Report Number: WUCSE-2005-30

2005-08-31

# NCBI BLASTN Stage 1 in Reconfigurable Hardware

Kwame Gyang

Recent advances in DNA sequencing have resulted in several terabytes of DNA sequences. These sequences themselves are not informative. Biologists usually perform comparative analysis of DNA queries against these large terabyte databases for the purpose of developing hypotheses pertaining to function and relation. This is typically done using software on a general multiprocessor. However, these data sets far exceed the capabilities of the modern processor and performing sequence similarity analysis is increasingly becoming less efficient. There is an urgent need for more efficient ways of querying large DNA sequences for sequence similarities. Here, we describe an FPGA-based hardware solution... **Read complete abstract on page 2.**

# NCBI BLASTN Stage 1 in Reconfigurable Hardware

Kwame Gyang

Complete Abstract:

Recent advances in DNA sequencing have resulted in several terabytes of DNA sequences. These sequences themselves are not informative. Biologists usually perform comparative analysis of DNA queries against these large terabyte databases for the purpose of developing hypotheses pertaining to function and relation. This is typically done using software on a general multiprocessor. However, these data sets far exceed the capabilities of the modern processor and performing sequence similarity analysis is increasingly becoming less efficient. There is an urgent need for more efficient ways of querying large DNA sequences for sequence similarities. Here, we describe an FPGA-based hardware solution that implements Stage 1 of NCBI BLASTN, a commonly used sequence analysis application.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2005-30

# NCBI BLASTN Stage 1 in Reconfigurable Hardware

Authors: Gyang, Kwame

August 31, 2004

Abstract: Recent advances in DNA sequencing have resulted in several terabytes of DNA sequences. These sequences themselves are not informative. Biologists usually perform comparative analysis of DNA queries against these large terabyte databases for the purpose of developing hypotheses pertaining to function and relation. This is typically done using software on a general multiprocessor. However, these data sets far exceed the capabilities of the modern processor and performing sequence similarity analysis is increasingly becoming less efficient. There is an urgent need for more efficient ways of querying large DNA sequences for sequence similarities. Here, we describe an FPGA-based hardware solution that implements Stage 1 of NCBI BLASTN, a commonly used sequence analysis application.

# NCBI BLASTN Stage 1 in Reconfigurable Hardware

**Kwame Gyang**

Kwame Gyang, “NCBI BLASTN Stage 1 in Reconfigurable Hardware,” Master’s Project, Department of Computer Science and Engineering, Washington University, August 2004, Technical Report WUCSE-2005-30, 2005.

School of Engineering and Applied Science
Washington University
Campus Box 1045
One Brookings Dr.
St. Louis, MO 63130-4899

# NCBI BLASTN STAGE 1 IN RECONFIGURABLE HARDWARE

Kwame Gyang
Department of Computer Science and Engineering
Washington University in St Louis

**Abstract**

*Recent advances in DNA sequencing have resulted in several terabytes of DNA sequences. These sequences themselves are not informative. Biologists usually perform comparative analysis of DNA queries against these large terabyte databases for the purpose of developing hypotheses pertaining to function and relation. This is typically done using software on a general multiprocessor. However, these data sets far exceed the capabilities of the modern processor and performing sequence similarity analysis is increasingly becoming less efficient. There is an urgent need for more efficient ways of querying large DNA sequences for sequence similarities. Here, we describe an FPGA-based hardware solution that implements Stage 1 of NCBI BLASTN, a commonly used sequence analysis application.*

## 1. Introduction

In recent years, there has been tremendous growth in DNA sequencing technologies resulting in large terabytes of DNA sequence maps. These sequences themselves are not informative. To develop hypotheses on the relation and function of a sequence, biologists need to perform comparative analysis on selected streams of gene sequences against these large DNA databases. Traditional string matching algorithms using software on general microprocessors are increasingly becoming slower, and hence, less attractive. This is partially because DNA sequence on the hard disk needs to be first loaded into main memory, and then to the cache for the CPU to execute the search algorithm on the data. The throughput is inhibited by the bandwidth of the I/O bus from the disk to main memory. Here, we explore a new approach to DNA string matching using reconfigurable hardware.

The most widely used software for rapid searching of nucleotides and protein databases is called BLAST, (**B**asic **L**ocal **A**lignment **S**earch **T**ool) [9]. There are several variants of BLAST, but this work focuses on the open source distribution of BLAST by the **N**ational **C**enter for **B**iological **I**nformation (NCBI) for finding similarities in nucleotides called NCBI BLASTN. As shown in Figure 1, NCBI BLASTN is a 3 stage pipeline consisting of word-matching, ungapped extension, and gapped extension stages. In the word-matching stage, a query is pre-loaded and broken up into overlapping smaller substrings of length w. For example, a query of *ACTGACTGACTG* can be broken up into 5 overlapping substrings {*ACTGACTG*,

*CTGACTGA*, *TGACTGAC*, *GACTGACT*, *ACTGACTG*} of length w = 8. (In this implementation w = 12, and the substring of length w is referred to, henceforth, as a w-mer). These w-mers are then compared to w-mers in a database. Matched w-mers, and their absolute positions in the query and database, are fed to the second stage of the pipeline. Each word match is filtered through stage 2, which tries to extend it into an ungapped alignment between query and database. Ungapped alignments with too few matching base pairs are discarded, while the remainder is further filtered in stage 3 of the pipeline.

To reduce the amount of unnecessary computations performed in stage 2, redundant matches can be eliminated prior to being sent to stage 2. This can be achieved by a redundancy filter that eliminates overlapping matches. Although each stage of BLASTN is more computationally intensive than the last, each stage also discards a substantial fraction of the inputs received from the previous stage. The volume of data that is processed at each gradually decreases. This fact is illustrated in Figure 1 by the triangular-shaped encapsulation of the 3 pipeline stages. The tapering of the triangle depicts the substantial elimination of input w-mers from stage 1 to stage 3. Table 1 from [2] quantifies the data reduction at each stage of the pipeline. The match rate $p_i$ in Table 1, represents the probability that an output from stage i is generated from an individual input into that stage. For stage 1, $p_1$ measures the number of matches per DNA base read from the database. In performances analysis performed by Krishnamurthy et al. [2], it was shown that the time spent in Stage 1 of the pipeline dominated the time spent in the other stages even though the other stages are more computationally intensive. They also concluded that Stage 1 was the performance bottleneck in the NCBI BLASTN pipeline. Table 2, also from [2], gives the distribution of the time spent in various stages of NCBI BLASTN with varying query sizes. Improving this stage would produce significant speed-up in the overall performance of the pipeline. This is the motivation behind this project. In this project, we implement stage 1 of NCBI BLASTN in hardware on a reconfigurable hardware platform with the goal of improving stage 1 of NCBI BLASTN, and hence, the entire pipeline. The remaining pipeline stages remain in software. It was also shown in [2] that stage 2 becomes the pipeline bottleneck after improvements in Stage 1. Hardware improvement of stage 2 will be implemented in future work.

In this work, the word-matching stage of the NCBI BLASTN pipeline was replaced by an FPGA-based hardware application. This hardware application consists of 2 main components; the hardware solution of NCBI stage 1 (henceforth referred to as hBLASTN) and a redundancy filter component. hBLASTN was implemented as a 5-stage pipeline; Input Processing, Parallel Bloom Filter, String Buffering, GGPerf Hashing, and Look-Up Table Units. Apart from the Parallel Bloom Filter and the redundancy filter components, which were based upon work done by [5] and [1] respectively, all other components in this work were designed and implemented by the author. hBLASTN and hardware application is used interchangeably throughout.

The content of this paper is organized as follows: Section 2 describes the Hardware Module Interface; Section 3 talks about the Data Input and Output Specifications; Section 4 describes the implementation of hBLASTN; Section 5 describes the redundancy filter; Section 6 provides analysis of timing and scalability of hBLASTN and finally, Section 7 describes tests performed to ascertain functional correctness of the application and Section 8 concludes by providing an overall summary and potential extensions to hBLASTN in future work.

**Figure 1: Pipeline stages of NCBI BLASTN [2]**

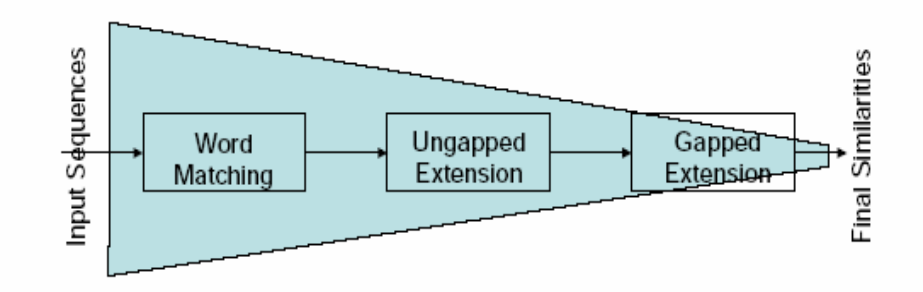



**Table 1: Match rate p across pipeline stages [2]**

| Query Size (bases) | Stage 1 ($p_1$) | Stage 2 ($p_2$) | Stage 3 ($p_3$) |
|---|---|---|---|
| 10K | 0.00858 | 0.0000550 | 0.320 |
| 100K | 0.0841 | 0.0000174 | 0.175 |
| 1M | 0.837 | 0.0000175 | 0.117 |

**Table 2: Percentage of pipeline time spent in each stage of NCBI BLASTN [2]**

| Query Size (bases) | Stage 1 | Stage 2 | Stage 3 |
|---|---|---|---|
| 10K | 86.5 ± 1.51% | 13.24 ± 1.99% | 0.23 ± 0.017% |
| 100K | 83.35 ± 1.28% | 16.57 ± 2.17% | 0.08 ± 0.007% |
| 1M | 85.29 ±2.40% | 14.68 ± 3.70% | 0.03 ± 0.002% |

## 2 . Hardware Module Interface

The top level interface of the hardware module is shown in Figure 2. This top level interface was modeled after a commercial version to conform to the commercial interface and to enhance portability and interchangeability of this design [7]. This interface borrows ideas from the network prototype wrapper interface [10] used in the **F**ield Programmable **P**ort e**X**tender (FPX) [8]. The typical arrangement using this interface is shown in Figure 2a where the hardware implementation, hBLASTN, is enveloped in a wrapper layer. The wrapper handles input/output regardless of the source of input data, i.e. whether from the network [8] or disk [7]. In this configuration, we concentrate only on the hBLASTN implementation and leave the input/output details to the wrapper layer. Furthermore, hBLASTN would still work seamlessly if the source of data was from the network (i.e. in this work, the data is sent via ATM cells to the hardware module) or if the source of data was from a disk (i.e. the commercial version).

**Figure 2: Module Input and Output Interface**

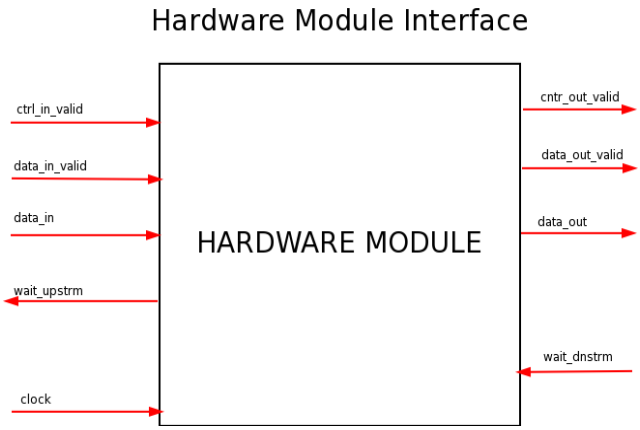
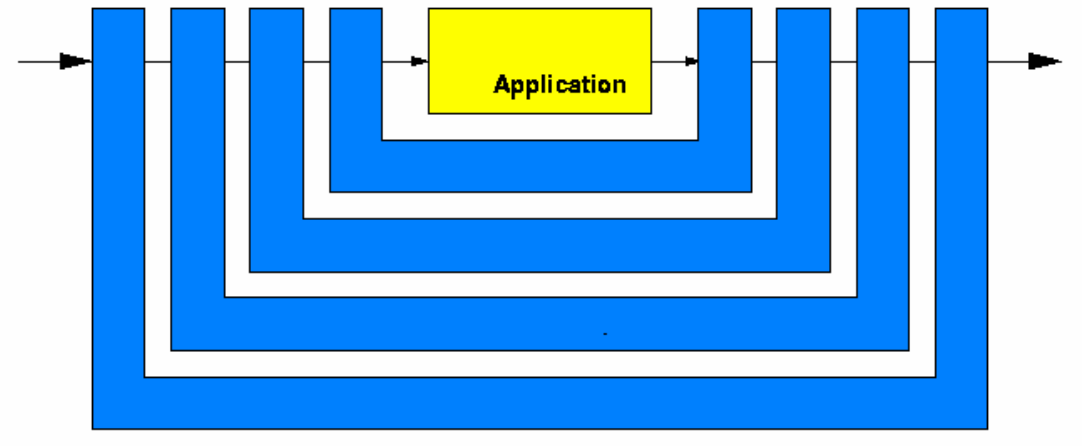



**Figure 2a: Hardware Module with a four-layered wrapper**



**Upstream Signals to Hardware Module**

**Inputs:**

**ctrl_in_valid (1 bit signal) –** This is a control signal to indicate that the data on the *data_in* bus holds valid control data. Active when high.

**data_in_valid (1 bit signal) –** Control signal to indicate that the data on the data_in bus holds valid data. Active when high.

**data_in (63:0 bus) -** 64 bit data and control bus. When *data_in_valid* is active, the 64 bits on the *data_in* bus should be interpreted as valid data. When *ctrl_in_valid* is active, data on the *data_bus* should be interpreted as a control command.

**clock (1 bit signal) –** The input clock is targeted at 80 MHz. This clock frequency is dependent on the critical path of the external modules to this hardware unit.

**Output:**

**wait_upstrm (1 bit signal) –** Control to tell upstream modules to temporarily stop sending data.

**Downstream Signals from Hardware Module**

**Outputs:**

**data_out_valid (1 bit signal) –** Control signal to indicate that the data on the data_out bus holds valid data. Active when high.

**ctrl_out_valid (1 bit signal) –** Control signal to indicate that the data on the data_out holds valid control data. Active when high.

**data_out (63:0 bus) –** When *data_out_valid* is active, the 64-bits should be interpreted as valid data. When *ctrl_out_valid* is active, data on the *data_out* should be interpreted as control command.

**Input:**

**wait_dnstrm (1 bit signal) –** Control signal to indicate that downstream modules cannot receive any more data.

**3. Data Input and Output Specification**

The hardware module receives strings of DNA sequences encoded using 4-bits per base as shown in Table 3 below. NCBI BLASTN chooses to implement the encoding using 2-bits per base. This compression minimizes storage and I/O bandwidth at the cost of being unable to process special characters through the pipeline. hBLASTN receives command (specific instruction to hBLASTN) and DNA base streams from a query and database via the 64-bit data_in bus. When data_in_valid is asserted, data_in is a data stream, and if ctrl_in_valid is asserted, data_in is a command. When both signals are asserted, hBLASTN interprets data_in as a command since it first examines the ctrl_in_valid signals. The wait_upstrm is asserted when

wait_dnstrm is asserted, or any of the hBLASTN's components are not ready, or the internal hBLASTN buffers are full. This causes the upstream components to stop sending data or command information to hBLASTN.

The output format is displayed in Table 4. The last 14 bits represent the position of w in the query string. The next 14 bits are debugging signals, and the remaining 36 bits represent the position of the w-mer in the database.

**Table 3: Table shows input data specification for hardware unit.**

| DNA CHARACTER / BASE | ENCODED BIT STREAM |
|---|---|
| A | 1100 |
| a | 1000 |
| C | 1101 |
| c | 1001 |
| G | 1110 |
| g | 1010 |
| T | 1111 |
| t | 1011 |
| N | 0010 |
| X | 0001 |

**Table 4: Table describes the output data specification for hardware unit.**

| 64-bit output value | Significance of bit Positions in Output |
|---|---|
| data_out [63 – 28] 36 bits | Represents position of w-mers in database stream |
| data_out[27 – 14 ] 14 bits | Represents debugging signals |
| data_out[13 – 0] 14 bits | Represents position of w-mers in query |

## 4. Description of hBLASTN

hBLASTN is a highly pipelined circuit. This was achieved by breaking complex processing with flip-flops in order to reduce critical paths but at the expense of overall latency. This enabled us to achieve clock rates of about 80 MHz, but the overall design has a 70 clock cycle latency. hBLASTN consists of five major pipeline stages or circuit components. These are the Input Processing Unit, Parallel Bloom Filter, String Buffering

Unit, GGPerf Hashing Unit, and a Look-Up Unit. Figure 3 shows the relationship among these pipeline stages. Bloom Filters are used to discover word-matches between query and database streams.

The goal of hBLASTN is to discover word-matches between the query and the database and also output the position of the matched w-mer in the database and all positions of all occurrences of the matched w-mer in the query. A simple counter is used to keep track of the database position, however, tracking the query position is not quite so simple. A typical approach is to maintain a hash table that will map each w-mer in the query to its position in the query. However, typical hashing results in collisions which need to be resolved. Resolving collisions in hardware consumes extra memory resources, which was critical especially on the Xilinx Virtex II XCV2000E FPGA device on which hBLASTN was designed. In trying to resolve this problem, a novel approach was taken. We implemented a Perfect Hashing Unit in hardware (GGPerf Hashing Unit Stage) which guaranteed zero collision and modest memory utilization. This is described further in section 4e.

All incoming data and control signals are first received via the Input Processing Unit. This component is responsible for interpreting commands and asserting or de-asserting appropriate control signals in the entire module, and also creating 16 simultaneous w-mers from 2 cycles of the 64-bit data on the input data bus. Output from the Input Processing Unit enters the Parallel Bloom Filter Unit. This component is responsible for performing membership queries and detecting w-mer matches in the database stream. Results of membership queries from this stage are buffered in the next stage of the pipeline, the String Buffering Unit, and the matched w-mers serialized into the subsequent stage, GGPerf Hashing Unit. This component performs perfect hashing on the w-mer to determine its position in the query stream. This computed w-mer position is used as an address by the Look-Up Table Unit to probe for other entries of w-mer in the query stream. At this point in the pipeline stage, we have the database position of the matched w-mer, which is tracked by means of a simple counter, and all possible positions of the w-mer in the query. The database position is combined with each of the possible w-mer positions, formatted according to the output specification described in Section 2 and is then output to the next stage of the implementation.

**Figure 3: hBLASTN pipeline Stages**

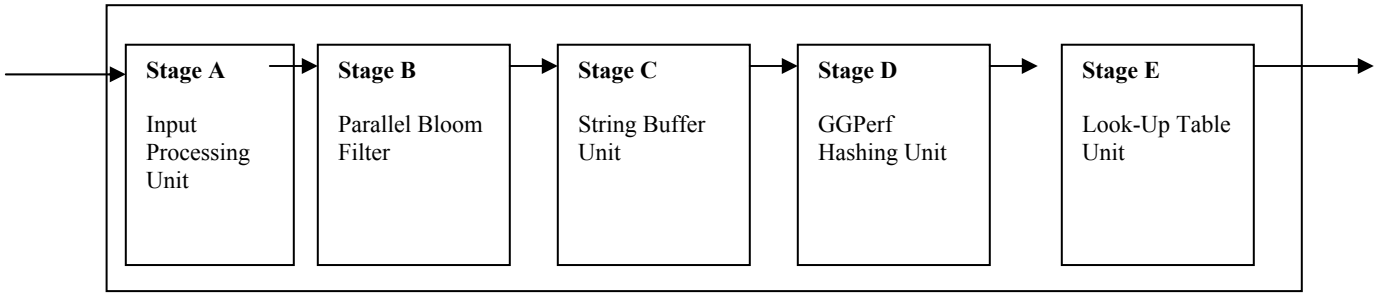


**Figure 3a: hBLASTN with Redundancy Filter**

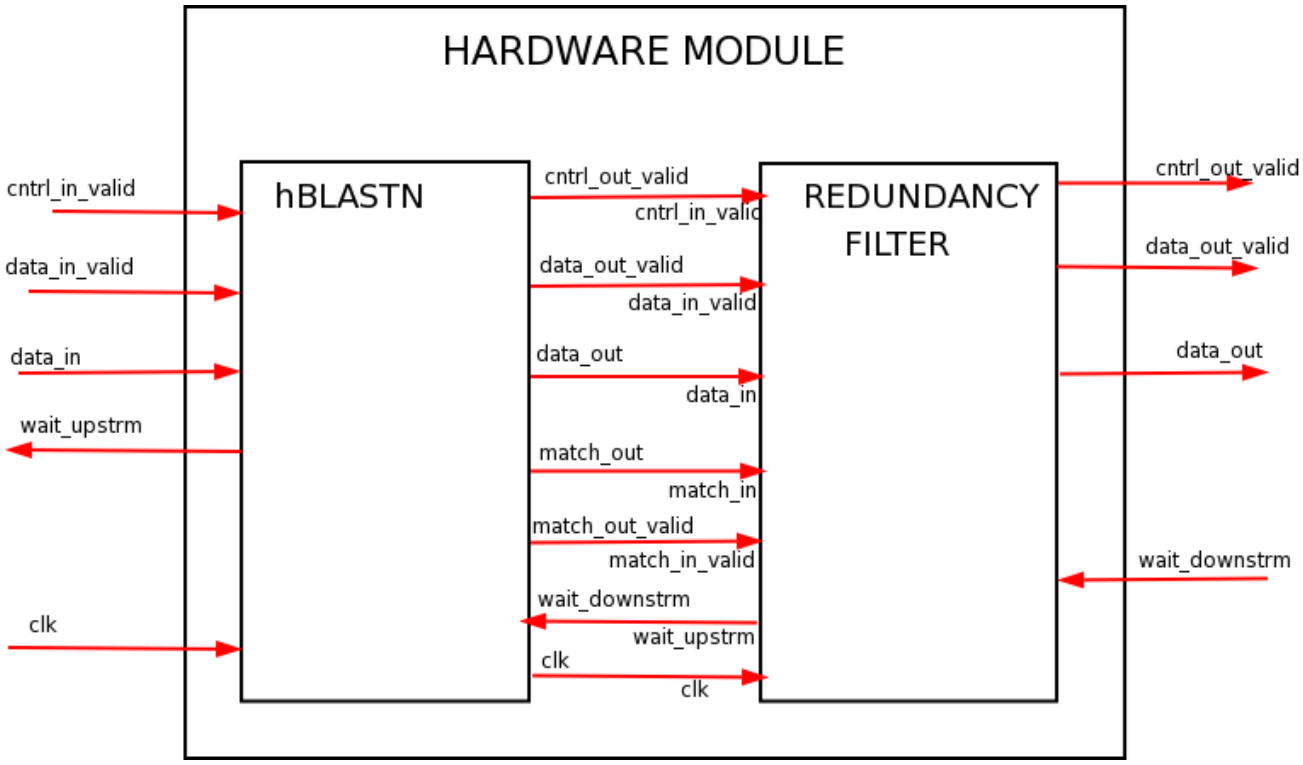
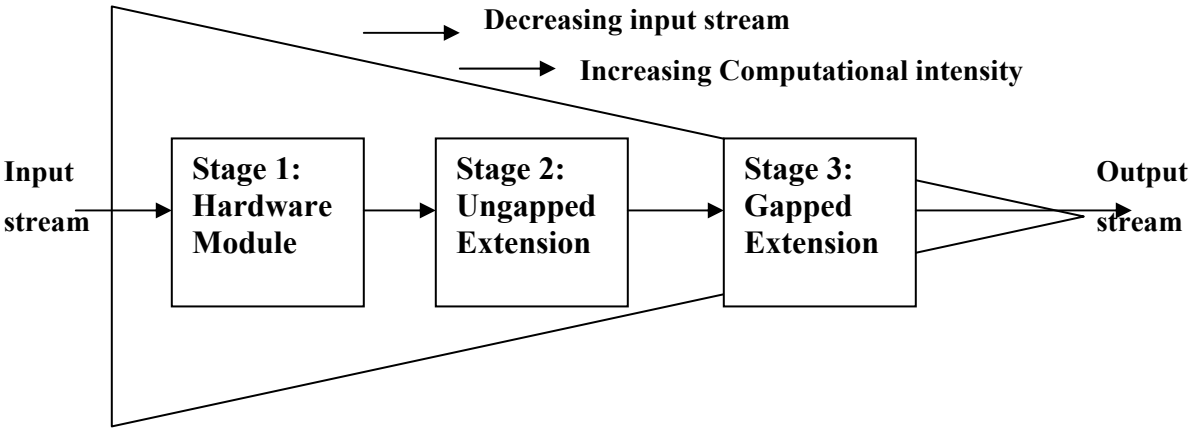



**Figure 3b: hBLASTN with Redundancy Filter in relation to NCBI BLASTN pipeline**




**4a. Input Processing Unit**

The Input Processing Unit is the first stage of hBLASTN pipeline and is made up of two components; the Controller and Fragmentator units. The Controller Unit is a big finite state machine that monitors the ctrl_in_valid, data_in_valid, and data_in signals to hBLASTN. When ctrl_in_valid is asserted, data_in is

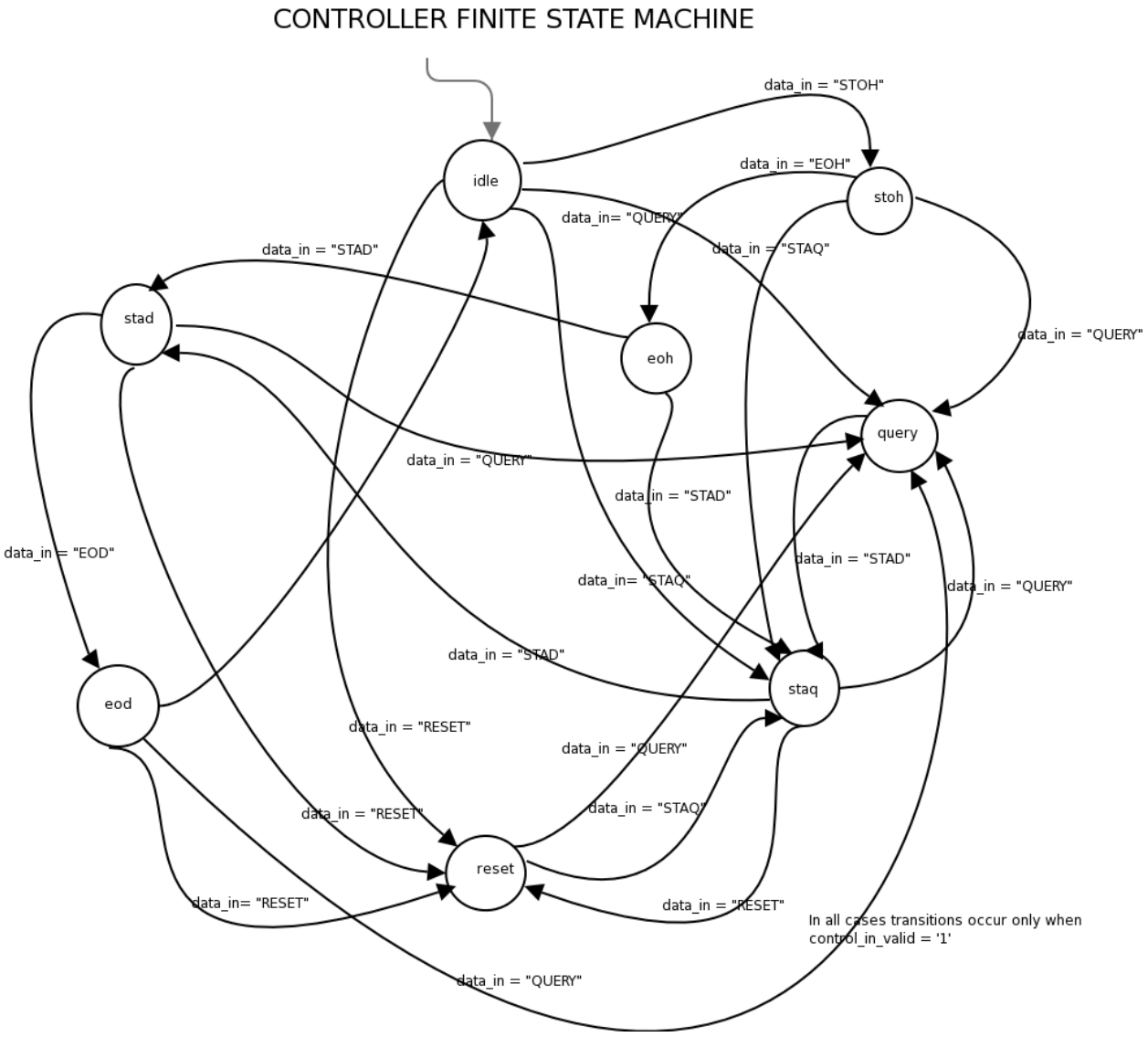
treated as a control or command signal to hBLASTN. Controller Unit handles a predefined set of commands and sets the appropriate output signals that control the entire module. These commands and function are shown in Table 5 below. The format of the command on the 64-bit data bus is described in [7].

**Table 5: hBLASTN Commands**

| Command | Function | Results |
|---|---|---|
| QW (QUERY) | Queries Hardware module for unique Module Num. | Return module number |
| RS (RESET) | Reset hBLASTN | Initialize memory, counters to zero |
| ED (EOD) | End of database stream | Reset database counter |
| EH (EOH) | End of duplicate stream | |
| SD (STAD) | Start of database stream | Initialize database counter |
| SQ (STAQ) | Start of query stream | |
| PASSTHU | Pass input data through module unmodified | Pass data though pipeline |
| CONFIG | Configure specific module | Specific module modified |
| SH(STOH) | Start of duplicate stream | |

All commands assert or de-assert control signals in the Hardware Module. For example, STAQ command asserts the enable, write enable, and bit data signals on the Parallel Bloom Filter component. This prepares the Unit to start writing incoming data into memory.

**Figure 9: Finite Stage Machine of Controller. (Note: Not all commands are not shown in this figure)**




On each rising edge of the clock, there is a new 64-bit data (or 16 bases) on the data_in bus if data_in_valid is asserted. The Fragmentator Unit fragments 32 bases into 16 w-mers (as outlined in Section 1) for the Parallel Bloom Filter component to use to either program (write to memory) or query (read from memory) the Bloom Filters. In this Fragmentator implementation, DNA bases were not case sensitive, hence 2-bit/base encoding was used for all valid DNA bases (i.e. A, C, T, and G). If a w-mer contained an invalid DNA base (i.e. X, N and T), an invalid bit was asserted. Each of the 16 w-mer is a 25-bit data bus corresponding to 12 bases (2-bits/base), and the last bit used to signal a valid w-mer.

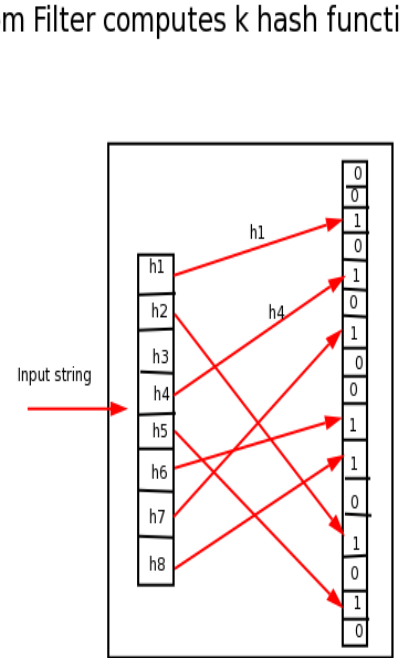
### 4b. Bloom Filter Overview

Bloom filter, a randomized, space-efficient data structure for representing a set of inputs and supports membership queries, was first proposed by Burton Bloom in 1970 and was originally used as spell checker on Unix platforms [3]. An important property of a Bloom filter is that it never generates a false negative result on membership queries, and has a rather small false positive rate. Bloom filters have been historically used in modern applications such as Content Networks, Summary Caches, Packet Routing and collaborating in overlay and peer-to-peer networks [4].

### Bloom Filters

A set of inputs, ***n*** is passed through a Bloom filter which performs ***k*** hashes on each **α** belonging to the set ***n***. This generates a set ***K*** outputs for each α, where |***K***| = ***k***. Each **β** belonging to ***K*** is used as an index into a bit vector of length ***m*** (previously initialized to 0) where that location is set to 1. This is called "programming the Bloom filter". Figure 4 illustrates this programming of the Bloom filter. The set of inputs, n could present the set of w-mers of a query for which comparative analysis is desired.

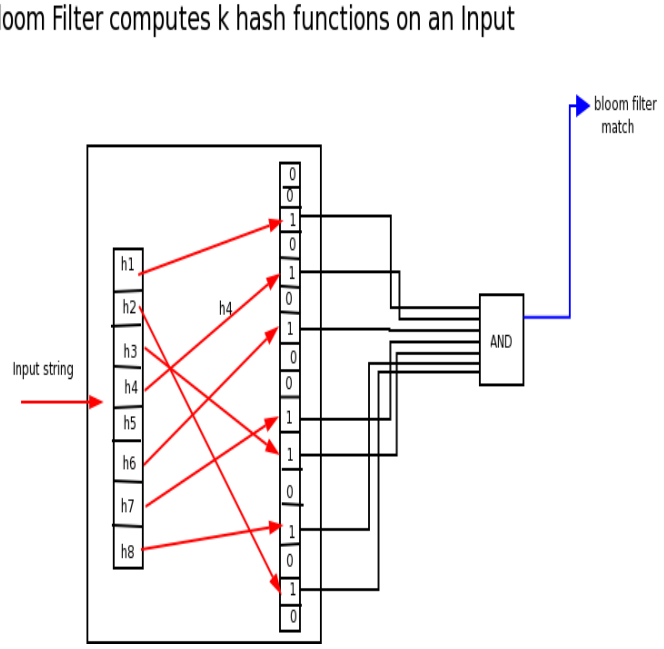
**Figure 4: A Bloom filter performs k = 8 hashes on the input string. Each hash value is used as an index to program the 1-bit array.**




Streaming database w-mers can then be also passed through the Bloom filter to check for membership. Again, the Bloom filter performs ***k*** hashes on an input. This generates a set ***K*** of hashed values which are

used as keys into the programmed ***m*** bit vector. In performing membership queries, the values in the hash locations are read and all ***K*** outputs are "logically ANDed". If the result is 1, this indicates, to a certain degree of confidence, a possible w-mer content match. Figure 5 illustrates the process of membership queries.

**Figure 5: Bloom filter performing membership queries.**




**FALSE POSITIVES ANALYSIS**

Bloom filters do have a small false positive rate. Hence a given w-mer from the database could be wrongly flagged by the Bloom filter as having matched a w-mer in the query. The false rate, ***f***, is given by the expression

$$f = (1 - e^{-nk/m})^k$$

where ***n*** is the number of w-mers programmed in Bloom filter, i.e. the query size. ***k*** is the number of hash functions, ***m***, the length of 1-bit array. The false positive rate, ***f***, can be reduced by choosing appropriate values for ***m*** and ***k*** for a given member set ***n*** [5]. It is quite clear from the equation that ***m*** needs to be quite large compared to ***n***. Also for a given ***m/n*** ratio, the false positive rate can be decreased by increasing ***k***, the number of hashes performed by the Bloom filter. In the optimal case, which the false positive probability minimized with respect to ***k***, ***k*** is given by the expression below.

$$k = (m/n)\ln 2$$

This results in a false positive probability of

$$f = (1/2)^k$$

### 4c. Parallel Bloom Filter

The Parallel Bloom Filter component is the second stage of hBLASTN pipeline and is responsible for identifying matches between w-mers in the query and the database stream. This is built from 16 large Bloom filters. Each large Bloom filter was built from 5 partial Bloom filters. Each partial Bloom Filter component consists of a 4096 by 1 dual port Block RAM. Each partial Bloom filter supports 2 simultaneous reads and writes, thus $k$ =2 in this configuration. By combining 5 partial Bloom filters into a large Bloom filter, each large Bloom filter now supports 10 reads and writes ($k$ = 10). Figures 5, 6 and 7 show the construction of the Parallel Bloom Filter. The combination of 16 large Bloom filters into a Parallel Bloom Filter Unit enables 16 simultaneous query w-mer look ups, thus improving throughput of hBLASTN. Since searching for query matches in the database stream is the most computationally intensive part of hBLASTN, this component is highly pipelined and exhibits significant parallelism. This component alone consumes 60% of the Block RAM resources on the Xilinx Virtex II 2000E FPGA.

**Figure 6: Diagram illustrates the internal components of the Partial Bloom filter**

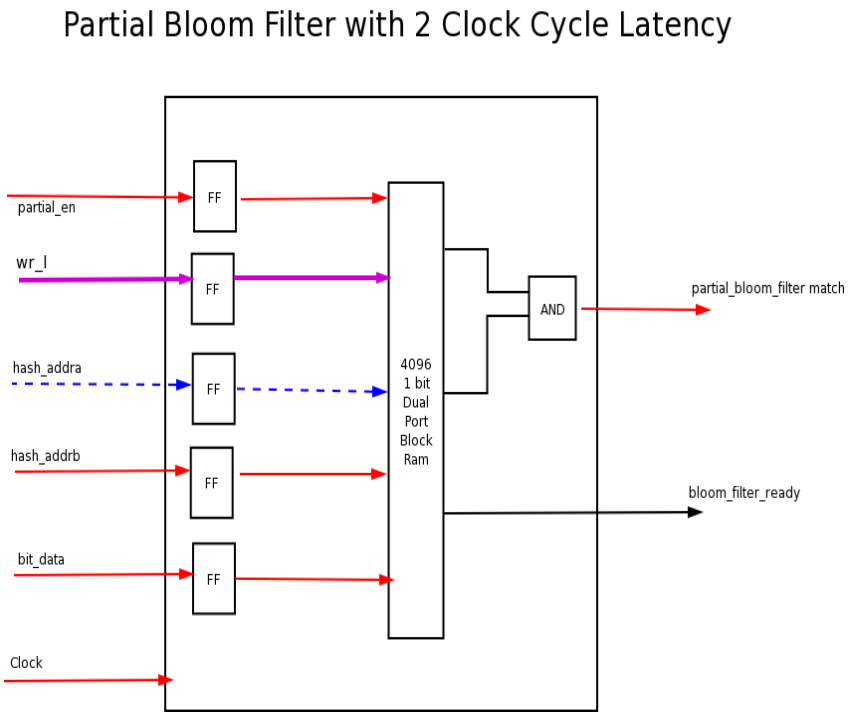


**Figure 7: Internal components of the Large Bloom filter.**

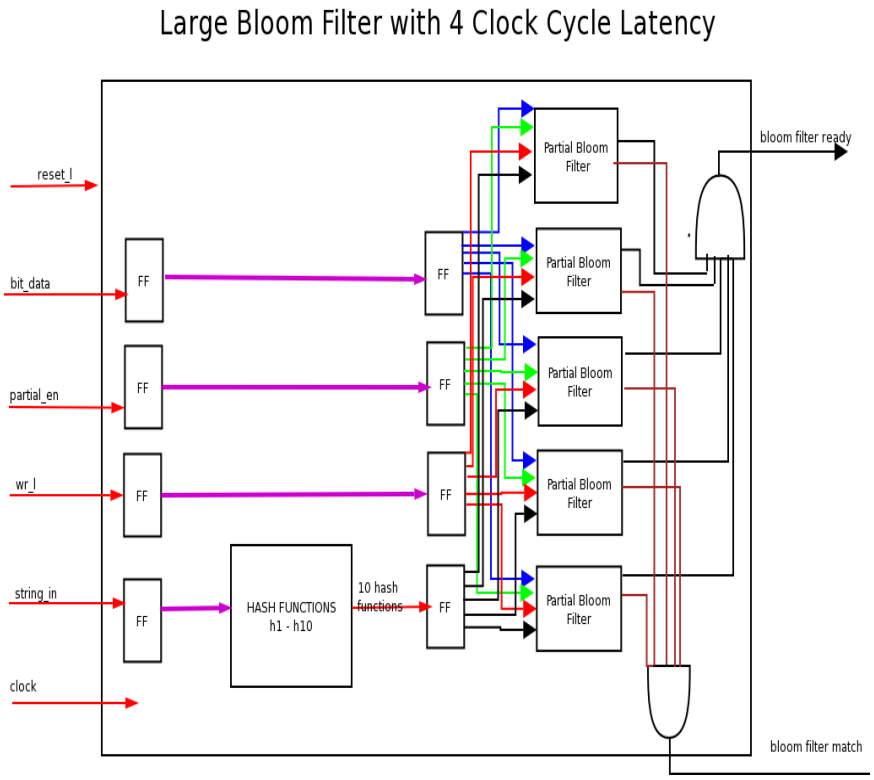
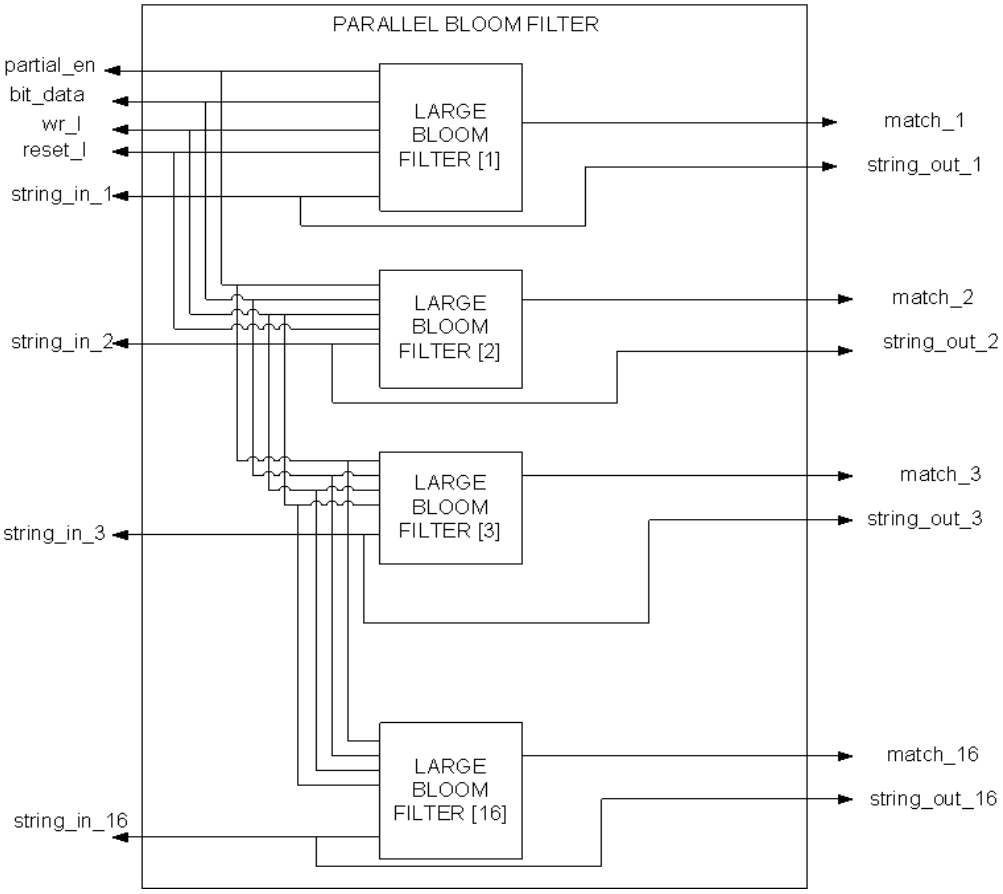


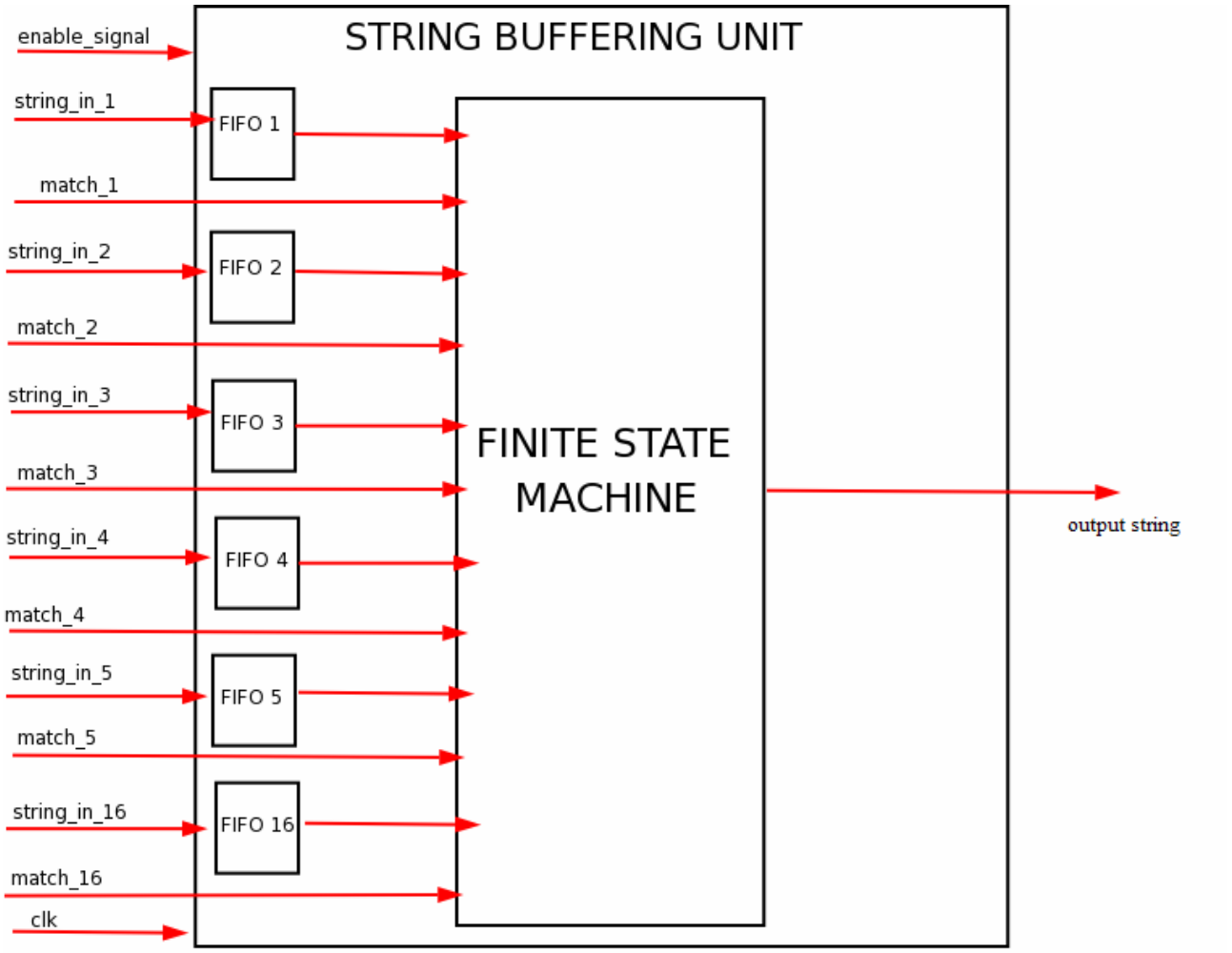

**Figure 8: This describes the internal structure of the Parallel Large Bloom Filter component**



### 4d. String Buffer Unit

String Buffer Unit (SBU) is the third stage of hBLASTN pipeline and consists a Finite State Machine (FSM) and 16 FIFOs or buffers, each capable of storing 16 entries. The previous stage, Parallel Bloom Filter unit, has 16 25-bit data output signal, each with its corresponding match bit signal. These match signals are asserted if its w-mer, with a high degree of certainty, matches a w-mer in the query. If any of the matches is asserted, the SBU is enabled. Its purpose is to accept 16 simultaneous 25-bit data output signals from the Parallel Bloom Filter and stream out, in order, each of the 16 w-mers to the next component in the pipeline as illustrated in Figure 10 below.

**Figure 10: String Buffering Unit internals**



When the SBU is enabled, all 16 25-bit data output signals from Parallel Bloom Filter unit and their corresponding match bit signal are written into the 16 buffers. The FSM is triggered as soon as the buffers are non-empty. The first task of the FSM is to read and latch all 16 entries. The next task of the FSM is to visit each of its 16 output states with the match bit asserted to output the 25-bit data to the next component in the pipeline, thus serializing the 16 parallel entries. The SBU was made more efficient by having the FSM visit only states with the match bit asserted. Since it takes 16 clock cycles, in the worst case, to serialize all 16 inputs to the SBU, the buffers can easily be filled. The SBU asserts a back pressure signal when the buffers are half-filled (due to the latency in the previous stages of the pipeline). The back pressure signal is fed to the wait_upstrm signal which causes hBLASTN not to receive any more input while the SBU processes entries in the buffers.

**4e. GGPerf Hashing Unit**

The GGPerf Hashing Unit is the fourth stage of the hBLASTN pipeline. Its function is to accept w-mers from the previous pipeline stage, SBU, and determine its position in the query. GGPerf Hashing Unit is a hardware implementation of GGPerf [6] which is an improvement over Gperf, commonly found in unix/linux operating systems. Gperf is a program-generating-program, meaning it is a program that generates a high level program. The generated program is a minimal perfect hash function capable of retrieving information in one probe [6]. GGPerf, **G**reater than **GPerf**, is a more robust and stable Gperf program.

**4e-1. What is a Minimal Perfect Hash Function**

In general a hash function, ***h***, is a function that maps an element (key) in a domain , **K** (the set of all w-mers in the query stream) to its corresponding address in the range, **I** (the set of all addresses). The address is used to retrieve a record from a database corresponding to the key. This database is the hash table. It is possible that ***h*** maps **n** (**n** > 1) keys to the same address. When this occurs, a collision is said to have happened. To resolve collisions, we need to relocate **n**-1 records to other locations, leaving only one record. Hash functions, therefore, yield probable addresses. In general, a hash function will map a key in the set **K** (where |**K**| = ***x***) into some interval of integers in the set **I**, say [0 …***m***-1], where ***m*** ≥ ***x***.

$$h$$

a: $\mathbf{K} \rightarrow \mathbf{I}$ , $|\mathbf{K}| \leq |\mathbf{I}|$

general hash function yields probable address

A perfect hash function, ***p***, on the other hand uniquely transforms each key (or domain element) into an address in the hash table without any collisions. Perfect hash functions yield definite address instead of probable address. A property of perfect hash function is that only a single key probe into the hash table is required to obtain the address thus improving time efficiency of the hash table [6].

$$b:\ \mathbf{K} \xrightarrow{p} \mathbf{I}\ ,\ |\mathbf{K}| \leq |\mathbf{I}| \quad \text{and } \boldsymbol{m} \geq \boldsymbol{x}$$

perfect hash function yields definite address

If $|\mathbf{K}| = |\mathbf{I}|$ and $\boldsymbol{m} = \boldsymbol{x}$, then ***p*** is a minimal hash function.

$$b:\ \mathbf{K} \xrightarrow{p} \mathbf{I}\ ,\ |\mathbf{K}| = |\mathbf{I}|$$

minimal perfect hash function

For a minimal perfect hash function, that hash table is the most time and space efficient [6].

**4e-2. GGPerf in hardware**

Input to GGPerf software is a query (see Appendix A) fragmented into overlapping w-mers (see Appendix B), and it generates another high level program (see Appendix C). This program consists of 4 tables of constants and a hash function. This hash function uses the 4 tables to compute a mapping between the key (w-mer in this case) and its address. GGPerf Hashing Unit only implements the hashing function algorithm in hardware. GGPerf source code was modified to generate a hardware-ready table of constants (see Appendix D) to be used with GGPerf Hashing Unit.

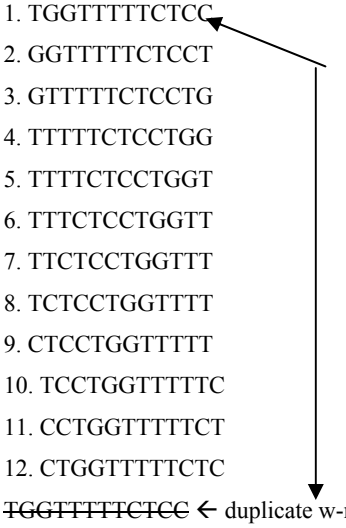
**4e-3. GGPerf Hardware Characteristics**

GGPerf software translates string keys into another high level program. This program is a minimal perfect hash function which translates the string keys into unique addresses. The string keys input to GGPerf have to be **unique**. To ensure the string keys are unique, a separate preprocessor program first parses and fragments the raw query and removes all occurrences of duplicates. The duplicates are processed separately and stored in a Look up Table. This is illustrated in the example below.

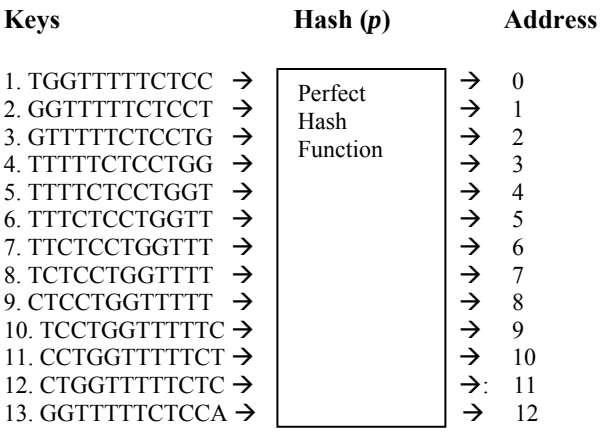
Consider this raw query:

**TGGTTTTTCTCCTGGTTTTTCTCCA**

This is fragmented into overlapping w-mers as shown below




By removing all duplicate occurrences, GGPerf software is now capable of processing these unique string keys (w-mers) into another high level program. The generated high level program has a minimal perfect hash function, ***p***, that can map each of the string keys to a unique address, ***a,*** such that ***a*** is a number between 0 and 12).




An important observation from the above analysis is the fact that the address computed by the minimal hash function on a key (w-mer) is the position of the w-mer in the raw query of hBLASTN. This statement is true

only if there are no duplicates. If duplicate w-mers are removed the w-mer GGTTTTTCTCCA at position 13 would hash to the address 12 as shown above but w-mer GGTTTTTCTCCA begins at position 13 in the query stream. Rather than removing duplicate w-mers, a *filler* is inserted in place of the duplicate. Fillers are unique w-mers which are randomly generated. By replacing the duplicate TGGTTTTTCTCC with a filler AAAAAAAAAAAA, the w-mer GGTTTTTCTCCA is moved to position 14 and hashes to address 13 which is the correct query position of this w-mer.

GGPerf Hashing Unit takes in as input, matched w-mers from the Parallel Bloom Filter Unit and computes its address which also corresponds to the position of the w-mer in the query stream. However, since the Parallel Bloom Filter has a false positive rate, it is possible for false w-mers to enter the GGPerf Hashing Unit. From the equations in section 4b, Bloom filters have a $(1/2)^{10}$ optimal false positive rate. Most of these false positive w-mers will be detected by the GGPerf Hashing Unit. The remainder will subsequently be detected by the other stages of NCBI BLASTN.

Furthermore, by inserting fillers in place of duplicates in the query stream, we decrease the selectivity of GGperf Hashing Unit, since any false positive w-mer which happens to be the same as the filler will have a valid GGPerf Hash address. This decrease in selectivity is minimized because of the relatively few duplicates that are usually detected in query streams.

**4f. Look Up Table**

The Look Up Table (LUT) is the last stage of the hBLASTN pipeline. In this implementation, LUT has as many entries as there are query bases. This assumes the worst case scenario where all the DNA bases in the query are the same. However, a best or average case could be implemented rather than the worst case scenario in order to efficiently utilize memory resources on the hardware platform. LUT is used to determine if there are any more occurrences of a w-mer in the query stream. This table is populated with the duplicates extracted from the query by the preprocessor software program. At the GGPerf Hashing Unit stage of the pipeline, we have identified a w-mer match between the query and database. We also know the position of the first occurrence of the w-mer in the query stream and its corresponding database position. The w-mer position from GGPerf Hashing Unit is used to probe the LUT for other w-mer positions (if any) of the same w-mer in the query stream. LUT is constructed from single port Block RAMs and its 30-bits wide and the w-mer position from GGPerf Hashing Unit is used as an address to the LUT. If the least significant bit (LSB) of the 30-bit record of an address is 0, then there are no further occurrences of the w-mer in the query stream. The first position is the only position of the w-mer in the query. However, if the LSB is 1, then the next 14-bits from the LSB corresponds to the next position of the w-mer in the query stream. The last 14-bits

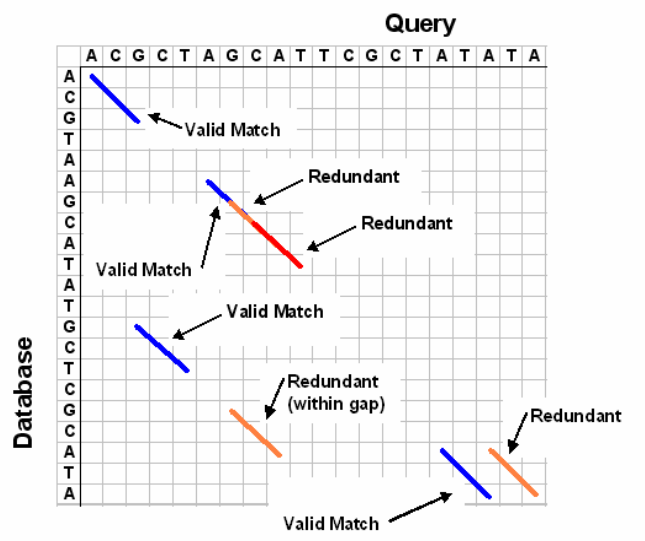
correspond to the address in the LUT where another position of the w-mer can be obtained if it is non-zero. This chaining of positions corresponding to a particular w-mer in the query stream is determined in software and streamed into the LUT during initialization phase of hBLASTN using STOH (Start of Duplicate Stream) and EOH (End of Duplicate Stream) commands.

## 5. Redundancy Filter

The following description of Redundancy filter is from work done in [1]

> *Redundant matches are seen in the form of overlapping matches. Given an exact match of length W and a trailing gap of length G, a second match is seen as redundant if it has query-database coordinates which lie within W+G characters of the previous match coordinates and its coordinates lie on the same row or diagonal as the previous match coordinates. It is the purpose of the hardware redundancy filter to remove these redundant matches. Figure 11 illustrates the concept of the database-query index matrix with examples of the various valid and redundant matches.*

**Figure 11: Database-Query Matrix [1]**




## 6. Timing and Scalability of hBLASTN

The main motivation for hBLASTN is to improve throughput of NCBI BLASTN. At 80MHz and with a 16 bases/clock data bus means hBLASTN has a maximum processing rate of 1280MBases/sec. This rate is not quite achievable in practice since back pressure signals in hBLASTN may slow down the input rate. There is

a significant pre-processing overhead before and during the initialization phase of hBLASTN. GGPerf software accounts for a significant fraction of this overhead. Table 6 shows the time spent by GGPerf in generating the high level program output for different query sizes.

**Table 6: GGPerf software runtime performance**[1]

| Size of Query | GGPerf time (seconds) |
|---|---|
| 1KBases | 1.33 |
| 10 KBases | 65.59 |
| 100 KBases | 5828.77 |
| 1MBases | Out of Memory |

Another concern is how well this design scales up as the size of the query increases. To analyze scalability of hBLASTN we examined how each stage of the pipeline performs when query size is scaled up. Also, we analyzed how well the design scales in hardware on the Xilinx XCV2000E FPGA. Stage A of the pipeline is independent of the size of query, hence scaling of query has no significant effect on the performance. Stage B, however, is affected significantly by the scaling up of query size while memory requirements of the Parallel Bloom Filter remain constant. Since the false positive rate is proportional to the size of query, an increase in the query size increases the false positive rate of the Parallel Bloom Filter Unit. A Block RAM is a 4096 bit array with 2 read/write ports. This can be used in a Bloom Filter to support 2 hash functions (k = 2). This filter can handle n = (m/k)ln2 = 1419 w-mers generating an optimal false positive rate of $(1/2)^{10}$ (for k = 10 hash functions) [5]. In order to achieve an optimal false positive rate for different query sizes, the memory requirements of the Parallel Bloom Filter Unit, ***m***, or the number of hash functions, ***k***, need to be adjusted according to the false positive analysis equations.

Moreover, false positive w-mers will be mostly eliminated during Stage D of the pipeline. This is because, GGPerf Hashing Unit will detect that a false positive w-mer has no address mapping. This will therefore be dropped from valid w-mer matches input into the LUT stage of the pipeline. However, a small fraction of input w-mers (typically about 10 -13 %) will not be eliminated and will continue into the next stage of the pipeline. Eventually, these false results will be detected and eliminated by stage B of the pipeline.

There is also no impact of scaling on Stage C, String Buffer Unit of the pipeline except that a significant number of false positive w-mers will cause the buffers in SBU to be constantly filled. This could potentially decrease the throughput of hBLASTN.

[1] Measurements performed on a dual 930MHz Pentium III processor with 512KB of memory

Stage D is independent of query size. However Stage E is dependent on scaling. Increasing query size can cause the buffers in LUT to fill quickly, and thus causing the back pressure signal to be asserted. This will decrease the throughput of hBLASTN. The entire performance of hBLASTN could be adversely affected by increasing query size as a result of Stages 3 and 5 if there are no corresponding changes to the memory requirements of the Parallel Bloom Filter Unit.

In analyzing how well resources on the Xilinx XCV2000E FPGA scaled up with query size, we generated several hardware GGPerf tables for different query sizes. These tables are significantly larger for increasing query sizes, and hence, quickly consume LUT resources on the FPGA device. Table 8 shows the percentage of LUT use on the FPGA with increasing query sizes.

**Table 8: Hardware LUT utilization [2]**

| Size of Query | Percentage of LUT |
|---|---|
| 1 KBases | 34% |
| 10 KBases | 43% |
| 100 KBases | 51% |
| 1 MBases | Unknown |

## 7. Tests and Validation

A testbench was created for each of the five pipeline components of hBLASTN as well as for the entire hBLASTN application. This allowed for separate pipeline stage simulation as well as for the entire hBLASTN. For stage 1 of the pipeline, Input Processing Unit, the testbench supplied the component with varying 64-bit data and command on the data_in bus. When cntrl_in_valid was asserted, the output control signals of the Unit was examined to determine if the appropriate control signals were asserted for the particular command. When the data_in_valid signal was asserted, the 16 output signals were examined for the correct w-mers. Similar testbenches were created for the other stages, and the outputs validated independently.

[2] Results from Synplicity

To validate the entire hBLASTN module, software which performs the exact function as hBLASTN was created. The same inputs were passed into the software application as well as the testbench for hBLASTN. The output from hBLASTN testbench simulation was written to a file (See Appendix E). Output from the software application is also written to a file. In order to compare entries in both files, a separate application parses the output from hbLASTN testbench, which is in hexadecimals, and generates an output similar to the format used in the software output (See Appendix F). A simple unix diff or compare command us used to compare both files, line by line. This validates the correct functionality of hbLASTN.

## 8. Conclusions and Future Work

This paper presented the design of a hardware implementation of Stage 1 of NCBI BLASTN, hBLASTN, with a Redundancy Filter on a reconfigurable hardware platform. hBLASTN provides a significant improvement in throughput, and performance of Stage 1 of NCBI BLASTN, and also a significant reduction in data flow to Stage 2 of the pipeline. hBLASTN's flexibility and scalability is limited by the memory provided by the hardware platform. hBLASTN also incurs significant offline software overhead. Notwithstanding these limitations however, hBLASTN offers significant improvement over software stage 1 NCBI BLASTN. In order to realize the full potential of hBLASTN, further work is needed to convert it from a behavioral model into a fully functional reconfigurable circuit. Also, further throughput and speed improvement can be realized with a hardware implementation of stage 2 of NCBI BLASTN.

**9. References**


[1] C. Behrens, J. Lancaster, and B. Wun "BLASTN Redundancy Filter in Reprogrammable Hardware", in *CSE 535 Final Project Submission*, Fall 2003.

[2] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang and J. Lancaster, "Biosequence Similarity Search on the Mercury System" in Proc. IEEE Int'l Conf. *of Application-specific Systems, Architecture, and Processors*, 2004. To appear.

[3] B. Bloom. "Space/time trade-offs in hashing code with allowable errors". *ACM*, 13(7):422-426, May 1970.

[4] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters", A survey.

[5] S. Dharmapurikar, M. Attig, and J. Lockwood, "Design and Implementation of a String Matching System for Network Intrusion Detection using FPGA-based Bloom Filters". *Washington University, Department of Computer Science and Engineering,* Technical Report, 2004.

[6] Jiejun Kong, "GGPerf: A Perfect Hash Function Generator", *http://www.cs.ucla.edu/jkong/public/soft/GGPerf*, 1997.

[7] M. Franklin, R. Chamberlain, M. Henricks, B. Shands, and J. White, "An Architecture for Fast Processing of Large Unstructured Data Sets", in *International Conference on Computer Design*" October 2004. To appear.

[8] J. Lockwood et al., "Field Programmable Port Extender (FPX) User Guide: Version 2.2", *Washington University, Department of Computer Science and Engineering*, Technical Report WUCS-02-15, June 2002.

[9] S. F. Altschul,W. Gish,W. Miller,E . W. Myers, et al. "Basic local alignment search tool." *Journal of Molecular Biology*, 215:403–10, 1990.

[10] F. Braun, J. Lockwood, and M. Waldvogel. "Reconfigurable router modules using network protocol wrappers". In *Proceedings of Field-Programmable Logic and Applications*, pages 254-263, Belfast, Northern Ireland, Aug 2001

## 9. Appendices

## APPENDIX A
**(Query Sample)**

```
TGGTTTTT
CTCCAGTT
TGTTATTT
GTCATTTC
```

## APPENDIX B
**(GGPerf Input)**

```
%{
// File Auto Generated from Preprocessor Program...
// Generated on Wed Jun 09 16:26:35 CDT 2004
%}
%%
TGGTTTTTCTCC
GGTTTTTCTCCA
GTTTTTCTCCAG
TTTTTCTCCAGT
TTTTCTCCAGTT
TTTCTCCAGTTT
TTCTCCAGTTTG
TCTCCAGTTTGT
CTCCAGTTTGTT
TCCAGTTTGTTA
CCAGTTTGTTAT
CAGTTTGTTATT
AGTTTGTTATTT
GTTTGTTATTTG
TTTGTTATTTGT
TTGTTATTTGTC
TGTTATTTGTCA
GTTATTTGTCAT
TTATTTGTCATT
TATTTGTCATTT
ATTTGTCATTTC
%%
```

## APPENDIX C
**(GGPerf generated high level program in software)**

```
// File Auto Generated from GGPerf Program...
// Generated on Wed Jun 09 16:26:35 CDT 2004

class G_index
{
   public static final int TOTAL_KEYWORDS = 21;
   public static final int MIN_HASH_VAL = 0;
   public static final int MAX_HASH_VAL = 20;
   public static final int MIN_CHAR_VAL = 'A';
   public static final int MAX_CHAR_VAL = 'T';
   public static final int MIN_WORD_LENG = 12;
   public static final int MAX_WORD_LENG = 12;
   private static final int MAX_TABLE_HEIGHT = 12;
   private static final int MAX_GRAPH_NODE_VAL = 105;
   private static final int T1[][]=
   {
   {41,-1,4,-1,-1,-1,54,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,52},
   {20,-1,56,-1,-1,-1,51,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,100},
   {19,-1,92,-1,-1,-1,52,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,92},
   {40,-1,48,-1,-1,-1,60,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,62},
   {69,-1,94,-1,-1,-1,81,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,103},
   {95,-1,5,-1,-1,-1,35,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,104},
   {85,-1,21,-1,-1,-1,73,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,40},
   {18,-1,91,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,15},
   {28,-1,45,-1,-1,-1,29,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,0},
   {13,-1,21,-1,-1,-1,2,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,60},
   {18,-1,58,-1,-1,-1,1,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,85},
   {61,-1,104,-1,-1,-1,32,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,52},
   };
   private static final int T2[][]=
   {
   {38,-1,41,-1,-1,-1,103,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,19},
   {31,-1,8,-1,-1,-1,31,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,63},
   {16,-1,60,-1,-1,-1,14,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,35},
   {34,-1,22,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,34},
   {44,-1,80,-1,-1,-1,34,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,77},
   {66,-1,9,-1,-1,-1,86,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,67},
   {68,-1,55,-1,-1,-1,45,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,80},
   {103,-1,43,-1,-1,-1,98,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,44},
   {8,-1,46,-1,-1,-1,24,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,49},
   {36,-1,89,-1,-1,-1,65,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,64},
   {16,-1,4,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,16},
   {32,-1,52,-1,-1,-1,46,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,26},
   };
   private static final int MAX_NODE_NUM = 35;
   private static final int node[]=
   {11,7,2,14,37,45,26,55,48,91,35,93,42,81,8,
   17,71,85,62,101,22,58,60,34,98,64,94,104,80,54,63,
   69,90,99,33};
   private static final int G[]=
   {0,0,0,1,0,2,0,3,0,4,0,5,0,6,5,
   0,8,0,9,0,10,0,11,11,0,13,18,17,0,15,16,
   13,0,19,20};
   private static String[] wordlist = new String[21];

   private static void init()
   {
```

```
            wordlist[0] = new String("TGGTTTTTCTCC");
            wordlist[1] = new String("GGTTTTTCTCCA");
            wordlist[2] = new String("GTTTTTCTCCAG");
            wordlist[3] = new String("TTTTTCTCCAGT");
            wordlist[4] = new String("TTTTCTCCAGTT");
            wordlist[5] = new String("TTTCTCCAGTTT");
            wordlist[6] = new String("TTCTCCAGTTTG");
            wordlist[7] = new String("TCTCCAGTTTGT");
            wordlist[8] = new String("CTCCAGTTTGTT");
            wordlist[9] = new String("TCCAGTTTGTTA");
            wordlist[10] = new String("CCAGTTTGTTAT");
            wordlist[11] = new String("CAGTTTGTTATT");
            wordlist[12] = new String("AGTTTGTTATTT");
            wordlist[13] = new String("GTTTGTTATTTG");
            wordlist[14] = new String("TTTGTTATTTGT");
            wordlist[15] = new String("TTGTTATTTGTC");
            wordlist[16] = new String("TGTTATTTGTCA");
            wordlist[17] = new String("GTTATTTGTCAT");
            wordlist[18] = new String("TTATTTGTCATT");
            wordlist[19] = new String("TATTTGTCATTT");
            wordlist[20] = new String("ATTTGTCATTTC");
    }

    private static int G_index(int n)
    {
            int i;
            for(i=0; i<MAX_NODE_NUM; i++)
               if(node[i] == n)
                        return i;
            return -1;
    }

    public static int hash(String key)
    {
      int i, leng=key.length(), f1=0, f2=0, n1=-1, n2=-1, t1, t2;
        for(i=0; i<leng; i++)
            {
               char c = key.charAt(i);
              if(c < MIN_CHAR_VAL || c > MAX_CHAR_VAL)
                        return -1;
               t1 = T1[i][key.charAt(i)-MIN_CHAR_VAL];
               t2 = T2[i][key.charAt(i)-MIN_CHAR_VAL];
               if(t1 == -1 || t2 == -1)
                        return -1;
               f1 += t1;
               f2 += t2;
            }
            f1 %= MAX_GRAPH_NODE_VAL;
            f2 %= MAX_GRAPH_NODE_VAL;
            n1 = G_index(f1);
            n2 = G_index(f2);
            return (G[n1]+G[n2]) % TOTAL_KEYWORDS;
    }

    public static String in_word_set(String key)
    {
   int len=key.length();

if(len<=MAX_WORD_LENG&&len>=MIN_WORD_LENG)
   {
               int ind = hash(key);
               if(ind <= MAX_HASH_VAL && ind >= 0)
                  {
                        String rec = wordlist[ind];
                        if(rec.compareTo(key) == 0)
                           return rec;
                  }
               }
               return null;
    }
}
```

## APPENDIX D

**(GGPerf generated hardware tables.)**

perfect_hash.vhd

```
--Author: Kwame Gyang
--Generated Automatically from GGPerf
-- Fri Jun 11 14:52:46 CDT 2004

LIBRARY IEEE;

PACKAGE perfect_hash IS
CONSTANT TOTAL_KEYWORDS    : INTEGER  := 21;
CONSTANT T_HASH_BITS       : INTEGER  := 7 ;
CONSTANT T_ROWS            :INTEGER  := 12 ;
CONSTANT  MAX_GRAPH_NODE_VAL:  INTEGER  := 105;
CONSTANT MAX_NODE_NUM      : INTEGER   := 36;
CONSTANT A_TO_MINVAL       : INTEGER   := 0;
CONSTANT C_TO_MINVAL       : INTEGER   := 2;
CONSTANT T_TO_MINVAL       : INTEGER   := 19;
CONSTANT G_TO_MINVAL       : INTEGER   := 6;

TYPE hash_array is ARRAY (0 to 19) of INTEGER;
TYPE perfect_hash_T1 is ARRAY (0 to T_ROWS -1) of hash_array;
TYPE perfect_hash_T2 is ARRAY (0 to T_ROWS -1) of hash_array;
SUBTYPE ggperf_integer is integer range 0 to 35 ;
TYPE perfect_hash_node is ARRAY (0 to MAX_NODE_NUM -1) of INTEGER;
TYPE perfect_hash_g is ARRAY(0 to MAX_NODE_NUM - 1 ) of INTEGER;
CONSTANT  T1 :   perfect_hash_T1  :=

 (
(  73,-1,24,-1,-1,-1,66,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7),
(  8,-1,7,-1,-1,-1,72,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,17),
(  30,-1,58,-1,-1,-1,9,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,80),
(  97,-1,72,-1,-1,-1,52,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,92),
(  12,-1,82,-1,-1,-1,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,77),
(  36,-1,29,-1,-1,-1,102,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,37),
(  9,-1,1,-1,-1,-1,50,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,14),
(  12,-1,69,-1,-1,-1,28,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,70),
(  92,-1,82,-1,-1,-1,80,-1,-1,-1,-1,-1,-1,-1,-1,-1,
```

```
   -1,-1,-1,-1,79),
(  44,-1,3,-1,-1,-1,19,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,104),
(  59,-1,38,-1,-1,-1,85,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,9),
(  63,-1,52,-1,-1,-1,82,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,55)
);


CONSTANT  T2 :   perfect_hash_T2  :=

 (
(  20,-1,77,-1,-1,-1,89,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,97),
(  36,-1,66,-1,-1,-1,73,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,79),
(  39,-1,84,-1,-1,-1,86,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,51),
(  81,-1,102,-1,-1,-1,40,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,81),
(  37,-1,32,-1,-1,-1,32,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,74),
(  98,-1,73,-1,-1,-1,75,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,12),
(  73,-1,32,-1,-1,-1,13,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,89),
(  85,-1,33,-1,-1,-1,97,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,17),
(  89,-1,22,-1,-1,-1,95,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,18),
(  96,-1,47,-1,-1,-1,48,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,48),
(  62,-1,49,-1,-1,-1,86,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,51),
(  34,-1,10,-1,-1,-1,14,-1,-1,-1,-1,-1,-1,-1,-1,
   -1,-1,-1,-1,81)
);


CONSTANT node : perfect_hash_node  :=

(
24,28,60,20,36,55,21,22,35,56,18,71,12,97,79,
   6,23,32,76,41,73,29,0,14,78,19,42,46,61,13,39,
   53,87,45,70,99);

CONSTANT  G : perfect_hash_g :=

(
0,0,0,1,0,2,0,3,0,4,5,0,6,0,7,
   0,8,0,9,0,10,5,0,12,0,13,0,14,0,15,0,
   16,20,18,12,20);


END perfect_hash;
PACKAGE BODY perfect_hash IS
END perfect_hash;
```

## APPENDIX E

(Hexadecimal output from hBLASTN)

```
0000000000000008
0000000510000001
0000000520000002
0000000530000003
0000000540000004
0000000550000005
0000000560000006
0000000570000007
0000000580000008
0000000590000009
00000005A000000A
00000005B000000B
00000005C000000C
00000005D000000D
00000005E000000E
00000005F000000F
0000000600000010
0000000610000011
0000000620000012
0000000630000013
0000000640000014
```

## APPENDIX F

(Test software output. Separate parses program formats hBLASTN hexadecimal output to software format shown here)

```
match found: string TGGTTTTTCTCC. dpos = 80, qpos = 0
match found: string GGTTTTTCTCCA. dpos = 81, qpos = 1
match found: string GTTTTTCTCCAG. dpos = 82, qpos = 2
match found: string TTTTTCTCCAGT. dpos = 83, qpos = 3
match found: string TTTTCTCCAGTT. dpos = 84, qpos = 4
match found: string TTTCTCCAGTTT. dpos = 85, qpos = 5
match found: string TTCTCCAGTTTG. dpos = 86, qpos = 6
match found: string TCTCCAGTTTGT. dpos = 87, qpos = 7
match found: string CTCCAGTTTGTT. dpos = 88, qpos = 8
match found: string TCCAGTTTGTTA. dpos = 89, qpos = 9
match found: string CCAGTTTGTTAT. dpos = 90, qpos =10
match found: string CAGTTTGTTATT. dpos = 91, qpos =11
match found: string AGTTTGTTATTT. dpos = 92, qpos = 12
match found: string GTTTGTTATTTG. dpos = 93, qpos = 13
match found: string TTTGTTATTTGT. dpos = 94, qpos = 14
match found: string TTGTTATTTGTC. dpos = 95, qpos = 15
match found: string TGTTATTTGTCA. dpos = 96, qpos =16
match found: string GTTATTTGTCAT. dpos = 97, qpos =17
match found: string TTATTTGTCATT. dpos = 98, qpos = 18
match found: string TATTTGTCATTT. dpos = 99, qpos = 19
match found: string ATTTGTCATTTC. dpos = 100,qpos= 20
```