Hardware Accelerated Protein Identification

by

Anish Alex

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science in the
Graduate Department of Electrical and Computer Engineering,
University of Toronto

Hardware Accelerated Protein Identification

Master of Applied Science, 2003

Anish Alex

Graduate Department of Electrical and Computer Engineering

University of Toronto

ABSTRACT

The proteins in living organisms perform almost every significant function that governs life. A protein's functionality depends upon its physical structure, which in turn depends on its constituent *sequence* of amino acids as specified by its gene of origin. While many protein sequences are known, many remain to be discovered. Recent advances in mass spectrometry are capable of determining unknown protein sequences but the process is very slow. We review a new method of *de-novo* protein sequencing that requires a fast search of the genome. In this thesis, we present the design of FPGA-based hardware that can perform this search in a very fast and cost-effective manner. This hardware solution is between 3 to 60 times more cost effective than an equivalent software platform. In addition, we provide a framework to estimate the cost of the hardware at a desired level of performance.

ACKNOWLEDGEMENTS

I would first like to sincerely thank my supervisor Jonathan Rose for his help and motivation over the past two years. This project would certainly not have progressed had it not been for his keen interest and desire to "find the problem that we are the solution to". On the same note I'd like to thank Dr. Christopher Hogue at Mt. Sinai's SLRI. This project is a somewhat uncommon collusion between different fields, and the idea would never have taken footing without Dr Hogue's fascination with custom computing.

Many thanks to Dr. Stephen Davies, Dr. Zvonko Vranesic and Dr. Paul Chow for volunteering to be on my committee, especially on such short notice.

To the TM3/TM4 crew : Thanks for all the help – not just with the TM3. Had it not been for Marcus, Dave and especially Josh, my understanding of hardware design would not have been what it is today.

To the folks at Mt. Sinai : A big thank you to Ruth Isserlin who provided all the biological knowledge that I missed out on from Grade 11 onwards. Special thanks to Gary Bader and Michel Dumontier who patiently helped me understand the problem.
Many thanks to Kelly Boutilier at MDSP for providing me with data and instrument specs on such short notice.

To Pavel Metalnikov and Paul O' Donnell at Mt. Sinai's Mass Spec labs, thanks for taking the time explain mass spectrometry to a slow learner.

The past two years have been a lot of fun. For this I must certainly thank the guys in the lab. Cheers to Evil Tom, Mehrdad, Lorraine – sorry – Lesley, Ahmad, Navid "Tranzor" Azizi, Aaron, Peter "Blackbeard" Jamieson, Jason, Chairman Andy, Reza, Rubil, Imad, Denis (Dr. D) and Ian. Paul, its sad not hearing a rant against humanity on Fri morning. Hope you're having fun down south.

To my housemates G-bo, Jason, Capt. Lorraine and Mighty Mitch, I don't think I could've had a better set of housemates. Well, maybe the Justice League – but for puny humans, you guys were a lot of fun.

Last, but certainly not least, I'd like to thank my family for supporting me in every sense of the word for the last 23 years.

There's probably other people who should be on the list, but no worries, if I didn't remember you, you'll be getting a cheque for $40[1].

[1] Cheques will not be honoured

Table of Contents

Glossary

TERM	DEFINITION
Alternative Splicing	Process by which a single DNA strand could be transcribed into several different RNA sequences
Amino Acid	Subunit of a protein/peptide
Base	nucleotide,a DNA moelcule, can be one of A,T,C,G
Codon	Set of three bases in an RNA strand; used as a template for amino acids
De novo	Novel or hitherto unknown
Digestion	The process of breaking amino acid bonds in a protein
DNA	Deoxyribonucleic Acid
FPGA	Field-Programmable Gate Array
Gene	A hereditary unit of DNA that is responsible for the synthesis of proteins in an organism
Genome	All the genes of an organism
In silico	On a computer
Nucleotide	base, a DNA moelcule, can be one of A,T,C,G
Peptide	Chain of amino acids; piece of a protein
Protein	Chain of amino acids that serves a specific function
Proteome	The set of all proteins encoded by a Genome
RNA	Ribonucleic Acid
SAC	System Administration Cost, the cost of maintaining and upgrading a computer cluster
Sequence	The order of bases in a DNA strand or amino acids in a protein
Trypsin	Enzyme that digests proteins at the Argnine(R) and Lysine(K) amino acids
Tryptic peptide	Peptide formed from digestion of protein by trypsin
VHDL	VHSIC Hardware Descrition Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1. Introduction

1.1. Introduction to Proteins and Protein Identification

Proteins and their interactions regulate the majority of processes in the human body. From mechanical support in skin and bones to enzymatic functions, the operation of the human body can be characterized as a complex set of protein interactions. Over the past fifty years thousands of proteins have been studied [5], but despite the efforts of scientists, many proteins and their functions have yet to be discovered [4]. The wealth of information that lies in these unknown proteins may well be the key to uncovering the mysteries that govern life. The subject of this research is to investigate the use of digital hardware to aid in a specific technique used to discover new proteins.

A protein is composed of a long chain of molecules known as amino acids, and the order of these amino acids is known as the *sequence* of the protein [2]. *Protein sequencing* – the process of identifying the sequence of a given protein – is a means of establishing the protein's identity, from which its functionality can be inferred. In the past, sequencing was a slow, manual process in which individual amino acids of a protein were analyzed chemically [15]. The nature of these methods meant that sequencing took many weeks, even for relatively small proteins. Advances in technology over the past two decades introduced the concept of protein sequencing by mass spectrometry [10]. A *mass spectrometer* (MS) is a device that takes a biological or chemical sample as input and measures the masses of the constituent particles of the sample. This information, in combination with molecular mass databases, can be used to identify the molecules in the sample. Proteins, however, are large molecules and cannot be analyzed in their intact form; they must be *digested* or broken up into smaller subunits known as *peptides*. It is these peptides that are analyzed to determine the identity of the protein.

Mass Spectrometry for protein analysis can be divided into 4 distinct steps:

1. An MS takes the peptides from a set of digested proteins and measures the mass of each peptide. It then selects an individual peptide, using its mass to discriminate it from the others.

2. The selected peptide is fragmented and a second MS then analyzes the peptide; this is followed by a complex computation that produces the sequence of the selected peptide.

3. After a short delay (approx 1 sec.), Step 2 is repeated for another peptide. This is done for each peptide from every protein in the sample.

4. The peptide sequences from individual proteins are grouped together and ordered to obtain the full sequence of the each protein.

These MS techniques greatly reduce the sequencing time, but protein identification still requires several days. With a few hundred peptides in a sample, a great deal of the delay in the MS process comes from having to repeat the sequencing process (step 2) for each peptide [6]. Judicious analysis of the sample shows that not every peptide needed sequencing to obtain the full protein sequence [8]. However, this analysis needs to be fast to maintain a high-throughput mass spectrometry flow. This need for faster sample analysis coupled with the availability of cheap computing power has given rise to several techniques to accelerate protein sequencing. In the following section we describe the latest techniques for protein sequencing and motivate our work to accelerate one kind of sequencing with the use of digital hardware.

1.2. Thesis Motivation

Recent revolutions in biology and computing have sought to alleviate the analysis bottleneck described above. As stated above, the major hurdle in sample analysis is the number of peptides in the protein sample. However, it is possible to identify a protein using only a few of its peptides. There are many *characterized proteins* (proteins whose sequence is known) in biological databases. Using a small set of peptides as queries to these databases, the intact protein sequence that they originated from can be identified.

Using this technique, a few peptides from any protein can act as a unique *fingerprint* for that protein. Once the intact protein sequence has been obtained, all its constituent peptides can be eliminated from further analysis. This technique greatly reduces the number of times Step 2 has to be repeated before all proteins in the sample are identified. This technique of *peptide mass fingerprinting* (PMF) can be used to identify proteins in mere fractions of a second [9].
The limitation of PMF, however, is that it requires that the intact protein sequence already be present in the database. In *de-novo* sequencing experiments, researchers attempt to sequence a hitherto unidentified or novel protein. By definition, these proteins do not exist in a protein database, making direct PMF infeasible.

However, information about the sequence of novel proteins can be obtained elsewhere. Cells use the information contained in genes as a template to create proteins [2]. With the recent successful sequencing of the Human Genome, the set of all human genes is now available to researchers. It is possible to obtain the sequence of a protein if its gene can be identified. In effect, the genome can be interpreted as a complete protein database, thus overcoming the barrier presented by standard PMF searches [1].

Due to physical limitations of the instrument, it takes approximately 1 second before the second MS step can be repeated. To make an efficient high throughput protein identification system, it is crucial to be able to perform the genome database search within this 1-second interval. If the MS is forced to wait in excess of this delay, it incurs a non-productive downtime, which reduces its throughput and is considered both inefficient and expensive. Software techniques to perform this interpretation of the genome have thus far been slow requiring approximately 1 minute on a modern processor [1].

Over the past two decades, the benefits of custom hardware for computation have been seen in various applications [18][19][20]. For tasks such as database searching, where the search space is large and the operations are simple and parallelizable, custom hardware implementations of the algorithm show significant performance gains over software [18].

Thus the focus of this thesis is the design of a practical hardware system capable of accelerating the *de-novo* sequencing process using the genome. Our goal is to develop hardware that is both cheaper and faster than equivalently functional software. Note also, that there are myriad applications that search the human genome for diverse purposes from tracking human evolution to complex drug design. There are many fields of research that will benefit from the ability to search rapidly through the Human Genome.

1.3. Thesis Organization

This thesis is organized as follows: The second chapter provides details of the background biology and the technology in which the hardware is implemented. The third chapter describes the design and implementation of the hardware and the fourth chapter provides the results of this work in comparison with software running comparable algorithms on commodity processors. We also provide a framework to help the interested reader calculate the cost of this high-speed search based on the cost and density of the FPGAs available at the time. The fifth chapter will describe the conclusions of this work and avenues for future research.

Chapter 2. Background

In this chapter we survey the details of protein sequencing, and some aspects of the underlying biology and instrumentation necessary to understand this research. In addition, we describe the programmable hardware platform used in our research. Section 2.1 provides an introduction to basic genetics and protein synthesis. Section 2.2 outlines the process of Mass Spectrometry as it applies to the protein sequencing approach that our work is based on. Section 2.3 describes some of the complexities of the biological systems that must be handled in our work. This ordering of biological concepts is done in hopes of allowing the reader to get an understanding of the core concepts of protein sequencing before considering issues of practicality. This is followed by a description of prior work in genome-based protein sequencing and hardware acceleration of biological algorithms in Section 2.4. Section 2.5 concludes the chapter with a description of the structure and relevant details of our implementation platform.

2.1. Introductory Biology

A theme of this work is the interaction between DNA and proteins. DNA is the template for protein formation. To better understand how the details of the two are related, the following sections present the key concepts behind DNA and protein interaction.

2.1.1. Deoxyribonucleic Acid (DNA)

Often described as the blueprint or life, Deoxyribonucleic Acid (DNA) is the core of genetic content passed between generations of organisms. DNA is a determining factor in almost all aspects of life, from appearance to health. The importance of DNA is related directly to its role in the production of proteins.

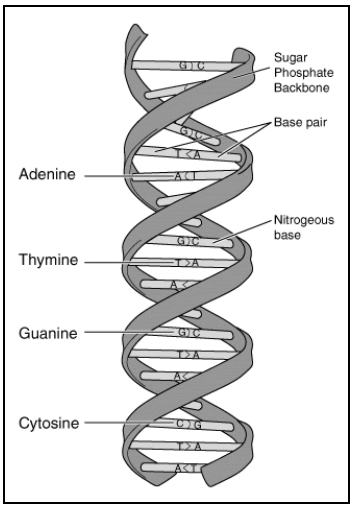


Figure 2-1: DNA Double Helix [24]

DNA is contained within the nucleus of a cell and exists in the double stranded structure shown in Figure 2-1 [24]. Each strand consists of a chain of nucleic acid molecules (also known as bases) linked by a phosphate backbone. There are four possible bases in DNA: Adenine (A), Thymine (T), Guanine (G), and Cytosine (C). Figure 2-1 shows that the bases on one strand bond to the other. This bonding can only occur between certain pairs. **A** will always bond with **T** while **G** will only bond with **C**; these pairings are referred to as complementary pairs or base pairs. Thus knowledge of the bases in one strand implies knowledge of the bases in the complementary strand [2], which is oriented in the opposite direction.

A strand of DNA can be represented as a string of ordered bases. The order of bases in the strand is important as DNA is used as a template in the creation of proteins and a change in the order of bases may result in the malformation of proteins. The DNA template is interpreted in units of three bases at a time – this set of three bases is known as a *codon*. Therefore DNA can also be thought of as a string of codons and it is these codon strands that act as templates for the creation of proteins. DNA strands within a cell are ordered into structures known as genes. Genes are DNA strands that are usually several thousands of bases long and each gene codes one or more proteins. Several genes are grouped together into larger structures known as chromosomes, and it is the set of chromosomes that is passed on as hereditary information between generations of

organisms [3]. The DNA sequence of all the chromosomes in an organism is known as its *genome* [24]. The hierarchical view of DNA in Figure 2-2 illustrates the relationship between these units.

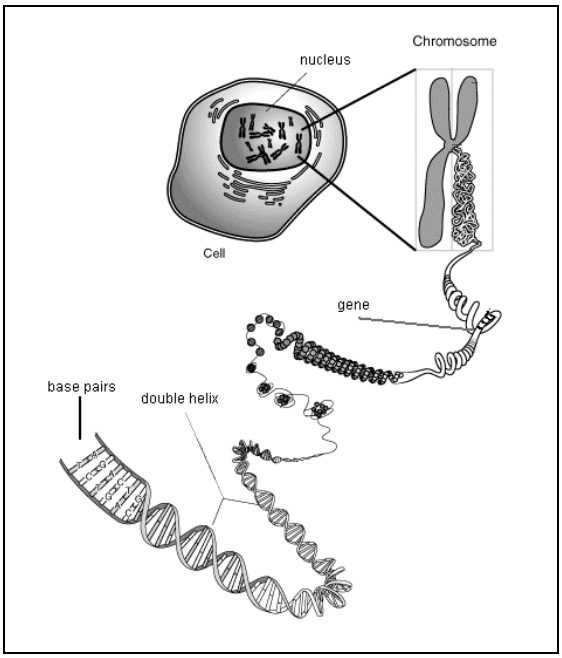


Figure 2-2 Genetic Hierarchy [24]

2.1.2. Protein Formation

The information stored in DNA governs the synthesis of proteins in an organism. Proteins are chemicals that provide both structural and enzymatic functions within a cell. They are required for everything from the formation of muscles and ligaments to the synthesis of various digestive enzymes. Almost every reaction within the body is some form of protein interaction, and so a better understanding of protein functions is clearly beneficial to biologists. It is the structure of a protein that determines its functionality and thus a great deal of effort has been directed towards determining the structure of every protein. Biologists can infer function from protein structure by comparing the structure of novel proteins with well-characterized proteins whose functions have already been

determined [7]. An understanding of how proteins are produced is essential to appreciate how their structure is determined.

Proteins are synthesized from DNA by a combination of processes known as *transcription* and *translation*. *Transcription* is the conversion of DNA to RNA (Ribonucleic Acid). RNA, like DNA, also consists of four bases, but Uracil (U) in RNA replaces Thymine (T) in DNA. For the purposes of this discussion we will treat Thymine and Uracil as equivalent molecules and only refer to Thymine. In essence, transcription results in the creation of a copy of the original DNA strand as shown in Figure 2-3.

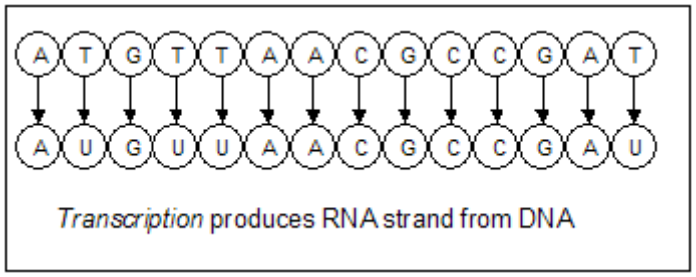


Figure 2-3: Transcription of RNA

The example in Figure 2-3 is simplified for clarity. The RNA strand that is transcribed from a DNA strand actually consists of the complementary bases, i.e., A is transcribed to U, C is transcribed to G etc. The key point to note is that the bases in the RNA strand can be inferred from the original DNA strand.

The RNA strands are then *translated* into proteins. This is done by structures known as ribosomes and transfer RNA (tRNA) that bond to the RNA strand converting groups of bases into molecules known as amino acids. Recall that the DNA (or RNA) strand is interpreted in codon blocks. Each codon, or set of three bases, represents a specific amino acid and the rules for translation are standard for most organisms including homo sapiens.

A table of codons and their corresponding amino acids is given in Table 2-1. To convert an RNA strand into a protein, it can first be thought of as a set of codons. The first base of a codon identifies the major row (T, C, A or G on the left side of Table 2-1), the second base identifies the major column, and the last base of a codon identifies the

specific codon and its corresponding amino acid. Consider the example of the codon TAC. The first base (T) indicates the first row, and the second base (A) indicates the third column. The final base (C) identifies the specific codon and its corresponding amino acid Tyrosine (Y). In this manner, any RNA codon strand can be *translated* to its corresponding set of amino acids, or protein strand.

First base of codon	Second base of codon: T	C	A	G	Third base of codon
T	TTT ***Phenylalanine (F)***	TCT ***Serine (S)***	TAT ***Tyrosine (Y)***	TGT ***Cysteine (C)***	T
	TTC ***F***	TCC ***S***	TAC ***Y***	TGC ***C***	C
	TTA ***Leucine (L)***	TCA ***S***	TAA ***STOP***	TGA ***STOP***	A
	TTG ***L***	TCG ***S***	TAG ***STOP***	TGG ***Tryptophan (W)***	G
C	CTT ***Leucine (L)***	CCT ***Proline (P)***	CAT ***Histidine (H)***	CGT ***Arginine (R)***	T
	CTC ***L***	CCC ***P***	CAC ***H***	CGC ***R***	C
	CTA ***L***	CCA ***P***	CAA ***Glutamine (Q)***	CGA ***R***	A
	CTG ***L***	CCG ***P***	CAG ***Q***	CGG ***R***	G
A	ATT ***Isoleucine (I)***	ACT ***Threonine (T)***	AAT ***Asparagine (N)***	AGT ***Serine (S)***	T
	ATC ***I***	ACC ***T***	AAC ***N***	AGC ***S***	C
	ATA ***I***	ACA ***T***	AAA ***Lysine (K)***	AGA ***Arginine (R)***	A
	ATG ***Methionine (M) or START***	ACG ***T***	AAG ***K***	AGG ***R***	G
G	GTT ***Valine (V)***	GCT ***Alanine (A)***	GAT ***Aspartic acid (D)***	GGT ***Glycine (G)***	T
	GTC ***V***	GCC ***A***	GAC ***D***	GGC ***G***	C
	GTA ***V***	GCA ***A***	GAA ***Glutamic acid (E)***	GGA ***G***	A
	GTG ***V***	GCG ***A***	GAG ***E***	GGG ***G***	G

Table 2-1: The Genetic Code – Mapping DNA to Amino Acids

Note that there is redundancy in the coding, as there are 64 codons and only 20 amino acids. In some of these cases the last base in the codon can be treated as a wildcard. For example the codon set GC* codes for Alanine, regardless of the last base. Recall that genes are simply long strands of DNA that can be grouped into codons and proteins are amino acid chains. Using this table, it is possible to translate genes to proteins and vice versa. An example of this process is presented in Figure 2-4.

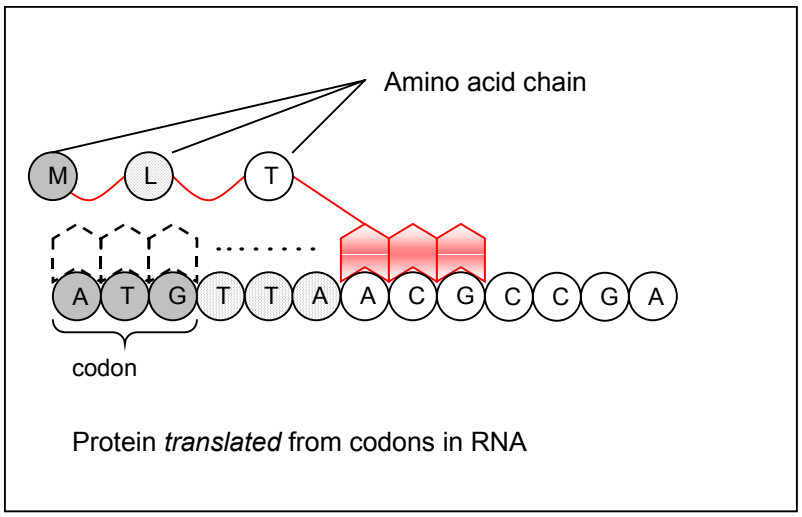


Figure 2-4: Translation to protein strand

The ribosome unit traveling down an RNA strand physically carries out the nucleic acid to amino acid translation process and synthesizes the protein by adding the amino acid corresponding to the codon being processed. In the example in Figure 2-4, the tRNA reads A as the first base, T as the second, and G as the third base of the codon. The tRNA adds the amino acid Methionine (M) to the current protein chain. The ribosome proceeds along the RNA strand until a STOP codon is reached, and a full protein is synthesized.

2.2. Mass Spectrometry Based Methods of Protein Sequencing

Recall that our ultimate goal is to sequence a protein, i.e. to identify the order of the constituent amino acids in a protein sample. Over the last few decades mass spectrometry has become the method of choice for high throughput protein sequencing [10]. A *Mass Spectrometer* (MS) is a tool that takes a chemical or biological sample as input and measures the masses to charge ratio of the sample's constituent molecules. The mass to charge is used to calculate the masses of the molecules in the sample and these masses are then used to determine the identity of the molecules. A more detailed description of this process is presented in Figure 2-5.

The MS identifies particles in the input sample by ionizing them and allowing the charged ions to fly over a detection plate. Identification of the ions relies on the fact that

heavier ions will not travel as far lighter ions and will thus fall to the detection plate sooner, as illustrated in Figure 2-5. Based on the ion's charge and position along the detection plate, the mass of the ion can be resolved [11].

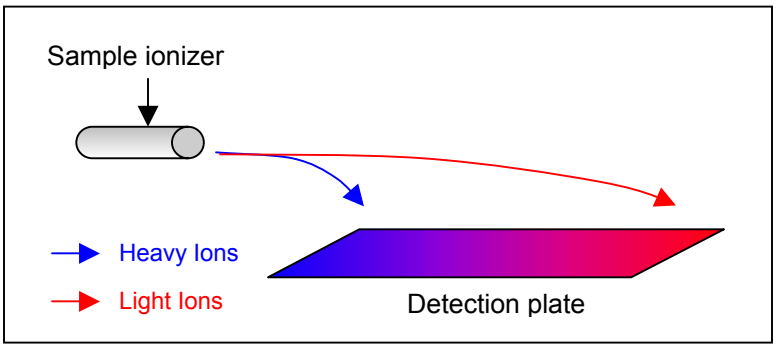


Figure 2-5: Mass Spectrometer

These measured masses are compared against known molecular masses to establish the identity of the molecules in the sample.

There are several different types of mass spectrometry, many of which are used for protein sequencing [9] [21]. One such approach, which will be the focus of this research, is the technique of tandem mass spectrometry [6]. An overview of this approach is presented in the following section to help understand the capabilities and limitations of the process.

2.2.1. Tandem Mass Spectrometry

Tandem Mass Spectrometry (often abbreviated as MS/MS as it uses two MS ion separation chambers) is a common technique used in protein identification studies. In preparation for MS/MS analysis, protein samples are treated to ensure that the MS devices can operate on them.

Since proteins are large chains that are several hundred amino acids in length, they are heavy (on a molecular scale) and most MS instruments cannot analyze them in their intact form. For this reason, proteins are usually broken down into smaller fragments known as *peptides* by a process known as *digestion* [42]. Digestion occurs by treating the protein sample with a *proteolytic* or digestive enzyme, which will cut the proteins in the

sample at certain known amino acid bonds. One such enzyme is *trypsin,* which is known as a specific enzyme for its property of cleaving proteins at specific amino acids (trypsin cleaves after the positively charged amino acids Arginine (R) and Lysine (K) provided that neither is immediately followed by Proline (P) in the protein sequence). The peptides created by trypsin digestion are referred to as tryptic peptides. An example of protein digestion is shown below in Figure 2-6. For simplicity, only a single protein is shown, but a real biological sample may have as many as 40 proteins in it [60].

MAVRAKPCOKLHNWF
Original protein in sample

MAVR AKPCOK LHNWF
After digestion – 3 smaller tryptic peptides (note cleavage after K and R but not KP)

Figure 2-6: Trypsin Digestion of Proteins

This group of tryptic peptides is now passed to the tandem MS for analysis. An overview of this process is given in Figure 2-7. The tandem MS or MS/MS consists of two MS units [12]. The first MS is used to measure the masses of the tryptic peptides and select individual peptides to send to the second MS (step 2 in Figure 2-7). The first MS produces a list of masses of all the tryptic peptides, which is known as the list of precursor or parent ions and will hereafter be referred to as the precursor ion scan (PIS) [12]. However, note that the first MS stage also contains peptides that were not in the original tryptic peptide set. These unwanted peptides might originate from a number of sources, such as proteins from the MS operator's skin through careless handling, contaminant proteins that could not be separated from the sample during preparation and other sources of contamination. This noise appears on the PIS list and makes it difficult to distinguish the interesting peptides from the contaminants.

The second MS breaks the peptide selected by the first MS into groups of amino acids. These groups consist of chains of one, two or more amino acids, effectively generating the substrings of the selected peptide. These groups are then ionized and the ion masses are used to deduce the identity and sequence of the amino acids in the peptide [43]. The

details of this process are described in Appendix A for the interested reader. Once the sequence for a single peptide is obtained, the user selects another peptide from the first MS and the sequencing step (step 3) is repeated. An important detail to note is that it takes between 500 ms to 1 second before the next peptide can be selected for sequencing [49]. Caution must be exercised in choosing the subsequent peptide; if a contaminant is chosen instead of a peptide of interest, both the sample and sequencing time will be wasted. Note that typical samples contain many proteins that must be analyzed [60]. After all the peptides from each of the proteins of interest are sequenced, they must be assembled to obtain the full protein sequence. This is a computationally intensive step, often requiring the manual intervention of an MS technician with experience in protein sequencing.

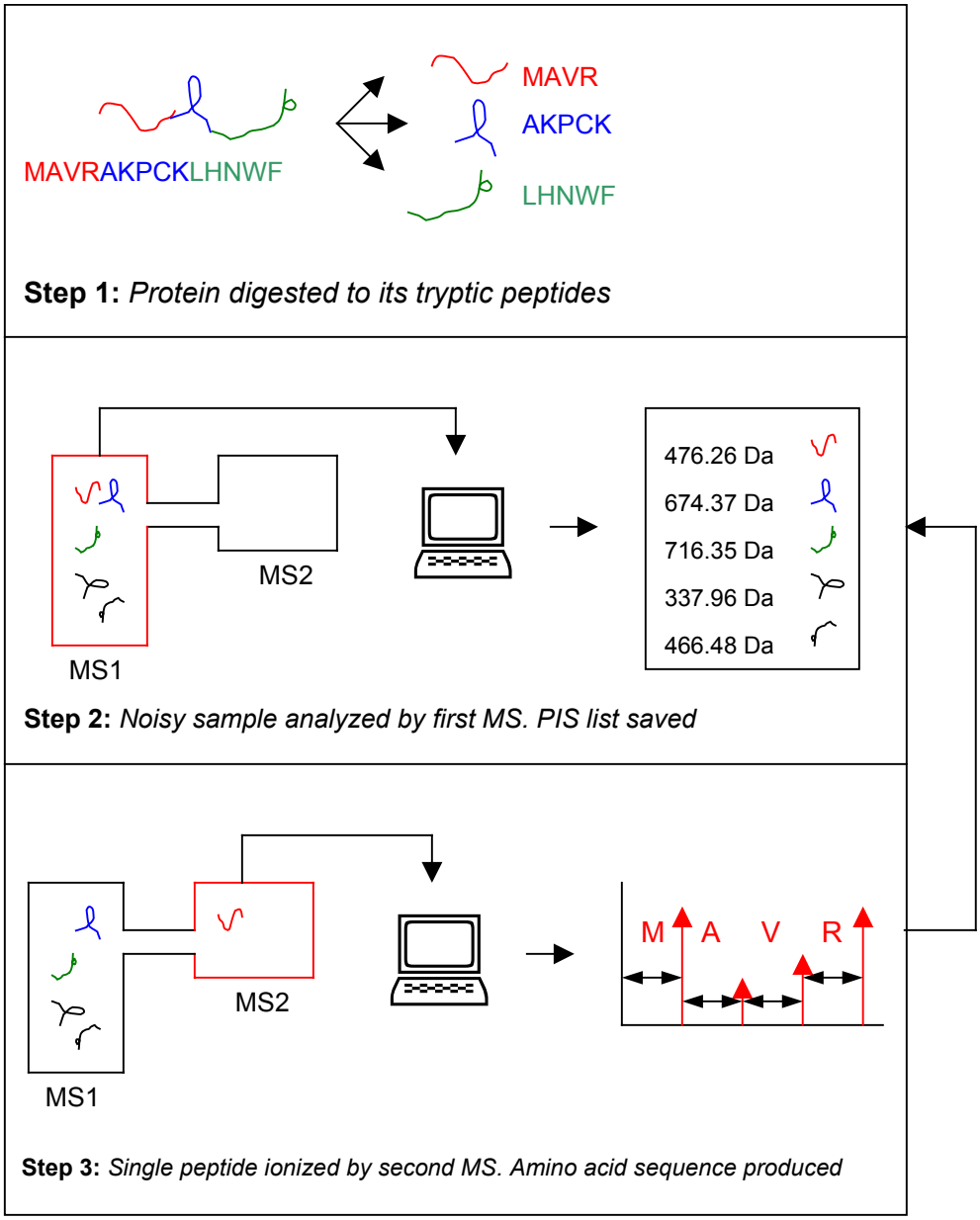


Figure 2-7: Tandem Mass Spectrometry Flow

There are three key limitations to this process:

1. *The sequencing step has to be repeated for each peptide in the PIS.* In the simple example shown in Figure 2-7 there are only two additional peptides to sequence after the first sequence is obtained. However, proteins can have between 50-900 tryptic peptides each [59] and the sequencing process in the second MS will have

to be repeated for each peptide. With multiple proteins in a sample, there may be thousands of peptides that have to be individually sequenced. Also, multiple sequencing steps will consume larger volumes of the sample. Since it is difficult to acquire large volumes of purified biological samples for medical experiments, conservation of the sample is critical [47].

2. *Sample preparation, as any chemical process, is subject to contamination*. It is impossible to prepare a protein sample that does not contain trace amounts of contaminants from the environment. These "noisy" samples will also appear in the MS output and there is no means of distinguishing them from the peptides to be sequenced. Further, a real protein sample will contain a great deal of noise, making it harder to identify relevant target peptides [47]. Therefore, in the cycle between step 3 and step 2 in Figure 2-7 there is no information that aids us in picking subsequent peptides to sequence. Any time spent accidentally analyzing these noisy data elements wastes more of the input protein sample.

3. *The peptides, once sequenced, must be placed in order*. Once all the sequences are obtained, a final step is needed to place the peptides in the correct order. As mentioned above, this is a demanding process, which frequently requires manual intervention.

As mentioned in Chapter 1, it is not strictly necessary to sequence every peptide in a protein to identify it. If the sequence of the protein is known and stored in a protein database, a few peptides can be used as a *fingerprint* to uniquely identify their parent protein [9]. However, this approach requires that the protein sequence exist in the database. As mentioned, our aim is to accelerate *de-novo* sequencing experiments, i.e. experiments where the goal is to sequence a hitherto unknown protein. By definition, a protein that has not been studied before cannot exist in a protein database; therefore the fingerprint approach cannot be implemented directly.

Large computer clusters are now available to improve analysis thereby lessening the restrictions imposed by the other two limitations, namely sample contamination and

peptide ordering. Regardless, de-novo protein sequencing still cannot be performed as a real time operation.

The input protein sample is usually difficult to obtain and small in quantity [48] especially in *de novo* experiments. The ionization process described above is destructive and consumes the sample rapidly. Thus being able to quickly distinguish between noise and interesting peptides would allow researchers to minimize the amount of sample required. In addition one could greatly improve the throughput of protein sequencing by reducing the need for manual intervention and reducing the number of peptides that have to be retrieved and sequenced from the first MS. With these goals in mind we consider a different approach to protein sequencing.

2.2.2. A New Search Strategy

With the recent successful sequencing of the human genome, the set of all human genes is now available to researchers [58][59]. Section 2.1.2 described how genes act as templates for the creation of proteins. In theory it is possible to derive the sequence of all the possible proteins of an organism given its genome [1] [16]. This implies that a complete protein database can be built, which then reopens the possibility of performing a *peptide mass fingerprint* (PMF) search. The PMF technique as described earlier, uses a few peptides as a fingerprint to uniquely identify its protein of origin.

To see how this approach works, let us consider the sequence that is output by the second MS (step 3 of Figure 2-7). This peptide was part of an intact protein before it was digested by trypsin and analyzed by the mass spectrometer. Since every protein must be synthesized from a gene, the human genome must contain the gene that originally coded the sample protein. Once this gene is located, it can be translated to its amino acid sequence using the codon translations given in Table 2-1. Consider the example in Figure 2-8: If the sequence produced by the second MS is "MAVR", it can be reverse-translated as follows:

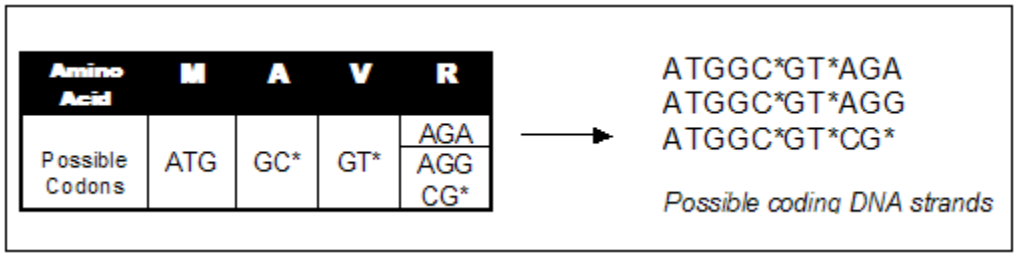


Figure 2-8: Reverse Translation

Note that the amino acids A, V and R can be synthesized by multiple different codons; thus there are many possible DNA strands that can create this peptide. The gene that coded the protein in the sample must have one of these DNA strands as its substring. If the possible DNA coding strands in Figure 2-8 are submitted as queries to a genome database, the true coding gene can be located. Then, using the information in Table 2-1, this gene can then be translated to a protein. However the human genome is a sequence of approximately 3.3 billion base pairs and a search for 3 strings of 12 bases (including wildcards) as shown above will likely yield multiple matches. If there are numerous locations in the genome that match the coding strands, we must resolve them to see which the true coding gene is. To this end we can utilize more information from the MS. From the first MS we have the precursor ion scan (PIS) list. Recall that the PIS is a list of the masses of the tryptic peptides in the protein sample. We will refer to this as the *true PIS*, as it is the set of masses that have been positively identified by the MS.

The true PIS contains mass information about every peptide in the protein sample and its can be used to resolve the problem of multiple matches described above. If each of the matching genes is translated to its corresponding protein, and each of these proteins is cleaved into its tryptic peptides, the masses of these tryptic peptides can now be calculated. In essence, we generate a *hypothetical PIS* for every matching gene. The hypothetical PIS that shows the greatest similarity to the true PIS corresponds to the original protein. Variations of this approach have been proposed by several researchers [1],[8]. An algorithm that implements this searching strategy is outlined in Figure 2-9.

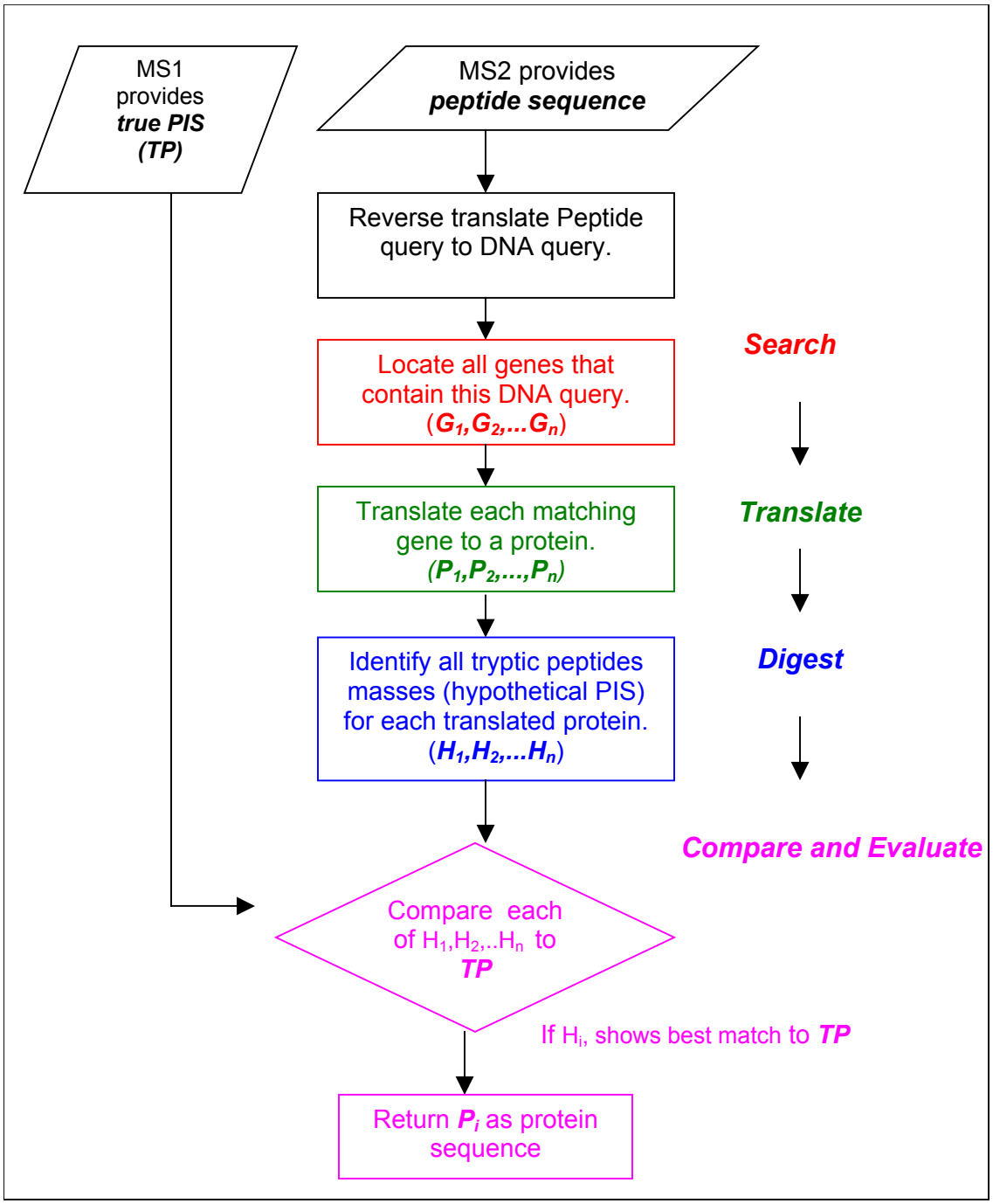


Figure 2-9: Algorithm Outline

To clarify the steps of the algorithm consider the example in Figure 2-10. The second MS produces the sequence of a single peptide (magtr) and the algorithm attempts to identify

the full sequence of the protein that this peptide originated from. To do this, the peptide is first reverse translated using the information in Table 2-1.

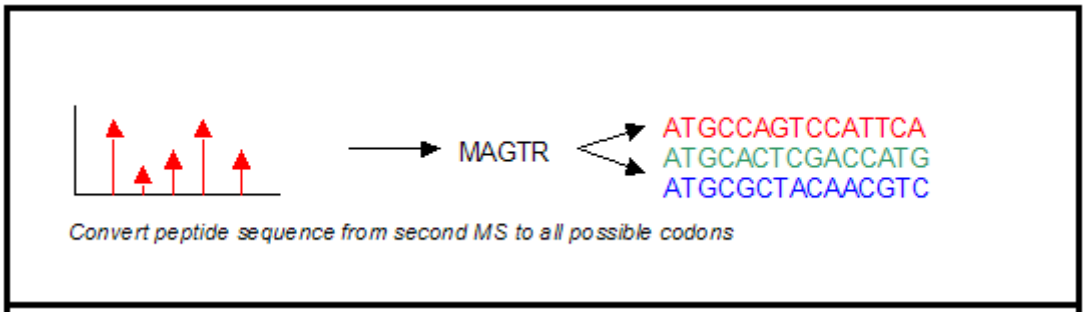


Figure 2-10: Searching the Genome Database

The DNA queries thus generated are located throughout the genome. Note in Figure 2-11, that we locate two possible genes in the database that contain the DNA query. Both of these genes are translated from DNA to amino acids, once again using the information in Table 2-1. We know that digestion by trypsin cleaves a protein at the K and R amino acids (if they are not followed by P). Using this rule, we identify all tryptic peptides from both of the translated proteins and calculate their masses. This generates two *hypothetical PIS* sets. This corresponds to the *translation* and *digestion* steps in Figure 2-9.

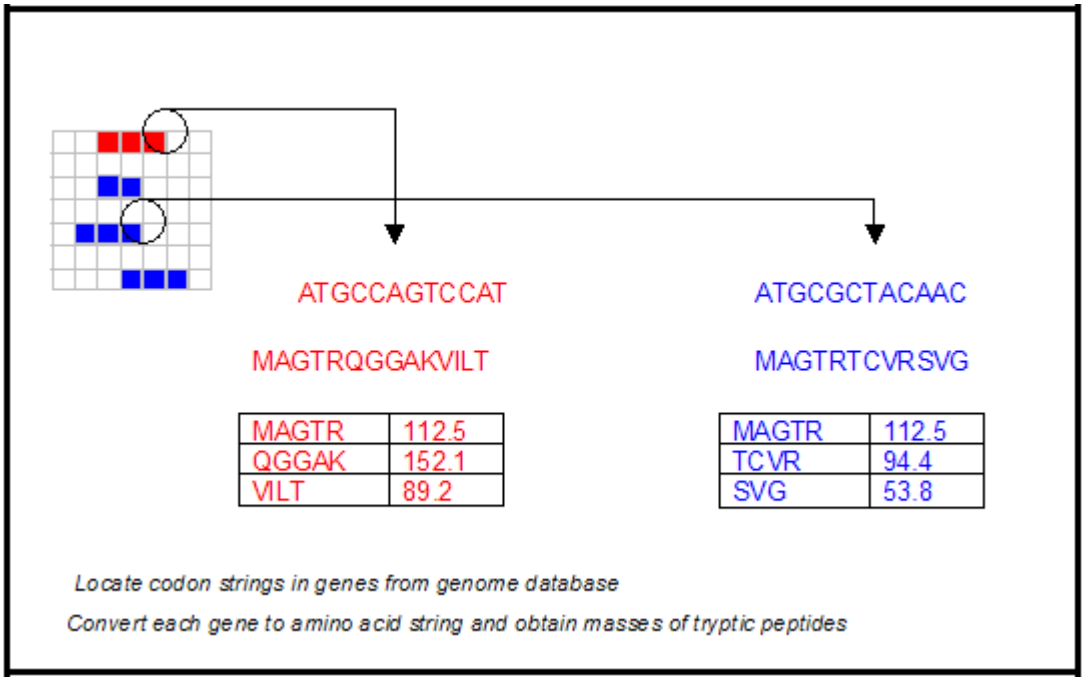


Figure 2-11: Translate genes and digest translated proteins

Each hypothetical PIS is then compared to the true PIS and it is clear that the gene corresponding to the protein "MAGTRQGGAKVILT" matches the true PIS more closely and is thus identified as the true coding gene, as shown in Figure 2-12.

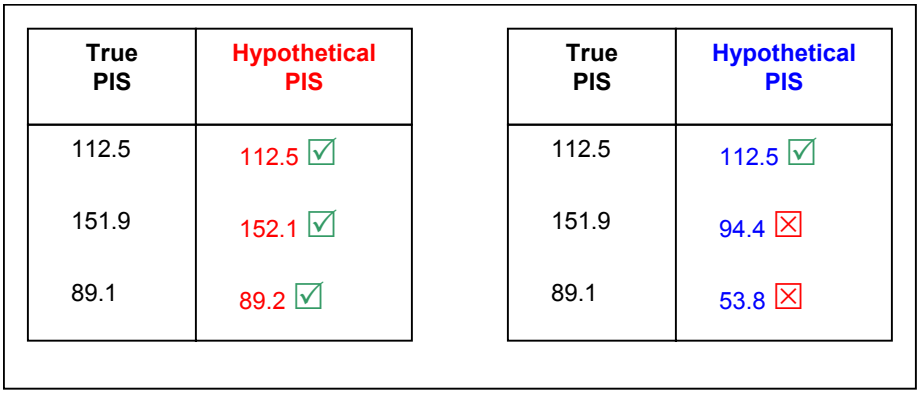


Figure 2-12: Compare digested peptides to PIS

Observe that identifying the coding gene in this manner implies that the protein sequence can be obtained by simply translating the gene. Unlike the traditional approach described

in Section 2.2.1 only one peptide from a protein (or two or three at most [36]) need be analyzed to obtain the full protein sequence.

There are a number of advantages to the technique described above:

- *Less sample is consumed*: If only a few peptides have to be identified, a smaller quantity of protein can be analyzed.

- *Sequencing time is shorter*: Using this approach, the multiple sequencing steps and final peptide ordering phase described above can be avoided allowing the sequencing speeds and overall MS throughput to be greatly increased.

- *We can make better decisions*: Given that we identify the full protein sequence, we can generate a list of peptide masses we expect to see if this is the protein being analyzed by the MS. When this list is compared against the PIS it will be easier to distinguish between true proteins in the sample and artifacts generated by noise from contaminant proteins as we now know what peptide masses should appear in the PIS. The cycle between step 2 and step 3 in Figure 2-7 is now a *feedback path* containing information in the form of the hypothetical PIS. This information can be used to identify masses in the true PIS and eliminate them from further analysis. Thus only peptides that we cannot identify with the hypothetical PIS need to be considered, drastically reducing the overall number of sequencing repetitions (step 3 in Figure 2-7) that have to be performed.

2.2.3. Requirements of the New Approach

To implement this approach to peptide sequencing four key features are required:

- *A method of locating potential coding genes within the genome.* A database search engine capable of locating query DNA strands within the genome is crucial to the functioning of this algorithm.
- *A method of translating the genes to find the masses of tryptic peptides they generate.* Once potential genes have been located, they must be translated and digested *in silico* (by computation) to obtain the masses of the tryptic peptides.
- *A method of comparing calculated tryptic peptide masses with masses detected by the first MS*. The tryptic peptides generated from each gene must now be compared with the PIS list of masses. Using a scoring algorithm, every matching mass can be ranked and thus a score for each gene match can be generated to help the user to quickly identify the true coding gene.
- *Fast overall processing time.* Since we will have to sequence multiple proteins in any realistic sample, we must be able to identify proteins in the time that the second MS generates a sequence. From [49] we know that the average time before the second MS can be reused to sequence another peptide is between 0.5 and 1 second. Therefore, any useful implementation of the above algorithm using the feedback path described in Section 2.2.2 must be able to produce a protein sequence within this timeframe.

Searching through the 3.3 billion base human genome [58] in a fraction of second requires enormous throughput. Fortunately this kind of search is highly parallelizable in both software and hardware. Applications of this nature are good candidates for custom hardware implementation, thus our goal in this research is to design a hardware system that meets the requirements of the sequencing algorithm as described above.

2.3. Practical Considerations

In Section 2.1 the basics of protein formation were explained. The methods of DNA translation described are true for simple organisms. However, for more complex organisms such as humans there are additional processes that affect protein formation. In addition, there are peculiarities of the genome database that must be addressed if it is to be used in the manner described in Section 2.2.2.

2.3.1. Reading Frames and Complementary Strands

In Section 2.1.2 an example of protein formation was shown. In it, the tRNA unit started at the codon ATG and moved in units of one codon (3 bases) along the RNA strand. In this simple example, the tRNA started at the beginning of the strand. However the genome is stored as a large set of DNA strands and while the translation starting points of many genes are known, many remain to be discovered. In short, it is extremely difficult to predict at which base protein translation actually begins [41]. Consider the example below.

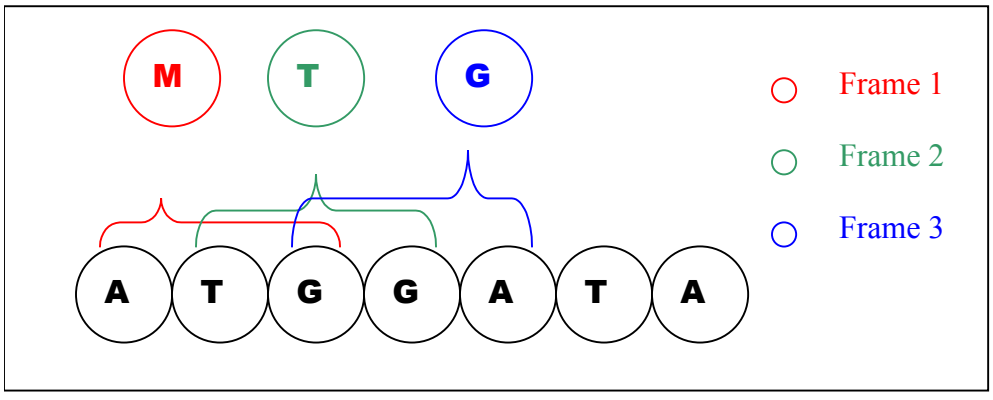


Figure 2-13: Reading Frames

Three different possibilities are shown in Figure 2-13. If protein translation starts at the first A, the first amino acid will be M (Methionine) and every subsequent codon will be processed with reference to ATG as the first codon (i.e. in this case the next codon will be GAT). If however, translation began one base ahead at the first T (using TGG as the first codon) the first amino acid would be T. The next codon would then be taken from this reference point (i.e. it would be ATA). Each of these possibilities is known as a *reading frame*. If translation begins at the first base in the sequence it is designated as Frame 1, if it begins at the second base it is designated as Frame 2 and so on. Note that in a given strand there are only three frames to consider. If translation began at the fourth base, it begins reading at Frame 1 with the difference that one codon (or amino acid) has been skipped [40].

Another detail to consider is that the Human genome is stored as single strands of DNA, i.e. the complement of a strand is not stored since it can be inferred from the original strand. A protein may be synthesized from either the original strand or its complement, and to account for this we must generate the proteins for both the strand stored in the genome database and its complement. It must be noted that the direction of translation is reversed for the complementary strand. The effect of this is illustrated in Figure 2-14.

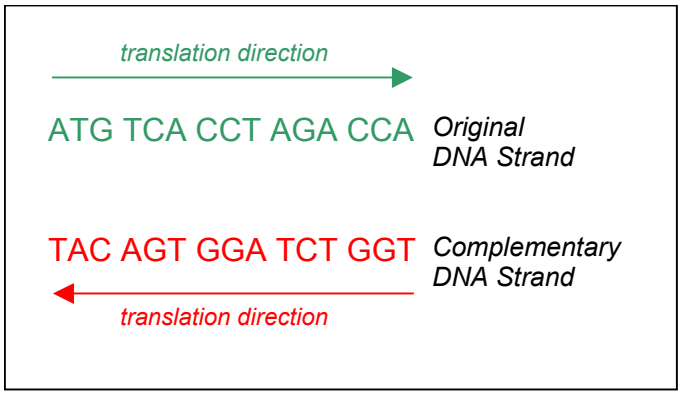


Figure 2-14: Translation of a Complementary DNA Strand

As stated in Section 2.1.1, the complementary DNA strand is a copy of the original with the Adenine (A) replaced by Thymine (T) and the Guanine (G) replaced by Cytosine (C). Figure 2-14 also shows that the direction in which protein translation proceeds is reversed

for the two strands. Note that the presence of the complementary strand implies that there are an additional 3 frames. The three frames of the complementary strand are designated Frame 4, Frame 5 and Frame 6 respectively [40]. Each of these frames must also be included with the original three in any calculations that occur as a result of gene to protein translation.

2.3.2. *Alternative Splicing*

In Section 2.1.2 the process of protein translation was described. It was implied that the tRNA unit traveled down the gene and based on the codons, it created a specific amino acid chain. This is the basis for translation, but in complex organisms, an additional process known as splicing occurs. Consider the earlier example from Figure 2-4, reprised in Figure 2-15.

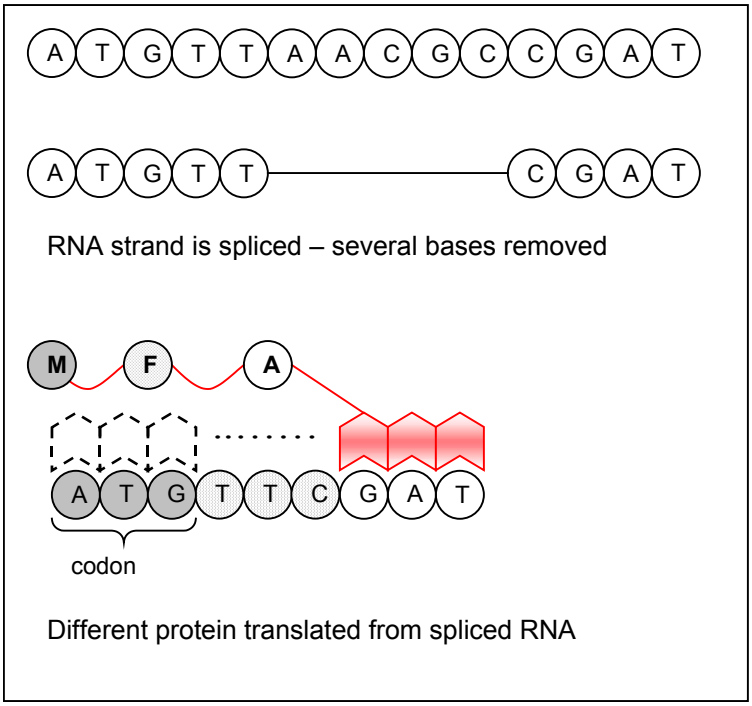


Figure 2-15: Alternative Splicing

After the original gene is transcribed from the DNA to an RNA strand, when splicing occurs, a small subsection is removed. In Figure 2-15, five bases are removed from a region of the RNA strand. The new strand is joined at the spliced bases (in this case **T**

and **C**) to form a new shorter strand. The mechanism behind splicing is not fully understood by biologists and is an active area of research. Since there is no way of determining splice sites a priori, it is not currently possible to translate a gene using only a codon table. However, only 30% of all genes produce alternatively spliced proteins [61][62]. It should be noted that this figure is an assumption based on current knowledge and that several genes exhibit far more splicing. For example 55% of all genes in chromosome 7 are alternatively spliced [52]. The approach we use in this work relies on direct translation of genes to identify proteins without accounting for splicing. However, an average protein is not spliced at many locations along its structure. If a spliced protein is chemically digested as described above, only tryptic peptides formed from a splice site will not have a corresponding coding sequence in a gene. The majority of tryptic peptides will not be from splice sites and thus can be detected by this approach. This is sufficient to confidently identify the gene of origin. Once the coding gene has been identified, more complex analysis may be done to attempt identification of the splice locations. The key notion here is to identify the true coding gene as rapidly as possible. It should be noted however, that of the 30% of genes that alternatively spliced, 98% follow canonical rules and many of these splice variants can be determined [62].

2.3.3. Unknown Bases in the Genome

One key detail that should be stated at the outset is the presence of ambiguities in the genome databases. In addition to the A, T C and G molecules of DNA, genomic databases also consist of an ambiguous base character '**N**' which stands for a**N**y of the four bases. These unresolved bases exist in genome databases as a result of the high throughput sequencing techniques that are commonly used, and while they will ultimately be resolved, the fact remains that ambiguous regions exist in biological databases [35].

2.3.4. Repeat Sequences in the Genome

Another biological reality is the presence of repeated DNA sequences throughout the genome. These *repeats,* as their name implies are merely sections of the genome that have a sequence of bases repeated continuously for a long stretch within a chromosome. Usually a 6 to 10 nucleotide sequence is repeated several thousand or even a million times. [37][38]. If such a DNA sequence is translated to amino acids, the peptide string will produce a set of repeating tryptic peptides upon digestion. Recall that we will be comparing the masses of calculated peptides to those detected by the MS. If a reasonable number of the calculated masses within a gene match those detected by the MS we regard the gene as good candidate coding gene for the sample protein. In a purely random DNA string (without repeats) one would not expect many matches to a query. However, consider the effect of a repeat sequence on the matching process. If a mass detected by the MS matches the mass produced by a repeat sequence it will produce a great number of matches simply due to the repetitive nature of the DNA in this region. It is apparent that an erroneous high score may be generated for a match due to repeats. One common solution to reduce these false positives in current biological database system is to remove or mask repetitive DNA sequences in the genome database. This simple approach is reasonable, as repeats generally do not code proteins. However, a great deal remains unknown about the genome and it would be ideal to search the genome in its unadulterated form. For this reason, we use the entire genome including repeats and provide an extension to the third requirement in Section 2.2.3 The comparison method should calculate scores that do not merely indicate the number of matching masses, but also reflect whether the match was made to peptide that appeared very frequently within a gene (for example by a repeat) or to a peptide that appeared relatively infrequently.

Various database-searching algorithms such as MOWSE use the *frequency of occurrence* of a peptide as a measure of its *significance* [9]. Since the probability of a real match between a query and the genome is considered statistically improbable [9], a match that occurs frequently can be treated as insignificant or a random match. The match scoring system will incorporate both the frequency of occurrence of individual peptides and the number of matches in the final score.

2.3.4.1. Significance of Matches

The concept of significance described above can best be understood by the example illustrated in Figure 2-16

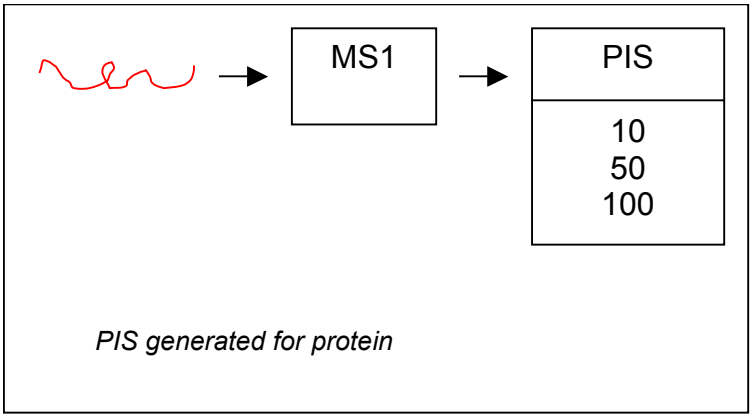


Figure 2-16: PIS of protein is generated by MS

In Figure 2-16, the protein sample in the MS is digested to 3 peptides whose masses are listed in the PIS. Peptide masses are usually defined in Daltons (Da) where 1 Da is the mass of a single Hydrogen atom. The PIS in Figure 2-16 indicates that peptide 1 has a mass of 10 Da, peptide 2 has a mass of 50Da and peptide 3 has a mass of 100 Da. For simplicity, we ignore any contaminants in the sample and only consider a single pure protein sequence.

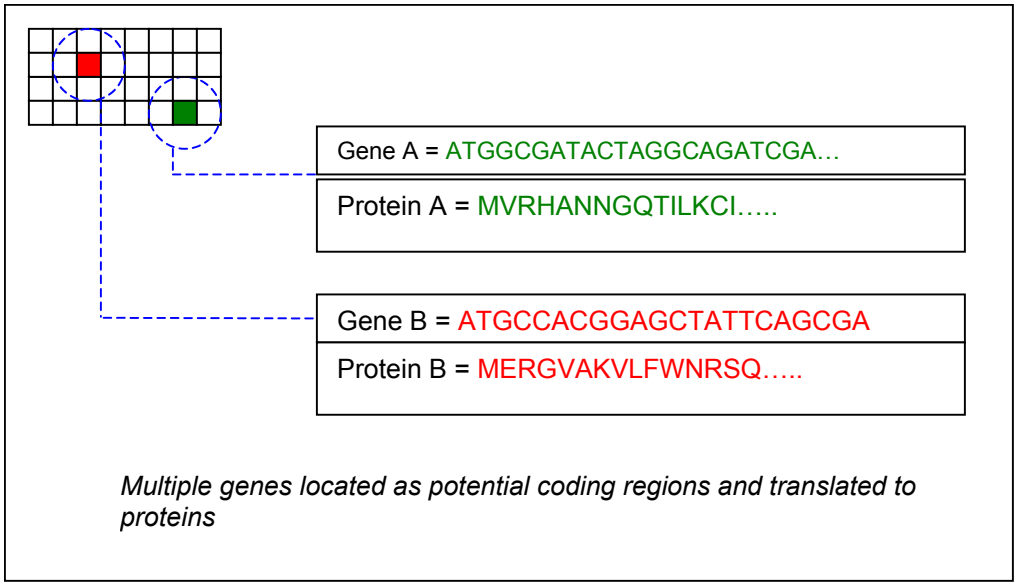


Figure 2-17: Two Potential Coding Genes are Located in the Genome

The sequence of a single peptide is generated and used as a query to the genome database. Figure 2-17 shows two candidate genes that may have coded the query peptide. Each of these genes is translated to a protein that is then split into its tryptic peptides. The masses of these peptides are then calculated and a histogram of peptide masses is built. The histogram illustrates how frequently a peptide within a certain mass range occurs in a given protein. This is the "*frequency of occurrence*" referred to in the previous section.

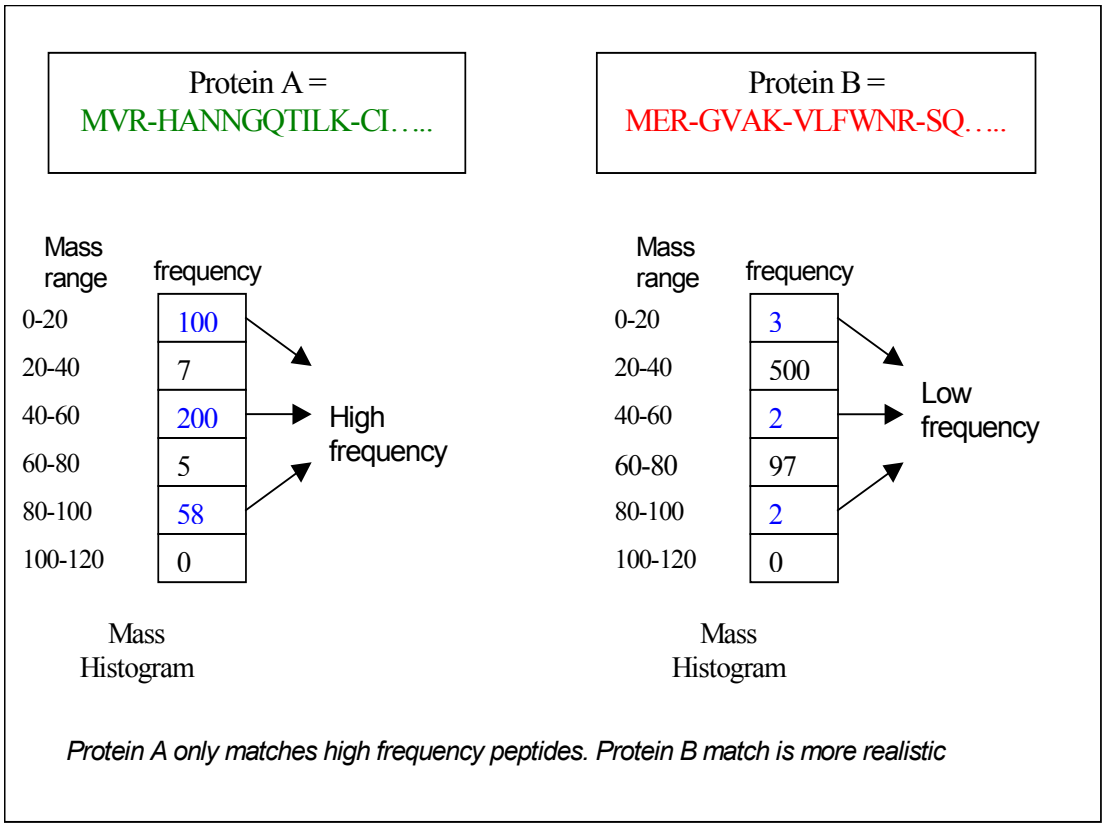


Figure 2-18: Identification of Significant Match

Gene A translates to a protein (protein-A) with a wide distribution of masses. There are 100 tryptic peptides that range in mass from 0 Da to 10 (the range of peptide 1), 200 in the 40-60 range (the range of peptide 2) and 58 in the 80-100 range (the range of peptide 3). Clearly the unknown protein in the MS may exhibit a mass match to some of the peptides in protein-A. However consider protein-B, which has only 3 fragments in the 0–10 range, 2 fragments in the 40-60 range and 2 in the 80-100 range. The distribution of mass is shown in Figure 2-18. Note that only the mass ranges into which the MS masses fall are considered, since these are the only ranges in which a true match can occur.

With a large number of peptides in the matching range, protein-A is hardly significant, as a mass match could have occurred simply by chance due to the overwhelming number of peptides that fell into the matching mass ranges. Protein-B on the other hand, has very few masses that fall into the matching range. If the calculated masses in this range meet the user specified threshold, this is a significant result as these matches are far less likely to have occurred by chance. Consequently the definition of a significant match hinges on

the frequency of occurrence described in Section 2.3.3. We define a match as a mass match that occurs between an MS detected peptide and a calculated peptide. A significant match occurs if the mass of the calculated peptide does not appear frequently within its constituent protein. A number of techniques to compute significance exist for biological database search algorithms. We adapt the approach proposed by the MOWSE algorithm for our purposes [9]. Note that scoring functions such as MOWSE are extremely sensitive to the data they operate on [46]. Biologists often spend a great deal of time developing scoring schemes for specific comparisons and warn that even advanced scoring schemes will suffer high rates of false positives when used with highly random data [63]. However the MOWSE algorithm used in peptide database searches suits our requirements well, and can be tuned by trial and error to work with the approach proposed in this work.

2.3.4.2. The MOWSE Algorithm

A number of algorithms that compare peptides from MS/MS expriments to protein databases are commercially available. For example the Sequest [68] MS/MS search attempts to correlate the theoretical spectra of proteins in a database with those identified by the MS. A protein match is ranked by using a count of the number of matching peptides and the sum of the intensities of these peptides. The Sonar MS/MS algorithm [67] also uses intensity information in ranking matching peptides. The algorithm described in Section 2.2.2 relies only on the masses and ignores the intensity information provided by the MS. Thus we adapt the MOWSE algorithm, in our implementation as it mostly closely meets our requirements. The MOWSE algorithm is targeted towards peptide mass databases that are used in Peptide Mass Fingerprinting (PMF) experiments. However, this is comparable to the approach described in Section 2.2.2, which is essentially a peptide mass search. The difference is that the approach in Section 2.2.2 obtains its protein database by translating the genome, while PMF experiments used databases of sequenced proteins.

The traditional MOWSE algorithm accepts a list of peptide masses detected by the MS and searches through a protein database to find a protein that may generate the same

peptide masses. However, MOWSE does more than just count the number of matching peptides. It also assigns a statistical weight to each peptide match by using the MOWSE factor matrix **M** [9]. In our approach **M** can be thought of as an array representing a histogram of masses. Each element of the array is a bin representing a range of masses. The bins record the number of peptides that fall into their mass range; in effect they record the *frequency of occurrence* of peptides of a certain mass. These frequencies are normalized by dividing them by the most frequent range to produce the final **M**.

$$m_i = \frac{f_i}{|f_{(\max)}|}$$

where $\mathbf{f}_i$ is the frequency of element i.

This is then used to calculate the score of an individual peptide match as:

$$Score = \frac{K}{(\prod_{i=1}^{n} m_i)}$$

where K is a scaling factor that can be set by the user, and n is the number of matches

This is not the traditional MOWSE scoring function, as the original was designed to operate on peptide sequences and not on translated DNA sequences. Nevertheless, this formula still captures the essence of the scoring algorithm, which is the frequency information provided by the MOWSE factor matrix.

To realize the scoring function above for a gene window, certain aspects of the computation must be adapted for hardware implementation.

$$\prod_{i=1}^{n} m_i = \frac{f_{m_1}}{f_{\max}} \times \frac{f_{m_2}}{f_{\max}} \times \Lambda \times \frac{f_{m_n}}{f_{\max}}$$

$$= \frac{\prod f_m}{(f_{\max})^n} \text{ where } \boldsymbol{n} \text{ is the number of matches.}$$

Thus, three key components define the score: the *product term*, the *maximum frequency* and the *number of matches*. For every mass range [1...n] in which we detect a match, we

take the product of the normalized frequency of the range. If a match occurs in a highly frequent range, the $\prod f_m$ term (and correspondingly the score) will be higher. Conversely, a match to an infrequent range will produce a low score. This "smaller-is-better" value for $\prod f_m$ can be used to assign a significance value to a match.

2.4. Prior Work in Software and Hardware Based Genome Searching

Researchers have considered using the genomes of organisms for protein sequencing in the past [1]. As mentioned in Chapter 1 custom hardware has also been used to accelerate various applications. However, we believe that this is first time the hardware implementation of the sequencing scheme described in Section 2.2.2 has been published. It is instructive to look at past attempts to use genomic data in both software and hardware contexts.

2.4.1. Software Searches of the Genome

Choudary et. al. have performed searches of the human genome using mass spectrometry data in the manner described above. Their research showed it to be a time consuming method prone to errors due to the quality of the genomic sequence and the immense volume of random data in an organism's genome [1]. Nevertheless, they note that with high quality MS data the genome could prove a useful tool in identifying novel coding sequences. However the size of the genome, coupled with memory bandwidth limitations on conventional processors restricted the speed of this method. The study in [1] showed search times of 3.5 minutes on a 600 MHz Pentium processor. This can be optimistically extrapolated to a search time of approximately 1 minute on a 2.4 GHz processor assuming that memory speeds scale with the processor. Recall that a practical implementation of the algorithm in Section 2.2.2 must be able to identify the coding gene within 1 second to avoid costly instrument downtime.

Despite the challenges posed above, complete high quality drafts of the human genome have been produced since the work in [1] and many of the errors due to erroneous and incomplete genomic data can now be resolved. Furthermore other studies such as those conducted by Kumar et. al [26] suggest that a wealth of information will go overlooked in protein sequencing studies if an organism's genome is not analyzed.

Note that our goal is to determine novel protein sequences. A number of techniques exist to characterize well-known protein sequences [8][9][10], but our challenge is to accelerate real-time de-novo protein sequencing. Therefore the ability to search the genome at high speed is crucial.

2.4.2. Hardware Searches of the Genome

The continuous growth of biological databases has created the demand for intensive computational power if these databases are to be analyzed within a practical timeframe. Several biological algorithms have already benefited from custom hardware acceleration, some of which are reviewed in this section.

Among the most well known algorithms that show improvement when implemented in hardware are those used for sequence alignment. These methods search through biological databases to look for strings similar to those provided by a user. Hoang and Lopresti describe hardware implementations of alignment algorithms that perform several orders of magnitude faster than their software counterparts [17][18]. The alignment algorithms in their work compute the *edit distance* between strings. The edit distance between two strings is the weighted cost of the operations required to convert one string to the other. The distance is computed using the common Smith-Waterman dynamic programming algorithm, which lends itself to hardware due to its parallelizable nature.

Commercial hardware units such as BioXL, which perform sequence alignment, are also available to researchers [20]. BioXL is capable of performing the Smith Waterman calculations in addition to several proprietary algorithms that perform similarity searches. The BioXL package is designed as a scalable system, which can grow based on the user's budget and requirements. Depending on cost concerns, the user can have a hardware

system that outperforms an identical software algorithm by a factor of 198. The core of the BioXL unit is a set of FPGAs containing hardware implementations of various search algorithms. Other algorithms, such as BLAST [23], which search both gene and protein databases, have been commercially implemented in systems such as DeCypher [19], which also use FPGA-based hardware searches. These searches are commonly used in similarity studies to establish the relationship between groups of proteins or groups of genes. The DeCypher hardware was created in response to the massive growth of genomic databases. The DeCypher system provides an economical alternative to purchasing large server farms to search large genomic databases. A number of biological search algorithms in addition to BLAST have been implemented in DeCypher, most of which seek to group similar genes and proteins into families. These hardware implementations show between 50 to 200-fold increase in speed with a 10 to 100-fold reduction in price-performance ratios when compared to equivalent software platforms.

2.5. Programmable Hardware Platform

Our goal in this work is to implement the genomic search engine, tryptic mass calculator and scoring algorithm in hardware to accelerate the de-novo protein sequencing process.

The hardware upon which the system is prototyped is the University of Toronto's Transmogrifier 3A (TM3A) reconfigurable platform [13]. The core of the system is a set of four interconnected reprogrammable chips known as Field-Programmable Gate Arrays (FPGAs). These allow the user to implement a new design by simply downloading it to the board from a PC. A brief description of FPGAs in general and the architecture of the TM3A are presented in the following sections. This is followed by a description of how a design is specified using a Hardware Description Language (HDL).

2.5.1. Field-Programmable Gate Arrays

FPGAs are reprogrammable chips that can have their logic functionality modified by a user. There are two key features of an FPGA that enable this programmable behaviour: programmable logic blocks and programmable routing. In Figure 2-19 the simplified

view of an FPGA is depicted. It can be seen that there are a number of columns of connected Configurable Logic Blocks (CLBs). The Configurable Logic Blocks often contain multiple Lookup Tables (LUTs) and flip flops. These LUTs implement any Boolean expression with a fixed number of inputs. In Figure 2-20, a 4-LUT (four input lookup table), which can implement any Boolean function of 4 inputs, is shown. The outputs of these functions can then be passed to various other LUTs or the input/output blocks (IOBs) of the FPGA. In the architecture depicted, there is also a flip-flop associated with each LUT, which is used to store the LUT output. Another feature of modern FPGAs is the embedded block RAM (BRAM) that is also connected to the routing racks [22]. This additional RAM provides greater storage capacity within the FPGA. The FPGAs in the TM3A are Xilinx Virtex 2000E FPGAs that have 38,000 LUTs and flip-flops and 64Kbits of RAM per chip.

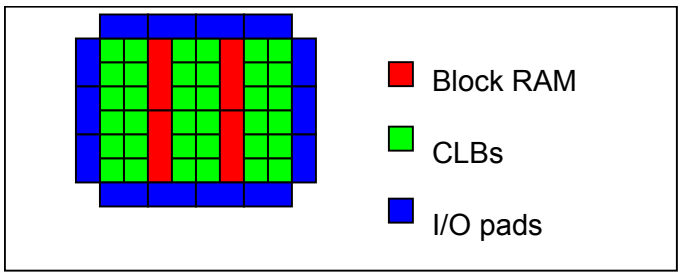


Figure 2-19: FPGA Architecture

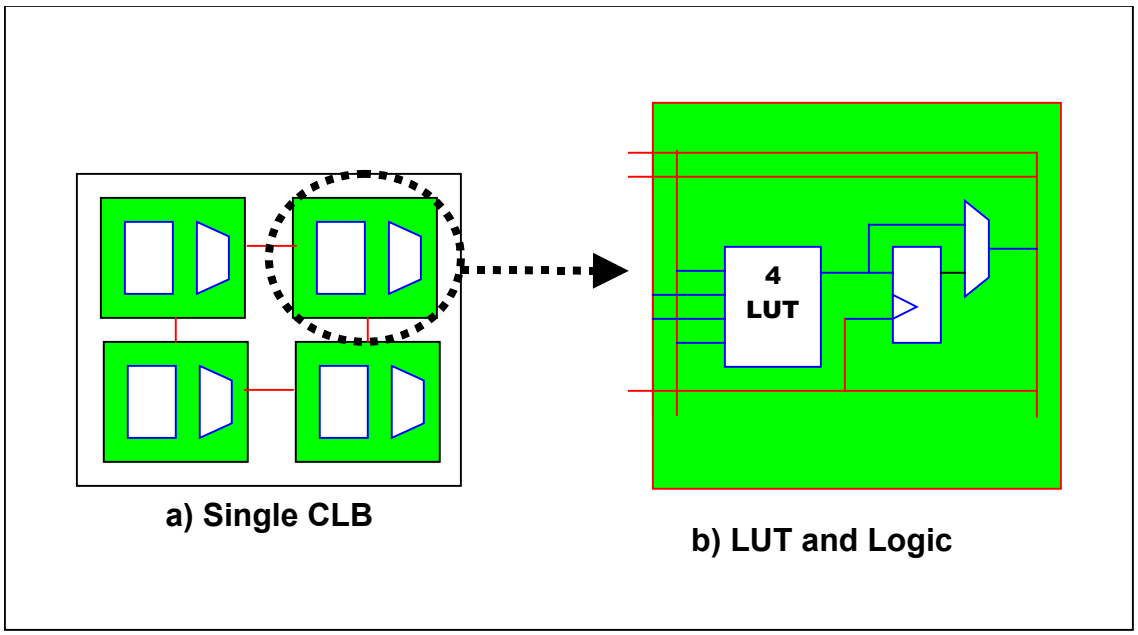


Figure 2-20: CLB and LUT details

2.5.2. *Hardware Description Languages (HDLs)*

To implement a circuit in an FPGA, the designer needs to describe it with a Hardware Description Language (HDL). The designs in this work were created using VHDL, (VHSIC• Hardware Description Language). VHDL is commonly used to describe a circuit at various levels. At a high level of abstraction it can describe how circuit components are connected together. Conversely it can be used at a detailed level to specify the behaviour of each of the individual circuit components. An illustrative example is provided below.

• Very High Speed Integrated Circuit

```
ENTITY and2 IS
        PORT
        (
        input1  : IN STD_LOGIC ;
        input2  : IN STD_LOGIC ;
        and2_out : OUT STD_LOGIC
        );
END and2;

ARCHITECTURE and2_behv OF and2 IS
BEGIN

        and2_out <= input1 AND input2 ;

END and2_behv;
```

Figure 2-21: VHDL definition of 2 input AND gate

The example in Figure 2-21 shows the VHDL specification for a 2 input AND gate. The boldface type highlights keywords reserved by the language. The AND gate is described as an **ENTITY** that has two input ports and a single output port. The behaviour of the entity is described in the architecture section, where the logical **AND** of the two inputs is assigned to the output of the circuit.

This simple example illustrates how a circuit component can be described in VHDL. A compiler then synthesizes this code into the hardware structures such as the LUTs described in Section 2.5.1.

2.5.3. Transmogrifier 3-A (TM3A)

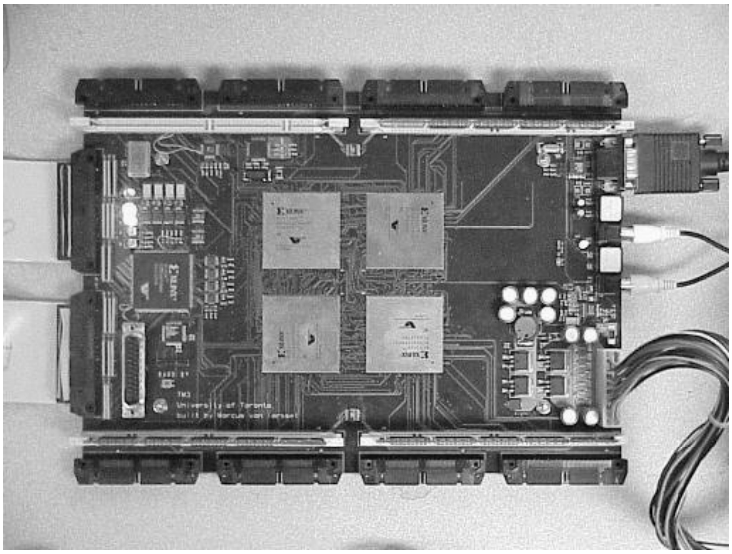


Figure 2-22: Transmogrifier 3-A

The TM3A (shown in Figure 2-22) is a reconfigurable hardware platform with 4 Xilinx Virtex 2000E FPGA chips that are interconnected to each other by a 98-bit bus [13]. This allows designs that are too large for a single FPGA to be spread over multiple chips. Each FPGA also has 2 megabytes of SRAM attached and various IO connectors. Data is read from the SRAM in 63-bit words. Each chip is also connected to a central housekeeping chip, which performs the configuration of the FPGAs and ensures that they are functioning within their operational limits. The housekeeping chip also interfaces the board with a PC.

The PC allows the user to download designs into the onboard FPGAs and to communicate with the board to provide input and receive output. A convenient software interface to connect circuit on the FPGAs to a C program running on the host PC has been developed, called the ports package **[14]**.

2.6. Summary

In this chapter, we have described the requisite biology to understand the design presented in our work. The challenges of conventional de novo protein sequencing by mass spectrometry have been examined. The advantages and shortcomings of using the human genome database to infer the sequence of novel proteins have been presented. The limitations of implementing these sequencing approaches in software and the appeal of custom hardware for similar algorithms have also been considered. A description of the implementation platform has also been provided as the architecture of this platform guides our design choices.

In the following chapter we describe the design of the hardware units that the device is comprised of. For each of the requirements listed in Section 2.2.3, we design hardware units that are optimized to perform specific calculations that are optimized to both accelerate the algorithm, and target the architectural features of the hardware.

Chapter 3. Design of a Hardware Search Engine, Mass Calculator and Scoring Unit

Overview

Any useful implementation of the sequencing approach described in the previous chapter demands the capacity for high-speed searches. This speed can only be achieved in software at high cost, as mentioned in Section 2.4.1. Custom hardware, as seen in Section 2.4.2, is often a practical solution for applications that process large volumes of data and can be easily parallelized. The core of the algorithm in Section 2.2.2 is a search through the genome that must be completed in approximately 500 ms to 1 s. Since a database search is intrinsically parallelizable and the search space is large, we implement the key units described in section 2.2.3 in hardware to achieve the speed requirements and avoid the costs of a large computing cluster.

The design takes three primary inputs, namely:

1. A peptide query from the MS, which is a string of 10 amino acids or less,
2. A genome database,
3. A list of peptide masses detected by the MS. (the true PIS described in Section 2.2.2)

The design produces a set of outputs for a given peptide query:

1. A set of gene locations, which can code the input peptide query
2. A set of scores for each gene location. The scores rank the genes based on the likelihood that they coded the protein in the sample.

The hardware identifies all locations in the genome that can code the peptide query and then translates these gene locations into their protein equivalents. It then compares the peptides in the translated proteins to the peptides detected by the MS and provides a ranking for each gene location based on how well it matches the masses detected by the MS. These gene locations can be translated to their protein sequence in a matter of a few milliseconds by using Table 2-1 or by using existing software packages [44][45].

The design is divided into three major subunits:

1. A search engine that locates all possible coding strands for a peptide query.
2. A tryptic mass calculator that translates all matching genes and produces the masses of all the corresponding tryptic peptides from the translated gene.
3. A scoring unit that compares calculated peptides against those stored in the PIS of the MS and ranks the matching gene locations.

This architecture is depicted in Figure 3-1. In the following sections we describe the inputs and explain how they are encoded within the system. We then describe each of the units in Figure 3-1 as we detail the flow of data through the system.

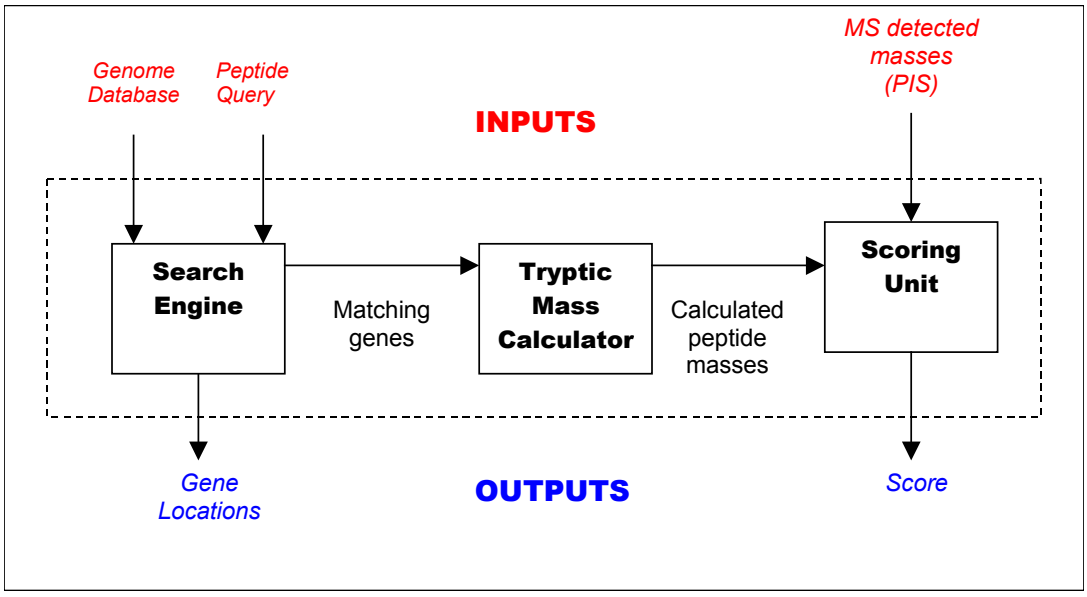


Figure 3-1: Device Architecture

3.1. Genome Database Coding and Compression

The genome database is one of the primary inputs to the system. To better understand the nature of operations performed on this database, a description the data encoding schemes used to store this database is provided.

The genome database is stored as an ASCII file of bases, and is available for download from several different institutions. The ASCII representation uses 8 bits per character, which allows for 256 unique characters to be stored. However, since there are only 5 different characters (the four bases A, T, C, G and the wildcard N) in the genome database 98% of the storage spaces is wasted. We thus encode this ASCII file using a different scheme that allows for better compression of the data. Each codon in the genome file is encoded using a 7-bit value that allows for 2^7=128 unique codons. Each codon consists of 3 characters and the characters themselves can be one of five values. Therefore there are 5^3=125 unique codons in the actual genome database. For example AAA = 0000000, AAT = 0000001, AAC = 0000010 etc. This encoding uses 2.3 bits per base wasting only 2.3% of the storage space (125 of 128 possibilities used).

Since the genomes of most organisms are large (15 million to 3.3. billion characters), it is not practical to store the genome database directly on-chip. Instead we store the genome database in RAM external to the FPGAs.

As the genome is read from external RAM into the device, it first passes through the decoder units illustrated in Figure 3-2. Each decoder takes in a 7 bit “compressed” codon from memory and produces a 9 bit “uncompressed” codon using the original 3-bit encoding scheme. The decoders themselves are BlockRAM units that are configured as ROMs. They accept the compressed string as an address and produce and produce an uncompressed bit-string as their output.

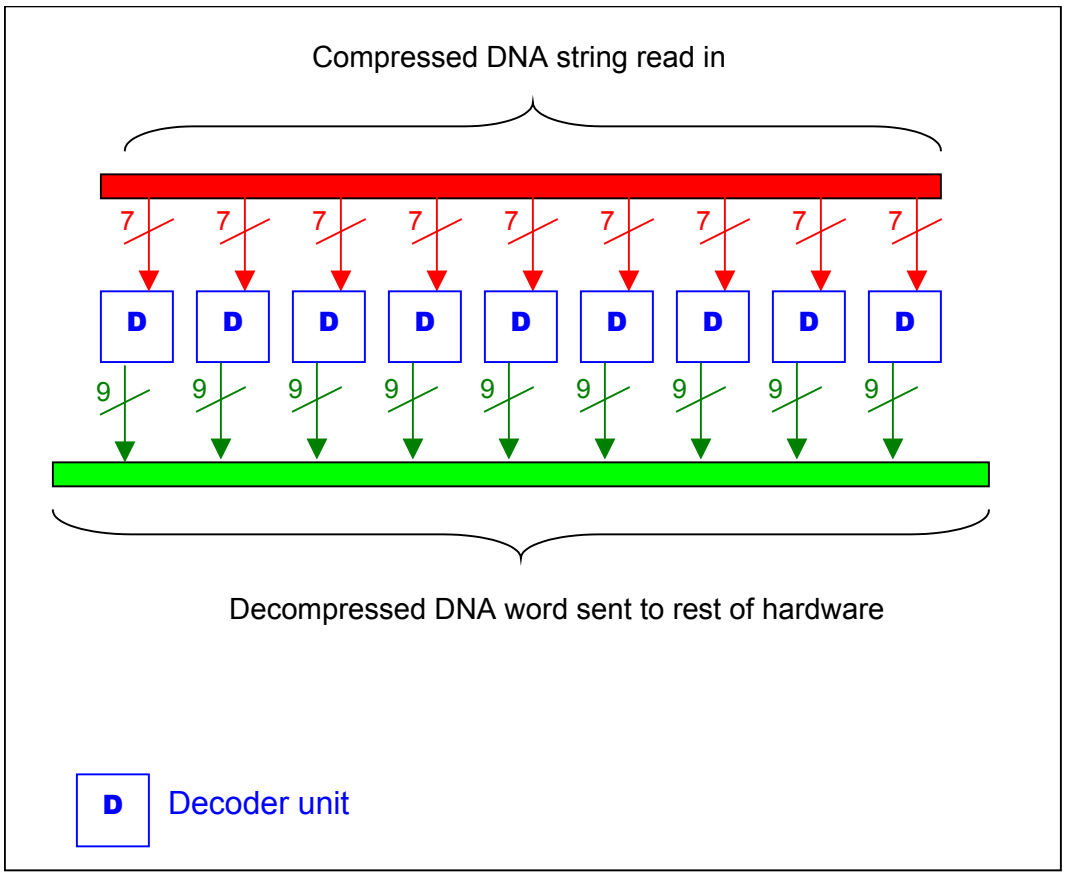


Figure 3-2: Genome Decompression

The uncompressed bit-string uses 3 bits per base that allows for eight possible characters, five of which are used (A = 000, T = 001, C = 010,G = 011 and N =100 for ambiguities). Thus a single codon is represented by a 9-bit value within our hardware as shown in Figure 3-2. The rest of the hardware units described in the following sections also use the 3-bit encoding scheme described above.

3.2. Peptide Query

Recall that the output of the second MS in an MS/MS experiment is a peptide sequence (i.e. a string of amino acids). This must be converted to an equivalent DNA representation to be compared against a genome database. This process was outlined in Section 2.2.2. Consider for example the case when the MS outputs the peptide sequence "MAVR". The goal of the algorithm is to locate all genes that can create this peptide.

Therefore we reverse translate each amino acid into the codons that it could have originated from.

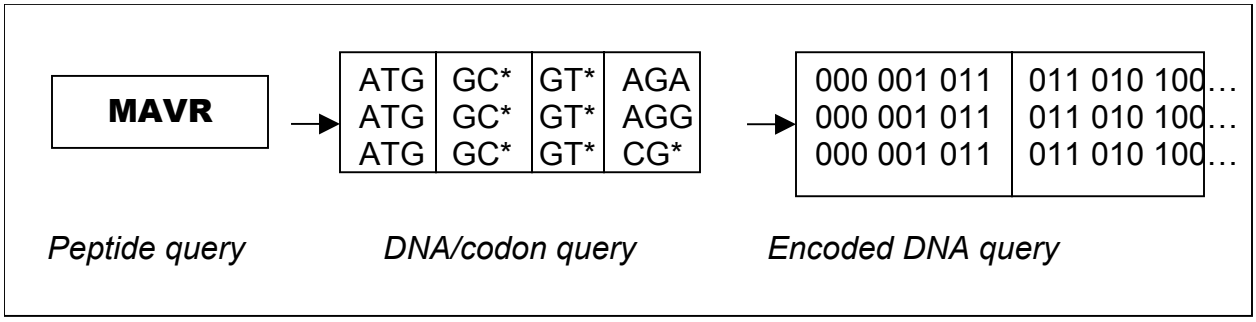


Figure 3-3: Query Reverse Translation

The peptide query is a string of no more than 10 amino acids (including wildcards). We chose this query size based on the average size of the sequencable portion of a tryptic peptide (approx. 10 amino acids) and the fact that a very short sequence of amino acids (often less than 7) can uniquely identify the protein it originated from [25].

Note that we allow the wildcarding of searches by the inclusion of a wildcard character in the query. This also serves to compress the query, as some amino acids with multiple codons will not need each codon explicitly enumerated (for example the amino acid Alanine (**A)** in the query above is expressed as GC*). This reverse translation is done on the host PC when the peptide query is received from the MS. Inspection of Table 2-1 shows that no more than three codons are needed to encode any amino acid when wildcards are employed. Thus we reverse translate each amino acid in the peptide to generate a codon, or DNA query that encapsulates all the possible coding strands for the peptide query as shown in Figure 3-3. Each of these DNA/codon queries are then encoded using the 3-bit scheme described above.

It was mentioned in Section 2.3.1 that genetic sequences are stored as either original DNA strands or their complements, but never both, since this is redundant. In the 3-bit encoding scheme, no information is stored to indicate the type of strand. Therefore we must also consider the complement of every strand in the database to ensure that all possible coding patterns within an organism's genome are examined. For this purpose,

the complement of the query is also generated. Thus the original peptide query is translated into six binary strings, three for the original DNA strand representation and three for its complement. The query, thus encoded, is submitted to the search engine, which locates all instances of the coding stands in the genome.

3.3. Search Engine

Recall that the primary objective of the search algorithm is to identify all possible locations in the genome from which a peptide may have originated. To accomplish this, the user provides a peptide query (inferred from the MS data), which is simply a string of amino acids. To compare these amino acids to a genome (DNA) database they must be reverse translated to codons as described in Section 3.2. The search engine takes these strings of codons as input, and outputs all positions within the genome that match the strings.

The purpose of implementing the search in hardware is to maximize speed. This speed is governed by the frequency with which the memory containing the genome can be clocked through the search engine. We define the parameter MEM_WIDTH to be the width of a memory word that is read into the search engine, i.e. the number of bits read into the system in every clock cycle. Thus the total number of clock cycles required to search through a genome in memory (with a size defined by SIZE_OF_GENOME) is given by $\frac{SIZE\_OF\_GENOME}{MEM\_WIDTH}$.

Consequently the total time to search through the database is given by:

$$Total\_Search\_Time = \frac{SIZE\_OF\_GENOME}{MEM\_WIDTH} \times \frac{1}{System\_Frequency}$$

Note that the total search time must be less than 1s for the search engine to be useful in the de-novo sequencing method described in Chapter 2. Furthermore, we speculate that there may be other applications that require high-speed searches of the genome. In Chapter 4 we will describe versions of this system capable of achieving search speeds in the order of a few hundred milliseconds.

3.3.1. Search Engine Operation

The search engine accepts queries, which consist of a set of DNA strings and their complements, and locates every position within the genome that matches any of these strings. The genome, which is stored in the RAM, is clocked in as a series of MEM_WIDTH-bit memory words. On every clock cycle the controller reads a new memory word into the system. This word is compared to the set of queries provided by the user. If a match is detected, the search engine controller returns the current memory address, which the user can then use to locate the coding gene. The VHDL description of the search engine controller is provided in Appendix B1 (control.vhd). A depiction of the architecture of this device is provided in Figure 3-4.

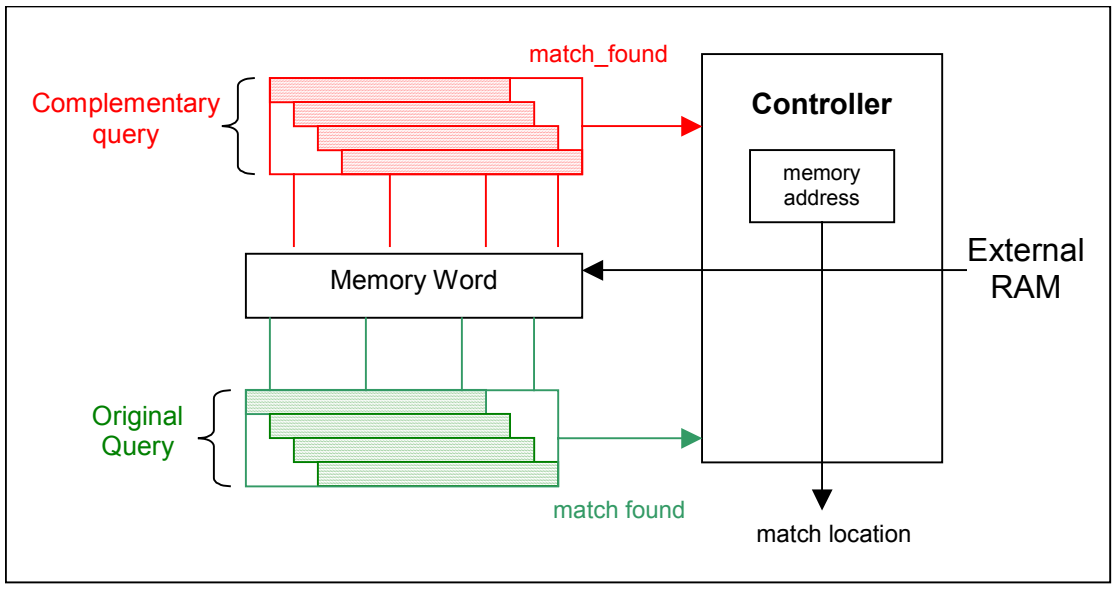


Figure 3-4: Full Search Engine Architecture

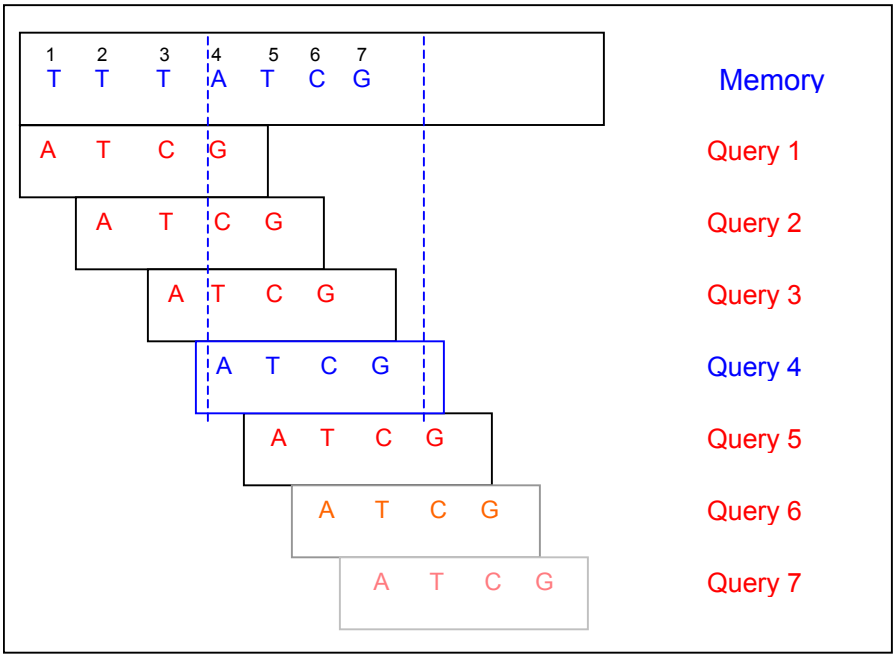


Figure 3-5: Searching the Genome

Once reset, the search engine controller enters *initialization* state in which the six DNA queries are read into the search engine. This is done in two clock cycles: one for the original DNA query, and one more for the complementary query described in Section 3.2. In the example in Figure 3-5, a simplified view of the architecture is presented, in which a single DNA query is performed. Note that the complementary query shown in Figure 3-4 is removed for simplicity, however the search operations performed on both strings are identical. The controller then moves into the *comparison* state in which memory words are continuously read into the search engine from external RAM. With a new word entering the engine in each cycle, every substring within the memory word must be compared to the query in a single cycle. To do this, multiple copies of the query are registered in hardware, and each one is simultaneously compared against the memory word. Note that we need as many copies of the query as there are bases in the memory word. This is apparent in the architecture shown in Figure 3-5 as each copy of the query is aligned with a successive base in the memory word.

Using the compression scheme of 7 bits per codon described in Section 3.1, the number of bases in a single memory word is parameterized as:

$$NUM\_BASES\_IN\_MEMWORD = MEM\_WIDTH \times \frac{7}{3}$$

Each copy of the query is stored in a *peptide unit*, and if any *peptide units* signal a match (as query 4 in the example in Figure 3-5), the controller exits the comparison state and returns the current memory address to the user, to be interpreted as a coding region for the query strand. The search engine then returns to the comparison state and the process continues until all the memory has been read.

It is apparent that the *peptide units* mentioned above are responsible for the core functionality of the search engine. To elucidate the details of the design, a description of the peptide unit follows.

3.3.2. Peptide Comparison Unit

The search process described above compares several identical copies of the query to a memory word to maximize throughput. Each query is stored in an individual peptide unit, which is depicted below.

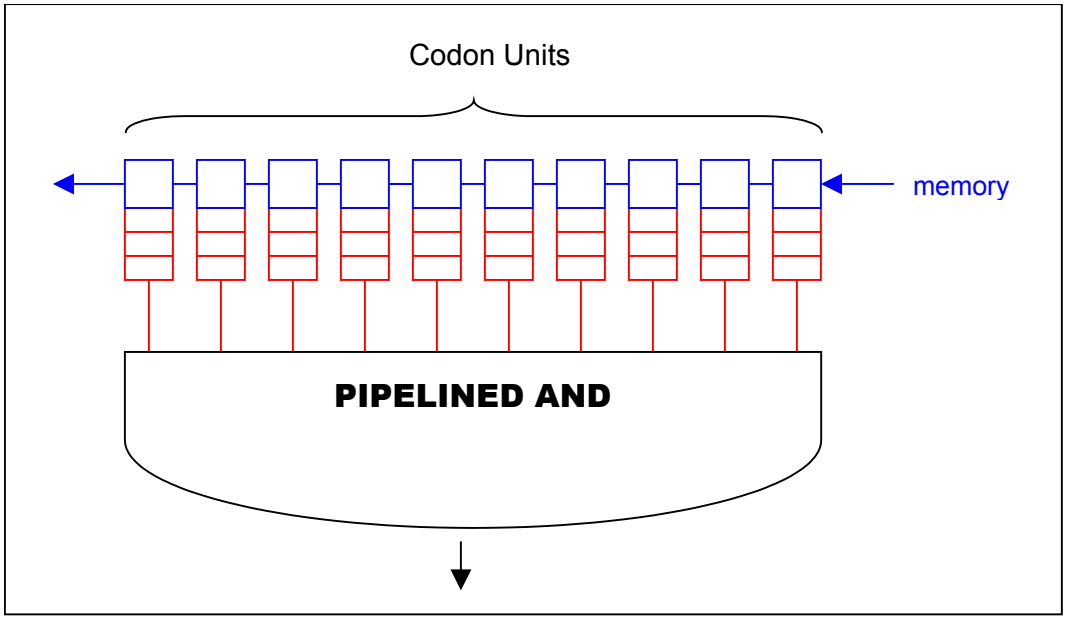


Figure 3-6: Peptide Unit Structure

A peptide comparison unit takes two inputs

a. A set of query codons (corresponding to the amino acids in the query)
b. A set of 10 codons from memory.

Figure 3-6 represents the general architecture of a peptide comparison unit. The query codons are stored in a set of *codon units*. Each of these units then receives codons from the memory word, which are compared against the query codons. Each unit produces a single match output that signals whether the codon from memory matches any of the query codons. If all of these match signals are activated simultaneously, a string of codons from memory that matches a set of query codons has been found. The VHDL description that instantiates the peptide comparison unit is presented in Appendix B 2 (protein.vhd)

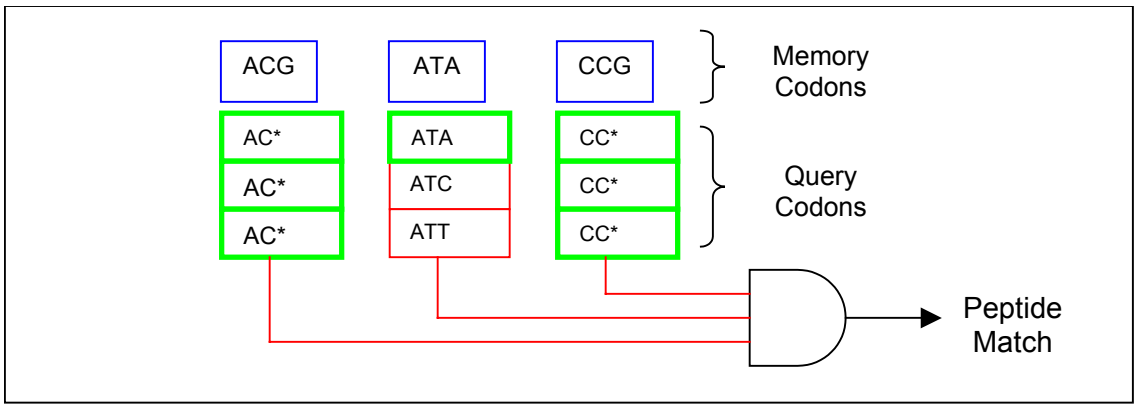


Figure 3-7: Peptide Unit Operation

In Figure 3-7 a simplified peptide comparison unit is depicted in operation. There are 3 sets of query codons, which are compared to the codons from memory. In Figure 3-7 the matching codons are highlighted. If at least one codon from each set shows a match to memory, the query has been found in the genome, or equivalently, a coding strand for the peptide query has been found.

Thus each of the codon sets signals a pipelined logical AND unit, and if all sets indicate a match, the peptide unit signals a match. A wide AND operation (logical AND with many

inputs) will incur significant delay if is to be completed in a single cycle. To avoid this delay and ensure fast circuit operation, we register the match signals from the units, then AND them as a pipelined operation.

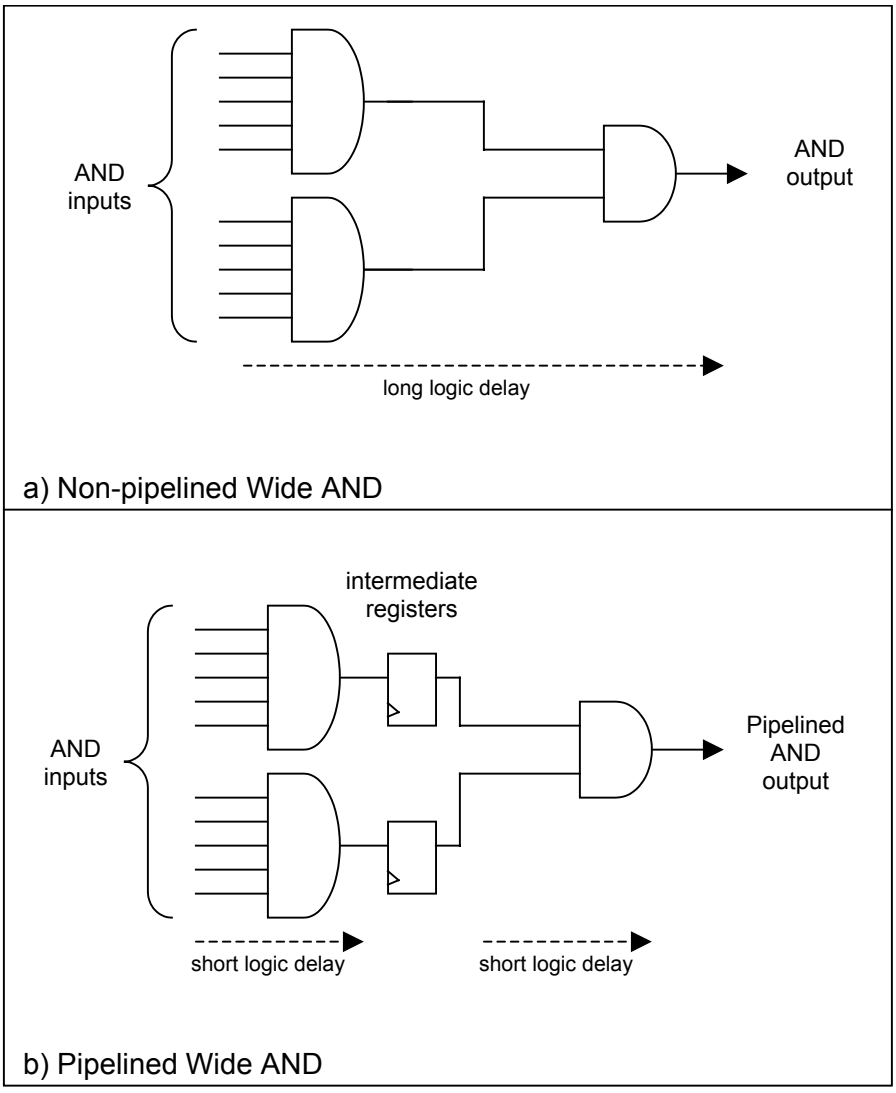


Figure 3-8: Pipelined AND Operation

Figure 3-8 contrasts a simple wide AND implementation with the pipelined version described above. In the non-pipelined unit, there is a comparatively long logic delay as the input pass through multiple gates to produce the output AND signal. If this delay is sufficiently high, it will constrain the maximum clock frequency of the circuit. In the pipelined implementation, the inputs are divided into two groups. Each of these groups is

individually ANDed in a single clock cycle. The results of this operation are stored in intermediate registers and ANDed together in the next clock cycle. This technique reduces the delay through logic and allows faster circuit operation. Note that the output of the pipelined AND is delayed by an additional clock cycle, but this is usually acceptable as the clock frequencies are sufficiently high, and the penalty of an extra cycle is negligible.

Figure 3-6 depicts the peptide unit as a set of *codon units*, as described above. It is the match signals from each of these codon units that are ANDed together to verify that all codons have detected a match in memory. These *codon units* are the building blocks upon which the search engine is built.

3.3.3. Codon Unit

The smallest fundamental unit of the search is the *codon unit*, which takes a set of three query codons and a single codon from memory as its input. It produces a match signal as its output. If any of the three query codons matches the memory codon, the match signal is activated. The set of three codons corresponds to the translation defined in Section 3.1. Recall than any amino acid can be represented as set of three codons or less. Thus a codon unit essentially determines whether a codon from memory is capable of coding a query amino acid.

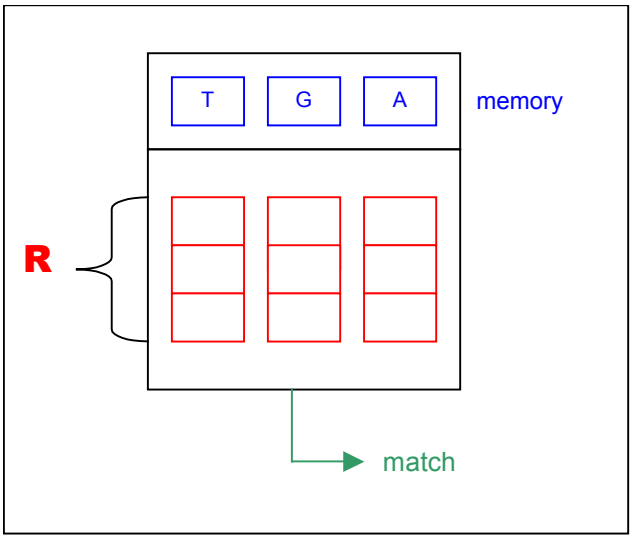


Figure 3-9: Codon Unit Operation

The operation of the codon unit is shown Figure 3-9. Assuming that the query amino acid is Arginine (R), it is translated to its equivalent codons AGA, AGG and CG* using Table 2-1. This is done in software before the query is submitted to the search engine hardware as described in Section 3.2. These three query codons are stored in the codon unit, and at every clock cycle, a new base from the genome in memory is read in and compared against the queries.

Delving deeper into the implementation, Figure 3-10 illustrates a detailed view of the codon unit. The bases in the three query codons are divided by position, i.e. the first base in every query codon is ANDed with the first base for a codon from memory, the second query base is ANDed with the second memory base and so on. From Figure 3-10, it is apparent that the codon unit only signals a match if each base from memory matches at least one query base in its corresponding position. The VHDL code that describes this architecture can be found in Appendix B 3 (amino.vhd)

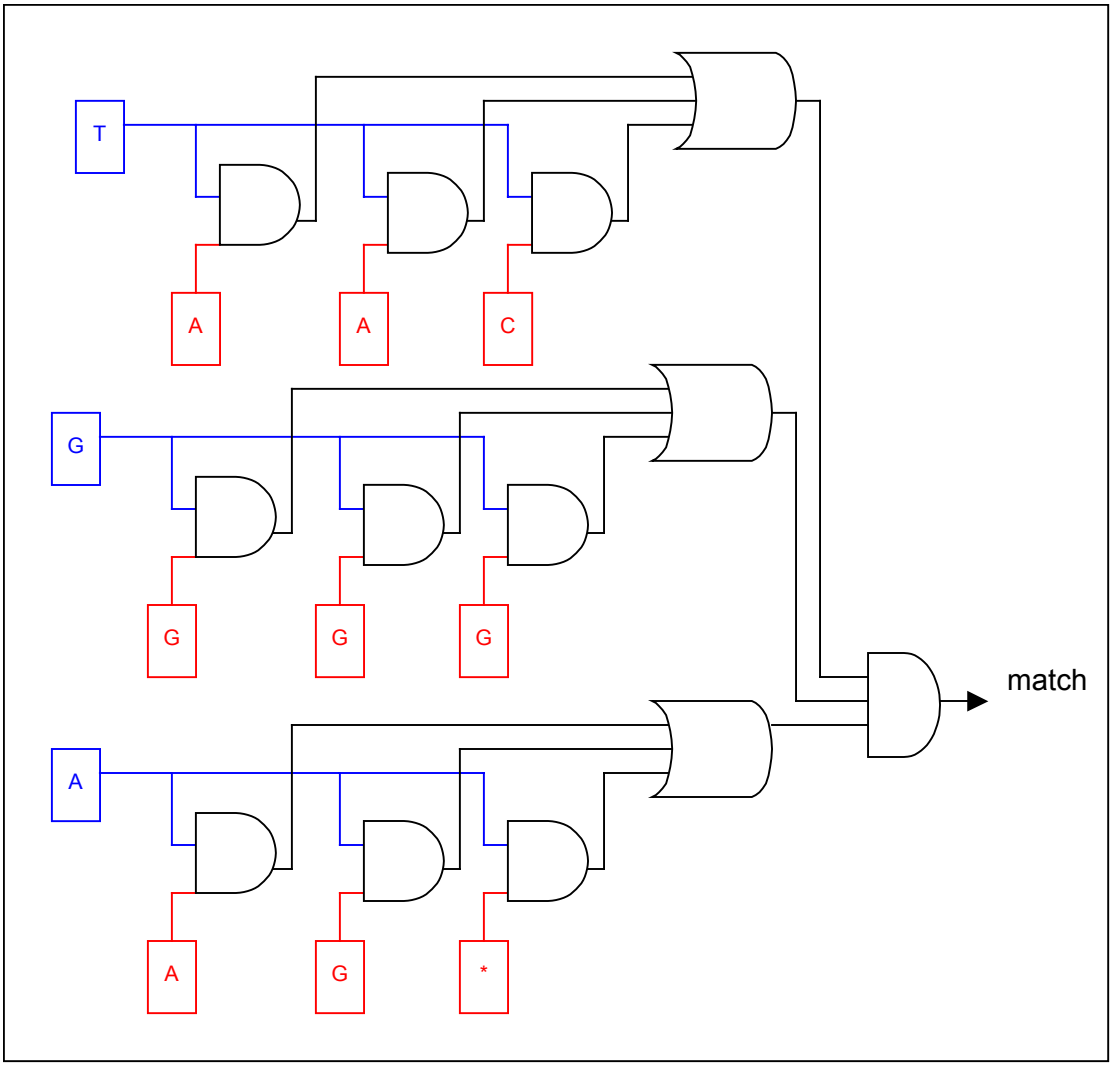


Figure 3-10: Implementation Details of Codon Unit

It is the match signal shown in Figure 3-10 that is passed into the pipelined AND in the peptide comparison unit, and ultimately to the controller, which then detects a hit and returns the corresponding memory address to the user.

3.3.4. *Interpreting Search Engine Outputs*

The search engine identifies memory addresses that contain a section of DNA capable of synthesizing the query peptide. In a biological sense, this corresponds to identifying coding genes within the genome. Figure 3-1 indicates that the *gene* at the hit location is then sent to the tryptic mass calculator for further processing.

However the stream of DNA from the genome database, which passes through the search engine, has no markers to indicate the start or end points of a gene. To overcome this lack of information, we use the average size of a gene to delineate the gene under consideration.

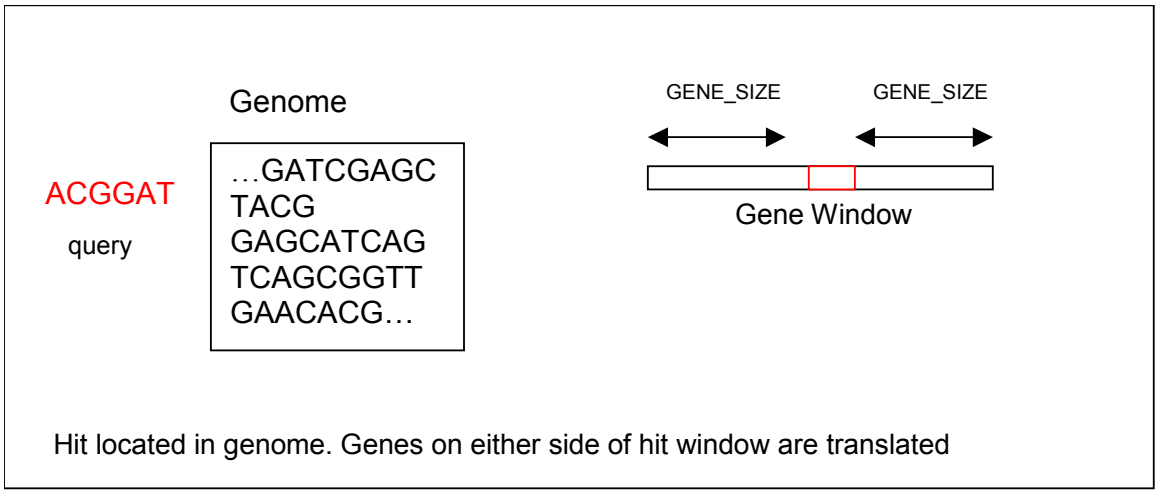


Figure 3-11: Selection of Gene

Defining the size of a gene as GENE_SIZE bases, we send a 2 x GENE_SIZE window of bases surrounding the hit to the calculator. This approach, as shown in Figure 3-11, allows the consideration of one gene preceding the hit and one gene following it. In practise, this window is implemented as a GENE_SIZE sized shift register. The input data to this shift register is obtained from the output of the decoder blocks described in Section 3.1. This data is in the uncompressed 3-bit form; therefore the depth of the shift register is GENE_SIZE x 3 bits. Data from the decoder is continually passed into the gene window register, which acts like a delay element, as its outputs are delayed by GENE_SIZE (its depth) relative to its input. When the search engine detects a hit, the

output of the gene window is sent to the tryptic mass calculator, which continues to read the gene window until it has processed 2 x GENE_SIZE bases.

This technique ensures that the calculator processes a reasonable amount of genomic data on either side of the hit location. However, the fixed size of the gene window adds an inherent error to further operations, as most genes will be of a different size. Regardless, if a reasonable portion of the gene is processed, it will still be possible to identify many of the peptides from the translated protein.

3.3.5. Summary of Search Engine Design and Operation

The original peptide query is translated from amino acids to sets of codons as described in Section 3.2. These codon strings are stored in the codon units that make up a peptide unit. Multiple identical copies of the peptide unit are instantiated to maximize the throughput of the search as described in Section 3.3.1. The search engine progresses incrementally through the address space of the genome stored in RAM, looking for a match to the queries. If a match is found, the current memory address is sent to the user as a gene location that codes the peptide query. Genomic data surrounding the hit location is then sent to the Tryptic Mass Calculator as illustrated in Figure 3-1.

3.4. Tryptic Mass Calculation

Overview

Referring back to Figure 3-1, we see that the search engine locates genes matching the peptide query and sends the corresponding addresses to the user. It remains to *translate* all matching genes to their protein equivalent, *digest* these proteins to peptides and *calculate* the masses of the peptides. Peptide masses from each translated protein are then compared with the PIS list described in Section 2.2.1 to determine which translated protein most closely matches the protein sample in the MS.

Note that the *tryptic mass calculator* receives matching genes as its input, and performs the translation, digestion and calculation operations described above to provide the peptide masses as outputs. To do this the calculator unit must translate the matching genes from the search engine into amino acids and locate the tryptic cut-sites as described in Section 2.2.1. To obtain tryptic peptide masses, the sum of masses of the amino acids from cut-site to cut-site is accumulated. These masses are then sent to the Scoring Units as illustrated in Figure 3-1.

As an overview of the mass calculation process, an example of the steps involved is presented.

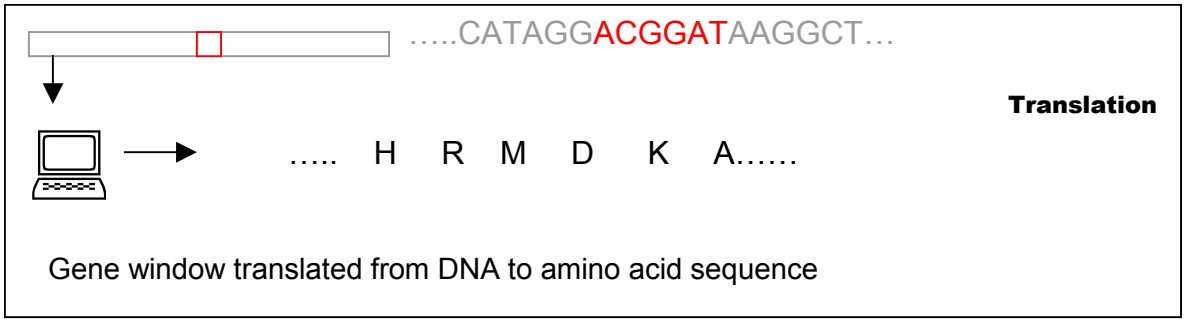


Figure 3-12: Translation of Gene to Protein

The DNA data from the gene window, i.e. the matching genes, are interpreted as a stream of codons, or equivalently, as an amino acid string. We are, in effect, *translating* the gene to its corresponding protein as shown Figure 3-12.

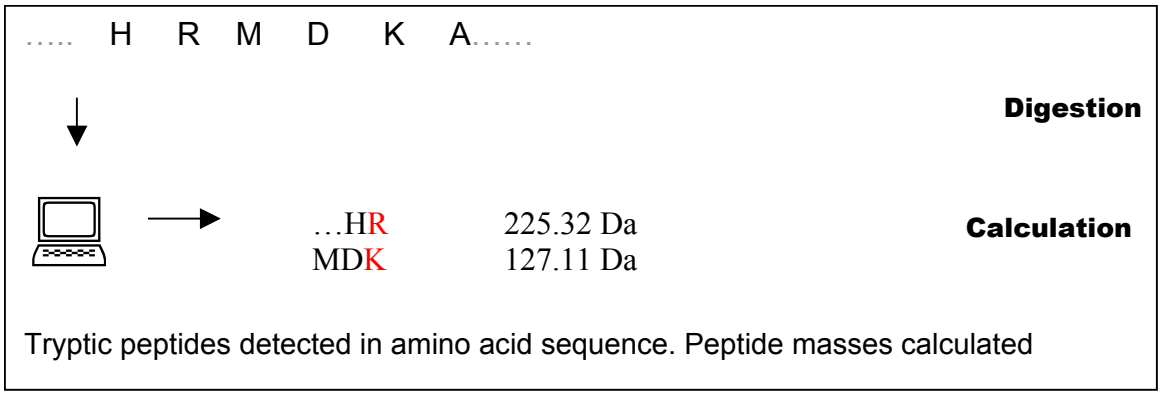


Figure 3-13: Digestion of Protein and Calculation of Tryptic Peptide Masses

Once a protein is translated, its tryptic peptides must be compared to those detected by the MS. To identify the tryptic peptides and *digest* the protein, the calculator detects the tryptic cut-sites (Lysine (K) and Argnine (R) amino acids) and *calculates* the accumulated mass of all amino acids between these cut-sites as illustrated in Figure 3-13.

3.4.1. Calculator Architecture

An architectural view of the calculator as depicted in Figure 3-14 shows a pipelined design that performs the translation, digestion and peptide mass calculations described above.

At every clock cycle, the controller for the calculator reads a new set of NUM_BASES_IN_MEMWORD bases from the gene window into the calculator. The calculator operates on this data in codon-sized units. Note that each stage of the calculator in Figure 3-14 has a single active codon attached to a detection unit and mass lookup table. The first stage of the calculator *translates* its first codon into the mass of its corresponding amino acid, which in turn is passed to a mass accumulator. In the next clock cycle the controller reads a new set of codons from the gene window into the calculator, and the remaining unprocessed codons from first stage are passed down. In the

second calculator stage, the second codon is processed in parallel with the first codon from the new set. Note also that the accumulator from the first stage passes its calculated mass to the second stage. Thus the mass of the first amino acid can be added to the mass of the second to *calculate* the mass of the peptide. If the detection units identify a tryptic cut-site (Argnine or Lysine amino acids not followed by Proline), *digestion* occurs and the accumulated peptide is output from the calculator. Observe that each stage of the calculator operates in an identical manner by receiving a set of codons, performing calculations on only a single codon and buffering the rest. These remaining codons are passed to the next stage in the subsequent clock cycle and the process is repeated until the entire gene has been processed. The VHDL representation of the behaviour of the calculator is given in Appendix B 4 (mod_calc.vhd).

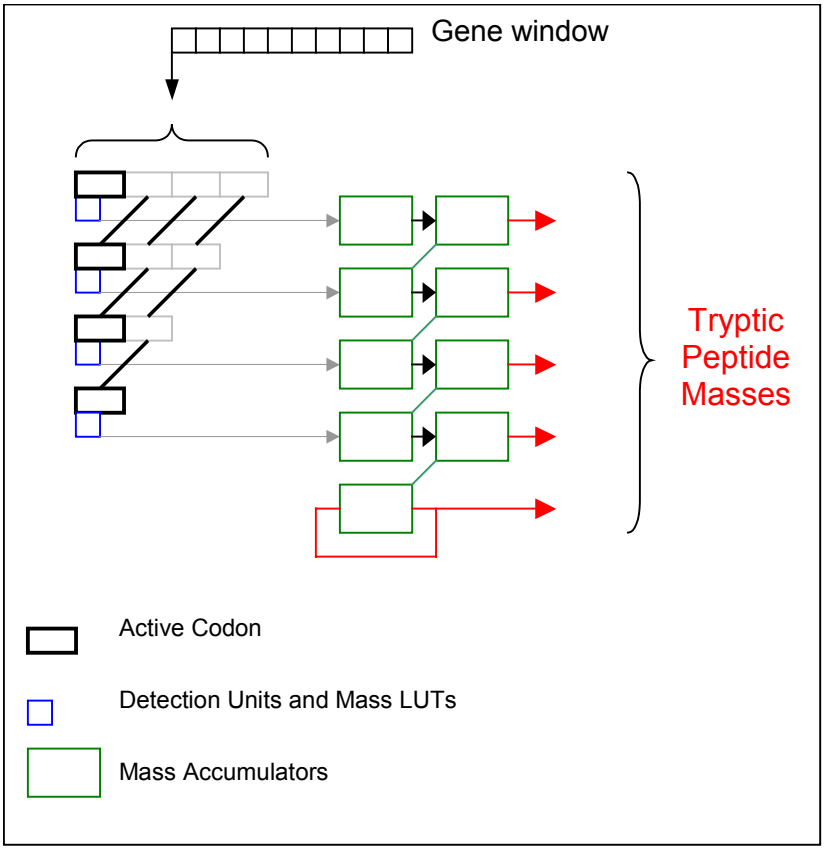


Figure 3-14: Calculator architecture

The matching gene is passed as input to the calculator, NUM_BASES_IN_MEMWORD at a time to match the memory throughput. The calculator operates on these bases in codon-sized units; therefore NUM_BASES_IN_MEMWORD/3 codons (defined as NUM_CODONS) are clocked into the calculator in every cycle. To maintain this throughput, the calculator needs at least NUM_CODONS stages operating in parallel, as there could be at most NUM_CODONS peptides in a single memory word. However, if a peptide spans more than a single memory word, the accumulated mass from the first memory word will have to be saved until the tryptic cut-site is detected in one of the following memory words. Thus an extra pipeline stage is required to accumulate intra-word peptides, resulting in a total of NUM_CODONS +1 stages operating in parallel to ensure that the calculator can meet the memory throughput.

For every hit detected by the search engine, the calculator processes a full gene window of bases. Thus for every hit, the calculator operates for a total of $\frac{GENE\_SIZE}{NUM\_BASES\_IN\_MEMWORD}$ corresponding to one cycle for every memory word in the genome. Note that an additional NUM_CODONS+1 cycles are required to process the codons that will remain the pipeline of the calculator. The following sections provide a detailed description of the architecture of the hardware used to perform the mass calculations.

3.4.2. Mass Calculation

For a detailed account of the operations performed by the calculator, consider Figure 3-15.

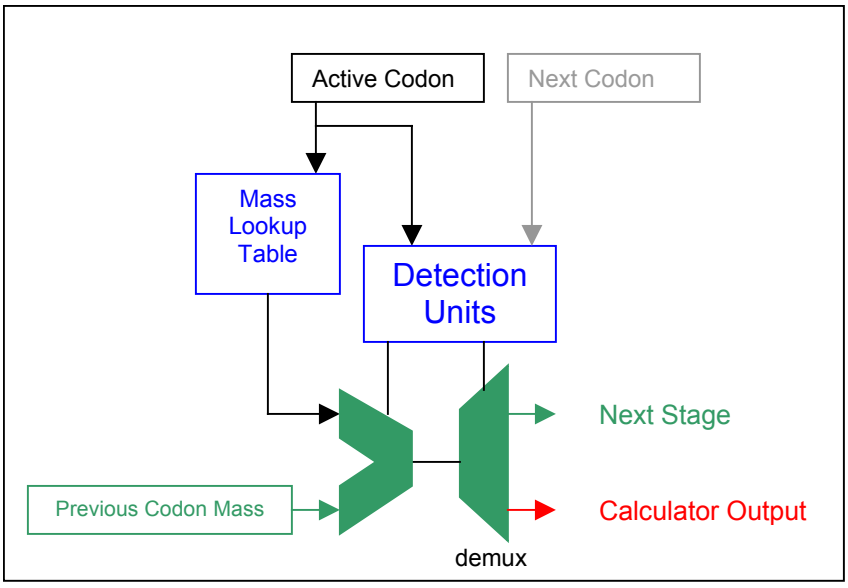


Figure 3-15: Single Stage of Calculator

Each stage of the calculator only processes its active codon, which is fed into a lookup table of masses and a set of detection units. The mass lookup table reads the codon and produces the mass of the corresponding amino acid effectively *translating* the codon. The detection unit looks for tryptic cut-sites in the codon stream. If no cut-site is detected, the mass of the previous codon is added to the mass of the active codon. However, if a cut-site is detected, i.e. we reach the end of a tryptic peptide, the accumulated mass is sent to the calculator output instead. Thus the detection units and mass accumulators control the *digestion* and *calculation* operations of the calculator.

3.4.3. Mass LUTs and Detection Units

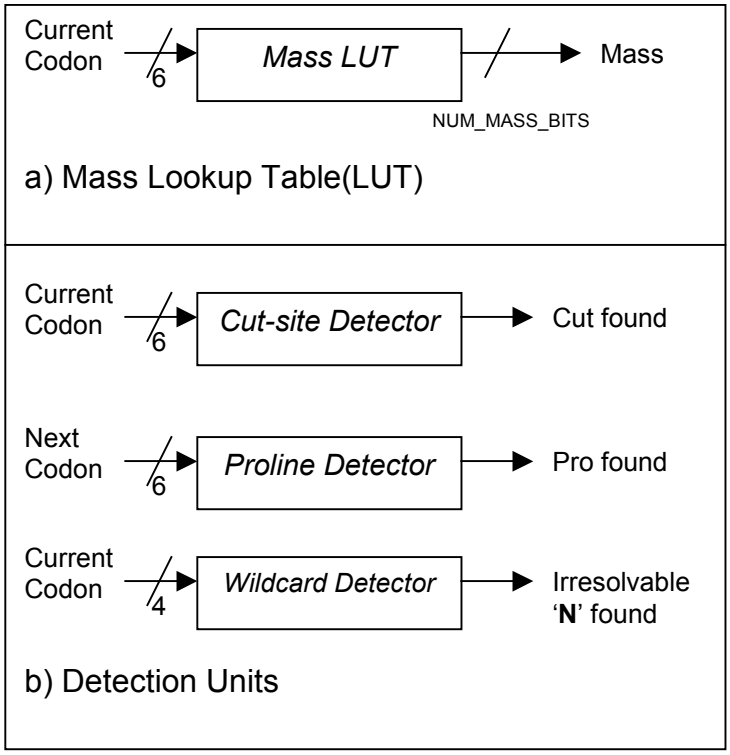


Figure 3-16: Calculator Subunits

The mass LUTs are implemented as ROM tables which accept a 6-bit codon as input and provide a mass value, which is NUM_MASS_BITS bits wide, as output. A codon size of 6 bits implies that only 2 bits are used to represent each of the 3 bases in contrast to the 3-bit per base scheme described thus far. To explain this disparity, consider the binary representation of the codons as described in 3.1. With only four real bases A,T,C and G, a two bit representation is sufficient to encapsulate all possibilities. Recall that the third bit is used to represent the wildcard character. Thus every mass is represented by two data bits and a single wildcard bit. As the mass lookup table is instantiated in BlockRAM, using a 9-bit input for every codon (3-bits per base) would require $2^9 = 512$ storage locations of NUM_MASS_BITS size in the BlockRAM. By using only the two data bits of a base, a codon can be represented in 6 bits. Such an implementation requires only $2^6 = 64$ storage locations. The controller for the mass calculator uses the wildcard bit in combination with the wildcard detector to determine whether there is sufficient

information to translate the codon into its amino acid mass.

The cut-site detection unit looks for the presence of a Lysine (K) or Argnine (R) amino acid in the codon stream. Recall that trypsin cleaves the protein at these amino acids provided that they are not followed by Proline. Thus the Proline detection unit looks ahead to the next codon (see Figure 3-15) to detect the presence of any codon that can synthesize the amino acid Proline. Both the cut-site and Proline detection units take a 6-bit codon as input and output a single bit indicating whether a cut-site or Proline codon was found in the input codon.

The wildcard detection unit looks for the presence of an irresolvable codon in the data from memory. Recall from Section 2.1.2, that the presence of a wild card or 'N' character in a codon does not automatically imply that the resultant amino acid cannot be resolved. In some of these cases, it is still possible to identify amino acid. The wildcard detection unit takes a 4-bit input (corresponding to the last two bases in a codon) and provides a 1-bit output, which is combined with the wildcard bits described above. The controller for the calculator uses this information to determine whether to save or discard the mass produced by a mass lookup table.

3.4.4. Complementary Strand Calculations

As with the search engine, the complementary DNA strand must be accounted for. The tryptic masses for both the strand stored in the genome, and its complement must be calculated. With the hardware above, the masses of tryptic peptides from the original strand can be calculated. For the complementary strand, a copy of this hardware is built which transposes and complements the codons. In Figure 3-17(a) an example string is shown alongside its reverse complement. Likewise, implementations of the cut-site, Proline and wildcard detection units for the complementary strand are instantiated within the calculator.

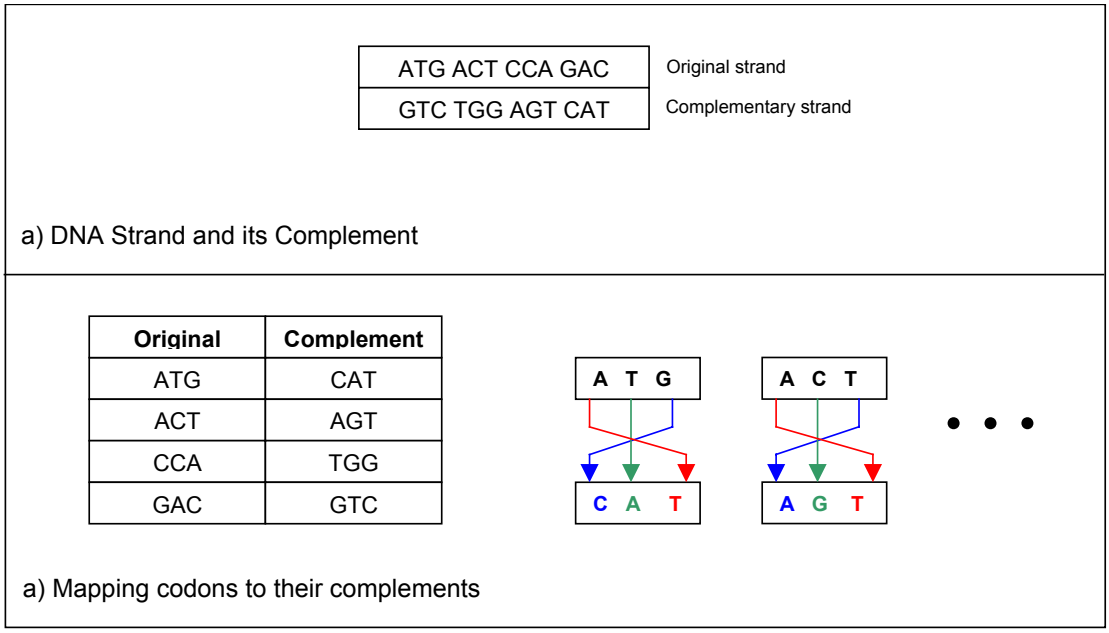


Figure 3-17: Complementary Strand Calculation

Note that to obtain the reverse complement, the original strand is transposed and the bases are replaced with their complements. This corresponds to the reversed translation direction described in Section 2.3.1. However, the codons read from memory arrive in the order of the original strand and do not follow the transposed order depicted in Figure 3-17(a). Thus the codons are accumulated in the forward direction for the original strand (as read from memory), but backwards for the complementary strand.

This merely implies that, for the complementary strand, tryptic mass calculations will begin at the end of the protein. Mass accumulation is an associative process which is unaffected by the direction in which its input codons arrive.

3.4.5. Six Frame Mass Calculation

Each calculator unit computes the masses of one strand and its complement. This accounts for one frame and its complement. To account for the other two frames and their complements, two more calculator units are instantiated; each starts calculations at one base position ahead of its predecessor (see Section 2.3.1 for explanation) and operates identically to the structure described above. To implement this, output of the gene window shift register is read at different base locations by each of the three calculators as shown in Figure 3-18.

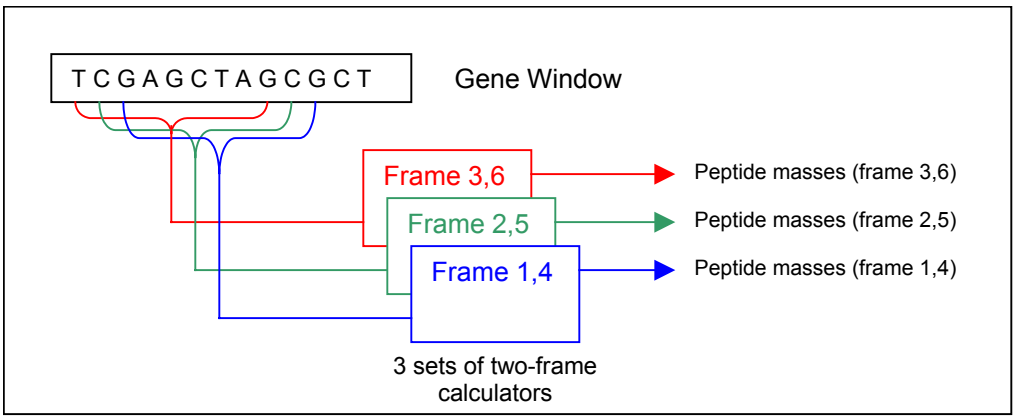


Figure 3-18: Parallel Six-Frame Calculations

3.4.6. Summary of Tryptic Mass Calculator Operations

The search engine identifies locations in the genome that can code the query peptide. The genes surrounding these locations are sent to the tryptic mass calculator to be translated into proteins and digested into tryptic peptides. The calculator then calculates the masses of these tryptic peptides. In the event that there are multiple matching genes, we now have a list of tryptic peptide masses that correspond to each gene. These masses are compared with the peptide masses detected by the MS to uniquely identify the true coding gene.

3.5. Scoring unit

Overview

From Figure 3-1 we see that the calculator described in Section 3.4 produces the masses of tryptic peptides for all genes that coded the peptide query. These calculated masses are then compared with the masses detected by the MS to determine which gene actually codes the protein in the sample. Figure 3-19 elaborates the representation of the scoring unit shown in Figure 3-1. The interested reader can find the VHDL description of this unit in Appendix B 5 (scorer.vhd)

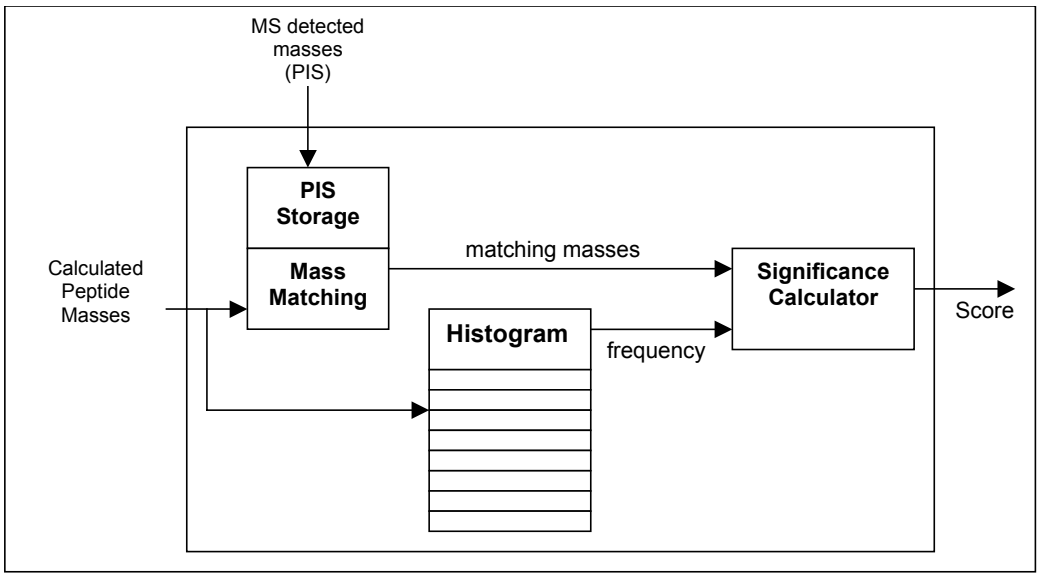


Figure 3-19: Scoring Unit Architecture

Observe that the inputs to the scoring unit are the calculated tryptic masses and the PIS list from the MS. After comparing the two sets of masses, the unit produces a score indicating the quality match. Thus, the scoring unit serves to rank each hit (or gene window) in order of significance. Significance here is defined as the likelihood that a given gene window contains the gene that actually codes the protein in the input sample. Section 2.3.4.2 describes how the significance is computed using a histogram that records the *frequency of occurrence* of mass ranges. To compute this score, the hardware

operates in three distinct states: *True PIS storage*, *histogram construction* and *score calculation*. In the first state the scoring unit merely saves the masses from the true PIS, which are primary inputs to our device. In the *histogram construction* state, peptide masses from the tryptic mass calculator are used to initialize the histogram. Once initialization is complete, the controller moves into the *score calculation* state in which it identifies matches between the calculated masses and those in the stored PIS. The matching masses are used in conjunction with the frequencies stored in the histogram to generate a score for the gene window.

Recall from Section 2.3.4.2, that the score consists of three major components: the ***product term***, the ***maximum frequency*** and the ***number of matches***. In the following sections, we explain the how operations performed in the three states listed above produce these three key components of the score.

3.5.1. True PIS Storage

Upon initialization, the masses detected by the MS (the *true PIS* as defined in Section 2.2.2) are sent as inputs to the scoring unit, which saves them in on-chip RAM. Later, as the calculator generates masses, each must be compared with the stored masses from the PIS. If they fall within a user-defined threshold of each other, a match is signaled.

The first step in this process is to store the mass values from the MS in the on-chip RAM. The storage uses a data-associative indexing scheme similar to *Content Addressable Memory* (CAM). A subset of the most significant bits of the mass value is used to divide the masses into specific ranges as illustrated in Figure 3-20.

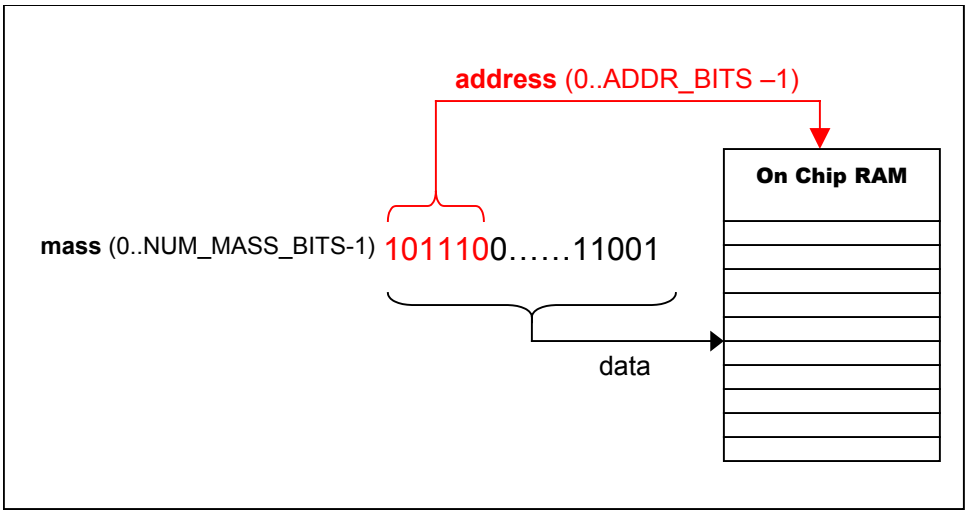


Figure 3-20: Data Associative Mass Storage

In Figure 3-20 a NUM_MASS_BITS sized mass value from the true PIS is sent to the on-chip RAM for storage. ADDR_BITS of the most significant bits from the mass value are used as an address into the on-chip RAM at which to store the mass. This storage method divides the masses into *ranges*; the range that a particular mass falls into is defined by its address. In the example in Figure 3-20, the mass will be stored at address 46 (101110).

It is clearly possible for two different masses to be stored at the same address if ADDR_BITS of their most significant bits are identical. To avoid this situation, we constrain the design such that ADDR_BITS must be sufficiently large enough to ensure that data will not be overwritten. Upon device initialization, each of the PIS masses from the MS is stored in the on-chip RAM using this technique.

3.5.2. Histogram Construction

In the second state, the scoring unit initializes a histogram with NUM_BINS bins. As the mass calculator operates, its outputs are passed into the scoring unit. Recall from Section 2.3.4.1, that the histogram records the frequency of occurrence of peptides in

different mass ranges. To this end, decoders are used to identify which range a given mass falls into and a set of counters is used to determine how many masses fall into a given range.

Figure 3-21 illustrates how the decoders and counters described are used to update the histogram. Refer to Appendix B 6 for the VHDL description of controller that implements this process (mod_frequency_table.vhd).

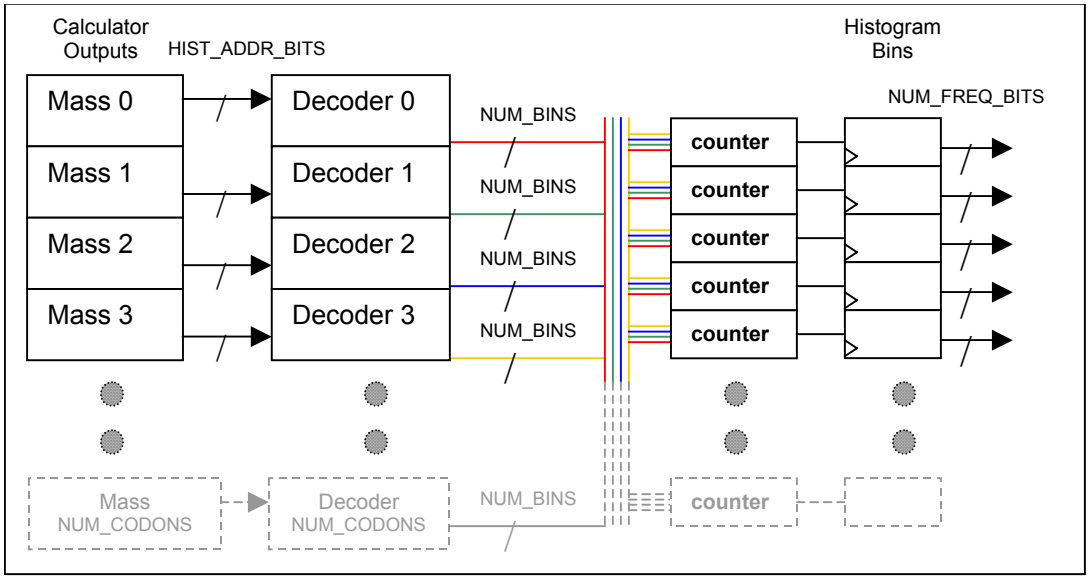


Figure 3-21: Building the Frequency Histogram

Firstly note that the bins in Figure 3-21 are simply a set of NUM_BINS registers that are NUM_FREQ_BITS bits in width. Each resister, or bin, represents a range of mass and contains the number of peptides in the current gene window that fall into this range. The counters at the inputs of these registers identify how many of the peptides from the calculator fall into a given range. The counter then updates the bin appropriately. Recall that the calculator is capable of producing NUM_CODONS + 1 masses in a single cycle. Thus in every clock cycle, any bin in the histogram can be incremented by a maximum of NUM_CODONS + 1 peptides.

As mentioned, binary decoders are used to determine the range into which a calculated mass falls. The decoder has $\log_2$(NUM_BINS) inputs and NUM_BINS outputs. Each

output signal of the decoder corresponds to one of the NUM_BINS bins. Therefore $\log_2$(NUM_BINS) bits of the mass (defined as HIST_ADDR_BITS) are required to determine the range a given mass falls into. There are NUM_CODONS + 1 decoders, each corresponding to single output of the calculator.

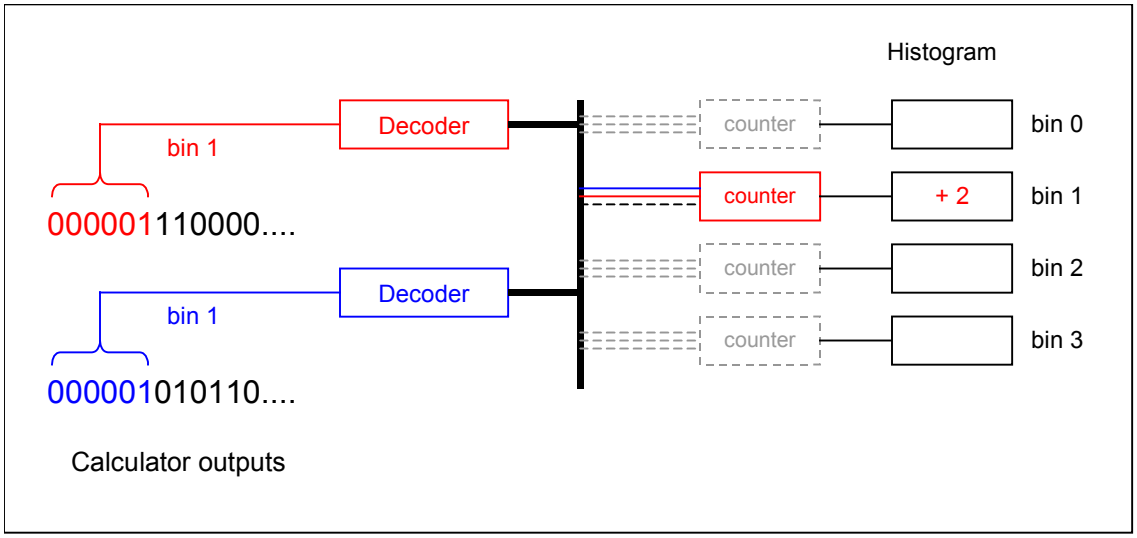


Figure 3-22: Updating the Histogram

An example of a histogram update is presented in Figure 3-22 for clarity. In this example two calculator outputs are shown. While both masses are different, HIST_ADDR_BITS of their most significant bits (6 bits in this example) are the same, thus both fall into the same bin (bin 1). Both decoders activate the output corresponding to bin 1, and the bin 1 counter correspondingly indicates that the histogram should increment the value in bin 1 by 2. Using this approach, we can record the frequency of occurrence for each calculated peptide mass. Once a full gene window has been processed, the bins are passed through a shift register, which *identifies the mass range that occurs most frequently*. The ***maximum frequency*** is one of the key components of the score and is returned to the user. The entire histogram update process occurs in parallel with the operation of the calculator, but an additional NUM_BINS cycles are required to identify the maximum frequency. The next phase uses this histogram to calculate the significance of the matching masses as shown in Figure 3-19.

3.5.3. Score Calculation

Once the masses from the PIS have been stored and the histogram has been initialized, the score calculation process begins. This process consists of two operations that occur in parallel: *mass matching* and *significance computation*. The mass matching operation compares every calculated mass to the PIS values saved in the on-chip RAM to identify any matches. The significance computation uses these matching masses to determine the significance of the gene window at a hit location. The two remaining components of the final score, namely the ***number of matches*** and the ***product term*** are calculated by these operations. The following sections describe the architecture and operation of the hardware that implements these operations.

3.5.3.1. Mass Matching

Once the histogram has been initialized, the masses from the tryptic peptide calculator are once again sent to the scoring unit. In this state however, the masses are not used to update the histogram. Instead, the calculated masses are compared with the true PIS masses that were stored earlier to identify any matches between the tryptic peptides in the current gene window and those detected by the MS. Figure 3-23 represents the architecture implemented to perform the mass matching operations.

The goal of the mass matching hardware is to identify calculated masses that fall within a user defined threshold of a value in the true PIS. Given a tryptic peptide mass from the calculator, we identify its closest corresponding mass in the true PIS by once again using data associative techniques. To see how the closest matches are identified, recall the storage scheme used to save the true PIS.

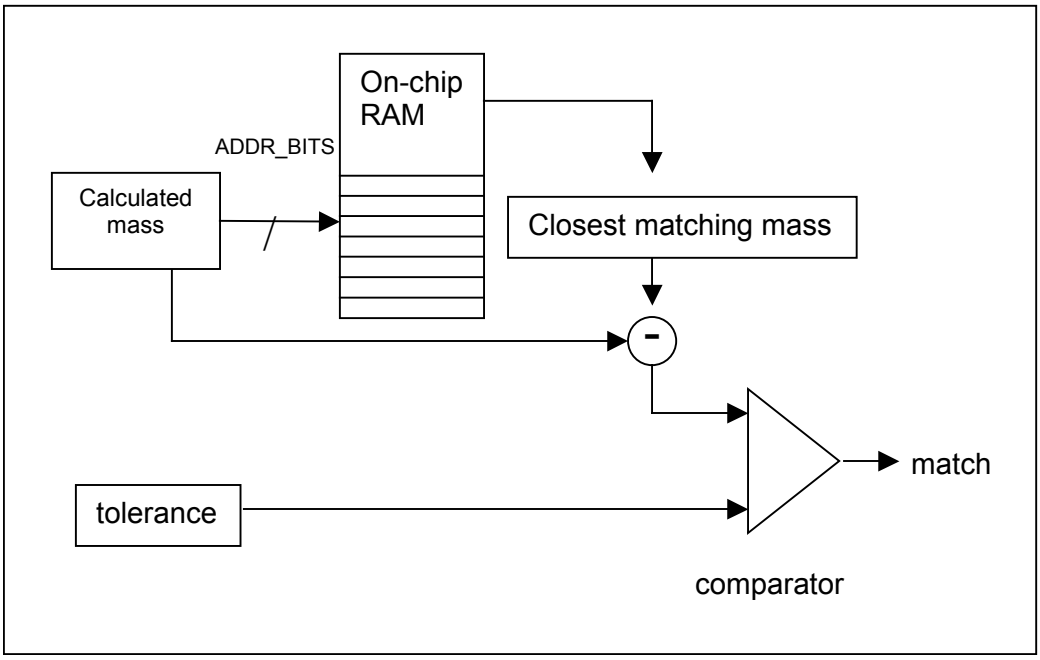


Figure 3-23: Mass Matching

The on-chip RAM, in which the true PIS masses are stored, is set into a read only mode and ADDR_BITS of the most significant bits of the masses from the calculator are used as addresses. Doing so retrieves the PIS mass that was stored at the same address, i.e. the retrieved PIS mass falls into the same range as the calculated mass.

The difference between the calculated mass and the stored PIS mass is then calculated. This difference is passed to a comparator along with a user-defined threshold. If the difference is less than or equal to the threshold, the comparator signals a match as illustrated in Figure 3-23. The match signal is passed to the controller, which increments a counter to keep track of the ***total number of matches*** found in a window. Recall that this is one of the key components of the final score for the current gene window.

The matching masses identified here are used in the significance calculation step where the final component of the score, namely the ***product term***, is computed. This process is detailed in the following section.

3.5.3.2. Significance Calculation for Matching Masses

In addition to the number of matches, the scoring algorithm described in Section 2.3.4.2 ranks the matches by significance. Figure 3-19 shows that the significance calculator receives frequency values from the histogram in addition to the matching mass values. The purpose of the significance calculator then, is to determine the ranges into which matching masses fall, and compute the product of the frequencies of these ranges. This corresponds to the ***product term*** defined in Section 2.3.4.2.

Recall that the peptide mass calculator can produce a maximum of NUM_CODONS+1 matching masses (i.e. every output of the calculator matches a mass value in the true PIS). To account for this event, the most significant HIST_ADDR_BITS bits of matching masses are used to identify the range the mass falls into. The frequency of this range is read from the appropriate bin of the histogram and placed in a pipeline as shown in Figure 3-24. As with the tryptic mass calculator, the pipeline is used to ensure that the product of the frequencies of multiple matching masses can be computed per cycle to meet the throughput of the calculator. Each of the NUM_CODONS+1 stages of the pipeline processes a single frequency value per cycle. In the subsequent cycle, the unprocessed frequencies from every stage are passed to the following stage. However, the processing units depicted in Figure 3-24 do not directly compute the product of the frequencies.

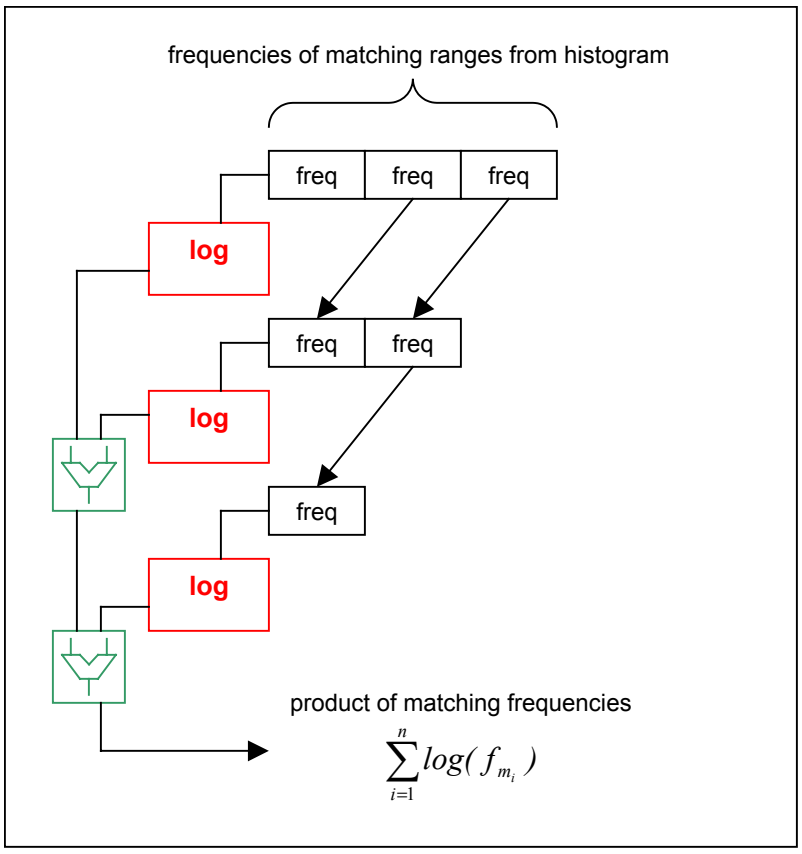


Figure 3-24: Calculation of the Product Term

To calculate the product of the frequencies in the pipeline, the technique of logarithmic addition is employed as represented by the log and accumulator blocks in Figure 3-24. This method relies on the fact that $log(\prod_{i=1}^{n} f_{m_i}) = \sum_{i=1}^{n} log(f_{m_i})$, where f_m corresponds to the frequency of a matching range and n is the total number of matches. Thus, instead of explicitly calculating the product of the frequencies, we take the sum of the logarithms of these values. The actual product can be determined by taking the inverse of the logarithm of the accumulated value. We use this approach primarily to ensure that the product term can span a large range. The logarithm units are NUM_FREQ_BITS bits wide allowing for values between 0 to $2^{NUM\_FREQ\_BITS}$ to be represented. These values are calculated in hardware by lookup tables, which take a NUM_FREQ_BITS sized frequency value as input and produce $\log_{10}$(frequency) as its output. Since the frequencies themselves are integer values from 0 to $2^{NUM\_FREQ\_BITS}$, this simple scheme is sufficient to calculate the

logarithms. The sum of these logarithms is computed by a set of accumulators to obtain the logarithm of the product term. This value is returned to the user, where the logarithm is inverted to obtain the final product term. This ***product term***, along with the ***maximum frequency*** and the total ***number of matches*** between the hypothetical PIS and the MS detected values, is returned to the user to calculate the final score given by.

$$Score = \frac{1}{\frac{product\_term}{(maximum\_frequency)^{total\_number\_of\_matches}}}$$

This corresponds to the scoring method described in Section 2.3.4.2.

A small product term indicates a match to an infrequent mass range, which corresponds to a high score as explained in Section 2.3.4.1. In practice, the actual score values produced by this formula vary in orders of magnitude i.e. high and low scores are typically several orders of magnitude apart. Therefore it is common for these scoring schemes to use 10 log(Score) as the final score value. In the results presented in Section 4 we adopt this notation.

3.5.4. Six Frame Score Calculations

Section 3.4.5 states that the calculators generate six frames of masses simultaneously. Each of these frames can be treated as an independent gene as each encodes a different set of tryptic peptides. Thus six corresponding scoring units, are instantiated in the hardware, each of which computes the score of an individual frame of the gene under consideration. Therefore each hit in the database is returned to the user with 6 sets of scoring information. Since only one of these six frames is the true coding region, the frame that generates the maximum final score for a given gene window is considered to be the true coding frame.

3.6. Design Summary

We have presented the design of a hardware system that meets the key requirements stated in Section 2.2.3. Figure 3-1 illustrates an overview of the key subunits of the device.

1. A search engine that accepts a peptide query from the MS and *locates all coding regions* of the peptide in the genome.

2. A tryptic peptide mass calculator that *translates and digests the genes* around the located coding regions to *produce the mass of the tryptic peptides* that are contained in the proteins encoded by these genes.

3. A scoring unit that accepts the calculated tryptic peptide masses (the hypothetical PIS as described in Section 2.2.2) and *compares the calculated masses to the true PIS from the MS*. The scoring unit assigns a score to each set of tryptic masses based on their significance. Each location identified by the search engine is associated with its score and returned to the user to determine the true coding region.

The final requirement presented in Section 2.2.3 was the ability to perform all the operations above within 1 second. In the following chapter, we look at the performance and cost of the hardware presented in this chapter in comparison to software running a similar algorithm. The results show that this design meets the speed requirements of current mass spectrometry at a significantly lower cost than an equivalent algorithm implemented in software.

Chapter 4. Implementation Details & Results

4.1. Overview

The protein identification system described in Chapter 3 performs a reverse translated peptide query search through a Genome database. It locates all genes that can potentially code the query peptide and translates them into proteins. It then uses a variant of the MOWSE algorithm to compare the masses of these translated proteins to the masses in the PIS of a tandem mass spectrometer. This technique identifies and ranks potential coding regions for a protein or set of proteins in an MS sample. The coding regions can be sent to gene finding programs [44][45] or homology search tools [31] to obtain the protein sequence. In this chapter, we specify the implementation parameters for the design presented in Chapter 3. We then demonstrate the functionality of the design by providing input data obtained from biological experiments. This is followed by an evaluation of the speed and area of the design as realized on several modern FPGAs. Finally, we present the cost of the hardware system in comparison to a software implementation capable of similar performance.

4.2. Assumptions and Approximations

In implementing the design described in Chapter 3, a number of design decisions were made based on constraints imposed by the hardware and knowledge of the biological attributes of the problem. This section describes the dataset used to obtain our results, and details the assumptions that governed our design decisions.

4.2.1. Using Simpler Organisms

Ideally the algorithms described here should be tested on the Human Genome, since most studies ultimately target human proteins. Unfortunately it is difficult to obtain MS data from human subjects for research purposes. However, biologists often capitalize on the similarity of the genomes of various organisms. The genomes of simple organisms ranging from flies to yeast have been used to infer the behaviour of more complex

organisms such as humans [33][34]. For our study we use MS data from the organism *Saccharomyces cerevisiae*, commonly known as baker's yeast. The yeast genome is an excellent model for the human genome since both are eukaryotes and thus share several similar proteins [34]. Further studies must be conducted with human MS data to verify the results presented here, but the insights gained from yeast can be used to identify the strengths and weaknesses of the algorithm. We define the operational parameters for the device in the following section, bearing in mind that the data in the yeast genome acts as the basis for our assumptions.

4.2.2. Implementation Parameters

The parameters defined in Chapter 3 are governed by physical constraints of the implementation platform and the particular biology of the organism under consideration. In this section, we detail the biological assumptions and hardware constraints that help us resolve the optimal values for the design parameters.

Input Data

The first key parameter is the size of the genome. The yeast genome [27] consists of 12070522 bases, which defines the parameter SIZE_OF_GENOME as 3.4 megabytes using the compression described in Section 3.1. For comparison, the human genome is 918 megabytes.

Search Engine

In the search engine, the most crucial parameters are MEM_WIDTH and NUM_BASES_IN_MEMWORD, as they dictate the throughput of the system at a given operating frequency. The memory word read from the TM3A is 64 bits wide, but the compression scheme operates on multiples of 7 bits; therefore we use a MEM_WIDTH of 63 bits. The compression scheme uses 7 bits to encode a codon (or 3 bases) resulting in a NUM_BASES_IN_MEMWORD of 27 bases.

Gene Window

After passing through the search engine, the uncompressed memory word enters the gene window before it is sent to the calculator. Recall from Section 3.3.4, that the size of the organism's gene governs the size of the gene window upon which the calculator operates. Studies of the genes in yeast have shown the average gene size to be approximately 1450 bases [32]. The gene window is thus implemented as 18-word 81-bit shift register (corresponding to a GENE_SIZE of 1458 bases). In contrast, the average gene size in human chromosome 7 is 70,000 bases with 10% of the genes as large as 500,000 bases. This expansion in size is due to more alternative splicing (55% of chromosome 7 genes are spliced as opposed to 4% in yeast) [52].

Mass Calculator

The bases from the gene window are read and translated by the calculator into peptide masses. Measurements on the dataset showed that tryptic peptides range in mass from 0 to 10 KDa a 20-bit mass value ((2^{20} = 1048576) allows for masses between 0 and 10,485.76 Da. However for an additional level of precision, 5 more bits are used to further divide these masses into 0.0003125 Da ranges. Thus NUM_MASS_BITS is set to 25 bits.

Scoring Unit

The masses from the calculator are passed to the scoring unit, which ranks them in a similar manner to the MOWSE algorithm. MOWSE defines bins of 100 Da, which we approximate by setting NUM_BINS to 128 bins. In the mass range between 0 and 10,485.76 Da, this translates to bins of approximately 82 Da. The choice of 128 bins in turn defines HIST_ADDR_BITS as 7 bits, as 7 bits of the mass are needed to identify 127 bins.

For convenience, these design parameters are listed in Table 4-1.

Parameter	Values (Yeast)	Values (Human)
MEM_WIDTH	**63** bits	**63**
SIZE_OF_GENOME	**3.4** Megabytes	**917** Megabytes
NUM_CODONS	**9** codons	**9** codons
GENE_SIZE	**1458** bases	**35000** bases
ADDR_BITS	**9** bits	**9** bits
NUM_MASS_BITS	**20** bits	**20** bits
NUM_BASES_IN_MEMWORD	**27** bases	**27** bases
HIST_ADDR_BITS	**7** bits	**7** bits
NUM_BINS	**128** bins	**128** bins
NUM_FREQ_BITS	**8** bits	**8** bits

Table 4-1: Design Parameters

The parameter values in Table 4-1 are chosen for a design with sufficient resources to perform the scoring operations accurately. In the following section we present the implementation details of a device designed with these parameter values.

4.3. Implementation Details

In this section, the particulars of the design implemented with the values in Table 4-1 are presented. Firstly, the functionality of the design when used with MS data is shown. In the subsequent sections, hardware and software platforms implementing the design at varying levels of performance are considered. Finally the costs of these systems are compared in an attempt to identify a practical solution.

4.3.1. Functionality

The following tests were performed to gauge the performance of the system with real MS data. The data used were obtained from the study performed in [60]. The study utilized Liquid chromatography tandem mass spectrometry (LC-MS/MS) analysis using a

Finnigan LCQ Deca ion trap mass spectrometer fitted with a Nanospray source. Protein identification was performed by the search engines Mascot [39], Sonar [67], Sequest [68] and PepSea [69]. Interested readers can find the PIS masses reproduced in Appendix D. The input sample used in the experiment contains two well-characterized proteins from Saccharomyces cerevisiae (baker's yeast):

1. A **Rab Escort Protein (REP) [**ACCESSION: NP_015015]
2. A heat shock protein from the SSB2 variant of the **HSP70** family [ACCESSION: NP_014190]

Rab Escort Protein (REP)

The REP in the protein sample is from the MRS6 family of proteins created by the MRS6 gene, located in yeast chromosome 15. A full gene map is located on the Saccharomyces Genome Database (SGD) [30].
Its coordinates in our database (i.e. the bases that the gene spans) are:

- from 1025599 to 1026956. (*located in Chromosome 15* [30])

Heat Shock Protein (HSP70)

The HSP70 family is coded by the SSB1 and SSB2 genes located on chromosomes 4 and 14 respectively. Our sample contains the SSB2 subfamily variant coded by the gene in chromosome 14.
Each of these chromosomes codes a different subfamily of the HSP70 proteins but both have extremely similar sequences (BLAST [31] of the 2 sequences shows 551 out of 613 matching amino acids (89% identity)). A full gene map is located on the SGD. (*located in Chromosome 4* [28], *located in Chromosome 14* [29])

Its coordinates in our database are:

- from 1427427 to 1429279. SSB1 variant (*located in Chromosome 4*)
- from 9661724 to 9663575. SSB2 variant (*located in Chromosome 14*)

Table 4-2 lists the some of the peptides that were provided as queries to the search engine alongside the hit locations reported by the search engine.

Protein	Query Sequences *(minimal query)*[2]	Hit Location(s)
REP	vpealqr *(vpealq)*	1025938
	saavggptyk *(saavg)*	1026060
HSP70	nttvptik *(nttvpt)*	1428705 9663002
	llsdffdgk *(llsdff)*	1428495 9662792
	tgldisddar *(tgldis)*	1428190 9662487
	fedlnaalfk *(fedlna)*	1428352 9662648

Table 4-2: Query peptides and hit locations for HSP70 and REP

The first important observation is that any query sequence greater than 5 amino acids in length always uniquely identifies a single coding region, eliminating the need for a scoring function. Note that the peptides from HSP70 are shown as originating from two hit locations. Recall that there are two variants of this family encoded by different genes, but having highly similar sequences. However the 11% difference in sequence guarantees that the set of tryptic peptides generated by both variants is not the same. The scoring system helps resolve the two hits and uniquely identify the protein in the sample.

[2] The *minimal query* (in italics under the query) is the shortest peptide sequence that still identifies a unique coding region

Protein	Query Sequences	Hit Location(s) *(Gene)*	Score
HSP70	nttvptik	1428705 *(SSB1)* 9663002 *(SSB2)*	62 89*
	llsdffdgk	1428495 *(SSB1)* 9662792 *(SSB2)*	65 89*
	tgldisddar	1428190 *(SSB1)* 9662487 *(SSB2)*	67 88*
	fedlnaalfk	1428352 *(SSB1)* 9662648 *(SSB2)*	66 88*

Table 4-3: Score identifies subfamily variant in HSP70

In Table 4-3, the HSP70 peptide queries are shown alongside their scores. In each case, the SSB2 encoding (indicated by the * next to the score) has a higher score, corresponding to the variant that is in the sample.

Each of the queries shown above is 5 amino acids or greater in length. An average sequence detected from a tryptic peptide may be up to 10 amino acids in length, but shorter sequences are common. Further, it is possible that only a short sequence can be determined for a long tryptic peptide due to instrument limitations, sample contamination etc. These shorter peptide queries to the genome have lower resolution and will result in multiple matches.

We consider a few smaller peptides to test the resolution of the scoring function. These peptides were also identified by the mass spectrometer, but are shorter than the average peptide length, thus they are likely to encounter multiple matches within the genome.

Protein	Query Sequences	Hit Location(s)	Score
REP	eyvpr	1026605 6672335 2264445	79* 76 66
	ilfak	1938133 1323971 5006575 6224783 1025581 5231459 9309092 3108258	96 90 89 84 72* 71 70 61

Table 4-4: Queries with multiple matches in REP

In Table 4-4 the queries “iflak” and “eyvpr” both generate false positives as expected. The query "eyvpr" in Table 4-4 is ranked correctly, and the true coding location gets the highest score. However, the second query is ranked incorrectly, with the true hit being ranked fifth. Recall that scoring functions are highly sensitive to the data that they operate on [46] and the MOWSE algorithm that we use was not intended for genome wide searches [9].

In cases where the query sequence is short and cannot be resolved to a unique gene location, multiple peptide queries may be used to identify the true coding region. This approach relies on the assumption that multiple matches are random, which may not always be true. For example, Table 4-3 showed multiple matches due to the fact that the two hit locations coded proteins that were similar or *homologues*. These matches were clearly not random, however most of the cases with multiple matches are random and occur due to the volume of data contained in the genome [1].

To see how multiple sequences can resolve the random false positive matches, such as those in Table 4-4, observe the distribution of match locations. Each match corresponds to a gene location that codes the query peptide. In non-homologous proteins it is unlikely that several proteins will share common peptide sequences. Peptide mass fingerprinting (PMF) techniques make use of this fact to use a few peptides to discriminate between tens of thousands of proteins in protein databases.

Any short peptide query will match the true gene location and may produce several false positives. Thus if several peptide queries are used, the matches will be clustered together (within the true coding gene) while the false positives will be randomly distributed throughout the genome.

This can easily be seen in the data in Table 4-4. The two true matches are only 1024 bases apart, which is within the size of a single gene. The next closest match occurs between the hit at 1026605 and 1323971, but these locations are 297366 bases apart. It is thus easy to identify the true hits as they are clustered together.

"ilfak" hit locations	Closest Match in "eyvpr"	Distance to closest match
1025581	1026605	1024
1323971	1026605	297366
1938133	2264445	326312
3108258	2264445	843813
5006575	6672335	1665760
5231459	6672335	1440876
6224783	6672335	447552
9309092	6672335	2636757

Table 4-5: Closest Distances between Match Locations

Table 4-5 shows the distance between the closest matches using the two peptide queries from Table 4-4. Using this information, we deduce that matches that are close to each other indicate the presence of peptides being coded by the same gene, which in turn corresponds to the true hit location. Thus, the inverse of the difference between match locations is used to identify the true coding gene.

In Figure 4-1 a scaled representation of the distance between the two queries is presented. The *inverse of the distance between matches* - which we define as "*closeness"* - is presented across all bases in the genome in Figure 4-1. Note that the closeness value is scaled by a factor of 1×10^{7} for better visualization.

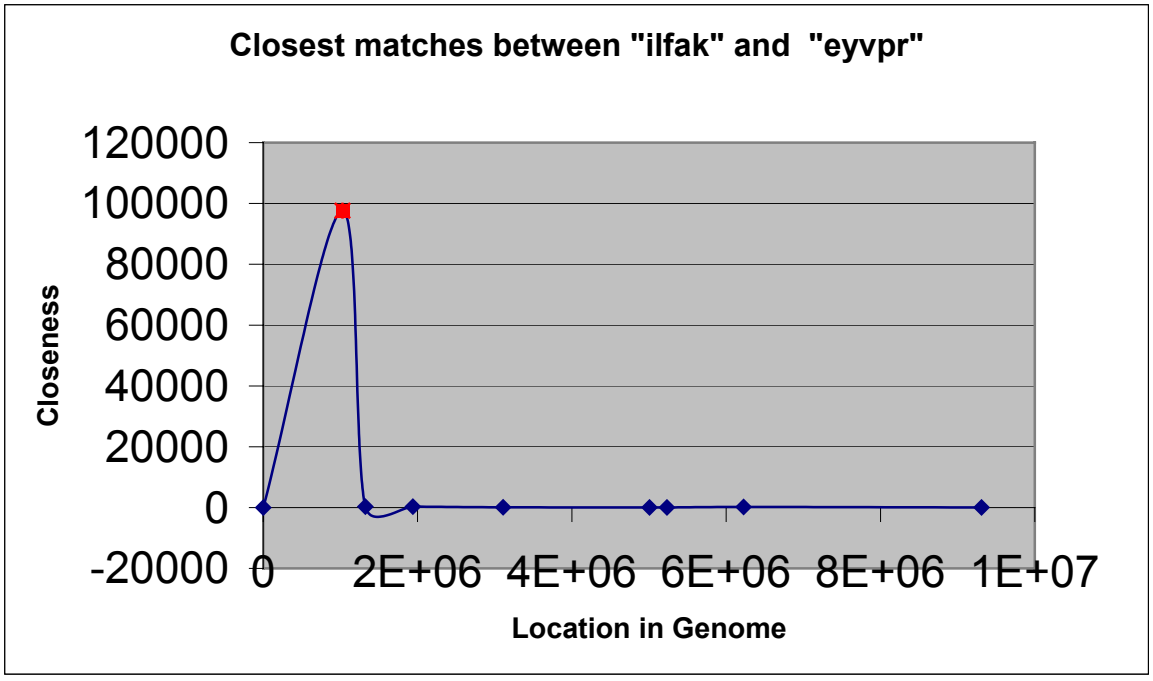


Figure 4-1: Visualizing "iflak" and "eyvpr" Hits Across the Genome

Note in Figure 4-1 that the true hit can be clearly distinguished from the other matches. Thus by using two peptides we can identify hits that cluster around a single gene and thereby discriminate a coding gene from random matches.

The short query peptides in Table 4-4 above are natural, i.e., the peptides occur naturally via trypsin digestion. However, similar cases arise if the quality of the sample is poor and only a few amino acids can be sequenced. In these cases, the MS may only be able to resolve a short length of full tryptic peptide, forcing the MS operator to search the database with a shorter query.

To replicate the effect of these low quality samples we search using queries that are smaller than the *minimal query*. In effect, we are using substrings of the queries in Table 4-2 to simulate the behaviour of "dirty" samples.

In the following example the two queries "saavggptyk" and "eyvpr" from Table 4-2 and Table 4-4 respectively are considered. To simulate low-quality sequences, we use the substrings "saav" and "eyvp" of these peptide sequences. For brevity the full set of

matches are deferred to Appendix C. However, the true hits are ranked 65th of 128 hits and 13th out of 48 hits for the queries "saav" and "eyvp" respectively. It is clear that the MOWSE scoring algorithm cannot distinguish the true coding locations from false positives. However, using the technique summarized in Table 4-5, we can look for the distance between hits.

The 5 closest matches are presented in Table 4-6.

"saav" Hit Locations	Closest Match in "eyvp"	Distance to Closest Match
1026060	1026605	545
7486943	7488841	1898
8964661	8965326	2305
10170118	10165117	5001
9383697	9378467	5230

Table 4-6: Distance Between Hits in "eyvp" and "saav"

As before, the inverse of the distance to the closest match – the *closeness* - between hits produces a map of the genome in which the true coding gene is easily identifiable.

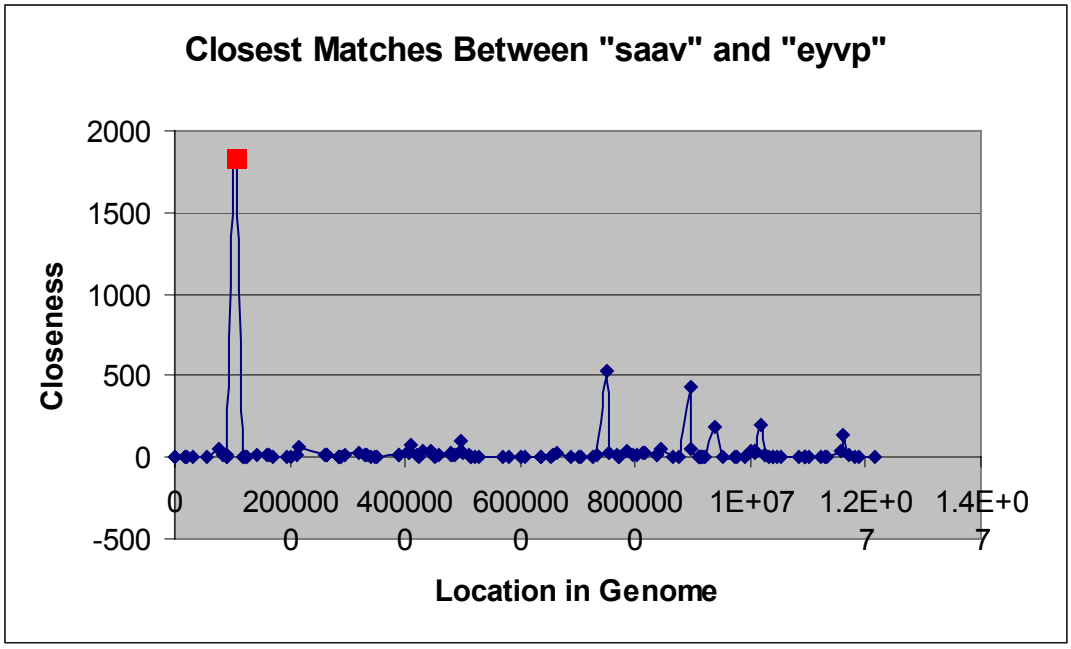


Figure 4-2: Visualizing Matches Between "*saav*" and "*eyvp*" across the genome

Note, that the true hit can easily be distinguished from 127 false positives, even when the query is only four amino acids long.

The results in Appendix C show that in many cases, the true coding region can be easily identified by using multiple queries. With a query of five amino acids, the true coding location was always correctly identified using two peptide queries to the database. When using a query length of four amino acids, the number of hits per query increases. With more hits, more queries are required to accurately identify the true coding region. Appendix C shows that using two queries of length four identifies the true hit in eight of 12 searches. Of the four erroneous cases, the true hit location is ranked 2nd in three of these and 3rd in the remaining case. In each of these cases, the distance between hits can be calculated in a few milliseconds, without significant impact on the speed of the search and score process.

4.3.2. Design Implementation on the TM3A

The TM3A described in Section 2.5.3, was the primary implementation platform for our design. Considering the architecture of the TM3A, the device was partitioned across four FPGAs as shown in **Error! Reference source not found.**.

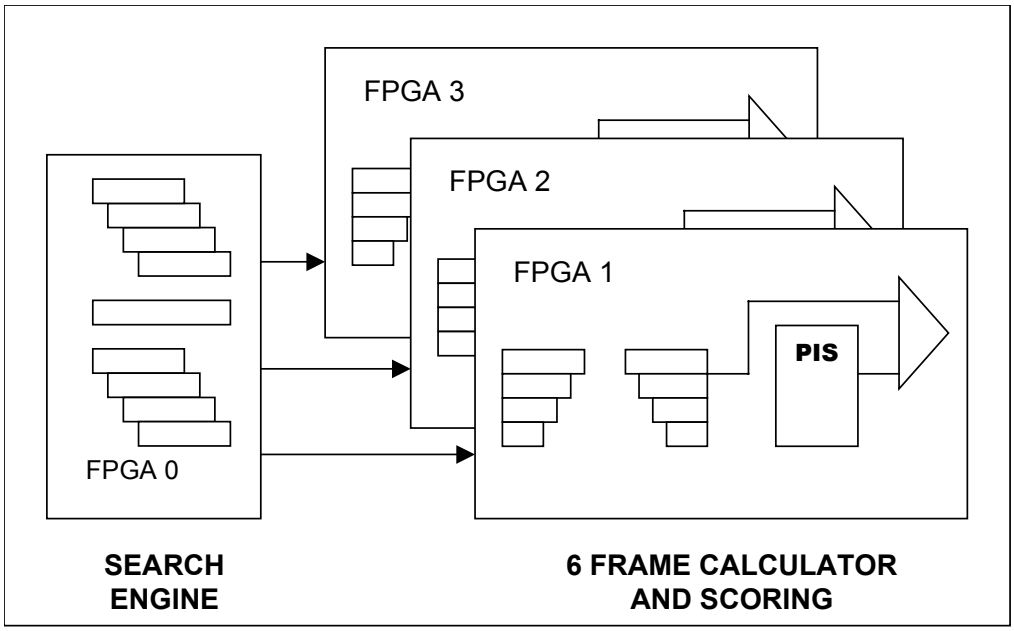


Figure 4-3 Device Partitioned Across TM3A

The design is partitioned as follows:

- FPGA 0: Search Engine and Gene Window
- FPGA 1: Mass Calculator and Scoring Units (for Frames 1 and 4)
- FPGA 2: Mass Calculator and Scoring Units (for Frames 2 and 5)
- FPGA 3: Mass Calculator and Scoring Units (for Frames 3 and 6)

Note that FPGAs 1, 2 and 3 have identical units implemented on them. The distinction lies in the data that they receive from the gene window. FPGA1 receives the data from the gene window directly, and produces the scores from Frame 1 and its complement (Frame 4). FPGA2 and FPGA3 receive the data from the gene window shifted by 1 base and 2 bases respectively, and correspondingly produce the scores of Frames 2 and 3 and their complements. Using this structure, the individual FPGAs can be classified by the units they implement. Therefore our design will be described in terms of search engine FPGAs and calculator and scoring unit FPGAs.

Unfortunately, compiling the design with the parameter values described in the previous section resulted in an implementation that did not fit on the TM3A due to

insufficient resources. The 25-bit mass and 128-bin histogram force the calculator and scoring units to occupy more area than is available on a Xilinx Virtex 2000E FPGA. In combination, these units occupy 44338 LUTs and flip-flops, but Table 4-7 shows that the Virtex 2000 E chips on the TM3A only have 38,400 LUTs and flip-flops.

FPGA	Number of LUTs and FFs	Block RAM (bits)	User IO pins
Virtex 2000E	38,400	655,360	804
Virtex II 8000	93,184	3,024,000	1,108
Stratix EP1-S20	18,460	1,669,248	586
Stratix EP1-S40	41,250	3,423,744	822
Stratix EP1-S80	79,040	7,427,520	1,238

Table 4-7: FPGA resource comparison

In an attempt to fit the device on the TM3A, the design was modified to use 18-bit masses with a 64-bin histogram thus reducing the area occupied by the calculator and scoring units. This modification enabled the units to fit on the TM3A, and the speed and area results for the individual FPGAs are presented below.

Design Platform	LUTs	FFs	Memory (bits)	Operating Frequency (MHz)	Search Time through Human Genome (s)
TM3A - Virtex 2000E	8,622	1,858	8,786	89	1.4

Table 4-8: Total Resources and Speed for Search Engine on Virtex 2000 E

Design Platform	LUTs	FFs	Memory (bits)	Operating Frequency (MHz)	Processing Time for Human Genome (s)
TM3A - Virtex 2000E	27,925	12,475	34,816	58	2.1

Table 4-9: Total Resources and Speed for Combined 2-Frame Calculator and Scoring Units on Virtex 2000E

Note that the searching and scoring times shown are for the human genome, and not yeast, as presented in Section 4.2.1. Recall that the ultimate goal of these sequencing experiments is to identify human proteins; the search times presented in Table 4-9 are more relevant when evaluating the practicality of our tool in useful biological experiments. The functionality of the device is not dependent upon the organism under consideration; indeed the only parameter affected is the value of SIZE_OF_GENOME, which is set to 918 megabytes (approximately 1 GB) when using the human genome.

From the tables above, it is apparent that the calculator and scoring units limit, and thus define, the system speed. Table 4-9 shows that it takes 2.1 seconds to identify and score all gene locations that match a single peptide query. This speed however is not achievable on the TM3A due to the limited speed of the SRAM. The operating frequencies in Table 4-8 and Table 4-9 apply only to the FPGA under consideration and are independent of memory speeds. The SRAM on the TM3A operates at a maximum frequency of 50 MHz making it the system bottleneck. Taking the memory speed into account, we restrict the operating frequency of the system to 50 MHz and calculate the operating time for a single query to be 2.4 seconds.

In addition to the memory bottleneck, further problems arise as a result of the reduction in accuracy mentioned above. Using the less accurate 18-bit mass representation and coarser 64-bin histogram severely lower the performance of the scoring algorithm, thus the area and system speed presented above are not representative of a practical design. Note that this limitation only applies to the calculator and scoring units. The search engine fits on a Virtex 2000E FPGA and is not affected by the reduced parameters. Regardless, it is obvious that the TM3A, while a practical prototyping tool, is not adequately equipped to implement this design.

To obtain realistic figures for area and speed, the design was recompiled with the parameters in Table 4-1 to target a set of modern FPGAs with more resources. These results are presented in the following section.

4.3.3. Design Implementation on Modern FPGAs

The limitations of the TM3A impose unacceptable constraints on the speed and accuracy of the design. To overcome these constraints, we describe the implementation of our design using modern FPGAs and high-speed commercial memory. The FPGAs under consideration are listed in Table 4-7.

Note that the newer FPGAs, namely the Xilinx Virtex II 8000 FPGA [56] and the Altera Stratix S40 and S80 FPGAs [57], all have more resources than the Virtex 2000E FPGAs on the TM3A. The Stratix S20 is included in Table 4-8 as it is the smallest FPGA upon which a search engine will fit.

The speed and resource utilization tables are once again partitioned into individual FPGAs. The implementation of the search engine on each of the FPGAs is shown in Table 4-10. Correspondingly the implementation of the calculator and scoring units upon the Virtex II 8000 and Stratix S40 and S80 FPGAs is shown in Table 4-11. Due to the lack of resources on the Stratix S20, the calculator and scoring units do not fit on it.

Search Engine

FPGA	LUTs	Flip Flops	Memory Bits	Operating Frequency (MHz)	Search Time (s)
Stratix S20	10,605	1,694	7,938	163	0.7
Stratix S40	10,605	1,694	7,938	152	0.8
Stratix S80	10,605	1,694	7,938	148	0.8

Table 4-10: Total Resources and Speed for Search Engine using Current Technology

The reduced operating frequency on the larger devices in Table 4-10 can be attributed to the fact that the smaller devices have shorter wires, which have less capacitance, and are thus faster.

Two Frame Calculator and Scoring Unit

FPGA	LUTs	Flip Flops	Memory Bits	Operating Frequency (MHz)	Search Time (s)
Virtex II 8000	28,786	15,552	204,800	62	1.97
Stratix S40	30,684	13,814	205,244	75	1.63
Stratix S80	30,684	13,814	205,244	75	1.63

Table 4-11: Total Resources and Speed for Combined 2-Frame Calculator and Scoring Units using Current Technology

The difference between the number of flip flops and memory bits between the Virtex and Stratix FPGA can be attributed to the different synthesis and mapping tools used to implement the circuits. Various parts of the circuit are mapped to different structures (LUTs or BlockRAM) by the tools, which are tailored to find the best possible implementation of a circuit on a given device.

Note once again that the operating frequencies reported in the tables are independent of memory speeds and are based on a 63-bit memory word as indicated in Table 4-1. However, as we are not constrained by the SRAM as in the TM3A, we choose commercial DDR SDRAM, which operates in excess of 266 MHz [53], well above the system frequencies listed above, ensuring that memory will not be the bottleneck in the system.

Note that the calculator and scoring units constitute the critical subsection of the design. From Table 4-11, we see that a peptide query can be located and its coding regions ranked within 1.63 seconds, slightly over the 1 second requirement presented in Section 2.2.3. To meet this speed requirement, consider the nature of the algorithm. The entire search and score process is highly parallelizable. By simply partitioning the genome into subsections and instantiating multiple copies of the hardware, the design can operate on each section simultaneously. Thus with two copies of the hardware, the entire search and score can be completed in $\frac{1.63}{2} = 0.82$ seconds to meet the requirements of Section 2.2.3.

The data in Table 4-10 and Table 4-11 show that a hardware system capable of searching the genome at very high speeds can be designed using current FPGA technology in combination with existing commercial RAM. Capitalizing on the intrinsically parallel nature of the algorithm, hardware units at various levels of performance can be designed to meet a user's cost and performance requirements. However, the parallel nature of this algorithm lends itself to software implementation as easily as hardware. In the following section we examine a software implementation of a similar algorithm and consider the resources required to implement it. This information will then be used to determine the most cost effective platform for this design.

4.3.4. Software

The software speeds and resources described here are taken from the study in [1]. The scoring algorithm in the study is MASCOT, which is based on MOWSE. The operations in [1] were performed on a 600 MHZ Pentium III PC, resulting in search and score times of 3.5 minutes (210 s) per query. To scale these values to current processor speeds, while presenting the software in the most favorable light, we assume a linear increase in speed if the algorithm is implemented on a modern processor. Based on this assumption, we state that the software can complete the task in 52.5 seconds on a 2.4 GHz processor. This claim implies that the process will experience a speedup factor of 4 when run on a processor that is four times as fast. Such a scaling in speed is unlikely, as memory bandwidth does not scale with processor speed, but this optimistic assumption presents the ideal performance of this algorithm in software. Regardless, a single modern processor still cannot achieve the 1-second search and score time defined in Section 2.2.3.

As with the hardware, the algorithm is highly parallelizable and indeed MASCOT is a threaded program, designed to be implemented in a multiprocessor environment [39]. To meet the 1-second operation time, we assume that processing time scales perfectly with cluster size, i.e. to halve the time, the cluster size must be doubled. Table 4-12 shows the

number of processors required to achieve performance that is comparable to the hardware.

Number of Processors	Processing time (s)
1	52.5
32	1.6
64	0.8

Table 4-12: Processing Time for Computing Cluster

Table 4-12 shows that a cluster of 64 processors can achieve the performance delivered by two copies of the hardware as described in the previous section. Thus both systems are capable of offering the same level of performance.

In the next section, the system is parameterized based on the resources required to achieve a user-defined level of performance. The required resources allow us to estimate and compare the costs of the hardware and software systems to evaluate the most cost-effective solution.

4.3.5. System Cost and Resource Estimation

The goal of this thesis was to design cost-effective hardware to accelerate the sequencing process. The previous sections have presented hardware and software approaches to solving this problem. In this section, we evaluate the cost of these systems at various levels of performance to determine the most practical platform for final implementation. In addition, to providing the cost of the hardware system described in Chapter 3, we detail the cost of a hardware system designed only to search the genome and locate coding regions for a peptide. Numerous biological algorithms require the ability to search through the genome without the scoring function detailed in Section 3.5. We hope that readers will find the cost to performance comparisons of the standalone search engine useful even outside of the scope of this work.

4.3.5.1. Cost of Software Platform

The processors described here are ASL Lancelot C 1851 blades [54]. Each blade is a 2.4 GHz Xeon Dual processor with 2 gigabytes of RAM and an 80-gigabyte hard disk. Table 4-13 shows the price of the system for various scan times.

Number of Processors	Scan time (s)	Number of Blades	Acquisition Cost (USD)
1	52.5	1	$1,962
32	1.6	16	$31,392
64	0.8	32	$62,784
512	0.1	256	$502,272

Table 4-13: Prices of Computer Clusters for Varying Performance

Since the blades are dual processor boards, 32 blades can implement the 64-processor search system described in the previous section at a cost of approximately $ 62,700.
In Section 3.3 the concept of high-speed genome searching outside the context of this work is mentioned. A plethora of biological applications require the ability to search through the genome at high speed. Table 4-13 shows the cost of computer clusters capable of searching the human genome at varying speeds from a minute to a hundred milliseconds.

Table 4-13 only lists the price of the system or its *acquisition cost*, but computer clusters usually suffer a system administration cost (SAC). The SAC includes cost of installation, maintenance and upgrading by a professional administrator. The work in [64] considers several different clusters with 23 nodes (processors) and identifies an average annual SAC that is equivalent to the acquisition cost. Table 4-14 shows the total operational cost of a computer cluster over a four-year lifetime.

Number of Processors	Scan time (s)	Acquisition Cost (USD)	Total Cost Acquisition + Administration (USD)
1	52.5	$1,962	$9,810
32	1.6	$31,392	$156,960
64	0.8	$62,784	$313,920
512	0.1	$502,272	$2,511,360

Table 4-14: Total Cost of Computer Cluster Over Four-year Lifetime

In addition to the SAC described above, the power consumption of a computer cluster may be quite significant. The average power consumption of a 2.4 GHz Pentium is approximately 57.8 W [66]. Table 4-15 lists the power consumption of clusters of various sizes.

Number of Processors	Scan time (s)	Power Consumed (W)
1	52.5	58
32	1.6	1,850
64	0.8	3,700
512	0.1	29,594

Table 4-15: Power Consumption of Computer Clusters

It is clear from Table 4-15 that a large cluster will draw a significant amount of power during its operation. In high throughput MS experiments, where the cluster is constantly in use, the cost of power will not be negligible.

Having identified the cost of using software to search and score through the genome, the next section presents the cost of hardware systems capable of offering similar levels of performance.

4.3.5.2. Cost of Hardware Platform for Full System

The design presented in Chapter 3 was implemented using the parameter values from Table 4-1 on a set of FPGAs to identify the optimal hardware system. Section 4.3.3 identifies the Stratix S20 as the smallest FPGA upon which a search engine can be implemented and further identifies the Stratix S40 as the smallest FPGA upon which the calculator and scoring units can be implemented. Therefore the most cost effective implementation for the entire system is achieved on a set of 4 FPGAs: one S20 for the search engine and three S40 FPGAs for the 6 frames of calculation and corresponding scoring units. Such a system requires sufficient RAM and a suitable PCB to act as a motherboard. We therefore make the following design decisions:

- Each set of 4 FPGAs requires a 10.5" x 14" – 14 layer PCB as its motherboard.
- Every search engine in the system has 2 GB of memory.

The second decision reflects practical issues in acquiring memory. We can use multiple hardware units to search subsections of the genome in parallel. Clearly a subsection of the genome will not require the storage space of the full genome. However, small memory modules are difficult to acquire commercially, and large memory modules can be purchased relatively inexpensively [53]. Thus each hardware unit contains a full 2 GB of memory even though this is unnecessary for the design. The costs of such a hardware system at various speeds are presented in Table 4-16. Note that a 50% margin has been added to the total cost as an estimate for the final purchase price.

Scan Time (s)	Number of S20 FPGAs	Number of S40 FPGAs	Cost of RAM (USD)	Cost of PCB (USD)	Cost of FPGAs (USD)	Purchase Price (USD)
1.6	1	3	$344	$131	$6,950	$11,137
0.8	2	6	$689	$262	$13,900	$25,426
0.1	16	48	$5,512	$2,100	$111,200	$225,469

Table 4-16: Price of Full System in Hardware

Table 4-16 lists the cost and resources required to implement the full system at different levels of parallelism corresponding to different speeds. Note that the prices for the FPGA [50] and PCB [51] are based on volume pricing for 500 units. A hardware system that meets the requirements defined in Chapter 2, i.e. a system that takes under 1 second to

search and score using a single peptide query, can be implemented for less than half of the acquisition cost of an equivalent software system.

For a fair comparison to software, we assume that the annual system administration costs (SAC) in hardware are equal to the acquisition cost. Table 4-17 shows the total cost of operating the hardware systems in Table 4-16. Recall that an individual hardware unit consists of 1 Stratix S20 FPGA and 3 Stratix S40 FPGAs.

Number of Hardware Units	Scan Time (s)	Total Cost Acquisition + Administration (USD)
1	1.6	$55,685
2	0.8	$127,130
16	0.1	$1,127,345

Table 4-17: Total Cost of Hardware-Clusters Over Four-year Lifetime

Table 4-17 shows that the total cost of the custom hardware implementation is less than half that of a software platform of equivalent performance.

The Stratix Power Calculator [65] is a tool that allows the designer to estimate the total power consumed by a design on a Stratix FPGA. Using the resource values from Table 4-10 and Table 4-11 the power consumed by the full hardware system is estimated as 7.6 W (1 W for a Stratix S20 containing search engine and 2.2 W for each of the three Stratix S40 containing the calculator and scoring units). The majority of the power is dissipated in the IO pins.

Number of Hardware Units	Scan Time (s)	Power Consumed (W)
1	1.6	7.6
2	0.8	15.2
16	0.1	121.6

Table 4-18: Power Consumption of Hardware-Clusters

All the FPGAs are running at 75 MHz and a 25% toggle rate is assumed for every flip flop and memory bit in the design. The results in Table 4-18 show that our custom hardware implementation consumes 200 times less power than general-purpose processor cluster. This reduction in total power consumption translates into a significantly lower operational cost over the lifetime of the cluster.

The speed of the search engine operating as an individual unit unencumbered by the calculator and scoring units may prove to be of greater interest to the reader. In the following section, we consider the cost of implementing the search engine as a standalone hardware unit.

4.3.5.3. Cost of Hardware Platform for Standalone Search Engine

The search engine operating as an isolated unit does not require the same number of FPGAs or a PCB of the same complexity as the full system described in the previous section. Therefore the following design decisions are made for the standalone search engine:

- A 10"x 4" - 8 layer PCB is required as the motherboard and can contain two FPGAs
- Every search engine in the system has 2 GB of memory

Using these constraints, we find that the Stratix S20 is the most cost effective FPGA upon which to implement the search engine as shown in Table 4-19

Search Time (s)	Number of Stratix S20 FPGAs	Cost of FPGAs (USD)	Cost of RAM (USD)	Cost of PCB (USD)	Purchase Price (USD)
0.8	1	$650	$344	$25.5	$1,530
0.4	2	$1,300	$689	$25.5	$3,021
0.2	4	$2,600	$1,378	$51	$6,044
0.1	8	$5,200	$2,756	$102	$12,087

Table 4-19: Cost of Standalone Search-Engine in Hardware

Note once again that a 50% margin is added to the total cost as an estimate for the final purchase price. As before, we estimate the total cost of the system over a four-year operational lifetime. Note that each hardware unit in Table 4-20 can contain two FPGAs.

Number of Hardware Units	Scan Time (s)	Total Cost Acquisition + Administration (USD)
1	0.8	$7,650
4	0.1	$60,435

Table 4-20: Total Cost of Hardware Search Engine Over Four-year Lifetime

Table 4-20 shows the total cost of the hardware based search engine assuming that the annual administration cost is equal to the purchase price. The hardware searching system costs approximately 40 times less than a software platform of comparable performance.

Intuitively, the power consumed by the hardware search engine is significantly lower than either the fully hardware system or the processor cluster.

Number of Hardware Units	Search Time (s)	Power Consumed (W)
1	0.8	1.8
4	0.1	7.2

Table 4-21: Power Consumption of Hardware Search Engine

Table 4-21 lists the power consumption of the hardware search engine for various search times. These power estimates were obtained from the Stratix Power Calculator assuming

a clock speed of 162 MHz and a 25% toggle rate for every flip flop an memory bit in the design. The power savings are even more significant in this case, with the hardware providing over 2000 times the power to performance ratio of a software cluster. These results indicate that there are significant advantages to performing genomic searches in hardware.

In the following section, the hardware and software cost are directly contrasted to ascertain the most economical solution for a desired level of performance.

4.3.5.4. Cost Comparison

This section summarizes the costs of the system, by dividing the solution into two broad categories, namely, low-performance and high performance. Here, low performance indicates search times in excess of a minute, which may be acceptable in many applications. However, as detailed in Section 2.2.3, our design must be able to identify and rank the coding locations for a peptide query in less than 1 second, thus demanding a high performance system. The ratio of software to hardware costs for different system speeds is given in Table 4-22. Note once again that the costs are based on a four-year operation lifetime for the both the software and hardware platforms.

Time (s)	Cost of Software Platform	Cost of Hardware Search and Score Platform	Cost of Hardware Search Engine	Software /Hardware Cost Ratio (Search + Score)	Software /Hardware Cost Ratio (Search)
60	$750	$10335	$7650	0.07	0.1
0.8	$313,920	$127135	$7650	2.5	41
0.1	$2,511,360	$1127350	$60435	2.2	41

Table 4-22: Ratio of Software to Hardware Cost for Different Processing Speeds

For slower searches of the genome, i.e. search times in excess of 1 minute, software is a far more cost effective solution than hardware. The software cost is based the quoted price on a 2.4 GHz Dell Dimension Desktop [55]. The cost of its hardware counterpart is

based on the cost of a single hardware board capable of implementing the full system, as described in Table 4-16. It is possible to design a hardware system using cheaper, slower FPGAs but if real time performance is not required, a PC is likely a far more flexible solution with a greater capacity for reuse in other applications. Moreover, a PC at half the price of the hardware system it is clearly a better choice. Therefore, at the low end of the performance spectrum, software is more practical vehicle for the searching and scoring process.

However, using the current cost and performance of the system as a measure of quality, hardware is clearly a better solution for a laboratory seeking the ability to search through genomes in real-time. At the high-performance end of the cost spectrum, hardware is more than three times as economical for equivalent level of performance as seen in Table 4-22. For a standalone search engine, hardware is more than 40 times as economical, making it an ideal platform for genomic studies.

The costs in Table 4-22 do not take power consumption into account. Section 4.3.5.2 showed that the performance to power ratio is far more favourable for hardware, than a cluster of general-purpose processors. Over the operational lifetime of the hardware platform, the power savings will likely translate to a substantial reduction in operational cost when compared with software.

In the following section, we present a means of estimating these costs based on the resources required to attain a given level of performance in hardware. Using these methods, designers in the future can estimate the cost of a hardware system using the technology available to them at the time.

4.3.5.5. Framework for estimating system cost

Table 4-19 and Table 4-16 list the current costs of designing such a hardware system. The key resources that determine this cost are: *the FPGAs*, *the RAM* and *the PCB*. The FPGA [50], RAM [53] and PCB [51] costs are obtained from current vendor and

manufacturer quotes. System designers in the future will likely have access to FPGAs with far more resources for which prices cannot be accurately predicted. As such we define the resources required for a given level of performance. Knowledge of the required resources will allow designers in the future to choose the most practical platform upon which to build their hardware. This section provides a framework to estimate the resources required to implement a hardware system at a given level of performance.

In general, to design a system that meets a specific level of performance, the required resources can be estimated by the three elements listed above: FPGAs, RAM and PCBs. The total cost of the hardware is then given by the number of FPGAs (defined as *NUM_FPGAs*), the total amount of RAM (*TOTAL_EXT_RAM*) and the number of PCBs (*NUM_PCBs*). Note that this cost is a function of the desired level of performance specified by the designer. The performance is specified by the time required to process an entire genome, thus the two variables that determine the hardware resources for the system are *size_of_genome* (in GB) and *search_time* (in seconds). Thus we define the *performance factor* $\mathbf{P} = \left\lceil \frac{size\_of\_genome}{search\_time} \right\rceil$. The designer can use the desired value of **P** to determine the cost of the system in the future. This cost is given by:

COST (**P**) = *(NUM_FPGAs*(**P**) *x FPGA_PRICE) + (TOTAL_EXT_RAM*(**P**) *x RAM_PRICE) + (NUM_PCBs*(**P**) *x PCB_PRICE)*

The number of FPGAs that contain a given amount of resources can only be evaluated for current technology. Any speculation on device capabilities in the future would likely be inaccurate. Therefore we classify an FPGA in terms of its key components, namely the LUTs, flip-flops and memory and user IO pins. Given these parameters, designers will be able to determine the most cost-effective FPGA or set of FPGAs at their time.

We define the total number of LUTs and flip-flops in a given FPGA as *FPGA_LUTs_FFs*, and the total on-chip RAM as *FPGA_RAM*, and the number of user IO pins as *FPGA_IO_PINS*. Using these parameters, a designer can determine the optimal FPGA for the device.

Again, the following results are divided into two units: one to provide resource estimates for the full search and score system and the other for the search engine as an independent unit.

Resources Required for Full Search and Score System:

The values for each of these parameters depend on the performance factor P described above. From Table 4-9, we see that a full implementation of the device requires 12,299 LUTs and flip-flops for the search engine and $3\times(44338)=133041$ LUTs and flip-flops for the calculator and scoring functions. Thus, with $FPGA\_LUTs\_FFs = 145313$, a 1 GB genome can be processed in 1.6 seconds. To generalize this we state that:

$FPGA\_LUTs\_FFs = 232500\times \mathbf{P}$

Correspondingly, the device in Table 4-10 requires 7938 on-chip memory bits for the search engine and $3\times(205244)=615732$ bits for the 3 calculators and the associated scoring functions. Thus 623670 on-chip memory bits are required to process the 1 GB genome in 1.6 seconds. Once again we generalize this to:

$FPGA\_RAM = 997872\times\mathbf{P}$

The design requires a total of 1014 pins to process the genome as described. This enables us to define:

$FPGA\_IO\_PINS = 1623\times\mathbf{P}$

Note that these assumptions are pessimistic, as we do not account for improvements in process technology, which will undoubtedly result in faster FPGAs.

Using these three parameters, designers in the future can determine the value of NUM_FPGAs based on the most cost effective devices available at the time. To determine the optimal number of FPGAs, a designer must compare the cost and resources of a few large FPGAs with those on many smaller FPGAs. This information can be easily obtained from datasheets and vendor price lists for the chosen device. The most

favourable solution implements the required resources at the minimum cost, thus defining the ideal value for NUM_FPGAs.

The next significant parameter is the amount of external RAM required. A single copy of a 1 GB genome can be searched in 1.6 seconds. As the level of parallelism increases and additional copies of the device are used to increase the system speed, multiple copies of the genome must be processed. This is generalized as:

$$TOTAL\_EXT\_RAM = \frac{1}{0.625} \times \mathbf{P}$$

Using the design presented in this work as a reference, we estimate that four FPGAs and the RAM can be connected on a single PCB without prohibitive complexity. This leads to the formula:

$$NUM\_PCBs = \frac{NUM\_FPGAs}{4}$$

The value of NUM_PCBs clearly hinges on an assumption of 4 FPGAs per board as defined in our design. The trend towards larger FPGAs implies that our design will eventually be able to fit on a single FPGA. When such technology becomes available, the size of the PCB can be scaled down correspondingly.

Note that each of these formulas is based on the design of the full search and score algorithm that operates on a 1 GB genome in 1.6 seconds. The formulas are intended to provide a sense of the required resources as the speed, and correspondingly the level of parallelism, within the system increase. If the required search time is less than 1.6 seconds, or the size of the genome is significantly less than 1 GB, the approximations provided here will be of little value, as the formulas encapsulate the trend in resource requirements for increasing levels of parallelism.

Resources Required for Standalone Search Engine:

As in the previous section, we distinguish the search engine from the full design as it may be of interest to the reader. For the standalone search engine, we define the resource requirements as a function of *search_time* and *size_of_genome* to allow the user to estimate system costs in the future. The formulas given below are based on the data in Table 4-10 and Table 4-11, and assume a standalone search engine can search a 1 GB genome in 0.8 seconds.

$$FPGA\_LUTs\_FFs = 9839 \times \mathbf{P}$$

$$FPGA\_RAM = 6350 \times \mathbf{P}$$

$$FPGA\_IO\_PINS = 313 \times \mathbf{P}$$

Once again, the actual value for NUM_FPGAs hinges on the technology available to the designer and can be determined based on the cost of devices in the future.

$$\text{TOTAL\_EXT\_RAM} = \frac{1}{1.25} \times \mathbf{P}$$

As in Section 4.3.5.3, we constrain the design to two FPGAs per PCB resulting in the formula:

$$\text{NUM\_PCBs} = \frac{NUM\_FPGAs}{2}$$

The caveats from the first set of formulas apply equally well to the approximations above. The formulas convey the trends in resource usage based on the search of a 1 GB genome in 0.8 seconds and any attempt to use them to approximate resource utilization for a system with lower performance will be fraught with error.

The formulas above model the resources required for various levels of parallelization, which in turn correspond to different levels of performance. As stated the performance is dictated by the time taken to process a genome of a given size. Using the resources estimation models above, designers in the future can estimate the resources required to implement either the full search and score system described in Chapter 3 or the search engine as an independent unit. These resources can then be used to determine the cost of the optimal solution based on the prices of devices available at the time.

4.4. Summary

From the above results, we see that with a sufficiently high quality sequence, a scoring function will not even be necessary. If the sequence is sufficiently large, it can act as a "fingerprint" by uniquely identifying its true coding region. In such an event, only the search engine hardware is required. Recall that the standalone search engine is considerably faster and cheaper than the full system and as MS technology improves, this will likely be a more cost-effective solution.

If, however, the protein sample is contaminated, it may be hard to obtain a large peptide query. In these cases, multiple hits need to be resolved to identify the true coding gene. It is clear from our results that the scoring function is limited to resolving differences between proteins and has difficulty identifying the false positives in the genome. This was expected due to the volume of random information contained in the genome and the fact that MOWSE was designed to target protein databases. Observations from prior work [1] also suggest that the genome should only be used as a search database for novel proteins due to the number of false positive matches that are found in unannotated genomic sequence [63].

Despite the difficulty of assigning accurate scores, we see that one can easily isolate the true coding region by using additional queries.

The approach presented here accelerates the sequencing process for novel proteins. Using either high or low quality peptides as a query to the database, the device is capable of rapidly locating the peptide's true coding location in the genome. Furthermore, it delivers

this performance at a significantly lower cost than a software implementation of equivalent functionality.

Chapter 5. Conclusions & Future Work

5.1. Thesis Summary

In this work we have studied the design of a hardware system designed to accelerate MS/MS based de-novo protein sequencing. The objective of this work has been to study the feasibility of a custom hardware implementation of a protein-sequencing algorithm. We believe that this is the first published hardware implementation of the sequencing approach described here. The results of this work show that hardware implementations of certain key features of the system provide significant improvements in speed at a lower cost than equivalently functional software.

5.2. Thesis Contributions

This thesis provides the following significant contributions:

1. The design of an FPGA-based hardware system capable of locating and ranking the coding regions of a peptide in an organism's genome. The hardware is between 3 and 60 times as cost-effective as an equivalent software platform.
2. The design of a fast comparison scheme based on data associativity (as described in Section 3.5.3.1). This hardware can be used to identify similar values in a single clock cycle.
3. A framework for estimating the cost of the hardware design in the future. The models presented in Section 4.3.5.5 allow designers to estimate the cost of the system at various levels of performance.

5.3. Future Work

The first and most practical extension to this work is to interface the system with a real mass spectrometer. Our prototype was tested using data from real MS experiments, but these data were used offline. It would be instructional to integrate the system with different mass spectrometers to see what other improvements could be made to the sequencing process. Also, as described in Appendix A, there is additional information in the MS output (for example intensity) that is often used for noise rejection. We have only used the masses from the MS output but it is likely that incorporating the intensity information into the scoring system will be beneficial.

In addition, a study using protein data from human samples would allow us to truly evaluate the benefits of this system as a tool for medical researchers. While the yeast genome is a good foundation on which to begin a study, further insight into the complexities of human biological systems can only be achieved by studying the human genome.

The scoring algorithm used in this work needs to be tuned to fit the dataset. We chose the MOWSE [9] algorithm, as it seemed best suited to our needs. However, there is a plethora of scoring algorithms, each of which must be considered before the best ranking scheme can be determined.

Another interesting area is exposed by the complexities of biological systems. In 2.3.2 we described the process of alternative splicing and mentioned that 98% of splice variants are canonical – i.e. they follow a recognized pattern of rules defining their start and end points. The current implementation of the design does not deal with the splice variants in hardware. If the splice variants and their masses could be calculated in hardware, they too could be compared to the PIS list to obtain a further degree of confirmation for the generated score.

Chapter 6. References

[1] Choudary J S., Blackstock W.P., Creasy D. M., Cottrell J.S., *"Interrogating the human genome using uninterpreted mass spectrometry data*",Proteomics, 2001, May;1(5):651-67.

[2] Lesk, Arthur M., **Introduction to Bioinformatics** . Oxford press, NY, 2002, pp. 6-7

[3] Baxevais and Ouellette, **Bioinformatics**, Wiley Interscience,N, 2001, pp. 253-257

[4] European Molecular Biology Lab (EMBL), http://www.embl-heidelberg.de/

[5] Sanger, F., "*The free amino groups of insulin*", Biochem. J., 1945, **39**:507-515.

[6] J. Alex Taylor and Richard S. Johnson "*Implementation and Uses of Automated de Novo Peptide Sequencing by Tandem Mass Spectrometry*", Analytical Chemistry, 2001, V 73, pp 2594-2604

[7] Brenner S.E. "*A tour of structural genomics*", Nature Reviews – Genetics, 2001, 2: pp 801-9.

[8] Eng J.K., McCormack, A.L., and Yates, J.R., III, "*An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database*". J. Am. Soc. Mass Spectrom., 1994, 5(11) pp. 976-89

[9] Pappin, D.J.C., Hojrup, P. and Bleasby, A.J., "*Rapid identification of proteins by peptide mass fingerprinting*". Curr Biol, 1993, 3(6), pp 327-32

[10] McLuckey S.A. and Wells J.M., "*Mass Analysis at the Advent of the 21st Century*", Chem Rev. 101 (2), 2001, pp. 571-606

[11] Washington University, Dept. of Chemistry. "*Instrumentation and Ionization Methods Tutorial"* http://wunmr.wustl.edu/~msf/ionmethd.html

[12] Richard Caprioli and Marc Sutter, "*Mass Spectrometry"*, http://ms.mc.vanderbilt.edu/tutorials/ms/ms.htm

[13] *TM3 Documentation*, University of Toronto, Dept. of ECE. http://www.eecg.toronto.edu/~tm3/

[14] *TM3 Ports Package Documentation,* University of Toronto, Dept. of ECE. http://www.eecg.toronto.edu/~tm3/ports.ps

[15] Edman, P. *"On the mechanism of the phenyl isothiocyanate degradation of peptides". Acta Chem. Scand.* (1956) 10, 761–768

[16] Burley S.K., Almo S.C., Bonanno J.B., Capel M., Chance M. R., Gaasterland T., Lin D., Šali A, Studier F.W. & Swaminathan S, "*Structural genomics: beyond the Human Genome Project"*, Nature Genetics, 1999, 23, pp. 151 - 157

[17] Hoang D.T. , Lopresti D. P., *"FPL Implementation of Systolic Sequence Alignment"* FPL 1992: 183-191

[18] Lopresti D. P., *"Rapid implementation of a genetic sequence comparator using FPGAs"* Adv. Res. VLSI, pp 139-152, 1991.

[19] DeCypher http://www.timelogic.com/technology.html

[20] BioXL http://www.cgen.com/products/bioxl.htm

[21] Biemann K., Cone C., Webster B.R., Arsenault G.P. *"Determination of the amino acid sequence in oligopeptides by computer interpretation of their high-resolution mass spectra*", J. Am. Chem. Soc., 1966, 88(23), p.5598-606

[22] Lewis D., Betz V., Jefferson D., Lee A., Lane C., Leventis P., Marquardt S., McClintock C., Pedersen B., Powell G., Reddy S., Wysocki C., Cliff R., and Rose J., *"The Stratix Routing and Logic Architecture"* FPGA '03, pp. 15-20, February 2003.

[23] Altschul, S.F., Gish, W., Miller, W., Myers, E.W. & Lipman, D.J. "*Basic local alignment search tool.*",J. Mol. Biol., 215 pp. 403-410, 1990

[24] DNA images, http://dna.com

[25] Sinclair B., "Software Solutions to Proteomics Problems", The Scientist, 2001 Oct, 15[20]:26

[26] Kumar A, Harrison PM, Cheung KH, Lan N, Echols N, Bertone P, Miller P, Gerstein MB, Snyder M. "*An integrated approach for finding overlooked genes in yeast*.", Nat Biotechnol 2002 Jan;20(1):58-63

[27] ftp://genome-ftp.stanford.edu/pub/yeast/data_download/sequence/ genomic_sequence/chromosomes/fasta/

[28] Partial Saccharomyces Chromosome IV map http://db.yeastgenome.org/ cgi-bin/SGD/ORFMAP/ORFmap?seq=YDL229W

[29] Partial Saccharomyces Chromosome XIV map http://db.yeastgenome.org/ cgi-bin/ SGD/ORFMAP/ORFmap?seq=YNL209W

[30] Partial Saccharomyces Chromosome XV map http://db.yeastgenome.org/ cgi-bin/SGD/ORFMAP/ORFmap?seq=YOR370C

[31] BLAST (2 sequence) http://www.ncbi.nlm.nih.gov/blast/bl2seq/bl2.html

[32] Sherman Fred, *An Introduction to the Genetics and Molecular Biology of the Yeast* Saccharomyces cerevisiae, http://dbb.urmc.rochester.edu/labs/Sherman_f/yeast/index.html, Chapters 1-5

[33] Rubin, Gerald M. "*The draft sequences: Comparing species*", Nature, 2001, 409, pp.820-821

[34] Stanchi F, Bertocco E, Toppo S, Dioguardi R, Simionati B, Cannata N, Zimbello R, Lanfranchi G, Valle G. "*Characterization of 16 novel human genes showing high similarity to yeast sequences*", Yeast. 2001 Jan 15;18(1), pp. 69-80.

[35] Bostanci Adam, "*Sequencing Human Genomes*", The Mapping Cultures of 20th Century Genetics, August 2003

[36] Steen H., Andersen J., Küster B., Podtelejnikov A., Rappsilber J., Henrik M., Mann M., "*Increasing the Throughput of Protein Identification Using Nanoelectrospray QqTOF Mass Spectrometry*", ASMS, 1999.

[37] S. cerevisiae - Repeat Sequence information http://www.yeastgenome.org/sequence_done.shtml

[38] H. Sapiens - Repeat Sequence information http://www.neuro.wustl.edu/neuromuscular/mother/dnarep.htm

[39] MASCOT http://www.matrixscience.com/cgi/index.pl?page=/search_form_select.html

[40] University of Wisconsin BioWeb System: *Sequence Analysis* http://bioweb.uwlax.edu/GenWeb/Molecular/Seq_Anal/Translation/translation.html

[41] A Guide to Molecular Sequence Analysis : *Open Reading Frames* http://www.sequenceanalysis.com/model/orf.html

[42] Medical Dictionary Database http://www.books.md/T/dic/trypsin.php

[43] Aebersold R, Mann M., "*Mass spectrometry-based proteomics*", Nature. 2003 Mar 13;422(6928):198-207.

[44] Net Gene Predictor http://www.cbs.dtu.dk/services/NetGene2/

[45] GLIMMER at TIGR http://www.tigr.org/~salzberg/glimmer.html

[46] Houle, John L., "*Database Mining in the Human Genome Initiative*", Whitepaper, Biodatabases, Amita Corporation, July 2000.

[47] 2D GEL ELECTROPHORESIS FOR PROTEOMICS TUTORIAL http://www.aber.ac.uk/parasitology/Proteome/Tut_2D.html

[48] Editorial, "*A Cast of Thousands*", Nature Biotechnology, March 2003 21 (3) p 213.

[49] Analyst QS Tutorials for Hybrid Quadrupole-TOF Mass Spectrometer (Pulsar), (Chaper 5) *Independent Data Acquisition*, Sciex Corp.

[50] Altera Corporation, North American price list (volumes 100-499), Aug 2003

[51] Leontti J., Private Communication, Camtech II Circuits, Sep 2003

[52] Steve Schaer, Personal Communication

[53] Kingston Technology, http://www.kingston.com

[54] ASL Inc. http://www.aslab.com

[55] Dell Computers http://www.dell.com

[56] Xilinx Corporation http://www.xilinx.com

[57] Altera Corporation http://www.altera.com

[58] Venter et al. "*The Sequence of the Human Genome*", Science, Feb 2001 291:1304-1351.

[59] International Human Genome Sequencing Consortium, "*Initial Sequencing and Analysis of the Human Genome*", Nature, 2001, 409, pp. 860-921

[60] Yuen Ho, Albrecht Gruhler, Adrian Heilbut, Gary D. Bader, Lynda Moore, Sally-Lin Adams, Anna Millar, Paul Taylor, Keiryn Bennett, Kelly Boutilier, Lingyun Yang, Cheryl Wolting, Ian Donaldson, Soren Schandorff, Juanita Shewnarane, Mai Vo, Joanne Taggart, Marilyn Goudreault, Brenda Muskat, Cris Alfarano, Danielle Dewar, Zhen Lin, Katerina Michalickova, Andrew R. Willems, Holly Sassi, Peter A. Nielsen, Karina J. Rasmussen, Jens R. Andersen, Lene E. Johansen, Lykke H. Hansen, Hans Jespersen, Alexandre Podtelejnikov, Eva Nielsen, Janne Crawford, Vibeke Poulsen, Birgitte D. Sorensen, Jesper

Matthiesen, Ronald C. Hendrickson, Frank Gleeson, Tony Pawson, Michael F. Moran, Daniel Durocher, Matthias Mann, Christopher W. V. Hogue, Daniel Figeys & Mike Tyers, "*Systematic identification of protein complexes in Saccharomyces cerevisiae by mass spectrometry*", Nature 2002 Jan 10;415(6868): 180-183

[61] Hastings, L.M. and Krainer, A. R., "*Pre-mRNA splicing in the new Millenium*", Current Opinion in Cell Biology, 2001, 13:302-309

[62] Thanaraj, T. A., and Clark F., "*Human GC-AG alternative intron isoforms with weak donor sites show enhanced consensus at acceptor exon positions*", Nucleic Acids Research, 2001, 29, (12) pp. 2581-2593.

[63] Perkins D.N., Pappin D.J., Creasy DM, Cottrell J.S.,"*Probability-based protein identification by searching sequence databases using mass spectrometry data*", Electrophoresis. 1999 Dec;20(18):3551-67.

[64] Feng W., Warren M.S., and Weigle E., "*Honey, I shrunk the Beowulf!*", ICPP 2002, pp 141-149

[65] Stratix power calculator, http://www.altera.com/products/devices/stratix/ utilities/power_calculator/stratix_power_calc.xls

[66] Intel (R) Pentium 4 Processor datasheet, http://developer.intel.com/design/pentium4/datashts/298643.htm

[67] Sonar MS/MS, http://www.genomicsolutions.com/search/index.html

[68] ThermoFinnigan Sequest, http://www.genomicsolutions.com/search/index.html

[69] MDS Proteomics Pepsea, http://www.mdsproteomics.com

Appendix A. Mass Spectrometry for Protein Identification

Mass Spectrometry is a process in which an input sample is ionized and the ions thus generated are separated according to their mass to charge ratio. The general mass spectrometry flow used in protein identification is depicted in Figure A-1 below.

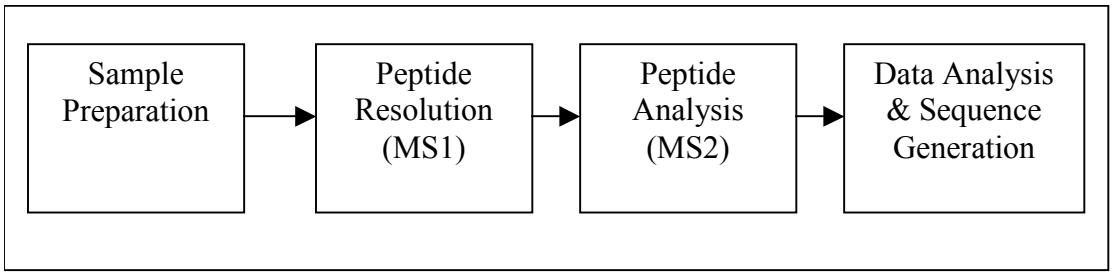


Figure A-1: Tandem Mass Spectrometry Flow

Once a biological sample is prepared for analysis, it is fed into a mass spectrometer (MS). Tandem mass spectrometry, as the name implies, involves two mass spectrometers (MS1 and MS2 shown in Figure A-1). The first MS provides a coarse analysis of the sample, and allows the user to select elements of the sample that can then be sent to the second MS for more detailed analysis.

Sample Preparation:

A protein sample being prepared for mass spectrometry should ideally only contain proteins of interest. However, current protein separation techniques cannot achieve this level of accuracy and most protein samples contain several contaminant proteins.

The purified samples are usually digested from their intact form into smaller peptides. Digestion is frequently performed using the enzyme trypsin, which is known as a specific enzyme for its property of cleaving proteins specifically after the Arginine (R) and Lysine (K) amino acids. However if a Proline (P) molecule follows the K or R amino acids, the bond will be stronger, preventing cleavage. Thus a protein is digested into tryptic peptides. An example is presented in Figure A-2 below.

MAVRAKPCOKLHNWF
Original protein in sample

MAVR AKPCOK LHNWF
After digestion – 3 smaller tryptic peptides (note cleavage after K and R but not KP)

Figure A-2: Protein Digestion

The intact protein is cut after every instance of a K or R amino acid except when followed by P. This process occurs to every protein in the sample, which is then fed into the mass spectrometer.

Peptide Resolution:

The next step of a conventional mass spectrometry experiment is Peptide Resolution. Here, the peptides in the sample are ionized and the mass to charge ratio of each ionized peptide is measured, and saved in a list known as the Parent Ion Scan (PIS). In addition to mass, the MS can also identify the concentration or intensity of a given substance in the sample. Individual parent ions (or ionized peptides) are selected by mass and moved to the next stage of analysis.

Peptide Analysis:

Each parent ion is then analyzed by a second mass spectrometer (MS2) to obtain its sequence. This is usually done through a technique known as Collision Induced Dissociation (CID). In CID, the parent ions are dissociated into their daughter fragments by collision with an inert gas. Consider the ion from the peptide "mavr" in the example above.

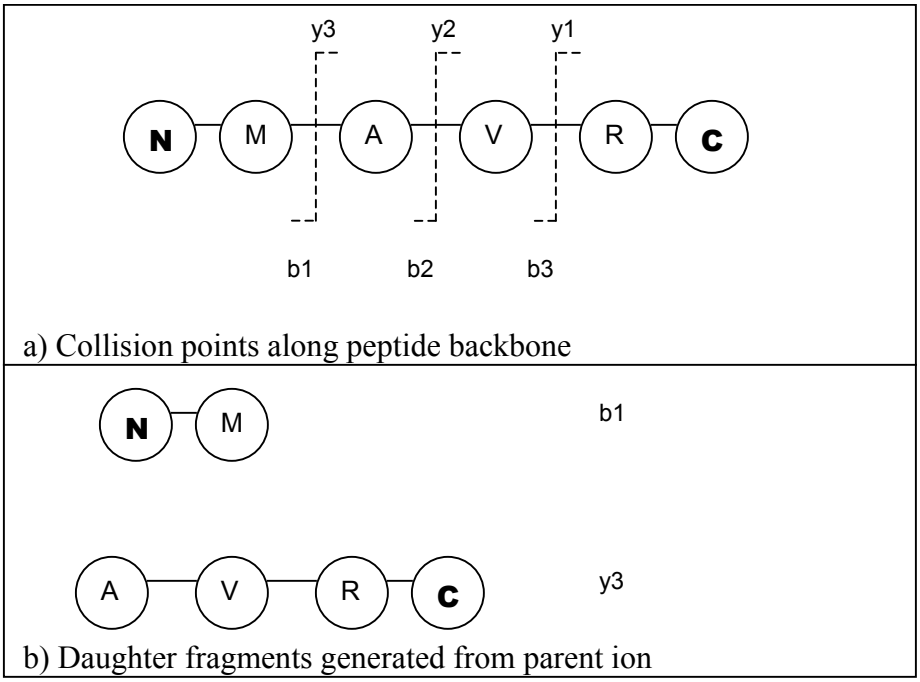


Figure A-3: Collision Induced Dissociation of Peptide

The molecules of the collision gas strike the peptide backbone i.e. the bonds that hold the amino acids together thus breaking the peptide into smaller fragments.

Note that the figure indicates two terminals present in every protein, the N and C terminal on either end of the peptide. Any daughter fragment induced by collision is either an N-terminal or C-terminal fragment, and referred to as a 'b-ion' or 'y-ion' respectively. These fragments are also identified by a subscript, which indicates the number of amino acids from their terminal they contain. For example, 'y_3' in the example above contains the first three amino acids starting at the C terminal of the peptide.

Ion Type	Set of all Daughter Ions	Mass of ion
b-ions	M	131
	MA	202
	MAV	301
y-ions (*read backwards*)	R	156
	RV	255
	RVA	326

Figure A-4: Daughter Ions of "MAVR"

The set of daughter ion fragments consists of all substrings of the parent peptide as shown in Figure A-4.

Data Analysis and Sequence Generation:

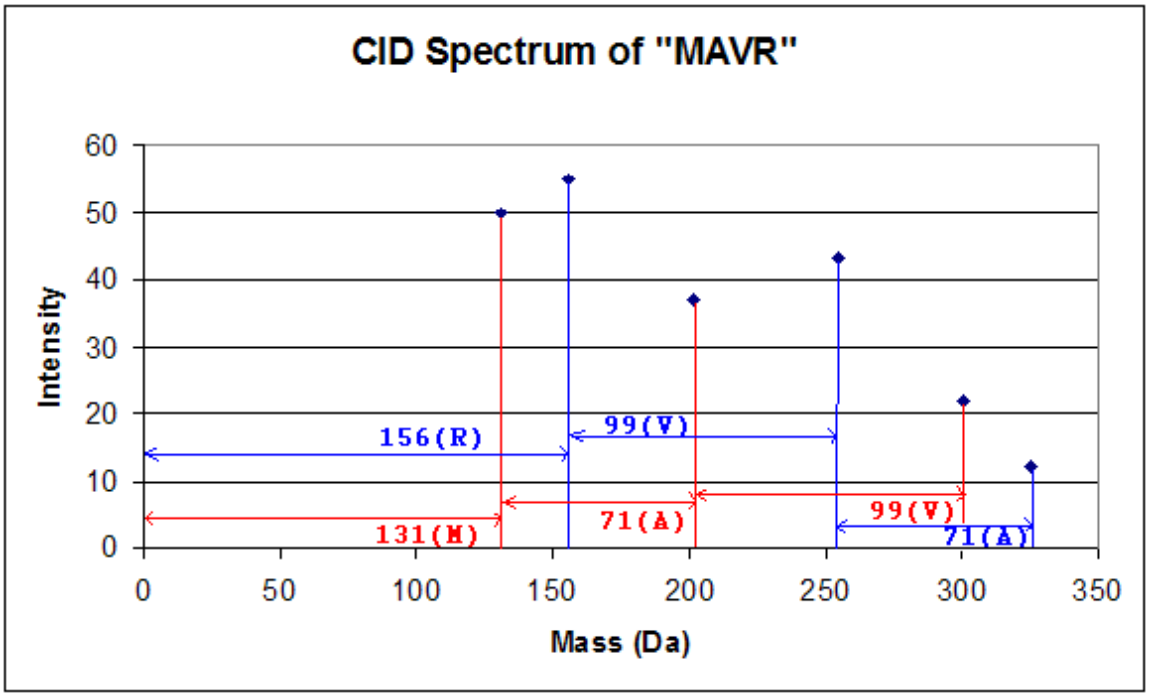


Figure A-5: Interpretation of Sequence from CID Sprectrum

From the CID spectrum in Figure A-5, each of the daughter ion fragments can be identified. The difference in mass between the peaks corresponds to the mass of a single amino acid and thus the sequence of individual fragments can be reconstructed.
There are various algorithms that then overlap the reconstructed fragment sequences and determine the full sequence of the original peptide.

In this manner each peptide from the original protein can be sequenced. Once the sequence of each tryptic peptides is known, a number of approaches can be used to deduce the sequence of the full protein. Several genetic algorithms have been used to match peptide sequences with those of existing proteins to look for common structures. Other heuristic approaches involve using physical chemistry to evaluate peptide configurations to determine a likely protein sequence.

Appendix B. VHDL Source Code

1. Search Engine Controller (control.vhd)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


entity control is
port (

                        tm3_clk_v0          : in std_logic;
                        tm3_sram_adsp       : out std_logic;
                        tm3_sram_data       : inout std_logic_vector(63 downto 0);
                        tm3_sram_addr       : out std_logic_vector(18 downto 0);
                        tm3_sram_we         : out std_logic_vector(7 downto 0);
                        tm3_sram_oe         : out std_logic_vector(1 downto 0);

                        main_reset          : in std_logic;
                        mem_scanned         : out std_logic;
                        match_address       : out std_logic_vector(18 downto 0);
                        codonin             : in std_logic_vector(269 downto 0);
                        tm3want             : out std_logic;
                        sunready            : in std_logic;
                        reset               : out std_logic;
                        mem_for_frame       : out std_logic_vector(63 downto 0);
                        freq_enable         : out std_logic;
                        calc_enable         : out std_logic;
                        score_sent          : in std_logic


);
end control;


architecture ctrl_behv of control is


        component genebuffer
                port (
                clock: IN std_logic;
                data: IN std_logic_VECTOR(62 downto 0);
                q: OUT std_logic_VECTOR(62 downto 0);
                load: IN std_logic);
        end component;



                component fullprot
                port (
                                fpClk               : in std_logic;
                                codonInp            : in std_logic_vector(0 to 269);
                                memwindow           : in std_logic_vector(0 to 149);
                                foundHit            : out std_logic
                                );
```

```
end component;

type ctrlStates is (rst,load1,load2,save,meminit1,meminit2,meminit3,hand1,hand2,reenter,mademateh,returnscore,memstate1,done);

signal memory_word : std_logic_vector(0 to 188);
signal dataword : std_logic_vector(63 downto 0);

signal query1 : std_logic_vector(0 to 269);
signal query2 : std_logic_vector(0 to 269);

signal stored_data : std_logic_vector(62 downto 0);
signal freq_window_out_buffer : std_logic_vector(62 downto 0);
signal mass_window_out_buffer : std_logic_vector(62 downto 0);
signal mem_to_frames : std_logic_vector(0 to 125);
signal freq_mem_to_frames : std_logic_vector(0 to 125);
signal mass_mem_to_frames : std_logic_vector(0 to 125);
signal load_gene_window : std_logic;
signal load_mass_window : std_logic;

signal calc_operation : std_logic_vector(8 downto 0);
signal freq_operation : std_logic_vector(8 downto 0);

signal testnet : std_logic;

signal currAddr : std_logic_vector(18 downto 0);
signal codon_ctr : std_logic_vector(0 to 0);
signal currState : ctrlStates;
signal nextState : ctrlStates;
signal mainhit : std_logic;
signal cmplhit : std_logic;
signal freq_enable_line : std_logic;
signal calc_enable_line : std_logic;

attribute syn_black_box : boolean;
attribute syn_black_box of genebuffer : component is true;

begin

reset <= main_reset;

freq_genewindow : genebuffer port map (
                    clock => tm3_clk_v0,
                    data => memory_word(0 to 62),
                    q => freq_window_out_buffer,
                    load => load_gene_window);

mass_genewindow : genebuffer port map (
```

```
                                        clock => tm3_clk_v0,
                                        data => freq_window_out_buffer,
                                        q => mass_window_out_buffer,
                                        load => load_gene_window);

proteinblock : fullprot port map (
                                        fpClk => tm3_clk_v0,
                                        codonInp => query1,
                                        memwindow => memory_word(0 to 149),
                                        foundhit => mainhit
                                        );

complmntblock : fullprot port map (
                                        fpClk => tm3_clk_v0,
                                        codonInp => query2,
                                        memwindow => memory_word(0 to 149),
                                        foundhit => cmplhit
                                        );


process(currState,currAddr,codon_ctr,mainhit,cmplhit,score_sent,sunready,main_reset,calc_operation )
                begin

                        calc_enable_line <= '0';
                        freq_enable_line <= '0';
                        load_gene_window <= '0';
                        tm3want <= '0';
                        tm3_sram_we <= "11111111";
                        tm3_sram_oe <= "01";
                        tm3_sram_adsp <= '1';
                        tm3_sram_addr <= currAddr;
                        tm3_sram_data <= (others => 'Z');
                        mem_scanned <= '0';
                        nextState <= rst;


                        case(currState) is

                                when rst =>
                                        nextState <= load1;

                                when load1 =>
                                        tm3want <= '1';
                                        tm3_sram_data <= dataword;

                                        if sunready = '1' then
                                                nextState <= load2;
                                        else
                                                nextState <= load1;
                                        end if;

                                when load2 =>
                                        tm3want <= '0';

                                        if sunready = '0' then
                                                nextState <= save;
                                        else
                                                nextState <= load2;
```

```
            end if;


when save =>
            tm3_sram_addr <= currAddr;
            tm3_sram_adsp <= '0';
            tm3_sram_oe <= "01";

if codon_ctr = "1" then
            nextState <= meminit1;
else
            nextState <= load1;
end if;




when meminit1 =>
            tm3_sram_addr <= currAddr;
            tm3_sram_adsp <= '0';
            tm3_sram_oe <= "01";

            nextState <= meminit2;



when meminit2 =>

            tm3_sram_addr <= currAddr;
            tm3_sram_adsp <= '0';

            tm3_sram_oe <= "01";
            nextState <= meminit3;




when meminit3 =>



            tm3_sram_addr <= currAddr;
            tm3_sram_adsp <= '0';

            tm3_sram_oe <= "01";

            if score_sent = '1' then
                        load_gene_window <= '1';
                        nextState <= memstate1;
            else
                        load_gene_window <= '0';
                        nextState <= meminit3;
            end if;

when memstate1 =>

            if calc_operation > "000000000" then
                        calc_enable_line     <= '1';
            else
                        calc_enable_line <= '0';
            end if;
```

```
                                        if freq_operation > "000000000" then
                                                freq_enable_line      <= '1';
                                        else
                                                freq_enable_line <= '0';
                                        end if;

                                        load_gene_window <= '1';


                                        tm3_sram_addr <= currAddr;
                                        tm3_sram_adsp <= '0';
                                        tm3_sram_oe <= "01";

                                        if (mainhit = '1') or (cmplhit = '1') or (currAddr >=
"1000000000000000000") then
                                                nextState <= mademateh;

                                        elsif (score_sent = '1')then
                                                nextState <= memstate1;
                                        elsif (score_sent = '0')then
                                                nextState <= returnScore;
                                        end if;



                                when mademateh =>

                                        if currAddr >= "1000000000000000000" then
                                                mem_scanned <= '1';
                                                nextState <= done;
                                        else
                                                nextState <= memstate1;
                                        end if;



                                when returnScore =>
                                        if score_sent = '1' then
                                                nextState <= mademateh;
                                        else
                                                nextState <= returnScore;
                                        end if;


                                when done =>
                                        nextState <= done;



                                when others =>


                        end case;


                end process;

                process(tm3_clk_v0,main_reset,codon_ctr,mainhit,cmplhit,calc_operation)
                begin
```

```
if main_reset = '1' then

        currState <= rst;


elsif rising_edge(tm3_clk_v0) then

        --if freq_operation > "000000000" and freq_operation < "000001111" then
        if freq_enable_line= '1' then
                mem_for_frame <= freq_mem_to_frames(0 to 63);
        --elsif calc_operation > "000000000" and calc_operation < "000001111" then
        elsif calc_enable_line = '1' then
                mem_for_frame <= mass_mem_to_frames(0 to 63);
        end if;

        freq_enable <= freq_enable_line;
        calc_enable <= calc_enable_line;

        currState <= nextState;

        case (currState) is

        when rst =>
                codon_ctr <= (others => '0');
                currAddr <= (others => '0');
                dataword <= (others => '0');
                calc_operation <= (others => '0');
                freq_operation <= (others => '0');

        when load1 =>

                dataword <= (others => '1');


        when load2 =>



        when save =>

                codon_ctr <= codon_ctr+1;

                if codon_ctr = "0" then
                        query1 <= codonin;
                else
                        query2 <= codonin;
                end if;


        when meminit1 =>


                memory_word(0 to 62)  <= tm3_sram_data(63 downto 1);
                currAddr <= "00000000000000000000";


        when meminit2 =>

                memory_word(63 to 125 )  <= tm3_sram_data(63 downto 1);
                currAddr <= "00000000000000000001";
```

```
                        when meminit3 =>

                                calc_operation <= (others => '0');
                                memory_word(126 to 188 )  <= tm3_sram_data(63 downto 1);
                                currAddr <= "00000000000000000010";



                        when memstate1 =>

                        if (mainhit = '1') or (cmplhit = '1') then
                                freq_operation <= "000000001";
                        elsif freq_operation > "000000000" and freq_operation < "0000011110" then
                                freq_operation <= freq_operation + 1;
                        elsif freq_operation = "0000011110" then
                                freq_operation <= (others => '0');
                                calc_operation <= "000000001";
                        end if;

                        if calc_operation > "000000000" and calc_operation < "0000011110" then
                                calc_operation <= calc_operation + 1;
                        elsif calc_operation = "0000011110" then
                                calc_operation <= (others => '0');
                        end if;


                                match_address <= currAddr;

                                --mem_to_frames(0 to 62) <= mem_to_frames(63 to 125);
                                --mem_to_frames(63 to 125) <= window_out_buffer;
                                freq_mem_to_frames(0 to 62) <= freq_mem_to_frames(63 to 125);
                                freq_mem_to_frames(63 to 125) <= freq_window_out_buffer;

                                mass_mem_to_frames(0 to 62) <= mass_mem_to_frames(63 to 125);
                                mass_mem_to_frames(63 to 125) <= mass_window_out_buffer;



                                for i in 0 to 125 loop
                                        memory_word(i) <= memory_word(i+63);
                                end loop;
                                memory_word(126 to 188 )  <= tm3_sram_data(63 downto 1);
                                currAddr <= currAddr + 1;

                        when done =>
                        when others=>

                        end case;

                end if;
                end process;
end ctrl_behv;
```

2. Peptide Comparison Unit (protein.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity protein is
generic (numAA:integer:=10);
	port (

		pClk			: in std_logic;
		potentialCodons1	: in std_logic_vector(0 to (9*numAA )-1);
		potentialCodons2	: in std_logic_vector(0 to (9*numAA )-1 );
		potentialCodons3	: in std_logic_vector(0 to (9*numAA )-1 );
		memWord			: in std_logic_vector( 0 to (9*numAA)-1 );
		onehit			: out std_logic
								);
end protein;

architecture prot_behv of protein is

	signal rowHit : std_logic_vector(numAA-1 downto 0);
	signal phitline : std_logic;
	signal hi :  std_logic;

	component amino
		port (
				aClk				: in std_logic;
				codonin1		: in std_logic_vector(0 to 8);
				codonin2		: in std_logic_vector(0 to 8);
				codonin3		: in std_logic_vector(0 to 8);
				memPort		: in std_logic_vector(0 to 8);
				hit				:	out std_logic
		);
	end component;

	component big_and
		Port (	clk : in std_logic;
			And_in : in std_logic_vector(11 downto 0);
			And_out : out std_logic);
	end component;




	begin

		hi <= '1';

		rowOfAminos :	for i in 0 to numAA-1 generate
			oneAA : amino port map (
				aClk => pClk,
				codonin1 => potentialCodons1( 9*i to (9*i+8) ),
				codonin2 => potentialCodons2( 9*i to (9*i+8) ),
				codonin3 => potentialCodons3( 9*i to (9*i+8) ),
				hit		=> rowHit(i),
				memPort => memWord(9*i to (9*i+8) )

									);
		end generate rowOfAminos;


		andaminos : big_and port map (
				clk => pClk,
				And_in(0) => rowHit(0),
				And_in(1) => rowHit(1),
```

```
                And_in(2) => rowHit(2),
                And_in(3) => rowHit(3),
                And_in(4) => rowHit(4),
                And_in(5) => rowHit(5),
                And_in(6) => rowHit(6),
                And_in(7) => rowHit(7),
                And_in(8) => rowHit(8),
                And_in(9) => rowHit(9),
                And_in(10) => hi,
                And_in(11) => hi,
                And_out => phitline
                                                );

        process(pClk)

        begin
                if rising_edge(pClk) then
                        onehit <= phitline;
                end if;
        end process;



end prot_behv;
```

3. Codon Unit (amino.vhd)

```
library ieee;
use ieee.std_logic_1164.all;
use work.all;


entity amino is
port (
      aClk                              : in std_logic;
      codonin1                  : in std_logic_vector(0 to 8);
      codonin2                  : in std_logic_vector(0 to 8);
      codonin3                  : in std_logic_vector(0 to 8);
      memPort                           : in std_logic_vector(0 to 8);
      hit                               : out std_logic
);
end amino;

architecture amino_behv of amino is


signal memhit : std_logic;
signal directhit : std_logic;

                begin

                process( aClk,codonin1, memPort )

                        begin

                if rising_edge(aClk) then

if (( (codonin1(2) = '1' or memPort(2) = '1' ) or ( codonin1(0 to 1) = memPort(0 to 1) ) ) and
   ( (codonin1(5) = '1' or memPort(5) = '1'  ) or ( codonin1(3 to 4) = memPort(3 to 4) ) ) and
   ( (codonin1(8) = '1' or memPort(8) = '1'  ) or ( codonin1(6 to 7) = memPort(6 to 7) ) )) or
```

```
  (( (codonin2(2) = '1' or memPort(2) = '1'  ) or ( codonin2(0 to 1) = memPort(0 to 1) ) ) and
   ( (codonin2(5) = '1' or memPort(5) = '1'  ) or ( codonin2(3 to 4) = memPort(3 to 4) ) ) and
   ( (codonin2(8) = '1' or memPort(8) = '1' ) or ( codonin2(6 to 7) = memPort(6 to 7) ) )) or

  (( (codonin3(2) = '1' or memPort(2) = '1'  ) or ( codonin3(0 to 1) = memPort(0 to 1) ) ) and
   ( (codonin3(5) = '1' or memPort(5) = '1'  ) or ( codonin3(3 to 4) = memPort(3 to 4) ) ) and
   ( (codonin3(8) = '1' or memPort(8) = '1' ) or ( codonin3(6 to 7) = memPort(6 to 7) ) )) then


                              hit <= '1';

                              else

                              hit <= '0';

                              end if;


                              end if;


                              end process;

 end amino_behv;
```

4. Tryptic Peptide Mass Calculator Controller (mod_calc.vhd)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


entity mod_calc is
generic( num_stages : integer := 10;
                  mass_bits : integer := 25 );
port (
                              clk                         : in std_logic;
                              calc_reset                  : in std_logic;
                              enable                      : in std_logic;
                              ramword                     : in std_logic_vector(63 downto 0);
                              masses                      : out std_logic_vector(0 to (num_stages)*(mass_bits)-1);
                              mass_save                   : out std_logic_vector(1 to 8);
                              complement_masses           : out std_logic_vector(0 to (num_stages)*(mass_bits)-1);
                              complement_mass_save        : out std_logic_vector(1 to 8);
                              rdy                         : out std_logic

            );
end mod_calc;

architecture calc_flow of mod_calc is
```

```
------------------------------------------------------------
-- Fragment detection units and mass LUTs

component masslut
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (5 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		enable		: IN STD_LOGIC  := '1';
		q		: OUT STD_LOGIC_VECTOR (mass_bits-1 DOWNTO 0)
	);
end component;

component cleavecheck
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (5 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		enable		: IN STD_LOGIC  := '1';
		q		: OUT STD_LOGIC_VECTOR (1 DOWNTO 0)
	);
end component;


COMPONENT ambigna IS
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		clken		: IN STD_LOGIC ;
		q		: OUT STD_LOGIC_VECTOR (0 DOWNTO 0)
	);
END COMPONENT;

---------------------------------------------------------------------




------------------------------------------------------------
-- Basically the same components; modified to produce values for the complementary strands

component compl_masslut
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (5 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		enable		: IN STD_LOGIC  := '1';
		q		: OUT STD_LOGIC_VECTOR (mass_bits-1 DOWNTO 0)
	);
end component;

component compl_cleavecheck
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (5 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		enable		: IN STD_LOGIC  := '1';
		q		: OUT STD_LOGIC_VECTOR (1 DOWNTO 0)
	);
end component;
```

```
COMPONENT compl_ambigna IS
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		clken		: IN STD_LOGIC ;
		q		: OUT STD_LOGIC_VECTOR (0 DOWNTO 0)
	);
END COMPONENT;

---------------------------------------------------------------------



	signal third_pos_check: std_logic_vector(1 to num_stages-1);
	signal ambig : std_logic_vector(1 to num_stages-1);
	signal word_stage : std_logic_vector(0 to 252);
	signal discard_buff2 : std_logic_vector(1 to num_stages);
	signal mlut_out : std_logic_vector(((num_stages-1)*mass_bits) -1 downto 0);
	signal mass_a   : std_logic_vector(0 to (num_stages-1)*(mass_bits)-1);
	signal mass_b   : std_logic_vector(0 to (num_stages-1)*(mass_bits)-1);
	signal discard : std_logic_vector(1 to num_stages);
	signal discard_buff : std_logic_vector(1 to num_stages);
	signal wordaccum : std_logic_vector(0 to (2*mass_bits)-1);
	signal accumsave : std_logic_vector(0 to (2*mass_bits)-1);
	signal init_ctr : std_logic_vector(3 downto 0);
	signal mass_break : std_logic_vector(1 to num_stages-1);
	signal slidingwindow : std_logic_vector(0 to (num_stages-1)*(mass_bits)-1);
	signal break_in_stage : std_logic_vector(1 to num_stages);
	signal bs_buff : std_logic_vector(0 to 2);
	signal following_break : std_logic_vector(1 to num_stages);
	signal save_b :std_logic_vector(1 to num_stages-1);
	signal slide_save :std_logic_vector(0 to num_stages-1);
	signal fb_buff : std_logic_vector(1 to num_stages);
	signal wildcard : std_logic_vector(1 to num_stages-1);
	signal sd_buff : std_logic_vector(1 to num_stages);
	signal sd_buff2 : std_logic_vector(1 to num_stages);
	signal start_detected : std_logic_vector(1 to num_stages);


-- Now all the same signals but for the complementary strand
	signal compl_third_pos_check: std_logic_vector(1 to num_stages-1);
	signal compl_ambig : std_logic_vector(1 to num_stages-1);
	signal compl_discard_buff2 : std_logic_vector(1 to num_stages);
	signal compl_mlut_out : std_logic_vector((num_stages-1)*mass_bits -1 downto 0);
	signal compl_mass_a   : std_logic_vector(0 to (num_stages-1)*mass_bits -1);
	signal compl_mass_b   : std_logic_vector(0 to (num_stages-1)*mass_bits -1);
	signal compl_discard : std_logic_vector(1 to num_stages);
	signal compl_discard_buff : std_logic_vector(1 to num_stages);
	signal compl_wordaccum : std_logic_vector(0 to (2*mass_bits)-1);
	signal compl_accumsave : std_logic_vector(0 to (2*mass_bits)-1);
	signal compl_init_ctr : std_logic_vector(3 downto 0);
	signal compl_mass_break : std_logic_vector(1 to num_stages-1);
	signal compl_slidingwindow : std_logic_vector(0 to (num_stages-1)*32 -1);
	signal compl_break_in_stage : std_logic_vector(1 to num_stages);
	signal compl_bs_buff : std_logic_vector(0 to 2);
	signal compl_following_break : std_logic_vector(1 to num_stages);
	signal compl_save_b :std_logic_vector(1 to num_stages-1);
	signal compl_slide_save :std_logic_vector(0 to num_stages-1);
	signal compl_fb_buff : std_logic_vector(1 to num_stages);
	signal compl_wildcard : std_logic_vector(1 to num_stages-1);
```

```
        signal compl_sd_buff : std_logic_vector(1 to num_stages);
-----------------------------------------------------------------



  signal m1 : std_logic_vector(0 to mass_bits-1);
  signal m2 : std_logic_vector(0 to mass_bits-1);
  signal m3 : std_logic_vector(0 to mass_bits-1);
  signal m4 : std_logic_vector(0 to mass_bits-1);
  signal m5 : std_logic_vector(0 to mass_bits-1);
  signal m6 : std_logic_vector(0 to mass_bits-1);
  signal m7 : std_logic_vector(0 to mass_bits-1);
  signal m8 : std_logic_vector(0 to mass_bits-1);

  signal cm1 : std_logic_vector(0 to mass_bits-1);
  signal cm2 : std_logic_vector(0 to mass_bits-1);
  signal cm3 : std_logic_vector(0 to mass_bits-1);
  signal cm4 : std_logic_vector(0 to mass_bits-1);
  signal cm5 : std_logic_vector(0 to mass_bits-1);
  signal cm6 : std_logic_vector(0 to mass_bits-1);
  signal cm7 : std_logic_vector(0 to mass_bits-1);
  signal cm8 : std_logic_vector(0 to mass_bits-1);




        type massStates is (reset,summing);
        attribute ENUM_ENCODING : STRING;
        attribute ENUM_ENCODING of massStates : type is "0 1";
        signal currState      : massStates;
        signal nextState      : massStates;


--      attribute syn_black_box         : boolean;
--      attribute syn_black_box of masslut       : component is true;
--      attribute syn_black_box of cleavecheck: component is true;
--      attribute syn_black_box of ambigna      : component is true;

--      attribute syn_black_box of compl_masslut          : component is true;
--      attribute syn_black_box of compl_cleavecheck    : component is true;
--      attribute syn_black_box of compl_ambigna          : component is true;

        begin



m1 <= mass_b(0 to mass_bits-1);
m2 <= mass_b(mass_bits to (mass_bits)+mass_bits-1);
m3 <= mass_b(2*mass_bits to (2*mass_bits)+mass_bits-1);
m4 <= mass_b(3*mass_bits to (3*mass_bits)+mass_bits-1);
m5 <= mass_b(4*mass_bits to (4*mass_bits)+mass_bits-1);
m6 <= mass_b(5*mass_bits to (5*mass_bits)+mass_bits-1);
m7 <= mass_b(6*mass_bits to (6*mass_bits)+mass_bits-1);
m8 <= accumsave(mass_bits to (mass_bits)+mass_bits-1);


cm1 <= compl_mass_b(0 to mass_bits-1);
cm2 <= compl_mass_b(mass_bits to (mass_bits)+mass_bits-1);
cm3 <= compl_mass_b(2*mass_bits to (2*mass_bits)+mass_bits-1);
cm4 <= compl_mass_b(3*mass_bits to (3*mass_bits)+mass_bits-1);
```

```
cm5 <= compl_mass_b(4*mass_bits to (4*mass_bits)+mass_bits-1);
cm6 <= compl_mass_b(5*mass_bits to (5*mass_bits)+mass_bits-1);
cm7 <= compl_mass_b(6*mass_bits to (6*mass_bits)+mass_bits-1);
cm8 <= compl_accumsave(mass_bits to (mass_bits)+mass_bits-1);



mass_save(1 to num_stages-1) <= save_b  ;
mass_save(num_stages) <= slide_save(num_stages-1) ;

complement_mass_save(1 to num_stages-1) <= compl_save_b  ;
complement_mass_save(num_stages) <= compl_slide_save(num_stages-1) ;



masses(0 to ((num_stages-1)*(mass_bits))-1) <= mass_b;
masses(((num_stages-1)*(mass_bits)) to ((num_stages-1)*(mass_bits))+mass_bits-1) <=accumsave((mass_bits) to
(2*mass_bits)-1);

complement_masses(0 to ((num_stages-1)*(mass_bits))-1) <= compl_mass_b;
complement_masses(((num_stages-1)*(mass_bits))      to      ((num_stages-1)*(mass_bits))+mass_bits-1)      <=
compl_accumsave((mass_bits) to (2*mass_bits)-1);


strand_ambiguites : for stage in 0 to num_stages-2 generate

      one_wildcard : ambigna PORT MAP (
                  address(0)              => word_stage(( 63+(63*(stage-1) - 9*(((stage-1)*(stage-1) + (stage-1))/2)) )
),
                  address(1)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+1),
                  address(2)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+3),
                  address(3)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+4),
                  clock       => clk,
                  clken       => enable,
                  q(0)        => ambig(stage+1) );
end generate;



strand_masses : for stage in 0 to num_stages-2 generate
      mlut: masslut PORT MAP (
                  address(5)              => word_stage(( 63+(63*(stage-1) - 9*(((stage-1)*(stage-1) + (stage-1))/2)) )),
                  address(4)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+1),
                  address(3)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+3),
                  address(2)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+4),
                  address(1)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+6),
                  address(0)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+7),
                  clock       => clk,
                  enable      => enable,
                  q           => mlut_out( ((stage*mass_bits)+mass_bits-1) downto stage*mass_bits)
```

```
	);

end generate;



strand_breaks : for stage in 0 to num_stages-2 generate
		clv : cleavecheck PORT MAP (
		address(5)		=> word_stage(( 63+(63*(stage-1) - 9*(((stage-1)*(stage-1) + (stage-1))/2)) )),
		address(4)		=>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+1),
		address(3)		=>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+3),
		address(2)		=>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+4),
		address(1)		=>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+6),
		address(0)		=>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+7),
		clock	=> clk,
		enable	=> enable,
		q(1)	=> sd_buff(stage+1),
		q(0)	=> fb_buff(stage+1)
	);
end generate;



-----------------------------------------------------------------
-- Now the portmappings for the complementary devices

compl_str_ambiguites : for stage in 0 to num_stages-2 generate
	one_compl_wild : compl_ambigna PORT MAP (
		address(0)		=>word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+6),
		address(1)		=>word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+7),
		address(2)		=>word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+3),
		address(3)		=>word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+4),
		clock	=> clk,
		clken	=> enable,
		q(0)	=> compl_ambig(stage+1)
	);
end generate;



compl_str_masses : for stage in 0 to num_stages-2 generate
	compl_mlut : compl_masslut PORT MAP (
		address(5)		=>word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+6),
		address(4)		=>word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+7),
		address(3)		=>word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+3),
		address(2)		=>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+4),
		address(1)		=> word_stage(( 63+(63*(stage-1) - 9*(((stage-1)*(stage-1) + (stage-1))/2)) )),
```

```
                address(0)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+1),
                clock       => clk,
                enable      => enable,
                q           => compl_mlut_out( ((stage*mass_bits)+mass_bits-1) downto stage*mass_bits)
        );

end generate;



compl_str_breaks : for stage in 0 to num_stages-2 generate
compl_clv : compl_cleavecheck PORT MAP (
                address(5) => word_stage(6),
                address(4)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+7),
                address(3)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+3),
                address(2)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+4),
                address(1)              => word_stage(( 63+(63*(stage-1) - 9*(((stage-1)*(stage-1) + (stage-1))/2)) )),
                address(0)              =>  word_stage((  63+(63*(stage-1)  -  9*(((stage-1)*(stage-1)  +  (stage-1))/2))
)+1),
                clock       => clk,
                enable      => enable,
                q(1)        => compl_sd_buff(stage+1),
                q(0)        => compl_fb_buff(stage+1)
        );
end generate;



-----------------------------------------------------------------


                process(currState,enable)
                begin
                if enable = '1' then
                        case currState is

                                when reset =>
                                        nextState <= summing;

                                when summing =>
                                        nextState <= summing;

                                when others =>
                                        nextState <= reset;

                        end case;
                end if;
                end process;


                process(clk,enable,calc_reset,word_stage)
                begin

                for stage in -1 to num_stages-3 loop
                        third_pos_check(stage+2) <= word_stage(71+(63*stage - 9*((stage*stage + stage)/2)));
                        compl_third_pos_check(stage+2)  <=  word_stage(65+(63*stage  -  9*((stage*stage  +
stage)/2)));
```

```
		end loop;


			if calc_reset = '1' then

				currState <= reset;

			elsif rising_edge(clk) then

	if enable = '1' then
-------------------------------------------------------------------------------
	-- Events that occur on every enabled edge
		currState <= nextState;
	-- All for the original (not complementary) strand

		bs_buff(1) <= bs_buff(0);
		bs_buff(2) <= bs_buff(1);
		following_break <= fb_buff;
		sd_buff2 <= sd_buff;
		start_detected <= sd_buff2;
		discard <= discard_buff;
		following_break(8) <= following_break(7);

	-- Same as above but for complementary strand

		compl_bs_buff(1) <= compl_bs_buff(0);
		compl_bs_buff(2) <= compl_bs_buff(1);
		compl_following_break <= compl_fb_buff;
		compl_discard <= compl_discard_buff;
		compl_following_break(8) <= compl_following_break(7);
-------------------------------------------------------------------------------

		if init_ctr >= 9 then
			rdy <= '1';
		else
			rdy <='0';
		end if;


		case currState is

		when reset =>

			init_ctr <= (others => '0');

-- All the initializations for the original strand
			wordaccum <= (others => '0');
			accumsave <= (others => '0');
			word_stage <= (others => '0');
			mass_a <= (others => '0');
			mass_b <= (others => '0');
			slide_save <= (others => '0');
			slidingwindow <= (others => '0');
			start_detected <= (others => '0');
			save_b <= (others => '0');
			bs_buff <= "100";
			break_in_stage <= (others => '0');
			following_break <= (others => '0');


-- All the initializations for the complementary strand
			compl_wordaccum <= (others => '0');
```

```
                        compl_accumsave <= (others => '0');
                        compl_mass_a <= (others => '0');
                        compl_mass_b <= (others => '0');
                        compl_slide_save <= (others => '0');
                        compl_slidingwindow <= (others => '0');
                        compl_save_b <= (others => '0');
                        compl_bs_buff <= "100";
                        compl_break_in_stage <= (others => '0');
                        compl_following_break <= (others => '0');


                when summing =>


                        if (init_ctr <= 8) then init_ctr <= init_ctr + 1; end if;


-----------------------------------------------------------------------------------------------
-- The first a-register always gets the mass of the first amino acid in every word
                        mass_a(0 to mass_bits-1) <= mlut_out(mass_bits-1 downto 0);
                        slide_save(0) <= sd_buff(1);
                        slidingwindow(0 to mass_bits-1)<= (others => '0');
                        bs_buff(0) <= '0';

--Similar setup for complementary strands
                        compl_mass_a(0 to mass_bits-1) <= compl_mlut_out(mass_bits-1 downto 0);
                        compl_slide_save(0) <= compl_sd_buff(1);
                        compl_slidingwindow(0 to mass_bits-1)<= (others => '0');
                        compl_bs_buff(0) <= '0';
-----------------------------------------------------------------------------------------------


-- This is the actual word pipeline, It starts with the full 63 bit word and at every stage it
-- processes 9 bits (one codon = one amino acid) until all 63 bits = 7 amino acids have been
-- processed (both the original and complementary strands use this pipe)

                        word_stage(0 to 62) <= ramword(63 downto 1);
                        for stage in 0 to num_stages - 3 loop
                                word_stage( ( 63+(63*stage - 9*((stage*stage + stage)/2)) ) to (((63+(63*stage
- 9*((stage*stage + stage)/2)) + (62 - (9*(stage+1) ) ) ) ) ) ) <= word_stage( (72+(63*(stage - 1) - 9*(((stage -
1)*(stage - 1) + (stage - 1))/2)) ) to ( 72+(63*(stage - 1) - 9*(((stage - 1)*(stage - 1) + (stage - 1))/2)) + (62 -
(9*((stage - 1)+2) ) )) );
                        end loop;



-----------------------------------------------------------------------------------------------
-- Wild card detectors for the original strand. They check every stage for a wild card in the
-- first two codons (guranteed wildcard) or the specific codons that will create ambiguity if
-- there is a wildcard in the third position


                for stage in -1 to num_stages-3 loop
                        wildcard(stage+2) <= word_stage(65+(63*stage - 9*((stage*stage + stage)/2))) OR
word_stage(68+(63*stage - 9*((stage*stage + stage)/2))) OR ( third_pos_check(stage+2) and ambig(stage+2) );
                end loop;


-- Same thing for the complementary strand, the only difference is that the ambiguity has
-- to be interpreted differently.

                for stage in -1 to num_stages-3 loop
```

```
                compl_wildcard(stage+2) <= word_stage(71+(63*stage - 9*((stage*stage + stage)/2)))
OR word_stage(68+(63*stage - 9*((stage*stage + stage)/2))) OR ( compl_third_pos_check(stage+2) and
compl_ambig(stage+2) );
        end loop;


-------------------------------------------------------------------------------------------
-- Keeps track of which words should not be saved (flushes the buffer on a wildcard)
                discard_buff(1) <= wildcard(1);
                for i in 2 to num_stages-1 loop
                        discard_buff(i) <= (discard_buff(i-1) OR wildcard(i-1) );
                end loop;

                        if slide_save(7) = '1' and (discard_buff(8) = '1') then

                                discard_buff(8) <= '0';
                        else
                                discard_buff(8) <= discard_buff(7);

                        end if;


-- Same for the complementary strand
-- Keeps track of which complementary fragments should not be saved (flushes the buffer on a wildcard)
                compl_discard_buff(1) <= compl_wildcard(1);
                for i in 2 to 7 loop
                        compl_discard_buff(i) <= (compl_discard_buff(i-1) OR compl_wildcard(i-1) );
                end loop;

                        if compl_slide_save(7) = '1' and (compl_discard_buff(8) = '1') then

                                compl_discard_buff(8) <= '0';
                        else
                                compl_discard_buff(8) <= compl_discard_buff(7);

                        end if;
-------------------------------------------------------------------------------------------

-- Keeps track of whether a certain word has seen a breakpoint yet. If it has not, then its starting
-- point was in some previous word. If it has seen a break, then it can be saved right away (its
-- starting point was in this word.)


                break_in_stage(1) <= bs_buff(2) or sd_buff(1) ;

                for i in 2 to 7 loop
                        break_in_stage(i) <= (break_in_stage(i-1) OR following_break(i)) or
sd_buff(i);
                end loop;

                        break_in_stage(8) <= break_in_stage(7);

-- The same for the complementary strand

                compl_break_in_stage(1) <= compl_bs_buff(2) or compl_sd_buff(1) ;

                for i in 2 to 7 loop
                        compl_break_in_stage(i) <= (compl_break_in_stage(i-1) OR
compl_following_break(i)) or compl_sd_buff(i);
                end loop;

                        compl_break_in_stage(8) <= compl_break_in_stage(7);
```

```
---------------------------------------------------------------------------------------------------


-- Stuff to deal with the sliding window
-- This is for the original strand
		for i in 1 to 6 loop

			if following_break(i)='0' and sd_buff(i+1) ='0' then
				slidingwindow((i)*mass_bits to (mass_bits)*(i)+(mass_bits-1)) <= slidingwindow((i-1)*(mass_bits) to (mass_bits)*(i-1)+(mass_bits-1));

				if slide_save(i-1) = '1' then
					slide_save(i) <= '1';
				else
					slide_save(i) <= '0';
				end if;

			else

				if break_in_stage(i) = '0'  then
					slide_save(i) <= '1';
					slidingwindow( ((mass_bits)*(i)) to ((mass_bits)*(i))+(mass_bits-1)) <= mass_a( ((mass_bits)*(i-1)) to ((mass_bits)*(i-1))+(mass_bits-1));
				else
					slidingwindow((i)*(mass_bits) to (mass_bits)*(i)+(mass_bits-1)) <= slidingwindow((i-1)*(mass_bits) to (mass_bits)*(i-1)+(mass_bits-1));
						if slide_save(i-1) = '1' then
							slide_save(i) <= '1';
						else
							slide_save(i) <= '0';
						end if;

				end if;

			end if;

		end loop;


		slide_save(7) <= (slide_save(6) or ( (not save_b(7)) and following_break(8) and (break_in_stage(7)) ) ) and (not discard(8) );


-- COMPLEMENTARY STRAND
-- Same thing : sliding window for the complementary strand

		for i in 1 to 6 loop

			if compl_following_break(i)='0' and compl_sd_buff(i+1) ='0' then
				compl_slidingwindow((i)*(mass_bits) to (mass_bits)*(i)+(mass_bits-1)) <= compl_slidingwindow((i-1)*(mass_bits) to (mass_bits)*(i-1)+(mass_bits-1));

				if compl_slide_save(i-1) = '1' then
					compl_slide_save(i) <= '1';
				else
					compl_slide_save(i) <= '0';
				end if;
```

```
                else

                        if compl_break_in_stage(i) = '0'  then
                                compl_slide_save(i) <= '1';
                                compl_slidingwindow(                ((mass_bits)*(i))                to
((mass_bits)*(i))+(mass_bits-1)) <= compl_mass_a( ((mass_bits)*(i-1)) to ((mass_bits)*(i-1))+(mass_bits-1));
                        else
                                compl_slidingwindow((i)*(mass_bits) to (mass_bits)*(i)+(mass_bits-
1)) <= compl_slidingwindow((i-1)*(mass_bits) to (mass_bits)*(i-1)+(mass_bits-1));
                                        if compl_slide_save(i-1) = '1' then
                                                compl_slide_save(i) <= '1';
                                        else
                                                compl_slide_save(i) <= '0';
                                        end if;

                        end if;

                end if;

        end loop;


        compl_slide_save(7)  <=  (compl_slide_save(6)  or  (  (not  compl_save_b(7))  and
compl_following_break(8) and (compl_break_in_stage(7)) ) ) and (not compl_discard(8) );




----------------------------------------------------------------------------------------------
-- ORIGINAL STRAND
-- The following loop determines when to add or flush the buffers
-- Stuff to deal with the actual summation and sending to scorer


        for i in 1 to 6 loop

                if following_break(i)='0' and sd_buff(i+1) ='0' then
                        mass_a(((mass_bits)*i)  to  (((mass_bits)*i)+(mass_bits-1))  )  <=  mlut_out(
(((mass_bits)*i)+(mass_bits-1))  downto  ((mass_bits)*i))  +  mass_a((i-1)*(mass_bits)  to  ((mass_bits)*(i-
1))+(mass_bits-1));
                        save_b(i) <= '0';
                else
                        mass_a(((mass_bits)*i)  to  (((mass_bits)*i)+(mass_bits-1))  )  <=  mlut_out(
(((mass_bits)*i)+(mass_bits-1)) downto ((mass_bits)*i));


                        if break_in_stage(i) = '0'  then
                                save_b(i) <= '0';
                        else
                                if discard(i) = '0' then
                                        save_b(i) <= '1';
                                end if;
                        end if;

                end if;

        end loop;
```

```
		if following_break(7) = '0' then
			save_b(7) <= '0';
		else

					if break_in_stage(7) = '0' then
						save_b(7) <= '0';
					else

						if discard(7) = '0' then
							save_b(7) <= '1';
						end if;

					end if;

		end if;



-- COMPLEMENTARY STRAND
-- The logic appears identical, but the mluts (the mass lookup tables) have been mapped differently to
-- account for the transposed and complemented nucleic acids within a word

		for i in 1 to 6 loop

			if compl_following_break(i)='0' and compl_sd_buff(i+1) ='0' then
					compl_mass_a(((mass_bits)*i)  to  (((mass_bits)*i)+(mass_bits-1))  ) <=
compl_mlut_out( (((mass_bits)*i)+(mass_bits-1)) downto ((mass_bits)*i)) + compl_mass_a((i-1)*(mass_bits) to
((mass_bits)*(i-1))+(mass_bits-1));
					compl_save_b(i) <= '0';
			else
					compl_mass_a(((mass_bits)*i)  to  (((mass_bits)*i)+(mass_bits-1))  ) <=
compl_mlut_out( (((mass_bits)*i)+(mass_bits-1)) downto ((mass_bits)*i));


					if compl_break_in_stage(i) = '0'  then
						compl_save_b(i) <= '0';
					else
						if compl_discard(i) = '0' then
								compl_save_b(i) <= '1';
						end if;
					end if;

			end if;

		end loop;




		if compl_following_break(7) = '0' then
			compl_save_b(7) <= '0';
		else

					if compl_break_in_stage(7) = '0' then
						compl_save_b(7) <= '0';
					else

						if compl_discard(7) = '0' then
							compl_save_b(7) <= '1';
						end if;

					end if;
```

```
                end if;

-----------------------------------------------------------------------------------------

        -- ORIGINAL STRAND
        -- The b registers are sent to scorer and the final accumulator
        -- The previous amino acid mass

                mass_b <= mass_a;

        -- COMPLEMENTARY STRAND
                compl_mass_b <= compl_mass_a;

-----------------------------------------------------------------------------------------

-- ORIGINAL STRAND
-- word accumulation if a single mass spans more than one word

                        if slide_save(6) = '1' then

                                if (discard(8) = '0') then
                                                if save_b(7) = '0'  then

                                                        accumsave <= wordaccum +
mass_b((num_stages-2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1) + slidingwindow((num_stages-
2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1);

                                                else
                                                        accumsave <= wordaccum +
slidingwindow((num_stages-2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1);
                                                end if;
                                end if;

                                wordaccum <= (others => '0');
                        else
                                if (discard(8) = '0' ) then
                                        accumsave <= wordaccum + mass_b((num_stages-2)*(mass_bits) to
(num_stages-2)*(mass_bits)+mass_bits-1);

                                        if following_break(7) = '0' then
                                                if save_b(7) = '0' then
                                                        wordaccum <= wordaccum +
mass_b((num_stages-2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1);
                                                else
                                                        wordaccum <= (others => '0');
                                                end if;


                                        else

                                                if save_b(7) = '0' then
                                                        wordaccum <= wordaccum +
mass_b((num_stages-2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1);
                                                else
                                                        wordaccum <= (others => '0');
                                                end if;
                                        end if;
                                end if;

                        end if;
```

```
-- COMPLEMENTARY STRAND
-- Same accumulation for the complementary strand

                        if compl_slide_save(6) = '1' then

                                if (compl_discard(8) = '0') then
                                        if compl_save_b(7) = '0'  then
                                                compl_accumsave <= compl_wordaccum +
compl_mass_b((num_stages-2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1) +
compl_slidingwindow((num_stages-2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1);
                                        else
                                                compl_accumsave <= compl_wordaccum +
compl_slidingwindow((num_stages-2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1);
                                        end if;
                                end if;

                                compl_wordaccum <= (others => '0');
                        else
                                if (compl_discard(8) = '0' ) then
                                        compl_accumsave <= compl_wordaccum +
compl_mass_b((num_stages-2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1);

                                        if compl_following_break(7) = '0' then
                                                if compl_save_b(7) = '0' then
                                                        compl_wordaccum <= compl_wordaccum +
compl_mass_b((num_stages-2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1);
                                                else
                                                        compl_wordaccum <= (others => '0');
                                                end if;


                                        else

                                                if compl_save_b(7) = '0' then
                                                        compl_wordaccum <= compl_wordaccum +
compl_mass_b((num_stages-2)*(mass_bits) to (num_stages-2)*(mass_bits)+mass_bits-1);
                                                else
                                                        compl_wordaccum <= (others => '0');
                                                end if;
                                        end if;
                                end if;

                        end if;







                when others =>

        end case;

        end if; -- for Altera's enable
        end if;

        end process;

end calc_flow;
```

5. Scoring Unit Controller (scorer.vhd)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity scorer is
        generic(  num_stages : integer := 10;
                  mass_bits : integer := 25;
                  tolerance_bits : integer := 3;
                  num_freq_bits : integer := 8;
                  num_bins: integer := 128;
                  selected_mass_bits : integer := 9;
                  encoder_mass_bits : integer := 7 );
    port (

            tm3_clk_v0          : in std_logic;
            reset               : in std_logic;
            MS_input            : in std_logic_vector((mass_bits-1) downto 0);
            score_tm3want       : out std_logic;
            score_sunready      : in std_logic;
            score_tm3ready      : out std_logic;
            score_sunwant       : in std_logic;
            hitlocation         : out std_logic_vector(18 downto 0);
            scan_complete       : out std_logic;
            good_match          : out std_logic_vector(0 to num_stages-1);
            compl_good_match    : out std_logic_vector(0 to num_stages-1);
            mem_scanned         : in std_logic;
            match_address       : in std_logic_vector(18 downto 0);
            mem_for_frame       : in std_logic_vector(63 downto 0);
            freq_product        : out std_logic_vector(0 to num_freq_bits-1);
            num_matches_out     : out std_logic_vector(7 downto 0);
            hist_max_freq       : out std_logic_vector(num_freq_bits-1 downto 0);
            compl_freq_product  : out std_logic_vector(0 to num_freq_bits-1);
            compl_num_matches_out : out std_logic_vector(7 downto 0);
            compl_hist_max_freq : out std_logic_vector(num_freq_bits-1 downto 0);
            calc_enable         : in std_logic;
            freq_enable_signal  : in std_logic;
            score_sent          : out std_logic

    );
end scorer;

architecture score_struct of scorer is

------------------------------------------------------------
-- Statistics for low/high frequency mass ranges
component mod_frequency_table
port (
                  clk           : in std_logic;
                  rst           : in std_logic;
                  enb           : in std_logic;
                  evaluate_mass : in std_logic;
                  max_freq : in std_logic_vector(0 to 5);
                  save_freq : in std_logic;
                  low_freq_peptides : out std_logic_vector(0 to num_stages-1);
                  mass_valid : in std_logic_vector(0 to num_stages-1);
                  matching_stages : in std_logic_vector(0 to num_stages-1);
                  hist_max_freq : out std_logic_vector(0 to num_freq_bits-1);
                  Pi_f : out std_logic_vector(0 to num_freq_bits-1);
                  mass_ranges       : in std_logic_vector(0 to (num_stages*7)-1));
end component;
```

```
-------------------------------------------------------
-- 128 entry RAM Block to store the MS detected values
component spec_vals
	port (
	address: IN std_logic_VECTOR(8 downto 0);
	clock: IN std_logic;
	data: IN std_logic_VECTOR(24 downto 0);
	q: OUT std_logic_VECTOR(24 downto 0);
	wren: IN std_logic);
END component;


-------------------------------------------------------
-- Fragment Mass Calculator
	component mod_calc
				port (
					clk : in std_logic;
					calc_reset : in std_logic;
					enable : in std_logic;
					ramword : in std_logic_vector(63
downto 0);
					masses : out std_logic_vector(0 to
(num_stages)*(mass_bits)-1);
					mass_save : out std_logic_vector(1 to
num_stages);
					complement_masses : out std_logic_vector(0 to
(num_stages)*(mass_bits)-1);
					complement_mass_save: out std_logic_vector(1 to num_stages);
					rdy : out std_logic);
	end component;
-------------------------------------------------------
-- Tolerance comparators to check how closely the detected values match the DB
component thresh_comp
	port (
	dataa: IN std_logic_VECTOR(2 downto 0);
	datab: IN std_logic_VECTOR(2 downto 0);
	clock: IN std_logic;
	AleB: OUT std_logic);
end component;
-------------------------------------------------------
-- ROMs to help count the total number of matches

component count_rom
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		enable		: IN STD_LOGIC  := '1';
		q		: OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
	);
end component;

-------------------------------------------------------

	type matchStates is
(rst,soft_rst,read1_MS1_data,read2_MS1_data,initialize,mem_load1,mem_load2,mem_save,return_score1,return_
score2,compare, done);
	signal currState : matchStates;
	signal currState_buffer : matchStates;
	signal nextState : matchStates;
	signal nextState_buffer : matchStates;
```

```
	signal memvar                                   : std_logic_vector(0 to 63);
	signal load_compare                             : std_logic;
	signal calc_difference                          : std_logic;
	signal hi                                       : std_logic;
	signal mass_line                                : std_logic_vector(0 to
(num_stages)*(mass_bits)-1);
	signal compl_mass_line                  : std_logic_vector(0 to (num_stages)*(mass_bits)-1);
	signal mass_save_line                           : std_logic_vector(1 to num_stages);
	signal compl_mass_save_line: std_logic_vector(1 to num_stages);

	signal freq_mass_line                           : std_logic_vector(0 to
(num_stages)*(mass_bits)-1);
	signal compl_freq_mass_line             : std_logic_vector(0 to (num_stages)*(mass_bits)-1);
	signal freq_mass_save_line                      : std_logic_vector(1 to num_stages);
	signal compl_freq_mass_save_line        : std_logic_vector(1 to num_stages);

	signal user_tolerance                           : std_logic_vector(tolerance_bits-1 downto 0);

	signal pipe_mass_line                           : std_logic_vector(0 to
(num_stages)*(mass_bits)-1);
	signal pipe_compl_mass_line: std_logic_vector(0 to (num_stages)*(mass_bits)-1);
	signal pipe2_mass_line                          : std_logic_vector(0 to
(num_stages)*(mass_bits)-1);
	signal pipe2_compl_mass_line            : std_logic_vector(0 to (num_stages)*(mass_bits)-1);

	signal mass_index_buffer1  : std_logic_vector( 0 to (num_stages)*(encoder_mass_bits)-1);
	signal mass_index_buffer2  : std_logic_vector( 0 to (num_stages)*(encoder_mass_bits)-1);
	signal mass_index : std_logic_vector( 0 to (num_stages)*(encoder_mass_bits)-1);

	signal diff                                     : std_logic_vector(
((mass_bits)*(num_stages))-1 downto 0);
	signal compl_diff                               : std_logic_vector( ((mass_bits)*(num_stages))-1
downto 0);

	signal absdiff                                  : std_logic_vector(
((mass_bits)*(num_stages))-1 downto 0);
	signal compl_absdiff                            : std_logic_vector(
((mass_bits)*(num_stages))-1 downto 0);

--	signal good_match : std_logic_vector(0 to num_stages-1);


	signal spec_mass                                : std_logic_vector( (mass_bits-1) downto 0);
	signal stored_spec_mass                         : std_logic_vector( ((mass_bits)*(num_stages))-1
downto 0);
	signal compl_stored_spec_mass           : std_logic_vector( ((mass_bits)*(num_stages))-1 downto
0);

	signal stored_spec_mass_reg                     : std_logic_vector( ((mass_bits)*(num_stages))-1
downto 0);
	signal compl_stored_spec_mass_reg       : std_logic_vector( ((mass_bits)*(num_stages))-1 downto
0);

	signal match_ctr                                : std_logic_vector(7 downto 0);
	signal mem_ctr                                  : std_logic_vector(7 downto 0);
	signal index                                    : std_logic_vector(0
to((num_stages)*(selected_mass_bits))-1 );
	signal compl_index                              : std_logic_vector(0
to((num_stages)*(selected_mass_bits))-1 );
	signal frame_calc_ready                         : std_logic;
	signal freq_calc_ready                          : std_logic;
```

```
	signal match_found			: std_logic_vector( (num_stages)-1 downto 0);
	signal compl_match_found	: std_logic_vector( (num_stages)-1 downto 0);

	signal num_matches			: std_logic_vector(7 downto 0);
	signal compl_num_matches	: std_logic_vector(7 downto 0);

	signal curr_num_match			: std_logic_vector( 3 downto 0);
	signal compl_curr_num_match		: std_logic_vector( 3 downto 0);

	signal msb_below_thresh : std_logic_vector(num_stages-1 downto 0);
	signal lsb_below_thresh : std_logic_vector(num_stages-1 downto 0);

	signal compl_msb_below_thresh : std_logic_vector(num_stages-1 downto 0);
	signal compl_lsb_below_thresh : std_logic_vector(num_stages-1 downto 0);

	signal low_freq_peptides : std_logic_vector(0 to num_stages-1);
	signal compl_low_freq_peptides : std_logic_vector(0 to num_stages-1);

	signal freqtable_mass_line : std_logic_vector(0 to (num_stages*7)-1);
	signal compl_freqtable_mass_line : std_logic_vector(0 to (num_stages*7)-1);
	signal max_freq : std_logic_vector(0 to 5);
	signal freq_en_buff : std_logic;
	signal save_freq : std_logic;
	signal evaluate_mass : std_logic;
	signal freq_mass_valid : std_logic_vector(0 to num_stages-1);
	signal compl_freq_mass_valid : std_logic_vector(0 to num_stages-1);
	signal table_enable : std_logic;
--	signal max_freq : std_logic_vector(0 to 5);
	signal pipe_low_freq : std_logic_vector(0 to (num_stages*3)-1);
	signal compl_pipe_low_freq : std_logic_vector(0 to (num_stages*3)-1);

	signal reg_freq_enable : std_logic;
	signal reg_calc_enable : std_logic;

--	attribute syn_black_box : boolean;
--	attribute syn_black_box of spec_buffer: component is true;
--	attribute syn_black_box of count_rom: component is true;

begin
hi <= '1';
user_tolerance <= "001";
table_enable <= (freq_enable_signal AND freq_calc_ready) OR (calc_enable and frame_calc_ready);
evaluate_mass <= calc_enable;
max_freq <= "011001";

	selector_units :  for i in 0 to num_stages-1 generate
		single_stage_buffer : spec_vals PORT MAP (
			address	=> index( selected_mass_bits*i to (selected_mass_bits*i)+selected_mass_bits-
1 ),
			clock	=> tm3_clk_v0,
			data	=> spec_mass,
			wren	=> load_compare,
			q	=> stored_spec_mass( (mass_bits*i)+(mass_bits-1) downto mass_bits*i )
			);
		end generate selector_units;


	complement_selector_units :  for i in 0 to num_stages-1 generate
		compl_stage_buffer : spec_vals PORT MAP (
			address	=>		compl_index(		selected_mass_bits*i		to
(selected_mass_bits*i)+selected_mass_bits-1 ),
```

```
                clock       => tm3_clk_v0,
                data        => spec_mass,
                wren        => load_compare,
                q           =>      compl_stored_spec_mass(     (mass_bits*i)+(mass_bits-1)     downto
mass_bits*i )
                );
        end generate complement_selector_units;



freqTable : mod_frequency_table port map(
                                        clk         => tm3_clk_v0,
                                        rst                 => reset,
                                        enb                 => table_enable,
                                        evaluate_mass => evaluate_mass,
                                        max_freq => max_freq,
                                        save_freq => save_freq,
                                        low_freq_peptides => low_freq_peptides,
                                        mass_valid => freq_mass_valid,
                                        matching_stages => match_found,
                                        hist_max_freq => hist_max_freq,
                                        Pi_f => freq_product,
                                        mass_ranges         => freqtable_mass_line  );


compl_freqTable : mod_frequency_table port map(
                                        clk         => tm3_clk_v0,
                                        rst                 => reset,
                                        enb                 => table_enable,
                                        evaluate_mass => evaluate_mass,
                                        max_freq => max_freq,
                                        save_freq => save_freq,
                                        low_freq_peptides => compl_low_freq_peptides,
                                        mass_valid => compl_freq_mass_valid,
                                        matching_stages => compl_match_found,
                                        hist_max_freq => compl_hist_max_freq,
                                        Pi_f => compl_freq_product,
                                        mass_ranges         => compl_freqtable_mass_line  );



 frame1_calculator : mod_calc port map(
                clk                                     => tm3_clk_v0,
                calc_reset                      => reset,
                enable                          => calc_enable,
                ramword                                 => mem_for_frame,
                masses                          => mass_line,
                mass_save                       => mass_save_line,
                complement_masses   => compl_mass_line,
                complement_mass_save => compl_mass_save_line,
                rdy                                     =>  frame_calc_ready  );


 freq_calculator : mod_calc port map(
                clk                                     => tm3_clk_v0,
                calc_reset                      => reset,
                enable                          => freq_enable_signal,
                ramword                                 => mem_for_frame,
                masses                          => freq_mass_line,
                mass_save                       => freq_mass_save_line,
```

```
                complement_masses   => compl_freq_mass_line,
                complement_mass_save => compl_freq_mass_save_line,
                rdy                                     =>  freq_calc_ready  );


        check_difference: for i in 0 to num_stages-1 generate
                mass_compare : thresh_comp PORT MAP (
                dataa       => absdiff( (mass_bits*i)+(tolerance_bits-1) downto mass_bits*i),
                datab       => user_tolerance,
                clock       => tm3_clk_v0,
                AleB        => lsb_below_thresh(i)
        );
                end generate check_difference;




        compl_check_difference: for i in 0 to num_stages-1 generate
                compl_mass_compare : thresh_comp PORT MAP (
                        dataa       => compl_absdiff( (mass_bits*i)+(tolerance_bits-1) downto mass_bits*i),
                        datab       => user_tolerance,
                        clock       => tm3_clk_v0,
                        AleB        => compl_lsb_below_thresh(i)
                        );
                end generate compl_check_difference;




        m_counter : count_rom PORT MAP (
                address     => match_found,
                clock       => tm3_clk_v0,
                enable      => hi,
                q           => curr_num_match
        );


        cm_counter : count_rom PORT MAP (
                address     => compl_match_found,
                clock       => tm3_clk_v0,
                enable      => hi,
                q           => compl_curr_num_match
        );



        process(currState,MS_input,match_ctr,mem_ctr,score_sunready,freq_enable_signal,calc_enable,score_sunwa
nt,mem_scanned)
        begin


                        load_compare <= '0';
                        calc_difference <= '1';

                        score_tm3want <= '0';
                        score_tm3ready <= '0';
                        score_sent <= '1';
                        scan_complete <= '0';

                        -- I'll clock it, the delay is too much (and make sure the freq_en_buff gets a max_fan
restriction
```

```
		--if falling_edge(freq_enable_signal) then
		--if freq_en_buff = '1' and freq_enable_signal = '0' then
		--		save_freq <= '1';
		--else
		--		save_freq <= '0';
		--end if;

case currState is

		when rst =>
				nextState <= read1_MS1_data;
				nextState_buffer <= read1_MS1_data;

		when read1_MS1_data =>
				score_sent <= '0';
				score_tm3want <= '1';

				if score_sunready = '1' then
						nextState <= read2_MS1_data;
						nextState_buffer <= read2_MS1_data;
				else
						nextState <= read1_MS1_data;
						nextState_buffer <= read1_MS1_data;
				end if;

		when read2_MS1_data =>
				score_sent <= '0';
				score_tm3want <= '0';

				if score_sunready = '0' then
						nextState <= initialize;
						nextState_buffer <= initialize;
				else
						nextState <= read2_MS1_data;
						nextState_buffer <= read2_MS1_data;
				end if;


		when initialize =>
				load_compare <= '1';
				score_sent <= '0';

				if (match_ctr = "01111111") then
						nextState <= soft_rst;
						nextState_buffer <= soft_rst;
				else
						nextState <= read1_MS1_data;
						nextState_buffer <= read1_MS1_data;
				end if;




		when compare =>


				--if (mem_ctr <= 29) then
				if calc_enable = '1' then
						nextState <= compare;
						nextState_buffer <= compare;
				else
						nextState <= return_score1;
```

```
                        nextState_buffer <= return_score1;
                end if;


        when return_score1 =>
                score_sent <= '0';
                score_tm3ready <= '1';

                if score_sunwant = '1' then
                        nextState <= return_score2;
                        nextState_buffer <= return_score2;
                else
                        nextState <= return_score1;
                        nextState_buffer <= return_score1;
                end if;


        when return_score2 =>
                score_sent <= '0';
                score_tm3ready <= '0';

                if score_sunwant = '0' then
                        nextState <= soft_rst;
                        nextState_buffer <= soft_rst;
                else
                        nextState <= return_score2;
                        nextState_buffer <= return_score2;
                end if;


        when soft_rst =>

                if calc_enable = '1'  then
                        nextState <= compare;
                        nextState_buffer <= compare;
                else
                        nextState <= soft_rst;
                        nextState_buffer <= soft_rst;
                end if;


        when done =>

                scan_complete <= '1';
                nextState <= done;
                nextState_buffer <= done;


        when others =>
                nextState <= rst;
                nextState_buffer <= rst;


    end case;

end process;



process(tm3_clk_v0,reset,freq_calc_ready,frame_calc_ready,calc_difference,mass_line,compl_mass_line,mem_scanned)
```

```
	begin


		if reset= '1' then
			currState <= rst;
		elsif mem_scanned = '1' then
			currState <= done;
		elsif rising_edge(tm3_clk_v0) then

-- save the "matching" mass, or at least the first bits to use as an index for the PIS
		for i in 0 to num_stages-1 loop
			mass_index_buffer1( (i*encoder_mass_bits) to (i*encoder_mass_bits) + encoder_mass_bits-1 ) <= pipe2_mass_line( (i*mass_bits) to (i*mass_bits) + encoder_mass_bits-1 );
		end loop;
			mass_index_buffer2 <= mass_index_buffer1;
			mass_index <= mass_index_buffer2;

-- register these two so I can pipeline the sig and move it away from the BRAM
			stored_spec_mass_reg <= stored_spec_mass;
			compl_stored_spec_mass_reg <= compl_stored_spec_mass;

-- wideor changed

				currState <= nextState;
				currState_buffer <= nextState_buffer;

--these two enables have become clocked signals
--			table_enable <= freq_enable_signal OR calc_enable;
--			evaluate_mass <= calc_enable;

			if freq_en_buff = '1' and freq_enable_signal = '0' then
				save_freq <= '1';
			else
				save_freq <= '0';
			end if;

			freq_en_buff <= freq_enable_signal;

			for i in 0 to num_stages-1 loop
				good_match(i) <= pipe_low_freq(i) AND match_found(i);
			end loop;

			for i in 0 to num_stages-1 loop
				compl_good_match(i) <= compl_pipe_low_freq(i) AND compl_match_found(i);
			end loop;


			for i in 0 to 1 loop
				pipe_low_freq(num_stages*i to num_stages*i+(num_stages-1)) <= pipe_low_freq(num_stages*(i+1) to num_stages*(i+1)+(num_stages-1) );
			end loop;
				pipe_low_freq(num_stages*2 to num_stages*2+(num_stages-1)) <= low_freq_peptides;

			for i in 0 to 1 loop
				compl_pipe_low_freq(num_stages*i to num_stages*i+(num_stages-1)) <= compl_pipe_low_freq(num_stages*(i+1) to num_stages*(i+1)+(num_stages-1) );
			end loop;
				compl_pipe_low_freq(num_stages*2 to num_stages*2+(num_stages-1)) <= compl_low_freq_peptides;
```

```
                        for i in 0 to num_stages-1 loop
                                if evaluate_mass = '0' then
                                        freq_mass_valid(i) <= freq_mass_save_line(i+1);
                                        freqtable_mass_line(i*7 to (i*7)+6) <= freq_mass_line(i*mass_bits
to ((i*mass_bits)+6));
                                else
                                        freq_mass_valid(i) <= mass_save_line(i+1);
                                --      freqtable_mass_line(i*7 to (i*7)+6) <= mass_line(i*mass_bits to
((i*mass_bits)+6));
                                        freqtable_mass_line(i*7 to (i*7)+6) <= mass_index(i*7 to (i*7)+6);
                                end if;
                        end loop;


                        for i in 0 to num_stages-1 loop
                                if evaluate_mass = '0' then
                                        compl_freq_mass_valid(i) <= compl_freq_mass_save_line(i+1);
                                        compl_freqtable_mass_line(i*7 to (i*7)+6) <=
compl_freq_mass_line(i*mass_bits to ((i*mass_bits)+6));
                                else
                                        compl_freq_mass_valid(i) <= compl_mass_save_line(i+1);
                                        --compl_freqtable_mass_line(i*7 to (i*7)+6) <=
compl_mass_line(i*mass_bits to ((i*mass_bits)+6));
-- FIX
                                        compl_freqtable_mass_line(i*7 to (i*7)+6) <= mass_index(i*7 to
(i*7)+6);
                                end if;
                        end loop;




                if freq_calc_ready = '1' then
                        pipe_mass_line                  <= mass_line;
                        pipe_compl_mass_line    <= compl_mass_line;

                        pipe2_mass_line                 <= pipe_mass_line;
                        pipe2_compl_mass_line   <= pipe_compl_mass_line;

                end if;

                                num_matches_out <= num_matches;
                                compl_num_matches_out <= compl_num_matches;

                        if (frame_calc_ready = '1') and (calc_enable = '1') then
                                num_matches <= num_matches + "0000"+ curr_num_match;
                                compl_num_matches <= compl_num_matches + "0000"+
compl_curr_num_match;
                        end if;


                        for i in 0 to num_stages-1 loop
                                msb_below_thresh(i) <= NOT (absdiff((mass_bits*i)+3) OR
absdiff((mass_bits*i)+4) OR absdiff((mass_bits*i)+5) OR absdiff((mass_bits*i)+6) OR
absdiff((mass_bits*i)+selected_mass_bits) OR absdiff((mass_bits*i)+num_stages) OR absdiff((mass_bits*i)+9)
OR absdiff((mass_bits*i)+10) OR absdiff((mass_bits*i)+11) OR absdiff((mass_bits*i)+12) OR
absdiff((mass_bits*i)+13) OR absdiff((mass_bits*i)+14) OR absdiff((mass_bits*i)+ 15 ) OR
absdiff((mass_bits*i)+ 16 ) OR absdiff((mass_bits*i)+ 17 ) OR absdiff((mass_bits*i)+ 18 ) OR
```

```
absdiff((mass_bits*i)+ 19 )OR absdiff((mass_bits*i)+ 20 ) OR absdiff((mass_bits*i)+ 21 ) OR absdiff((mass_bits*i)+ 22 ) OR absdiff((mass_bits*i)+ 23 ) OR absdiff((mass_bits*i)+ mass_bits-1 ) );
                                compl_msb_below_thresh(i) <= NOT (compl_absdiff((mass_bits*i)+3) OR compl_absdiff((mass_bits*i)+4) OR compl_absdiff((mass_bits*i)+5) OR compl_absdiff((mass_bits*i)+6) OR compl_absdiff((mass_bits*i)+selected_mass_bits) OR compl_absdiff((mass_bits*i)+num_stages) OR compl_absdiff((mass_bits*i)+9) OR compl_absdiff((mass_bits*i)+10) OR compl_absdiff((mass_bits*i)+11) OR compl_absdiff((mass_bits*i)+12) OR compl_absdiff((mass_bits*i)+13) OR compl_absdiff((mass_bits*i)+14) OR compl_absdiff((mass_bits*i)+ 15 ) OR compl_absdiff((mass_bits*i)+ 16 ) OR compl_absdiff((mass_bits*i)+ 16 ) OR compl_absdiff((mass_bits*i)+ 17 ) OR compl_absdiff((mass_bits*i)+ 18 ) OR compl_absdiff((mass_bits*i)+ 19 ) OR compl_absdiff((mass_bits*i)+ 20 ) OR compl_absdiff((mass_bits*i)+ 21 ) OR compl_absdiff((mass_bits*i)+ 22 ) OR compl_absdiff((mass_bits*i)+ 23 ) OR compl_absdiff((mass_bits*i)+ mass_bits-1 ) );

                                msb_below_thresh(i) <= NOT (absdiff((mass_bits*i)+3) OR absdiff((mass_bits*i)+4) OR absdiff((mass_bits*i)+5) OR absdiff((mass_bits*i)+6) OR absdiff((mass_bits*i)+selected_mass_bits) OR absdiff((mass_bits*i)+num_stages) OR absdiff((mass_bits*i)+9) OR absdiff((mass_bits*i)+10) OR absdiff((mass_bits*i)+11) OR absdiff((mass_bits*i)+12) OR absdiff((mass_bits*i)+13) OR absdiff((mass_bits*i)+14) OR absdiff((mass_bits*i)+ 15 ) OR absdiff((mass_bits*i)+ 16 ) OR absdiff((mass_bits*i)+ 17 ) OR absdiff((mass_bits*i)+ 18 ) OR absdiff((mass_bits*i)+ mass_bits-1 ) );
                                compl_msb_below_thresh(i) <= NOT (compl_absdiff((mass_bits*i)+3) OR compl_absdiff((mass_bits*i)+4) OR compl_absdiff((mass_bits*i)+5) OR compl_absdiff((mass_bits*i)+6) OR compl_absdiff((mass_bits*i)+selected_mass_bits) OR compl_absdiff((mass_bits*i)+num_stages) OR compl_absdiff((mass_bits*i)+9) OR compl_absdiff((mass_bits*i)+10) OR compl_absdiff((mass_bits*i)+11) OR compl_absdiff((mass_bits*i)+12) OR compl_absdiff((mass_bits*i)+13) OR compl_absdiff((mass_bits*i)+14) OR compl_absdiff((mass_bits*i)+ 15 ) OR compl_absdiff((mass_bits*i)+ 16 ) OR compl_absdiff((mass_bits*i)+ 16 ) OR compl_absdiff((mass_bits*i)+ 17 ) OR compl_absdiff((mass_bits*i)+ 18 ) OR compl_absdiff((mass_bits*i)+ mass_bits-1 ) );


                                match_found(i) <=  msb_below_thresh(i) AND lsb_below_thresh(i);
                                compl_match_found(i) <= compl_msb_below_thresh(i) AND compl_lsb_below_thresh(i);

                        end loop;




                        case currState_buffer is

                                when rst =>
                                        match_ctr <= (others => '0');
                                        mem_ctr <= (others => '0');
                                        diff <= (others => '0');
                                        compl_diff <= (others => '0');
                                        absdiff <= (others => '0');
                                        compl_absdiff <= (others => '0');

                                        num_matches <= (others => '0');
                                        compl_num_matches <= (others => '0');
                                        match_found <= (others => '0');
                                        compl_match_found <= (others => '0');
                                        spec_mass <= (others => '0');
                                        index <= (others => '0');
                                        compl_index <= (others => '0');

                                when soft_rst =>
-- reset all the intermediate accumulators
                                        match_ctr <= (others => '0');
                                        mem_ctr <= (others => '0');
```

```
                    diff <= (others => '1');
                    compl_diff <= (others => '0');
                    absdiff <= (others => '1');
                    compl_absdiff <= (others => '0');
                    num_matches <= (others => '0');
                    compl_num_matches <= (others => '0');
                    msb_below_thresh <= (others => '0');
                    match_found <= (others => '0');
                    compl_match_found <= (others => '0');
                    spec_mass <= (others => '0');
                    mem_ctr <= (others => '0');
                    hitlocation <= match_address;
                    index <= (others => '0');
                    compl_index <= (others => '0');


                when initialize =>
                    match_ctr <= match_ctr + 1;
                    spec_mass <= MS_input;


                for i in 0 to num_stages-1 loop
                    index( (selected_mass_bits*i) to ((selected_mass_bits*i)+(selected_mass_bits-1)) ) <= MS_input((mass_bits-1) downto (mass_bits-selected_mass_bits) );
                    compl_index( (selected_mass_bits*i) to ((selected_mass_bits*i)+(selected_mass_bits-1)) ) <= MS_input((mass_bits-1) downto (mass_bits-selected_mass_bits) );
                end loop;

                when mem_save =>
                    memvar <= mem_for_frame;


                when compare =>

                mem_ctr <= mem_ctr + 1;

                for i in 0 to num_stages-1 loop

                    diff( (mass_bits*i)+(mass_bits-1) downto mass_bits*i ) <= stored_spec_mass((mass_bits*i)+(mass_bits-1) downto mass_bits*i ) - pipe2_mass_line(mass_bits*i to (mass_bits*i)+(mass_bits-1));
                    compl_diff( (mass_bits*i)+(mass_bits-1) downto mass_bits*i ) <= compl_stored_spec_mass((mass_bits*i)+(mass_bits-1) downto mass_bits*i) - pipe2_compl_mass_line(mass_bits*i to (mass_bits*i)+(mass_bits-1) ) ;


                    absdiff( (mass_bits*i)+(mass_bits-1) downto mass_bits*i ) <= abs( diff( (mass_bits*i)+(mass_bits-1) downto mass_bits*i ) ) ;
                    compl_absdiff( (mass_bits*i)+(mass_bits-1) downto mass_bits*i ) <= abs(compl_diff( (mass_bits*i)+(mass_bits-1) downto mass_bits*i ) );

                end loop;

                for i in 0 to num_stages-1 loop
                    if mass_save_line(i+1) = '1' then
                        index( (selected_mass_bits*i) to ((selected_mass_bits*i)+(selected_mass_bits-1)) ) <= mass_line( (mass_bits*i) to ((mass_bits*i)+selected_mass_bits)-1 );
                    else
```

```
                                                index(                  (selected_mass_bits*i)                  to
((selected_mass_bits*i)+(selected_mass_bits-1)) ) <= (others => '1');
                                        end if;

                                        if compl_mass_save_line(i+1) = '1'  then
                                                compl_index(            (selected_mass_bits*i)          to
((selected_mass_bits*i)+(selected_mass_bits-1))      )      <=      compl_mass_line(      (mass_bits*i)      to
((mass_bits*i)+selected_mass_bits)-1  );
                                        else
                                                compl_index(            (selected_mass_bits*i)          to
((selected_mass_bits*i)+(selected_mass_bits-1)) ) <= (others => '1');
                                        end if;
                                end loop;


                        when return_score1 =>

                                when others =>

                        end case;

                end if;
        end process;

        end score_struct;
```

6. Histogram Architecture (mod_frequency_table.vhd)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mod_frequency_table is
                generic(  num_stages : integer := 10;
                          num_freq_bits : integer := 8;
                          size : integer := 8*8 ;
                          shift : integer := 8;
                          num_bins: integer := 128 );
port (
        clk               : in std_logic;
        rst               : in std_logic;
        enb               : in std_logic;
        evaluate_mass     : in std_logic;
        max_freq          : in std_logic_vector(0 to 5);
        save_freq         : in std_logic;
        low_freq_peptides : out std_logic_vector(0 to num_stages-1);
        mass_valid        : in std_logic_vector(0 to num_stages-1 );
        matching_stages   : in std_logic_vector(0 to num_stages-1);
        hist_max_freq     : out std_logic_vector(0 to num_freq_bits-1);
        Pi_f              : out std_logic_vector(0 to num_freq_bits-1);
        mass_ranges       : in std_logic_vector(0 to (num_stages*7)-1)

);
end mod_frequency_table;

architecture mod_stats of mod_frequency_table is

-- decoder to decide which range is being incremented
```

```
component bin_decoder
        port (
        address: IN std_logic_VECTOR(6 downto 0);
        clock: IN std_logic;
        q: OUT std_logic_VECTOR(127 downto 0);
        clken: IN std_logic);
end component;
-----------------------------------------------------
-- ROMs to help count the total number of matches
component count_rom
        port (
        address: IN std_logic_VECTOR(7 downto 0);
        clock: IN std_logic;
        enable: IN std_logic;
        q: OUT std_logic_VECTOR(3 downto 0));
end component;
-----------------------------------------------------
-- check to see if any of the frequency bins meet low thresh
component or_34
   Port (
                                                clk : in std_logic;
                                             or_in : in std_logic_vector(127 downto 0);
      or_out : out std_logic);
end component;
-----------------------------------------------------
-- log conversion LUTs

component logtable
        port (
        A: IN std_logic_VECTOR(5 downto 0);
        CLK: IN std_logic;
        QSPO_CE: IN std_logic;
        QSPO: OUT std_logic_VECTOR(7 downto 0));
end component;

-----------------------------------------------------

        type freqStates is (reset,update_stats,locate_max_freq,rank_masses);
        signal currState : freqStates;
        signal nextState : freqStates;
        signal full_max_freq : std_logic_vector(0 to num_freq_bits-1);
        signal element_counter : std_logic_vector(6 downto 0);
        signal hist_max_freq_reg : std_logic_vector(0 to num_freq_bits-1);
        signal frequency : std_logic_vector(0 to (num_bins * num_freq_bits)-1);
        signal saved_freq : std_logic_vector(0 to (num_bins * num_freq_bits)-1);
        signal saved_frequency_table : std_logic_vector(0 to (num_bins * num_freq_bits)-1);
        signal increment_range : std_logic_vector(0 to (128*num_stages)-1);
        signal rev_increment_range : std_logic_vector((128*num_stages)-1 downto 0 );
        signal increment_amount : std_logic_vector(0 to (num_bins*4)-1);
        signal addr : std_logic_vector(0 to (num_bins*8)-1);
        signal bin_incr : std_logic;
        signal flagged_ranges : std_logic_vector(0 to (num_bins*num_stages)-1);
        signal freq_table_copies : std_logic_vector(0 to (num_freq_bits*num_bins*num_stages)-1);
        signal low_freq_range : std_logic_vector(0 to num_bins-1);
        signal pipe_mass_valid : std_logic_vector(0 to num_stages-1);
        signal matching_mass : std_logic_vector(0 to num_stages-1);
        signal frequency_pipeline : std_logic_vector(0 to (num_freq_bits*num_stages)-1);

        signal log_accum : std_logic;
        signal logadder_pipe : std_logic_vector(0 to (num_freq_bits* (((num_stages*num_stages)+num_stages)/2) )-
1);
        signal log_val_stages : std_logic_vector(0 to (num_stages*num_freq_bits)-1 );
```

```
        signal log_val_accum : std_logic_vector(0 to (num_stages*num_freq_bits)-1);
        signal temp_test  :std_logic_vector(0 to (num_stages  * num_freq_bits)-1 );

begin



        rev_increment_range <= increment_range ;
        full_max_freq <= "00" & max_freq;




log_convert : for i in 0 to num_stages-1 generate
convert_freq :  logtable port  map(
        A  =>  logadder_pipe(  ((  size+(size*(i-1)  -  shift*(((i-1)*(i-1)  +  (i-1))/2))  )  +  2)   to  ((  size+(size*(i-1)  -
shift*(((i-1)*(i-1) + (i-1))/2)) ) + 7) ),
        CLK => clk,
        QSPO_CE => evaluate_mass,
        QSPO => log_val_stages( i*num_freq_bits to (i*num_freq_bits) + (num_freq_bits-1) )

        );
end generate log_convert;




range_selectors :    for i in 0 to num_stages-1 generate
                                                        range_decoder : bin_decoder port map(
                                                                address=>  mass_ranges(  7*i
to (7*i + 6) ),

                                                                clock => clk,
                                                                clken => mass_valid(i),
                                                                q  =>  increment_range(128*i
to  (128*i)+127)
                                                        );
                                                end generate range_selectors;


incrementors :       for i in 0 to num_bins-1 generate
                                                        range_increment_value: count_rom port map (
                                                                address => addr(i*8 to (i*8)+7),
                                                                clock => clk,
                                                                enable => bin_incr,
                                                                q => increment_amount(i*4 to (i*4)+3)
                                                        );
                                                end generate incrementors;


good_ranges : for i in 0 to num_stages-1 generate
                                                check_mass_range: or_34 port map (
                                                        clk => clk,
                                                        or_in => flagged_ranges(i*128 to (i*128)+127 ),
                                        or_out =>low_freq_peptides((num_stages-1)-i)
                        );
                                                end generate good_ranges;
```

```
process(currState,evaluate_mass,save_freq)
begin
                bin_incr <= '0';

        case currState is

                when reset =>
                        nextState <= update_stats;

                when update_stats =>
                        bin_incr <= '1';

                        if save_freq = '1' then
                                nextState <= locate_max_freq;

                        else
                                nextState <= update_stats;

                        end if;

                when locate_max_freq =>
                        if element_counter = "1111111" then
                                nextState <= rank_masses;

                        else
                                nextState <= locate_max_freq;

                        end if;


                when rank_masses =>
                        if evaluate_mass = '0' then
                                nextState <= update_stats;

                        else
                                nextState <= rank_masses;

                        end if;

                when others =>


        end case;

end process;

process(enb,clk)
begin


        if rst = '1' then
                currState <= reset;
        elsif rising_edge(clk) then

        if (enb = '1') then

                currState <= nextState;
                pipe_mass_valid <= mass_valid;
                matching_mass <= matching_stages;
```

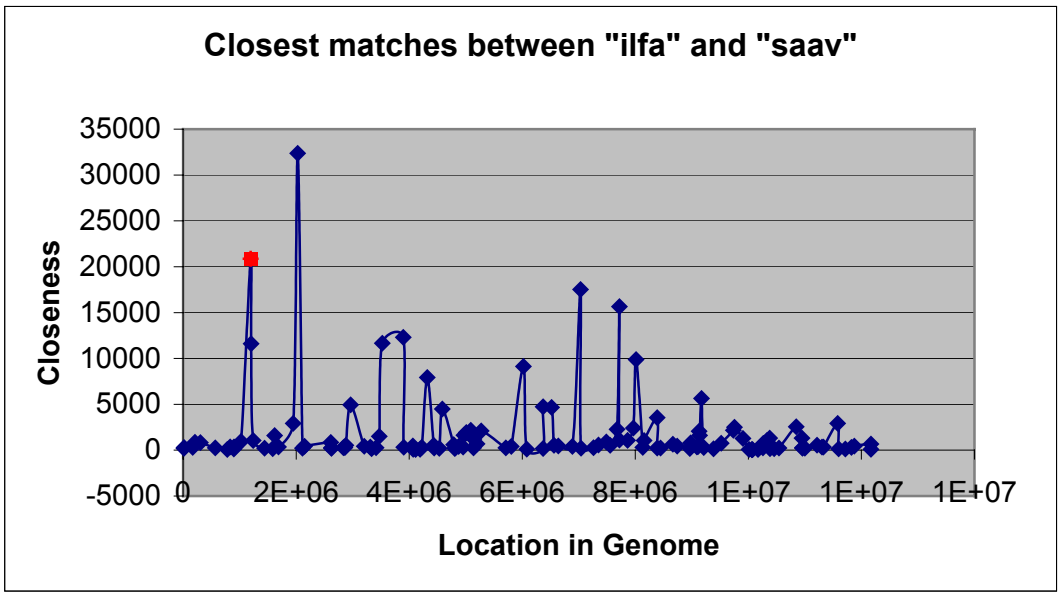
```
logadder_pipe <= (others => '0');


logadder_pipe(64 to 119) <= logadder_pipe(8 to 63);
logadder_pipe(120 to 167) <= logadder_pipe(72 to 119);
logadder_pipe(168 to 207) <= logadder_pipe(128 to 167);
logadder_pipe(208 to 239) <= logadder_pipe(176 to 207);
logadder_pipe(240 to 263) <= logadder_pipe(216 to 239);
logadder_pipe(264 to 279) <= logadder_pipe(248 to 263);
logadder_pipe(280 to 287) <= logadder_pipe(272 to 279);


for i in 0 to num_bins-1 loop
    addr(i*8 to (i*8)+7) <= rev_increment_range(i) & rev_increment_range(i+128) & rev_increment_range(i+(2*128)) & rev_increment_range(i+(3*128)) & rev_increment_range(i+(4*128)) & rev_increment_range(i+(5*128)) & rev_increment_range(i+(6*128)) & rev_increment_range(i+(7*128) );

end loop;


for i in 1 to num_stages-2 loop
    log_val_accum(i*num_freq_bits to (i*num_freq_bits)+(num_freq_bits-1)) <= log_val_accum((i-1)*num_freq_bits to ((i-1)*num_freq_bits)+(num_freq_bits-1)) + log_val_stages( (i+1)*num_freq_bits to ((i+1)*num_freq_bits) + (num_freq_bits-1) ) ;
end loop;
    log_val_accum( (num_stages-1)*num_freq_bits to ( (num_stages-1)*num_freq_bits)+(num_freq_bits-1)) <= log_val_accum((num_stages-2)*num_freq_bits to ((num_stages-2)*num_freq_bits)+(num_freq_bits-1)) + log_val_accum( (num_stages-1)*num_freq_bits to ( (num_stages-1)*num_freq_bits)+(num_freq_bits-1)) ;
    Pi_f <= log_val_accum( (num_stages-1)*num_freq_bits to ( (num_stages-1)*num_freq_bits)+(num_freq_bits-1));

frequency_pipeline <= (others => '0');




case (currState) is


when reset =>
    frequency <= (others => '0');
    low_freq_range <= (others => '0');
    frequency_pipeline <= (others => '0');
    log_val_accum <= (others => '0');
--  logadder_pipe <= (others => '0');

when update_stats =>
    hist_max_freq_reg <= (others => '0');
    for i in 0 to num_bins-1 loop

    saved_freq <= frequency;

        if evaluate_mass = '0' then
            frequency( i*num_freq_bits to (i*num_freq_bits) + num_freq_bits-1 ) <= frequency( i*num_freq_bits to (i*num_freq_bits) + num_freq_bits-1 ) + increment_amount(i*4 to (i*4)+3);

            log_val_accum <= (others => '0');
```

```
                                        else
                                                for i in 0 to num_stages-1 loop
                                                        saved_frequency_table <= frequency;

                                                end loop;

                                                frequency <= (others => '0');


                                        end if;

                                end loop;

                        when locate_max_freq =>

                                hist_max_freq <= hist_max_freq_reg;
                                element_counter <= element_counter+1;
                                if (saved_freq(0 to num_freq_bits-1) >= hist_max_freq_reg) then
                                        hist_max_freq_reg <= saved_freq(0 to num_freq_bits-1);
                                end if;

                                for i in 0 to num_bins-2 loop
                                        saved_freq(i*(num_freq_bits)                    to
(i*(num_freq_bits)+num_freq_bits-1))    <=      saved_freq((i+1)*(num_freq_bits)        to
((i+1)*(num_freq_bits)+num_freq_bits-1) ) ;
                                end loop;


                        when rank_masses =>



                                temp_test <= (others=> '0');
                                if evaluate_mass = '1' then

                                        for i in 0 to num_stages-1 loop
                                        if matching_mass( (num_stages-1) - i) = '1' then
                                                for j in 0 to num_bins-1 loop
                                                        if increment_range( (i*num_bins) + j )
= '1' then
                                                                logadder_pipe(
i*num_freq_bits to (i*num_freq_bits + (num_freq_bits-1))  ) <= saved_frequency_table( (127-j)*num_freq_bits to
((127-j)*num_freq_bits)+ (num_freq_bits-1));
--                                                              temp_test(    i*num_freq_bits
to (i*num_freq_bits + (num_freq_bits-1))  ) <= "01001101";
                                                        end if;
                                                end loop;
                                        end if;
                                        end loop;

                                end if;


                        when others =>

                end case;

        end if;
        end if;

end process;
```

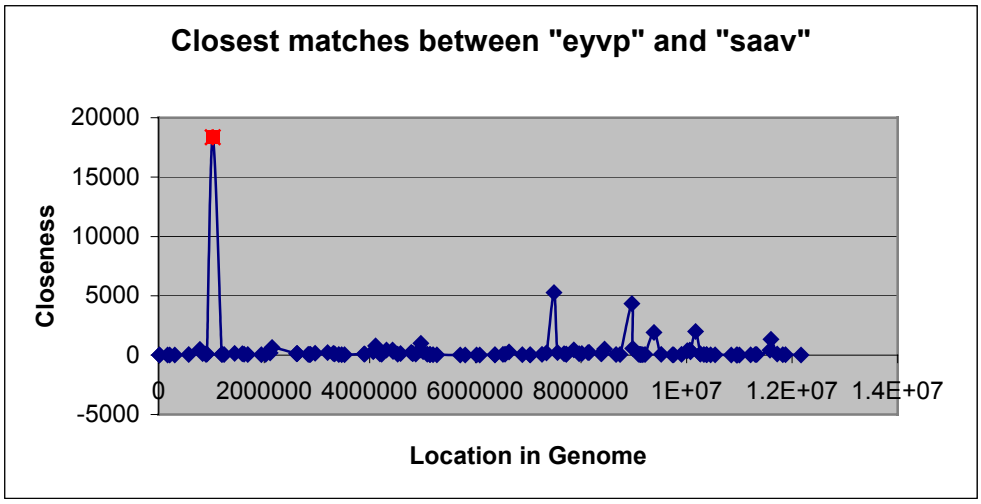
```
end mod_stats;
```

Appendix C. Scoring and Distance Results for Sample Peptides

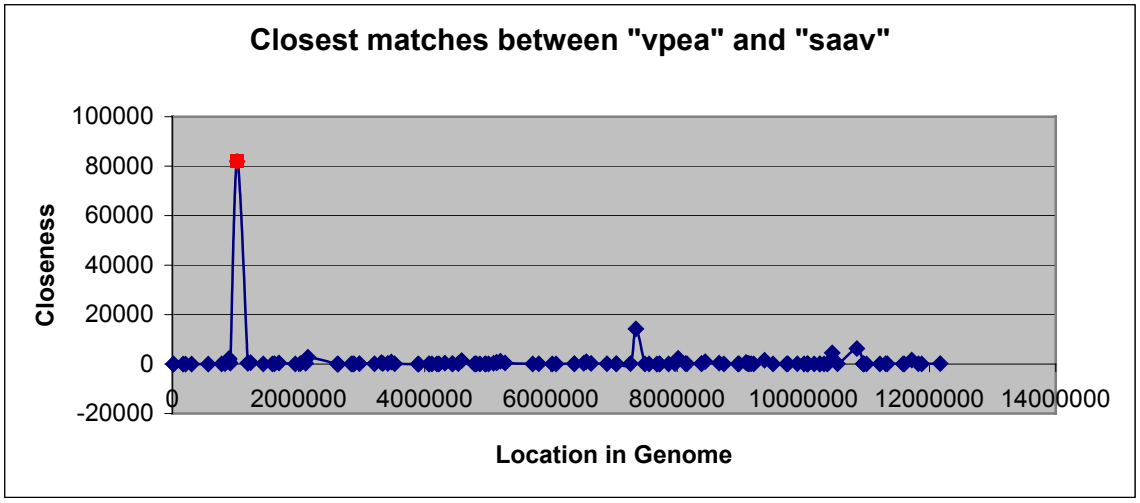
1. Results for GDP Dissociation Inhibitor Peptides



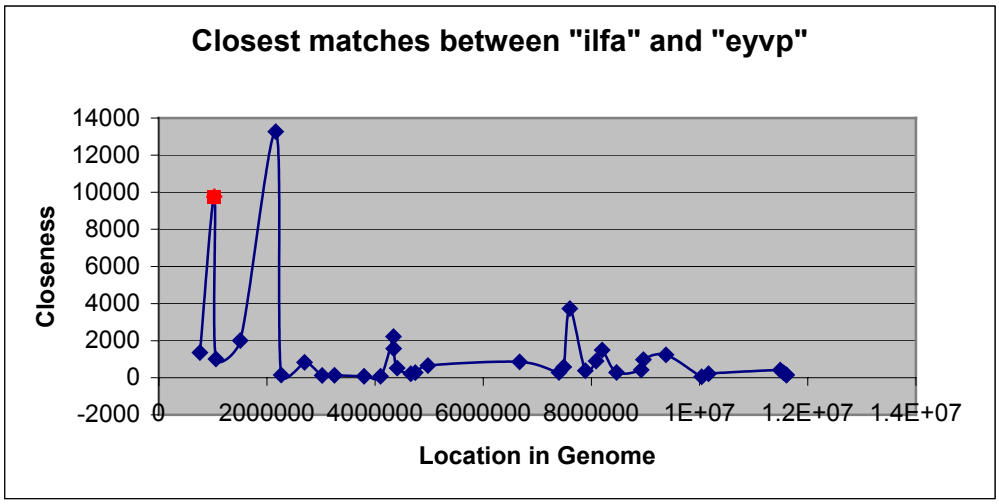
The true hit (square marker) is ranked 2nd of 128 hits.



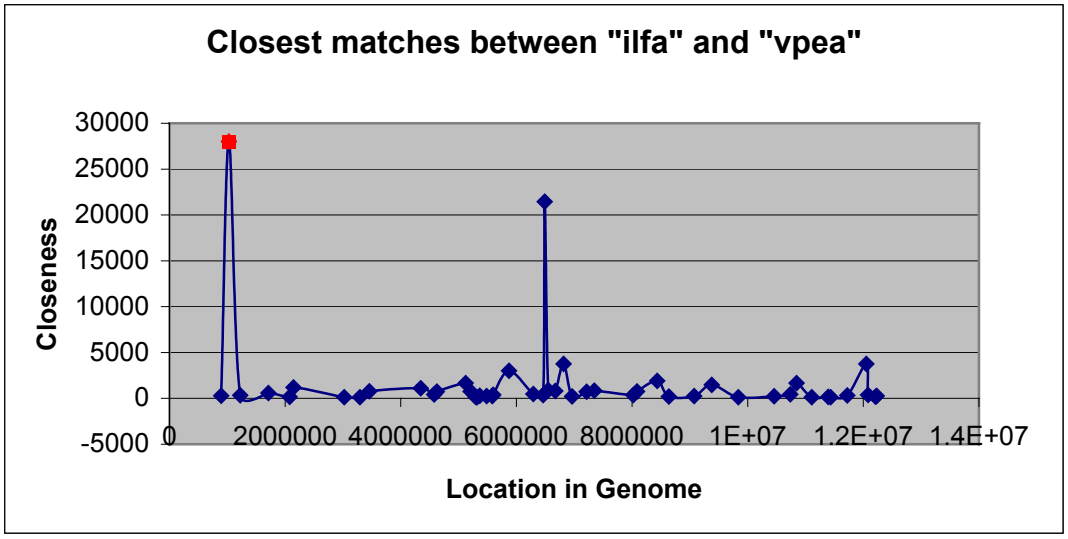
The true hit (square marker) is ranked 1st of 128 hits.



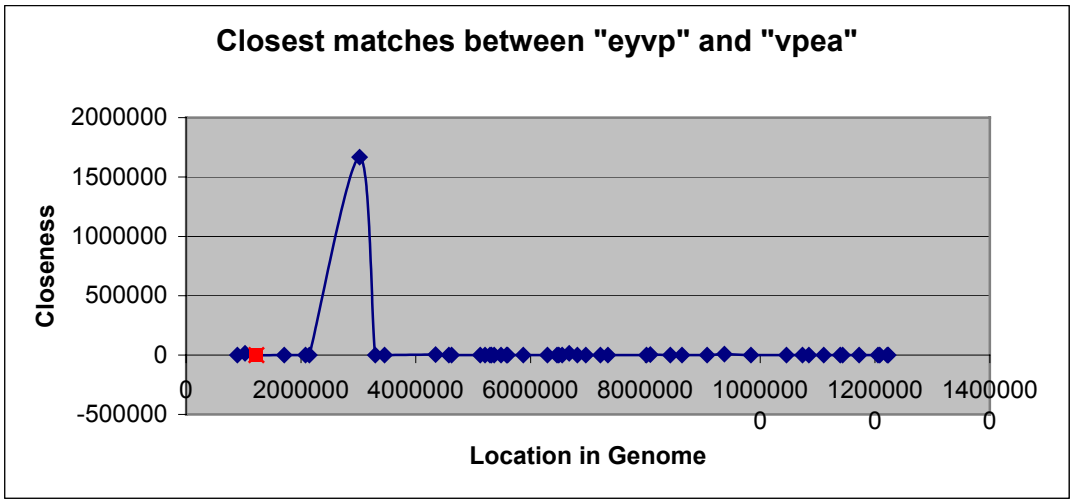
The true hit (square marker) is ranked 1st of 128 hits.



The true hit (square marker) is ranked 2nd of 34 hits. (The true hits are spaced 1024 bases apart, but there are two false positives that are 754 bases apart).



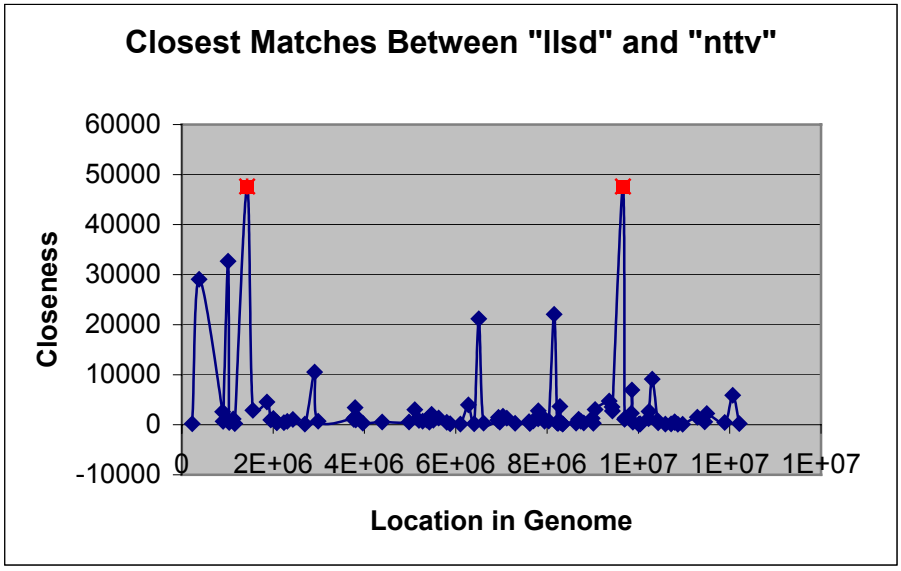
The true hit (square marker) is ranked 1st of 48 hits.



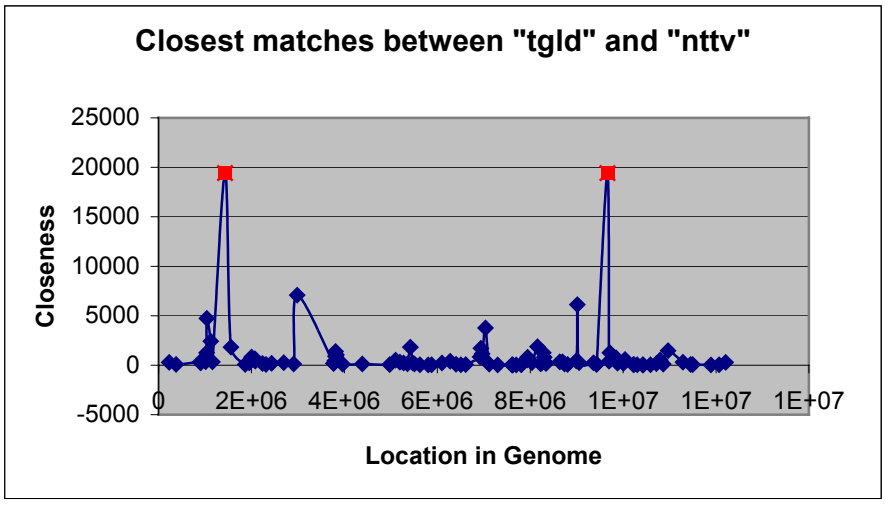
The true hit (square marker) is ranked 2nd of 48 hits (The true hits are spaced 667 bases apart, but there are two false positives that are 6 bases apart).

2. Results for Heat Shock Protein 70 Peptides

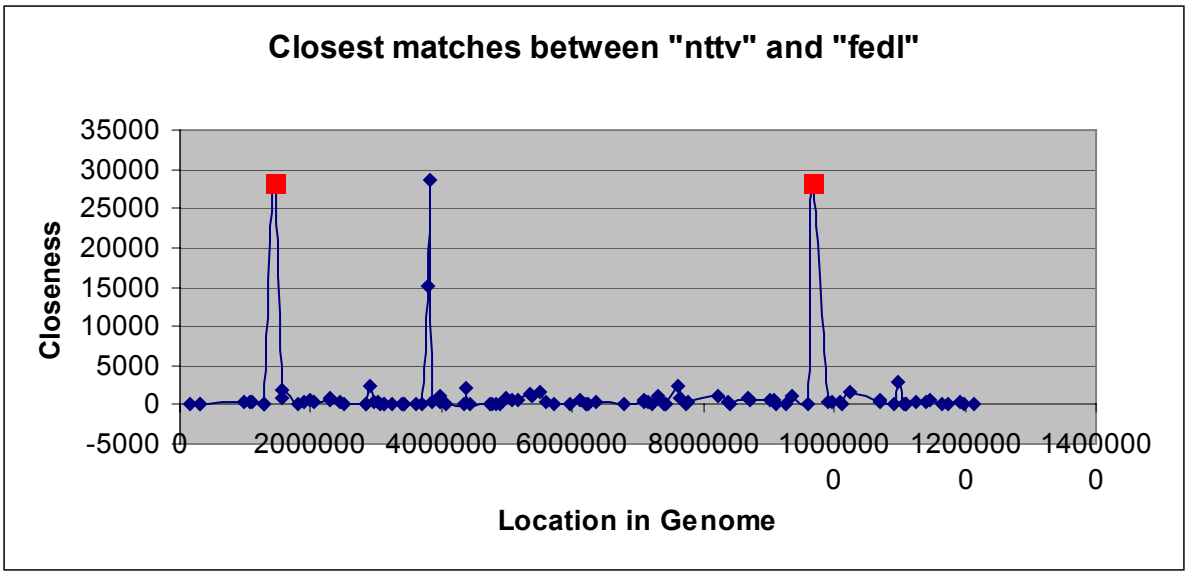
As mentioned in Chapter 4, HSP70 has two subfamilies that are highly similar, thus it is hard to distinguish between the protein and its homologue using *closeness* as a measure. However the score from the scoring unit can always be used to distinguish the subfamily in the sample.



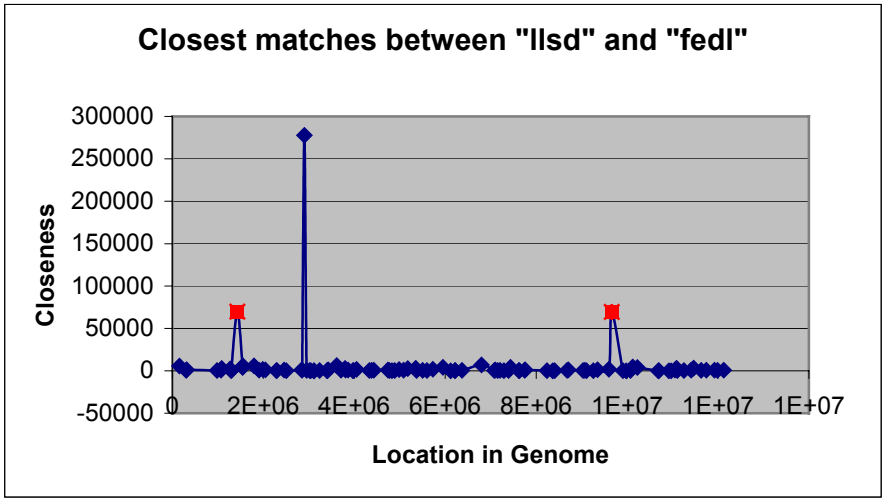
The true hit and its homologue (square marker) are ranked 1st of 105 hits.



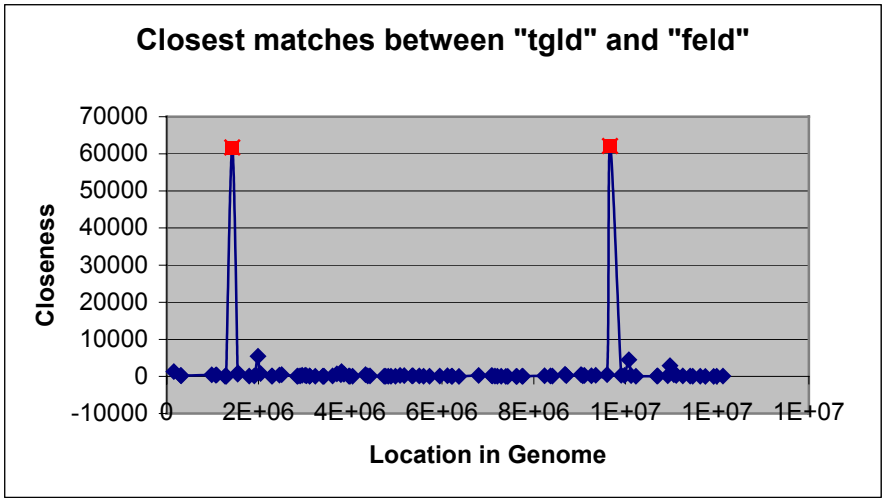
The true hit and its homologue (square marker) are ranked 1st of 105 hits.



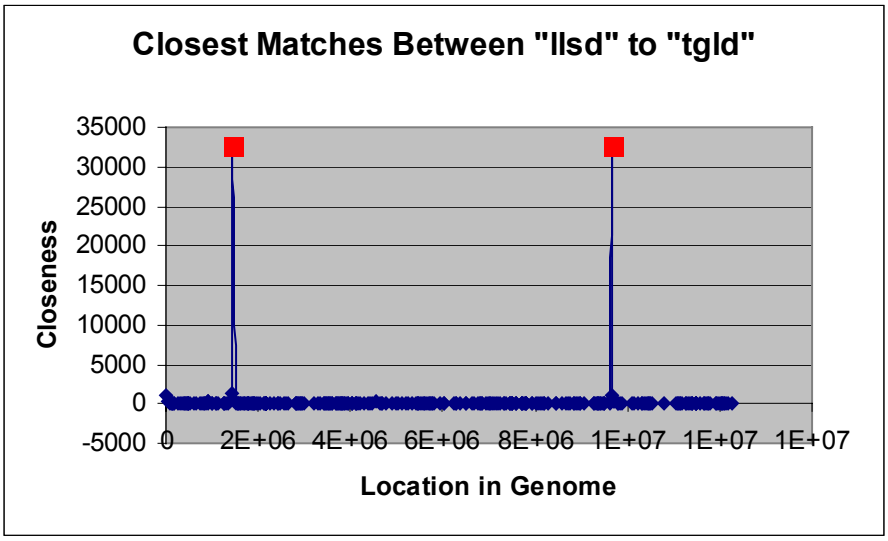
The true hit and its homologue (square marker) are ranked 2nd of 104 hits. (The true hits 353 bases apart, but there are two false positives that are 350 bases apart)



The true hit and its homologue (square marker) are ranked 2nd of 104 hits. (The true hits are 143 bases apart, but there are two false positives that are 36 bases apart)



The true hit and its homologue (square marker) are ranked 1st of 104 hits.



The true hit and its homologue (square marker) are ranked 1st of 306 hits.

Appendix D. Precursor Ion Scan (PIS) Masses

The following values (in Daltons) were used to obtain the results in Chapter 4.

453.17	552.57	624.12	688.01	758.49	822.42	924.92	1032.42	1112.46	1226.14
459.11	552.75	624.18	688.22	758.54	824.14	929.41	1035.94	1112.48	1226.49
459.18	556.92	624.95	689.69	761.49	831.04	937.26	1041.68	1115.52	1232.64
463.11	557.1	625.96	692.35	769.74	831.06	944.51	1041.69	1117.49	1237.18
463.13	561.76	633.67	694.42	772.26	838.69	945.75	1050.67	1119.22	1242.72
464.04	564.43	638.24	696.36	775.95	838.73	948.16	1050.67	1125.57	1242.77
464.1	567.35	639.31	698.11	777.47	839.54	948.23	1053.72	1126.47	1247.38
464.1	569.12	639.97	699.52	777.64	842.54	952.69	1053.88	1131.73	1247.86
464.12	576.44	640.16	702.75	783.19	842.56	962.37	1056.25	1131.76	1256.67
488.33	577.53	640.62	708.48	783.26	847.76	962.4	1056.64	1137.67	1260.67
497.41	577.71	643.65	709.69	785.07	847.96	962.49	1057.6	1147.4	1260.7
502.94	582.04	643.7	712.19	785.15	851.35	965.22	1062.68	1156.42	1265.78
503.03	583.57	649.74	712.5	785.81	855.52	965.33	1062.7	1157.51	1270.63
503.06	584.71	650.31	714.61	788.48	865.72	965.41	1062.9	1158.55	1270.63
503.08	584.74	655.43	714.63	788.51	865.72	965.42	1064.19	1167.59	1277.5
503.08	590.98	657.1	716.22	792.43	865.74	966.55	1065.45	1167.64	1278.67
503.39	591.5	659.24	720.95	792.69	865.79	976.07	1067.52	1169.59	1278.68
504.16	591.58	664.52	722.19	798.06	868.01	986.1	1068.56	1181.69	1284.53
505.15	592.69	664.8	722.41	798.42	871.59	990.52	1076.29	1181.78	1296.56
508.69	593.06	665.45	722.44	798.69	872.8	1000.39	1077.61	1185.82	1314.57
511.93	593.63	665.71	722.63	799.03	873.47	1000.48	1078.48	1185.83	1316.68
517.57	593.68	672.61	723.18	804.64	873.52	1002.49	1078.51	1190.67	1324.98
520.05	593.96	672.71	727.32	804.85	875.44	1004.77	1080.77	1191.48	1343.66
520.07	595.38	673.57	729.4	806.33	875.48	1006.65	1080.97	1199.77	1357.45
521	596.17	674.46	730.61	807.04	882.45	1006.7	1082.65	1205.45	1359.44
521	596.32	676.69	730.71	807.22	882.46	1008.59	1082.67	1206.73	1369.56
521.4	596.72	678.39	730.75	807.3	885.45	1011.04	1084.17	1209.96	1371.64
521.76	606.95	678.46	730.86	807.56	886.06	1011.19	1084.2	1210.01	1375.77
526.07	608.43	678.48	731.54	812.49	886.31	1013.69	1084.63	1210.03	1375.82
527.74	608.43	682.44	736.46	812.51	891.5	1014.12	1088.4	1210.23	1377.61
527.74	610.02	682.53	736.55	812.72	891.5	1014.47	1088.95	1210.28	1383.37
531.95	610.07	683.72	740.21	815.41	901.45	1020.78	1090.61	1213.44	1384.63

532.1	610.1	684.01	740.7	816.34	905.75	1021.7	1090.67	1214.66	1386.25
534.71	610.17	684.04	741.71	816.66	905.75	1022.19	1091.02	1218.78	1392.41
538.06	611.38	686.56	741.73	817.48	907.44	1022.95	1093	1218.79	1409.32
538.11	620.66	687.27	744.57	817.52	917.78	1023.29	1097.96	1220.15	1419.51
547	621.71	687.45	747.8	821.31	919.41	1024.31	1098.55	1220.98	1424.64
547.04	622.04	687.62	757.93	821.4	922.41	1028.24	1101.42	1224.15	
550.04	622.18	687.93	758	822.39	922.47	1032.25	1102.21	1224.6	