions and advice. Finally, I would like to thank Logan Gunthorpe for making my nightmare of a codebase manageable with his magic Python scripts.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 A Brief Introduction to Sequence Alignment

Comparison between sequences of nucleotides or amino acids is one of the most frequently performed tasks in bioinformatics. Quite often, sequences with unknown functionality are compared with databases of known sequences; these are known as *query* and *database* sequences, respectively. From a biological viewpoint, sequences with high similarity are likely to have functional, structural, or evolutionary relationships. These similarity measurements are called sequence alignments, or simply alignments.

Many different algorithms for performing alignments have been devised. Two of the more important algorithms are the Smith-Waterman algorithm [1] and Basic Local Alignment Search Tool, or BLAST [2]. BLAST is a heuristic algorithm that offers good accuracy at greatly reduced computational cost [3], and for this reason it has become the *de facto* standard for first-pass sequence database scans. Smith-Waterman, while having greater computational complexity, is guaranteed to provide the most optimal local subsequence alignment between two sequences. It is often used to refine results of less accurate alignments, including BLAST, and it remains the primary alignment method in processes requiring very high accuracy. Due to the relative importance and popularity of these algorithms in bioinformatics and related fields, we chose to conduct research into improved hardware-based acceleration.

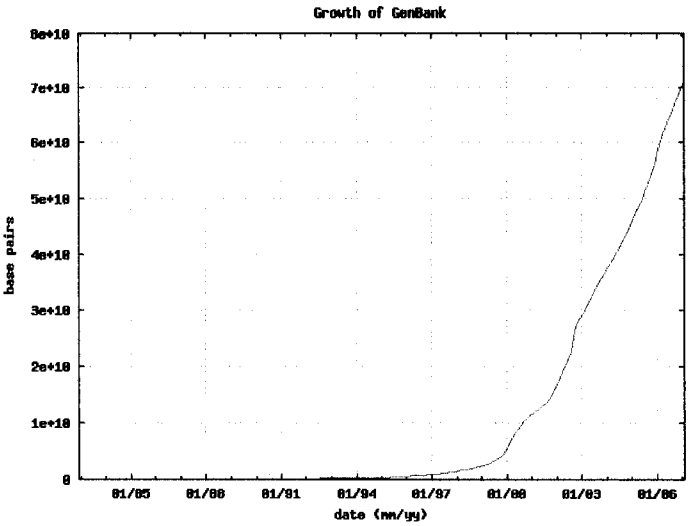

Figure 1.1: Growth of the NCBI GenBank database over time.

1.2 Motivation

Sequence alignments are computationally expensive tasks, and the sheer size and rate of growth of sequence databases are making software-driven alignments increasingly impractical. A typical operation is to compare a newly-discovered sequence with every sequence contained in a database, the largest of which consist of thousands or millions of known sequences and billions of base pairs. Furthermore, these databases are experiencing exponential growth, doubling in size over periods of as little as 2 years.

As an example of sequence database growth, Figure 1.1 shows the growth of the NCBI (National Center for Biotechnology Information) GenBank database, a major database of nucleotide sequences [4]. Other major databases, such as the Uniprot (Universal Protein) database [5] exhibit comparable volume and growth curves. Consequently, there is a high demand for alignment acceleration technology. In particular, application-specific hardware can achieve speedup ranging from tens to hundreds by performing parallel computation. This research seeks to meet that demand by creating new hardware-based alignment systems for Smith-Waterman and BLAST that offer practical improvement over those already available.

1.3 Platform

A reconfigurable platform is ideal for the task of accelerating Smith-Waterman and BLAST. One of the main reasons for this is that the same chip can be configured to run either algorithm. In addition, both algorithms can be run using nucleotide or amino acid symbol sets and a variety of parameters, such as substitution matrices and gap insertion penalties. Accommodating these on a non-reconfigurable platform would be difficult. Consequently we chose to conduct this research on an FPGA (field-programmable gate array) platform.

The platform we used for this research is the Opal Kelly XEM3010-1500P, which is an experimentation module based around a Xilinx Spartan-3 XC3S1500-4FG320 FPGA. Our work with BLAST targets the larger Spartan-3 XC3S4000. A USB interface and multi-platform API (application programming interface) is provided for communication with a host PC.

1.4 Achievements

Our Smith-Waterman system is based on partitioning the alignment into one or more sub-alignments, in which the query sequence is aligned with a segment of the database sequence. These sub-alignments are joined together to obtain the final result.

This architecture gives our design two major achievements. The first is that it allows our system to support very long query sequences - up to 8192 symbols for sequences of both nucleotides and amino acids. This is greater than any previously published FPGA-based implementation of Smith-Waterman. The second is that it makes our design less sensitive to the speed of the communication link, as each sub-alignment requires only a few symbols from the database sequence in order to proceed, during which time sequence data continues to be streamed over the link. Our system thus achieves higher efficiency than previous designs by reducing or completely eliminating the amount of time spent idle while waiting for new data. More detailed discussions of this aspect can be found in Section 2.8.2.

Our BLAST system provides a unique contribution in the simplified version of BLAST we use. By omitting certain aspects of BLAST that are difficult to implement in hardware, we obtain an algorithm capable of a much faster and simpler hardware implementation. Our system architecture of a single hit look-up unit, followed by multiple parallel processing paths for determining which hits are candidates to be passed on to a single extension unit, is completely novel. Our two-table design for hit look-ups provides more efficient memory use. Finally, our BLAST system is the first FPGA BLAST system to implement the two-hit condition for extension.

1.5 Biochemical Terminology

Several biochemical and biological terms are used throughout this document. The purpose of this section is to provide their definitions and clarify their meaning in the context of sequence alignments.

A *nucleotide* is, from a chemical viewpoint, a molecule consisting of a heterocyclic base, a sugar, and one or more phosphate groups. Nucleotides are the structural units of DNA and RNA. More generally, a *nucleic acid* is a macromolecule made up of nucleotides - DNA and RNA are the most common types of nucleic acid. In DNA, the four nucleotide bases are adenine (A), cytosine (C), guanine (G) and thymine (T). In RNA, thymine is usually replaced by uracil (U), which usually does not occur in DNA. Sequence alignments using nucleotide alphabets are almost always performed on DNA. As such, this document treats nucleotides as the set of A, C, G, and T, to the exclusion of U. Hence, any chain of nucleotides discussed in this document is referred to as *DNA*.

An *amino acid*, when the term is used in biochemistry, refers to molecules with the general formula $H_2NCHRCOOH$, where R is an organic substituent. The *standard amino acids* are a set of 20 amino acids which are directly encoded for protein synthesis by the canonical or standard genetic code, making them much more important in the study of proteins than general amino acids. This gives rise to the 20-symbol amino acid alphabet used to describe proteins and peptides in

sequence alignments. When the term "amino acid" is used in the document, it actually refers to the standard amino acids.

Proteins and *peptides* are polymers made up of amino acids in a specific order. The exact distinction between the two is a point of some contention in the biochemistry community. In its most basic form, the difference is that peptides are short (as in they contain fewer amino acids) while proteins are long. Proteins are a topic of more interest in sequence alignments than peptides, because their bigger size often gives them secondary structural and functional properties that are absent in peptides. Since the focus of this document is sequence alignments, the term "protein" is used to refer to any sequence of amino acids, even though shorter sequences may be more appropriately called peptides.

Chapter 2

Smith-Waterman

2.1 The Smith-Waterman Algorithm

This section contains a brief overview of the Smith-Waterman algorithm and prior research conducted into methods of accelerating it. It is intended to give the reader sufficient background knowledge to fully understand the hardware architecture devised in this research. For a full explanation of the algorithm, the reader is referred to [1] and [6].

The Smith-Waterman algorithm is a dynamic programming algorithm used to perform sequence alignments. It compares two sequences by computing the distance between them - the minimal cost of transforming one sequence into the other using substitution, insertion and deletion as elementary operations. Each operation has an associated cost.

The mathematical model of Smith-Waterman used in this research is shown below:

$$H(i,j) = \max\{0, E(i,j), F(i,j), H(i-1,j-1) + Sbt(A_i, B_j)\} \quad \text{for } 1 \leq i \leq m, 1 \leq j \leq n. \tag{2.1}$$

$$E(i,j) = \max\{H(i,j-1) - \alpha, E(i,j-1) - \beta\} \quad \text{for } 0 \leq i \leq m, 1 \leq j \leq n. \tag{2.2}$$

$$F(i,j) = \max\{H(i-1,j) - \alpha, E(i-1,j) - \beta\} \quad \text{for } 1 \leq i \leq m, 0 \leq j \leq n. \tag{2.3}$$

Consider two sequences A and B with respective lengths of m and n. The Smith-Waterman algorithm calculates all elements of a matrix H, with dimensions $m \times n$. H is referred to as the score matrix. A single element of the matrix, $H(i,j)$, represents the similarity of two subsequences of A and B ending at respective positions i and j.

Border values for all three equations are specified as $H(i,0) = E(i,0) = H(0,j) = F(0,j) = 0$ for $0 \leq i \leq m, 0 \leq j \leq n$. The $Sbt(i,j)$ term refers to a substitution cost table. This table assigns every possible combination of symbols an associated similarity score. $Sbt(i,j)$ is the substitution value for the ith symbol in A and the jth symbol in B. The terms α and β are gap insertion costs: α is the cost of the first gap, while successive gaps are assigned a cost of β. This type of gap model is generally known as the *affine gap model*. The use of this model with Smith-Waterman is described in [6]. Generally, α has a greater magnitude to reflect the assumption that opening a new gap represents greater dissimilarity than extending an existing one. The biological reason for this is that insertions and deletions often occur in blocks of multiple residues. Thus, two sequences may have multiple regions of high similarity separated by regions of low similarity. This gap model reduces the penalties incurred by long gaps, thus increasing the algorithm's sensitivity. The matrices E and F are intermediate variables used in the calculation of scores involving gapped sequences.

A simple example of a Smith-Waterman alignment calculation is shown in Figure 2.1. Two sequences of nucleotides, ACGTTT and ACCTT, are aligned using the substitution matrix shown in Table 2.1. Matches are assigned a score of 5, mismatches a score of -4. The gap penalties are set to $\alpha = -9$ and $\beta = -2$. The bolded elements indicate the trackback path from the maximal element 16 at $H(5,5)$. The pair of segments with maximal similarity is:

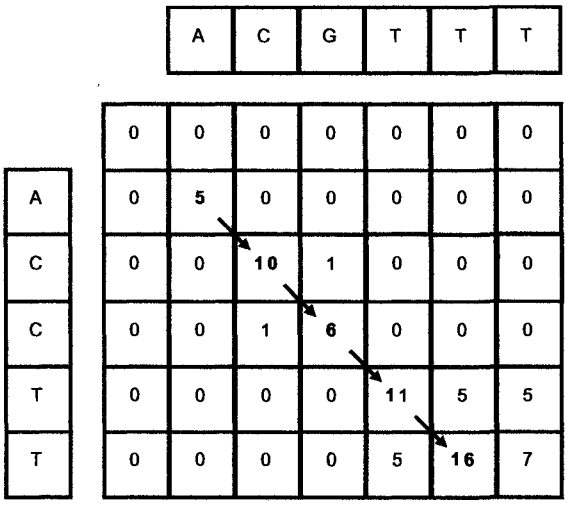

Figure 2.1: Score matrix H for the sequences ACGTTT and ACTT.

```
ACGTTT
|||||
ACCTT-
```

This example contains a mismatched pair at $(3, 3)$ but no gaps. However, the effect of opening a gap can be seen at $(2, 3)$, $(3, 2)$, and $(5, 6)$. The score matrix in its entirety contains useful information for alignments between highly similar sequences, such as the positions of individual gaps and insertions. However, the vast majority of alignments are run with the goal of finding a handful of highly similar sequences from a database of many thousands. In this case, the only useful information is the maximum score obtained. Sequences generating scores exceeding a certain threshold of interest are then investigated further. Because this is by far the most common use case, a hardware-accelerated implementation need only return this particular value to be useful.

Table 2.1: Substitution Matrix for the Example in Figure 2.1

	A	C	G	T
A	5	-4	-4	-4
C	-4	5	-4	-4
G	-4	-4	5	-4
T	-4	-4	-4	5

2.2 Parallel Computation

A highly useful property of Smith-Waterman is that it is possible to compute many elements of H in parallel. Referring to Equations 2.1 - 2.3, we see that each element is derived from elements in the preceding row and column. In general, $H(i, j)$ is dependent on $H(c, d)$, where $1 \leq c \leq i$ and $1 \leq d \leq j$. We also note that any given element $H(i, j)$ can be computed concurrently with all elements in the diagonal $H(i - n, j + n)$. These diagonals run from bottom left to top right through the matrix. Figure 2.2 shows the dependency network and diagonals of concurrently computable elements. As a consequence, elements along said diagonals are independent of one another and may be calculated simultaneously. In this way, the alignment is performed in $m + n - 1$ sequential steps, rather than the $m \times n$ steps needed to perform it serially.

2.3 Prior Work

The inherent parallelism of Smith-Waterman has been exploited in a number of previous designs. The most basic method of mapping the concurrent computation of diagonal elements is to use a linear systolic array of processing elements (PEs), in which one sequence is held statically within the array, while the other is shifted through sequentially. The PEs each compute one element of the matrix H per shift. At any given time, the elements of H being computed form an independent diagonal as described in Section 2.2. This process is illustrated in Figure 2.3. By registering the results of each PE, the array forms a modular pipeline to which additional PEs can be added without impacting the maximum clock frequency.

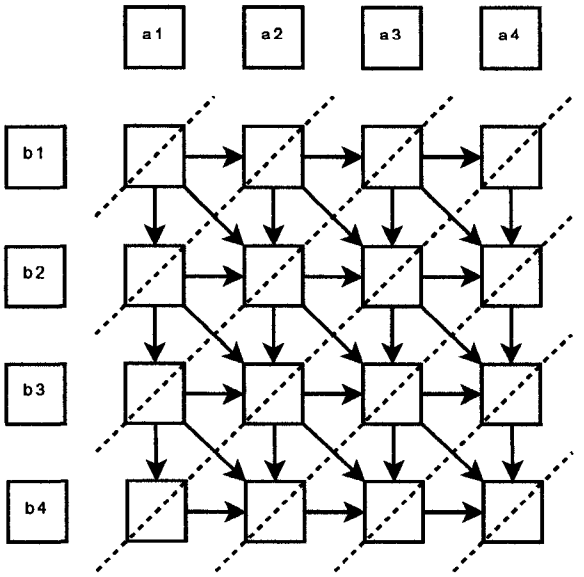

Figure 2.2: Dependency network for H. Elements along the marked diagonals can be computed concurrently.

For a PE to calculate $H(i, j)$, it requires only the results obtained by itself during the last time step and from its predecessor in the array during the last two time steps. Thus, only local interconnections are required between the PEs. This is the main advantage of this architecture, as shorter connections allow the pipeline to be run at higher clock frequencies, and less die area is used for routing signals compared to an architecture with many global interconnections.

2.3.1 Comparison of Our System with Previous Systems

Some early efforts to implement hardware accelerated Smith-Waterman used application specific systolic arrays, such as BioSCAN [7] and SAMBA [8]. The system in [7] uses a very long array of PEs implemented on ASICs, which is interfaced to a workstation using an expansion card. It is very simple in this regard; no partitioning or external controller is used. Due to sheer number of PEs, the system achieves very fast performance. However, it implements only a very rudimentary ungapped matching algorithm, which is likened to ungapped Smith-Waterman. However, it shares some commonality with our design in the linear PE array based architecture.

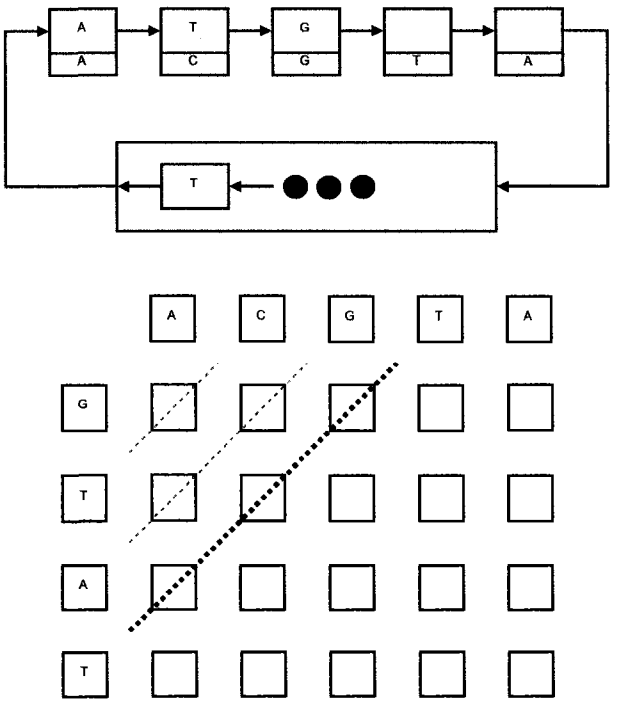

Figure 2.3: Smith-Waterman computed on a linear array of processing elements.

The system in [8] uses a 128 PE array implemented on ASICs, with an FPGA interface to an external memory chip. Long sequences are handled by partitioning along the query sequence. The partitioning algorithm is similar to ours, but only data from the last PE is saved to memory, which requires partitioning to take place along the query sequence. This results in a loss of flexibility as it places an upper bound on the database sequence length, and prevents the database sequence from being streamed, as in our system. Real performance is around 300 MCUPS, which is much slower than our system but represented a speed-up factor of 10-100 over the desktop PCs of the day, which in that respect is comparable to our system. In addition, this system implements the complete affine gap penalty and has a PE design very similar to ours.

Other systems exploit this principle using SIMD (single-instruction, multiple-data) systems. Two examples of such systems with a specific focus on biological

alignments are presented in [9] and [10]. These systems are quite different from our own in that they use 2-dimensional arrays of systolic processors configured as Smith-Waterman PEs. Diagonals are computed literally on this grid by activating the appropriate processors at the appropriate time. Clusters of PCs, each equipped with an expansion card with a certain size grid ([9] uses 32x32), divide the work and share border values to continue the calculation. Although these processors achieve greater performance than our system (real performance figures for amino acid sequences of about 2.5 GCUPS are reported, compared to a peak of 1.5 GCUPS for our system), the 2-D array and PC cluster platform is inefficient and much more costly.

An interesting approach described in [11] runs Smith-Waterman on high-end PC graphics cards, achieving parallel computation with OpenGL commands. However, this approach is entirely in software, albeit with creative use of commodity hardware, and so has little in common with our system beyond the parallel computation principles. Performance also falls well short of modern FPGA implementations, including our own (< 1 GCUPS).

There has also been some previous research performed using reconfigurable hardware. An early architecture presented in [12] uses the Splash-2 platform. Like [7], it simply uses a long systolic array without any additional logic or control, and computes edit distance rather than performing a full Smith-Waterman alignment. It does not have any provisions for sequences longer than the array - query sequences are limited to 384 symbols or less.

TimeLogic has developed hybrid workstations with integrated FPGAs for accelerating Smith-Waterman and other bioinformatics applications [13]. These systems are essentially workstation PCs with deeply integrated FPGAs for accelerating application-specific tasks. The FPGAs have the advantage of rapid access to the workstation's other resources such as RAM. Like [8], the alignment is computed on a systolic array, and intermediate results are written to RAM and read back into the array as needed. One advantage is that with access to a large pool of RAM, the maximum sequence size of this system is unlimited for practical purposes. The

disadvantage, of course, is that an entire workstation is needed to support the FPGA calculation.

More recently published designs, running purely on FPGAs, include [14] and [15]. The system in [14] uses a pipelined PE design with 4 stages. This design requires the system to work on 2 segments of the alignment simultaneously to avoid idle cycles. Long query sequences are handled by splitting the query into multiple segments and aligning each against the entire database, as in [8]. One consequence of the multi-threaded computation and partitioning is that some segments must overlap previous segments to avoid losing data, which causes a loss of efficiency. In addition, the system only operates on DNA sequences. Performance is 3.86 GCUPS on a Xilinx XCV2000E FPGA, but due to inefficiencies in this system, our system achieves equal or better performance with a smaller FPGA.

The design presented in [15] is both the most recently published and the most similar to our own. This system uses a single-cycle PE design, with the main innovation being that each PE stores only the column of the substitution matrix corresponding to the query symbol rather than the entire matrix. Long sequences are handled by storing multiple columns per PE, giving this design a maximum query sequence length of 1512 symbols with affine gap penalties, compared to our limit of 8192. The performance of this system is about 3 times greater than ours, though it uses an FPGA about 3 times as large.

The main contributions of our design are partitioning the database sequence and using block RAMs to achieve the longest maximum query length of any previously published FPGA architecture. The remainder of this chapter describes our design in greater detail. Our design also gives the unique benefit of desensitization to slow communications links, as well as the portability and economy of running on a lower-end FPGA.

2.4 Our Design

For our design, we sought to improve upon these earlier designs. One common deficiency lies in the maximum supported sequence lengths. As stated in Section

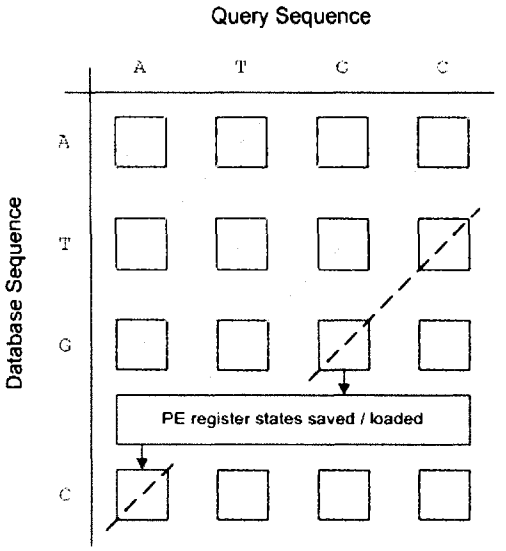

Figure 2.4: The alignment is partitioned into multiple sub-alignments in which the query sequence is aligned with database subsequences of length N, where N is the length of the PE array. A dual-port RAM is used to join successive slices.

2.3, one sequence must be stored statically inside the PE chain. As long as the PE chain is longer than one of the sequences, the alignment can be computed in a single pass of the other sequence through the PE chain. However, many identified sequences have lengths of hundreds or thousands of symbols, which is far too much for even the largest FPGAs [4]. Some way of handling sequences longer than the number of available PEs is obviously required.

Our design addresses the limitation on query length by dividing the computation of H, E and F into multiple passes. Consider a linear PE array of length N, a query sequence A of length Q, and a database sequence B of length D. Computation of the score matrix H is split into $\lceil D/N \rceil$ sections, each with a query sequence of length Q and a database sequence of length N. Computation of each section is carried out by rotating the database sequence segment through the PE array continuously, while a separate controller component loads symbols from the query sequence into the PEs. The individual sections are joined together using a scratch buffer to store register states from each PE. Because the scratch buffer must handle a read and a write per clock cycle, a dual-port RAM is used. When a PE

performs a computation for an element adjacent to the "border" between sections, it writes its register state to the scratch buffer. Likewise, when the next section is evaluated, the PEs read their initial register states from the scratch buffer. These transactions are controlled by a central controller. With each initial state loaded into a PE, the appropriate query symbol for the current state of the calculation is loaded as well. In cases where $Q > N$, a single PE may load a previously saved state multiple times per slice. The query sequence is stored in a separate RAM. This process is illustrated in Figure 2.4.

Dividing the alignment in this manner also desensitizes our design to slow communications links. Before computation begins, the entire query sequence is transferred and stored in memory. Since the query sequence is typically much shorter than the database it is being aligned with, this delay is not significant compared to the total computation time. Once the first N elements of the database sequence have been transferred, computation of the first section of the score matrix begins. As long as the next N elements of the database sequence are received before the first section is complete, computation of the next section may begin without having to pause for more data. Thus, for sufficiently long query sequence lengths, the performance of our system is limited by the rate of computation rather than the speed of the data link. However, for shorter query sequences, the system must pause at the end of each section until the next database segment is received. Pausing the system is accomplished by disabling all registers in the PE array.

The FPGA we targeted for our architecture, the Xilinx Spartan-3 XC3S1500, features integrated block RAM modules that meet our memory requirements. The RAM used to store the query sequence, the dual-port RAM for storing intermediate PE states, and the I/O buffers are synthesized using these block RAM modules. Thus, no logic resources are consumed by memory and no off-chip memories are needed.

All communications between the system and host PC pass through FIFO input and output buffers. Reserved command words are used to delimit the begin and end of the query and database sequences. Computation begins automatically once the

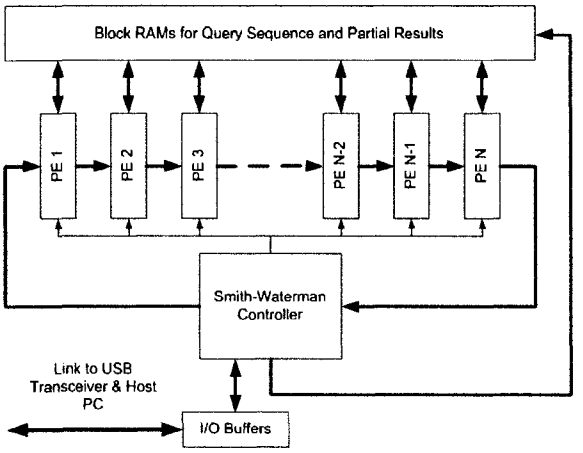

Figure 2.5: Top-level system architecture. This consists of N processing elements, a control element, and RAMs for storage and I/O buffering.

system has received the first N symbols of the database, and upon completion the result is written to the output buffer. The Opal Kelly XEM3010-1500P development board uses a USB interface to communicate with the host. HDL modules for the USB interface and communication endpoints are provided in the Opal Kelly API.

A top level diagram of our design is shown in Figure 2.5.

2.5 The Processing Element

In this section, we will examine the individual Smith-Waterman processing element (PE) in detail. A PE implements all the operations needed to compute one element $H(i, j)$ of the score matrix per clock cycle, and forward the necessary results to the next PE in the array. In addition to the basic Smith-Waterman functionality, our PE design also reports the location of the maximal element.

Figure 2.6 shows a simplified RTL schematic of the processing element. Ports on the left are inputs from the previous PE, and ports on the right are outputs to the next PE (except for the first and last PE in the array, which are connected to the controller). The ports at the top are inputs and outputs to the query and scratch RAMs.

An input from the previous PE is equivalent to an input from the previous col-

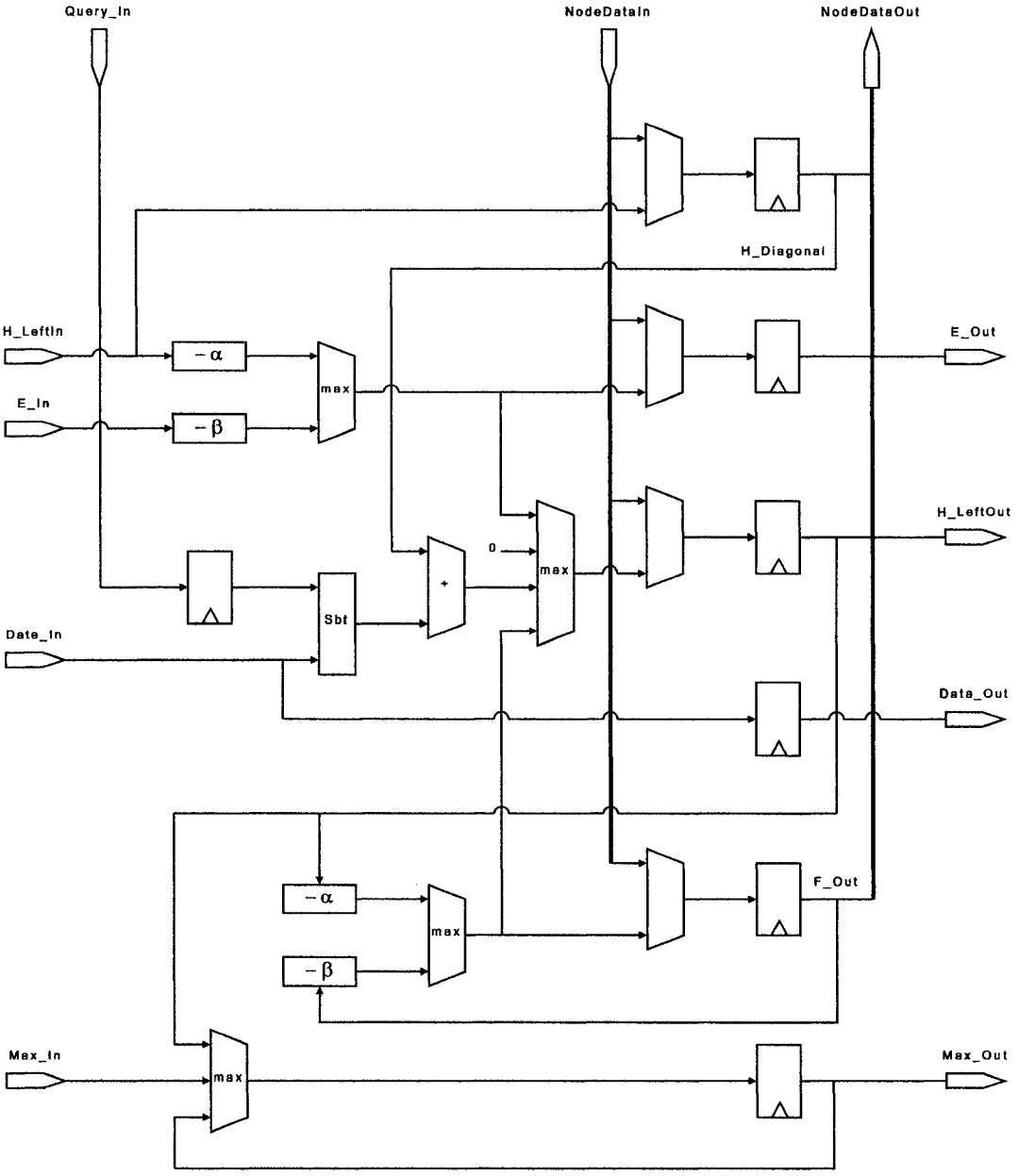

Figure 2.6: Simplified RTL schematic of the processing element.

umn of the result matrices. Hence, the inputs `H_LeftIn` and `E_In` are equivalent to $H(i, j-1)$ and $E(i, j-1)$, respectively. Likewise, a delay of one clock cycle is equivalent to shifting one row down. The signal labeled `H_Diagonal`, equivalent to $H(i-1, j-1)$, is obtained by registering `H_LeftIn`. Note that the calculation of F is internal to the PE, since it is dependent only on prior results from the same column.

In addition to calculating H, E, and F, each PE also stores a copy of the highest similarity score it has seen in the evaluation thus far. It is registered and output to the next PE along with the calculation results, and at each step it is compared with the current similarity scores. This is to ensure that the value of the best local alignment is propagated through to the controller.

Calculation results are stored in a register file. This links the PEs together in a modular pipeline, so adding or eliminating PEs does not affect the maximum clock frequency of the pipeline. Results needed in future calculations are routed back into the PE for use during the next clock cycle. One consequence of this design is that it increases the number of sequential steps needed to perform the alignment from $n+m-1$ to $n+m+1$. One extra clock cycle is needed to compute the final value of $H(i, j)$ using the results stored during the last cycle; another is needed to propagate the final result from the last PE to the output. The overall impact on performance is minimal - performance scales with $N \times \frac{N}{N+1}$ instead of N.

When a PE performs a computation for an element adjacent to the "border" between two sub-alignments, its register state is written to the scratch RAM. Likewise, when the next section is evaluated, the PEs read their initial register states from the RAM and the new query symbol is loaded. Each PE has an input signal to control when register states are loaded from RAM. There is also a pause signal which disables all registers. This is used when the system must wait for more sequence data before proceeding.

2.5.1 Substitution Matrix Design

To calculate the value of $H(i, j)$, a PE requires a lookup from the substitution table $Sbt(A_i, B_j)$, which contains similarity scores for each possible combination of symbols in the two sequences. Because every PE must perform a lookup from this table each clock cycle, every PE contains its own local copy.

Substitution tables in our system are described by a VHDL file containing nested `case` statements. These VHDL files are generated automatically by a Python script, using scoring data provided in tabular form, much like that shown in Table 2.1. Thus, it is very easy to generate substitution matrices with arbitrary scores and an arbitrary number of symbols. The resulting hardware is in the form of distributed ROM synthesized from the reconfigurable portions of the FPGA.

The decision not to use reprogrammable substitution matrices was motivated by several factors. A substantial amount of research has been conducted into substitution matrices, with the consensus that the choice of matrix has a significant effect on the accuracy and usefulness of alignment results for both nucleotide [16] and amino acid matrices [17], [18]. Furthermore, there is also some benefit in scaling the substitution scores dynamically, which can only be accomplished on fully reprogrammable matrices. Two examples are varying substitution scores with respect to their positions in the sequences [19], as well as scaling the entire matrix depending on the total lengths of the sequences [16].

Therefore, it would be highly useful to have reprogrammable score tables, so that the score set best suited to a particular application could be loaded and used. This would be possible by storing the substitution matrix in a RAM, but this approach faces several disadvantages; particularly increased resource consumption.

The most vital disadvantage is that fewer resources would be available for the scratch buffer or for PEs. If block RAM is used, less is available for the scratch buffer, which is contrary to our primary design goal of accommodating long sequence lengths. Furthermore, since block RAM modules on our platform have 1 cycle of latency on reads [20], our PE design would have to accommodate this by implementing pipelining, which would thus increase scratch memory requirements,

reduce maximum sequence length, and require more complicated control logic.

Distributed RAM (that is, RAM synthesized from the configurable logic portions of the FPGA) faces similar disadvantages. Reprogrammable memory consumes twice as many resources as ROM [20], and thus would greatly increase the footprint of each PE. This has an especially large impact on the resources consumed by the larger amino acid substitution matrices. Using distributed RAM for the substitution matrices would thus reduce the length of the PE chain by an unacceptable amount.

In addition, as our system uses a reconfigurable platform, it is possible to synthesize designs with any desired substitution matrix and rapidly reconfigure the hardware. Configuration files containing standard matrices such as the BLOSUM series [21] can be kept on the host PC, and used to reconfigure the device as needed. Granted, this method is not as flexible, but having a wide array of pre-synthesized matrices at the user's disposal partially negates this inflexibility.

2.5.2 Gap Penalty Design

Gap penalties are similar to the substitution matrix in that it is desirable to have them reconfigurable. Optimizing gap penalties for a specific application has been shown to significantly increase performance in Smith-Waterman [18]. Gap penalties themselves are implemented with subtracters, as shown in Figure 2.6, so making them reconfigurable is simply a matter of storing the gap penalty in a register. Unlike the substitution matrix, this does not result in any serious design issues. Gap penalties are reconfigured in all PEs simultaneously by a command from the controller. Each PE has control inputs for loading α and β, as well as a dedicated input for the new gap penalty.

2.6 The Controller

The controller unit is a finite state machine (FSM) responsible for managing the input of data into the PEs and collecting results. This includes unpacking sequence

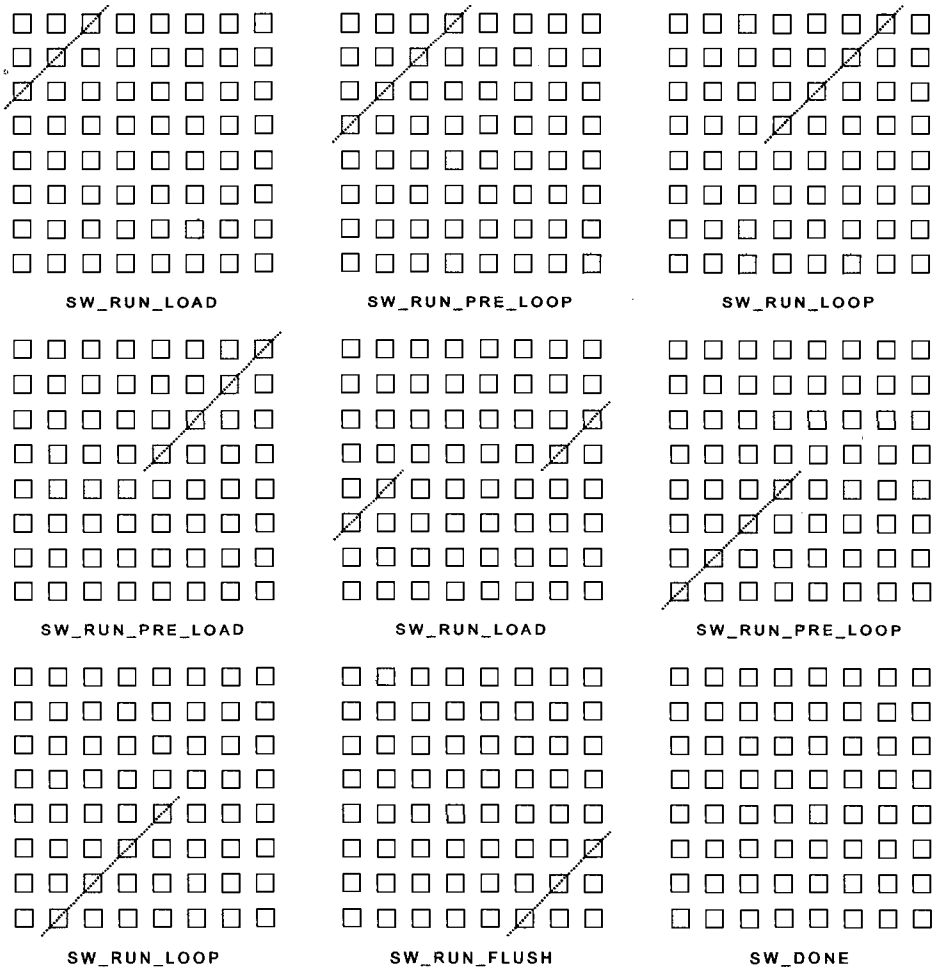

Figure 2.7: Controller states at various points in the alignment calculation for a system with $N = 4$.

data from an input pipe, managing transactions between the PEs and the query and scratch memories, and placing the final results in an output pipe.

The states are soft-coded to take advantage of optimization efforts in the synthesis tools. A list of the states used in the controller, along with brief descriptions for each, is shown in Table 2.6. In addition, Figure 2.7 illustrates the relationship between the controller state and progress of the calculation. A state transition diagram is shown in Figure 2.8.

The state `SW_DONE` is the system's idle state. Transitions out of this state are triggered by reading a command word from the input pipe. Command words are

Table 2.2: Smith-Waterman controller states.

State Label	Description
SW_DONE	System idle state.
SW_GET_QUERY_LEN	System is awaiting the length of the query sequence.
SW_LOAD_QUERY	System is receiving the query sequence and loading it into RAM.
SW_RUN_LOAD	Alignment is running, with new database subsequence being loaded into PE array.
SW_RUN_LOOP	Alignment is running, with database symbols being looped from the end of the PE array back to the beginning.
SW_RUN_PRE_LOAD	Transitional state between SW_RUN_LOOP and SW_RUN_LOAD.
SW_RUN_PRE_LOOP	Transitional state between SW_RUN_LOAD and SW_RUN_LOOP.
SW_RUN_FLUSH	State used at the end of a calculation to flush pending results from the PE chain into the controller.
SW_WRITE_DB_LOCATION	System is writing the DB sequence index of the maximum to the output pipe.
SW_WRITE_QUERY_LOCATION	System is writing the query sequence index of the maximum to the output pipe.
SW_WRITE_SCORE	System is writing the maximal alignment score to the output pipe.
SW_LOAD_ALPHA	System is loading a new value to be used for the gap penalty α.
SW_LOAD_BETA	System is loading a new value to be used for the gap penalty β.
SW_RESET	System is resetting. All registers are set to their default values.

available for the host to begin a query sequence (`ctrlStartQuery`), begin a database sequence (`ctrlStartDB`), and load new values of α and β, which are labelled (`ctrlLoadAlpha` and `ctrlLoadBeta` respectively).

Upon reception of `ctrlStartQuery`, the controller state transitions immediately to SW_GET_QUERY_LEN. The next word from the input pipe is read and

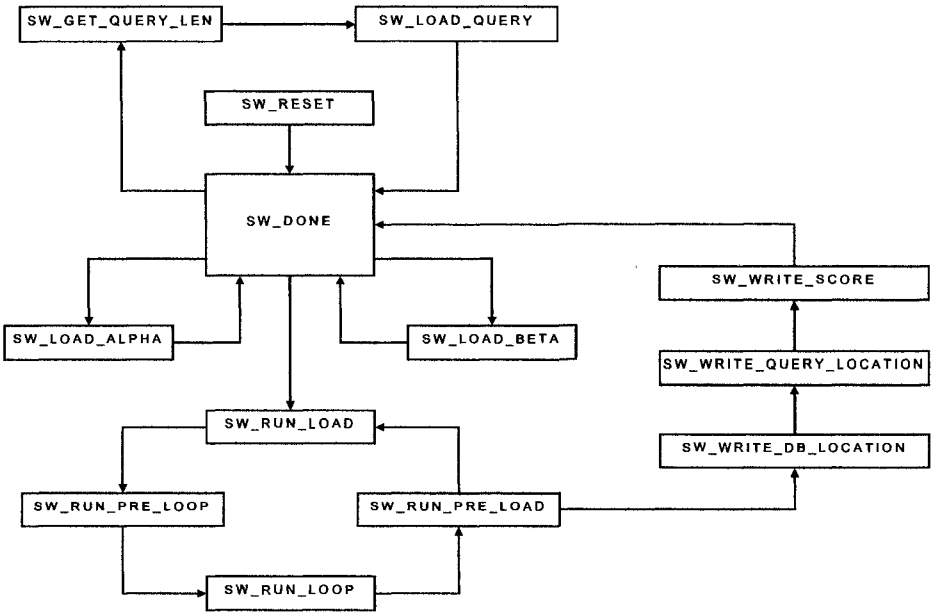

Figure 2.8: Smith-Waterman state transition diagram.

interpreted as the length of the incoming query sequence. The state then transitions unconditionally to SW_LOAD_QUERY, in which words from the input pipe are read, unpacked into individual symbols and written to the query RAM. The previous query sequence, if one was present, is overwritten. This continues until a null symbol is detected by the controller, indicating the end of the sequence. The state then transitions back to SW_DONE.

The states SW_LOAD_ALPHA and SW_LOAD_BETA are used in updating the gap penalty registers inside the PEs. Upon reception of `ctrlLoadAlpha` or `ctrlLoadBeta`, the corresponding state transition is made and the next word is read from the input pipe. This is the new value of α or β. On the next clock cycle, the new value is loaded into the PEs, and then the system transitions back to SW_DONE.

The reception of the command word `ctrlStartDB` triggers an immediate transition to SW_RUN_LOAD, and begins the alignment calculation. The address pointers for the query and scratch memories are reset, and the controller loads the first query symbol and initial register state into the first PE. If this is the first segment

of the alignment, the initial register state is set to 0 instead of read from the scratch memory. On the next clock cycle, the controller feeds the first database symbol into the first PE, along with initial inputs of 0 for H_LeftIn and E_In. In addition, the controller loads the second query symbol and initial register state into the second PE. This pattern continues, with the controller loading query symbols and initial states into the PEs one step ahead of the database, until the last PE in the chain has been loaded. After this point, the system transitions to SW_RUN_PRE_LOOP. In the first occurrence of this state, the controller stops reading new symbols from the input pipe and begins writing PE register states to the scratch memory. Note that this is the point in the alignment at which the first cell at the boundary between two database segments is calculated (see Figure 2.7). In addition, the system is not ready to begin looping database symbols yet, as the first database symbol has only propagated through to the last PE. Two more cycles are needed - one to complete pending calculations in the PEs, and another to load a new query symbol and register state into the first PE, as a PE cannot perform a calculation and load a register state at the same time. Note that as a consequence of this delay, a bubble is created in the PE pipeline where one PE in the chain is being loaded instead of calculating. In light of this delay, output from the last PE is routed through a register in the controller before being input into the first PE. The system then transitions to SW_RUN_LOOP.

In SW_RUN_LOOP, database symbols and other outputs leaving the last PE are looped back into the input of the first PE. There is one read from the query and scratch memories per cycle, along with one write to the scratch memory. The address pointers for each are incremented after each operation. Computation proceeds much as it did in SW_RUN_LOAD, with the controller loading query symbols and register states ahead of the database and writing boundary values to the scratch memory. The difference is that the inputs to the first PE come from the outputs of the last PE, instead of having database symbols read from the input pipe and the other inputs set to 0.

The state transitions to SW_RUN_PRE_LOAD once the end of the query sequence

is reached. This state prepares the system for a return to the SW_RUN_LOAD state by resetting the query and scratch read address pointers, and performing a read from the input pipe to obtain new database symbols if necessary. It is otherwise equivalent to SW_RUN_LOOP. It transitions unconditionally to SW_RUN_LOAD. This cycle of four states, from SW_RUN_LOAD to SW_RUN_PRE_LOOP to SW_RUN_LOOP to SW_RUN_PRE_LOAD and then finally back to SW_RUN_LOAD, is repeated until a null symbol is received, indicating the end of the database sequence.

In order to simplify the bookkeeping of address pointers, database sequence data, and the system structure, it is assumed that both the query and database sequence lengths are integer multiples of N, where N is the number of PEs in the system. It is the responsibility of the host to round sequences up to the nearest multiple by padding with the symbol N, which has a negative similarity score when aligned with any symbol, including itself. A consequence of this assumption is that the end of the database will always be detected in the SW_RUN_PRE_LOAD state. When this happens, the system enters the SW_RUN_FLUSH state. This state flushes pending results out of the PE chain by counting down clock cycles from N. When the countdown reaches zero, the system transitions to a result writing phase. The first state is SW_WRITE_DB_LOCATION, in which the database index of the maximum is written. SW_WRITE_QUERY_LOCATION and SW_WRITE_SCORE follow immediately afterwards, after which the system state returns to SW_DONE. The host can then initiate a new command.

2.6.1 Location Finding

One of the controller's features is the determination of the location (that is, the indices within the query and database sequences) of the maximal element within the score matrix. This information is useful for post-processing, because it reveals the subsequences of interest, and further examination can be concentrated in that area to the exclusion of the rest of the sequences. In alignments with multiple occurrences of a certain maximum, only the first one seen by the system is reported.

Location finding is accomplished by having each PE tag new maxima with their

node IDs. The node ID is simply the position of the PE within the chain, from 0 to $N-1$. This node ID is propagated forward along with the maximum. When a new maximum reaches the controller, it is possible to determine exactly where it occurred from the accompanying node ID.

The location in the query sequence is simple to determine, as it will always be some multiple k of N, plus the node ID. The value of k is tracked using a counter that increments when the query address pointer passes a multiple of N.

The database location is found using another counter. This one counts the number of symbols that have been input to the PE chain. Because our system works by aligning N-length database subsequences against the full query, it resets after reaching $N-1$. When a new maximum arrives, the difference between N and the nodeID is subtracted from the counter to obtain the location.

2.6.2 Pausing

Communication with the host is conducted through a vendor-supplied pipe interfaces mated with FIFOs for both input and output (these components are described in more detail in Chapter 4.1). Because the database sequence is streamed from the host, it is possible for the system to process an N symbol section of the database sequence before the next N symbols are available to be loaded. To deal with this contingency, some way of pausing the system is needed.

The pause function is implemented by simply inverting the input FIFO empty signal. When this FIFO is empty, all registers in the system are disabled and retain their old values until the FIFO contains data again.

One complication is that the block RAMs have one cycle of latency for reads, so additional logic is required to avoid missing a read during the pause/unpause cycle. The output of each RAM is connected to a register, and the registered output and direct output fed into a multiplexer. During normal operation, the direct data from the RAM is used. On the first cycle after a pause, the registered data is used.

2.7 Example

This section describes an alignment between two very short sequences to illustrate the workings of our system. The query sequence will be AGGCTTA, the database sequence CGGCTTGCC, and the system will be a 5 PE system configured for nucleotide sequences. The substitution matrix used scores 1 for a match and -3 for a mismatch.

First, the host PC must pad the sequences up to the nearest multiple of N using the negative (N) symbol. This symbol has a negative substitution score with all symbols, including itself, so padding sequences with these has no effect on the result. The query and database sequence become AGGCTTANNN and CGGCTTGCCN respectively.

Next, the host sends the `ctrlStartQuery` command word, followed by 1 word containing the length of the query sequence in symbols (10), followed by the sequence itself. The query sequence is stored in the query memory. The alignment has not actually begun at this point, as the first N symbols of the database sequence are needed to begin it.

After sending the query sequence, the host sends the `ctrlStartDB` command word, followed immediately by the database data. Reception of `ctrlStartDB` causes all registers in the PE chain to reset, ensuring a clean slate. After the reset, the system loads the first PE with the first query symbol from the query RAM and an initial register state from the scratch RAM. We will call this time $t = 0$. However, since this is the first "section" of the alignment, the system intercepts the data from the scratch RAM and replaces it with all 0's, since this is the boundary value of the score matrix.

On the next clock cycle, $t = 1$, the system does the same for the second PE, and inputs the first symbol of the database sequence into the first PE. The first PE calculates $H(1, 1)$ using the 2 symbols.

At $t = 2$, the third PE is loaded with the third query symbol. The database symbol initially in the first PE is shifted into the second PE, while the second symbol

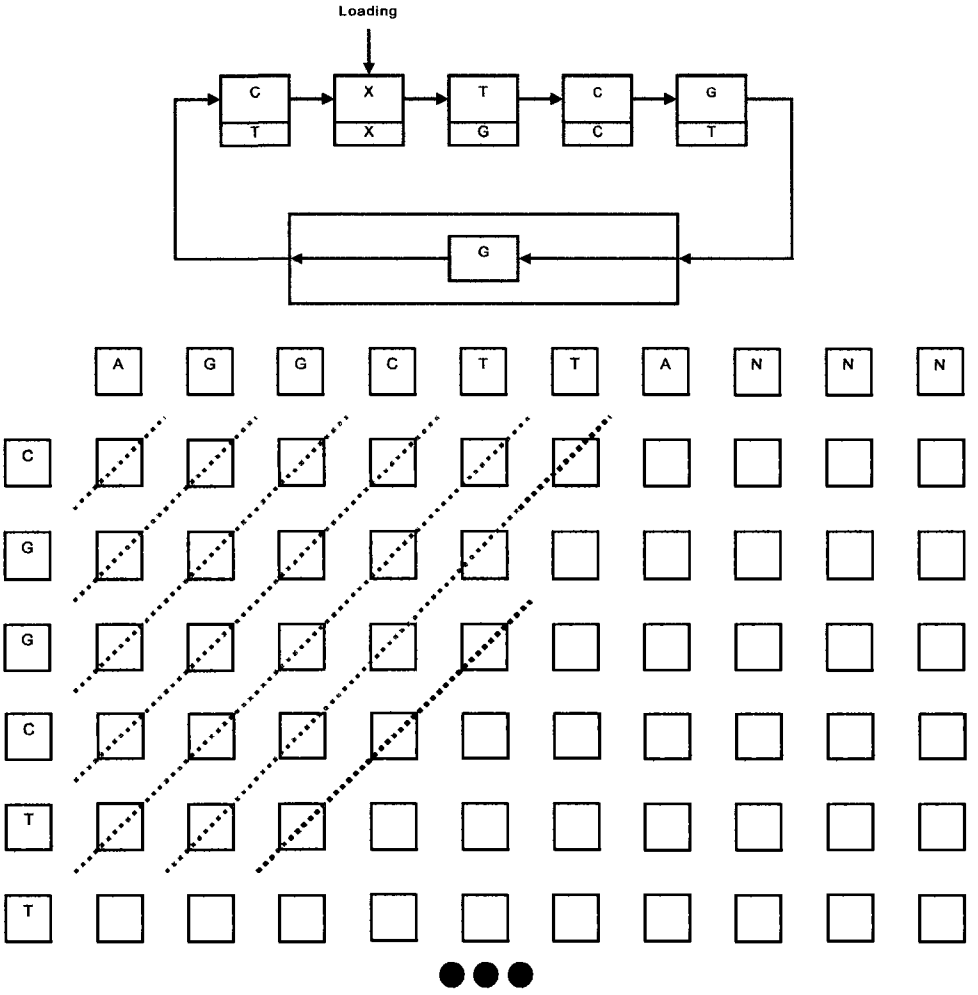

Figure 2.9: The example system at time $t = 7$.

in the database is loaded into the first PE. On this cycle, $H(1,2)$ and $H(2,1)$ are computed.

This cycle continues until $t = 5$, when the last PE is loaded with a query symbol. At this point, the controller switches to loop state, in which database symbols input to the first PE are taken from the output of the last PE, rather than using new symbols from the input pipe. At $t = 6$, the sixth symbol of the query sequence is loaded into the first PE. The database symbol shifted out of the last PE is registered in the controller. The reason for this is a PE cannot be loaded with a new state or query

symbol and perform a calculation at the same time. Thus, $H(5,2)$ is not actually calculated until the next clock cycle, $t = 7$. The state of the system at $t = 7$ is shown in Figure 2.9.

At $t = 11$, the system begins loading new database symbols again. The first PE is loaded with the first symbol of the query sequence and the saved register state that the first PE was at immediately after completing the calculation of $H(5,1)$. Since this is the second section of the alignment, the scratch buffer contains valid PE register states which are loaded prior to beginning a calculation of a second-section element of H.

The process continues as above until $t = 20$, at which point there are no more database symbols to align and symbols shifted out of the last PE are simply discarded. At $t = 24$, the last element of H is calculated, and at $t = 25$, the result is shifted into the controller and the alignment is complete.

2.8 Synthesis and Evaluation

2.8.1 Configurations

Our design supports a wide variety of different configurations. Our build environment for this work is set up in such a way that all configurable parameters - symbol set and substitution matrix, score width, number of PEs, and maximum query depth - are read from a file by a build script, which then inserts them into the VHDL files and generates the binary file used to configure the FPGA. Our standard configurations consist of one with a 12-bit score width, for general purpose use, and a 7-bit score width, which is intended for database scanning applications in which the goal is to identify sequences scoring above a certain threshold of interest. For nucleotides, a basic match/mismatch substitution matrix is used, while amino acid configurations use the BLOSUM62 matrix [21]. All configurations support a maximum query sequence length of 8192 symbols. Any query sequence length down to 1 symbol is supported by padding the sequence with a negative (N) symbol, which has a negative substitution score with all symbols, including itself. Theoretical

Table 2.3: Smith-Waterman system configurations

	Nucleotide		Amino Acid	
Score width (bits)	12	7	12	7
PEs (N)	64	110	32	38
Max. query length	8192	8192	8192	8192
F_{clk} (MHz)	46	49	40	41
Theoretical max. performance (GCUPS)	2.90	5.34	1.24	1.52

maximum performance is calculated by multiplying $N - 1$ PEs active at any given time by F_{clk}. In practice, this limit can only be asymptotically approached as the lengths of the sequences tend to infinity, the reason being that the "corners" of the matrix, in which fewer than $N - 1$ PEs are active in the alignment, become less significant as the sequences become larger.

Table 2.3 contains a summary of these configurations and their maximum clock frequencies.

All four of these configurations consume 100% of the Spartan-3 XC3S1500-4FG320's block RAM resources, and above 95% of the logic resources. Although it is possible to squeeze another PE or two out of all the above setups, doing so degrades the maximum clock frequency enough that the performance penalty from the slower clock overcomes the gain from the extra PEs. The final number of PEs for each design was chosen to maximize theoretical performance, as determined by $(N - 1) \times F_{clk}$.

2.8.2 Performance Testing

Performance of the actual system was measured using test builds that counted the total clock cycles and idle clock cycles on the chip. These counts were written to the output pipe in place of location statistics. These changes did not otherwise impact the operation of the device, and the location was still calculated, though not reported back to the host. The critical path and maximum clock frequency were also unaffected.

The test methodology was to first generate a random query sequence on the host

PC and write it to a buffer. Next, a random database sequence of a particular length is generated, and 150 consecutive copies are written to the buffer. All sequences are separated by 10 bytes of nulls (`0x00`). The device is reset, programmed with the binary under test, and then the entire buffer is send to the device with a single call of the `WriteToPipeIn()` API function in Python. The test is repeated using the same buffer 100 times, and the best, mean, and worst times are recorded. The purpose of calculating the mean is that data transfers to and from the device are performed using USB bulk data transfers, which do not provide guaranteed amounts of capacity or latency. Thus, we would expect some degree of randomness or jitter in the results, as the device could be left idle for significant amounts of time waiting for new data.

Each configuration was subjected to two rounds of testing: one with a short query (length of $2 \times N$, which is the shortest supported length), and one with a long query of over 1000 symbols. The length of the long query was set to the multiple of N closest to and greater than 1000. Each round of testing consisted of several database runs, with database sequence lengths covering the same range as the query sequence.

2.8.3 Performance Measurements and Analysis

Figures 2.11 - 2.18 show the performance measurements for each of the four system configurations specified in Table 2.3. The sequence type, number of PEs, internal word size (score width), query sequence length, and clock frequency are also shown in the figure captions. Each system has two sets of plots - one with short query sequences and one with long query sequences as described in the previous section. The top plots in each figure contain the directly measured data - total computation time and idle time. The bottom plots show derived data. On the left is non-idle time, which is presented for comparison to idle time. The bottom right plot shows the performance of the system in CUPS. CUPS is a unit often used to describe the performance of alignment systems, which stands for *cell updates per second.* A cell update consists of a complete computation of one element of the score matrix

Table 2.4: Performance impact of various factors on our Smith-Waterman systems.

Parameter	Performance Effects
Database length	Increases logarithmically with increasing length, due to lowered impact of "corners" in which not all PEs are active.
Query length	Same as database length, with the added effect of increasing query length reducing average idle time. Thus query length has a much greater impact than database length.
Sequence type	Higher for nucleotide than amino acid sequences due to reduced size and complexity of substitution matrix.
Score width	Performance increases as score width is reduced due to increased number of PEs, and lower complexity resulting in higher clock frequencies.

H. Table 2.4 summarizes observations made from these plots.

The observations show that all systems increase in CUPS performance as the database sequence increases in size. The reason for this is that at the beginning and end of the alignment, not all the PEs are used in the calculation. For the first N cycles of the calculation, the database has just begun to be shifted through the PE array, and the PEs that are later in the chain sit idle. The opposite situation happens during the last N cycles, during which the earlier PEs are idle. For shorter sequences, these "corners" of the score matrix make up a more significant part of the alignment. As the plots show, performance increases logarithmically as the database sequence grows.

The same affect is also present for different lengths of query sequences, but the query sequence length has a more powerful primary effect on performance. Longer query sequences give the system more time to stream the next N database symbols over the communication link, resulting in less idle time and higher CUPS performance. Note that on average, idle times for runs with short query sequences are higher for runs on the same system with longer query sequences. Consequently, our system performs better the longer the query sequence becomes.

Performance for nucleotide sequences is much higher than performance for

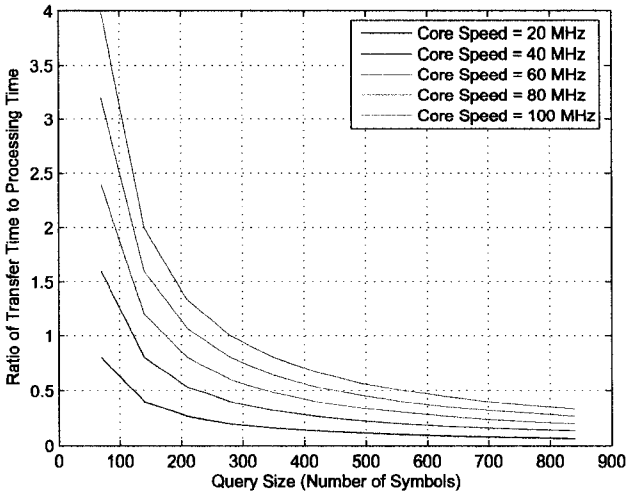

Figure 2.10: The ratio of transfer time to processing time.

amino acid sequences. The reason for this is that amino acid systems have much fewer PEs and a slightly slower clock frequency, both effects of the larger alphabet. While the substitution matrix for nucleotide systems is only 5×5, for amino acids it is 21×21 (both include an extra symbol "N", which scores negatively against everything). The larger matrix takes up a great deal more resources and is also slower, which adds to the system's critical path.

The final factor examined is the score width, or internal word width of the PE chain. As described in Section 2.8.1, we used two different word sizes: 12 bits for general use, and 7 bits for high-speed computations at the risk of saturating the score values. Reducing the word size reduces the amount of logic resources needed for each PE, which allows us to fit more of them on the device. The 7-bit configuration for nucleotide sequences allows a much larger PE chain, giving a corresponding increase in performance proportional to the number of PEs. For amino acids, the effect is less significant due to the relatively inelastic resource usage of the ROM used for the substitution matrix. The increase from 32 to 38 PEs is only a 19% increase, while reducing the word size on the nucleotide systems increases the PE count from 64 to 110, a 72% increase.

The observations also reveal a few odd irregularities with respect to the com-

munications interface. Some degree of randomness is expected in the amount of time spent idle, due to timing inconsistencies in the USB bulk transfers used to stream the database sequence. However, with some particular sequence lengths, the system spends much more or much less time idle than with the adjacent sequence lengths. One notable example is in Figure 2.12, in which all test runs with a database sequence of length 320 completed without a single cycle spent idle, yet those for 256 and 384 spent between 0ms and approximately 20ms idle from run to run. Then, with a database length of 576 symbols, all runs spent around 20ms idle, even though the best runs for all other database sequence lengths achieved zero idle time. Since each test is repeated 100 times to measure the mean, we can conclude that this behaviour is consistent and systematic, suggesting that this is likely due to peculiarities in the vendor's API or USB driver.

The results also confirm our expectations regarding the ratio of communication time to computation time. Theoretical ratios of communication time to computation time are plotted in Figure 2.10. This plot assumes a nucleotide system with 70 PEs and a USB communication rate of 80 Mbit/s. Where the ratio exceeds unity, the system is communication-limited and must spend time idle while waiting for new data. Our observations confirm this analysis. In the best case test runs with long query sequences, the system spends no time idle and achieves performance approaching the theoretical maximum. However, with short query sequences, there is always an observed idle time and CUPS performance drops significantly.

2.8.4 Comparison With Desktop Microprocessors

As with all application-specific hardware, the performance of our system as compared to a standard desktop PC microprocessor is an important benchmark. There must be a significant speed-up in order to justify the additional costs compared to a software solution.

For the performance of a software system running on a desktop PC, we use a reference figure of 45 MCUPS reported in [11]. This is for a highly optimized C implementation of Smith-Waterman, running on a PC with an Intel Pentium 4

processor at 3.0 GHz. This figure applies to both amino acid and nucleotide sequences, since the size of the substitution table does not have a significant impact on performance with PC hardware.

As Figures 2.11 - 2.18 show, the performance of our system varies greatly between sequence types, configurations, and sequence lengths, unlike the PC-based system. The reasons for this are explained in Section 2.8.3 and summarized in Table 2.4. Because of this, we will consider the amino acid and nucleotide cases separately. By comparing the average performance of our system against the 45 MCUPS reference, we see that our system achieves speed-up ranging from approximately 10 to 30 for amino acid sequences and 20 to 100 for nucleotide sequences.

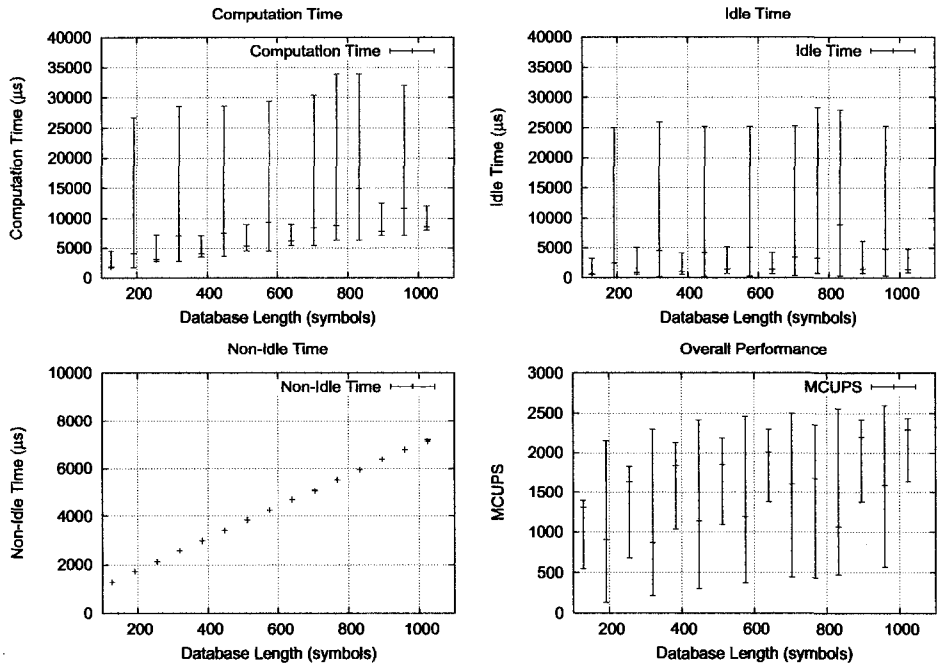

Figure 2.11: Measurements for a nucleotide alphabet system with 64 PEs, 12 bit score width, 128-length query sequences, and $f_{clk} = 46$ MHz.

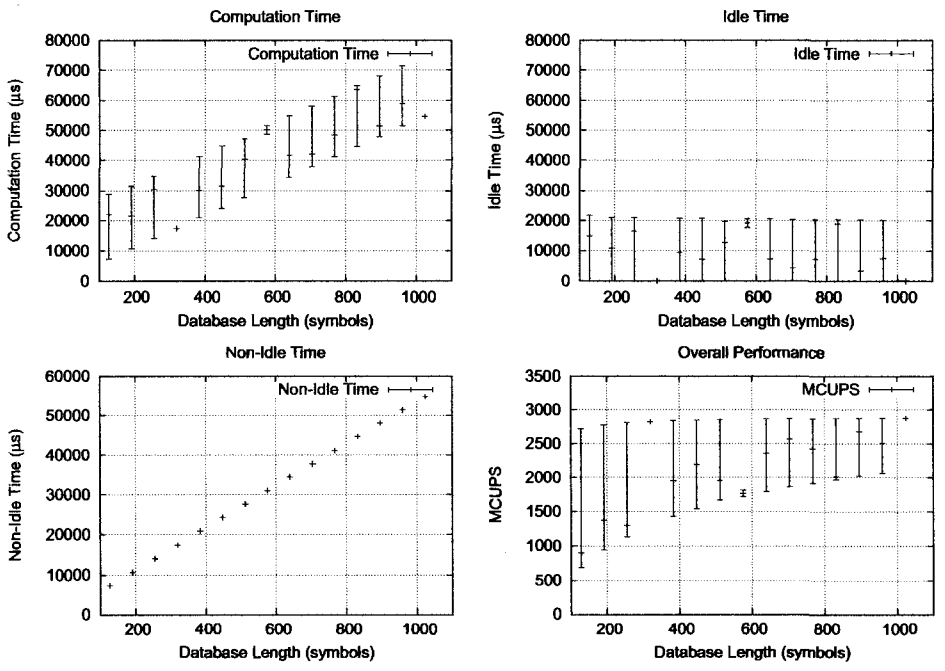

Figure 2.12: Measurements for a nucleotide alphabet system with 64 PEs, 12-bit score width, and 1024-length query sequences.

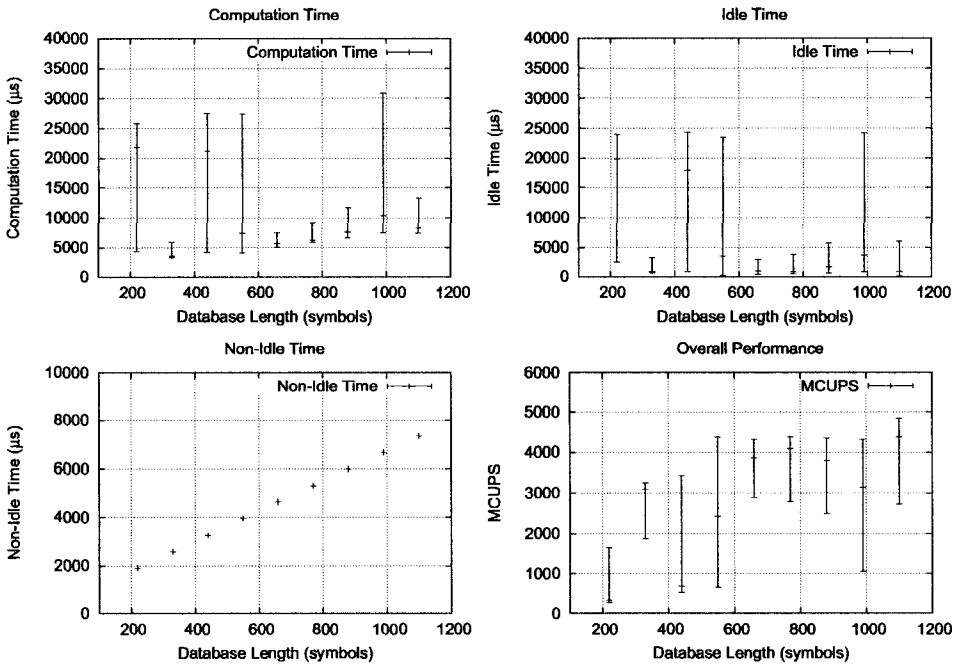

Figure 2.13: 110 PE nucleotide system with 7-bit scores and 220 symbol query sequences.

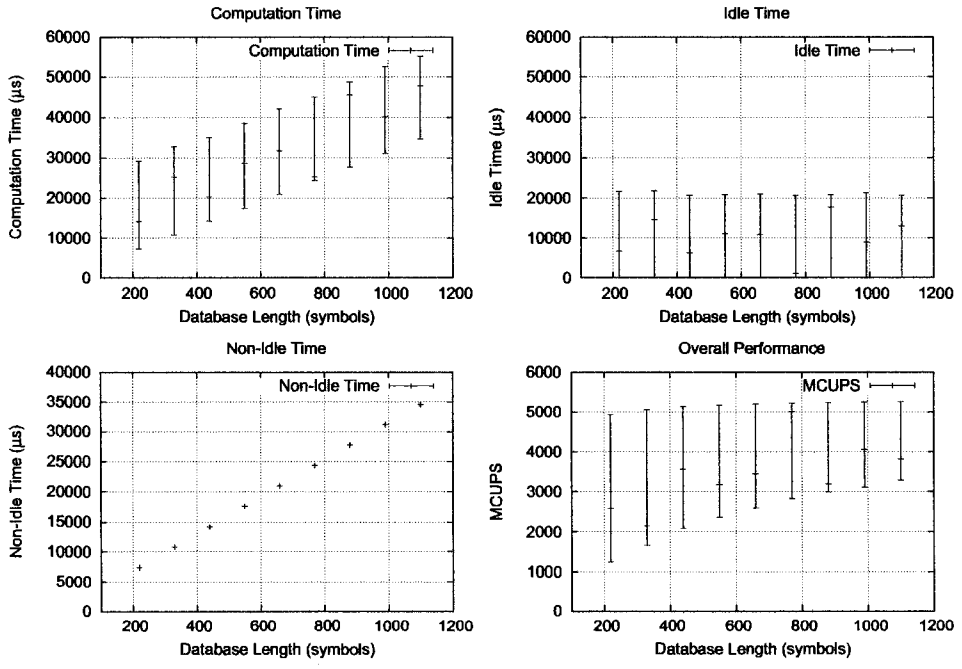

Figure 2.14: 110 PE nucleotide system with 7-bit scores and 1100 symbol query sequences.

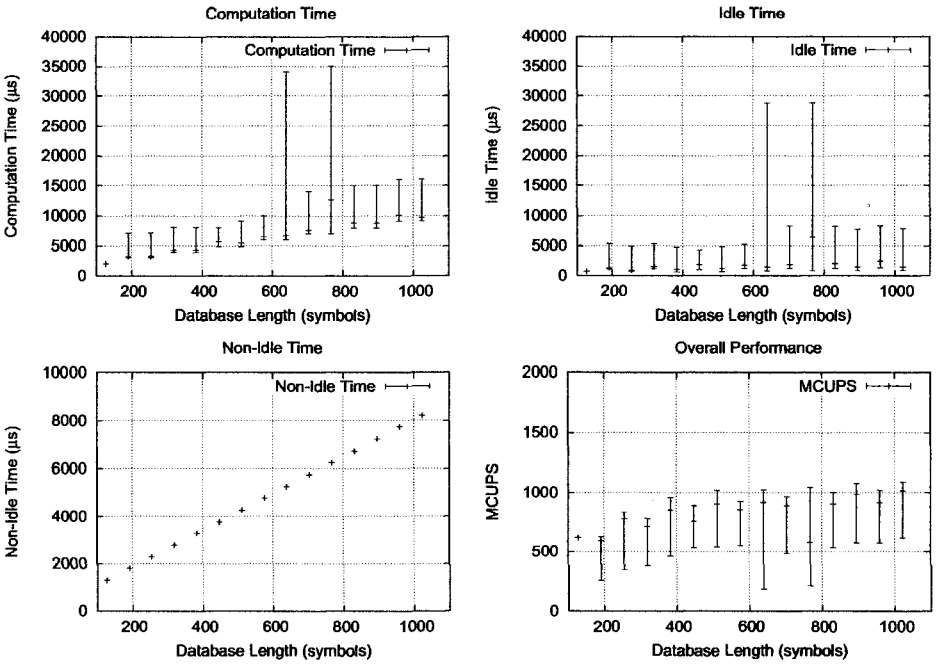

Figure 2.15: 32 PE amino acid system with 12-bit scores and 64 symbol query sequences.

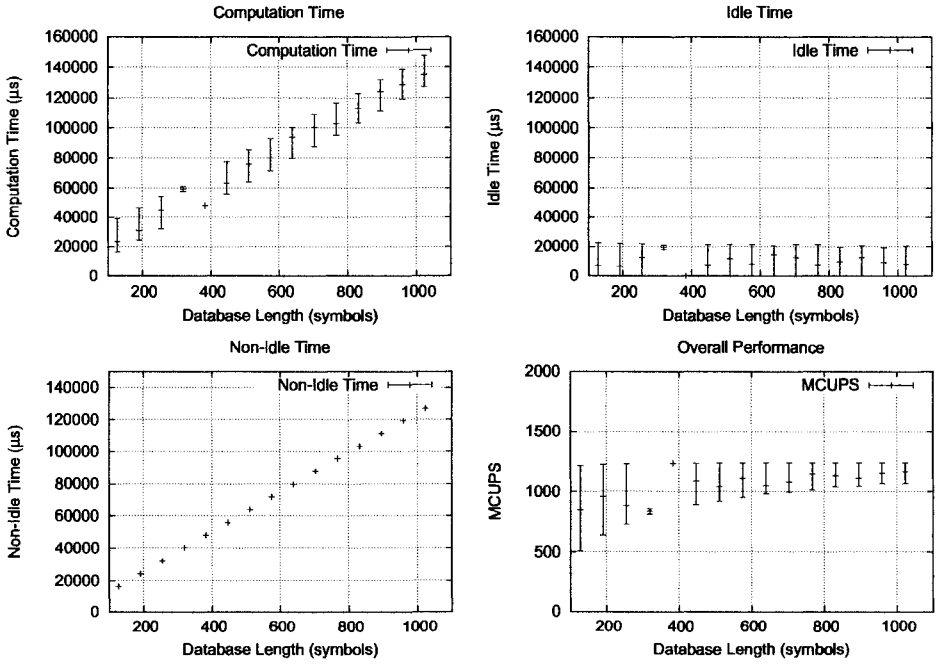

Figure 2.16: 32 PE amino acid system with 12-bit scores and 1024 symbol query sequences.

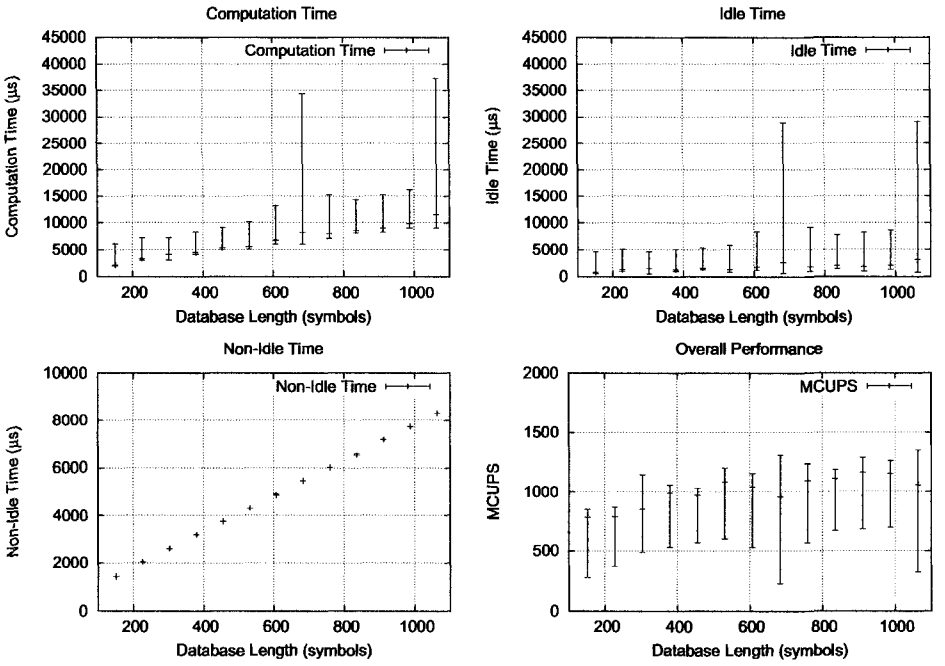

Figure 2.17: 38 PE amino acid system with 7-bit scores and 76 symbol query sequences.

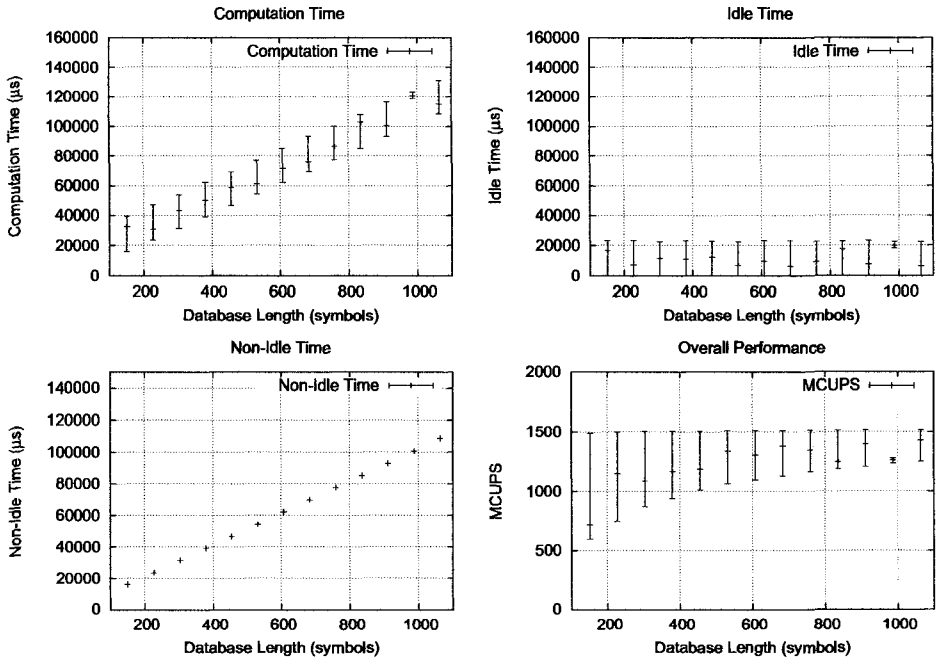

Figure 2.18: 38 PE amino acid system with 7-bit scores and 1026 symbol query sequences.

Chapter 3

BLAST Filter

BLAST, or *Basic Local Alignment Search Tool*, is a sequence alignment algorithm like Smith-Waterman. Unlike Smith-Waterman, which is guaranteed to find the optimal subsequence alignment, BLAST is a heuristic algorithm that aims to find the best balance between sensitivity and computational cost [22]. Research is currently underway to develop a partial implementation of the NCBI BLAST algorithm on the same Spartan-3 based integration boards as our Smith-Waterman design.

However, because the proposed BLAST design is too large to fit on the platform we used for our Smith-Waterman system, the Spartan-3 XC3S1500 FPGA, we have shifted our target FPGA to the Spartan-3 XC3S4000.

Because some aspects of modern gapped BLAST are impractical to implement in a hardware-accelerated design, we have instead devised a partial implementation that aims only to identify word hits likely to lead to successful extensions as defined in the original algorithm. By simplifying the BLAST extension technique and two-hit criterion, we obtain an algorithm highly adaptable to hardware that is capable of vastly higher computational rates than a full gapped BLAST implementation [23]. Despite these simplifications, the analysis in [23] and simulation results shown Section 3.6 demonstrate that with simple post-processing, our algorithm perfectly replicates the results of NCBI BLAST software.

Since our design does not attempt to implement BLAST in its entirety, we call this design a *BLAST filter* since it filters out the vast majority of non-similar sequences. The sequences which pass the filter would then be passed on for post-

processing, which combines redundant results before presenting them to the end user.

3.1 Background Information

BLAST was originally devised as an improvement to the earlier FASTA algorithm, with which it shares many similarities [24]. The original BLAST algorithm in [2] uses a two-stage process. In the first stage, a word of length W is taken from the database sequence. The query sequence is scanned for words (also of length W) that are sufficiently similar to the database word - these matches are called *hits* or *seeds*. The method used to calculate this similarity is direct comparison - the alignment scores of each symbol pair (Q_j, D_j) are added, and if the sum exceeds a specified threshold, the match is considered a hit. Hits are passed to the second stage, in which the algorithm attempts to extend the hit in both directions, to determine whether the hit was part of a higher-scoring subsequence match. The extension in one direction continues until the running alignment's score drops a certain parameter D below the maximum score yet obtained, and then the extension is continued in the other direction. D is called the drop-off threshold. The database word is then advanced one symbol, and thus the algorithm iterates through the database sequence until the end is reached.

A revised version of BLAST adds a two-hit criterion that must be satisfied before extension on a hit is attempted [22]. The reason for this is that extensions are very slow compared to the hit detection stage, accounting for over 90% of the total processing time. Furthermore, many hits occur in isolation, and do not lead to successful extensions. The two-hit criterion overcomes these deficiencies by passing a hit for extension only if a second hit has occurred on the same diagonal, the two hits are within A symbols of one another, and they do not overlap. Furthermore, the hit must not have been covered by the extension of a previously extended hit.

There are also numerous extension algorithms that can be used with BLAST. Smith-Waterman is possible, but is rarely used in practice, because its exhaustive nature makes it very slow and this exacerbates the extension bottleneck. The

ungapped extension algorithm simply compares corresponding symbols from the two sequences and adds their similarity score to a running total, like the algorithm used to detect hits. In [22], a heuristic extension algorithm allowing gaps (gapped BLAST) is specified.

3.2 Initial Research

A literal hardware implementation of BLAST is very challenging, since some key operations do not translate well to hardware. The best example of this is the extension, which can continue for an arbitrary length in either direction, causing long feedback loops and datapath control problems. Furthermore, the extension in the second direction must begin with the maximum score attained in the first direction, which makes effective parallelization impossible. In addition, hit detection tends to produce data in bursts - one database word may not occur anywhere in the query, while another could occur many times in succession, overwhelming the downstream components, and necessitating a complex system of queues and stalls.

Preliminary research on this project, undertaken in [23], demonstrates that a partial implementation of BLAST, with simplifications to render it more hardware-friendly, can act as a very high-speed filter to remove most non-extendable hits from consideration. The proposed low-complexity system is thus called a BLAST filter, and is intended to be used with software tools for pre- and post-processing to serve as a complete implementation of BLAST.

It is estimated that the proposed system could compute alignments at the equivalent rate of 100 GCUPS on our platform. This figure was estimated by assuming an average of 1 word (and thus 1 symbol) aligned per clock cycle, at a clock frequency of 100 MHz, against a maximum query size of 1024 symbols. This restriction on the query length is necessary to keep the average number of hits per word down to a manageable amount.

In addition to this performance goal, we set the requirement that no extensions found by the complete NCBI BLAST software escape detection by our system. A small reduction in selectivity is tolerable. Simulations of our hardware described

in Section 3.6 show that our system achieves equal sensitivity and, with a simple post-processing step to eliminate redundant results, equal selectivity. We use the NCBI BLAST algorithm as our guideline, which is available in source and binary forms from their website at [25]. NCBI BLAST adheres closely to [22] but offers more configuration options that are not mentioned in the original paper. We will pursue an implementation with the default parameters using the ungapped extension algorithm.

3.3 Scope of this Work

The research presented in this document is limited the hardware implementation of our simplified BLAST, and not the development of the algorithm itself. The original reference paper for the algorithm, [23], was co-authored by the commercial sponsors of this work, and is thus proprietary and not available for publication or viewing by third parties. However, the information salient to implementation, validation, and evaluation of the algorithm is included in this document. Where necessary, the aspects of the algorithm are described and experimentally validated, but it is stressed that this work does not include the development of the algorithm.

3.4 Comparison with NCBI BLAST

The BLAST algorithm described in this work is actually a subset of the BLAST algorithm maintained by NCBI. The portions of NCBI BLAST that would require a complex hardware implementation are omitted or reduced in scope by fixing the values of variable parameters or introducing hard maximum limits for others. This section describes the differences between NCBI BLAST and our algorithm, in order of processing. A table summarizing these differences can be found in Table 3.4.

In the hit detection phase, the first difference is that our algorithm has a fixed word size of $W = 3$, while BLAST allows arbitrary word sizes. The threshold for hit detection is variable in both, though the range of values in our system is restricted by the number of bits used in arithmetic operations. Next, our algorithm

Table 3.1: Comparison between NCBI BLAST and our algorithm.

Our Algorithm (BLAST subset)	NCBI (full) BLAST
Parameters for word size and 2-hit filter maximum separation are fixed at the default NCBI BLAST values ($W = 3$ and $A = 40$ respectively).	These parameters are variable.
Threshold score for hit detection is fixed at the default value (11).	Threshold score can be varied.
The 2-hit filter passes hits that were covered by extensions of previous hits.	Hits that were covered by a previous extension are discarded.
Extensions are performed using the ungapped extension algorithm.	A variety of extension algorithms are available (both gapped and ungapped).
Extensions continue for 50 symbols from the boundary of the original hit, or until the end of the sequence is reached.	Extensions end if the sequence end is reached, but otherwise may continue for an arbitrary number of symbols.
Maximum score is not passed from an extension in one direction to the extension in the other direction.	The extension in the other direction is initialized with the maximum score achieved in the first direction.

assumes a maximum of 63 hits for any given word, while BLAST can handle an arbitrary number of hits per word.

In the two-hit filter phase, our system uses applies a fixed value of $A = 40$ for the maximum distance between two hits. This value can be adjusted arbitrarily in BLAST. Furthermore, our algorithm ignores the criterion that a hit cannot have been contained within the extension of any previous hit.

Finally, in the extension phase, the window size of our algorithm is fixed at $N = 50$, although it can easily be reduced or increased as high as $N = 66$. In full BLAST, the extension window size is not defined, as extensions continue until either the end of the sequence or until the drop-off criterion is met. Furthermore, our extension algorithm executes the extension in both directions simultaneously, using initial scores equal to the similarity score of the original word. The maximum score attained in one direction does not carry over to the other direction as it does in BLAST.

3.5 Comparison with Previous Designs

Previous work on implementing BLAST in hardware has focused on accelerating individual stages of the algorithm, or producing a partial implementation rather than attempting a complete implementation. For instance, [26] implements a parallel hit scanning system on an FPGA intended to operate alongside a host PC. Hits are passed back to the host PC, where the remaining stages of the algorithm are performed in software. There are also similar systems that run on hybrid systems: PCs or workstations with reconfigurable platforms deeply integrated. The *Mercury* system is popular for these applications; two examples using this system are presented in [27] and [28]. These systems implement hit detection in hardware, then pass the results to software for the extension stages. Comparisons between these systems and ours are not generally applicable, since our system requires hit locations to be pre-computed and stored in memory. However, our design could benefit through integration with one of these systems, as it would make pre-computation of hits unnecessary.

Another design, presented in [29], uses a massively parallel architecture and performs hit extension as well as detection. Each parallel processing unit contains a hit scanner, a hit extender, and a local copy of the query sequence. Words from the database sequence are directed to an idle unit, where the unit scans for hits and attempts extensions on those hits. However, it can only perform BLAST on nucleotide sequences and does not implement the 2-hit criterion described in [22]. Our design can perform BLAST only on amino acid sequences and does implement the 2-hit criterion. The designs also differ in that only a portion of the datapath in our design - namely, the 2-hit filter section - is parallel. Our design has only a single unit for hit look-up and hit extension.

Another approach can be seen in [30], which implements a BLAST-like string matching algorithm on an FPGA. Although it is not a complete implementation of BLAST, this design implements a similar algorithm that first looks for matches of shorter substrings, then attempts to extend them. We have chosen to take a simi-

lar approach, applying simplifications to the algorithm that allow us to implement all stages of the simplified BLAST in hardware. Both our design and [30] are essentially single-pass BLAST approximations, though they differ in that [30] uses stream scanning to identify hits, which requires substitution matrices and logic, while our design is based on look-up tables and requires hit locations to be precomputed. In addition, [30] does not implement the two-hit filter rule.

Overall, our design presents several original concepts. The two table design for looking up hits is more efficient than a single table design, as it allows memory savings by taking advantage of overlapping and redundant sets of hits. It is the first architecture to implement a 2-hit filter. The datapath design of a single hit source outputting to multiple parallel hit filters, which in turn output their results to a single extension unit, is a completely novel approach. Finally, like our Smith-Waterman design, our BLAST design is distinguished by running on a portable, low-cost platform.

3.6 Simulation and Validation

In order to test the validity and performance of our algorithm and BLAST filter hardware, we created a software model of the hardware written in C. This software model performs a cycle-by-cycle simulation of the proposed hardware blocks. This section presents our findings for three different criteria: conformity of results to NCBI BLAST, performance and duty cycle, and the false positive rate of our two-hit filter design.

All of these simulations used randomly-generated sequences. Because amino acids appear with varying frequencies in actual proteins, our generated sequences use the same proportions as found in the UniprotKB/Swiss-Prot database. These statistics were obtained from the UniprotKB/Swiss-Prot database release notes [31].

3.6.1 Equivalence with NCBI BLAST

The most important design goal of the BLAST filter is that the results it returns match the results returned by BLAST software as closely as possible. In particular,

there must be no loss in sensitivity - every region of similarity found by BLAST must have a corresponding result in our system. However, we can tolerate a loss of selectivity, meaning that the BLAST filter may return more results than BLAST software, including ones that are superfluous or insignificant. These results can easily be removed in post-processing, while a missed result is much more costly.

To verify that our system will perform satisfactorily, we ran alignments on 100 different sequence pairs on both NCBI BLAST and our software model. BLAST is public domain software and can be downloaded in source and binary forms from the NCBI website [25]. For these tests, we used the version 2.2.17 binary, running the `bl2seq` executable. This program uses the `blastp` algorithm for amino acid sequences.

All query sequences were length 1024, the maximum supported by our system. Database sequences varied between 100 and 1000 symbols in length. Sequences were randomly generated, except for a segment of random length between 10 and 100 that was rigged to be identical in both sequences. This was to guarantee at least one successful extension per test. The tests used a word size of 3, BLOSUM62 scoring matrices, a hit threshold score of 11, a maximum two-hit window of 40, and ungapped extension.

In all tests, our system and the NCBI software reported equivalent results. Our system, however, returned numerous redundant records for each region of similarity, whereas the NCBI software reported each region only once. The reason for this is that a full implementation of BLAST will not attempt to extend hits that fall within the extension of any previous hit. Due to the difficulty of implementing this rule in hardware, our system simply ignores it. The result is that every similar region is reported multiple times. Since our system does check that hits are non-overlapping, a similar region of length L is reported approximately $\frac{L}{3}$ times, as shown in Figure 3.1. Since these redundant results are trivial to remove in post-processing, there is no real loss of selectivity to the end user.

Despite the duplicate reports, our system generally agreed with the BLAST software on the score, length, and boundaries of the similar regions. Two conditions

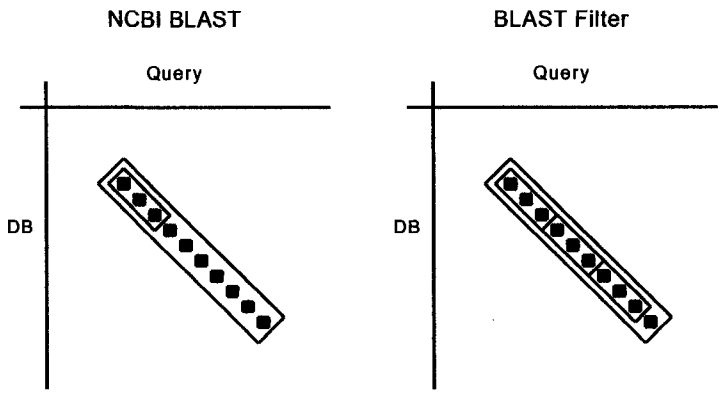

Figure 3.1: The BLAST filter reports separate results for every word in a region of similarity, while the NCBI software reports it only once. The inner boxes show the original hit boundaries.

were observed in which the results differed. The first is in long regions of similarity, specifically those longer than 53 symbols. Because our system has a maximum extension window of 50 symbols, it stops once this maximum is reached and reports the result. In other words, a single record will only cover up to 50 symbols from the boundary of the original 3-symbol hit. In software, extensions may continue indefinitely. However, the multiple hits reported by our system can be stitched together in post-processing to obtain a single record covering the entire similar region.

The second condition occurs when a hit contains one dissimilar symbol pair, but the other two symbol pairs score highly enough to put the hit over the detection threshold. The result is that the dissimilar pair is included in the reported result, giving it a lower score and length longer by 1 symbol than the correct BLAST result, which filters out the negatively-scoring pair. Again, these are very easy to remove in post-processing, so this is not a major problem.

Based on these results, we can conclude that with a post-processing step, the BLAST filter is every bit as accurate as ungapped BLAST running in software, and thus that our design meets our goals for sensitivity and selectivity.

3.6.2 Performance

The maximum performance goal of 100 GCUPS for our BLAST filter is based on 1 database symbol per clock being aligned over a 1024 symbol query sequence at a clock frequency of 100 MHz. However, this figure assumes that no stalls will occur - an unrealistic assumption, since any word with over 4 hits will result in stalls as the location LUT processes them at a rate of 4 per cycle. Furthermore, the 2-hit filters and extender have queues that will raise stalls when full.

To test the true performance of our system, the software model was ran 1000 times with random sequences, and the number and source of stalls was recorded. The query length for these tests was 1024 and the database length 1000. Long queries represent the worst case in terms of number of stalls raised, because each word will have more hits on average, which require more stalls to process.

On average, each sequence incurred 94 stalls. Over 1000 symbols, this gives our system a duty cycle of 0.914. Thus, to achieve performance of 100 GCUPS with 1024-length queries, the clock frequency must be 107 MHz or greater. This speed is realistically attainable with our hardware design, provided it is aggressively pipelined.

The vast majority of stalls (98%) were raised by the location LUT. This was to be expected, since any word with more than 4 hits will cause this block to stall, and words with many hits may even cause a multi-cycle stall (see Section 3.7.2 for full details). The remainder were raised by the 2-hit filter queue. The extender queue never raised a single stall throughout these tests.

Consequently, we could raise the duty cycle of our system by having the location LUT process more hits in parallel, and having more 2-hit filters to process these hits, although a platform with more RAM resources would be required to do this.

3.6.3 Two-Hit Filter False Positives

One final characteristic of our system validated with the software model is the rate of hits falsely validated by the 2-hit filter. Because the database sequence length is unbounded, the diagonal array used to keep track of previous hits must be infinitely

deep in a perfect filter. To create a practical filter, we only store the bottom 12 bits of the database index, creating an array 4096 elements deep. For database sequences longer than this, there is a potential for aliasing resulting in a false positive. See Section 3.7.3 for a full explanation.

Note that for the purposes of this section, a false positive refers only to hits falsely validated due to aliasing. Hits that are within the extension of a previous hit and pass through the filter are not considered false positives.

A rough estimation of the proportion of hits validated by the 2-hit filter that are false positives can be obtained by considering an infinitely long database sequence. When a hit enters the 2-hit filter, it is falsely validated if any of the three previous hits on the same diagonal satisfy the condition $4096 \times k - L \leq J \leq 4096 \times k + L$. We can assume the probability of false positive events in which $k > 1$ is negligible, simplifying the above equation to $4096 - L \leq J \leq 4096 + L$. If we model hits as a Poisson process, the probability of at least one of the three previous hits meeting this condition is approximately:

$$P \approx 2\lambda L \left(1 + 4096\lambda + \frac{(4096\lambda)^2}{2}\right) e^{-4096\lambda} \tag{3.1}$$

Where λ is the rate of hits. Our simulations showed that on average, each word produces approximately 3 hits in a 1024-length query sequence, so $\lambda \approx 3/1024$. The above equation thus evaluates to approximately $1.26 \cdot 10^{-3}$, or about 1 hit in every 792. Experimental results for a variety of database sequence lengths are summarized in Table 3.6.3. $P_{fptotal}$ is defined as the number of false positives as a proportion of all hits, while $P_{fpvalid}$ is defined as the number of false positives as a proportion of hits that pass the 2-hit filter.

These results show that our estimate for the false positive rate was accurate for long sequences. The false positive rate is lower in shorter sequences because there is zero probability of a false positive occurring in the first 4096 symbols of the database sequence. As a proportion of validated hits, the false positive rate plateaus at about 1 in 50.

Table 3.2: Simulated 2-hit filter false positive rates for a variety of database lengths.

Database length	$P_{fptotal}$	$P_{fpvalid}$
< 4096	0	0
5000	$4.97 \cdot 10^{-4}$	$8.69 \cdot 10^{-3}$
10000	$1.07 \cdot 10^{-3}$	0.0184
15000	$1.05 \cdot 10^{-3}$	0.0180
20000	$1.13 \cdot 10^{-3}$	0.0193
25000	$1.24 \cdot 10^{-3}$	0.0211
30000	$1.28 \cdot 10^{-3}$	0.0217
35000	$1.26 \cdot 10^{-3}$	0.0214

In order to determine whether false positives impacted the speed or accuracy of our design, we repeated the tests in Sections 3.6.1 and 3.6.2 with database sequences of length 100000. There was no difference in the results, meaning that false positives do not occur at a rate sufficient to create a bottleneck at the extender, and no false positive hit ever lead to a successful extension that was not also covered by a legitimate extension.

As a practical matter, false positives are of little concern with our system. One reason for this is that the vast majority (99.7%) of the sequences in the UniProtKB/Swiss-Prot database are shorter than 2500 amino acids [31], and so will never lead to a false positive. In addition, as the simulations mentioned above show, our system can easily deal with the false positives generated by sequences much longer than even the longest database sequences - the longest sequence in UniProtKB/Swiss-Prot is 34350 amino acids in length. Finally, unless the threshold score for a successful extension is set very low, the extension of any falsely validated hit will also be covered by a legitimately validated extension, so a falsely validated hit appearing in the final results is an extremely unlikely event. It is worth noting that this never occurred in any of our simulations, although it remains theoretically possible.

3.7 Implementation

Based on our experiences with Smith-Waterman, we decided to develop BLAST with decentralized, block-level control rather than with a single central controller. The Smith-Waterman controller ended up becoming unexpectedly complex, which reduced the clock frequency as well as made it more difficult to design, test, and debug. Besides avoiding these implementation headaches, this decision was primarily motivated by the speed goal of 100 GCUPS. This goal would be highly difficult or impossible to obtain with a complex central unit.

Another design feature arising from the need for a high clock frequency is the division of the system into simple blocks, separated by pipeline registers. Inter-block flow control is achieved with data valid and stall signals wherever they are needed. Valid signals move forward through the pipeline, signalling to the next block whether or not the current set of results is valid. Stalls propagate backwards, indicating to prior blocks that the system is swamped and a stall is needed to prevent data from being overwritten. Obviously we wish to avoid stalls as much as possible; as such, blocks where bottlenecks can occur are equipped with queues capable of handling a temporary surplus of data. Note that this valid/stall control scheme eliminates the need for global pausing. If the system is awaiting more data, the first block simply de-asserts the valid signal.

The individual hardware blocks will now be described, beginning with the first blocks in the datapath.

3.7.1 Sequence Unpacker

The datapath begins at the sequence unpacker, which is responsible for unpacking words from the input pipe and outputting individual symbols. Only the database sequence is handled this way; the query sequence is not needed until much later in the datapath. These symbols are loaded into a shift register to create the current database word. The word size W is fixed at 3, the default value for amino acids. The unpacker also counts the database position of the current word and forwards it

onward.

Given the database word, we must first find the location of all the corresponding hits in the query sequence. This is done with a lookup table, with the word itself used to calculate the index. Given the numerical representations of the word symbols $S_2S_1S_0$, the index is calculated by $S_2 \times 24^2 + S_1 \times 24 + S_0$. The reason for using 24 as a radix, rather than 20 for the number of amino acids, is to simplify the multiplication logic. The number 24 and its square, 576, both have only two 1's in their binary representation, allowing us to multiply by these numbers very quickly. There are thus $24^3 = 13824$ total indices, requiring 14 bits.

Now is a good time to note that our system only supports amino acid alignments. The reason nucleotides are not supported is that the default wordlength for nucleotides is $W = 11$. Using the 2-table system described above would thus require indices up to $4^{11} = 4194304$ and lookup tables several megabytes in size. This is many times beyond the capacity of our Spartan-III FPGA. Consequently there are no plans to adapt our system to nucleotide sequences any time in the near future.

3.7.2 Look-Up Tables

Hit locations are pre-computed in software and loaded in the tables before the start of the alignment. This must be done for each new query sequence (note that hit locations do not depend on the database). This is similar to the design in [29], which also detects hits using a look-up table pre-computed from the query sequence. Because this system is intended for use in first-pass database scans against huge databases, this pre-processing step should not pose a significant delay.

Because our system now lacks a central entity for distributing data, both lookup tables are connected directly to the input pipe. To distinguish between datastreams meant for specific components, we have developed a SIRO module (serial in, random out). Upon reception of a unique command word, these modules begin loading data from the pipe serially. This continues until an end word is detected. The device then works in RAM mode. As in Smith-Waterman, command words use a reserved bit to ensure that they do not appear in data.

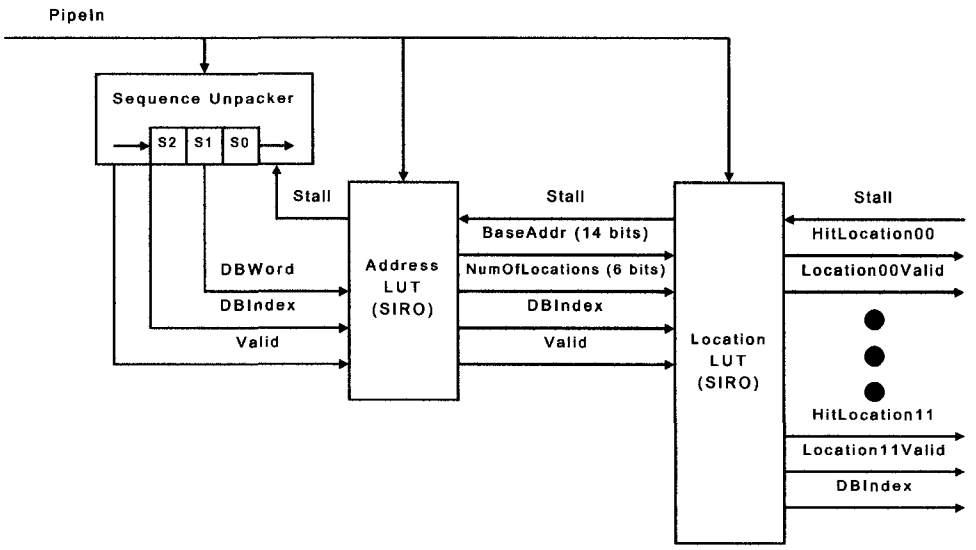

Figure 3.2: The origin of the BLAST filter datapath, showing the sequence unpacker and hit LUTs.

The reason for the 2-table configuration is that the number of hits from word to word is variable. As such, there is no way to look up a list of hits for a particular word in a single step without reserving a uniform maximum number of records for each word. This is obviously impractical, so instead we first look up a base address and number of hits from the first table, known as the address table. The actual locations are stored in the second table, which is called the location table. Empirically, it has been shown that the location table can be kept at a depth of $2^{14} = 16384$ by taking advantage of redundancies (for instance, overlapping hit sets between different words). Furthermore, tests with random sequence data show that the maximum number of hits in a 1024-length query rarely exceeds 31. We therefore impose an upper limit of $2^6 - 1 = 63$ hits, giving the address table output a width of 20 bits. Cases in which the complete hit set is too large or a single word has more than 63 hits must be handled by the pre-processing software. Hence, the total size of the address table is 13824×20 bits, while the location table is 16384×10 bits - the location table contains the actual locations of hits within the query sequence, hence the 10 bits of width.

In actuality, we split the location LUT is split into 4 parallel memories of $4096 \times$

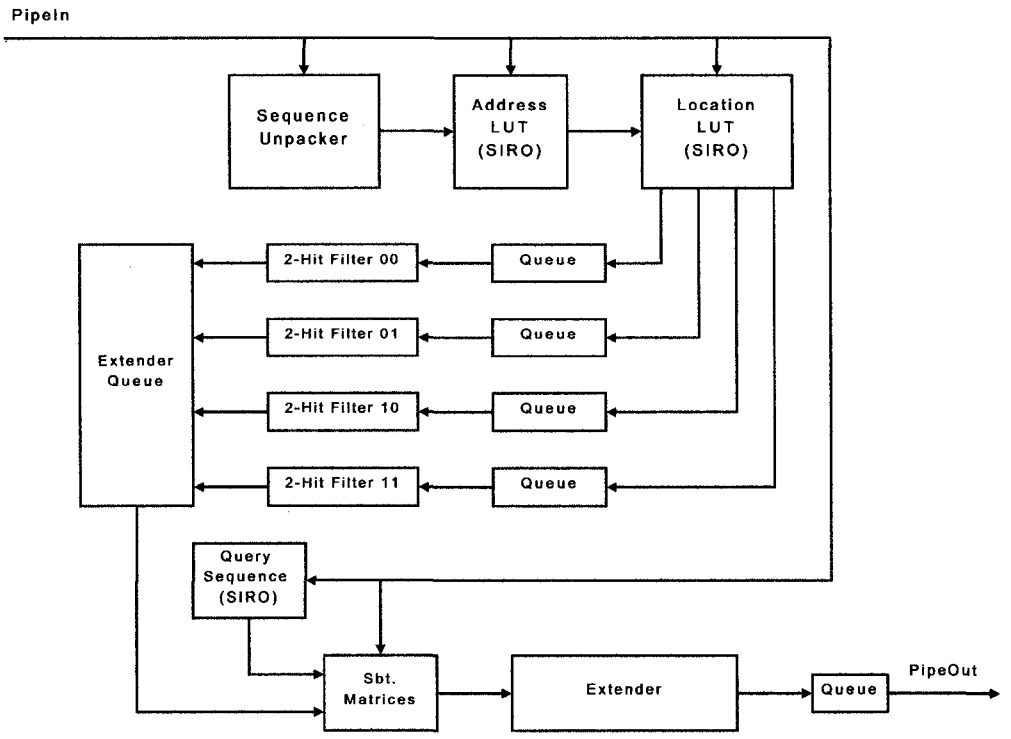

Figure 3.3: Simplified block diagram of the complete BLAST filter.

10 bits each. Thus, the location LUT can output up to 4 hit locations per clock cycle. The reason for this is maintaining throughput - if hits are processed serially, a serious bottleneck develops when a word has multiple hits. Analysis in [23] reveals that we can expect an average of 3 hits per cycle with random sequence data and a query length of 1024.

One memory contains the locations of hits with memory addresses ending in binary 00, the other three containing hits located at 01, 10, and 11 addresses. These memories are addressed simultaneously using the upper 12 bits of the output from the address LUT. However, the lower 2 bits are also input into this block, where they serve an important function. If we required each "row" across the four-memory table to contain only locations belonging to a particular hit, we could ignore the lower 2 bits entirely. However, this would make for very inefficient use of memory, as there would be unused cells if the number of locations was not a multiple of 4. Therefore to maximize efficiency, location sets are packed into the location LUT

without any explicit delimitation or other overhead. Sets can begin and end at any address. However, this introduces a new problem: how do we handle lookups that are split over 2 rows?

The key to the solution is that the lower 2 bits indicate the memory in which the set starts. We can therefore use a small logic section to increment the address presented to each memory as appropriate, placing a "line wrap" in the read from the table. For example, if the set began at an address ending in 01, the reads from the 01, 10, and 11 memories would use the upper 12 bits as supplied from the address LUT, but the 12 bits presented to the 00 memory would need to be incremented. This case is illustrated in Figure 3.4 - the memory locations being read are highlighted, with the 00 read taking place from one row further down from the others.

This arrangement allows us to always look up 4 locations per cycle. In the event of a hit yielding more than 4 locations, a stall is requested until the number of locations remaining to be looked up is 4 or less. In this case, the upper 12 bits of the address are incremented after each cycle to advance to the next row in the table. If the number of locations is not a perfect multiple of 4, the last read will produce invalid data in the uppermost hits. These are dealt with by using data valid signals, which are generated from combinational logic taking the lower 2 address bits and the number of locations as input.

A block diagram showing the sequence unpacker and lookup tables is shown in Figure 3.2. A more detailed diagram of the logical operation of the location LUT is shown in Figure 3.4.

3.7.3 Two-Hit Filter

The two-hit filter applies the BLAST two-hit criterion to incoming hits from the location LUT. If the criterion is satisfied, then the hit is passed on for extension. There are four filters working in parallel - one for each output quadrant from the LUT.

As stated in Section 3.1, the two-hit criterion is that a hit is extended only if a previous hit has occurred on the same diagonal, the two hits are within A symbols

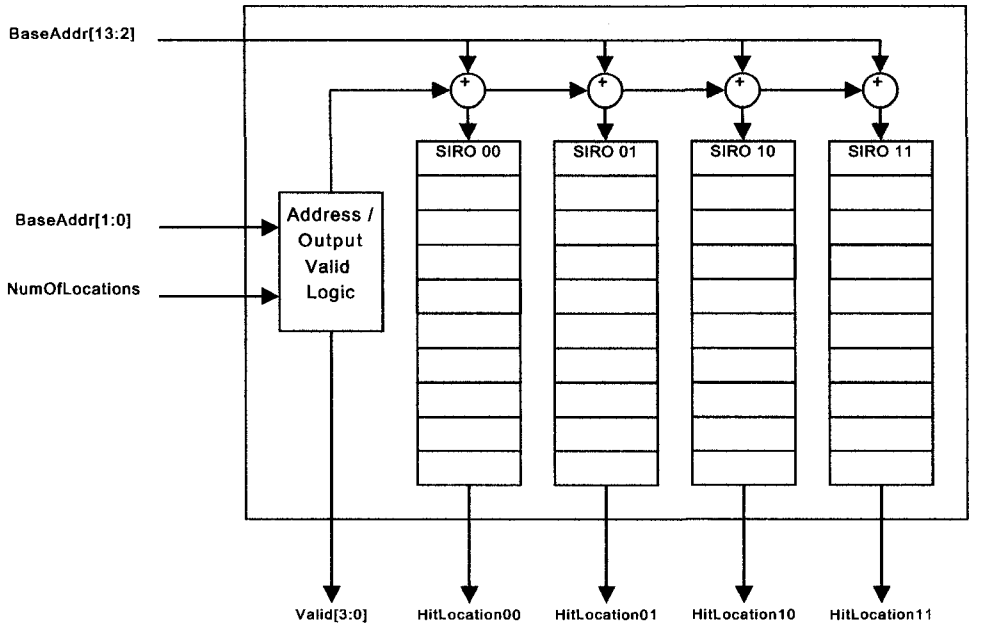

Figure 3.4: The BLAST location LUT.

of one another, they do not overlap, and the hit is not contained in the ungapped extension of any previously extended hit. The default value of A in NCBI BLAST is 40.

An exact hardware implementation of this step is problematic due to the need to check whether the current hit was covered by any previous extension. Applying the BLAST filter philosophy to the problem, we introduce our first heuristic to increase hardware-friendliness: pass the hit onward for extension if there is a hit within the group of last W hits on the same diagonal that is more than W and less than A symbols away. We choose to examine the W most recent hits to guarantee that a non-overlapping hit will be found if one exists - if successive database words lead to hits along the same diagonal, the $W - 1$ most recent hits will overlap, but the Wth will not.

By not searching for hits covered by past extensions, our two-hit filter passes approximately one out of every 10 hits as opposed to one out of every 20 for a complete NCBI two-hit detector. Hence, an average of approximately 0.3 hits per cycle are passed for extension. This will lead to some redundant hits in the output,

but the volume of data reaching the extender is reduced to a manageable level.

To simplify the hit detection logic, we describe hit locations using the diagonal co-ordinate and database position of the first symbol. To elaborate, given two hits at co-ordinates (j_1, i_1) and (j_2, i_2), where j is the position in the database and i the position in the query, they lie on the same diagonal iff $j_2 - i_2 = j_1 - i_1$. It is more convenient to express a hit position in terms of the database co-ordinate j and the diagonal co-ordinate $d = (j - i)|_{mod|Qs|}$, where $|Qs|$ is the size of the current query sequence. Using this co-ordinate system, the diagonal check is simply $d_1 = d_2$. Under this parametrization, given a hit at (j, d), our two-hit criterion is satisfied if there exists a hit (j', d') such that:

$$W < j - j' < L, \quad d = d' \tag{3.2}$$

One final issue that must be addressed is that the length of the database sequence is unbounded, and therefore so is the diagonal array used to keep track of the positions of previous hits. In order to reduce the depth of the array, we can store $J = j|_{mod2^k|Qs|}$ instead of j, where k is a positive integer. If $|Qs| = 1024$, J is the number represented by the lower $k + 10$ bits of j. Even for small values of k, $j = J$ for all but the longest database sequences. However, long databases can cause false positives. Analysis in Section 3.6 shows that for $k = 2$, we can expect a false positive rate of approximately 1 out of every 800 hits. From this we can conclude that the number of false positives allowed by this shortcut is insignificant.

The hardware implementation of the two-hit filter consists of three dual-port block RAMs and a combinational logic unit. RAM 1 contains the locations of the last hits along each diagonal, while RAM 2 and RAM 3 contain the second and third last hits, thus covering all W previous hits. An incoming hit indexes loads from these RAMs, producing the locations of the last 3 hits along the same diagonal. The combinational logic then checks if the two-hit criterion is fulfilled. If so, the hit is passed on to the next block. The hit is also written to the first RAM, as it is now the most recently observed hit along its particular diagonal. Likewise, the

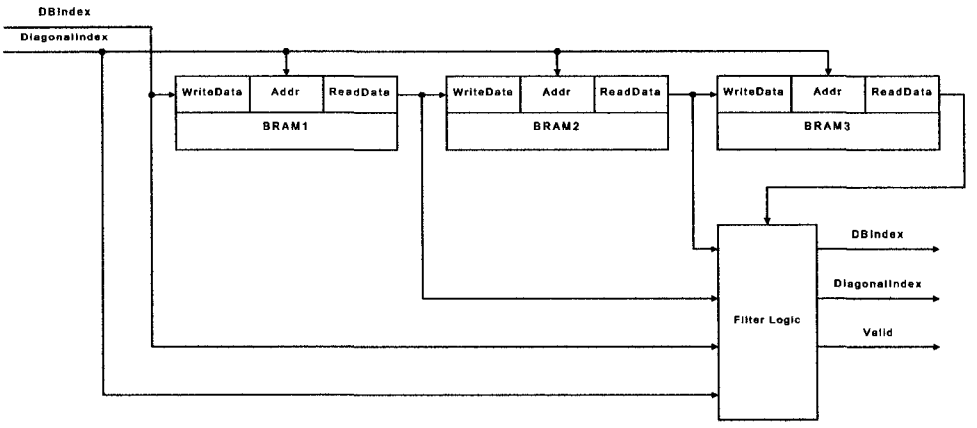

Figure 3.5: The BLAST two-hit filter module.

hits previously stored in RAM are shifted to the next RAM over, with the last hit overwritten. This entire operation can be executed in a single clock cycle by setting the RAMs in read-before-write mode. Note that this design does not contain any feedback loops, and thus can be pipelined if necessary to achieve our speed goal. Figure 3.5 shows a block diagram of a single 2-hit unit. Note that the complete system has four of these running in parallel.

In addition to the filter itself, an interface from the location LUT output is needed. The reason for this is that the two-hit filters partition the work according to which diagonal the hits lie on, while the hit locations coming from the location LUT are unsorted. In fact, it is possible for all 4 locations read in a given clock cycle to lie on the same diagonal, requiring them to be queued or to stall the system while a single filter services each hit sequentially. Obviously we would like to avoid the latter solution, so we have devised a two-hit filter queue that can accept up to 4 inputs in a single clock cycle, and multiplexes them to a single output.

The two-hit filter queue consists of 4 simple FIFO queues in parallel. The individual queues are read in a round robin fashion. If any queue is almost full, it asserts a stall signal and takes priority until the stall signal is de-asserted. Data from empty queues is flagged invalid. Figure 3.6 shows a block diagram of a single two-hit filter queue module. Each of the 4 two-hit filters has a separate queue, with all 4 outputs from the location LUT routed to each queue. Write control is accomplished

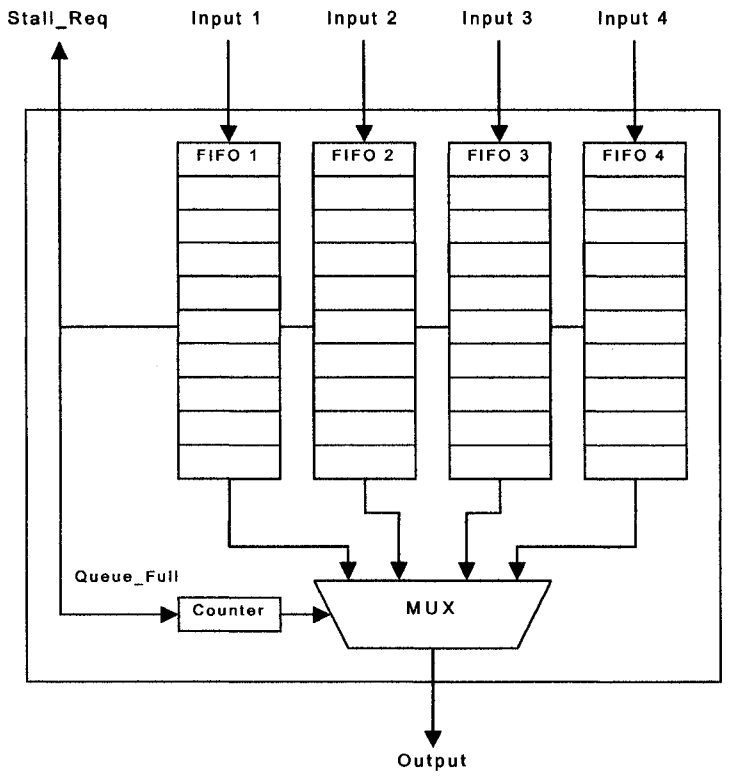

Figure 3.6: Simplified architecture of the two-hit filter queue.

by looking at the lower 2 bits of the incoming location - locations ending in 00 go to the 00 queue, locations ending in 01 go to the 01 queue, and so on. If incoming data is marked as invalid, it is simply not written.

Each of the 4 queues is implemented as a shift register look-up table (SRL). This design is much more efficient than dual-port memory. In Spartan-3 FPGAs, an SRL queue is equivalent to single-port distributed RAM in resource utilization [32], while dual-port memory consumes twice that [20].

These queues allow us to minimize stalls and maintain throughput. Since we can expect a roughly even distribution of hits across the 4 filters, as well as across the 4 individual FIFOs making up a queue, the probability of a stall is quite low even with short queues. A queue length of 16 is sufficient, though it should be possible to make them even shorter without significantly impacting system performance.

3.7.4 Extender

The extender is responsible for performing the ungapped extensions on hits that pass the two-hit filter. Like the two-hit filter before it, we have modified the extension algorithm in order to have a simpler and faster hardware implementation.

The NCBI BLAST ungapped extension algorithm first adjusts the boundary of the original hit to the highest scoring subsequence within the hit. Then, the left extension occurs until the running score drops a threshold D below the maximum observed score. Finally, the right extension is started using the running and maximum scores from the left extension as initial values. The extensions are performed sequentially, with drop-off threshold checks performed at every step. Also note that the left and right extensions are not independent of one another, and may continue for an arbitrary length. As mentioned earlier, these characteristics are highly inconvenient from a hardware perspective.

Our simplified ungapped extension algorithm performs the extension by exactly N steps in either direction, regardless of whether the drop-off condition is met. The right extension is initialized with the score of the original hit, rather than the running and maximum scores of the left extension. Thus the left and right extensions are independent. The extension is considered successful if the extension to either side reaches the end of the window without meeting the drop-off condition, or if the drop-off condition is met on both sides, the maximum observed score exceeds a certain threshold. Simulations in Section 3.6 have shown that any extension successful with the NCBI ungapped extension algorithm is also successful with our algorithm, although our algorithm produces redundant results. We will use $N = 50$ as a default.

Our extension algorithm can be implemented as a tree structure which completes the extension in $2log_2(N)$ steps. Figure 3.7 shows the extender architecture for a small ($N = 12$) window size. This is possible because in an ungapped extension algorithm, each symbol comparison is independent of all others, and the results can be merged together to obtain the final result. Consider the running scores for two subsequences in an extension, S_1 and S_2, and the maximal scores S_{1max} and

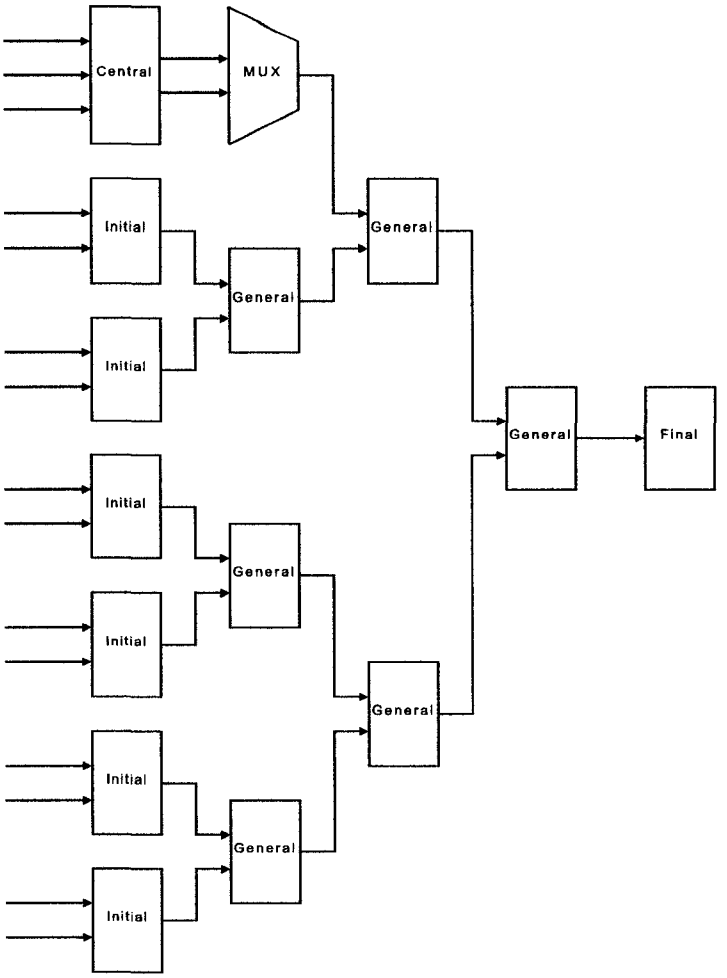

Figure 3.7: Extender architecture.

S_{2max}. If we concatenate the subsequences, the running score of the concatenation is obviously

$$S = S_1 + S_2 \tag{3.3}$$

The maximum score of the concatenation, as obtained by starting with sequence 1 and calculating through to the end of sequence 2, is

$$S_{max} = max(S_{1max}, f \times (S_{2max} + S1)) \tag{3.4}$$

The term f represents a drop-off flag. If the drop-off condition has been met, $f = 0$, and if it has not been met, $f = 1$. Recall that the drop-off condition is determined by $S_{max} - S > D$.

The extender is made up of several different types of unit, as shown in Figure 3.7. However, all units perform basically the same function of calculating scores and checking the drop-off condition. The central unit handles the original hit of 3 symbols and also takes the place of a 2nd level general unit. Two outputs are generated; one for a left extension and one for a right extension. The difference between the two is that the order of the symbols is reversed. Initial units do not have inputs for drop-off flags from previous units, since initially the drop-off condition is not met. General units do have these inputs. The final unit examines the result of the extension, and if the extension succeeded, sends the hit location to an output pipe queue along with the maximum and running scores. Pipeline registers are placed between each level of the tree. This allows a left extension to be started one cycle, and the right extension the next. Thus the extender can effectively compute a complete extension every 2 clock cycles.

It must be noted that the extender requires several more elements besides the extension tree. The first element of the extender is a queue that combines the data streams from all four of the two-hit filters back to a single stream. The exact same 4-to-1 queue used in front of the two-hit filters, described in Section 3.7.3 and Figure 3.6 can also be used to queue hits awaiting extension. As we have already determined that an average of 0.3 hits pass the two-hit filters per cycle, the extender will not be a long-term bottleneck. However, it is still possible to fill the queue and cause a stall in extreme cases.

Note also that the inputs to the extension tree are the similarity scores of every symbol within the extension window of both sequences. As a result, we need to retrieve the proper part of both the query and database sequences, and look up the scores for each pair of symbols. We now propose methods of doing so in a small, fixed number of cycles so as not to interfere with the rest of the system's operation.

The query sequence is fairly easily handled because it is not used in any other

block, unlike the database sequence. It can be stored in a RAM within the greater extender architecture and the appropriate segment retrieved as needed. In order to store the query sequence in such a way that any given segment of length $N + W$ can be quickly retrieved, we use five block RAMs of 256 depth and 72-bit width. These RAMs store an array of 256, 72 symbol subsequences. Each RAM stores a single bit of each of the 72 symbols, and as each amino acid symbol requires five bits, we have five RAMs.

As we have imposed a maximum query length of 1024, the memory depth of 256 allows us to store subsequences with a granularity of four symbols. Thus, address 0 would contain the sequence $[Q_0..Q_{71}]$, address 1 would contain $[Q_4..Q_{75}]$, and so on. After the sequence has been read from memory, it must then be shifted a maximum of three symbols to the left to obtain the desired segment. It is also necessary to reverse the order of the symbols depending on whether they are for a left or right extension.

With subsequence lengths of 72, this design supports extension window sizes N as large as 66. A 72-bit memory width was chosen because this results in the largest memory that will fit in a BRAM unit. All RAM units are placed in SIRO configurations so they can be easily loaded with the query sequence before the calculation begins.

Several methods for retrieving the database sequence are under consideration. The simplest is to relay the entire database window through the system pipeline, from the sequence unpacker all the way to the extender. This makes the entire subsequences for both extension directions immediately available. However, it is also highly costly in terms of resources. It requires 103 symbols, or 515 bits, to be registered at every pipeline stage, as well as in each queue entry. However, since the BLAST filter is expected to consume only about one fifth of the FPGA's fabric resources [23], we can set aside a large amount as distributed RAM, or register files. One quarter of the Spartan-3 XC3S4000 yields about 108 kilobits of single-port distributed RAM, enough for approximately 200 copies [20]. As long as queue lengths are limited, this design will easily fit within the device. It may also be

possible to conserve memory in queues using shift registers and hold/store logic, as successive database windows will be identical or contain a great deal of overlap. The relative simplicity and speed of this design make it the most desirable option.

Another proposal is to place database symbols in a long shift register, and multiplex the appropriate set of symbols into the extender upon request, using a very large crossbar switch. Yet another is to use a block RAM setup like the one used to retrieve the query sequence. Both use far fewer resources than the pipeline option, but add a large amount of complexity. This complexity arises from the variable number of clock cycles that pass between the issue of a database word and the time the resulting hits reach the extender.

Finally, once the appropriate subsequences are available, the similarity scores between each pair of symbols must be looked up. The proposed way of implementing the substitution matrix is with dual-port block RAMs. Each table requires 24×24 addresses and a depth of 5 bits, with the imposition that scores cannot exceed 31. This is not an issue with standard substitution matrices. The address into the RAM can simply be the concatenation of the two symbols. In dual-port mode, each RAM can handle two pairs of symbols at once, which reduces the number of RAMs needed and the initialization time. Arbitrary scoring matrices can be set by using the RAMs within SIRO modules.

3.7.5 Resource Consumption

Resource consumption was briefly discussed in Section 3.7.4. Here it will be analyzed in greater detail.

Nominally, the BLAST filter is a very memory-intensive system. Large RAMs are needed for the address LUT, location LUT, and query sequence. In fact, these components alone consume so much memory that they became the prime motivator for moving from the Spartan-3 XC3S1500 to the larger XC3S4000, which has triple the block RAM modules.

Table 3.3 shows estimated resource consumption of the BLAST system in both block RAMs and logic fabric (measured in configurable logic blocks, or CLBs).

Table 3.3: BLAST filter resource consumption estimates

	Usage estimation	Total available
CLBs (excluding queues)	1204	6912
CLBs (including queues)	4116	6912
Block RAMs	74	96

Note that the first estimate for CLB use does not take queues or pipeline registers into account. Due to the large distributed RAM consumption of the queues and pipeline registers, they will be considered and discussed separately.

Because we have exact knowledge of our block RAM requirements, we can very accurately predict the system's total block RAM consumption. Calculations in [23] estimate that the system will consume 74 out of 96 total block RAM modules. The estimate for CLB consumption is rougher, but we predict usage of about 1204 out of 6912 exclusively for logic functions. The distributed RAM used in queues and pipelining will consume much more.

Because our primary look-up table blocks use block RAMs and comparatively little supporting logic, we can afford to allocate a large proportion of the device's logic fabric as distributed RAM for the 2-hit filter queue, extender queue, and pipelining between blocks. Our system requires 20 queues of extremely large width: 515 bits to carry the database segment towards the extender, plus 26 bits for the database index, and 10 bits for the query index, making a total of 551 bits. If all our queues are made 16 entries deep for optimal CLB utilization [32], the total memory requirement is 176320 bits. At 64 bits per CLB for a shift register look-up table (SRL) based queue, this translates to 2755 CLBs, which we can easily spare.

Pipeline registers vary in width, depending on which blocks they are connecting. However, all registers before the extender must carry the 515-bit database, meaning any other data consumes a relatively insignificant amount of memory. If we conservatively allocate 10000 bits for pipelining, an additional 157 CLBs are consumed, based on a rate of 64 bits of registers per CLB [20]. This brings the total to 4116.

With these figures, we can conclude that the Spartan-3 XC3S4000 device can

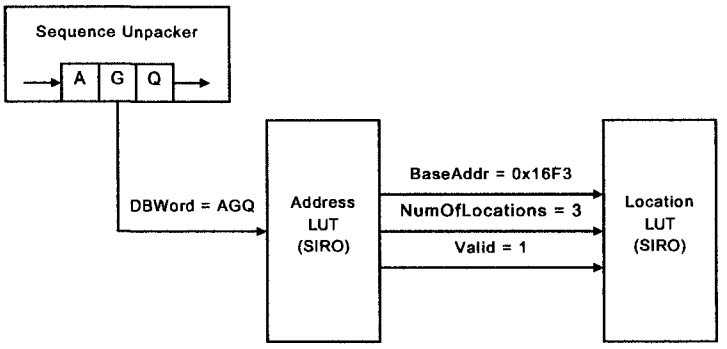

Figure 3.8: The word AGQ being processed by the Address LUT.

easily accommodate our BLAST filter design, with a large margin of safety for both logic and block RAMs.

3.8 Example

This section describes an example of the BLAST filter in operation to clarify the workings of each component. This example will cover a single database word occurring somewhere in the middle of a complete alignment, and its consequent hits as they travel through the 2-hit filter, extender, and finally reach the output.

The example begins with the word AGQ appearing at the database unpacker. This word is accompanied in parallel by its numerical index in the database sequence, which is used in later units. This word is passed to the index calculator, which indexes the word based on the lexicographical order of the standard amino acids - A is 0, G is 5, and Q is 13. Hence, the index of this word is $A \times 24^2 + G \times 24^1 + Q$, or 133. This index passes to the Address LUT to look up the base address and number of hits for this word stored in the Location LUT. The LUTs are pre-computed and loaded before the database scan begins, so let us assume that the Address LUT returns a base address of `0x16F3` and 3 hits. The operations of the unpacker and Address LUT are shown in Figure 3.8.

This address and hit count pass to the Location LUT, which stores the actual locations of the hits corresponding to the word AGQ. Since the base address ends in binary `11`, the base address is incremented by 1 to read the successive entries at

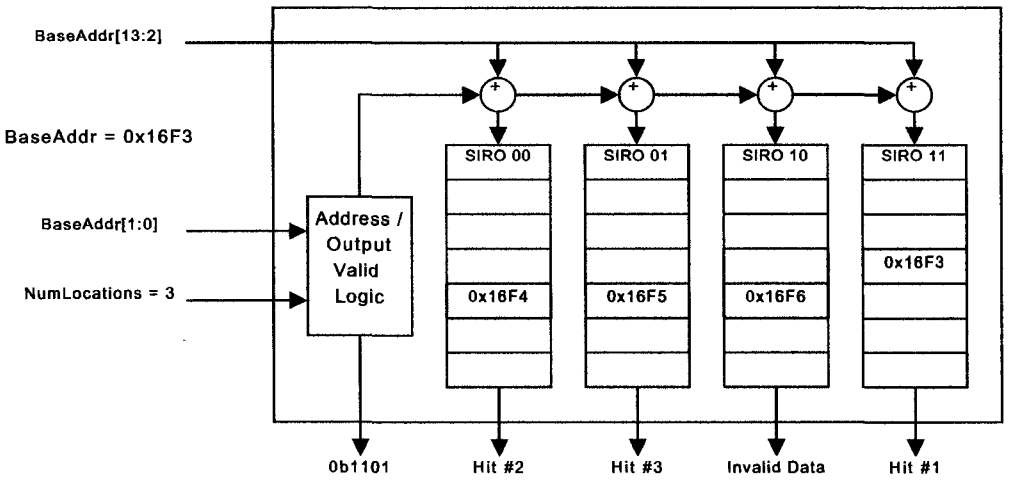

Figure 3.9: The base address and hit count from the Address LUT being used to look up hits from the Location LUT.

`0x16F4`, `0x16F5`, and `0x16F6`. Because there are only 3 hits, however, the hit loaded from `0x16F6` is flagged invalid - see Figure 3.9.

Next, these hits are input to the 2-hit filters. Each filter deals with one quadrant of diagonals - one for diagonal addresses ending in each of binary `00`, `01`, `10`, and `11`. A crossbar switch forwards each hit location to the appropriate filter by looking at the bottom 2 bits of their addresses. Two of the hits have diagonal co-ordinates ending in `00` - they are both written to queues belonging to the `00` filter. The third hit ends in `10` and so is written to that filter. The fourth hit, being invalid, is simply discarded.

Next, the 2-hit filters check each hit for fulfilment of the 2-hit criteria. The diagonal co-ordinate of each is used as an address into 3 RAMs; the data in each RAM corresponds to query co-ordinates of the previous 3 hits on that diagonal. To pass, a hit must be no more than 40 symbols away from a previous hit, and not overlap any previous hit. The `10` hit passes and is forwarded to the extender queue, but the other two hits fail. In all cases, the new hit is stored in memory to compare with future hits. For illustration purposes, let us assume the succeeding hit occurs on diagonal co-ordinate `0x0418` and database co-ordinate `0x0032`. Figure 3.10 shows this hit passing through the filter.

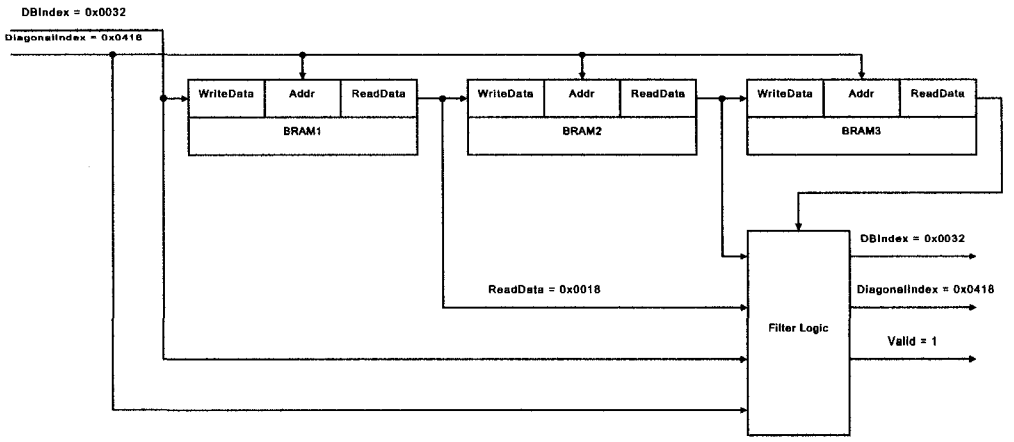

Figure 3.10: The hit on diagonal `0x0418` and database index `0x0032` passes because a previous hit on diagonal `0x0418` occurred at DB index `0x0018`, meaning they are `0x14` (20) symbols apart.

Once in the extender, the query co-ordinate of the hit is read. This is used to load the query segments needed for the extension. The database segment is already present, since it has been moving through the datapath alongside the hit in shadow registers. On the first cycle, the left extension is done, so the segment to the left of the hit is read and input into the score matrices along with the left section of the database sequence. The same thing is done for the right extension on the next cycle. The resulting scores filter through the extension tree, and result in a score of 162, which exceeds the threshold of interest for this alignment. The score and co-ordinates of the hit are passed on to the output unit, where they wait in a queue until read by the host.

Chapter 4

Common Components

This chapter describes the common components used in both the Smith-Waterman and BLAST projects. A common top-level interface was used to provide communications between the host PC and the alignment algorithm on chip. In addition, the hardware and vendor-provided API is also briefly described in this section, and their impact on the implementations of both algorithms is analyzed.

4.1 Host Interface

The platform used in our Smith-Waterman research, the Opal Kelly XEM3010-1500P development board, uses a Spartan-3 XC3S1500-4FG320 FPGA with fixed connections to a USB interface, SDRAM module, and PLL module. Our BLAST platform, the Opal Kelly XEM3050, is highly similar but contains a larger Spartan-3 XC3S4000-FG676. The top-level design entity is provided by the vendor. This section describes the interface used to connect the external host with the alignment logic.

A top level diagram of the host interface is shown in Figure 4.1. The USB Host encompasses several Opal Kelly endpoints, which are provided as pre-compiled HDL files. In VHDL, these components are instantiated with a generic specifying their address, which serves as a unique identifier. These identifiers are used in the host side to interact with the modules via a provided software API. The API is available on Windows, Mac OS X, and Linux platforms, giving our sys-

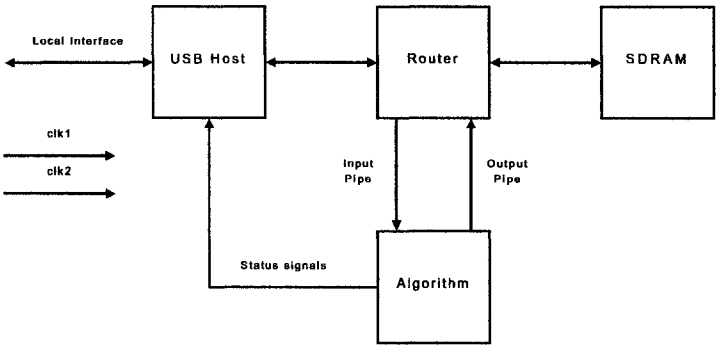

Figure 4.1: Top-level diagram of the host interface.

tem portability across the vast majority of PCs. A typical function call to the API might be `OKWriteToPipeIn(0x15, length, buf)`, which would write `length` bytes from `buf` to the PipeIn module at address `0x15`.

The Opal Kelly pipe interfaces, `OKBTPipeIn` and `OKBTPipeOut`, are used to provide bulk data transfers in and out of the system through the USB. These pipes have a hard-coded word size of 16 bits, hence many of our own components, such as the input and output FIFOs, use the same word size. The `OKTriggerIn` module is used to provide external triggers, which are used in our design for debugging purposes. The modules `OKWireIn` and `OKWireOut` provide virtual signals or variables that can be read from and written to by the host. These are used to enable or disable various components within the system, in the case of `OKWireIn`, and provide status signals to the host in the case of `OKWireOut`. For details on these components, the reader is directed to [33].

The router is used to direct datastreams between the USB host, SDRAM, and algorithm. Data from the input pipe can be directed to either the SDRAM or algorithm, while the output pipe always receives data from the algorithm. The router also contains counters for the number of words passed to the algorithm and SDRAM; these are used for debugging. The SDRAM module requires a controller, which is not currently implemented. However, future plans include using the SDRAM to enable even longer query sequence lengths in both Smith-Waterman and BLAST.

The algorithm module is either the Smith-Waterman or BLAST core. It uses a standard interface that both the Smith-Waterman and BLAST designs conform to. The purpose of designing our system in this way is so that the host and SDRAM interfaces would be fully interchangeable between the Smith-Waterman and BLAST configurations. The standard interface also permits future algorithms could also be "dropped in" to the algorithm component. Besides the input and output pipes, ports for various status signals are provided, such as a count of the number of results waiting to be read, and overflow reporting for the input and output FIFOs. These FIFO modules perform data buffering and clock domain crossing for the algorithm.

There are a total of three clock domains in this system: one for the USB host, one for the SDRAM, and one for the algorithm. However, since the USB host is provided by the vendor and has a fixed maximum clock frequency, the board provides a special maximum-frequency clock for it. Thus, we don't need to create our own clock for it. The SDRAM clock is present in our designs, but is unconnected since the SDRAM module is unimplemented. The algorithm clock is used to drive the algorithm core. The frequency varies depending on the algorithm type and configuration, but can be easily set and adjusted through API function calls.

4.2 Host Communication and Data Format

Except for special-purpose debugging and diagnosis signals, all data passing between the host and the system goes through the `OKBTPipeIn` and `OKBTPipeOut` interfaces. These interfaces' fixed word width of 16 bits was a major influence in our host-system communication design.

On the input side, the MSB of each word is used to distinguish between commands and data. If the MSB is set, the word is a command or delimiter. In Smith-Waterman, these commands are used to signal the beginning of query and database sequences, for example. Although BLAST lacks a central control unit with which to interpret and execute commands, this bit serves a similar purpose of acting as a reserved bit to trigger writes to the serial-in random-out (SIRO) modules.

If the MSB is cleared, the word is interpreted as data. The SIRO modules in

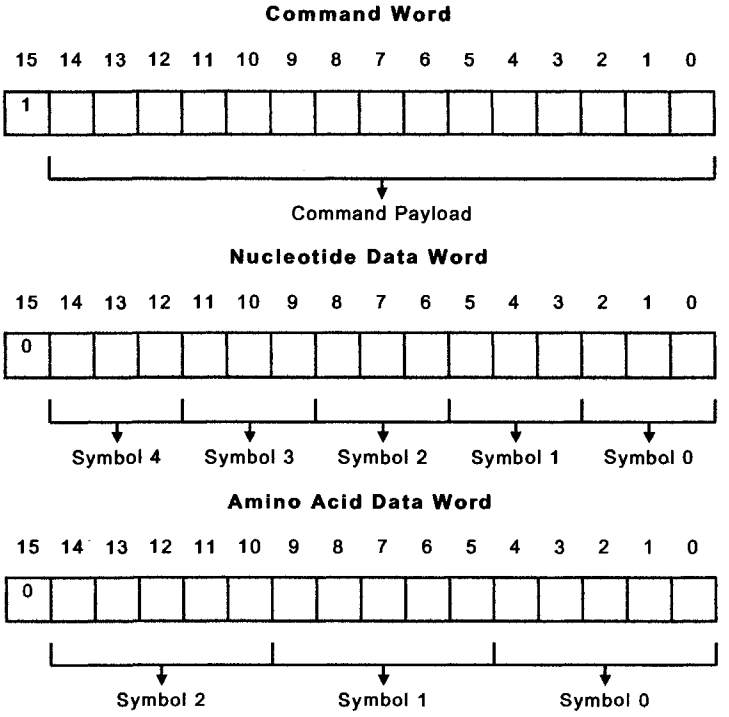

Figure 4.2: Input word data formats.

BLAST simply take raw data as input. Biological sequence data is divided into 'symbols', each representing one letter in the alphabet of nucleotides or amino acids. Nucleotides are 3 bits, and thus are packed in 5 to a word. Although there are only 4 nucleotides, which can be represented with 2 bits, it is useful to define extra special-purpose symbols. Among these are a symbol that scores negatively with all symbols, including itself, and a symbol for representing a gap. Hence we use 3 bits rather than 2. The 20-symbol amino acid alphabet requires 5 bits per symbol, and thus 3 symbols are delivered per data word. Like our nucleotide symbol set, the amino acid set also contains special-use symbols, but in this case their introduction does not cause symbols to use up extra bits.

All sequence data is in little-endian format. Figure 4.2 illustrates the formats of a command word and both types of sequence data word.

Results, as similarity scores and locations, are returned in raw format. In general, each element of a single 'result' is sent as either 1 or 2 words, depending on

whether more than 16 bits are required. For example, in our Smith-Waterman system, all result elements are transmitted as 2 words. The first contains the lower 16 bits, while the second contains the upper 16 bits.

Chapter 5

Conclusions

This document describes research in which a new FPGA-based implementation of the Smith-Waterman algorithm was created, and a high-speed, low-complexity implementation of simplified BLAST was proposed. Our Smith-Waterman design sets a new benchmark by handling longer sequences than any other previously published FPGA implementation. It is also distinguished by running on a low-cost, portable platform with an inexpensive Spartan-3 series FPGA, while still achieving large speed-up factors over current microprocessors. As shown in Section 2.8.4, speed-up factors range from 10 to 30 for amino acid sequences and 20 to 100 for nucleotide sequences.

Our proposed BLAST is unique in devising several new heuristics to allow more effective hardware acceleration. Our design presents several new approaches to BLAST hardware acceleration, most notably the parallel hit look-up and 2-hit filtering stages of the algorithm.

Future work will focus on integrating our platform's unused SDRAM module into our designs, which will allow even longer sequence lengths, and improved flexibility handling the large amounts of data that sequence alignments entail. There are also several approaches to improve and extend our BLAST design, such as performing parallel hit detection on chip rather than requiring hits to be pre-computed by the host.

Bibliography

[1] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[3] H. Nash, D. Blair, and J. Grefenstette. Comparing algorithms for large-scale sequence analysis. In *Proc. IEEE Intl. Symp. Bioinformatics and Bioengineering*, volume 2, pages 89–96, November 2001.

[4] National Center for Biotechnology Information. NCBI-GenBank flat file release 159.0 distribution release notes. Technical report, NCBI-GenBank, 2007.

[5] UniProt Consortium. The universal protein resource (UniProt). *Nucleic Acids Research*, 35 (Database Issue):193–197, 2007.

[6] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[7] R. K. Singh, D. L. Hoffman, S. G. Tell, and C. T. White. BioSCAN: a network sharable computational resource for searching biosequence databases. *Comput. Appl. Biosci.*, 12(3):191–196, 1996.

[8] P. Guerdoux-Jamet and D. Lavenier. SAMBA: hardware accelerator for biological sequence comparison. *Comput. Appl. Biosci.*, 13(6):609–615, 1997.

[9] B. Schmidt, H. Schröder, and M. Schimmler. Massively parallel solutions for molecular sequence analysis. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 201, Washington, DC, USA, 2002. IEEE Computer Society.

[10] A. Di Blas, D. M. Dahle, M. Diekhans, L. Grate, J. Hirschberg, K. Karplus, H. Keller, M. Kendrick, F. J. Mesa-Martinez, D. Pease, E. Rice, A. Schultz, D. Speck, and R. Hughey. The UCSC Kestrel parallel processor. *IEEE Transactions on Parallel and Distributed Systems*, 16(1):80–92, 2005.

[11] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a GPU. In *IPDPS 2006 Parallel and Distributed Processing Symp.*, volume 20, April 2006.

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

[13] Active Motif Inc. `http://www.timelogic.com`.

[14] Y. Yamaguchi, T. Maruyama, and A. Konagaya. High speed homology search with FPGAs. In *Proc. Pacific Symposium on Biocomputing*, pages 271–282, 2002.

[15] T. F. Oliver, B. Schmidt, and D. L. Maskell. Reconfigurable architectures for bio-sequence database scanning on FPGAs. *IEEE Transactions on Circuits and Systems II*, 52(12):851–855, December 2005.

[16] D. J. States, W. Gish, and S. Altschul. Improved sensitivity of nucleic acid database searches using application-specific scoring matrices. *METHODS: A Companion to Methods in Enzymology*, 3(1):66–70, August 1991.

[17] S. F. Altschul. Amino acid substitution matrices from an information theoretic perspective. *Journal of Molecular Biology*, 219:555–565, 1991.

[18] W. R. Pearson. Comparison of methods for searching protein sequence databases. *Protein Science*, 4:1145–1160, 1995.

[19] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.

[20] Xilinx Inc. Spartan-3 FPGA Family: Complete Data Sheet.

http://www.xilinx.com. August 2005.

[21] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. In *Proc. Nat. Acad. Sci.*, volume 89, pages 10915–10919, 1992.

[22] S. F. Altschul, T. L. Madden, A. A. Schaeffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[23] S. Bates, B. Knudsen, P. Meulemans, T. Rollingson, and O. Zaboronski. 100 GCUPS low complexity BLAST filter for FPGA.

[24] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. In *Proc. Natl. Acad. Sci. USA*, volume 85, pages 2444–2448, April 1988.

[25] National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/.

[26] K. Muriki, K. D. Underwood, and R. Sass. RC-BLAST: towards a portable, cost-effective open source hardware implementation. In *Proc. 19th IEEE Intl. Symp. Parallel and Distributed Processing (IPDPS 2005)*, April 2005.

[27] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster. Biosequence similarity search on the Mercury system. In *ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference on (ASAP'04)*, pages 365–375, Washington, DC, USA, 2004. IEEE Computer Society.

[28] A. Jacob, J. Lancaster, J. Buhler, and R. Chamberlain. FPGA-accelerated seed generation in Mercury BLASTP. *FCCM*, 0:95–106, 2007.

[29] E. Sotiriades, C. Kozanitis, and A. Dollas. FPGA based architecture for DNA sequence comparison and database search. In *Proc. 20th IEEE Intl. Symp. Parallel and Distributed Processing (IPDPS 2006)*, April 2006.

[30] M. Herbordt, J. Model, Y. Gu, B. Sukhwani, and T. VanCourt. Single pass, BLAST-like, approximate string matching on FPGAs. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Cus-*

tom Computing Machines (FCCM'06), pages 217–226, Washington, DC, USA, 2006. IEEE Computer Society.

[31] Swiss Institute of Bioinformatics. UniProtKB/Swiss-Prot protein knowledgebase release 54.1 statistics. `http://ca.expasy.org/sprot/relnotes/relstat.html`.

[32] Xilinx Inc. Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Generation FPGAs. `http://www.xilinx.com/bvdocs/appnotes/xapp465.pdf`. May 2005.

[33] Opal Kelly Inc. Opal Kelly FrontPanel User Manual. `http://www.opalkelly.com`. September 2005.