IMPLEMENTAÇÃO EM HARDWARE DE UM ACELERADOR HIBRIDO VITERBI-PLAN7/ALGORITMO DAS DIVERGÊNCIAS PARA COMPARAÇÃO DE PROTEINAS

Juan Fernando Eusse Giraldo

ORIENTADOR: RICARDO PEZZUOL JACOBI

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPGENE.DM - 403A/09
BRASÍLIA/DF: 19 DE NOVEMBRO – 2009

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

IMPLEMENTAÇÃO EM HARDWARE DE UM ACELERADOR HIBRIDO VITERBI-PLAN7/ALGORITMO DAS DIVERGÊNCIAS PARA COMPARAÇÃO DE PROTEINAS

JUAN FERNANDO EUSSE GIRALDO

ORIENTADOR: RICARDO PEZZUOL JACOBI

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPGENE.DM - 403A/09

BRASÍLIA/DF: NOVEMBRO – 2009

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

IMPLEMENTAÇÃO EM HARDWARE DE UM ACELERADOR HIBRIDO VITERBI-PLAN7/ALGORITMO DAS DIVERGÊNCIAS PARA COMPARAÇÃO DE PROTEINAS

JUAN FERNANDO EUSSE GIRALDO

DISSERTAÇÃO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE.

APROVADO POR:

_____

Prof. Ricardo Pezzuol Jacobi Ph.D (CIC/UnB)
(Orientador)

_____

Prof. Ivan Saraiva Silva Ph. D (DIMAP/UFRN)
(Examinador Externo)

_____

Prof. Carlos Llanos Ph. D (ENM/UnB)
(Examinador Externo)

BRASILIA/DF, 16 DE NOVEMBRO DE 2009

FICHA CATALOGRÁFICA

EUSSE GIRALDO, JUAN FERNANDO
Implementação Em Hardware De Um Acelerador Hibrido Viterbi-Plan7/Algoritmo Das Divergências Para Comparação De Proteínas.
xvii, 225p., 210 x 297 mm (ENE/FT/UnB, Mestre, Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia.
Departamento de Engenharia Elétrica.
1. Comparação de Proteínas 2.Aceleração
3. Divergências 4.Viterbi – Plan7
5. VHDL 6. FPGA
I. ENE/FT/UnB II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

EUSSE., J. F. (2009). Implementação Em Hardware De Um Acelerador Hibrido Viterbi-Plan7/Algoritmo Das Divergências Para Comparação De Proteínas. Dissertação de Mestrado em Engenharia Elétrica, Publicação PPGENE.DM-403A/09, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 219p.

CESSÃO DE DIREITOS

AUTOR: Juan Fernando Eusse Giraldo.

TÍTULO: Implementação Em Hardware De Um Acelerador Hibrido Viterbi-Plan7/Algoritmo Das Divergências Para Comparação De Proteínas.

GRAU: Mestre ANO: 2009

_____

Juan Fernando Eusse Giraldo
SCLN 407 Bloco C, Asa Norte
70855-530 Brasília – DF – Brasil.

AGRADECIMENTOS

Ao meu orientador, professor Ricardo Pezzuol Jacobi, pela oportunidade de desenvolver o meu conhecimento sob a sua orientação, pela paciência e pela ajuda prestada durante o tempo de estudo.

A professora Nahri Moreano, suas explicações e sua inteligência fizeram possível a feliz terminação do trabalho.

A professora Alba Cristina Magalhaes de Melo pelas dicas que ajudaram a feliz conclusão deste trabalho.

A minha família pelo apoio neste passo tão importante da minha vida.

Para Marcela por estar siempre a mi lado, por amarme tal como soy y por que se siempre estaremos juntos.

Aos colegas do LAICO/LDCI/LTSD pelas suas contribuições e pelo ótimo ambiente de trabalho.

Para os meus amigos especialmente para o Augusto, o Edcelio, o Vladimir, o Sergio, o Diego, o Sebastian e a Juliana. Sem seu apoio e carinho constante não seria quem eu sou.

Para Torres, quien sin darse cuenta cambio la vida de muchas personas y que nunca será olvidado.

ABSTRACT

HARDWARE IMPLEMENTATION OF A HYBRID VITERBI-PLAN7/DIVERGENCE PROTEIN COMPARISSON ACCELERATOR IN VHDL

Author: Juan Fernando Eusse Giraldo

Advisor: Ricardo Pezzuol Jacobi

Electrical Engeneering Graduate Program

Brasília, November 2009

Hidden Markov Models are a powerful tool for protein organization and identification because they allow identifying and classifying highly representative structures and functional units inside the amino acid chains that form them. The Viterbi algorithm is one of the most used algorithms in protein comparison and identification using Hidden Markov Models, and is implemented inside the open source software HMMER [1][2], which is widely used among the scientific community. Due to the exponential growth in the size of protein databases in the past years, the necessity to accelerate software execution to reduce comparison and search times rose. In this master thesis, a hardware accelerator is implemented in VHDL in order to reduce those processing times in the protein comparison and search processes. The implemented accelerator uses a new algorithm which enables the system (Hardware+Software) to economize processing time by reducing the number of calculations needed to perform a comparison. The accelerator not only produces the similarity score for a sequence when compared against a profileHMM but also produces the parameters to limit the region of the Dynamic Programming Matrices that must be reprocessed to generate the alignment. The implemented accelerator produces a maximum gain of up to 182 times when compared to unaccelerated software. A new performance measurement strategy is introduced in this work, which not only takes into account the

acceleration achieved by the hardware, but also the post-processing stages that follows hardware made comparisons.

RESUMO

IMPLEMENTAÇÃO EM HARDWARE DE UM ACELERADOR HIBRIDO VITERBI-PLAN7/ALGORITMO DAS DIVERGÊNCIAS PARA COMPARAÇÃO DE PROTEINAS

Autor: Juan Fernando Eusse Giraldo

Orientador: Ricardo Pezzuol Jacobi

Programa de Pós-graduação em Engenharia Elétrica

Brasília, Novembro de 2009

Os Modelos Ocultos de Markov (*HMM – Hidden Markov Models*) constituem uma poderosa ferramenta para mapeamento e organização de proteínas, uma vez que permitem reconhecer estruturas altamente representativas e unidades funcionais dentro das cadeias de aminoácidos que as conformam. O Viterbi é um dos principais algoritmos para comparação e identificação de proteínas (sequências de aminoácidos) baseados em HMM, e é implementado dentro do software livre HMMER [1][2], muito utilizado na comunidade científica. Nos últimos anos, devido ao crescimento exponencial das bases de dados que armazenam proteínas, surge a necessidade de acelerar a execução do software para reduzir os tempos de processamento dos algoritmos de comparação. Neste trabalho de mestrado, é realizada a aceleração do software HMMER para alinhamento de sequências biológicas através da implementação de um acelerador em hardware. O acelerador proposto utiliza um novo algoritmo chamado de Algoritmo das Divergências, o qual permite ao sistema completo (Hardware+Software) economizar uma grande quantidade de cálculos para gerar os alinhamentos de proteínas. O Hardware produz a medida de similaridade da proteína com o modelo HMM e os índices inicial e final da porção de interesse da sequência de aminoácidos como uma primeira etapa de filtragem. Isto, quando gerado pelo acelerador,

significa uma economia de processamento adicional para o software, o qual tem que reprocessar dita região para gerar o alinhamento da sequência com o profileHMM, e contribui com a aceleração da execução do algoritmo. O Acelerador atinge ganhos de até 182x quando comparado com o software não acelerado. Além disso, o trabalho propõe uma nova medida para a comparação do desempenho e realiza medições exatas acerca da aceleração atingida ao integrar o acelerador ao fluxo de execução do software.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIMBOLOS, NOMENCLATURA E ABREVIAÇÕES

ADN	Acedo Desoxirribonucléico
VHDL	Linguagem de Descrição de Hardware (VHSIC Hardware Description Language)
HMM	Modelo Oculto de Markov (*Hidden Markov Model*)
FSA	Autômato de Estados Finito (*Finite State Automaton*)
GPU	Unidade de Processamento Gráfico (*Graphic Processing Unit*)
FPGA	Arranjo de Portas Programável no Campo (*Field Programable Gate Array*)
RAM	Memória de Acesso Randômico (*Random Access Memory*)
FIFO	Memória *First-In First-Out*
EP	Elemento de Processamento
AS	Arranjo Sistólico
MHz	Um milhão de Hertz
GHz	Um bilhão de Hertz
GB	2^{32} bytes
FSM	Máquina de Estados Finita (*Finite State Machine*)
UC	Unidade de Controle
CUPS	Células Atualizadas por Segundo (*Cell Updates per Second*)
MCUPS	Um milhão de CUPS
GCUPS	Um bilhão de CUPS
MBpS	Um milhão de bytes por segundo

1- INTRODUÇÃO

Nos últimos anos, a comunidade científica tem visto como os avanços tecnológicos permitiram a identificação - com uma freqüência cada vez maior - de novas proteínas, as quais são integradas às bases de dados existentes [3][4]. Com o crescimento exponencial dessas bases de dados, os algoritmos de comparação [5][6] de proteínas levam cada vez mais tempo para ser executados [7-16], e faz-se necessária a aceleração desses algoritmos para melhorar o desempenho dos processos de pesquisa, nos quais a comparação de proteínas dentro das bases de dados é uma tarefa essencial e repetitiva. Os Modelos Ocultos de Markov (*Hidden Markov Models - HMM*) são amplamente usados pelos algoritmos de comparação de proteínas. Dentre estes destaca-se o Viterbi-Plan7 [1], sendo sua implementação em software [17] uma das mais utilizadas pelos pesquisadores na atualidade. O desenvolvimento de aceleradores implementados em hardware para o algoritmo de Viterbi [7-16] contribuiu para reduzir o sue tempo de execução, através da migração das tarefas mais computacionalmente intensivas - o cálculo das matrizes de programação dinâmica - para hardware dedicado. O módulo em hardware é usualmente utilizado como uma primeira etapa de filtragem para descartar sequências com alinhamentos ruins, as quais representam aproximadamente o 99% do tempo de processamento.

Além da aceleração por hardware, uma outra estratégia para reduzir o tempo de processamento das matrizes é apresentada em [19]. Um novo algoritmo, proposto inicialmente para comparação de sequências de ADN, chamado de Algoritmo das Divergências [19] calcula, em software, a faixa que delimita a região das matrizes de programação dinâmica na qual se encontra o alinhamento local ótimo (Smith-Waterman) entre duas sequências. Ao obter essa faixa, que geralmente é uma fração bem pequena das matrizes completas, a recuperação do alinhamento ocorre, em media, de maneira bastante rápida [19]. O Algoritmo das Divergências foi adaptado para o problema da comparação de proteínas [37][38] e sua validade como um acelerador de processamento foi testada chegando à conclusão de que a inclusão dele no processo de comparação através de HMMs somente é viável se o algoritmo for implementado em paralelo.

A presente dissertação de mestrado realiza a implementação em hardware, mais especificamente em VHDL, do Algoritmo Viterbi-Plan7, e acrescenta ao hardware proposto uma implementação da estratégia proposta no Algoritmo das Divergências. Os objetivos são testar a validade do algoritmo [19] na hora de fazer comparação de proteínas (sequências de aminoácidos) e fornecer aos pesquisadores da área de biologia molecular uma ferramenta de aceleração que permita agilizar seus projetos de pesquisa. Entre as principais contribuições do trabalho podemos citar:

- A validação da modificação feita ao Algoritmo das Divergências como uma ferramenta de aceleração dos tempos de execução das operações de comparação de proteínas.
- A implementação de um acelerador em VHDL, o qual funciona como uma primeira etapa de filtragem para o software e ao mesmo tempo calcula as coordenadas para o Algoritmo das Divergências.
- A caracterização do desempenho do software HMMER [17], para achar estatísticas de tempos de execução e de pontuações para sequências reais.
- A codificação de um *testbench* automatizado para a simulação e caracterização do desempenho do acelerador.
- A introdução de uma medida de desempenho que pondera todos or atrasos do sistema, para a medição da aceleração atingida tanto pelo hardware proposto quanto pelo processamento de uma região limitada das matrizes de programação dinâmica.

O trabalho foi dividido em seis capítulos. No Capítulo dois, são apresentados os objetivos do trabalho. No Capítulo três foi feita uma revisão dos conceitos de análise de sequências biológicas e os algoritmos usados nessa análise, além de investigar os trabalhos correlatos. No Capítulo quatro, é feita a proposta do acelerador já com o Algoritmo das Divergências integrado, bem como são feitas as propostas para a modificação do fluxo de execução da aplicação e as medidas que serão utilizadas para realizar a avaliação de desempenho. No Capítulo cinco, são explicadas a metodologia de projeto e a implementação do acelerador e do seu *testbench*, e são apresentados os resultados experimentais obtidos dentro do trabalho de mestrado. Finalmente, no Capítulo seis, são apresentadas as conclusões e as recomendações para trabalhos futuros.

2 - OBJETIVOS

2.1 - OBJETIVO GERAL

Projetar, implementar e testar um acelerador em hardware que execute o Algoritmo Plan7 para a comparação de proteínas com Modelos Ocultos de Markov gerados a partir de alinhamentos feitos em famílias de proteínas, incluindo a geração dos coeficientes descritos no Algoritmo das Divergências; e inclur o acelerador no fluxo de execução do software HMMER para avaliar a sua utilidade no processo de aceleração do mesmo.

2.2 - OBJETIVOS ESPECÍFICOS

a. Analisar a estrutura do Algoritmo Plan7 para busca em Modelos Ocultos de Markov (HMMs) e do Algoritmo das Divergências, para determinar as possibilidades da implementação de um acelerador em hardware capaz de gerar tanto a pontuação de uma sequência *query* quanto os dados para o algoritmo das divergências.

b. Identificar as dependências de dados dentro dos algoritmos e adaptá-los para sua paralelização e posterior implementação em hardware.

c. Projetar um acelerador em hardware que realiza a execução do Algoritmo de Viterbi e das divergências, e que possa ser integrado dentro do fluxo de execução do software HMMER para acelerá-lo.

d. Realizar a implementação em VHDL do acelerador proposto, validar sua operação e obter seu desempenho através de um *testbench.*

e. Desenvolver um *testbench* automático para os processos de test do acelerador proposto.

f. Desenvolver medidas de desempenho que descrevam de uma maneira apropriada a aceleração atingida ao integrar o acelerador proposto ao fluxo de execução do software.

g. Recolher as medidas de desempenho atingidas pelo acelerador, e calcular a aceleração do sistema.

h. Analisar o desempenho atingido pelo sistema ao incorporar o acelerador, e observar se a inclusão dele ajuda a incrementar o desempenho atingido pelo software não acelerado.

3 - REVISÃO BIBLIOGRÁFICA

3.1 - CONCEITOS BÁSICOS PARA ANÁLISE DE SEQUÊNCIAS BIOLÓGICAS

Uma sequência biológica consiste em um agrupamento linear de um alfabeto de elementos constitutivos (geralmente aminoácidos) que descrevem a estrutura química de moléculas biológicas [10]. É de interesse para os cientistas não só encontrar estas sequências através de ferramentas como espectrômetros, mas também analisar as semelhanças entre elas, para achar as sequências relacionadas e determinar se essa relação é devida a um processo evolutivo, que geralmente conserva traços característicos para uma família de sequências.

Devido à alta taxa com que estas novas sequências são encontradas, há um crescimento exponencial das bases de dados onde elas são armazenadas [3][4], e uma comparação de uma sequência com aquelas presentes na base de dados representa um grande esforço computacional. As Figuras 3.1 e 3.2 apresentam o crescimento das duas principais bases de dados de armazenamento de sequências biológicas (proteínas).

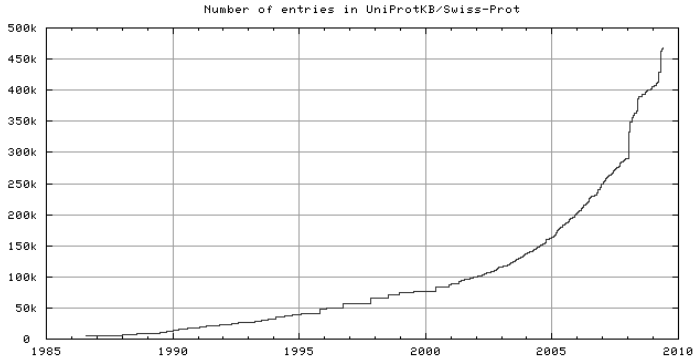


Figura 3.1: Crescimento da base de dados UniProtKB/Swiss-prot [3].

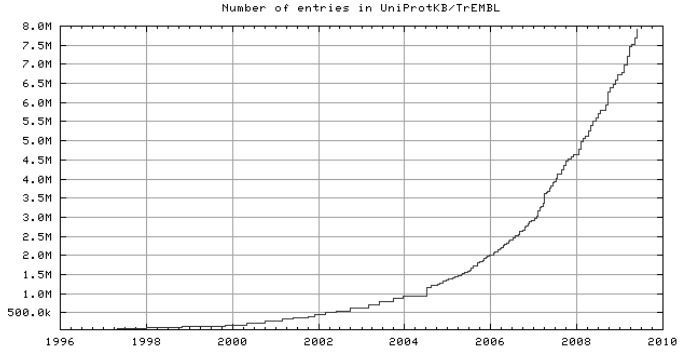


Figura 3.2: Crescimento da base de dados UniProtKB/TrEMBL [3].

Este ritmo de crescimento exponencial e o crescente interesse na análise deste tipo de sequências por parte dos biólogos geram uma grande necessidade (e um grande interesse) no desenvolvimento de ferramentas que automatizam o trabalho repetitivo, tanto em software [1][2][20-22], quanto em hardware [7-16][23]. Porém, as ferramentas desenvolvidas têm evolução relativamente lenta, em comparação com o crescimento das bases de dados, e novas soluções devem ser constantemente desenvolvidas para alcançar os requerimentos de desempenho.

3.1.1 - Alinhamentos

A melhor maneira de determinar a função biológica de uma sequência (proteína ou ADN) dentro de um organismo é através de experimentação direta [5], mas é de grande utilidade ter uma idéia aproximada de algumas propriedades da sequência antes de começar essa experimentação. A maioria das técnicas desenvolvidas para fazer comparação de sequências começa com uma sequência desconhecida (*query*) e compara-a com as bases de dados que armazenam famílias de sequências conhecidas para construir um alinhamento. A partir deste alinhamento podem ser achados segmentos da sequência que são similares à referência, o que pode significar similaridade funcional ou estrutural [24]. Os alinhamentos consistem em determinar quais porções de uma sequência são similares a porções de outra sequência ou a porções de uma família de sequências (unidades funcionais ou *motifs*), para identificar relações funcionais, estruturais ou mesmo evolutivas. A Figura 3.3 apresenta um alinhamento para 10 proteínas pertencentes à família I-set Imunoglobulina. Da figura pode se ver que o alinhamento serve para identificar os principais grupos funcionais da família de proteínas (*structure*), também pode se observar que os aminacidos que são comuns em todas as sequências (ou que não variam muito) são agrupados numa sequência especial chamada de consenso, a qual representa os grupos funcionais presentes em todas as proteínas da família.

```
structure:  ddd.....eeeeee.......fffffffff.......ggggggggggg.
1tlk        IDYDEEGNCSLTISEVCGDDDAKYTCKAVNSL-----GEATCTAELLVET
AXO1_RAT    SQTT----GNLYIARTNASDLGNYSCLATSHMDFSTKSVFSKFAQLNLAA
AXO1_RAT    VLA-----GDLRFSKLSLEDSGMYQCVAENKH-----GTIYASAELAVQA
AXO1_RAT    VTSD----GTLIIRNISRSDEGKYTCFAENFM-----GKANSTGILSVRD
AXO1_RAT    AKETI---GDLTILNAHVRHGGKYTCMAQTVV-----DGTSKEATVLVRG
NCA2_HUMAN  VLSN----NYLQIRGIKKTDEGTYRCEGRILARG---EINFKDIQVIVNV
NCA2_HUMAN  VVWNDDSSSTLTIYNANIDDAGIYKCVVTGEDG----SESEATVNVKIFQ
NCA2_HUMAN  FSDDSS---QLTIKKVDKNDEAEYICIAENKA-----GEQDATIHLKVFA
NRG_DROME   QGHYG---KSLVIRQTNFDDAGTYTCDVSNGVG----NAQSFSIILNVNS
NRG_DROME   KTND----NSLTIAKTMELDSGEYTCVARTRL-----DEATARANLIVQD
consensus:  ..........L.+..+...+.+.Y.C.................+.+.+..
```

Figura 3.3: Alinhamento múltiplo e sequência de consenso para 10 elementos da família I-set imunoglobulina.

Existem duas classes básicas de alinhamentos: locais e globais. Os alinhamentos globais alinham uma sequência, do inicio até o fim, com outra sequência ou com um conjunto de sequências. O maior problema dos alinhamentos globais é que eles tentam alinhar todos os elementos da sequência *query* com cada um dos elementos das sequências com as quais a mesma está sendo alinhada. Isto somente é eficiente para conjuntos de sequências muito similares e não engloba sequências que compartilham origens evolutivas, mas tiveram muitas mutações durante a sua evolução. Esta característica dos alinhamentos globais restringe a análise e pode ignorar características evolutivas de uma família de sequências, tais como trechos de elementos iguais, mas em diferentes posições, e que representam uma mesma característica funcional. Os alinhamentos locais procuram achar regiões de similaridades que possam representar características funcionais, evolutivas ou estruturais similares entre sequências muito diferentes e muito grandes. A Figura 3.4 mostra dois exemplos, um de alinhamento global e outro de um alinhamento local.

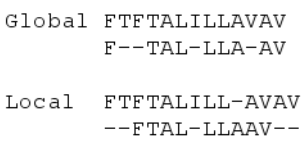


Figura 3.4: Exemplo de alinhamento local e global [25].

Os alinhamentos podem ser feitos entre duas sequências para estabelecer a similaridade entre elas, ou podem ser feitos entre um conjunto de sequências para achar segmentos do conjunto que são altamente similares, que podem ser de grande importância na hora de identificar comportamentos similares entre elas. Os segmentos similares achados em alinhamentos múltiplos são chamados *motifs*, e definem características de uma função bioquímica específica. Um exemplo é o *zinc finger motif* CXX(XX)CXXXXXXXXXXXXHXXXH, achado em uma família muito ampla de proteínas no ADN [36]. Os *motifs* são geralmente usados para identificar famílias de sequências relacionadas. Existe um grande número de técnicas e algoritmos para realizar alinhamentos entre duas ou mais sequências [5][6]. O foco do presente trabalho consiste na aceleração da busca do alinhamento ótimo de uma sequência com um modelo probabilístico de uma família de sequências (*profileHMM*), obtido a partir do alinhamento múltiplo conhecido dessa família de sequências, razão pela qual na próxima seção introduz os princípios da programação dinâmica e os principais algoritmos de comparação, necessários para entender o algoritmo de comparação.

3.1.2 - Programação dinâmica

A programação dinâmica é tanto um método de otimização matemática quanto uma técnica de programação, geralmente utilizada para dividir problemas complexos em partes mais simples que são executadas recorrentemente. Esses problemas complexos geralmente requerem soluções ótimas, onde é necessário maximizar/minimizar uma função. Para que um problema possa ser resolvido através de um algoritmo de programação dinâmica, ele deve possuir duas características: uma subestrutura ótima e subproblemas superpostos. A primeira característica significa que a solução do problema geral pode ser obtida como a combinação dos subproblemas que o constituem. A segunda característica quer dizer que o problema geral possui um número limitado de problemas, os quais são executados uma e outra vez, em vez de serem gerados novos subproblemas. Entre os exemplos mais comuns da programação dinâmica estão a solução para a recorrência de Fibonacci, a geração de matrizes balanceadas, o Algoritmo *System R* para consultas em bases de dados, problemas de processamento de imagens e problemas de alinhamento, que são o foco deste trabalho e que serão analisados na próxima seção. O Algoritmo 3.1 apresenta a solução de programação dinâmica para a recorrência de Fibonacci, onde os menores valores da série são calculados antes dos maiores, reduzindo o armazenamento a dois coeficientes, o atual e o anterior.

```
function fib(n)
   var previousFib := 0, currentFib := 1
   if n = 0
       return 0
   else if n = 1
       return 1
   repeat n − 1 times
      var newFib := previousFib + currentFib
      previousFib := currentFib
      currentFib  := newFib
return currentFib
```

Algoritmo 3.1: Solução de programação dinâmica para a recorrência de Fibonacci.

3.2 - ALGORITMOS PARA ALINHAMENTO DE PROTEÍNAS

Existem dois tipos de algoritmos para realizar alinhamentos de proteínas. Um deles toma uma sequência *query* e compara-a com todas as sequências de uma base de dados (uma de cada vez) para obter o melhor alinhamento (global ou local), e estabelece uma pontuação para o alinhamento obtido. Se a pontuação obtida for boa, então é altamente provável que a

sequência que está sendo comparada pertença à mesma família da sequência da base de dados contra a qual foi comparada. Este tipo é chamado de alinhamento de pares (*pairwise alingment*). O outro tipo de algoritmos tem como entradas alinhamentos múltiplos de famílias de proteínas que possuem características similares e gera um modelo probabilístico da família. A partir deste modelo probabilístico, e de algoritmos de programação dinâmica, uma sequência pode ser comparada com a família de proteínas e pode-se obter uma pontuação de quão similares elas são entre si. Esse último tipo de algoritmo é chamado de alinhamento múltiplo.

3.2.1 - Alinhamento de pares (*pairwise alingment*)

São alinhamentos feitos entre duas sequências para determinar se elas estão relacionadas. Neste tipo de alinhamentos, geralmente tem-se uma sequência que vai ser alinhada a outras sequências extraídas de uma base de dados. Depois de fazer o alinhamento, uma pontuação é calculada para o alinhamento para determinar se o mesmo se deu por casualidade ou se as duas sequências alinhadas estão verdadeiramente relacionadas. Para determinar o alinhamento ótimo e a pontuação desse alinhamento, um sistema de pontuação baseado na relação logarítmica entre as probabilidades de que um par de aminoácidos esteja alinhado e a probabilidade de que o par se apresente independentemente é desenvolvido. A Equação 3.1 apresenta esta relação, onde *a* e *b* são os aminoácidos das duas sequências, p_{ab} é a probabilidade dos aminoácidos aparecem ao mesmo tempo, q_a e q_b são as probabilidades independentes de cada aminoácido e *s(a,b)* é a razão logarítmica entre essas probabilidades. *s(a,b)* é chamada de razão *log-odds* e descreve a probabilidade de que *a* e *b* sejam achados alinhados em vez de serem achados não alinhados [5].

$$s(a,b) = \log\left(\frac{p_{ab}}{q_a q_b}\right) \tag{3.1}$$

Dentro deste modelo de pontuação, também são considerados fenômenos evolutivos como a substituição, inserção ou remoção de elementos nas sequências. Probabilidades são associadas aos fenômenos de substituição e penalizações são associadas aos fenômenos de inserção e remoção (*gaps*). Para cada par de elementos da sequência se determina a probabilidade de ocorrência e uma matriz de probabilidades. As filas da matriz estão conformadas pelos elementos da primeira sequência e as colunas da matriz são os

elementos da outra. Logo depois de obter a matriz de probabilidades, o algoritmo determina o melhor alinhamento dependendo do seu tipo (local ou global). A pontuação do alinhamento obtido é definida como a soma destes alinhamentos individuais ao longo dos elementos das duas sequências, como é apresentada na equação 3.2. Pontuações positivas indicam um bom alinhamento, enquanto pontuações negativas indicam alinhamentos não representativos.

$$S = \sum_i s(x_i, y_i) \tag{3.2}$$

A Figura 3.5 apresenta a matriz de probabilidades e o alinhamento obtido para duas sequências (HEAGAWGHEE e PAWHEAE) utilizando o Algoritmo de Needleman-Wunsch para achar alinhamentos globais [5].

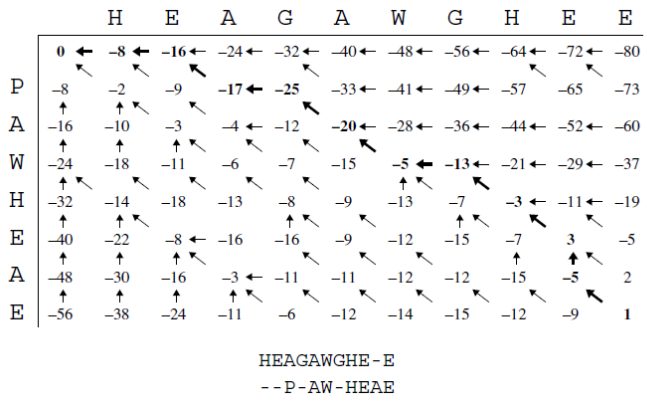


Figura 3.5: Matriz de probabilidades e alinhamento global para duas sequências.

3.2.1.1 - Algoritmo de Needleman-Wunsch (alinhamentos globais)

O Algoritmo de Needleman-Wunsch [26] é um algoritmo de programação dinâmica pensado para obter o alinhamento global ótimo entre duas sequências, ao obter alinhamentos ótimos para as subsequências que as conformam. Sejam ***x*** e ***y*** duas sequências a serem alinhadas, o algoritmo propõe a criação de uma matriz bi-dimensional onde cada linha de índice *i* está associada ao elemento *x(i)* e cada coluna de índice *j* está associado o elemento *y(j)*. *M(i,j)*, é a pontuação do melhor alinhamento entre os segmentos $x_{1..i}$ e $y_{1..j}$. O elemento *M(i,j)* da matriz é calculado baseado em três possibilidades:

- x_i pode estar alinhado com y_i.
- x_i pode estar alinhado com um *gap*.

- y_j pode estar alinhado com um gap.

M(i,j) é a maior pontuação produzida pelas três possibilidades. O algoritmo também guarda qual das três possibilidades foi a maior, para no final seguir o conjunto de ponteiros e produzir o alinhamento. A Figura 3.6 apresenta estas três possibilidades e a Equação 3.3 apresenta a pontuação que elas produzem.

$$M(i,j) = \max \begin{cases} M(i-1,j-1) + s(x\_i, y\_j) \\ M(i-1,j) - d \\ M(i,j-1) - d \end{cases} \tag{3.3}$$

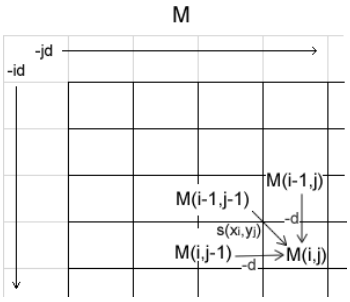


Figura 3.6: Geração do elemento *M(i,j)* da matriz de substituição M para as sequências *x* e *y*, onde $s(x_i,y_j)$ é a probabilidade de que os elementos estejam alinhados, e *d* é a penalização por um alinhamento com um gap.

O valor final de M é a pontuação para o alinhamento ótimo das duas sequências. Para se achar o alinhamento, basta seguir os ponteiros guardados pelo mesmo no cálculo da pontuação. O algoritmo original de Needleman-Wunsch penalizava todos os *gaps* de um agrupamento da mesma forma, o que está errado sob o ponto de vista evolutivo. Ele foi otimizado por Gotoh [27] para penalizar mais o primeiro *gap* do alinhamento do que os *gaps* restantes, o que modela de uma maneira mais adequada o comportamento biológico das proteínas.

3.2.1.2. Algoritmo de Smith-Waterman (alinhamentos locais)

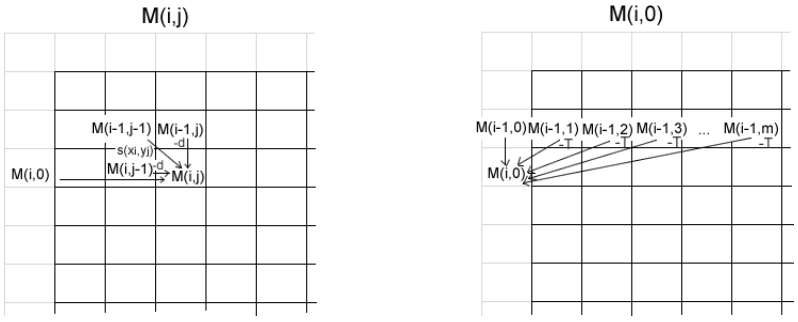
É um algoritmo usado para detectar similaridades entre segmentos das sequências que estão sendo comparadas (*x* e *y*). Esses segmentos podem indicar similaridade funcional ou estrutural entre as sequências, ou então uma origem evolutiva comum. O algoritmo original

procurava achar o melhor alinhamento local dentro da sequência, mas versões posteriores o modificaram para achar todos os alinhamentos locais com uma pontuação maior do que um valor mínimo *T* [5]. Seja *y* a sequência com o *motif*, e *x* a sequência onde o algoritmo vai procurá-lo. No alinhamento final, *x* estará dividido em seções que correspondem a partes de *x* que são similares a partes de *y*. As Equações 3.4, 3.5 e 3.6 apresentam a relação usada para achar os alinhamentos e as Figuras 3.7 a e b apresentam os valores possíveis para *M(i,j)* e *M(i,0)*. O algoritmo para encontrar o alinhamento é exatamente igual ao Algoritmo de Needleman-Wunsch [26].

$$M(0,0) = 0 \tag{3.4}$$

$$M(i,0) = max\begin{cases} M(i-1,0) \\ M(i-1,j) - T \qquad j = 1 \ldots, m \end{cases} \tag{3.5}$$

$$M(i,j) = max\begin{cases} M(i,0) \\ M(i-1,j-1) + s(x_i, y_j) \\ M(i-1,j) - d \\ M(i,j-1) - d \end{cases} \tag{3.6}$$



(a) (b)

Figura 3.7: (a) Representação gráfica dos possíveis valores para M(i,j). (b) Representação gráfica para os possíveis valores de M(i,0).

3.2.1.3 - Autômato de estado finito (FSA – *Finite State Automaton*)

Uma das principais deficiências nos algoritmos anteriormente apresentados está na estratégia de penalização quando existem alinhamentos com agrupamentos de *gaps*. Nestes casos, a penalização dada ao alinhamento é um múltiplo do comprimento do agrupamento, estratégia que não leva em conta que em alinhamentos de sequências biológicas

usualmente são achadas corridas de *gaps* de comprimento maior do que um. Uma melhoria no algoritmo é a incorporação de estados, mediante os quais o algoritmo está ciente de quando ocorrem corridas de *gaps* com comprimento maior que um. O alinhamento é descrito neste tipo de algoritmos como o caminho seguido através dos estados definidos [5].

3.2.1.4 - Heurísticas

Como os algoritmos de programação dinâmica geralmente têm uma complexidade de ordem *O(nm)*, e o tamanho das bases de dados é grande, a geração dos possíveis alinhamentos de uma sequência *query* contra uma base de dados pode levar um tempo considerável. Algoritmos heurísticos foram desenvolvidos para diminuir o tempo de busca, sacrificando sensibilidade na busca de possíveis alinhamentos, mas apresentando tempos de execução reduzidos. Esses algoritmos heurísticos utilizam sementes iniciais para iniciar a busca de alinhamentos locais com pontuações altas e depois realizam a geração do alinhamento para a sequência. Entre os algoritmos heurísticos mais conhecidos estão BLAST [20] e FASTA [21].

3.2.1.5 – Algoritmo das Divergências para alinhamento de pares (*pairwise alignment*)

Outra das técnicas desenvolvidas para reduzir o tempo de processamento dos algoritmos de programação dinâmica consiste em limitar as regiões das matrizes que são calculadas por esses algoritmos. O primeiro a propor um algoritmo que conseguia reduzir o número de cálculos necessários foi Fickett [18], e o fez para otimizar o cálculo do algoritmo original de Smith-Waterman, reduzindo o número de células das matrizes de programação dinâmica que deviam ser calculadas. No algoritmo de Fickett é definida, a priori, uma faixa das matrizes de programação dinâmica que será calculada. Se o cálculo das células das matrizes dentro da faixa produz um alinhamento, então o processamento é interrompido e o alinhamento é gerado, caso contrário, a faixa é aumentada e o calculo é repetido para as células dentro da faixa. Os principais problemas do algoritmo de Fickett são o desconhecimento do tamanho da faixa e o fato de que o cálculo tem que começar desde o canto superior esquerdo das matrizes, o que limita sua aplicação para sequências que se alinham muito bem [19]. A figura 3.8 ilustra o conceito básico por trás do algoritmo. A

figura da esquerda paresenta uma faixa predefinida que não contem o alinhamento e a figura da direta apresenta a modificação feita nessa faixa para incluir o alinhamento.

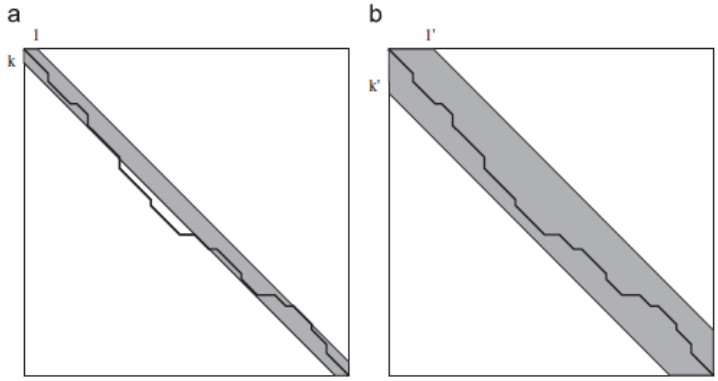


Figura 3.8: Algoritmo de Fickett [18].

Posteriormente foi introduzido o conceito de divergência [19] que consiste na obtenção das coordenadas para as diagonais propostas por Fickett [18] ao calcular as matrizes de programação dinâmica. O conceito foi introduzido principalmente para restringir o espaço de memória necessário para realizar o cálculo das matrizes e funciona em duas etapas. Na primeira, o cálculo das matrizes de programação dinâmica é feito em paralelo sem armazenar as matrizes, mas somente produzindo a pontuação para alinhamentos locais e as coordenadas para as diagonais que "envolvem" esses alinhamentos. A segunda etapa consiste no cálculo das células das matrizes que estão dentro das diagonais, e na geração desses alinhamentos locais para posterior análise. O conceito proposto por [19] é aplicado somente para operações de comparação e alinhamento de pares em sequências de DNA e é muito adequado para ser implementado tanto em software paralelo (vários processadores) quanto em hardware. Ele foi modificado por Moreano [37][38] para sua aplicação em operações de comparação de proteínas, e será analisado na seção 3.3.

3.2.2 - Alinhamentos Múltiplos, Famílias de Proteínas e *profileHMMs*

Um alinhamento múltiplo é o alinhamento de um número arbitrário de sequências, as quais são alinhadas para procurar características comuns entre elas. Em princípio, um alinhamento múltiplo funciona igual a um alinhamento de pares, mas é computacionalmente mais intensivo. Em um alinhamento múltiplo, os elementos homólogos das sequências são alinhados, procurando características estruturais e

evolutivas comuns. Idealmente, uma coluna de resíduos alinhados compartilha a mesma estrutura tri-dimensional, e é o resultado da evolução de um ancestral comum [24]. Uma correspondência na estrutura de duas ou mais proteínas, geralmente significa que as proteínas têm a mesma função, seja dentro do mesmo organismo ou em diferentes organismos. Ao construir um alinhamento múltiplo, é obtida uma subsequência que contém os elementos representativos da família, chamada de sequência de consenso, e que é utilizada para identificar possíveis membros da família. A Figura 3.9 mostra um exemplo de alinhamento múltiplo e a respectiva sequência de consenso [5]. A sequência de consenso possui o aminoácido no caso que as 10 sequências tenham o mesmo resíduo na mesma coluna ou um + quando os resíduos da coluna são altamente conservativos (são elementos predominantes entre as sequências que pertencem a família).

```
structure:  ddd.....eeeeee.......fffffffff.......ggggggggggg.
1tlk        IDYDEEGNCSLTISEVCGDDDAKYTCKAVNSL-----GEATCTAELLVET
AXO1_RAT    SQTT----GNLYIARTNASDLGNYSCLATSHMDFSTKSVFSKFAQLNLAA
AXO1_RAT    VLA-----GDLRFSKLSLEDSGMYQCVAENKH-----GTIYASAELAVQA
AXO1_RAT    VTSD----GTLIIRNISRSDEGKYTCFAENFM-----GKANSTGILSVRD
AXO1_RAT    AKETI---GDLTILNAHVRHGGKYTCMAQTVV-----DGTSKEATVLVRG
NCA2_HUMAN  VLSN----NYLQIRGIKKTDEGTYRCEGRILARG---EINFKDIQVIVNV
NCA2_HUMAN  VVWNDDSSSTLTIYNANIDDAGIYKCVVTGEDG----SESEATVNVKIFQ
NCA2_HUMAN  FSDDSS---QLTIKKVDKNDEAEYICIAENKA-----GEQDATIHLKVFA
NRG_DROME   QGHYG---KSLVIRQTNFDDAGTYTCDVSNGVG----NAQSFSIILNVNS
NRG_DROME   KTND----NSLTIAKTMELDSGEYTCVARTRL-----DEATARANLIVQD
consensus:  ..........L.+..+...+.+.Y.C.................+.+.+..
```

Figura 3.9: Alinhamento múltiplo e sequência de consenso para 10 elementos da família I-set imunoglobulina.

Proteínas que compartilham funcionalidade ou estrutura (ou as duas) geralmente são agrupadas em famílias de proteínas, e essas famílias são utilizadas para identificar ainda mais membros da família. Antigamente, a identificação de um novo membro de uma família de proteínas era feita através de um alinhamento de pares entre uma sequência *query* e um ou todos os elementos conhecidos da família. O principal problema do alinhamento de pares para identificar membros da família é que ele não consegue identificar as características estatísticas da mesma, e falha na identificação de membros distantes dela.

Já que os alinhamentos múltiplos fornecem informação de como as sequências dentro de uma família se relacionam entre elas, sendo utilizados para construir modelos probabilísticos das famílias de proteínas. Estes modelos utilizam Modelos Ocultos de Markov para determinar a estrutura estatística da família de proteínas, já que este tipo de modelo é adequado para extrair a estrutura interna de uma família de proteínas descrita por

um alinhamento múltiplo. O modelo extraído do alinhamento múltiplo é suficientemente geral para detectar os *motifs* e suas variações em sequências da família. Os *profileHMMs* são Modelos Ocultos de Markov para uma família de proteínas, extraídos a partir de um alinhamento múltiplo correto para ela. A construção de alinhamentos múltiplos [5] está fora do foco do nosso trabalho, e já que esses alinhamentos podem ser obtidos das bases de dados públicas, não é de nosso interesse discutir os algoritmos utilizados para construí-los.

Um *profileHMM* é a representação da estrutura estatística interna de uma família de proteínas. Matematicamente, um *profileHMM* é representado como um diagrama de estados [10]. Na sua versão mais básica, um *profileHMM* possui três tipos de estados, suficientes para realizar um alinhamento global de uma sequência *query* com o modelo gerado a partir do alinhamento múltiplo da família de sequências. O modelo básico possui três tipos de estados, característica que permite produzir alinhamentos com *gaps*. Os estados de *match* (M) correspondem a elementos da sequência que pertencem à sequência de consenso da família. Os estados de *insert* (I) correspondem a partes da sequência que não estão dentro do HMM, e que são emitidas (inseridas) dentro do alinhamento. Um estado de inserção pode emitir um ou mais elementos. Isso é modelado através da seta de transição que sai dele e chega nele. Os estados de *delete* (D) modelam *gaps* dentro da sequência, e são partes do modelo que não têm correspondência na sequência (inserem um *gap* "-" dentro do alinhamento final). Na literatura que aborda os *profileHMMs*, o grupo conformado pelos três estados M, I e D é denominado um nodo. A pontuação e o alinhamento da sequência *query* são produzidos ao inserir todos os elementos da sequência através do estado inicial (*Begin*) fazendo com que eles passem pelos nodos do modelo até o estado final (*end*), aplicando o Algoritmo de Viterbi [17]. A Figura 3.10 apresenta esta versão simplificada de um HMM [5]. As probabilidades de transição são representadas pelas setas da figura. As probabilidades de emissão não são apresentadas na figura.

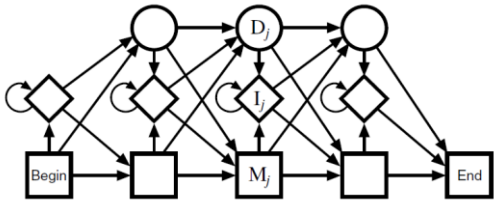


Figura 3.10: Versão simplificada de um profileHMM incluindo somente os estados de Match, Insert e Delete.

Como o foco do trabalho é a implementação de um acelerador para o software HMMER [1][2], devemos analisar a arquitetura do HMM que o software utiliza para realizar o alinhamento da sequência com o *profileHMM*. A arquitetura que o HMMER utiliza é denominada plan7 [1][2][17], e é uma modificação do HMM básico que inclui alinhamentos globais, locais e locais múltiplos (vários alinhamentos locais dentro de uma mesma sequência). A Figura 3.11 apresenta o *profileHMM* utilizado dentro do software HMMER.

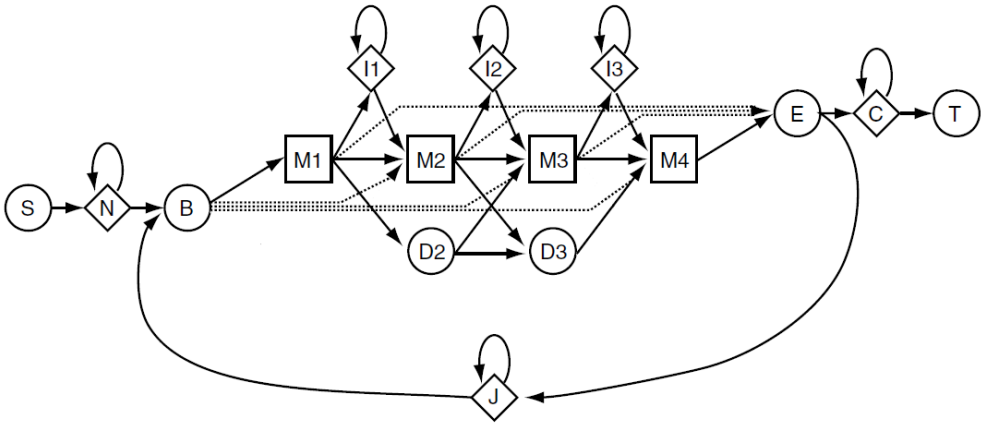


Figura 3.11: Arquitetura plan7 do Viterbi.

Na Figura 3.10, observa-se que o modelo possui, além dos estados tradicionais M, I, D, B (*begin*) e E (*end*), estados adicionais (S, N, C, T, J) que controlam as características algorítmicas do modelo, tais como o tipo de alinhamento a produzir (global ou local) e a capacidade do modelo para produzir alinhamentos locais *multi-hit*. Todos os alinhamentos produzidos pelo plan7 são alinhamentos globais, mas somente no sentido da sequência *query* ser inserida completamente dentro do *profileHMM* desde o estado inicial do algoritmo (S) até o estado terminal do algoritmo (T). Os alinhamentos locais e locais múltiplos são feitos através dos estados que controlam o algoritmo N, C, J, já que num alinhamento local, partes da sequência são "absorvidas" por estes estados. Alinhamentos locais da sequência com o modelo são controlados pelos estados N e C. Se a probabilidade de transição N->N é configurada como 0, os alinhamentos estão restringidos a começar no primeiro nodo do *profileHMM*. Se a probabilidade de transição C->C é configurada como 0, então os alinhamentos estão restringidos a terminar no último nodo do modelo. Alinhamentos locais do modelo com a sequência são controlados pelas transições B->M e

M->E. Alinhamentos multi-hit são controlados pela transição E->J e pelo estado J, que permitem que várias partes da sequência estejam alinhadas com o modelo [17]. Os estados M, I, N, C e J emitem um elemento da sequência quando se apresenta uma transição para eles, e por isso são chamados de estados emissores. Os estados D, S, B e E não emitem nenhum elemento, e são estados silenciosos que servem para modelar situações de *delete* na sequência ou condições algorítmicas específicas.

No software HMMER, é utilizado o Algoritmo de Viterbi [17] para calcular as matrizes de programação dinâmica para o *profileHMM*. O algoritmo calcula a pontuação ótima para a sequência *query* quando é inserida no HMM. Além do cálculo da melhor pontuação, também é possível gerar o alinhamento ótimo para a sequência, fazendo o *traceback* nas matrizes de programação dinâmica geradas pelo algoritmo. O Algoritmo 3.2 apresenta as equações para a arquitetura *plan7* implementada no HMMER. As transições estão representadas por *tr(estado1,estado2)* e as emissões por *em(estado,aminoácido).*

for j←0 to k do //Inicialização das matrizes de PD
  $M_{0,j} \leftarrow -\infty$
  $I_{0,j} \leftarrow -\infty$
  $D_{0,j} \leftarrow -\infty$
end for
for i←0 to n do
  $M_{i,0} \leftarrow -\infty$
  $I_{i,0} \leftarrow -\infty$
  $D_{i,0} \leftarrow -\infty$
  $E_{i,(0)} \leftarrow -\infty$
end for
$N_0 \leftarrow 0$
$B_0 \leftarrow tr(N, B)$
$C_0 \leftarrow -\infty$
$J_0 \leftarrow 0$

$$
\begin{aligned}
&for\ i \leftarrow 1\ to\ n\ //\text{para cada aminoácido da sequência}\\
&\quad for\ j\ \leftarrow 1\ to\ k\ //\text{Para cada nodo do profileHMM}\\
&\qquad M(i,j) = em_{(M_j,S_i)}\ + max\begin{cases} M_{i-1,j-1} + tr_{M_{j-1},M_j} \\ I_{i-1,j-1} + tr_{I_{j-1},M_j} \\ D_{i-1,j-1} + tr_{D_{j-1},M_j}, \\ B_{i-1} + tr_{B,M_j} \end{cases}\\
&\qquad I(i,j) = em_{(I_j,S_i)} + max\begin{cases} M_{i-1,j} + tr_{M_j,I_j} \\ I_{i-1,j} + tr_{I_j,I_j} \end{cases}\\
&\qquad D(i,j) = max\begin{cases} M_{i,j-1} + tr_{M_{j-1},D_j} \\ D_{i,j-1} + tr_{D_{j-1},D_j} \end{cases}\\
&\quad end\ for\\
&\quad N_i = N_{i-1} + tr_{N,N}\\
&\quad B_i = max\begin{cases} N_i + tr_{N,B} \\ J_i + tr_{J,B} \end{cases}\\
&\quad E_i = max_{\,1\le j\le k}(M_{i,j} + tr_{M_j,E})\\
&\quad J_i = max\begin{cases} J_{i-1} + tr_{J,J} \\ E_i + tr_{E,J} \end{cases}\\
&\quad C_i = max\begin{cases} C_{i-1} + tr_{C,C} \\ E_i + tr_{E,C} \end{cases}\\
&end\ for\\
&similaridade = C_n + tr_{C,T}
\end{aligned}
$$

Algoritmo 3.2: Equações do Algoritmo de Viterbi para um profileHMM de comprimento k e uma sequência *query* de comprimento n.

A Figura 3.12 apresenta o alinhamento emitido pelo software HMMER quando é comparada a sequência **sp|P18560|1101L_ASFB7** com o HMM para a família de sequências PF00560 da base de dados PFAM. O HMM foi obtido configurando o software HMMER para eliminar o estado J (alinhamentos locais *multi-hit*), já que o acelerador proposto no nosso trabalho implementa uma simplificação do Plan7 utilizado no HMMER (vide Seção 4.2). A figura apresenta os estados seguidos pela sequência e o alinhamento emitido pelo algoritmo para a sequência *query*.

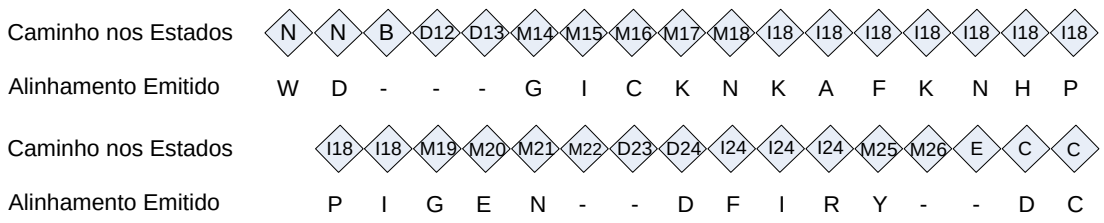


Figura 3.12: Alinhamento obtido através do HMMER para uma sequência aleatória com o HMM da família PF00560 da base de dados PFAM.

3.3 - ALGORITMO DAS DIVERGÊNCIAS APLICADO A PROFILEHMMS

O algoritmo é proposto por [37][38], e está baseado no princípio de que, em um alinhamento local de uma sequência com um profileHMM, somente uma porção da sequência está alinhada com ele, fato que permite que a porção da matriz de programação dinâmica que é necessário calcular seja reduzida. Esta porção se vê limitada pelo índice do elemento da sequência onde começa o alinhamento, pelo índice do elemento da sequência onde termina o alinhamento e por duas diagonais que limitam as células que devem ser calculadas dentro da matriz. A proposta do algoritmo supõe que as matrizes de programação dinâmica e os elementos limitadores foram já calculados num acelerador externo (geralmente mais rápido do que o software), o que permite descartar as sequências com pontuações muito baixas, mas esse acelerador não fornece as informações para gerar o alinhamento. A Figura 3.13 apresenta os elementos que limitam a região de interesse das matrizes de programação dinâmica.

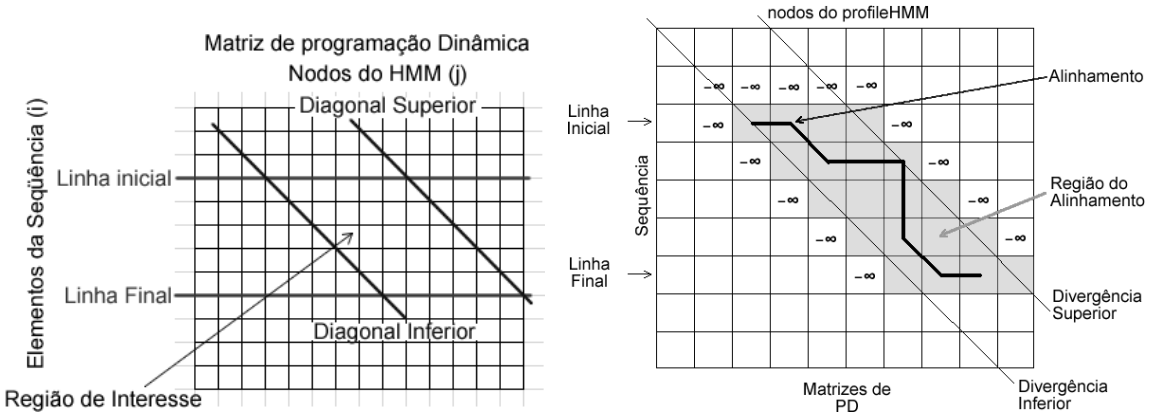


Figura 3.13: Principais Elementos do Algoritmo das divergências.

As diagonais superior e inferior são determinadas por dois parâmetros denominados de divergências, as quais são subtraídas das linhas inicial e final para determinar qual é o ponto do início e do final da região de interesse, além de indicar a trajetória das diagonais na matriz. Como foi mencionado anteriormente, as linhas inicial e final do algoritmo são os índices dos elementos da sequência *query* onde começa e termina o melhor alinhamento local possível com o profileHMM. A divergência inferior possui duas funções: indica o lugar do HMM onde o primeiro elemento do alinhamento começa (que corresponde à borda superior esquerda da região de interesse) e a trajetória da diagonal inferior da região de interesse da matriz de programação dinâmica. A divergência superior indica o lugar da

borda superior direita da região de interesse e a trajetória da diagonal superior dela. O algoritmo proposto [37][38] possui duas etapas. A primeira parte consiste no cálculo da pontuação para o melhor alinhamento local possível da linha inicial, da linha final, da divergência inferior e da divergência superior. Esta primeira parte será implementada no hardware proposto e é explicada nos capítulos 4 e 5. Ela procura, além de atuar como uma primeira etapa de filtragem, fornecer os dados necessários para a segunda parte de o algoritmo economizar cálculos na produção do alinhamento, ao processar somente a região de interesse da matriz de programação dinâmica. A segunda etapa do algoritmo tem como entrada os dados fornecidos para a primeira parte e realiza o cálculo do Algoritmo de Viterbi para a região de interesse delimitada por eles. Depois de fazer o cálculo do Algoritmo de Viterbi, o alinhamento pode ser obtido.

3.3.1 - Primeira Etapa do Algoritmo

Para a obtenção das linhas inicial e final, e das divergências superior e inferior, o Algoritmo de Viterbi tem que ser modificado, e, além das pontuações para os estados M, I, D, E, C, também aparecem os dados para a divergência superior, a divergência inferior e a linha inicial associadas com esses estados. O dado para a linha final do alinhamento só é calculado na etapa final do algoritmo (no estado C). A primeira parte do Algoritmo de Viterbi não é modificada, e consiste na inicialização das pontuações das matrizes e vetores de programação dinâmica para os estados M, I, D, E, B e C. O pseudocódigo desta parte é apresentado no Algoritmo 3.3. Ele é apresentado como foi implementado no hardware (Capítulos 4 e 5), sem o estado J, com um número arbitrário (k) de nodos no profileHMM e um número arbitrário de elementos (n) na sequência *query*.

for j←0 to k do //Inicialização das matrizes
  $M_{0,j} \leftarrow -\infty$
  $I_{0,j} \leftarrow -\infty$
  $D_{0,j} \leftarrow -\infty$
end for
for i←0 to n do
  $M_{i,0} \leftarrow -\infty$
  $I_{i,0} \leftarrow -\infty$
  $D_{i,0} \leftarrow -\infty$
  $E_{i,(0)} \leftarrow -\infty$
end for
$B_0 \leftarrow tr_{N,B}$
$C_0 \leftarrow -\infty$

Algoritmo 3.3: Inicialização das matrizes de programação dinâmica do Algoritmo de Viterbi.

A segunda parte da primeira etapa está encarregada da inicialização das matrizes de programação dinâmica para as divergências, de maneira similar à das matrizes para os estados tradicionais, mas com valores inicializados para as linhas e colunas iniciais das matrizes. Cada estado tem três matrizes, uma para a linha inicial, uma para a divergência inferior e outra para a divergência superior. No Algoritmo 3.4 são apresentadas as inicializações para todas as matrizes e na Figura 3.14 são representadas graficamente as inicializações feitas ele.

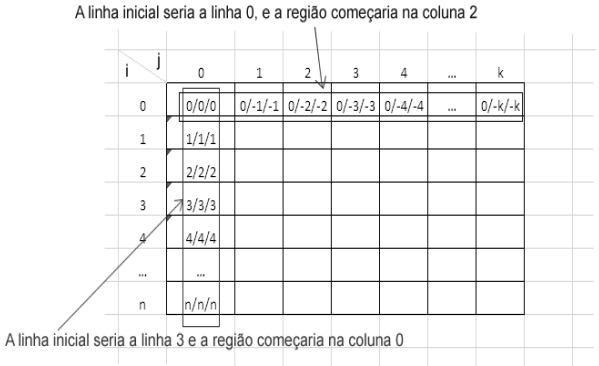


Figura 3.14: inicialização das matrizes de programação dinâmica para as divergências.

```
for i←0 to n do
    Mdiv_inf_{i,0} ← i
    Mdiv_sup_{i,0} ← i
    Mlin_ini_{i,0} ← i
    Idiv_inf_{i,0} ← i
    Idiv_sup_{i,0} ← i
    Ilin_ini_{i,0} ← i
    Ddiv_inf_{i,0} ← i
    Ddiv_sup_{i,0} ← i
    Dlin_ini_{i,0} ← i
    Ediv_inf_{i,0} ← i
    Ediv_sup_{i,0} ← i
    Ediv_ini_{i,0} ← i
end for

for j ← 0 to k do
    Mdiv_inf_{0,j} ← −j
    Mdiv_sup_{0,j} ← −j
    Mlin_ini_{0,j} ← 0
    Idiv_inf_{0,j} ← −j
    Idiv_sup_{0,j} ← −j
    Ilin_ini_{0,j} ← 0
    Ddiv_inf_{0,j} ← −j
    Ddiv_sup_{0,j} ← −j
    Dlin_ini_{0,j} ← 0
end for
Cdiv_inf_0 ← 0
Cdiv_sup_0 ← 0
Clin_ini_0 ← 0
Clin_fin_0 ← 0
```

Algoritmo 3.4: inicialização das matrizes de programação dinâmica para o Algoritmo das Divergências para uma sequência com *n* aminoácidos e um *profileHMM* com *k* nodos.

A terceira parte consiste no cálculo das pontuações para os estados tradicionais do Algoritmo de Viterbi (M, I, D, E), mas uma modificação é feita na recursividade para gerar os sinais de controle para o Algoritmo das Divergências. A modificação consiste em levar um registro de quais foram os termos escolhidos dentro dos máximos das equações para os estados, como é mostrado no Algoritmo 3.5.

$for\ i \leftarrow 1\ to\ n\ do$ //Para cada elemento da sequência
$B_i \leftarrow B_{i-1} + tr_{N,N}$
$for\ j \leftarrow 1\ todo\ k$ //Para cada nodo do profileHMM
$M_{i,j} \leftarrow em_{(M_j,S_i)} + \max_{4i}(M_{i-1,j-1} + tr_{M_{j-1},M_j}, I_{i-1,j-1} + tr_{I_{j-1},M_j}, D_{i-1,j-1} + tr_{D_{j-1},M_j}, B_{i-1} + tr_{B,M_j}, \& indice_M)$
$I_{i,j} \leftarrow em_{(I_j,S_i)} + \max_{2i}\left(M_{i-1,j} + tr_{M_j,I_j}, I_{i-1,j} + tr_{I_j,I_j}, \& indice_I\right)$
$D_{i,j} \leftarrow \max_{2i}\left(M_{i,j-1} + tr_{M_{j-1},D_j}, D_{i,j-1} + tr_{D_{j-1},D_j}, \& indice_D\right)$
$E_i \leftarrow \max_{2i}(E_i, M_{i,j} + tr_{M_j,E}, \& indice_E)$
$/*\ Calculo\ das\ divergências\ */$
$end\ for$
$C_i \leftarrow \max_{2i}(C_{i-1} + tr_{C,C}, E[i] + tr_{E,C}, \& indice\ C)$
$/*\ Calculo\ das\ divergências\ */$
$end\ for$
$pontuação \leftarrow C_{seq_{len}} + tr_{C,T}$
$/*\ Calculo\ das\ divergências\ */$
$return\ pontuação$

Algoritmo 3.5: Modificação do Algoritmo de Viterbi feita para conservar os índices dos máximos escolhidos.

Os índices conservados servem para observar qual dos estados foi tomado e, portanto, a procedência das pontuações obtidas, fato que permite aumentar ou diminuir a região de interesse na hora de reprocessar uma sequência com uma pontuação alta. A parte final do algoritmo consiste no cálculo das divergências e das linhas inicial e final da região de interesse. O Algoritmo 3.6 apresenta o cálculo das divergências.

O algoritmo é implementado em hardware por duas razões: para que o hardware seja uma primeira etapa de filtragem e elimine as sequências com uma pontuação inferior a um limiar definido pelo software, e para que, ao fazer o cálculo das divergências, o software economize tempo de processamento ao produzir o alinhamento. Para a produção do alinhamento e cálculo da pontuação por parte do software, o Algoritmo de Viterbi que rodará no nosso processador de propósito geral também teve que ser modificado.

$for\ i \leftarrow 1\ to\ n\ do$ //Para todos os elementos da sequência
$B_i \leftarrow B_{i-1} + tr_{(N,N)}$
$for\ j \leftarrow 1\ to\ k\ do$ //Para todos os nodos do profileHMM
$/*\ Calculo\ das\ matrizes\ de\ pontuação\ */$
$switch(indice_M)$ // Divergências para o estado M
case 0:
$Mdiv_{inf_{i,j}} \leftarrow Mdiv_{inf_{i-1,j-1}}$
$Mdiv_{sup_{i,j}} \leftarrow Mdiv_{sup_{i-1,j-1}}$
$Mlin_{ini_{i,j}} \leftarrow Mlin_{ini_{i-1,j-1}}$
break
case 1:
$Mdiv_{inf_{i,j}} \leftarrow max_2(i-j, Idiv_{inf_{i-1,j-1}})$
$Mdiv_{sup_{i,j}} \leftarrow min_2(i-j, Idiv_{sup_{i-1,j-1}})$
$Mlin_{ini_{i,j}} \leftarrow Ilin_{ini_{i-1,j-1}}$
break
case 2:
$Mdiv_{inf_{i,j}} \leftarrow max_2(i-j, Ddiv_{inf_{i-1,j-1}})$
$Mdiv_{sup_{i,j}} \leftarrow min_2(i-j, Ddiv_{sup_{i-1,j-1}})$
$Mlin_{ini_{i,j}} \leftarrow Dlin_{ini_{i-1,j-1}}$
break
case 3:
$Mdiv_{inf_{i,j}} \leftarrow i-j$
$Mdiv_{sup_{i,j}} \leftarrow i-j$
$Mlin_{ini_{i,j}} \leftarrow i$
break
$if(indice_I = 0)$ // Divergências para o estado I
$Idiv_{inf_{i,j}} \leftarrow max_2(i-j, Mdiv_{inf_{i-1,j}})$
$Idiv_{sup_{i,j}} \leftarrow min_2(i-j, Mdiv_{sup_{i-1,j}})$
$Ilin_{ini_{i,j}} \leftarrow Mlin_{ini_{i-1,j}}$
else
$Idiv_{inf_{i,j}} \leftarrow max_2(i-j, Idiv_{inf_{i-1,j}})$
$Idiv_{sup_{i,j}} \leftarrow min_2(i-j, Idiv_{sup_{i-1,j}})$
$Ilin_{ini_{i,j}} \leftarrow Ilin_{ini_{i-1,j}}$
end if
$if(indice_D = 0)$ // Divergências para o estado D
$Ddiv_{inf_{i,j}} \leftarrow max_2(i-j, Mdiv_{inf_{i,j-1}})$
$Ddiv_{sup_{i,j}} \leftarrow min_2(i-j, Mdiv_{sup_{i,j-1}})$
$Dlin_{ini_{i,j}} \leftarrow Mlin_{ini_{i,j-1}}$

$$
\begin{aligned}
&else\\
&\quad Ddiv_{inf_{i,j}} \leftarrow max_2\left(i-j, Ddiv_{inf_{i,j-1}}\right)\\
&\quad Ddiv_{sup_{i,j}} \leftarrow min_2\left(i-j, Ddiv_{sup_{i,j-1}}\right)\\
&\quad Dlin_{ini_{i,j}} \leftarrow Dlin_{ini_{i,j-1}}\\
&end\ if\\
&if(indice_E = 1)// \ Divergências\ para\ o\ estado\ E\\
&\quad Ediv_{inf_i} \leftarrow Mdiv_{inf_{i,j}}\\
&\quad Ediv_{sup_i} \leftarrow Mdiv_{sup_{i,j}}\\
&\quad Elin_{ini_i} \leftarrow Mlin_{ini_{i,j}}\\
&\quad col_{fin} \leftarrow j\\
&end\ if\\
&end\ for\\
&/*\ Calculo\ do\ vetor\ C\ */\\
&if(indice_C = 0)\\
&\quad Cdiv_{inf_i} \leftarrow Cdiv_{inf_{i-1}}\\
&\quad Cdiv_{sup_i} \leftarrow Cdiv_{sup_{i-1}}\\
&\quad Clin_{ini_i} \leftarrow Clin_{ini_{i-1}}\\
&\quad Clin_{fin_i} \leftarrow Clin_{fin_{i-1}}\\
&else\\
&\quad Cdiv_{inf_i} \leftarrow Ediv_{inf_i}\\
&\quad Cdiv_{sup_i} \leftarrow Ediv_{sup_i}\\
&\quad Clin_{ini_i} \leftarrow Elin_{ini_i}\\
&\quad Clin_{fin_i} \leftarrow i\\
&end\ if\\
&end\ for\\
&/*\ Calculo\ da\ pontuação\ */\\
&div_{inf} \leftarrow Cdiv_{inf_{seq_{len}}}\\
&div_{sup} \leftarrow Cdiv_{sup_{seq_{len}}}\\
&lin_{ini} \leftarrow Clin_{ini_{seq_{len}}}\\
&lin_{fin} \leftarrow Clin_{fin_{seq_{len}}}\\
&return\ pontuação
\end{aligned}
$$

Algoritmo 3.6: Equações para o cálculo das matrizes de programação dinâmica para as divergências.

3.3.2 - Segunda Etapa do Algoritmo

Depois do processamento da sequência feito pela primeira etapa, a segunda etapa faz a leitura da pontuação da sequência. Se a pontuação estiver acima do limiar definido, então lê os parâmetros para a linha inicial, a linha final, a divergência superior e a divergência inferior. Esses parâmetros calculados pela primeira etapa são fornecidos à segunda etapa para ela produzir o alinhamento ao processar somente a região de interesse. Primeiramente inicializa as células necessárias para realizar o processamento das matrizes de programação

dinâmica na região de interesse (a linha anterior à linha inicial e as anti-diagonais), para depois fazer a execução do Algoritmo de Viterbi. O processo de inicialização das matrizes é apresentado na Figura 3.15 e no Algoritmo 3.7.

Figura 3.15: Inicialização feita pelo software nas matrizes M, I, D para o cálculo da pontuação e a geração do alinhamento na região de interesse.

$$
\begin{aligned}
&/* \textit{Inicialização da linha anterior à linha inicial} */ \\
&j_{ini} \leftarrow max_2(0, lin_{ini} - 1 - div_{inf}) \\
&j_{fin} \leftarrow min_2\left(HMMnodes, lin_{ini} - div_{sup}\right) \\
&\textit{for } j \leftarrow j_{ini} \textit{ to } j_{fin} \textit{ do} \\
&\quad M_{lin_{ini}-1,j} \leftarrow -\infty \\
&\quad I_{lin_{ini}-1,j} \leftarrow -\infty \\
&\quad D_{lin_{ini}-1,j} \leftarrow -\infty \\
&\textit{end for} \\
&B_{lin_{ini}-1} \leftarrow tr_{N,B} + (lin_{ini} - 1) * tr_{N,N} \\
&C_{lin_{ini}-1} \leftarrow -\infty \\
&\\
&/* \textit{Inicialização das anti} - \textit{diagonais} */ \\
&\textit{for } i \leftarrow lin_{ini} \textit{ to } lin_{fin} \textit{do} \\
&\quad j_{ini} \leftarrow max_2\left(0, i - 1 - div_{inf}\right) \\
&\quad j_{fin} \leftarrow min_2(HMMnodes, i - div_{sup} + 1) \\
&\quad M_{i,j_{ini}} \leftarrow -\infty \\
&\quad I_{i,j_{ini}} \leftarrow -\infty \\
&\quad D_{i,j_{ini}} \leftarrow -\infty \\
&\quad M_{i,j_{fin}} \leftarrow -\infty \\
&\quad I_{i,j_{fin}} \leftarrow -\infty \\
&\quad D_{i,j_{fin}} \leftarrow -\infty \\
&\quad E_i \leftarrow -\infty \\
&\textit{end for}
\end{aligned}
$$

Algoritmo 3.7: Inicialização das matrizes e vetores de programação dinâmica feita pelo software no processamento da região de interesse.

Logo depois da inicialização, o algoritmo começa o processamento da região de interesse. Como as fronteiras direita e esquerda da região de interesse são diagonais, os índices j inicial e final devem ser recalculados para cada elemento i da sequência. O índice do valor máximo para a matriz M é mantido para gerar o alinhamento e é guardado numa matriz de índices que é utilizada no Algoritmo de *traceback* apresentado no apêndice A junto com o código em C feito para comprovar o funcionamento do algoritmo e obter estatísticas de desempenho. O Algoritmo 3.8 apresenta o processo de processamento da região de interesse, e é basicamente o Algoritmo de Viterbi, mas somente processa a sequência desde a linha inicial até a linha final, e apenas nos nodos do HMM delimitados pelas divergências.

$/* Para\ os\ simbolos\ delimitados\ pelas\ linhas\ inicial\ e\ final\ */$
$for\ i \leftarrow lin_{ini} to\ lin_{fin} do$
$B_i \leftarrow B_{i-1} + tr_{N,N}$
$j_{ini} \leftarrow max_2(1, i - div_{inf})\ //Deslocamento\ da\ fronteira$
$j_{fin} \leftarrow min_2(HMMnodes, i - div_{sup})$
$/* Para\ os\ nodos\ delimitados\ pelas\ fronteiras\ */$
$for\ j \leftarrow j_{ini} to\ j_{fin} do$
$/* Calculo\ das\ pontuações\ para\ M, I, D\ e\ E\ */$
$Mtrace_{i,j} \leftarrow indice_M$
$if\ (indice_M = 1)$
$col_{fin} \leftarrow j$
$end\ if$
$end\ for$
$/* Calculo\ do\ vetor\ C\ */$
$end\ for$
$/* Calculo\ da\ pontuação\ */$
$return\ pontuação$

Algoritmo 3.8: Processamento da região de interesse em software.

3.4 - TRABALHOS RELACIONADOS

Devido ao constante crescimento do número de sequências armazenadas dentro das bases de dados de proteínas [7-16], um grande número de pesquisadores está tentando acelerar os algoritmos de comparação e busca como o Smith-Waterman [28] e o Plan7 [1][2][17]. Já que o Plan7 e o software HMMER estão entre as ferramentas mais utilizadas para realizar comparação entre famílias de proteínas, um grande número de estratégias de otimização para acelerar a execução do software tem sido proposto. MPI-HMMER [11][14] explora a

execução do software sobre um cluster de computadores universitário, alem da otimização do software que se executa em cada nodo, através do conjunto de instruções SS2 da Intel [39]. Aplicações como SledgeHMMER [29] e "HMMER on the Sun Grid" [22] são serviços de comparação on-line. SledgeHMMER é uma versão otimizada do HMMER que executa as buscas sobre um servidor web de alta capacidade, com o fim de fornecer aos pesquisadores uma plataforma rápida para realizar comparações. "HMMER on the Sun Grid" é a implementação de HMMER sobre o cluster de servidores implementado pela Sun Microsystems. Outras aplicações como ClawHMMER [13] e GPU-HMMER [23] procuram paralelizar o Plan7 sobre unidades de processamento gráfico, já que estas consistem em múltiplas unidades funcionais que podem atingir altas taxas de processamento. A maioria dos aceleradores estudados utiliza o fato de o HMMER gastar a maior parte do tempo de execução processando sequências não significativas (sem relação com a família de proteínas de interesse), e dividem o fluxo de execução da aplicação em dois estágios. No primeiro estágio, um filtro para determinar se a seqüencia é significativa é implementado. Caso seja significativa (tenha uma pontuação alta), o segundo estágio re-processa a sequência e produz o alinhamento. A tendência de utilizar aceleradores baseados em FPGAs é observada nos trabalhos estudados [7-16], devido à sua velocidade relativamente grande e ao seu baixo custo de implementação.

Os aceleradores baseados em FPGAs geralmente consistem em um arranjo de Elementos de Processamento (EP) ligados em forma de Arranjo Sistólico (AS). Um AS deste tipo pode explorar o paralelismo dentro do Algoritmo de Viterbi ao eliminar o estado de realimentação J (vide capítulo quatro), ou pode explorar a independência no cálculo da pontuação entre sequências diferentes quando são comparadas com o mesmo profileHMM. Outra característica importante dos aceleradores baseados em FPGAs é que, como o hardware dentro de um FPGA é limitado, o cálculo da pontuação das sequências para profileHMMs muito compridos é dividido em um conjunto de varias iterações [7][8][10][16] e os resultados intermediários são armazenados dentro de memórias intermediárias (FIFO). As probabilidades de emissão e transição para os estados do profileHMM com o qual estão sendo comparadas as sequências *query* são armazenadas em memórias RAM implementadas no sistema e carregadas para cada iteração, com o fim de ocultar as latências causadas ao mudar entre uma iteração e outra. É importante levar em conta que a inserção de elementos de memória impõe limitações ao número máximo de EPs que podem ser inseridos dentro das FPGAs, devido ao fato delas somente possuírem

um número limitado de blocos de memória. Na Figura 3.16 é apresentado o diagrama de blocos de um acelerador baseado em FPGA.

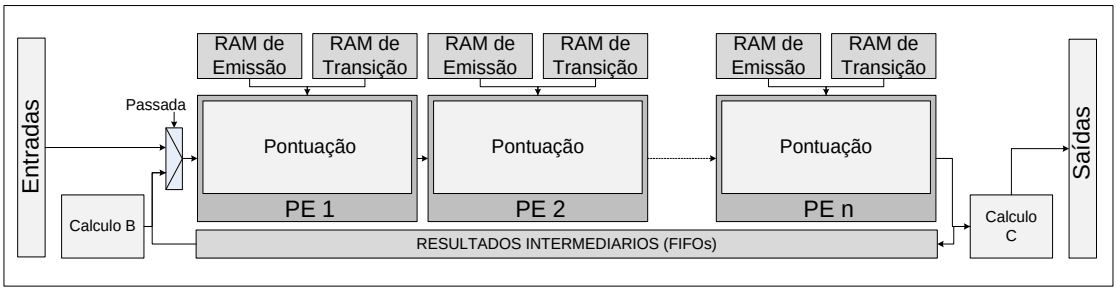


Figura 3.16: Diagrama de blocos de um acelerador baseado em FPGA.

Benkrid *et al.* [7] propõe um arranjo de 90 EPs capaz de processar uma sequência de 1024 elementos e compará-la com um profileHMM de até 1440 nodos. A estratégia que eles usam para a implementação do AS é a eliminação do estado de realimentação J para eliminar as dependências de dados que impedem a paralelização do algoritmo. A eliminação do estado J permite que o AS calcule aproximadamente uma antidiagonal da matriz de programação dinâmica em cada ciclo de relógio. Na saída do sistema, a pontuação para o melhor alinhamento local da sequência com o profileHMM é obtida. O seu Arranjo Sistólico foi implementado numa FPGA Xilinx Virtex 2, e atingiu uma freqüência máxima de operação de 100MHz. A medida de desempenho utilizada por eles é o número de células que podem calcular por segundo (CUPS – *Cell Updates per Second*), que é o número de elementos da matriz de programação dinâmica calculado em paralelo. [7] reporta um desempenho entre 5.2 e 9 GCUPS (*Giga Cell Updates per Second*), quando o sistema trabalha na freqüência máxima. A Figura 3.17 mostra o diagrama de blocos apresentado por eles. A principal desvantagem dessa arquitetura é a perda de sensibilidade devido à eliminação do estado J, pois, ao eliminar o estado J, somente a pontuação para o melhor alinhamento local da sequência com o profileHMM atual é calculada, e podem aparecer sequências com múltiplos alinhamentos locais onde a soma da pontuações de todos esses alinhamentos faz com que a sequência tenha uma pontuação total grande.

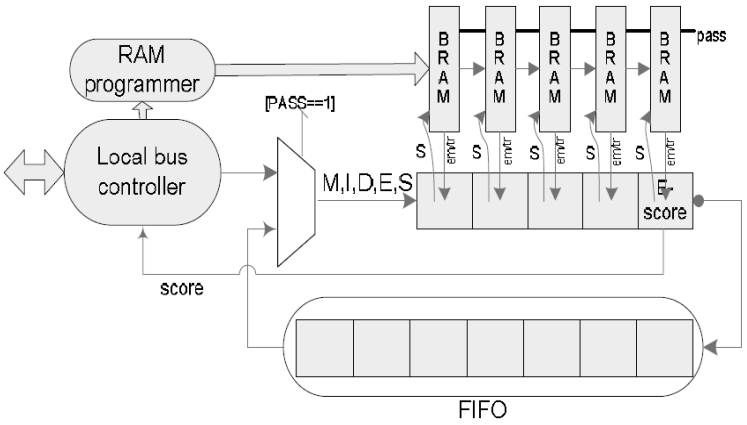


Figura 3.17: Diagrama de blocos proposto por Benkrid et al.

Maddimsetty *et al.* [8][10] propõe uma variação da estratégia adotada por [7], na qual o estado J é eliminado, mas duas instâncias do arranjo sistólico ligadas por um estado intermediário L (*linker*) são criadas. Esta estratégia procura reduzir as perdas introduzidas ao eliminar o estado J e consiste em calcular duas pontuações para cada sequência e comparar estas pontuações. Caso a segunda pontuação calculada seja maior do que a primeira, a sequência provavelmente possui múltiplos alinhamentos locais e é enviada para o segundo estágio de processamento. No trabalho, estima-se que com a arquitetura proposta pode-se obter uma redução de 10 vezes nas aparições de falsos positivos e de falsos negativos, sem prejudicar o desempenho geral do sistema. Baseado em suposições acerca da tecnologia utilizada, eles reportam um tamanho de 50 EPs operando numa freqüência de 200MHz e com um desempenho de 5 até 20GCUPS. No trabalho, o tamanho maior da sequência *query* e o número de nodos do profileHMM não são reportados. A Figura 3.18 apresenta o diagrama geral do sistema proposto por eles.

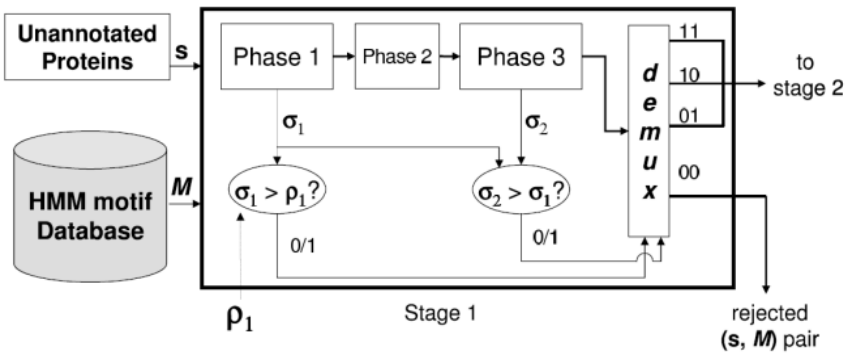
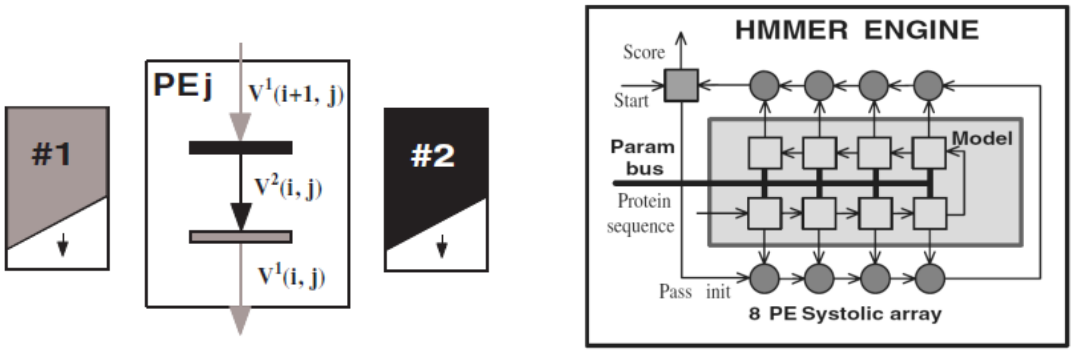


Figura 3.18: Arquitetura proposta por Maddimsetty et al. [8][10].

Jacob et al. [16] também elimina o estado J, mas propõe uma arquitetura que compara várias sequências com um profileHMM simultaneamente ao dividir o EP em várias etapas de *pipeline*. Esta comparação simultânea eleva o desempenho do Arranjo Sistólico e aproveita o paralelismo existente entre comparações de diferentes sequências com um mesmo profileHMM. No seu trabalho, eles implementam EPs conformados por 4 etapas de *pipeline* atingindo uma freqüência máxima de 180MHz e um desempenho de 10 GCUPS. Além disso, o trabalho reporta que, ao introduzir estes estágios de *pipeline*, o incremento na área do acelerador é de apenas 9%, enquanto a velocidade aumenta num fator de 2.5. A arquitetura proposta foi implementada numa FPGA Xilinx Virtex 2 6000 e suporta 68 EPs, sequências com tamanhos menores do que 1024 elementos e profileHMMs até de 544 nodos. No momento da publicação do trabalho (2007) o suporte para profileHMMs com mais de 544 nodos não estava implementado. As Figuras 3.19 a e b apresentam a estratégia de *pipeline* proposta onde os elementos das sequências são entrelaçados e o diagrama de blocos da arquitetura proposta.



(a) (b)

Figura 3.19: a) Estratégia de *pipeline* proposta para os EPs. b) Diagrama de blocos do sistema implementado [16].

Em Derrien *et al.* [30], uma metodologia para implementar *pipelines* dentro dos EPs é proposta baseando-se na representação matemática do Algoritmo de Viterbi, na qual o laço interno do algoritmo é transformado num Sistema de Equações Afins Recorrentes (SARE). Aplicando essa metodologia, eles realizam uma Exploração do Espaço de projeto utilizando como plataforma de prototipagem uma FPGA Xilinx Spartan 3 4000. Como resultado da exploração, eles obtêm um AS conformado por 22 EPs, com uma freqüência de relógio máxima de 66MHz e um desempenho de aproximadamente 1.3 GCUPS. A Figura 3.20 apresenta a arquitetura obtida através da metodologia proposta no trabalho.

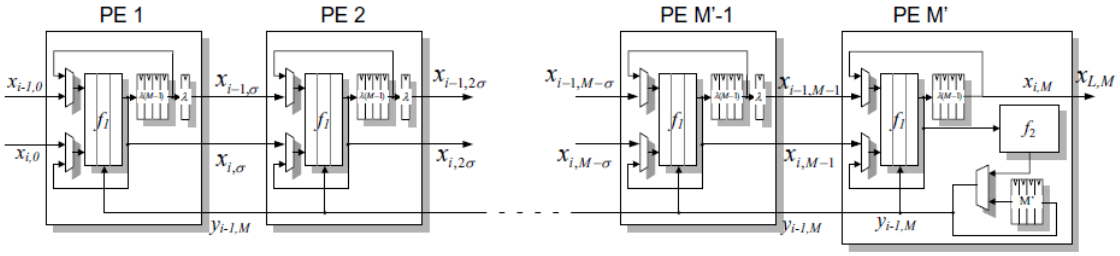
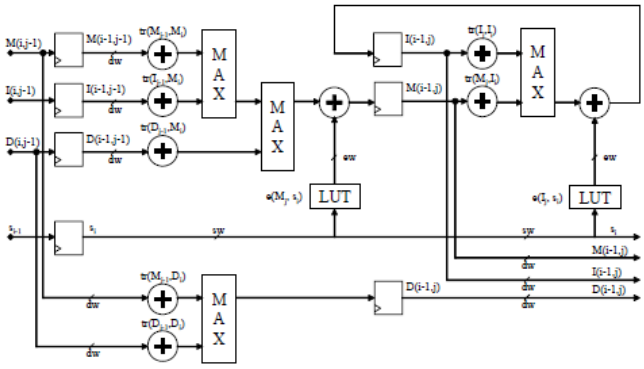
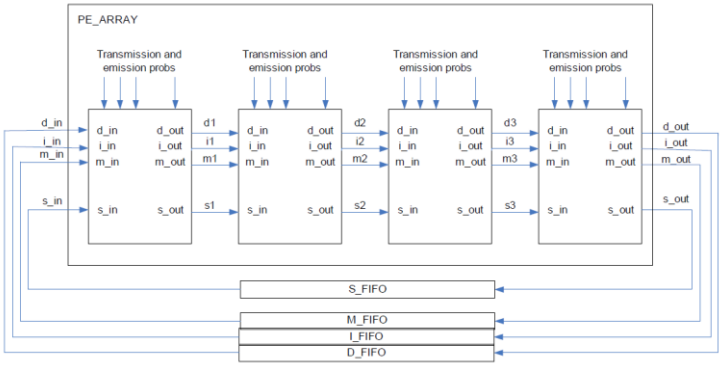


Figura 3.20: Arquitetura obtida por Derrien et al.

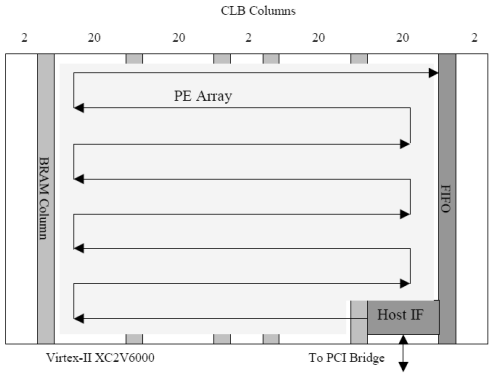
No trabalho de Oliver *et al.* [12], o arranjo tradicional é implementado, eliminando o estado J e calculando a pontuação para profileHMMs longos através de várias iterações e da inclusão de FIFOs para armazenar os resultados intermediários das iterações anteriores. No trabalho, eles implementam um sistema composto de um arranjo de 72 EPs funcionando em uma freqüência de 74MHz, e obtêm um desempenho estimado de aproximadamente 4 GCUPS. A síntese do sistema foi feita numa FPGA Xilinx Virtex 2 6000. As Figuras 3.21 a, b e c apresentam a arquitetura interna do EP proposto no trabalho, o diagrama geral de blocos do sistema e o *floorplan* do sistema dentro da FPGA.



(a)



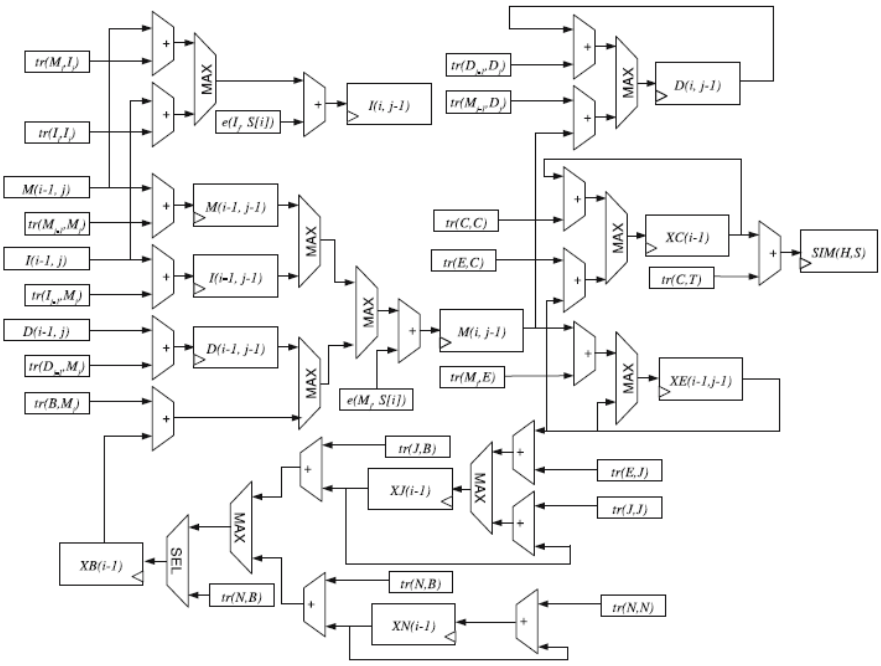
(b)



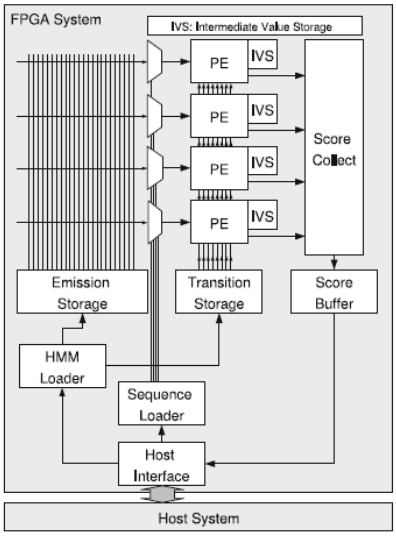
(c)

Figura 3.21: a) Arquitetura do EP. b) Diagrama de blocos. c) Floorplan na Virtex 2 6000 [12].

Walters *et al.* [14][31] fazem a primeira implementação de um plan7 completo sem eliminar o estado de realimentação J. Ao não eliminar a dependência de dados imposta pelo estado J, eles exploram o paralelismo entre comparações de sequências diferentes contra o mesmo HMM. O seu EP realiza o cálculo completo da pontuação da sequência e requer o armazenamento das pontuações da matriz de programação dinâmica para o elemento i-1 da sequência. Eles replicam esse EP para calcular a pontuação de até 10 sequências em paralelo. A implementação do sistema foi feita em uma FPGA Xilinx Spartan 3 1500 com 10 EP e um comprimento máximo do profileHMM de 256 nodos. A freqüência máxima para cada EP é de 70MHz, o que representa um desempenho máximo do sistema inteiro de 700 MCUPS. A implementação proposta pelo trabalho elimina as perdas de precisão introduzidas ao eliminar o estado J e inclui o acelerador dentro de cada um dos nodos de um cluster de computação, escalando o desempenho linearmente ao agregar um número maior de nodos no cluster. As Figuras 3.22 a e b apresentam a arquitetura dos EPs e o diagrama de bloco do sistema implementado.



(a)



(b)

Figura 3.22: a) Arquitetura dos EPs propostos. b) Diagrama de blocos do sistema completo [14][31].

Em [9], uma análise do software HMMER determina que o estado de realimentação J é raramente tomado, e uma nova unidade funcional que determina quando se apresenta a transição ao estado J é proposta. Quando ocorre uma transição para o estado J, a unidade de controle recalcula o valor do vetor B e recalcula os valores que foram computados com um valor errado de B. Essa unidade de controle também visa o armazenamento dos valores intermediários que são necessários tanto para o cálculo de profileHMMs compridos como

para o recálculo dos valores certos caso uma transição ao estado J esteja presente. A implementação deste sistema foi feita numa FPGA Xilinx Virtex 5 110-T com um número máximo de 25 EPs funcionando em uma freqüência de 130MHz. O desempenho reportado pelo trabalho é de 3.2 GCUPS e não são reportados tamanhos máximos da sequência nem do profileHMM. A Figura 3.23 apresenta o diagrama de blocos proposto no trabalho.

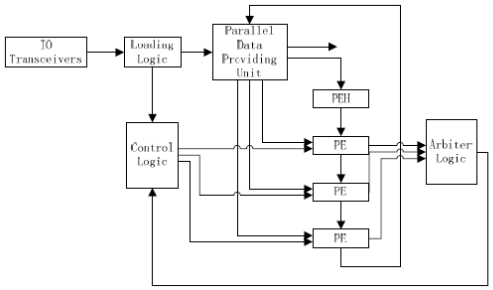


Figura 3.23: Diagrama de blocos do sistema proposto em [9].

De maneira análoga às outras arquiteturas apresentadas nesta seção, nosso trabalho não calcula o alinhamento em hardware, mas só fornece na sua saída a pontuação da comparação da sequência com o profileHMM e os dados necessários para realizar o cálculo do alinhamento em software através da aplicação do Algoritmo das Divergências. O fornecimento destes dados ajuda o software a reduzir o número de células que ele tem que calcular, permitindo uma aceleração ainda maior com respeito às alternativas de aceleração por hardware já apresentadas. A Tabela 3.1 apresenta um sumario das arquiteturas presentes nos trabalhos relacionados e de suas principais características.

Tabela 3.1: Sumario dos Trabalhos relacionados.

ref	**# EPs**	**# nodos máximo no profileHMM**	**Tamanho Máximo da Seq.**	**Plan7 Completo**	**Relógio (MHz)**	**Performance (GCUPS)**	**Ganho**	**FPGA**	**Características Especiais**
[7]	90	1440	1024	N	100	9	247	Xilinx 2VP100	Elimina o Estado J
[8] [10]	50	---	---	N	200	5 to 20	---	Not Synthesized	Introduz um acelerador de duas etapas.
[12]	72	1440	8192	N	74	3.95	195	XC2V6000	Elimina os Estados J e B.
[14] [31]	10	256	---	Y	70	7	300	XC3S1500	Implementa o plan7 completo
[30]	50	---	---	N	66	1.3	50	Xilinx Spartan 3 4000	Propõe uma metodologia para paralelizar os EPs
[16]	68	544	1024	N	180	10	190	Xilinx Virtex II 6000	Divide o EP em estados de *pipeline*
[9]	25	---	---	Y	130	3.2	56.8	Xilinx Virtex 5 110-T	Faz execução especulativa.

4 - PROJETO DO ACELERADOR

Neste capítulo são propostas tanto a Metodologia utilizada para desenvolver o projeto quanto a metodologia utilizada para realizar a verificação do mesmo. Além disso, um fluxo de execução é proposto para o sistema completo, visando tanto à execução em hardware para a produção da pontuação e das divergências para uma sequência quanto ao processamento desses dados no software para gerar o alinhamento. Também é proposta uma arquitetura para o acelerador de hardware, dividido em duas etapas: uma que produz a pontuação do Algoritmo de Viterbi e outra para o cálculo do Algoritmo das Divergências. Finalmente são propostas as medidas de desempenho para avaliar o funcionamento do acelerador e determinar o ganho obtido a partir do aceleramento e filtragem realizados pelo hardware e a economia feita ao processar somente a região de interesse das matrizes de programação dinâmica.

Para garantir o sucesso no desenvolvimento de um acelerador em hardware que implemente o Algoritmo de Viterbi conjuntamente com o Algoritmo das Divergências foram definidas duas metodologias de projeto. A primeira atinge o processo de análise dos algoritmos a serem implementados na arquitetura, enquanto a segunda atinge o processo de verificação seguido no processo de desenvolvimento.

4.1 – METODOLOGIA DE PROJETO

Como uma primeira etapa no desenvolvimento do acelerador, foi implementada uma versão em C/C++ para a especificação do algoritmo de Viterbi incluindo o algoritmo das divergências. Isso foi feito para avaliar os alinhamentos produzidos por ele, e para ratificar que esses alinhamentos são iguais aos produzidos pelo software HMMER. Depois da implementação deste "*golden model*", foi feita em SystemC a simulação em alto nível de uma primeira arquitetura para o acelerador. Esta arquitetura implementa o acelerador como um Arranjo Sistólico ligado por filas bloqueantes, mas não considera a possibilidade de profileHMMs mais longos do que o número de EPs implementados nele. Esta primeira arquitetura foi refinada para incluir as memórias FIFO para resultados intermediários, e desta forma suportar profileHMMs longos. Finalmente a arquitetura implementada em SystemC foi traduzida manualmente para a linguagem de descrição de Hardware VHDL e

foi simulada através de um *testbench* para avaliar seu funcionamento e fazer a posterior verificação dos resultados obtidos. A figura 4.1 apresenta o fluxo de projeto adotado para a implementação da arquitetura proposta.

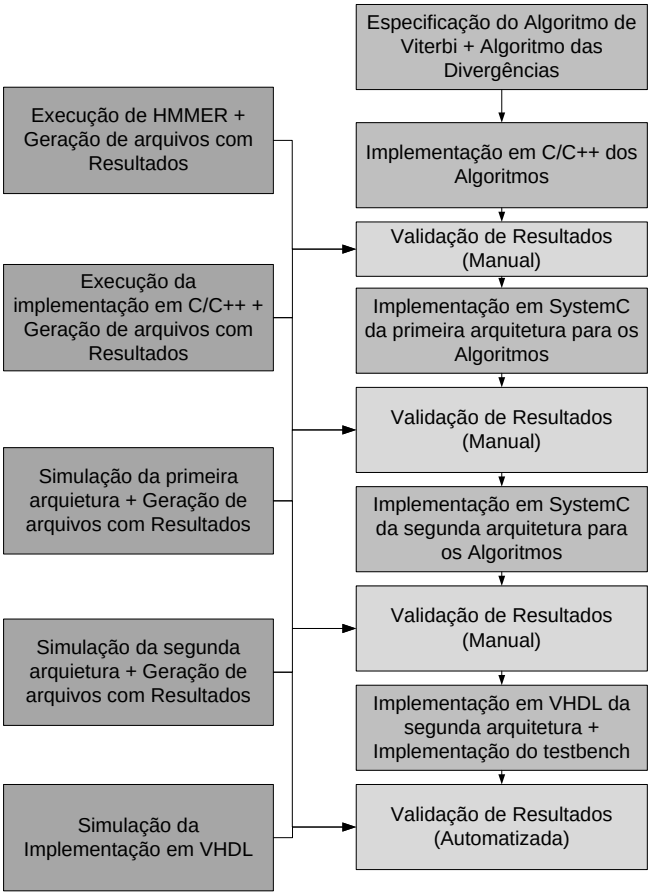


Figura 4.1: Fluxo de projeto para a implementação da arquitetura.

4.2 – METODOLOGIA DE VERIFICAÇÃO

Para a verificação das diferentes implementações feitas para o algoritmo de Viterbi foram adotadas varias estratégias. Na verificação da primeira implementação (feita em C/C++), foram gerados arquivos de texto com as pontuações e os alinhamentos gerados, e posteriormente foram comparados manualmente com as pontuações e os alinhamentos gerados pelo Software HMMER. Para a implementação em SystemC das duas arquiteturas seguintes, uma estratégia similar foi adotada, arquivos de texto com a saída de dados das arquiteturas foram gerados e essas saídas foram comparadas para avaliar os resultados. Para a arquitetura codificada em VHDL se tentou adotar uma estratégia similar as anteriores para avaliar os resultados da implementação, mas como a codificação foi feita

no nível de abstração RTL, a estratégia de verificação devia atingir todos os detalhes de sincronização entre os elementos de processamento. Além disso, o processo de verificação devia garantir que os acessos nas memórias que armazenam as probabilidades estivessem certos para cada ciclo de relógio da execução. Para garantir isso, foi incluído um processo de verificação dentro do *testbench*, o qual compara tanto os sinais importantes dependentes de sincronização (leitura de memórias, processos de carregamento de bancos de registradores, processos de carregamento de profileHMM nas memórias) quanto os resultados finais para as pontuações na saída da arquitetura proposta. Estas comparações são feitas com os arquivos de texto gerados para as anteriores arquiteturas e fecham o ciclo de verificação do funcionamento do sistema ao incorporar os modelos de mais alto nível no processo de verificação.

4.3 - FLUXO DE EXECUÇÃO MODIFICADO

Como foi explicado no capítulo 3, o software HMMER faz a comparação de sequências com profileHMMs através da implementação do Algoritmo de Viterbi para a arquitetura plan7. O propósito deste trabalho é acelerar a execução das comparações de sequências, ao implementar um acelerador em hardware baseado tanto no Algoritmo de Viterbi quanto no Algoritmo das Divergências explicado na Seção 3.3. Para fazer a aceleração dessa execução, uma etapa de hardware é incluída dentro do sistema. Além disso, é necessário modificar o fluxo de execução da aplicação, para fazer chamadas aos módulos de hardware. Essas chamadas correspondem ao carregamento das probabilidades de transição e emissão para um profileHMM, à inserção dos elementos para cada uma das sequências *query* e à recuperação dos dados de saída do acelerador. A Figura 4.2 apresenta as diferentes etapas do fluxo de execução proposto.

O software é executado através de um comando de console. Nesse comando são especificados tanto o profileHMM quanto o arquivo de sequências que serão comparadas a ele. O software então executa o processo de carregamento das probabilidades necessárias nas memórias do sistema, começando pelas probabilidades especiais (as transições N->N, N->B, C->C, E->C e C->T). Depois do carregamento das probabilidades especiais, o sistema carrega as probabilidades de transição e as probabilidades de emissão. A etapa seguinte acessa o arquivo de sequências: se houver sequências para processar, o sistema entra na etapa de processamento; se não, o sistema entrega os resultados do processamento

e finaliza a execução. Na etapa de processamento, os elementos da sequência *query* atual são inseridos no hardware acelerador, tantas vezes quantas iterações tenham que ser executadas (caso o profileHMM tenha mais nodos do que os implementados na FPGA). O hardware sinaliza ao software indicando que uma sequência foi processada e o software lê o dado da pontuação produzido pelo hardware. Se esta pontuação for maior do que o limiar definido pelo usuário, o software faz a leitura dos dados produzidos pelo hardware para o Algoritmo das Divergências, se não, o software olha se tem mais sequências para processar. Caso não se tenham mais sequências para ser processadas, o software encerra a execução. Se a pontuação da sequência for boa, o software faz o reprocessamento da região de interesse para a sequência *query* atual, produz o alinhamento e passa para a próxima sequência.

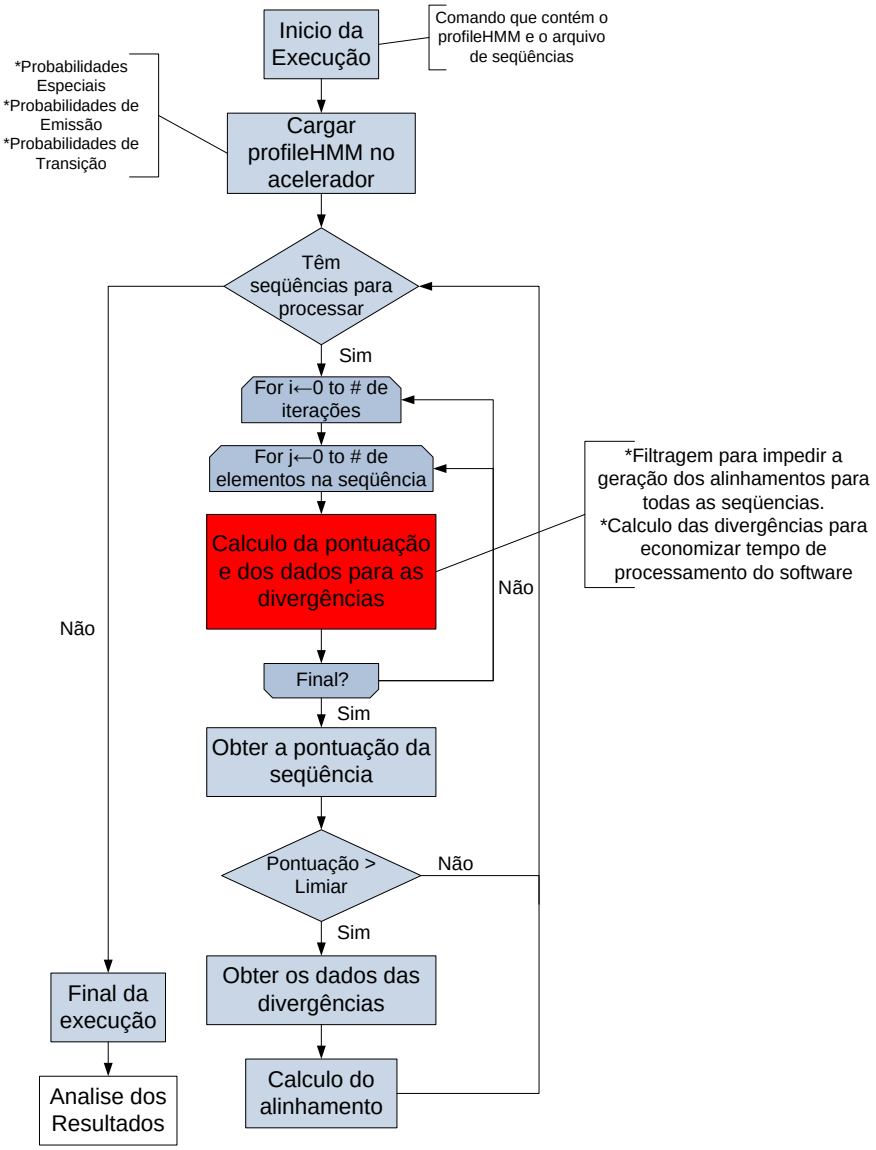


Figura 4.2: Fluxo de execução modificado

4.4 - DEPENDÊNCIAS DE DADOS E ALGORITMO DAS DIVERGÊNCIAS

No Algoritmo de Viterbi para a arquitetura Plan7 são encontradas dependências de dados que afetam diretamente a maneira de implementá-lo em hardware. A Figura 4.3 apresenta as dependências de dados para o Algoritmo de Viterbi do HMMER. Nessa figura, cada célula (i,j) das matrizes (M, I e D) corresponde à pontuação para o alinhamento produzido para os primeiros i elementos da sequência ($S_{1..i}$) e que termina no nodo j do profileHMM ($P_{1..j}$) , seja no estado M, I ou D. Como pode ser observado do Algoritmo 3.2, a pontuação para o estado M(i,j) depende dos valores (i-1,j-1) das matrizes M, I e D, além do valor (i-1) do vetor B. A matriz I depende dos valores (i-1,j) das matrizes I e D, e a matriz D depende dos valores (i,j-1) das matrizes M e D. Temos também que o vetor E tem dependência direta com as pontuações da fila inteira de valores para M(j,$j_{1...k}$) (parte cinza da Figura 4.3) e que o vetor J depende do seu valor anterior e da pontuação atual para o vetor E. Finalmente observamos que o valor para o vetor B depende do vetor J. É essa dependência de dados (B←J←E←M←B←J) que impede a geração de um arranjo sistólico para a paralelização do algoritmo.

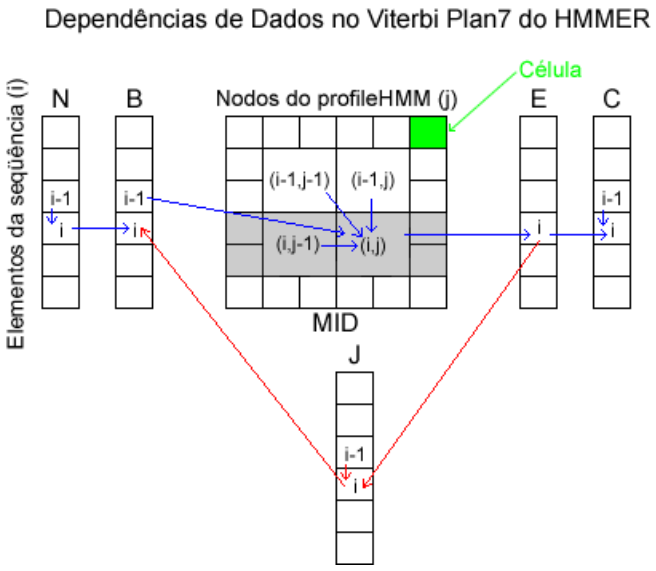


Figura 4.3: Dependências de dados no Viterbi do HMMER.

Uma simplificação no Algoritmo de Viterbi é proposta inicialmente em [12], onde é eliminado o estado J da arquitetura do Plan7, e portanto a detecção *multi-hit*, para simplificar o problema da dependência de dados. Além da simplificação proposta, o nosso trabalho reagrupa o Algoritmo de Viterbi, modificando o cálculo do vetor E para ser feito

em cada nodo do profileHMM para cada elemento da sequência, como o maximo entre o seu valor calculado no nodo anterior e o valor calculado no nodo atual. Isso foi feito a fim de simplificar ainda mais a dependência de dados e evitar calcular uma linha inteira para o estado M como pré-requisito para o cálculo do valor E_i.O Algoritmo 4.1 apresenta a versão modificada do Algoritmo de Viterbi. A Figura 4.4 apresenta as novas dependências de dados presentes no algoritmo. Da figura, percebe-se que agora as células das matrizes dependem no máximo de valores anteriores das mesmas matrizes, que não existem laços fechados que possam afetar o funcionamento do algoritmo e que o valor para o vetor E somente depende do valor anterior e do valor atual dele.

Modificações:
- *Sem estado J*
- *Cálculo do vetor E dentro do laço interno*
- *Cálculo dos vetores N e B condensado, e feito antes do laço interno*

Entradas:
- *H: profileHMM de k nodos (dado pelas probabilidades tr() e em())*
- *S: Sequência de comprimento n aminoácidos*

Saídas:
- *sim: pontuação de similaridade entre H e S*

Algoritmo:

for j←0 to k do //Inicialização das matrizes de PD
  $M_{0,j} \leftarrow -\infty$
  $I_{0,j} \leftarrow -\infty$
  $D_{0,j} \leftarrow -\infty$
end for
for i←0 to n do
  $M_{i,0} \leftarrow -\infty$
  $I_{i,0} \leftarrow -\infty$
  $D_{i,0} \leftarrow -\infty$
  $E_{i,(0)} \leftarrow -\infty$
end for
$B_0 \leftarrow tr(N, B)$
$C_0 \leftarrow -\infty$

$$
\begin{aligned}
&for\ i \leftarrow 1\ to\ n\ \text{//para cada aminoácido da sequência}\\
&\quad B_i = B_{i-1} + tr_{N,N}\\
&\quad for\ j\ \leftarrow 1\ to\ k\ \text{//Para cada nodo do profileHMM}\\
&\qquad M_{i,j} = em_{(M_j,S_i)} + max\begin{cases} M_{i-1,j-1} + tr_{M_{j-1},M_j} \\ I_{i-1,j-1} + tr_{I_{j-1},M_j} \\ D_{i-1,j-1} + tr_{D_{j-1},M_j}, \\ B_{i-1} + tr_{B,M_j} \end{cases}\\
&\qquad I_{i,j} = em_{(I_j,S_i)} + max\begin{cases} M_{i-1,j} + tr_{M_j,I_j} \\ I_{i-1,j} + tr_{I_j,I_j} \end{cases}\\
&\qquad D_{i,j} = max\begin{cases} M_{i,j-1} + tr_{M_{j-1},D_j} \\ D_{i,j-1} + tr_{D_{j-1},D_j} \end{cases}\\
&\qquad E_{i,(j)} = max\begin{cases} E_{i,(j-1)} \\ M_{i,j} + tr_{M_j,E} \end{cases}\\
&\quad end\ for\\
&\quad C_i = max\begin{cases} C_{i-1} + tr_{C,C} \\ E_{i,(k)} + tr_{E,C} \end{cases}\\
&end\ for\\
&similaridade = C_n + tr_{C,T}
\end{aligned}
$$

Algoritmo 4.1: Modificação do Algoritmo de Viterbi para eliminar a dependência de dados criada pelo estado J.

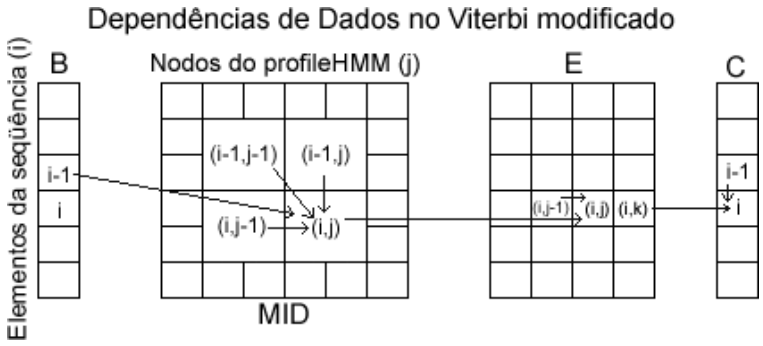


Figura 4.4: Dependências de dados no algoritmo modificado.

Além das dependências de dados dentro do Algoritmo de Viterbi, o Algoritmo das Divergências depende também do índice produzido pelos máximos das relações de recorrência. Como ele foi pensado para implementação em hardware, seus valores somente dependem de valores anteriores das divergências, do valor de i, do valor de j e dos índices produzidos no cálculo atual das pontuações P(i,j). Portanto, não produz novas dependências de dados que impossibilitem a paralelização e posterior implementação do acelerador.

4.5 - ARQUITETURA PROPOSTA

A arquitetura proposta consiste em um acelerador baseado no algoritmo modificado apresentado na seção anterior e no Algoritmo das Divergências apresentado no Capítulo 2. A arquitetura consiste em um arranjo de EPs interconectados, cada um dos quais calcula tanto a pontuação para a coluna *j* das matrizes de programação dinâmica quanto a informação para as divergências produzidas por essa pontuação.

Cada EP possui por duas etapas que realizam cálculos de forma independente. A primeira etapa realiza o cálculo das pontuações para as matrizes de programação dinâmica, e produz os índices dos valores máximos escolhidos pelo algoritmo de Viterbi (vide Algoritmo 3.5). A segunda etapa recebe os índices produzidos pela primeira etapa e calcula os dados para as divergências. Cada EP calcula uma das colunas da matriz de programação dinâmica já que são organizados em um arranjo sistólico e cada um deles corresponde ao nodo *j* do profileHMM. Outra afirmação válida sobre dos EPs, é que eles não armazenam as pontuações para todas as células das matrizes, somente os valores necessários para produzir a pontuação *P(i,j)*. A estratégia anterior funciona porque a pontuação total do alinhamento é determinada pelo valor do vetor E (Algoritmo 4.1) calculado ao longo do arranjo à medida que a sequência o atravessa.

A Figura 4.5 representa as anti-diagonais da matriz de programação dinâmica e sua relação com os EPs do arranjo sistólico, para um profileHMM com um número de nodos igual ao número de EPs no arranjo (somente uma iteração na execução). Na figura, os EPs relacionados com células marcadas com $\underline{P}$ estão ociosos, as setas demarcam as anti-diagonais das matrizes e as células verdes correspondem às das matrizes de programação dinâmica para cada elemento da sequência S_i. O *pipeline* é preenchido gradualmente, diminuindo em cada iteração do algoritmo o número de EPs ociosos até atingir o regime. Quando os elementos da sequência acabam, o *pipeline* se esvazia gradualmente aumentando em cada iteração do algoritmo o número de nodos ociosos.

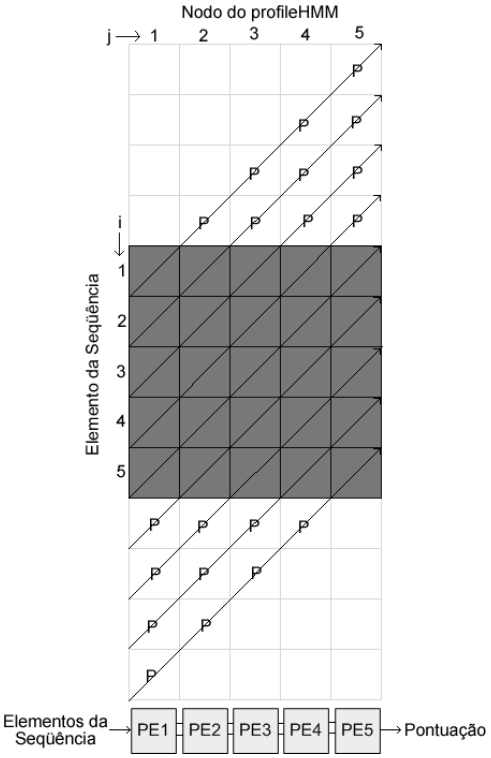


Figura 4.5: Correspondência de um arranjo de cinco EPs com cada nodo de um profileHMM com 5 nodos, visando uma posição aleatória da sequência query.

Como comumente são achados profileHMMs com um número maior de nodos do que o implementado em hardware, é necessário implementar um sistema capaz de processar as comparações de sequências com esse tipo de HMMs. Dentro do nosso sistema, a estratégia adotada é separar a computação deste tipo de profileHMMs em múltiplas iterações da sequência através do hardware. Para atingir isso, são divididas as matrizes de programação dinâmica em bandas verticais com comprimento C, onde C é o número de EPs implementados no hardware. Cada iteração computa as pontuações para toda a sequência somente em C colunas das matrizes de programação dinâmica e armazena a saída do ultimo EP em memórias tipo FIFO para utilizá-la como entrada para o primeiro EP na próxima iteração. O armazenamento dos dados intermediários é requerido pela dependência de dados no Algoritmo de Viterbi, explicada na seção anterior. A Figura 4.6 apresenta a divisão em iterações do processamento e a relação dos EPs com o cálculo das matrizes. Uma outra consideração no momento de implementar a arquitetura, foi extrair o cálculo dos vetores B e C para fora do EP garantindo que o arranjo sistólico seja o mais regular possível. Isso é possível pois os elementos das sequências (laço *for* exterior no Algoritmo 4.1) são inseridos pelo software (os vetores B e C são calculados no interior deste laço).

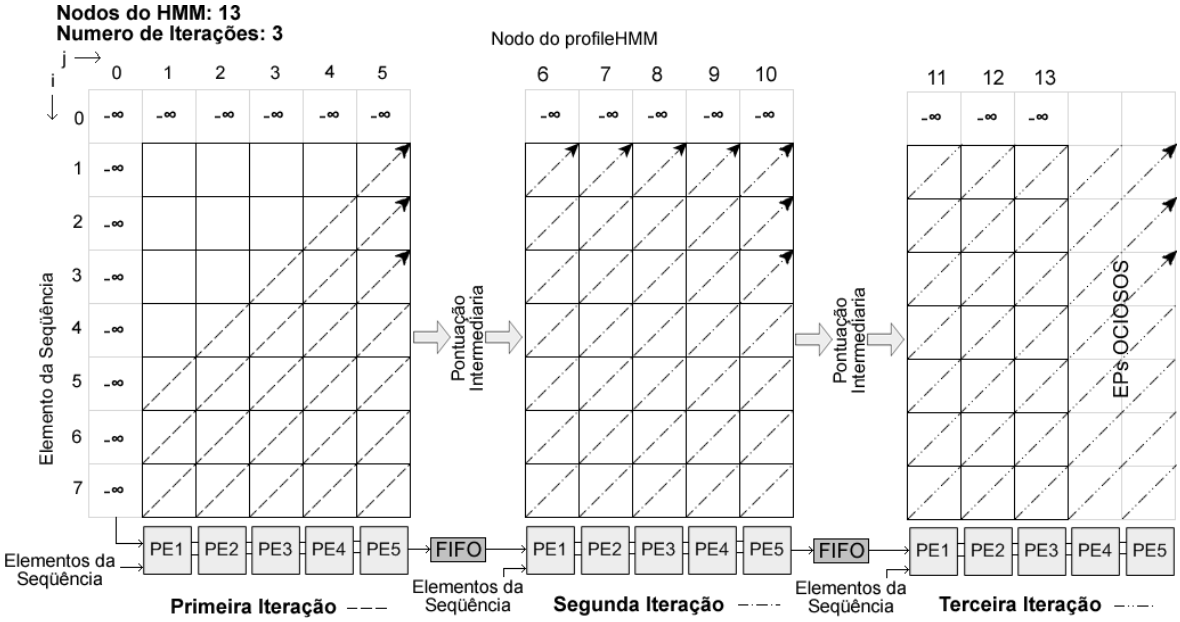


Figura 4.6: Processamento em varias iterações de um profileHMM com 13 nodos em um hardware com 5 EPs.

A arquitetura proposta consiste então em um conjunto de EPs interconectados. Cada um dos EPs possui uma memória para armazenar as probabilidades de emissão e uma para armazenar as probabilidades de transição. A unidade que calcula o vetor B é alocada antes do primeiro EP e a unidade que calcula o vetor C é alocada depois do último. As pontuações de saída do último EP para iterações intermediárias são armazenadas em FIFOs para posterior leitura. Na entrada de dados do primeiro EP é necessária a inclusão de um multiplexador para fazer a escolha entre os dados que vêm do software (para a primeira iteração) e os dados que vêm das FIFOs (para as outras iterações). A Figura 4.7 apresenta o diagrama geral da arquitetura. Uma descrição mais detalhada da estrutura e do funcionamento dos seus componentes é feita no capítulo 5.

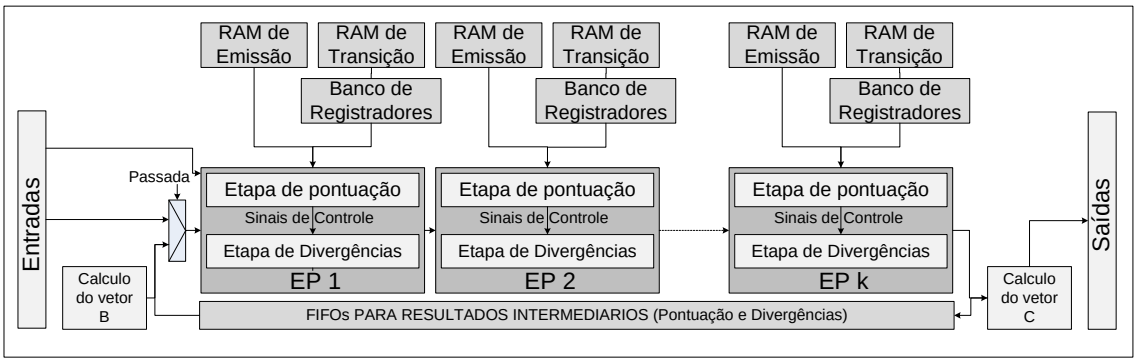


Figura 4.7: Diagrama de blocos da arquitetura proposta.

4.5.1 - Etapa de Cálculo da Pontuação

É a etapa da arquitetura que calcula as pontuações para o Algoritmo de Viterbi modificado. Cada um dos EPs inclui uma dessas etapas e calcula as pontuações de cada elemento da sequência para um nodo do profileHMM. As entradas desta etapa são as pontuações calculadas pela etapa de pontuação do EP imediatamente anterior para o elemento atual da sequência (o elemento da sequência passa por todos os EPs do Arranjo). As saídas são as pontuações calculadas para o elemento atual pela etapa de pontuação do EP atual. A etapa de pontuação também gera os sinais de controle para o cálculo do Algoritmo das Divergências. Estes sinais de controle serão enviados para a etapa das divergências, com atraso de um ciclo de relógio a fim de melhorar a freqüência de relógio ao reduzir o caminho crítico combinacional do hardware. A Figura 4.8 apresenta o diagrama de componentes da etapa, os termos para as transições e emissões vêm do *profileHMM* e como são alimentados no hardware depende da implementação, que será discutida no capítulo 5.

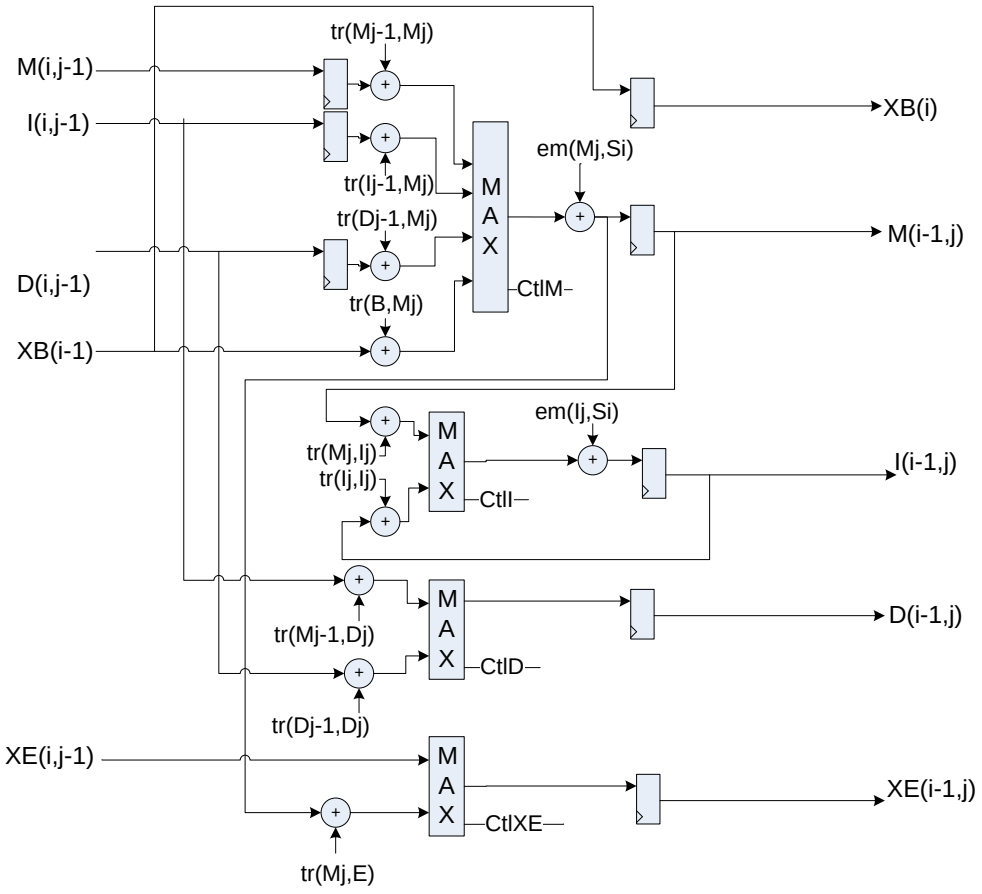
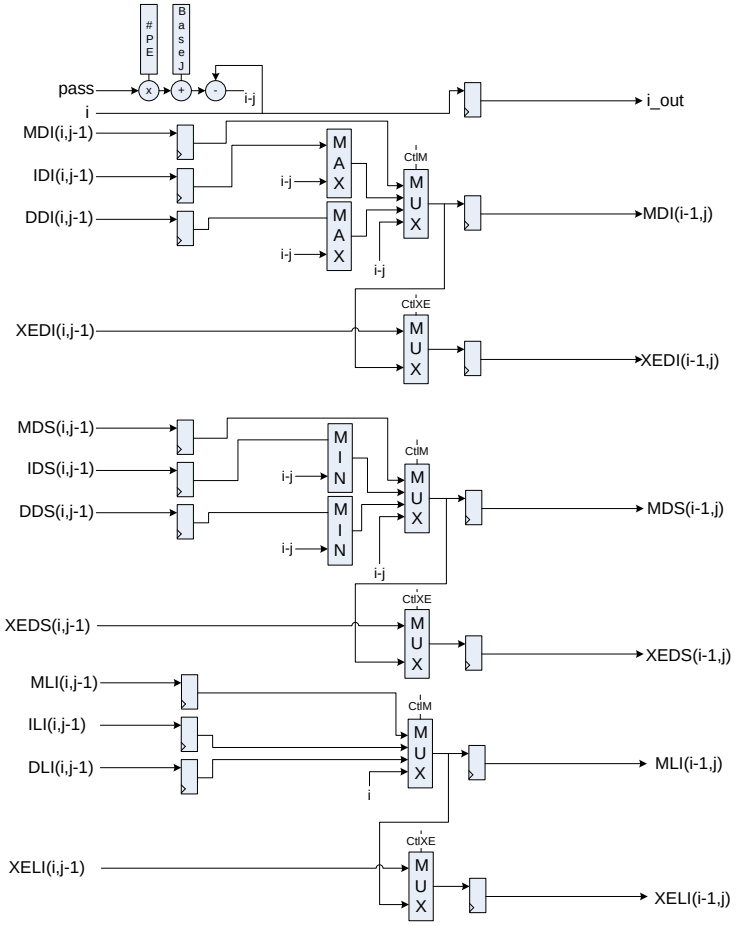


Figura 4.8: Etapa de pontuação dos EPs.

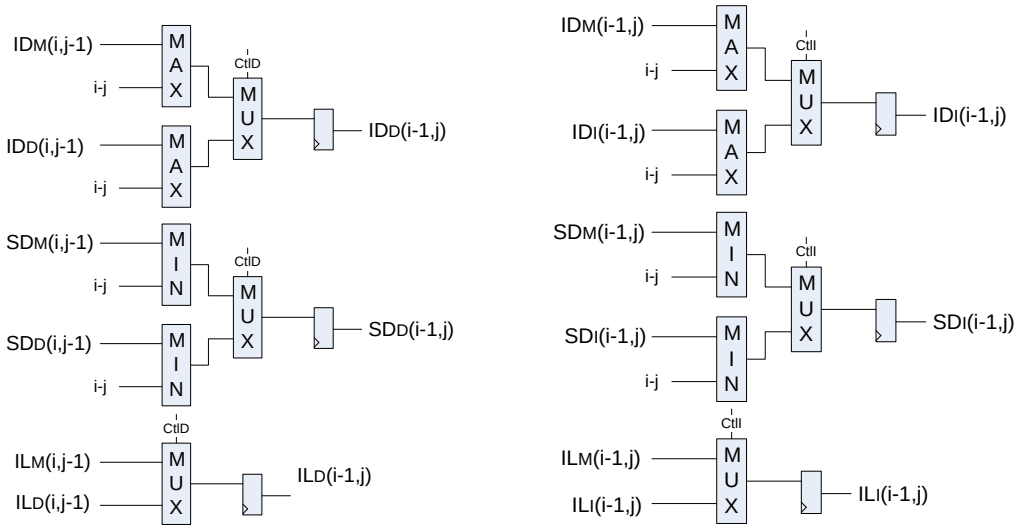
4.5.2 - Etapa de Cálculo das Divergências

A etapa das divergências realiza a implementação em hardware do Algoritmo das Divergências, apresentado na Seção 3.3. As entradas da etapa consistem nas divergências de saída da etapa anterior (divergências do EP anterior para o elemento atual da sequência) e nos sinais de controle emitidos pela etapa de cálculo de pontuações do EP. Estes sinais de controle indicam qual das escolhas foi feita nas relações de recorrência dos máximos, o que reflete no nodo inicial e final para o cálculo da região de interesse no algoritmo.

O hardware também precisa do índice do EP no arranjo e do número total de EPs no arranjo. Esses parâmetros são utilizados para o processo de inicialização (Algoritmo 3.4) dos registradores no começo do algoritmo e ao mudar de iteração no caso dos profileHMMs longos. A Figura 4.9 apresenta o diagrama de componentes da etapa das divergências.



(a)



(b) (c)

Figura 4.9: Etapa para o cálculo das divergências do EP. a) Para o estado M e o estado E. b) Para o estado I. c) Para o estado D.

Alem do cálculo das divergências no EP, uma etapa deve ser inserida na unidade de cálculo do vetor C. Essa etapa toma o valor do índice$_C$ gerado pela unidade de comparação para o valor do vetor C e aplica o equivalente em hardware da parte para o cálculo das divergências no Algoritmo 3.6. A unidade de cálculo para o vetor C recebe como entradas a saída da pontuação para o estado E, o índice do elemento atual da sequência (*i*), as probabilidades de transição *tr(C,C)* e *tr(E,C)* e as saídas do último EP para as divergências. Como saídas a unidade entrega a pontuação para o alinhamento terminando no elemento S_i e o cálculo final feito para as divergências do estado C. A Figura 4.10 apresenta a arquitetura proposta para a unidade de cálculo do vetor C.

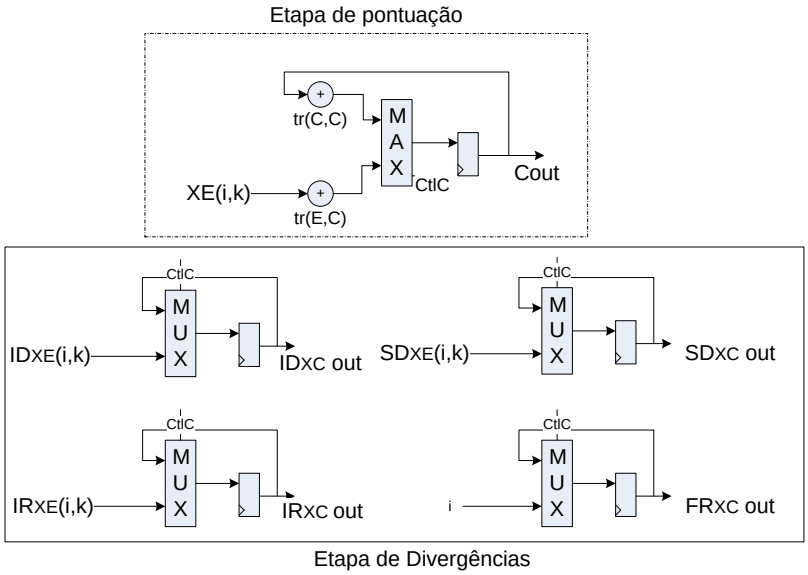


Figura 4.10: Unidade de Cálculo do vetor C.

4.6 - MEDIDAS DE DESEMPENHO PROPOSTAS

Para avaliar o desempenho do acelerador proposto é necessário definir uma métrica que permita compará-lo com o software sem aceleração. A medida escolhida para este trabalho foi o tempo de processamento dos elementos do sistema. O tempo de execução do software sem acelerar será comparado com o tempo de execução do hardware adicionado ao tempo de processamento do software somente na região de interesse. Um número elevado de testes é feito para garantir que os resultados obtidos sejam confiáveis e que o sistema proposto de fato acelere a execução das buscas no software HMMER.

As quantidades a seguir são de grande importância:

- O número total de sequências *query*, $S_{t;}$
- O número total de profileHMMs, P_t;
- O número total de comparações feitas, N_t;
- O tempo que o software HMMER leva para realizar uma comparação da sequência S_i com o profileHMM j, $t_{s(i,j)}$;,
- O tempo total de execução do software sem acelerar, T_{ss};
- O tempo de execução do software processando a região de interesse da sequência S_i para a comparação com o profileHMM j, $t_{reg(i,j)}$;

- O tempo gasto pelo software na comunicação e o controle do hardware de aceleração, $t_{con(i,j)}$;
- O tempo gasto pelo acelerador para produzir o resultado para a sequência S_i, $t_{h(i,j)}$;
- O tempo total do sistema acelerado e G o ganho em desempenho atingido ao incluir o acelerador no sistema, T_{sa};

As Equações 4.1 até 4.4 apresentam a relação entre essas quantidades.

$$N_t \leftarrow S_t P_t \tag{4.1}$$

$$T_{ss} \leftarrow \sum_{i=1}^{S_t} \sum_{j=1}^{P_t} t_{s_{i,j}} \tag{4.2}$$

$$T_{sa} \leftarrow \sum_{i=1}^{S_t} \sum_{j=1}^{P_t} (t_{h_{i,j}} + t_{reg_{i,j}} + t_{con_{i,j}}) \tag{4.3}$$

$$G \leftarrow \frac{T_{ss}}{T_{sa}} \tag{4.4}$$

As quantidades $t_{s(i,j)}$, $t_{h(i,j)}$, $t_{con(i,j)}$ e $t_{reg(i,j)}$ são obtidas de forma experimental, e os métodos utilizados para encontrá-las são descritos nas seções posteriores deste trabalho. Além das medidas temporais de desempenho, também será comparada a quantidade de células por segundo que processam tanto o software quanto o hardware, com o fim de comparar o nosso acelerador com os outros aceleradores presentes na literatura.

4.6.1 - Análise de Desempenho do software HMMER

Para fazer a medida de desempenho do software HMMER, adotamos a metodologia apresentada em [5], na qual um conjunto muito grande de comparações é feita na implementação do HMMER 2.3.2, o tempo para cada comparação é adquirido e uma regressão linear é implementada para relacionar o comprimento do *profileHMM* e o número de elementos da sequência com o tempo de execução. Nos testes feitos na nossa plataforma de referência (Intel Centrino Duo 1.66GHz, 2GB de memória RAM) foi gerada uma regressão linear para o tempo de processamento do software HMMER partindo dos dados obtidos. A expressão obtida gerou um erro médio de mais de 50% nas estimações dos tempos de processamento do HMMER, pelo que geramos uma regressão de segunda

ordem para estimar os tempos de processamento do software HMMER original. Essa expressão da forma $a_2x^2+a_1x+a_0$, consegue se aproximar de uma maneira adequada com os resultados encontrados para a execução do HMMER. Os resultados dos testes feitos na nossa plataforma e a geração da expressão para determinar o desempenho do software HMMER são analisados no Capítulo 5.

4.6.2 - Análise do Desempenho do Hardware

A estimação do desempenho do hardware é composta por duas partes. A primeira consiste na quantidade de células das matrizes de programação que esse é capaz de calcular a cada segundo. A segunda consiste no tempo que ele gasta para realizar o processamento de um conjunto de sequências com determinado HMM. Devido a estas quantidades serem altamente dependentes da implementação em VHDL, a sua derivação é deixada para o Capítulo 5 no qual a estrutura do hardware e seu funcionamento são explicados e uma formula teórica para o desempenho é obtida. Os resultados da fórmula obtida no Capítulo 5 são comparados com os dados obtidos a partir da execução do *testbench* e ela prova ser uma ferramenta exata para a estimação do desempenho do hardware.

5 - IMPLEMENTAÇÃO DA ARQUITETURA E RESULTADOS EXPERIMENTAIS

A arquitetura proposta no Capitulo 4 foi codificada em VHDL e seu funcionamento foi validado no programa de simulação Modelsim para Altera de Mentor Graphics [32]. A síntese feita atinge um máximo de 85 EPs dentro de uma FPGA Stratix II 180, consumindo o 84% do FPGA, o que deixa espaço suficiente para a implementação do controle de comunicação para o Arranjo Sistólico. O funcionamento do controle de comunicação é proposto, mas na implementação feita não foi incluída, devido à falta de um FPGA suficientemente grande para o projeto. Um banco de registradores de transição é inserido já que o EP deve ter acesso a nove probabilidades de transição em todo momento (precisaríamos de uma RAM de nove portas). Esse banco de registradores é carregado no inicio de uma comparação ou de uma iteração em 5 ciclos de relógio (duas probabilidades de transição por ciclo), por um controlador dentro da memória RAM de armazenamento de transições.

Como o arranjo implementado deve suportar profileHMMs longos, um determinado EP em uma posição tem que calcular mais que uma coluna das matrizes de programação dinâmica (Figura 4.5). Na implementação, foi necessária a inclusão de um controle de iteração, tanto no EP quanto nas memórias de armazenamento. Esse controle de iteração reconhece dois elementos de sequência especiais que são inseridos pela unidade de controle geral, um para o incremento da iteração (‘*’) e um para o fim do processamento da sequência (‘@’).

O elemento de fim de iteração faz com que o EP reconheça que o sistema vai entrar na próxima iteração, incremente o registro interno de iteração e inicialize os registradores tanto para o Algoritmo de Viterbi quanto para o Algoritmo das Divergências. Além disso, na memória RAM para as emissões, o elemento de fim de iteração faz que o registro de iteração seja incrementado. Na memória de armazenamento de transições, faz com que o controlador carregue o banco de registradores com as transições para a próxima iteração, além de incrementar o registrador de iteração dela. O elemento do fim de iteração causa o *reset* dos registradores de iteração e o carregamento do banco de registradores de transição com as probabilidades para a primeira iteração, este último a fim de processar uma nova sequência imediatamente após do termino da anterior.

5.1 - IMPLEMENTAÇÃO EM VHDL

5.1.1 - Arranjo Sistólico (AS)

Na Figura 4.6 pode se observar que o Arranjo Sistólico é composto principalmente pelos Elementos de Processamento (EPs), as memórias RAM de bloco para o armazenamento das probabilidades, as unidades de cálculo dos vetores B e C e as memórias FIFO para o armazenamento temporal dos resultados intermediários. Cada um dos componentes do AS contém sub-componentes que possibilitam o funcionamento correto do sistema, e serão analisados de forma individual nas seções seguintes. O código VHDL é um código configurável em tempo de compilação e os parâmetros do AS podem ser modificados a vontade pelo projetista para explorar o espaço de projeto. Os parâmetros da arquitetura que podem ser modificados são: o número de EPs no AS, o tamanho da palavra do AS (16 ou 32bits), o tamanho da memória de armazenamento de probabilidades de emissão, o tamanho da memória de armazenamento de probabilidades de transição e o tamanho das memórias FIFO para resultados intermediários. A Figura 5.1 apresenta o diagrama de blocos da implementação em VHDL, mostrando os sinais de entrada e saída de todos os blocos. No Apêndice C é mostrado o código VHDL para a implementação da arquitetura, incluindo o arquivo de configuração dos parâmetros da arquitetura e o *testbench.*

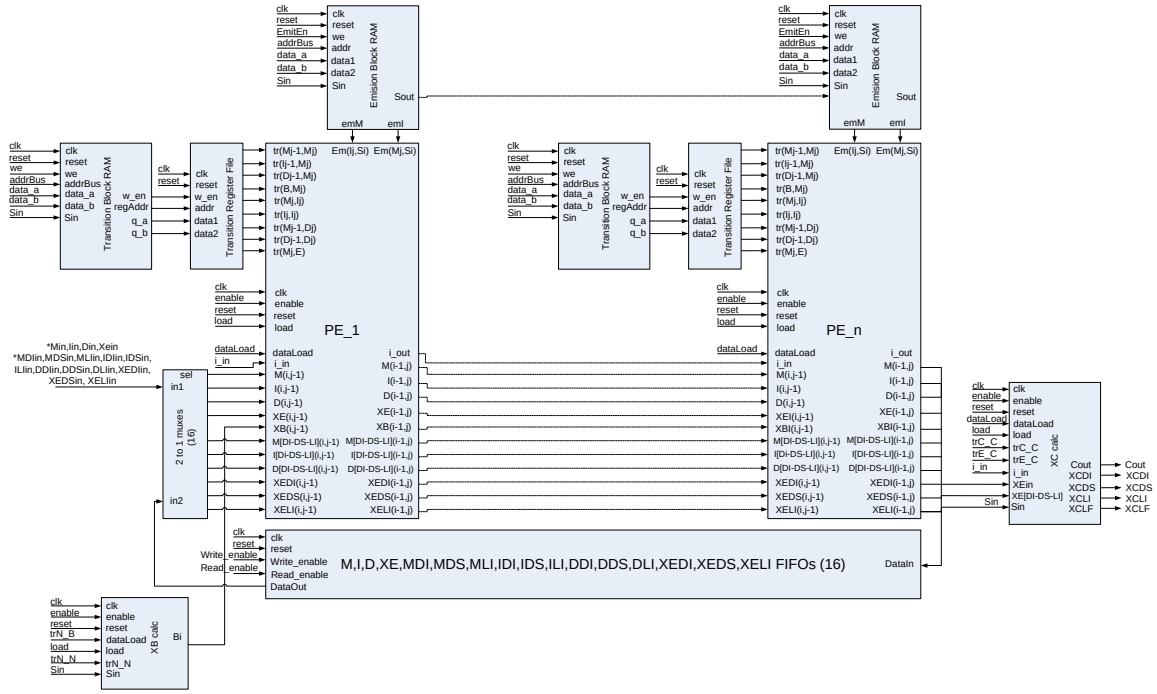


Figura 5.1: Diagrama de blocos completo do Arranjo Sistólico

5.1.1.1 - Elemento de Processamento (EP)

O elemento de processamento é o encarregado de calcular uma das colunas das matrizes de programação dinâmica, tanto para o Algoritmo de Viterbi quanto para o Algoritmo das Divergências. Como pode ser observado na Figura 5.2, o EP recebe como entrada os seguintes sinais:

- *tr(X,X):* São as probabilidades de transição para o Nodo *j* (N_j) do profileHMM (Figura 3.10), eles vêm do banco de registradores de transição, que é carregado pelo controlador dentro da memória de armazenamento de probabilidades de transição.
- *em(Si,X):* São as probabilidades de emissão para o elemento S_i da sequência no estado X, elas vem diretamente da memória de armazenamento.
- *Clk:* sinal de relógio
- Enable: sinal proveniente da unidade de controle geral que habilita o funcionamento do EP.
- *Reset:* sinal que zera os registradores internos, geralmente é usada no começo da execução para garantir que o arranjo sistólico comece a partir de um estado conhecido.
- *Load:* sinal utilizado pelo controle geral para inicializar os registradores internos na comparação da primeira sequência com o profileHMM. Após da primeira comparação, o sinal de load no EP é gerado internamente pelo controlador de iteração.
- *dataLoad:* É o sinal utilizado para inicializar os registradores para o Algoritmo de Viterbi, geralmente possui o valor -∞, o que corresponde à fila zero ou à coluna zero das matrizes de programação dinâmica.
- *I_in:* É o índice do elemento da sequência que está passando por o EP no momento e é requerido para o cálculo das divergências (Figura 4.8).
- *M, I, D, XE, XB:* São as saídas do EP anterior, necessárias para o cálculo das pontuações. Estas entradas representam a célula (i,j-1) das matrizes de programação dinâmica.
- **(DI,DS,LI):* São as saídas para as divergências dos estados M, I, D, E do EP anterior, necessárias para o cálculo das divergências para o nodo do EP atual.

Os sinais de saída do EP geralmente são ligados na entrada do próximo EP no AS, de modo que o elemento da sequência passa por todos os EPs de maneira análoga como o faria no Algoritmo de Viterbi. Os sinais de saída do EP também são apresentados na Figura 5.2 e são:

- *I_out:* é o elemento *i_in* atrasado em um ciclo de relógio. A finalidade da saída é permitir que o elemento S_i da sequência passe por todos os EPs, o que torna possível que a pontuação para o alinhamento que termina em S_i seja calculada.
- M, I, D, XE, XB: São saídas que vão para o próximo EP. Estas saídas correspondem a célula *(i,j-1)* da matriz de programação dinâmica.
- *(DI,DS,LI): São as divergências calculadas para o nodo *j,* e são ligadas com o próximo EP para permitir que ele calcule as divergências para o nodo *j+1*.

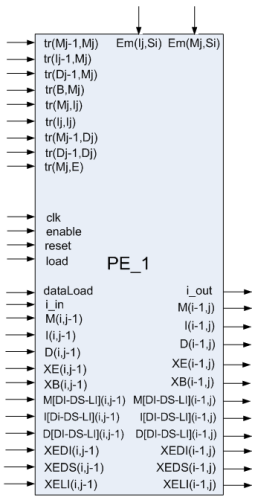


Figura 5.2: Entradas/Saídas do EP.

Nas Figuras 4.7 e 4.8 pode se observar que o EP esta conformado por unidades aritméticas saturadas, operadores de máximos e mínimos modificados para entregar o índice do máximo e registradores internos. Além destes operadores, o EP também possui o controle de iteração, que reconhece os elementos especiais (‘*’ e ‘@’) e modifica o registrador de iteração.

Os operadores aritméticos saturados realizam operações de soma garantindo que o resultado da operação nunca seja maior do que 32767 ou menor do que -32768. A restrição é imposta pelo algoritmo já que a probabilidade de uma transição inexistente é 0 e seu respectivo *logOdds* é -∞. O somador proposto utiliza um somador tradicional e verifica se as condições de overflow da soma foram atingidas, fazendo a escolha do máximo positivo

ou do mínimo negativo segundo seja o caso. A Figura 5.3 mostra o diagrama de blocos geral para o somador proposto.

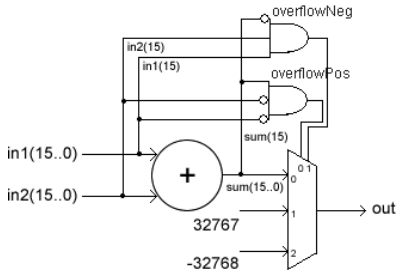


Figura 5.3: Diagrama de blocos para um somador saturado.

Os operadores de máximo e mínimo foram modificados no projeto para entregar o índice do elemento escolhido. O índice entregue pelo operador vai ser utilizado pelo Algoritmo das Divergências como o sinal de controle para o algoritmo. No projeto temos máximos de duas e quatro entradas, e mínimos de duas entradas. Os máximos e mínimos de duas entradas foram implementados utilizando comparadores de 16 bits, entregando como índice o sinal de saída do comparador. O diagrama de blocos para os máximos e mínimos de duas entradas é apresentado na Figura 5.4.

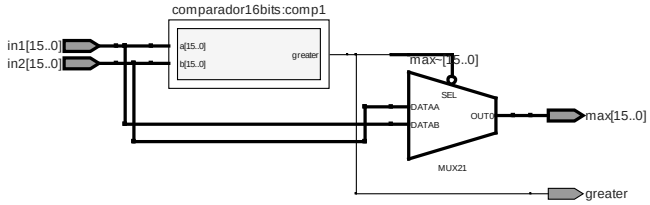


Figura 5.4: Diagrama de blocos para os máximos e mínimos de duas entradas.

Para incrementar a freqüência geral do sistema, o máximo de quatro entradas necessário para a relação de recorrência do estado M foi projetado como 6 comparadores de duas entradas operando em paralelo, seguido de multiplexadores que fazem a escolha entre uma das quatro entradas. A Figura 5.5 mostra o diagrama de blocos do máximo de quatro entradas.

Os registradores internos podem ser divididos em duas classes: a primeira pertence às matrizes de programação dinâmica e a segunda são registradores de controle interno do EP. Os registradores de programação dinâmica são elementos de memória distribuídos que

servem para armazenar temporariamente as células da programação dinâmica e das divergências que pertencem à dependência de dados (Célula*(i-1,j-1)*, Célula*(i-1,j)*). Os registradores de controle armazenam o valor da iteração atual, o índice *(i-1)* do elemento (S_{i-1}) anterior e os valores dos sinais de controle para as divergências. A principal causa da divisão do EP em duas etapas é o armazenamento dos sinais de controle para as divergências, que é feito para cortar o caminho de dados crítico. Pela razão anterior, o cálculo das divergências está atrasado em um ciclo de relógio do cálculo das pontuações, mas isto não altera o funcionamento do algoritmo e, portanto, do sistema.

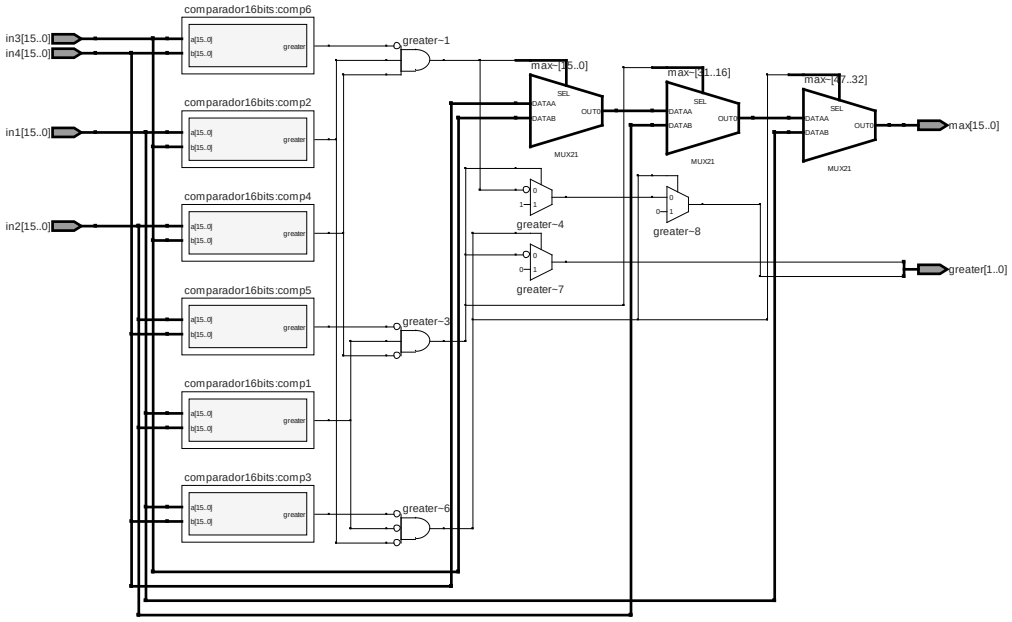


Figura 5.5: Diagrama de blocos para o máximo de quatro entradas

O controle de iteração é um contador que incrementa ou faz o reset do registrador de iterações quando um dos elementos especiais aparece no EP. O controlador de iterações também é responsável pela inicialização dos registradores de programação dinâmica quando ocorre uma mudança de iteração. Para esse propósito ele ativa um sinal de inicialização interno cada que um dos elementos especiais aparece.

5.1.1.2 - Banco de Registradores de Transições

Já que o EP precisa acessar as nove probabilidades de transição do nodo ao mesmo tempo, e que as memórias RAM de bloco presentes fisicamente na FPGA suportam apenas duas

portas, é necessária a implementação de um banco de registradores que possa ser carregado com as probabilidades antes do começo da comparação. O banco de registradores possui duas entradas de dados ligadas na memória de armazenamento de transições, uma entrada para o endereço (que endereça dois registradores cada vez), uma entrada de habilitação de escrita para evitar corromper os dados acidentalmente, um *reset* para zerar os dados, e nove saídas que correspondem às transições (*tr(X,X)*) para o nodo N_j do profileHMM. O diagrama de pinos do banco é apresentado na Figura 5.6.

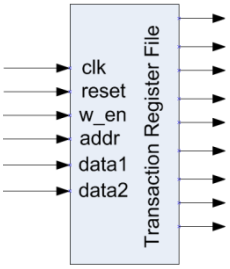


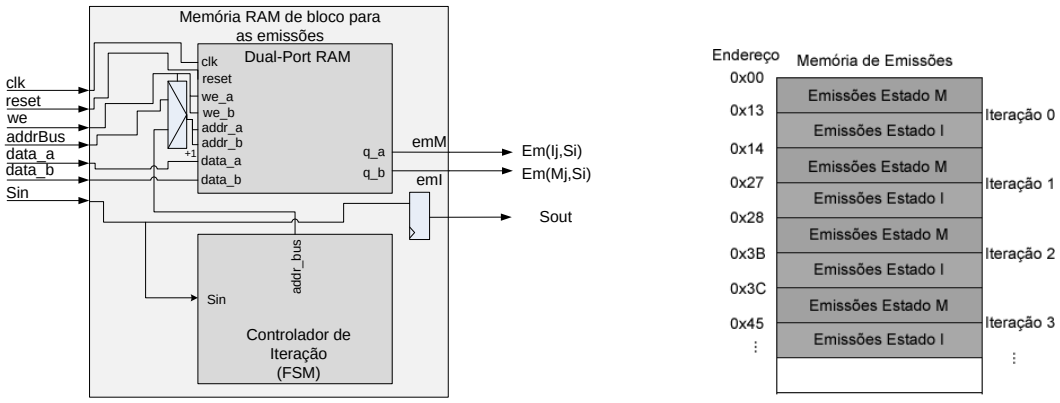
Figura 5.6: Diagrama de pinos do banco de registradores

5.1.1.3 - RAM de probabilidades de Emissão

É uma memória RAM de duas portas modificada para ser endereçada de duas maneiras diferentes quando está sendo programada (escrita) e quando está sendo lida. O modo de programação é ingressado quando o pino *w_en* é ativado. Neste modo, memória toma seu endereço da porta *addr* e armazena os dados das portas *data1* e *data2* nos endereços *addr* e *addr+1*. Este modo é utilizado pela unidade de controle para armazenar as probabilidades de emissão dos profileHMMs antes de começar a operação do sistema. O tempo de carregamento das probabilidades varia dependendo do profileHMM mas duas probabilidades podem ser carregadas em cada borda de subida do relógio. No modo de execução, a memória é endereçada pelo registrador interno de iterações e pelo elemento S_i da sequência para o qual está sendo calculada a pontuação no EP e suas saídas são as probabilidades de emissão para os estados M e I do profileHMM.

A memória de emissões armazena as probabilidades de emissão para o estado M e para o estado I de todas as iterações a ser calculadas no EP ao que a memória está ligada, ou seja que armazena 40 probabilidades por iteração. Ela contém um controlador interno para o número da iteração que é incrementado o zerado quando detecta um dos elementos

especiais (‘*’ ou ‘@’) na entrada S_i. Além disso, a memória de emissões é a encarregada de propagar o elemento da sequência através do arranjo. Para isto, é incluído um registrador interno que é escrito na borda de subida do relógio. A Figura 5.7 (a) apresenta o diagrama de pinos da memória de emissões, enquanto a Figura 5.7 (b) apresenta a estrutura dos dados dentro da memória.

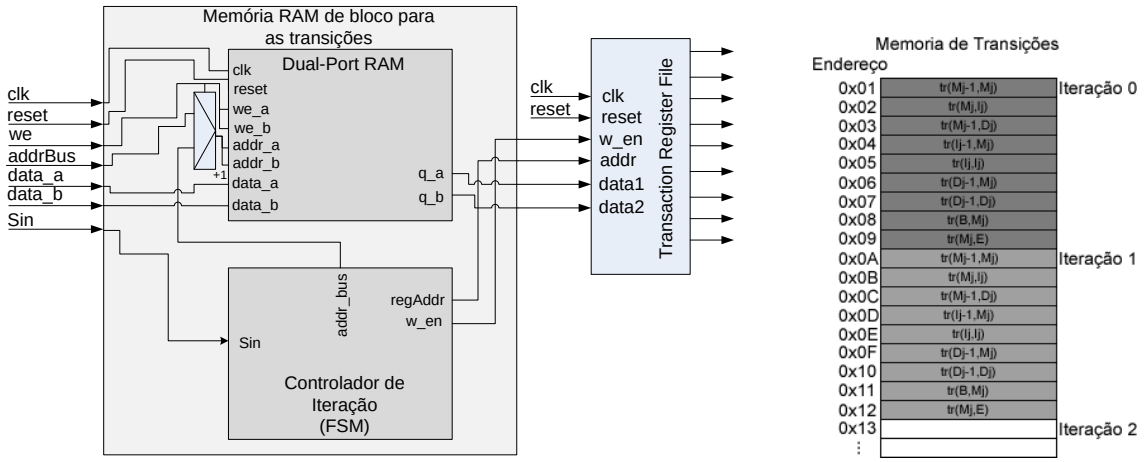


(a) *(b)*

Figura 5.7: a)Diagrama de pinos da memória de emissões. b) Estrutura da memória de emissões.

5.1.1.4 - RAM de probabilidades de Transição

A RAM de Probabilidades de Transição é a memória mais complexa do sistema. Ela também possui dois modos de endereçamento, o de programação e o de execução. No modo de programação ela se comporta como uma memória RAM de duas portas normal: recebe o endereço pela porta *addr_bus* e os dados pelas portas *data_a* e *data_b*. Logo depois que a programação termina (*we* é desativado) ela entra em modo de execução. Neste modo, uma máquina de estados controla a programação do Banco de Registradores de Transição automaticamente em cinco ciclos de relógio, ou seja, gera o endereço tanto para a memória quanto para o Banco além de ativar a linha de escrita dele. O controle interno do modo de execução da memória também detecta quanto apareceu um elemento especial (‘*’ ou ‘@’) e decide se é necessário reprogramar o banco de registradores com probabilidades novas (caso se tenha uma nova iteração ou esteja voltando para a primeira iteração) ou se a programação é desnecessária. Na Figura 5.8 (a) é apresentado o diagrama de pinos para a Memória de Transições e como esta se liga ao Banco de Registradores. A Figura 5.9 (b) apresenta a estrutura certa do armazenamento na Memória de Transições.



(a) *(b)*

Figura 5.8: a)Diagrama da Memória de Transições e conexão com o Branco de Registradores de Transição. b) Estrutura dos dados armazenados dentro da Memória de Transições.

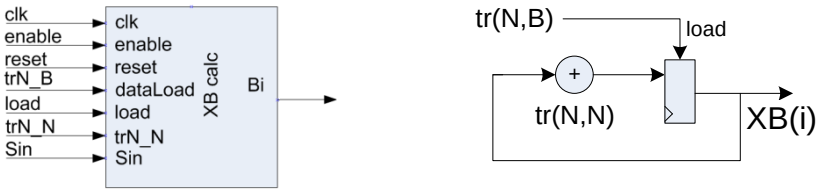
5.1.1.5 - FIFOs de Resultados Intermediários

As memórias FIFO de Resultados Intermediários são as encarregadas de armazenar a saída do último EP, no caso seja necessário executar varias iterações para comparar a sequência com um profileHMM longo e de inserir esses dados no primeiro EP no começo da iteração seguinte. Como característica principal elas podem ser lidas e escritas ao mesmo tempo, o que permite o funcionamento da estratégia de elementos especiais para esconder os tempos de espera entre iterações. A memória é controlada por meio dos pinos de habilitação de leitura (*readEnable*) e de escrita (*writeEnable*) e os dados de entrada/saída são fornecidos/recebidos pelos pinos *dataIn* e *dataOut*. O pino de *reset* não apaga os dados dentro da FIFO, mas reinicia os contadores internos, fazendo o equivalente a uma inicialização. Os pinos de saída *FifoEmpty* e *FifoFull* são utilizados pela unidade de controle para controlar o funcionamento do AS.

5.1.1.6 - Unidade de Cálculo do Vetor B

É a unidade encarregada de calcular o vetor B a partir das probabilidades *tr(N,N)* e *tr(N,B)*. A unidade possui um registrador interno que armazena o valor de B_i. Este registrador é carregado por meio da linha de controle *load* com o valor da porta *DataLoad* (*tr(N,B)*) no

momento de iniciar execução ou no momento em que um dos elementos especiais definidos aparece para marcar o final de uma iteração ou o começo de uma nova sequência. Para perceber quando um dos elementos especiais aparece, foi necessário colocar como entrada da unidade o elemento S_i da sequência. A saída B_i corresponde ao valor do vetor B para o elemento *i* dela sequência *query*. A Figura 5.9 (a) mostra o diagrama de pinos da unidade e a Figura 5.9 (b) apresenta o diagrama de blocos dela.



(a) (b)

Figura 5.9: a) Diagrama de pinos da unidade de cálculo do vetor B. b)Diagrama de blocos da unidade.

5.1.1.7 - Unidade de Cálculo do Vetor C

É a unidade encarregada de calcular o vetor C do Algoritmo de Viterbi e as divergências para o estado C. As suas entradas de controle são: *reset* para levar o valor da unidade ao estado inicial, *load* para carregar os registradores com o valor inicial *dataLoad* (sempre é -∞) e *enable* para habilitar a sua execução. A unidade possui também um controle interno que percebe quando um dos elementos iniciais passa por ela, reiniciando os registradores internos para o valor inicial. As entradas de dados da unidade correspondem às saídas para o estado E do ultimo EP, com elas calcula tanto a pontuação quanto as divergências de saída. O diagrama de pinos da unidade implementada em VHDL é apresentado na Figura 5.10.

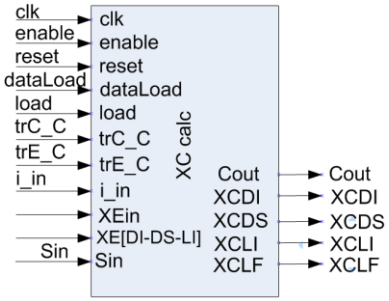


Figura 5.10: Diagrama de pinos para a implementação da unidade de cálculo para o estado C.

5.1.1.8 - Multiplexadores de Entrada

Os multiplexadores de entrada decidem se os dados que entram no primeiro EP são aqueles fornecidos pela Unidade de Controle (UC) ou aqueles lidos das memórias FIFO. O multiplexador é necessário por que na primeira iteração da sequência no arranjo, os dados que ingressam no primeiro EP são dados de inicialização ($-\infty$), e são fornecidos pela unidade de controle, enquanto que, em regime, os dados são lidos das FIFOs. A unidade de controle somente fornece dados quando se trata da primeira iteração da comparação de uma sequência, razão pela qual a linha de seleção dos multiplexadores é controlada por ela e é ativada quando a unidade muda de iteração.

5.1.1.9 - Registradores para probabilidades Especiais e Número de Iterações

As probabilidades para os estados especiais (*tr(N,N), tr(N,B), tr(C,C) e tr(E,C)*) são armazenadas em registradores especiais fora do Arranjo Sistólico (AS), os quais são ligados com os sinais correspondentes dos módulos para o cálculo dos vetores B e C. Eles são modificados pela unidade de controle na hora de programar um novo profileHMM no AS. O número de iterações é enviado também no momento de programar um novo profileHMM já que este é utilizado pelas Máquinas de Estados Finitos (FSM) de Programação das memórias de armazenamento de probabilidades para gerar o endereço de destino dos dados que são enviados pelo processador.

5.1.2. Unidade de Controle e Comunicação (UC)

A Unidade de Controle e Comunicação é o elemento do sistema encarregado de se comunicar com o processador de propósito geral ao fazer a interface entre o AS e o barramento do sistema (AMBA, PCIx, Avalon, etc.). Essa unidade é composta por um bloco de hardware que recebe os comandos e os dados enviados pelo processador e sinaliza o AS para começar execução. Após a execução, obtém os dados e envia-os para o software realizar a leitura e o posterior processamento. A Figura 5.11 apresenta o diagrama do sistema completo, incluindo o AS, a UC e o Processador de Propósito Geral.

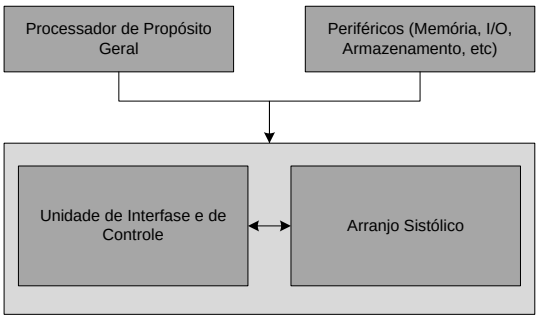


Figura 5.11: Diagrama completo do sistema.

A unidade de controle não foi implementada em VHDL devido à falta de definição de um sistema host já que, dependendo do barramento do sistema, a comunicação da unidade de controle com o processador varia. Ela então deve descrever um conjunto de processos que serão executados de maneira seqüencial para o correto funcionamento do AS, e o correto cálculo da pontuação para a sequência *query*.

Inicialmente, a UC está em repouso, e ela espera pela sinalização do processador para programar um novo profileHMM dentro das memórias de armazenamento ou para o início de uma nova comparação para uma sequência *query* com o profileHMM já programado. É sugerido então que a implementação da unidade de controle inclua registradores para controlar as FSM que coordenam a programação e o funcionamento do AS e registradores para comprovar o status do sistema num momento determinado. Logo depois de receber o sinal de começo, a UC pode executar dois processos. O primeiro consiste na programação de um novo profileHMM e o segundo na comparação de uma nova sequência *query* com um profileHMM pré-programado. A FSM principal da unidade de controle é apresentada na Figura 5.12.

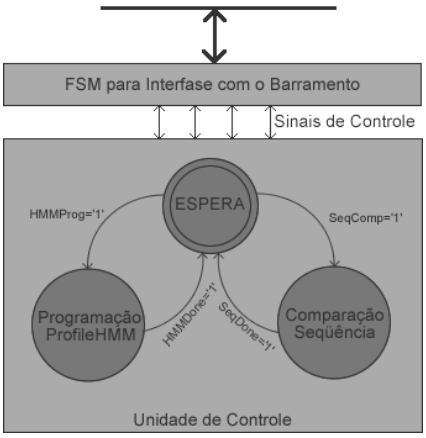


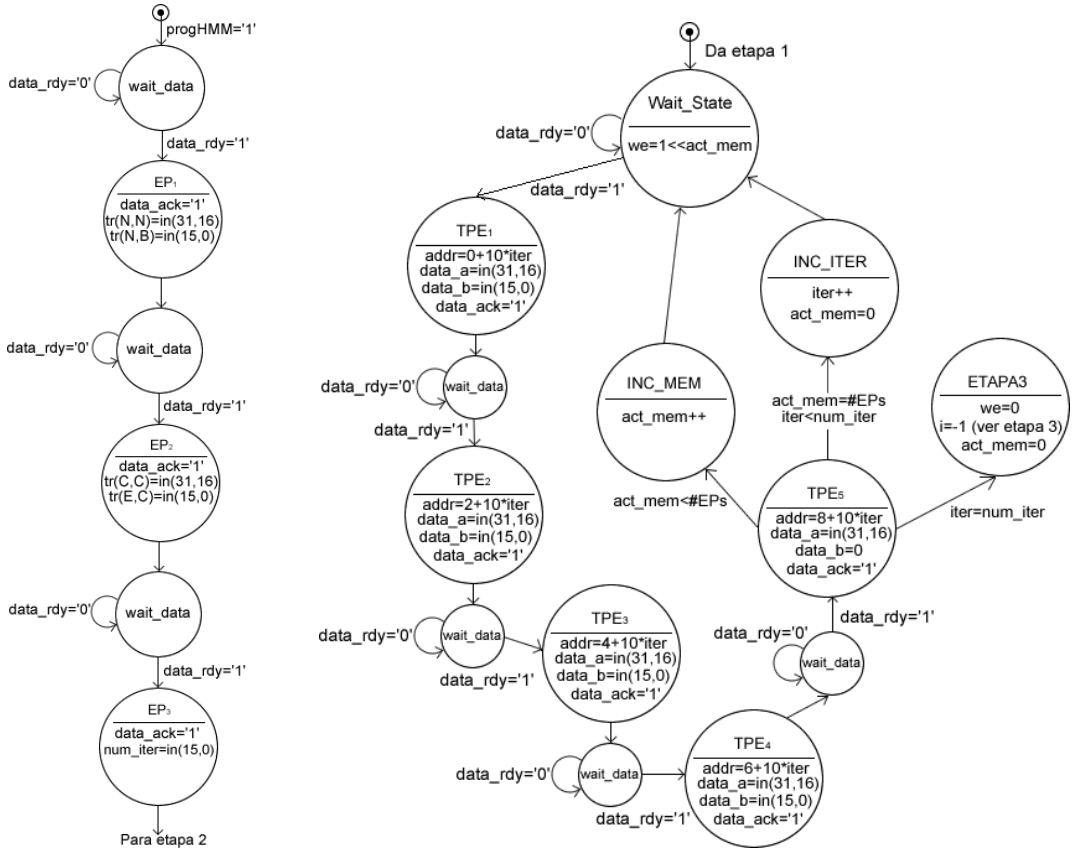
Figura 5.12: FSM principal da unidade de controle.

A programação de um profileHMM possui três etapas. Na primeira, os registradores para as probabilidades de transição dos estados especiais e o número de iterações são escritos, na segunda, são programadas as memórias de transições com as probabilidades para todos os EPs em todas as iterações (um EP pode representar mais de um nodo do profileHMM para HMMs longos, Figura 4.5). Finalmente, na terceira etapa, as probabilidades de emissão para todos os EPs são programadas. Logo depois dessas etapas, a FSM principal da unidade de controle deve voltar para esperar um sinal de start. A execução de uma comparação consiste no recebimento serial dos elementos da sequência *query* (incluindo os elementos especiais) que vêm da interface de comunicação do sistema (o barramento). Para cada elemento a UC calcula o índice i do elemento S_i (que também é o valor de entrada para as divergências do primeiro EP na primeira iteração) e a iteração na qual o sistema está. Baseado nos dados anteriores e no número total de EPs do sistema, modifica os sinais de controle para as memórias FIFO de resultados intermediários e espera o tempo de reprogramação do Banco de Registradores de Transição entre iterações.

5.1.2.1 - Etapa de Programação de profileHMMs

A etapa começa depois que a UC recebe o sinal de começo da Interface de Comunicação com o Barramento (ICB). A FSM de programação vai para o estado de recebimento de probabilidades especiais e iterações, fica esperando que apareçam os dados enviados pelo processador na ICB, grava-os num registrador de probabilidades especiais ou de iterações e envia a confirmação de leitura para a ICB. Como existem quatro registradores de 16bits para guardar as probabilidades especiais, a etapa tem que repetir o procedimento 5/#bits_do_barramento vezes. Logo depois de receber os dados, a FSM de programação

passa para a segunda etapa, onde as probabilidades de transição são armazenadas. A Figura 5.13 (a) apresenta o fluxo da primeira etapa para um barramento de 32bits.



(a) *(b)*

Figura 5.13: a) Fluxo de execução da primeira etapa de programação. b)Segunda etapa da FSM de programação.

A segunda etapa da FSM de programação recebe os dados de transição que devem ser armazenados, habilita as linhas de escrita das memórias de transição (*we* é um vetor com #EPs posições), calcula a iteração para a qual as probabilidades estão sendo armazenadas, calcula o endereço de memória destino dos dados levando em conta a iteração atual e decide se todos os dados foram armazenados, para passar para a etapa 3. O diagrama de estados dessa etapa pode ser visto na Figura 5.13. A etapa três da FSM de programação é similar à etapa dois. Ela recebe os dados para as memórias de emissões, habilita e modifica as linhas e as variáveis de controle necessárias para o armazenamento dos dados em todas as memórias do sistema. Depois de fazer o armazenamento das probabilidades de emissão para todos os EPs em todas as iterações, a FSM volta para o estado de espera da FSM principal. A Figura 5.14 apresenta o fluxo dos estados para a terceira etapa.

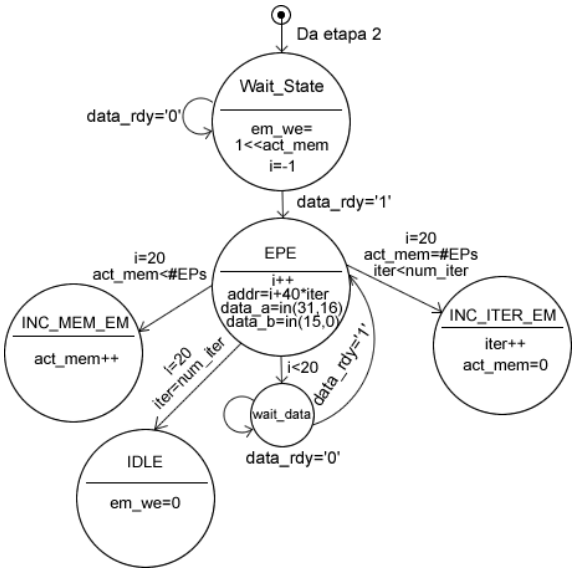


Figura 5.14: Diagrama de estados para a terceira etapa da FSM de programação do sistema.

5.1.2.2 - Etapa de Comparação de Sequências

Após receber o sinal de começo de comparações, a FSM principal da UC vai para o estado de comparação de sequências. Neste estado, a UC é a encarregada de inserir os elementos da sequência *query* na entrada de dados do AS para todas iterações. Além disso, neste estado a UC gera os sinais de controle para as FIFO de Armazenamento de Resultados Intermediários e para os Multiplexadores de Entrada. Gera também o índice *i* para o elemento S_i atual da sequência (que é a entrada para os dados das divergências para o primeiro EP na primeira iteração), o sinal de *enable* para o AS e os tempos de espera para a reprogramação ao mudar de iteração. É importante perceber que a máquina de estados somente recebe os elementos da sequência. A mudança de iterações interna de cada EP é feita automaticamente quando cada EP detecta um dos elementos especiais (‘*’, ‘@’) definidos na implementação do trabalho. A unidade de controle utiliza os elementos especiais para re-inicializar os contadores e os sinais de controle internos dela somente. Uma representação gráfica da máquina de estados que deve ser implementada na UC é mostrada na Figura 5.15.

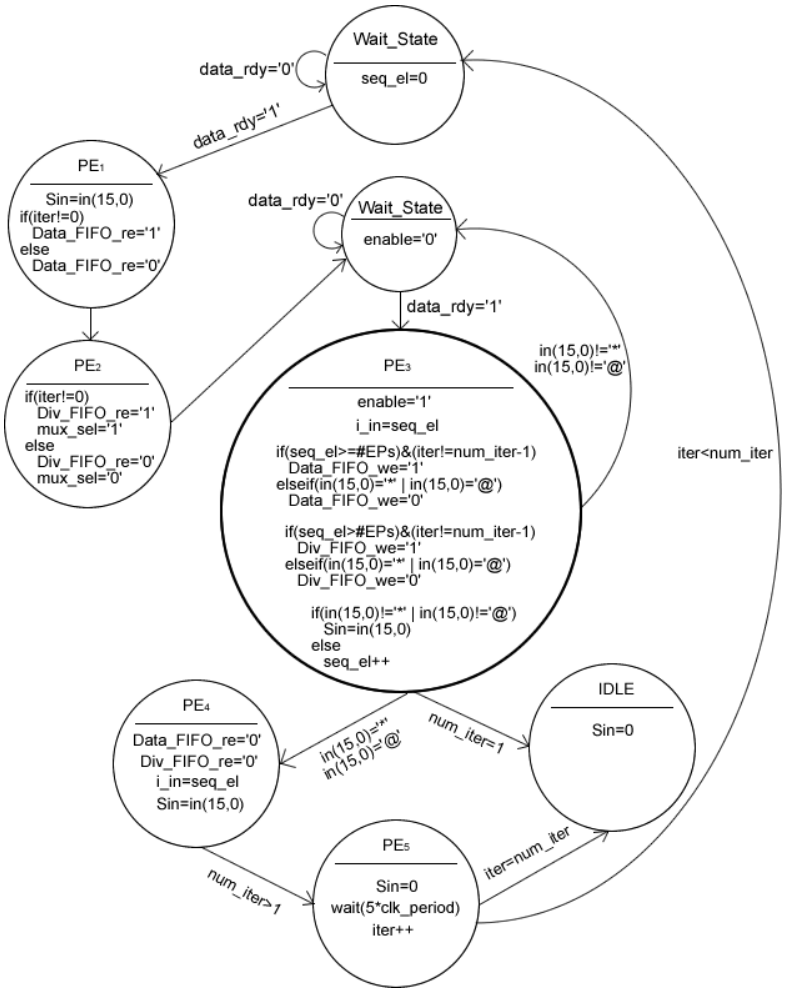


Figura 5.15: Fluxo da FSM para comparações de sequência.

Na Figura 5.15 podem ser observados alguns detalhes de implementação que surgiram principalmente devido à natureza síncrona dos elementos que constituem o AS. Vemos como o primeiro elemento da sequência e a linha de habilitação de leitura das FIFOs para os estados M, I, D, E devem ser escritos um estado antes (ciclo de relógio) de começar o processamento nos EPs para fornecer tempo suficiente para as memórias serem lidas. Pode se olhar também como o cálculo das divergências está atrasado em um ciclo de relógio e a linha de leitura das suas FIFO de Resultados Intermediários é habilitada um ciclo depois que a das matrizes de programação dinâmica. Além disso, pode se observar o tempo que a máquina deve esperar para que o controlador interno das memórias de transição faça a reprogramação automática das probabilidades de transição no Banco de Registradores, caso se tenha um profileHMM longo, e que no caso de ter somente uma iteração, não é necessária a reprogramação do Banco.

5.2 – TESTBENCH

O *testbench* foi desenvolvido em VHDL e simulado no software Modelsim [32], ele implementa as máquinas de estados propostas na seção anterior e simula a interação com um processador de propósito geral ao ler os dados de arquivos de texto e ao escrever os dados em arquivos de texto de saída. Isso foi feito para conferir o resultado do funcionamento da implementação feita para o Arranjo Sistólico. Todas as etapas do processo de execução do sistema foram simuladas: a programação do profileHMM, a comparação de um arquivo com múltiplas sequências com o profileHMM e a obtenção dos resultados obtidos para a comparação. O *testbench* lê três arquivos de entrada: um para as probabilidades de transição, um para as probabilidades de emissão e um para o conjunto de sequências *query* e escreve dois arquivos de saída: um para as pontuações e as divergências geradas durante as comparações e um para os ciclos de relógio consumidos no processamento de cada sequência *query*. O diagrama do sistema simulado pode ser visto na Figura 5.16.

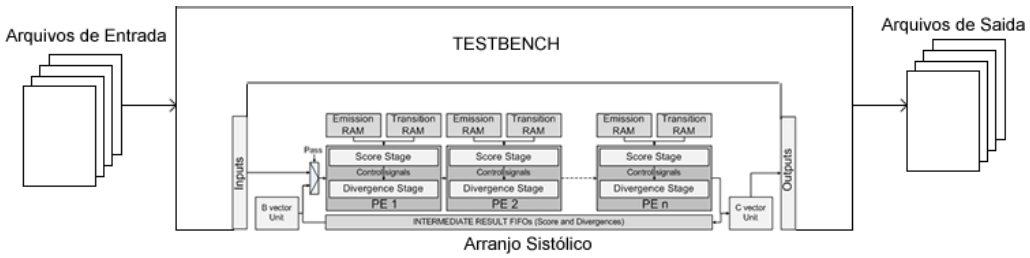


Figura 5.16: Diagrama de blocos do AS com o *testbench*.

No primeiro arquivo de saída (*out_general*), o *testbench* informa o número de ciclos de relógio que leva para carregar o profileHMM, o número de EPs implementados no hardware, o número de iterações necessárias para comparar uma sequência com o profileHMM programado no hardware e o tamanho da sequência *query*. O segundo arquivo de saída (*out_scores*) informa o número da sequência processada dentro do arquivo, seu tamanho, o número de ciclos que o sistema leva desde o início da simulação, a pontuação para o Algoritmo de Viterbi calculada pelo hardware e o resultado do hardware para o Algoritmo das Divergências. As saídas do primeiro e o segundo arquivo são apresentadas nas Figura 5.17 (a) e (b).

```
Beginning test bench operation
Cycle count:                    0
Finished Special probs loading
Cycle count:                    4
Finished transition probs loading
Cycle count:                    254
Number of PEs:                  25
Number of passes:               2
Finished emission probs loading
Cycle count:                    1254
>sp|Q4U9M9|104K_THEAN 104 kDa microneme/rhoptry antigen OS=Theileria annulata GN=T
Sequence Size:                  894
>sp|P15711|104K_THEPA 104 kDa microneme/rhoptry antigen OS=Theileria parva GN=TP04
Sequence Size:                  925
>sp|Q43495|108_SOLLC Protein 108 OS=Solanum lycopersicum PE=2 SV=1
Sequence Size:                  103
>sp|P18646|10KD_VIGUN 10 kDa protein OS=Vigna unguiculata PE=3 SV=1
Sequence Size:                  76
>sp|P18560|1101L_ASFB7 Protein MGF 110-1L OS=African swine fever virus (strain Bad
Sequence Size:                  271
>sp|P18559|1102L_ASFB7 Protein MGF 110-2L OS=African swine fever virus (strain Bad
Sequence Size:                  105
```

(a)

Seq	Length	XCout	LinIni/LinFin		DivInf/DivSup		cycles
1	893	0	0	892	0	0	3074
2	924	0	0	923	0	0	4936
3	102	0	0	101	0	0	5154
4	75	0	0	74	0	0	5318
5	270	0	0	269	0	0	5872
6	104	0	0	103	0	0	6094
7	124	0	0	123	0	0	6356
8	82	0	0	81	0	0	6534

(b)

Figura 5.17: Saídas geradas pelo *testbench* para o primeiro (a) e o segundo arquivo (b).

O código fonte é incluído no Apêndice C como uma indicação de como deve ser implementada a Unidade de Comunicação e de Controle, mas não é possível fazer a síntese do código já que é VHDL puramente funcional.

5.3 - RESULTADOS EXPERIMENTAIS

Nesta seção, os resultados experimentais obtidos são apresentados. Eles incluem a estimativa de desempenho do HMMER rodando em uma plataforma de testes, a fórmula do desempenho obtido para o hardware através da simulação feita em VHDL da implementação da arquitetura proposta e os resultados obtidos para o software modificado, o qual somente faz o processamento da região de interesse das matrizes de programação dinâmica. Além disso, comparações são feitas entre o desempenho dos dois sistemas (software puro x software/hardware).

5.3.1 - Análise do Desempenho do Software HMMER

A estimativa de desempenho foi obtida utilizando a metodologia apresentada no capítulo 4. As comparações foram feitas amostrando randomicamente 2000 sequências da base de dados uniprot.sprot [3] e realizando a busca nos 10340 profileHMMs na base de dados PFAM-A. Além disso, o software *hmmsearch* foi modificado para tomar o tempo que a rotina para o Algoritmo de Viterbi leva para ser executada, obtendo dados independentes para ela e para o Algoritmo de *traceback*. A amostragem automática das 2000 sequências foi feita com um programa codificado em Java. Para a comparação e coleta dos dados das comparações um script de Linux foi gerado e para a geração da regressão que gera a expressão de segunda ordem, Matlab foi utilizado. O código fonte desses programas é apresentado no Apêndice B. A Figura 5.18 apresenta os gráficos gerados pelo Matlab para

os dados obtidos. A linha azul representa os dados obtidos para o tempo de execução da rotina para o Algoritmo de Viterbi, a linha verde representa o tempo total de execução do programa *hmmsearch*, a linha vermelha é os resultados gerados pela avaliação da expressão gerada e a linha preta é o tempo gastado pelo Algoritmo de *traceback*.

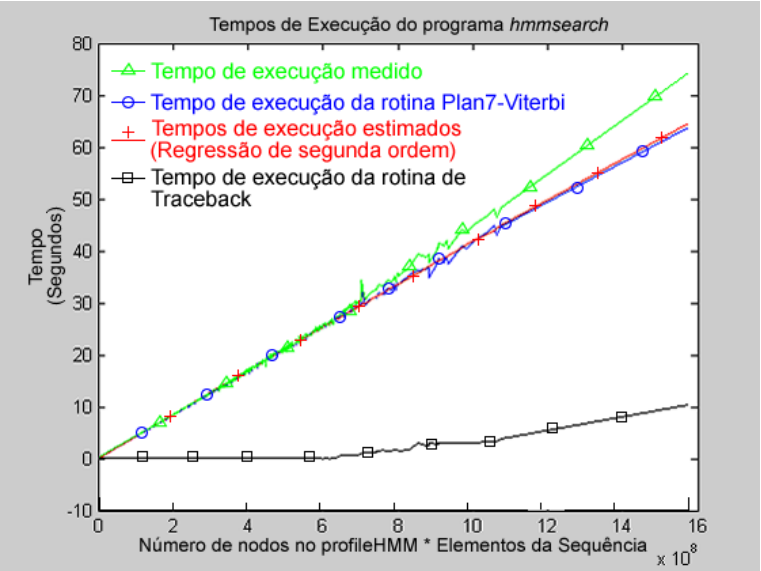


Figura 5.18: Tempos de Execução para o software hmmsearch.

Na figura, pode-se observar que o tempo de execução do Algoritmo de *traceback* vai ficando mais significativo quanto maior é o comprimento do profileHMM. A expressão calculada para o tempo de execução do software HMMER ao fazer uma comparação de uma sequência *i* de comprimento *l* com um profileHMM *j* com *m* nodos para a plataforma de referencia é mostrada na Equação 5.1.

$$t_{s_{i,j}} \leftarrow -1{,}3684 * 10^{-18}(m_j l_i)^2 + 4{,}3208 * 10^{-8}(m_j l_i) - 0{,}1160 \tag{5.1}$$

A Tabela 5.1 apresenta dados obtidos tanto do software quanto utilizando a expressão obtida no MATLAB, para comparações de quatro conjuntos de sequências diferentes com 6 profileHMMs. Da tabela, pode-se observar que a aproximação é mais acertada quando se avalia para profileHMMs longos. Baseado nos tempos obtidos na Tabela 5.1, podemos estimar a taxa em CUPS que a implementação em software do Algoritmo de Viterbi atinge ao ser rodada pelo software HMMER, mostrada na Equação 5.2, onde $t_{s(i,j)}$ é o tempo gasto pelo software para comparar uma sequência *query j* com o profileHMM *i*, m_i é o número

de nodos do profileHMM i, l_j é o comprimento (número de aminoácidos) do conjunto de sequências de teste j, N_t é o número total de conjuntos de sequências de teste e H_t é o número total de profileHMMs para os que foram feitas as comparações. Da avaliação da Equação 5.2, achamos que o desempenho médio para o Algoritmo de Viterbi implementado no software HMMER é de 23,157 MCUPS.

$$T_{CUPS} = \frac{\sum_{i=1}^{H_t} \sum_{j=1}^{N_t} \frac{m_i l_j}{t_{s_{i,j}}}}{H_t N_t} \tag{5.2}$$

Tabela 5.1: Estimação do tempo de execução do hmmsearch x tempo real.

Numero total de aminoácidos do Conjunto de Sequências	*Nodos do ProfileHMM*	*Tempo Real (Total)*	*Tempo Real (Viterbi)*	*Tempo Estimado (Viterbi)*	*% Erro*
687406	*788*	*23,40*	*23,28*	*22,871*	*1,75*
	10	*0,40*	*0,35*	*0,1810*	*48,2*
	226	*6,55*	*6,47*	*6,5635*	*1,42*
	337	*9,85*	*9,8*	*9,8199*	*0,2*
	2295	*74,49*	*64,09*	*64,6425*	*0,8*
	901	*26,32*	*26,25*	*26,1199*	*0,4*
697407	*788*	*24,34*	*23,48*	*23,2158*	*1,12*
	10	*0,47*	*0,41*	*0,1853*	*54,1*
	226	*6,68*	*6,66*	*6,6602*	*0,003*
	337	*9,88*	*9,83*	*9,9634*	*1,35*
	2295	*78,87*	*62,89*	*65,5334*	*4,2*
	901	*27,55*	*25,96*	*26,4938*	*2,05*
700218	*788*	*24,40*	*23,37*	*23,3082*	*0,2644*
	10	*0,42*	*0,38*	*0,1865*	*50,9211*
	226	*6,76*	*6,72*	*6,6873*	*0,4866*
	337	*10,26*	*9,84*	*10,0037*	*1,6636*
	2295	*81,23*	*62,25*	*65,7849*	*5,6786*
	901	*27,41*	*26,33*	*26,5989*	*1,0213*
712734	*788*	*25,42*	*24,03*	*23,7193*	*1,2930*
	10	*0,42*	*0,37*	*0,1919*	*48,1351*
	226	*6,82*	*6,77*	*6,8083*	*0,5657*
	337	*10,09*	*10,01*	*10,1832*	*1,7303*
	2295	*81,07*	*63,81*	*66,8985*	*4,8402*
	901	*27,46*	*26,82*	*27,0665*	*0,9191*

5.3.2 - Derivação do Desempenho do Hardware Proposto

Para a obtenção do desempenho atingido pela unidade de aceleração de hardware foi analisado o desempenho observado durante a simulação do sistema feita por meio do *testbench* e foi extraída uma formula geral para o desempenho do AS. Ela leva em conta todos os possíveis *delays* inseridos pela programação ou leitura dos elementos sincrônicos do sistema (memórias, banco de registradores, FIFOs). O processo de transmissão de dados desde o processador de propósito geral até o AS não é incluído no cálculo, já que o sistema será inserido no futuro em um barramento PCI Express [33] ou HyperTransport [34], que fornecem taxas de transmissão muito superiores aos 130MBps requeridos pelo acelerador, pelo que $t_{con(i,j)}$ na Equação 4.3 é considerado como 0.

Seja m_i o número de nodos do profileHMM *i* programado no sistema, S_j o tamanho da sequência *query j* sendo processada, *n* o número de EPs implementados no sistema, *f* a máxima freqüência atingida pelo sistema depois da síntese física na FPGA, T_{hw} o desempenho atingido pelo sistema em CUPS, t_{htotal} o tempo de processamento do sistema e $t_{h(i,j)}$ o tempo de processamento para uma comparação da sequência *j* com o profileHMM *i*. O desempenho em CUPS T_{hw}, o tempo t_{htotal} de processamento do sistema e o tempo gasto no processamento de uma sequência $t_{h(i,j)}$ estão completamente caracterizados pelas Equações 5.3, 5.4 e 5.5.

Em 5.3, o termo $25n\lceil m_i / n \rceil$ é o número de ciclos gasto carregando as probabilidades de emissão e transição de um profileHMM nas memórias RAM do Arranjo, n é o tempo de enchimento do AS, $(S_j + 6)\lceil m_i / n \rceil$ são os ciclos de relógio gastos pela comparação da sequência (sendo *6* o tempo de reprogramação das probabilidades de transição dentro do banco de registradores em uma mudança de iteração e S_j o número de aminoácidos da sequência *j*) e $S_j * m_i$ é o número de células calculadas na comparação da sequência S_j com o profileHMM m_i. A constante *-2* aparece já que na primeira iteração depois da carga das probabilidades não é necessário realizar carregamento do banco de registradores (depois de uma escrita na memória o controle carrega o banco automaticamente), e seria a diferença entre o tempo necessário (4 ciclos de relógio) para carregar as probabilidades especiais do modelo (*tr(N,N),tr(N,B),tr(E,C),tr(C,C)*) e o tempo necessário para programar o banco de registradores de transição.

$$T_{hw} = \frac{\sum_{i=1}^{\#HMMs}\sum_{j=1}^{\#Seqs} S_j m_i}{\sum_{i=1}^{\#HMMs}\left[\left(\sum_{j=1}^{\#Seqs}(S_j+6)\left\lceil\frac{m_i}{n}\right\rceil\right)+25n\left\lceil\frac{m_i}{n}\right\rceil+n-2\right]} * f \quad (5.3)$$

$$t_{h_{total}} = \frac{\sum_{i=1}^{\#HMMs}\left[\left(\sum_{j=1}^{\#Seqs}(S_j+6)\left\lceil\frac{m_i}{n}\right\rceil\right)+25n\left\lceil\frac{m_i}{n}\right\rceil+n-2\right]}{f} \quad (5.4)$$

$$t_{h_{(i,j)}} = \frac{(S_j+6)\left\lceil\frac{m_i}{n}\right\rceil+25n\left\lceil\frac{m_i}{n}\right\rceil+n-2}{f} \quad (5.5)$$

A síntese no FPGA foi feita para AS com 25, 50, 75 e 85 EPs (Seção 5.3.4). Nas Tabelas 5.2, 5.3, 5.4 e 5.5 podem ser vistos os resultados obtidos para o desempenho das diferentes sínteses feitas utilizando o mesmo conjunto de sequências e profileHMMs de teste aos apresentados na Tabela 5.1. Os dados apresentados são exclusivamente os dados obtidos para o Algoritmo de Viterbi.

Tabela 5.2: Resultados de desempenho para uma implementação com 25 EPs.

#EPS	*Numero total de aminoácidos do Conjunto de Sequências*	*Nodos do ProfileHMM*	T_{hw} *(GCUPS)*	t_h *(segundos)*
25	*687406*	*788*	1,7468	0,3101
		10	0,7093	0,0097
		226	1,6031	0,0969
		337	1,7075	0,1357
		2295	1,7695	0,8915
		901	1,7274	0,3586
	697407	*788*	1,7468	0,3146
		10	0,7093	0,0098
		226	1,6031	0,0983
		337	1,7075	0,1376
		2295	1,7695	0,9045
		901	1,7274	0,3638
	700218	*788*	1,7468	0,3159
		10	0,7093	0,0099
		226	1,6031	0,0987
		337	1,7075	0,1382
		2295	1,7695	0,9081
		901	1,7274	0,3652
	712734	*788*	1,7468	0,3215
		10	0,7093	0,0100
		226	1,6031	0,1005
		337	1,7075	0,1407
		2295	1,7695	0,9244
		901	1,7274	0,3718

Tabela 5.3: Resultados de desempenho para uma implementação com 50 EPs.

#EPS	*Numero total de aminoácidos do Conjunto de Sequências*	*Nodos do ProfileHMM*	T_{hw} *(GCUPS)*	t_h *(segundos)*
50	*687406*	*788*	3,4903	0,1552
		10	0,7086	0,0097
		226	3,2033	0,0485
		337	3,4118	0,0679
		2295	3,5358	0,4462
		901	3,3607	0,1843
	697407	*788*	3,4904	0,1574
		10	0,7086	0,0098
		226	3,2033	0,0492
		337	3,4119	0,0689
		2295	3,5359	0,4527
		901	3,3608	0,1870
	700218	*788*	3,4905	0,1581
		10	0,7086	0,0099
		226	3,2034	0,0494
		337	3,4120	0,0692
		2295	3,5359	0,4545
		901	3,3608	0,1877
	712734	*788*	3,4906	0,1609
		10	0,7086	0,0101
		226	3,2035	0,0503
		337	3,4121	0,0704
		2295	3,5360	0,4626
		901	3,3609	0,1911

Tabela 5.4: Resultados de desempenho para uma implementação com 75 EPs.

#EPS	*Numero total de aminoácidos do Conjunto de Sequências*	*Nodos do ProfileHMM*	T_{hw} *(GCUPS)*	t_h *(segundos)*
75	*687406*	*788*	4,9293	0,1068
		10	0,6880	0,0097
		226	3,8876	0,0388
		337	4,6377	0,0485
		2295	5,0942	0,3010
		901	4,7691	0,1262
	697407	*788*	4,9295	0,1083
		10	0,6880	0,0099
		226	3,8878	0,0394
		337	4,6379	0,0492
		2295	5,0944	0,3053
		901	4,7693	0,1280
	700218	*788*	4,9296	0,1088
		10	0,6880	0,0099
		226	3,8878	0,0396
		337	4,6379	0,0494
		2295	5,0945	0,3066
		901	4,7693	0,1286
	712734	*788*	4,9298	0,1107
		10	0,6880	0,0101
		226	3,8880	0,0403
		337	4,6382	0,0503
		2295	5,0947	0,3120
		901	4,7696	0,1308

Tabela 5.5: Resultados de desempenho para uma implementação com 85 EPs.

#EPS	*Numero total de aminoácidos do Conjunto de Sequências*	*Nodos do ProfileHMM*	T_{hw} *(GCUPS)*	t_h *(segundos)*
85	*687406*	*788*	5,4203	0,0971
		10	0,6877	0,0097
		226	5,1815	0,0291
		337	5,7949	0,0388
		2295	5,8468	0,2622
		901	5,6341	0,1068
	697407	*788*	5,4205	0,0985
		10	0,6877	0,0099
		226	5,1817	0,0296
		337	5,7952	0,0394
		2295	5,8471	0,2660
		901	5,6344	0,1084
	700218	*788*	5,4206	0,0989
		10	0,6877	0,0099
		226	5,1818	0,0297
		337	5,7953	0,0396
		2295	5,8472	0,2671
		901	5,6345	0,1088
	712734	*788*	5,4209	0,1007
		10	0,6878	0,0101
		226	5,1821	0,0302
		337	5,7956	0,0403
		2295	5,8475	0,2719
		901	5,6348	0,1108

Das tabelas pode-se observar que o desempenho depende diretamente do número de EPs implementados dentro do sistema e do número de nodos do *profileHMM*. Entre maior seja o número de EPs implementados dentro do arranjo sistólico, maior será o paralelismo atingido pelo mesmo. Pode-se observar também que o desempenho depende diretamente

do número de nodos do *profileHMM*, e que se este é um múltiplo do número de EPs implementado dentro do sistema, o desempenho é ótimo. Isso acontece já que quando o número de nodos do *profileHMM* não é um múltiplo exato do número de EPs, temos EPs osciosos programados com probabilidades 0 ou -∞, que desperdiçam ciclos de relógio e afetam o desempenho do sistema. As Figuras 5.19 até 5.24 apresentam as estatísticas de desempenho em CUPS e tempo de processamento obtidas para os sistemas quando o número de EPs varia para dois dos profileHMMs testados (um longo e um curto). Também apresentam a dependência do desempenho em relação ao comprimento da sequência e a dependência do desempenho em relação ao número de nodos do profileHMM programado. Das figuras, pode-se observar que as variáveis que mais influenciam o desempenho são o comprimento do profileHMM comparado e o número de EPs no sistema.

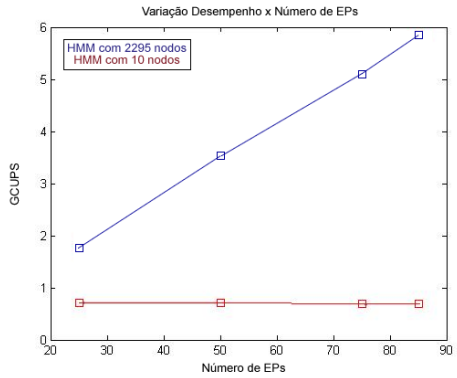


Figura 5.19: Desempenho do Sistema em GCUPS x Numero de EPs.

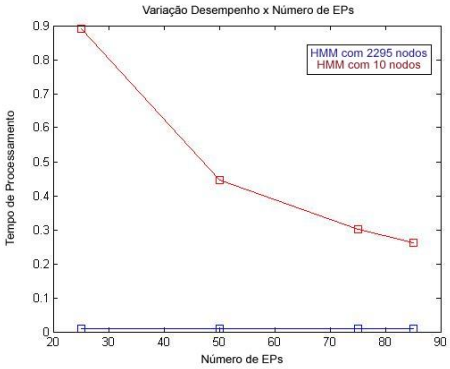


Figura 5.20: Desempenho do Sistema em Segundos x Numero de EPs.

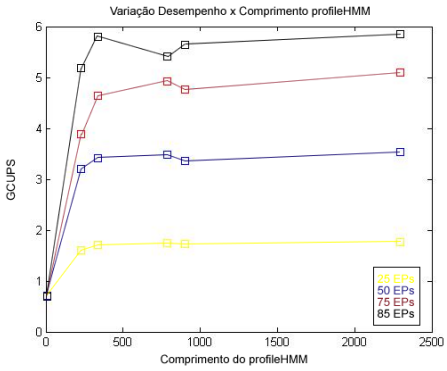


Figura 5.21: Desempenho do Sistema em GCUPS x Comprimento do ProfileHMM.

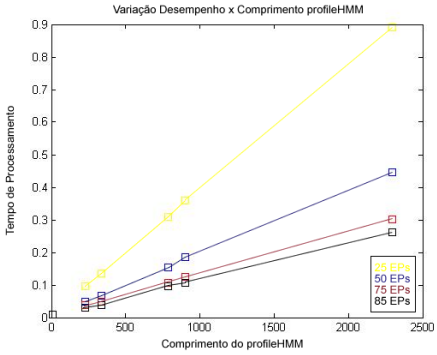


Figura 5.22: Desempenho do Sistema em Segundo x comprimento do ProfileHMM.

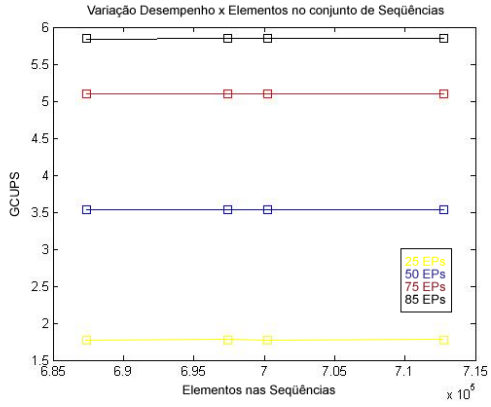


Figura 5.23: Desempenho do Sistema em GCUPS x Número de elementos das sequências de teste.

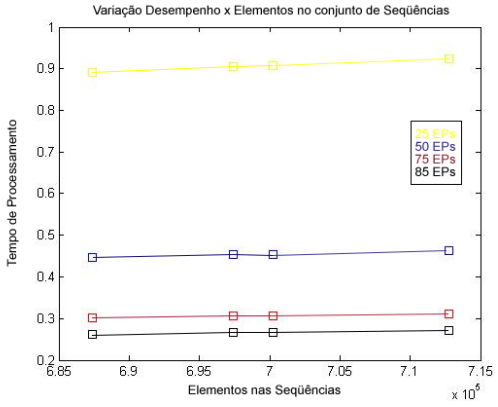


Figura 5.24: Desempenho do Sistema em Segundos x Número de elementos das sequências de teste

A figura 5.25 apresenta os resultados para o desempenho em CUPS para nossos testes, dela podemos observar como a queda no desempenho quando o numero de nodos no profileHMM não é um múltiplo exato do numero de EPs implementados no Arranjo.

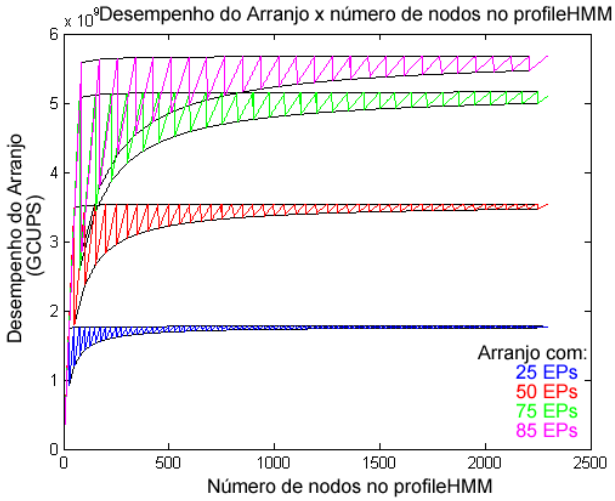


Figura 5.25: Dependencia do Desempenho com o Número de nodos no profileHMM.

5.3.3 - Estimativa do Desempenho para o Software Processando as Regiões Significativas

Como o software codificado em C/C++ para o teste do conceito das divergências não está otimizado (o HMMER está) e a porcentagem das matrizes de programação dinâmica que o software tem que re-processar varia para todos os alinhamentos, o tempo de execução para

esse software não é uma boa medida do ganho atingido no desempenho ao processar somente as regiões de interesse das matrizes de programação dinâmica. Para fazer uma estimativa realista do desempenho do software HMMER, quando processando somente as regiões de interesse, observamos o comportamento do Algoritmo de Viterbi quando é variada a pontuação considerada como significativa (limiar) para um conjunto de sequências. É necessário observar tanto o número de sequências com uma pontuação maior que o limiar quanto à porcentagem média de células na região de interesse que o software HMMER deveria reprocessar para as sequências acima desse limiar.

Realizamos a execução do Algoritmo de Viterbi para os 20 profileHMMs top da base de dados PFAM [4] utilizando os nossos quatro conjuntos de sequências aleatórios obtidos da Uniprot [3] e guardamos a pontuação para todas as sequências em um arquivo de texto de saída. Logo depois processamos esse arquivo de saída para obter uma análise estatística do número de sequências no nosso conjunto de teste acima de um limiar determinado e a porcentagem media de células na região de interesse que seriam reprocessadas pelo HMMER para essas sequências. Geramos o gráfico para essa análise observando qual foi à porcentagem média de reprocessamento obtida para as sequências significativas de cada limiar. As Figuras 5.26 até 5.29 apresentam os dados obtidos.

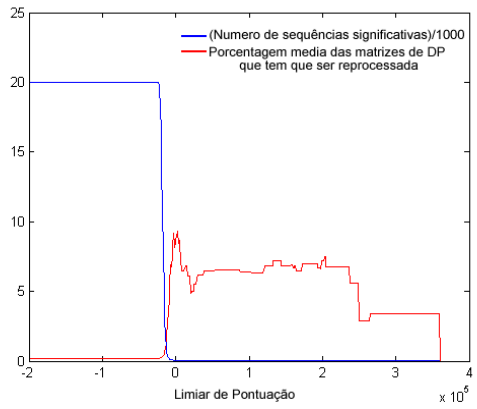


Figura 5.26: Número de sequências significativas e Porcentagem media de células x Limiar de Pontuação, para o conjunto de teste com 687406 aminoácidos.

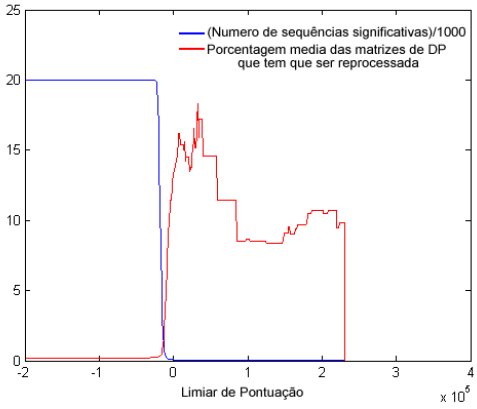


Figura 5.27: Número de sequências significativas e Porcentagem media de células x Limiar de Pontuação, para o conjunto de teste com 697407 aminoácidos.

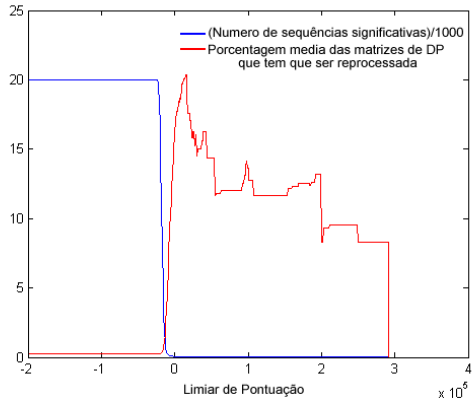


Figura 5.28: Número de sequências significativas e Porcentagem media de células x Limiar de Pontuação, para o conjunto de teste com 700218 aminoácidos.

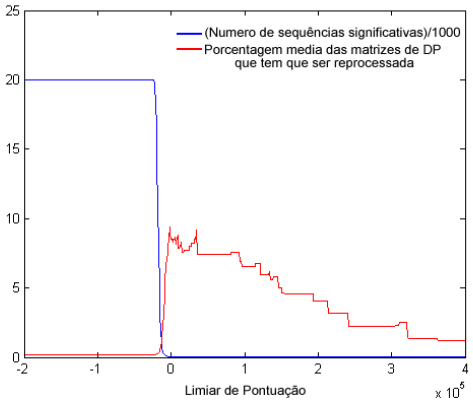


Figura 5.29: Número de sequências significativas e Porcentagem media de células x Limiar de Pontuação, para o conjunto de teste com 712734 aminoácidos.

Das figuras pode-se observar que para pontuações consideradas como significativas a porcentagem média máxima de células que o algoritmo modificado deve calcular é de 22%, e que o número de sequências significativas é aproximadamente 1% das sequências do conjunto de teste, ainda relaxando o limiar para incluir alinhamentos muito ruins. Para derivar a estatística de desempenho da segunda etapa (de reprocessamento) utilizamos os dados obtidos para o tempo de execução das 2000 sequências no HMMER e multiplicamos pela porcentagem média de sequências significativas (p_s) o que resulta no tempo que o software demoraria em produzir os alinhamentos completos para as sequências significativas. Depois, multiplicamos o tempo de execução obtido pela porcentagem média de células que o algoritmo modificado deve calcular (p_c) utilizando os dados para as divergências. Finalmente, tomamos o tempo obtido como o tempo gasto pela segunda etapa (software acelerado) para processar as células para a região de interesse das sequências significativas. A Equação 5.5 apresenta o cálculo feito para obter o tempo que o algoritmo tomaria para calcular a região de interesse onde t representa o tempo total do HMMER para realizar a comparação da sequência S_i com o profileHMM m_j.

$$t_{reg_{i,j}} = t * p_s * p_c \tag{5.5}$$

A partir da Tabela 5.1 podemos obter os tempos estimados de processamento das regiões de interesse das sequências consideradas como significantes que servirão para realizar as

comparações de desempenho na Seção 5.3.5. A Tabela 5.6 mostra os tempos obtidos para os conjuntos de sequências e os profileHMMs de teste.

Tabela 5.6: Tempos de processamento das regiões de interesse estimados.

Numero total de aminoácidos do Conjunto de Sequências	*Nodos do ProfileHMM*	*Tempo de processamento do conjunto de sequências inteiro no HMMER.*	*Tempo de processamento das sequências significativas no HMMER.*	*Tempo gasto pela segunda etapa para gerar o alinhamento partindo da região de interesse (Estimado)*
687406	*788*	*23,40*	*0,234*	*0,0515*
	10	*0,40*	*0,004*	*0,0009*
	226	*6,55*	*0,0655*	*0,0144*
	337	*9,85*	*0,0985*	*0,0217*
	2295	*74,49*	*0,7499*	*0,1639*
	901	*26,32*	*0,2632*	*0,0579*
697407	*788*	*24,34*	*0,2434*	*0,0535*
	10	*0,47*	*0,047*	*0,001*
	226	*6,68*	*0,668*	*0,0147*
	337	*9,88*	*0,988*	*0,0217*
	2295	*78,87*	*0,7887*	*0,1735*
	901	*27,55*	*0,2755*	*0,0606*
700218	*788*	*24,40*	*0,2440*	*0,0537*
	10	*0,42*	*0,042*	*0,0009*
	226	*6,76*	*0,676*	*0,0149*
	337	*10,26*	*0,1026*	*0,0226*
	2295	*81,23*	*0,8123*	*0,1787*
	901	*27,41*	*0,2741*	*0,0603*
712734	*788*	*25,42*	*0,2542*	*0,0559*
	10	*0,42*	*0,042*	*0,0009*
	226	*6,82*	*0,682*	*0,015*
	337	*10,09*	*0,1009*	*0,0222*
	2295	*81,07*	*0,8107*	*0,1784*
	901	*27,46*	*0,2746*	*0,0604*

5.3.4 - Síntese e Área

A síntese do Arranjo Sistólico implementado foi feita pelo software Quartus II da Altera [32] para uma FGPA Stratix II EP2S180F1508C3, que tem um total de 143520 ALUTs, 143520 registradores dedicados, 1171 pinos de propósito geral, 768 blocos para o processamento de sinais e 9383040 bits de memória *on-board* [33]. A síntese foi feita para uma arquitetura de 25, 50, 75 e 85 EPs, capaz de suportar um total de 25 iterações e um tamanho máximo de 8192 elementos em uma sequência *query*. Para a implementação feita no presente trabalho a arquitetura atinge freqüências desde 67MHz até 71MHz, após de programar o Quartus II para otimizar a síntese para velocidade e não para área. A Tabela 5.7 apresenta os resultados obtidos para o processo de síntese lógica. Da tabela pode se observar que o numero de bits de memória utilizados não é diretamente proporcional ao número de EPs, já que o maior número de bits de memória utilizados pelo sistema são das FIFOs de armazenamento de resultados intermediários (2097152). O número de LUTs utilizadas é proporcional ao número de EPs implementados, e é aproximadamente de 1200 LUTs para cada EP.

Tabela 5.7: Resultados da Síntese Lógica.

Número de EPs	Iterações	Nodos do ProfileHMM	Tamanho da Sequência	ALUTs	Registradores Dedicados	Bits de Memória	% de utilização	Freq. Max. de Relógio (MHz)
25	25	625	8192	31738	18252	2609152	25	71
50	25	1250	8192	59750	35294	3121152	49	71
75	25	1875	8192	93132	52520	3663152	75	69
85	25	2125	8192	103940	59285	3837952	84	67

Para a implementação foi escolhida a síntese com 85 EPs, já que é a que maior desempenho atinge para a maioria dos profileHMMs da PFAM.

5.3.5 - Desempenho do Sistema Completo e Ganhos Obtidos

Partindo das tabelas das seções anteriores e das equações apresentadas na Seção 4.4, podemos achar o tempo de execução do sistema proposto (Hardware com 85 EPs +

Software) para os profileHMMs e os conjuntos de sequências de teste. A Tabela 5.8 apresenta os tempos de execução achados.

Com os dados obtidos podem ser feitas varias comparações em termos do desempenho. A primeira é a comparação dos tempos de execução entre o software não acelerado e o sistema proposto, incluindo o tempo que a parte de software do sistema gasta em reprocessar as sequências significativas para obter o alinhamento. Gostaríamos de ressaltar que nenhum dos trabalhos relacionados fornece estatísticas para esse tempo, o qual tem influencia no desempenho. A segunda consiste na comparação da taxa de CUPS atingida pelo sistema proposto com as atingidas pelos trabalhos relacionados e a terceira é a comparação do tempo que os trabalhos relacionados gastam para produzir o alinhamento completo contra o tempo gasto pelo Algoritmo das Divergências para calcular o mesmo alinhamento somente processando a região de interesse, gerando a estatística do ganho obtido somente nessa etapa. A Tabela 5.9 apresenta a comparação dos tempos de execução entre o software não acelerado e o sistema proposto, e o ganho obtido.

Tabela 5.8: Tempo total de processamento do sistema proposto

Numero total de aminoácidos do Conjunto de Sequências	*Nodos do ProfileHMM*	*Tempo Hardware*	*Tempo Software (processando somente as regiões de interesse)*	*Tempo Total ($t_{sa(i,j)}$)*
687406	*788*	0,0971	*0,0515*	*0,1486*
	10	0,0097	*0,0009*	*0,0106*
	226	0,0291	*0,0144*	*0,0435*
	337	0,0388	*0,0217*	*0,0605*
	2295	0,2622	*0,1639*	*0,4261*
	901	0,1068	*0,0579*	*0,1647*
697407	*788*	0,0985	*0,0535*	*0,152*
	10	0,0099	*0,001*	*0,0109*
	226	0,0296	*0,0147*	*0,0443*
	337	0,0394	*0,0217*	*0,0611*
	2295	0,2660	*0,1735*	*0,4395*
	901	0,1084	*0,0606*	*0,169*
700218	*788*	0,0989	*0,0537*	*0,1526*
	10	0,0099	*0,0009*	*0,0108*
	226	0,0297	*0,0149*	*0,0446*
	337	0,0396	*0,0226*	*0,0622*
	2295	0,2671	*0,1787*	*0,4458*
	901	0,1088	*0,0603*	*0,1691*
712734	*788*	0,1007	*0,0559*	*0,1566*
	10	0,0101	*0,0009*	*0,011*
	226	0,0302	*0,015*	*0,0452*
	337	0,0403	*0,0222*	*0,0625*
	2295	0,2719	*0,1784*	*0,4503*
	901	0,1108	*0,0604*	*0,1712*

Tabela 5.9: Tempo de execução do Software não acelerado x Tempo de execução do Sistema Completo.

Numero total de aminoácidos do Conjunto de Sequências	*Nodos do ProfileHMM*	*Tempo HMMER*	*Tempo Sistema Proposto*	*Ganho Obtido*
687406	*788*	*23,40*	*0,1486*	*157,4697*
	10	*0,40*	*0,0106*	*37,7358*
	226	*6,55*	*0,0435*	*150,5747*
	337	*9,85*	*0,0605*	*162,8099*
	2295	*74,49*	*0,4261*	*174,8181*
	901	*26,32*	*0,1647*	*159,8057*
697407	*788*	*24,34*	*0,152*	*160,1316*
	10	*0,47*	*0,0109*	*43,1193*
	226	*6,68*	*0,0443*	*150,7901*
	337	*9,88*	*0,0611*	*161,7021*
	2295	*78,87*	*0,4395*	*179,4539*
	901	*27,55*	*0,169*	*163,0178*
700218	*788*	*24,40*	*0,1526*	*159,8952*
	10	*0,42*	*0,0108*	*38,8889*
	226	*6,76*	*0,0446*	*151,5695*
	337	*10,26*	*0,0622*	*164,9518*
	2295	*81,23*	*0,4458*	*182,2118*
	901	*27,41*	*0,1691*	*162,0934*
712734	*788*	*25,42*	*0,1566*	*162,3244*
	10	*0,42*	*0,011*	*38,1818*
	226	*6,82*	*0,0452*	*150,885*
	337	*10,09*	*0,0625*	*161,44*
	2295	*81,07*	*0,4503*	*180,0355*
	901	*27,46*	*0,1712*	*160,3972*

Da Tabela 5.5 pode se observar que o hardware atinge um desempenho desde 687,8 MCUPS até 5,8475 GCUPS, que comparado com a taxa de CUPS achada para o HMMER dá um ganho de entre 29 e 254 vezes a capacidade do algoritmo sem acelerar. A medida

anterior é calculada para comparar o desempenho do nosso sistema (com um EP bem mais complexo) com os sistemas já desenvolvidos em outros trabalhos, mas afirmando que uma medida de desempenho mais acertada deve incluir o tempo de reprocessamento na etapa de software, já que ela é a que gera o alinhamento para a sequência. A tabela 5.10 apresenta o desempenho do nosso sistema quando comparado com as arquiteturas analisadas na Seção 3.4.

O desempenho do nosso sistema não varia muito com relação aos trabalhos relacionados, mas a principal vantagem do aumento da complexidade no EP está na economia de tempo de processamento ao gerar o alinhamento para as sequências significativas, onde o ganho médio é de 4,54 vezes, como pode ser visto na Tabela 5.11. No cálculo do tempo de reprocessamento em software (HMMER) foi utilizado o resultado achado na Seção 5.3.5 para o número de sequências significativas em cada comparação (cerca de 1%). Dos resultados apresentados nas tabelas anteriores pode se observar que o hardware implementado em combinação com um software para o Algoritmo das Divergências é uma ferramenta de aceleração útil na hora de fazer comparações de sequências de proteínas com profileHMMs, já que o sistema completo (Hardware+Software) atinge ganhos de até 180 vezes no desempenho quando comparado com o software HMMER rodando na plataforma de testes.

Tabela 5.10: Sumario dos Trabalhos relacionados.

ref	# EPs	# nodos máximo no profileHMM	Tamanho Máximo da Seq.	Plan7 Completo	Relógio (MHz)	Performance (GCUPS)	Ganho	FPGA	Características Especiais
[7]	90	1440	1024	N	100	9	247	Xilinx 2VP100	Elimina o Estado J
[8] [10]	50	---	---	N	200	5 to 20	---	Not Synthesized	Introduz um acelerador de duas etapas.
[12]	72	1440	8192	N	74	3.95	195	XC2V6000	Elimina os Estados J e B.
[14] [31]	10	256	---	Y	70	7	300	XC3S1500	Implementa o plan7 completo
[30]	50	---	---	N	66	1.3	50	Xilinx Spartan 3 4000	Propõe uma metodologia para paralelizar os EPs
[16]	68	544	1024	N	180	10	190	Xilinx Virtex II 6000	Divide o EP em estados de *pipeline*
[9]	25	---	---	Y	130	3.2	56.8	Xilinx Virtex 5 110-T	Faz execução especulativa.
nosso	85	2295	8192	N	67	5.8	254 (182*)	Altera Stratix II 180	

*Queda no desempenho ao incluir todos os atrasos no processamento.

Tabela 5.11: Economia no tempo de reprocessamento de sequências significativas.

Numero total de aminoácidos do Conjunto de Sequências	*Nodos do ProfileHMM*	*Tempo de reprocessamento HMMER*	*Tempo de reprocessamento Software Acelerado*	*Ganho Obtido na Etapa de Reprocessamento*
687406	*788*	*0,234*	*0,0515*	*4,5436*
	10	*0,004*	*0,0009*	*4,4444*
	226	*0,0655*	*0,0144*	*4,5486*
	337	*0,0985*	*0,0217*	*4,5391*
	2295	*0,7449*	*0,1639*	*4,5448*
	901	*0,2632*	*0,0579*	*4,5457*
697407	*788*	*0,2434*	*0,0535*	*4,5495*
	10	*0,0047*	*0,001*	*4,7000*
	226	*0,0668*	*0,0147*	*4,5442*
	337	*0,0988*	*0,0217*	*4,5529*
	2295	*0,7887*	*0,1735*	*4,5458*
	901	*0,2755*	*0,0606*	*4,5462*
700218	*788*	*0,244*	*0,0537*	*4,5437*
	10	*0,0042*	*0,0009*	*4,6666*
	226	*0,0676*	*0,0149*	*4,5369*
	337	*0,1026*	*0,0226*	*4,5398*
	2295	*0,8123*	*0,1787*	*4,5456*
	901	*0,2741*	*0,0603*	*4,5456*
712734	*788*	*0,2542*	*0,0559*	*4,5474*
	10	*0,0042*	*0,0009*	*4,6666*
	226	*0,0682*	*0,015*	*4,5466*
	337	*0,1009*	*0,0222*	*4,5450*
	2295	*0,8107*	*0,1784*	*4,5442*
	901	*0,2746*	*0,0604*	*4,5463*

6 - CONCLUSÕES E TRABALHOS FUTUROS

Foi projetado um acelerador em hardware para o Algoritmo de Viterbi da arquitetura Plan7 modificada para incluir o Algoritmo das Divergências. O Arranjo Sistólico foi codificado em VHDL, sintetizado no Quartus II de Altera e Simulado no Modelsim para validar seu funcionamento e obter as estatísticas de desempenho para comparar tanto com o software HMMER quanto com os aceleradores desenvolvidos nos trabalhos relacionados.

O sistema proposto processa os dados em duas etapas. Na primeira etapa, a pontuação para o algoritmo da sequência é calculada e os dados para as divergências são obtidos. Na segunda etapa, é determinado se a sequência é significativa através de limiares definidos pelo usuário e se ela for, é reprocessada em software para gerar o alinhamento para a sequência processando somente as células delimitadas pelo Algoritmo das Divergências.

Foi feita uma caracterização completa do desempenho do software HMMER em uma plataforma de teste composta por um processador Intel Centrino Duo de 1.66GHz e 2GB de memória RAM. A partir dos resultados dessa caracterização se achou que o desempenho máximo do HMMER na plataforma é de 23,157 MCUPS e se obteve uma formula geral para calcular os tempos de processamento dele.

Dos testes feitos no HMMER, foi percebido que somente cerca de 1% das sequências comparadas são significativas para um modelo determinado, resultado que valida a proposição de um sistema conformado por uma primeira etapa de filtragem.

Foi testado também o Algoritmo das Divergências implementado em software para achar a porcentagem média de células da região de interesse que são calculadas na etapa de reprocessamento. Este resultado nos permitiu calcular uma estimativa do tempo gasto pela segunda etapa do sistema para processar as regiões de interesse das sequências significativas.

Foi desenvolvido um *testbench* para avaliar o funcionamento do Arranjo Sistólico completo com dados reais. Ele tem como entrada arquivos de texto com as probabilidades de emissão de elementos, as probabilidades de transição entre estados e os elementos das

sequências *query* e na sua saída são apresentados os resultados calculados pelo hardware para o Algoritmo das Divergências e para as pontuações do Algoritmo de Viterbi.

Uma nova medida de desempenho que inclui tanto o tempo de processamento na etapa de hardware quanto o tempo gasto na etapa de reprocessamento é proposta. Baseando-nos nesta medida, achamos que o sistema proposto gera um ganho de entre 37 e 182 vezes quando comparado com o HMMER executado somente em software.

Utilizando a medida clássica de desempenho baseada em CUPS, obtemos que o ganho da primeira etapa do sistema (o Hardware) está entre 29 e 254 vezes o desempenho atingido pelo HMMER. Esse resultado é muito bom quando comparado aos trabalhos relacionados já que nosso EP é muito mais complexo do que os EPs implementados nesses trabalhos. Também se acha no trabalho que comparada com a etapa de reprocessamento dos trabalhos relacionados, as quais executam o HMMER para as matrizes de programação dinâmica inteiras, a nossa etapa que implementa o Algoritmo das Divergências apresenta um ganho de 4,54x quando comparados os tempos de processamento.

Como conclusão deste trabalho temos dois artigos submetidos, um para congresso e outro para revista. O primeiro foi submetido para o congresso DATE 2010 e o segundo foi submetido para uma edição especial em computação reconfigurável da revista IEEE Transactions on Parallel And Distributed Processing.

A implementação da unidade de Comunicação e Controle do sistema não foi feita por não contar com o recurso físico para a sua prototipagem, mas as Máquinas de Estados Finitos necessárias para seu funcionamento foram propostas e seu funcionamento foi emulado no código funcional do *testbench*. Uma interface de barramento para o sistema deve ser definida e a implementação de um código sintetizável para a Unidade de Comunicação e Controle é deixada como trabalho futuro.

Mais um trabalho futuro consiste na integração do sistema completo com o PC host para avaliar o desempenho do sistema completo incluindo overheads de programação e comunicação.

A implementação de um acelerador que inclua o algoritmo das divergências numa versão completa do Viterbi-Plan7 para avaliar o desempenho do sistema e a aceleração que ele obtém com respeito ao HMMER sem acelerar, é deixada também como trabalho futuro.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] HMMER: biosequence analysis using profile hidden Markov models. http://hmmer.janelia.org, 2006.

[2] S Eddy, S. R. (1998) “Profile hidden Markov Models.” In: Bioinformatics, 14(9), 755-763.

[3] The Universal Protein Resource (UniProt) Home Page. http://www.uniprot.org. Last Access: June 2009.

[4] Sanger’s Institute PFAM protein sequence database Home Page. http://pfam.sanger.ac.uk/. Last Access: May 2009.

[5] Durbin, R., Eddy, S., Krogh, A. and Mitchison, Biological Sequence Analysis Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press 2008.

[6] Leon, D. and Markel, S., Sequence Analysis in a nutshell: A guide to Common Tools and Databases. O’Reilly 2003.

[7] K. Benkrid, V. Panagiotis and K. Server (2008) “A High Performance Reconfigurable Core for Motif Searching Using Profile HMM.” In: Proceedings of the 2008 IEEE Adaptive Hardware Systems (AHS 2008), Amsterdam, Hetherlands. 285-292.

[8] Maddimsetty, R. P., Buhler, J., Chamberlain, R. D., Franklin, M. A., Harris B. (2006) “Accelerator design for protein sequence HMM search.” In: Proceedings of the 20th annual international conference on Supercomputing. Cairns, Queensland, Australia.

[9] Sun, Y., Li, P., Gu, G., Wen, Y., Liu, Y., and Liu, D. (2009) “HMMer acceleration using systolic array based reconfigurable architecture.” In: Proceeding of the ACM/SIGDA international Symposium on Field Programmable Gate Arrays. Monterey, California, USA. 282-282.

[10] Maddimsetty, R. P. (2006) Acceleration of Profile-HMM Search for Protein Sequences in Reconfigurable Hardware. Master's Thesis, School of Engineering and Applied Science, Washignton University, 80p.

[11] Improving MPI-HMMER’s Scalability With Parallel I/O. Technical Report. http://www.cse.buffalo.edu/tech-reports/2008-11.pdf.

[12] Oliver T.; Schmidt B.; Jakop Y.; Maskell D. (2009) “High Speed Biological Sequence Analysis with Hidden Markov Models on Reconfigurable Platforms.” In: IEEE Transactions on Information Technology in Biomedicine, Accepted for future publication. Volume PP, Forthcoming.

[13] Horn, D. R., Houston, M. and Hanrahan, P. (2005) “ClawHMMER: A Streaming HMMer-Search Implementation.” In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing. Seattle, Washington, USA. 11-19.

[14] Walters, J. P., Meng, X., Chaudhary, V., Oliver, T., Yeow, L. Y., Schmidt, B., Nathan, D., and Landman, J. (2007) “MPI-HMMER-Boost: Distributed FPGA Acceleration.” In: Journal of VLSI Signal Processing Systems, 48(3), 223-238.

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

[16] Jacob, A.C., Lancaster, J.M., Buhler, J.D., Chamberlain, R.D. (2007) “Preliminary results in accelerating profile HMM search on FPGAs.” In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2007. Long Beach, California, USA.

[17] HMMER User Guide. http://hmmer.janelia.org.

[18] Fickett, J.W. (1984) “Fast optimal alignments.” In: Nucleic Acids Res. 12 (1). 175-179.

[19] Batista, R. B., Boukerche, A., and de Melo, A. C. (2008) “A parallel strategy for biological sequence alignment in restricted memory space.” In: Journal of Parallel and Distribituted Computing, 68(4), 548-561.

[20] FASTA web page. http://www.ebi.ac.uk/Tools/fasta/index.html. Last Access, June 2009.

[21] BLAST web page. http://blast.ncbi.nlm.nih.gov/Blast.cgi. Last Access, June 2009.

[22] HMMER on the Sun Grid Project web page. https://hmmer.dev.java.net/. Last Access, July 2009.

[23] GPUHMMER user guide. http://mpihmmer.org/userguideGPUHMMER.htm.

[24] Protein Motifs in Sequence Analysis. www.sinauer.com_pdf_nsp-protein-1-16.

[25] http://www.dnabaser.com/articles/sequence alignment/Global-local-alignment.png

[26] Needleman, S. B. and Wunsch, C. D. (1970) “A general method applicable to the search for similarities in the amino acid sequence of two proteins.” In: Journal of Molecular Biology, (48), 443-453.

[27] Gotoh, O. (1982) “An improved algorithm for matching biological sequences.” In: Journal of Molecular Biology, (162), 705-708.

[28] Smith, T. F. and Waterman, M. S. (1981) “Identification of common molecular subsequences.” In: Journal of Molecular Biology, (147), 195-197.

[29] SledgeHMMER web page. http://bioapps.rit.albany.edu/sledgeHMMER/. Last Access, July 2009.

[30] Derrien, S., Quinton, P. (2007) “Parallelizing HMMER for Hardware Acceleration on FPGAs.” In: IEEE International Conference on Application -specific Systems, Architectures and Processors, ASAP 2007. Montréal, Canada.

[31] Oliver, T., Yeow, L. Y., and Schmidt, B. (2008) “Integrating FPGA acceleration into HMMer.” In: Parallel Computing, (34)11, 681-691.

[32] Altera Modelsim product Brief. http://www.altera.com/products/software/quartus-ii/modelsim/qts-modelsim-index.html. Last Access, July 2009.

[33] Stratix 2 reference manual, Altera Web Page. http://www.altera.com/literature/lit-stx2.jsp. Last Access July 2009.

[34] Dell PCIExpress Technology Blog. http://www.dell.com/content/topics/global.aspx/vectors/en/2004_pciexpress?c=us&l=en. Last Access, July 2009.

[35] HyperTransport Consortium web page. http://www.hypertransport.org/default.cfm?page=Technology. Last Access, July 2009.

[36] Protein Motifs. http:// www.sinauer.com/pdf/nsp-protein-1-16.pdf. Last Access, August 2009.

[37] Moreano, N. Trabalho de pos-doutorado desenvolvido no Departamento de Ciencias da Computação da Universidade de Brasilia. Brasilia/DF, Brasil. 2009.

[38] A HHMER Hardware Accelerator using Divergences. Artigo Aceito para Publicação no congresso DATE 2010. Dresden/Alemanha.

[39] Conjunto de instruções Intel SS2. http://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations. Last Access, November 2009.

APÊNDICES

A - CÓDIGO FONTE DO ALGORITMO DE VITERBI MODIFICADO

O código do algoritmo em C consiste de quatro arquivos que programam as funções necessárias para o funcionamento do mesmo. No arquivo min_max.c são implementadas as funções de máximos e mínimos para as equações de recorrência apresentadas no capitulo 4, no arquivo read.c são implementadas as funções de leitura externa de dados, no arquivo print_exat.c são implementadas as funções de visualização de resultados e no arquivo viterbi_exat.c são implementadas as funções de processamento, tanto o algoritmo de Viterbi quanto o *traceback*.

A.1 – VITERBI_EXAT.C

```
/*********************************************************************
 Viterbi.c
 Implements the Viterbi algorithm: given a hidden Markov models (HMMs)
and a
 sequence of observations, find the sequence of states in the HMM that
has the
 highest probability of producing that sequence of observations.

 The sequence of observations corresponds to a sequence of aminoacids,
and the
 HMM corresponds to a family of proteins.

 The HMM architecture used is the Plan7, without the J state.
 The probabilities are represented as integer scores.

*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* Constants */

#define INFTY                       987654321
#define MAX_LEN_SEQ          27000
#define MAX_NODES            600
#define ALPHABET_SIZE   20
#define MAX_LEN_LINE         1000

typedef struct
{
      char state ;
      int i, j ;
} t_trc ;

/*-------------------------------------------------------------------*/
/* Global declarations */
```

```
int n_nodes, /* Number of HMM nodes */
       len_seq, /* Sequence length */
    len_trc ;

/* Emission probabilities */
int em_M[MAX_NODES+1][ALPHABET_SIZE],
       em_I[MAX_NODES+1][ALPHABET_SIZE] ;

/* Transition probabilities */
int tr_N_B,
    tr_N_N,
    tr_E_C,
    tr_C_T,
    tr_C_C ;

int tr_M_M[MAX_NODES+1],  /* Mj -> Mj+1 */
    tr_M_I[MAX_NODES+1],  /* Mj -> Ij   */
    tr_M_D[MAX_NODES+1],  /* Mj -> Dj+1 */
    tr_I_M[MAX_NODES+1],  /* Ij -> Mj+1 */
    tr_I_I[MAX_NODES+1],  /* Ij -> Ij   */
    tr_D_M[MAX_NODES+1],  /* Dj -> Mj+1 */
    tr_D_D[MAX_NODES+1],  /* Dj -> Dj+1 */
    tr_B_M[MAX_NODES+1],  /* B  -> Mj   */
    tr_M_E[MAX_NODES+1] ; /* Mj -> E    */

char alphabet[ALPHABET_SIZE] = "ACDEFGHIKLMNPQRSTVWY" ;

char seq[MAX_LEN_SEQ+1] ;

int seq_ind[MAX_LEN_SEQ+1] ;

/* Matrizes e vetores da programação dinâmica, calculados pelo algoritmo
de Viterbi */
int M[MAX_LEN_SEQ+1][MAX_NODES+1],
    I[MAX_LEN_SEQ+1][MAX_NODES+1],
    D[MAX_LEN_SEQ+1][MAX_NODES+1],
    XB[MAX_LEN_SEQ+1],
    XC[MAX_LEN_SEQ+1],
    XE[MAX_LEN_SEQ+1] ;

/* Matrizes e vetores da programação dinâmica, calculados pelo algoritmo
de Viterbi,
   apenas na área delimitada pelas divergências */
int M_d[MAX_LEN_SEQ+1][MAX_NODES+1],
    I_d[MAX_LEN_SEQ+1][MAX_NODES+1],
    D_d[MAX_LEN_SEQ+1][MAX_NODES+1],
    XB_d[MAX_LEN_SEQ+1],
    XC_d[MAX_LEN_SEQ+1],
    XE_d[MAX_LEN_SEQ+1] ;

/* Caminho da programação dinâmica */
int M_trc[MAX_LEN_SEQ+1][MAX_NODES+1],
       I_trc[MAX_LEN_SEQ+1][MAX_NODES+1],
       D_trc[MAX_LEN_SEQ+1][MAX_NODES+1],
       M_trc_d[MAX_LEN_SEQ+1][MAX_NODES+1] ;

/* Divergências inferior e superior e linhas inicial e final */
int M_div_inf[MAX_LEN_SEQ+1][MAX_NODES+1],
    M_div_sup[MAX_LEN_SEQ+1][MAX_NODES+1],
    M_lin_ini[MAX_LEN_SEQ+1][MAX_NODES+1],
```

```
    I_div_inf[MAX_LEN_SEQ+1][MAX_NODES+1],
    I_div_sup[MAX_LEN_SEQ+1][MAX_NODES+1],
    I_lin_ini[MAX_LEN_SEQ+1][MAX_NODES+1],
       D_div_inf[MAX_LEN_SEQ+1][MAX_NODES+1],
    D_div_sup[MAX_LEN_SEQ+1][MAX_NODES+1],
    D_lin_ini[MAX_LEN_SEQ+1][MAX_NODES+1],
    XE_div_inf[MAX_LEN_SEQ+1],
    XE_div_sup[MAX_LEN_SEQ+1],
    XE_lin_ini[MAX_LEN_SEQ+1],
    XC_div_inf[MAX_LEN_SEQ+1],
    XC_div_sup[MAX_LEN_SEQ+1],
    XC_lin_ini[MAX_LEN_SEQ+1],
    XC_lin_fin[MAX_LEN_SEQ+1] ;

int div_inf,
    div_sup,
    lin_ini,
    lin_fin,
    col_fin ;

/* Trace */
t_trc trace[2*MAX_LEN_SEQ] ;

/* Numero células da matriz M de programação dinâmica calculadas pelo
algoritmo de Viterbi,
   completo e apenas na área delimitada pelas divergências */
int n_celulas_tot,
    n_celulas_div ;

/*---------------------------------------------------------------------*/

#include "read.c"
#include "min_max.c"
#include "print_exat.c"

/*---------------------------------------------------------------------*/
/* Implementation of algorithm 6, with computation of divergences
*/

int viterbi(void)
{
      int i, j, score, index_max ;

      n_celulas_tot = len_seq * n_nodes ;

      /* Initialize scores */
      for (j = 0 ; j <= n_nodes ; j ++)
      {
            M[0][j] = -INFTY ;
            I[0][j] = -INFTY ;
            D[0][j] = -INFTY ;
      }
      for (i = 1 ; i <= len_seq ; i ++)
      {
            M[i][0] = -INFTY ;
            I[i][0] = -INFTY ;
            D[i][0] = -INFTY ;
            XE[i]   = -INFTY ;
      }
      XB[0] = tr_N_B ;
      XC[0] = -INFTY ;
```

```
      /* Initialize divergences */
      for (i = 1 ; i <= len_seq ; i ++)
      {
            M_div_inf[i][0] = i ;
            M_div_sup[i][0] = i ;
            M_lin_ini[i][0] = i ;
            I_div_inf[i][0] = i ;
            I_div_sup[i][0] = i ;
            I_lin_ini[i][0] = i ;
            D_div_inf[i][0] = i ;
            D_div_sup[i][0] = i ;
            D_lin_ini[i][0] = i ;
            XE_div_inf[i]   = 0 ;
            XE_div_sup[i]   = 0 ;
            XE_lin_ini[i]   = 0 ;
      }
      for (j = 0 ; j <= n_nodes ; j ++)
      {
            M_div_inf[0][j] = -j ;
            M_div_sup[0][j] = -j ;
            M_lin_ini[0][j] = 0 ;
            I_div_inf[0][j] = -j ;
            I_div_sup[0][j] = -j ;
            I_lin_ini[0][j] = 0 ;
            D_div_inf[0][j] = -j ;
            D_div_sup[0][j] = -j ;
            D_lin_ini[0][j] = 0 ;
      }
      XC_div_inf[0] = 0 ;
      XC_div_sup[0] = 0 ;
      XC_lin_ini[0] = 0 ;
      XC_lin_fin[0] = 0 ;

      /* For each symbol of sequence */
      for (i = 1 ; i <= len_seq ; i ++)
      {
            XB[i] = XB[i-1] + tr_N_N ;

            /* For each node of HMM */
            for (j = 1 ; j <= n_nodes ; j ++)
            {
                  M[i][j] = em_M[j][seq_ind[i]] + max4i(M[i-1][j-1] +
tr_M_M[j-1],
                                                        I[i-1][j-1] +
tr_I_M[j-1],
                                                        D[i-1][j-1] +
tr_D_M[j-1],
                                                        XB[i-1]     +
tr_B_M[j],
                                                        &index_max) ;
                  M_trc[i][j] = index_max ;

                  switch (index_max)
                  {
                        case 0 :
                              M_div_inf[i][j] = M_div_inf[i-1][j-1] ;
                              M_div_sup[i][j] = M_div_sup[i-1][j-1] ;
                              M_lin_ini[i][j] = M_lin_ini[i-1][j-1] ;
                        break ;
                        case 1 :
```

```
                              M_div_inf[i][j] = max2(i-j, I_div_inf[i-
1][j-1]) ;
                              M_div_sup[i][j] = min2(i-j, I_div_sup[i-
1][j-1]) ;
                              M_lin_ini[i][j] = I_lin_ini[i-1][j-1] ;
                        break ;
                        case 2 :
                              M_div_inf[i][j] = max2(i-j, D_div_inf[i-
1][j-1]) ;
                              M_div_sup[i][j] = min2(i-j, D_div_sup[i-
1][j-1]) ;
                              M_lin_ini[i][j] = D_lin_ini[i-1][j-1] ;
                        break ;
                        case 3 :
                              M_div_inf[i][j] = i-j ;
                              M_div_sup[i][j] = i-j ;
                              M_lin_ini[i][j] = i ;
                        break ;
                  }

                  I[i][j] = em_I[j][seq_ind[i]] + max2i(M[i-1][j] +
tr_M_I[j],
                                                        I[i-1][j] +
tr_I_I[j],
                                                        &index_max) ;
                  I_trc[i][j] = index_max ;

                  if (index_max == 0)
                  {
                        I_div_inf[i][j] = max2(i-j, M_div_inf[i-1][j]) ;
                        I_div_sup[i][j] = min2(i-j, M_div_sup[i-1][j]) ;
                        I_lin_ini[i][j] = M_lin_ini[i-1][j] ;
                  }
                  else /* index_max == 1 */
                  {
                        I_div_inf[i][j] = max2(i-j, I_div_inf[i-1][j]) ;
                        I_div_sup[i][j] = min2(i-j, I_div_sup[i-1][j]) ;
                        I_lin_ini[i][j] = I_lin_ini[i-1][j] ;
                  }

                  D[i][j] = max2i(M[i][j-1] + tr_M_D[j-1],
                                  D[i][j-1] + tr_D_D[j-1],
                                  &index_max) ;
                  D_trc[i][j] = index_max ;

                  if (index_max == 0)
                  {
                        D_div_inf[i][j] = max2(i-j, M_div_inf[i][j-1]) ;
                        D_div_sup[i][j] = min2(i-j, M_div_sup[i][j-1]) ;
                        D_lin_ini[i][j] = M_lin_ini[i][j-1] ;
                  }
                  else /* index_max == 1 */
                  {
                        D_div_inf[i][j] = max2(i-j, D_div_inf[i][j-1]) ;
                        D_div_sup[i][j] = min2(i-j, D_div_sup[i][j-1]) ;
                        D_lin_ini[i][j] = D_lin_ini[i][j-1] ;
                  }

                  XE[i]   = max2i(XE[i],
                                  M[i][j] + tr_M_E[j],
                                  &index_max) ;
```

```
                  if (index_max == 1)
                  {
                        XE_div_inf[i] = M_div_inf[i][j] ;
                        XE_div_sup[i] = M_div_sup[i][j] ;
                        XE_lin_ini[i] = M_lin_ini[i][j] ;
                        col_fin = j ;
                  }
            } /* for j */

            XC[i] = max2i(XC[i-1] + tr_C_C,
                          XE[i]   + tr_E_C,
                          &index_max) ;

            if (index_max == 0)
            {
                  XC_div_inf[i] = XC_div_inf[i-1] ;
                  XC_div_sup[i] = XC_div_sup[i-1] ;
                  XC_lin_ini[i] = XC_lin_ini[i-1] ;
                  XC_lin_fin[i] = XC_lin_fin[i-1] ;
            }
            else
            {
                  XC_div_inf[i] = XE_div_inf[i] ;
                  XC_div_sup[i] = XE_div_sup[i] ;
                  XC_lin_ini[i] = XE_lin_ini[i] ;
                  XC_lin_fin[i] = i ;
            }
      } /* for i */

      score = XC[len_seq] + tr_C_T ;

      div_inf = XC_div_inf[len_seq] ;
      div_sup = XC_div_sup[len_seq] ;
      lin_ini = XC_lin_ini[len_seq] ;
      lin_fin = XC_lin_fin[len_seq] ;

      return score ;

} /* viterbi */

/*--------------------------------------------------------------------------*/
/* Implementation of algorithm 6, but computing only cells in the region
   delimited by divergences computed previously
*/

int viterbi_divergence(void)
{
      int i, j, score_div, index_max, j_ini, j_fin ;

      n_celulas_div = 0 ;

      /* Initialize scores */

      /* Initialize line before initial line */
      j_ini = max2(0,       lin_ini - 1 - div_inf) ;
      j_fin = min2(n_nodes, lin_ini - 1 - div_sup + 1) ;

      for (j = j_ini ; j <= j_fin ; j ++)
      {
            M_d[lin_ini - 1][j] = -INFTY ;
```

```
            I_d[lin_ini - 1][j] = -INFTY ;
            D_d[lin_ini - 1][j] = -INFTY ;
      }
      XB_d[lin_ini - 1] = tr_N_B + ((lin_ini - 1) * tr_N_N) ;
      XC_d[lin_ini - 1] = -INFTY ;

      /* Initialize anti-diagonal below inferior divergence and anti-
diagonal above superior divergence*/
      for (i = lin_ini ; i <= lin_fin ; i ++)
      {
            j_ini = max2(0,       i - div_inf - 1) ;
            j_fin = min2(n_nodes, i - div_sup + 1) ;

            M_d[i][j_ini] = -INFTY ;
            I_d[i][j_ini] = -INFTY ;
            D_d[i][j_ini] = -INFTY ;

            M_d[i][j_fin] = -INFTY ;
            I_d[i][j_fin] = -INFTY ;
            D_d[i][j_fin] = -INFTY ;

            XE_d[i]       = -INFTY ;
      }

      /* For each symbol of sequence */
      for (i = lin_ini ; i <= lin_fin ; i ++)
      {
            XB_d[i] = XB_d[i-1] + tr_N_N ;

            j_ini = max2(1,       i - div_inf) ;
            j_fin = min2(n_nodes, i - div_sup) ;

            n_celulas_div += j_fin - j_ini + 1 ;

            /* For each node of HMM */
            for (j = j_ini ; j <= j_fin ; j ++)
            {
                  M_d[i][j]   = em_M[j][seq_ind[i]] + max4i(M_d[i-1][j-1]
+ tr_M_M[j-1],
                                                            I_d[i-1][j-1]
+ tr_I_M[j-1],
                                                            D_d[i-1][j-1]
+ tr_D_M[j-1],
                                                            XB_d[i-1]
+ tr_B_M[j],
                                                            &index_max) ;
                  M_trc_d[i][j] = index_max ;

                  I_d[i][j]   = em_I[j][seq_ind[i]] + max2i(M_d[i-1][j]
+ tr_M_I[j],
                                                              I_d[i-1][j]
+ tr_I_I[j],
                                                              &index_max)
;

                  D_d[i][j]   = max2i(M_d[i][j-1] + tr_M_D[j-1],
                                      D_d[i][j-1] + tr_D_D[j-1],
                                      &index_max) ;

                  XE_d[i]     = max2i(XE_d[i],
                                      M_d[i][j] + tr_M_E[j],
```

```
                                        &index_max) ;
                  if (index_max == 1)
                  {
                        col_fin = j ;
                  }
            } /* for j */

            XC_d[i] = max2i(XC_d[i-1] + tr_C_C,
                            XE_d[i]   + tr_E_C,
                            &index_max) ;
      } /* for i */

      score_div = XC_d[lin_fin] + tr_C_C * (len_seq - lin_fin) + tr_C_T ;

      return score_div ;

} /* viterbi_divergence */

/*--------------------------------------------------------------------------*/

void trace_back_viterbi_divergence(void)
{
      int i,
          j,
          t ;
      char last_state ;

      t_trc trace_inv[2*MAX_LEN_SEQ] ;

      t = 0 ;

      trace_inv[t].state = 'T' ;
      trace_inv[t].i     = 0 ;
      trace_inv[t].j     = 0 ;
      t ++ ;

      for (i = len_seq ; i > lin_fin ; i --)
      {
            trace_inv[t].state = 'C' ;
            trace_inv[t].i     = i ;
            trace_inv[t].j     = 0 ;
            t ++ ;
      }

      trace_inv[t].state = 'C' ;
      trace_inv[t].i     = 0 ;
      trace_inv[t].j     = 0 ;
      t ++ ;

      trace_inv[t].state = 'E' ;
      trace_inv[t].i     = 0 ;
      trace_inv[t].j     = 0 ;
      t ++ ;

      trace_inv[t].state = 'M' ;
      trace_inv[t].i     = lin_fin ;
      trace_inv[t].j     = col_fin ;
      t ++ ;
      last_state = 'M' ;

      i = lin_fin ;
```

```
j = col_fin ;

while (M_trc[i][j] != 3)
{
      switch (M_trc[i][j])
      {
            case 0 :
                  i -- ;
                  j -- ;
                  trace_inv[t].state = 'M' ;
                  trace_inv[t].i     = i ;
                  trace_inv[t].j     = j ;
                  t ++ ;
                  last_state = 'M' ;
            break ;
            case 1 :
                  i -- ;
                  trace_inv[t].state = 'I' ;
                  trace_inv[t].i     = i ;
                  trace_inv[t].j     = j - 1 ;
                  t ++ ;
                  last_state = 'I' ;
            break ;
            case 2 :
                  j -- ;
                  trace_inv[t].state = 'D' ;
                  trace_inv[t].i     = 0 ;
                  trace_inv[t].j     = j ;
                  t ++ ;
                  last_state = 'D' ;
            break ;
      }
}

trace_inv[t].state = 'B' ;
trace_inv[t].i     = 0 ;
trace_inv[t].j     = 0 ;
t ++ ;

for (i = lin_ini - 1 ; i >= 1 ; i --)
{
      trace_inv[t].state = 'N' ;
      trace_inv[t].i     = i ;
      trace_inv[t].j     = 0 ;
      t ++ ;
}

trace_inv[t].state = 'N' ;
trace_inv[t].i     = 0 ;
trace_inv[t].j     = 0 ;
t ++ ;

trace_inv[t].state = 'S' ;
trace_inv[t].i     = 0 ;
trace_inv[t].j     = 0 ;
t ++ ;

len_trc = t ;

/* Reverse trace */
```

```
      for (t = len_trc - 1 ; t >= 0 ; t --)
      {
            trace[len_trc - 1 - t] = trace_inv[t] ;
      }

} /* trace_back_viterbi_divergence */

/*-------------------------------------------------------------------------*/

int main(int argc, char **argv)
{
      int s, i, score, score_div, end_f_seq ;
      int min_len_seq, max_len_seq ;
      double avg_len_seq, reduc, avg_reduc, worst_reduc, best_reduc ;
      FILE *f_seq ;

      if (argc != 3)
      {
            printf("Error: viterbi <hmm file> <sequence file>\n") ;
            exit(0) ;
      }

      /* Open HMM file (hmmn format), read HMM from file, and close HMM
file */
      read_hmm_file(argv[1]) ;

      //print_hmm() ;

      /* Open sequences file (FASTA format) */
      f_seq = open_sequence_file(argv[2]) ;

      printf("HMM file %s\n", argv[1]) ;
      printf("Nodes %d\n", n_nodes) ;
      printf("Sequence file %s\n", argv[2]) ;
      printf("Seq   Length   Score/ScoreDiv   ColFin   LinIni/LinFin
DivInf/DivSup   CelsTot/CelsDiv   %% Cels\n") ;

      avg_len_seq = 0.0 ;
      min_len_seq = MAX_LEN_SEQ ;
      max_len_seq = 0 ;

      avg_reduc   = 0.0 ;
      worst_reduc = 0.0 ;
      best_reduc  = 100.0 ;

      s         = 0 ;
      end_f_seq = 0 ;

      while (! end_f_seq)
      {
            s ++ ;

            end_f_seq = read_sequence(f_seq, seq) ;

            len_seq = strlen(seq) - 1 ;

            seq_ind[0] = -1 ;
            for (i = 1 ; i <= len_seq ; i ++)
            {
                  seq_ind[i] = symbol_index(seq[i]) ;
            }
```

```
            score = viterbi() ;

            score_div = viterbi_divergence() ;

            avg_len_seq += (double) len_seq ;
            if (len_seq < min_len_seq)
            {
                  min_len_seq = len_seq ;
            }
            if (len_seq > max_len_seq)
            {
                  max_len_seq = len_seq ;
            }

            reduc = (double) (n_celulas_div * 100) / (double)
n_celulas_tot ;
            avg_reduc += reduc ;
            if (reduc > worst_reduc)
            {
                  worst_reduc = reduc ;
            }
            if (reduc < best_reduc)
            {
                  best_reduc = reduc ;
            }

            //trace_back_viterbi_divergence() ;

            printf("%3d %8d %7d/%8d %8d %8d/%6d %8d/%6d %9d/%7d %8.1f
%s\n",
                   s, len_seq, score, score_div, col_fin, lin_ini,
lin_fin, div_inf, div_sup,
                   n_celulas_tot, n_celulas_div, reduc, (score !=
score_div) ? " ==> ERROR" : "") ;

            //print_dp(M, I, D, XB, XE, XC) ;
            //print_divergence() ;
            //print_dp(M_d, I_d, D_d, XB_d, XE_d, XC_d) ;
            //print_dp_path(M_trc, I_trc, D_trc) ;
            //print_trace(len_trc, trace) ;
      }

      fclose(f_seq) ;

      avg_len_seq = avg_len_seq / (double) s ;
      avg_reduc   = avg_reduc / (double) s ;

      printf("Sequences: Total   Average_Len   Min_Len   Max_Len\n") ;
      printf("           %5d   %11.1f   %7d   %7d\n", s, avg_len_seq,
min_len_seq, max_len_seq) ;
      printf("Reduction: %% Average   %% Worst   %% Best\n") ;
      printf("           %9.1f   %7.1f   %6.1f\n", avg_reduc,
worst_reduc, best_reduc) ;

      return 0 ;

} /* main */

/*---------------------------------------------------------------*/
```

A.2 - MIN_MAX.C

```
/*-------------------------------------------------------------------------*/

int min2(int a0, int a1)
{
      int min ;
      min = a0 ;
      if (a1 < min)
      {
            min = a1 ;
      }
      return min ;
} /* min2 */

/*-------------------------------------------------------------------------*/

int max2(int a0, int a1)
{
      int max ;
      max = a0 ;
      if (a1 > max)
      {
            max = a1 ;
      }
      return max ;
} /* max2 */

/*-------------------------------------------------------------------------*/

int max2i(int a0, int a1, int *index_max)
{
      int max ;
      max          = a0 ;
      (*index_max) = 0 ;
      if (a1 > max)
      {
            max          = a1 ;
            (*index_max) = 1 ;
      }
      return max ;
} /* max2i */

/*-------------------------------------------------------------------------*/

int max3(int a0, int a1, int a2)
{
      int max ;
      max = a0 ;
      if (a1 > max)
      {
            max = a1 ;
      }
      if (a2 > max)
      {
            max = a2 ;
      }
      return max ;
```

```
} /* max3 */

/*--------------------------------------------------------------------------*/

int min3(int a0, int a1, int a2)
{
      int min ;
      min = a0 ;
      if (a1 < min)
      {
            min = a1 ;
      }
      if (a2 < min)
      {
            min = a2 ;
      }
      return min ;
} /* min3 */

/*--------------------------------------------------------------------------*/

int max4i(int a0, int a1, int a2, int a3, int *index_max)
{
      int max ;
      max          = a0 ;
      (*index_max) = 0 ;
      if (a1 > max)
      {
            max          = a1 ;
            (*index_max) = 1 ;
      }
      if (a2 > max)
      {
            max          = a2 ;
            (*index_max) = 2 ;
      }
      if (a3 > max)
      {
            max          = a3 ;
            (*index_max) = 3 ;
      }
      return max ;
} /* max4i */

/*--------------------------------------------------------------------------*/
```

A.3 – PRINT_EXAT.C

```
/*--------------------------------------------------------------------------*/

void print_hmm(void)
{
      int i, j ;

      printf("------------------------------------------------------------
----------------------------------------\n") ;
      printf("HMM - %d nodes\n", n_nodes) ;

      printf("\nM emission probabilities\n      ") ;
```

```
      for (i = 0 ; i < ALPHABET_SIZE ; i ++)
      {
            printf(" %8c", alphabet[i]) ;
      }
      printf("\n") ;

      for (j = 0 ; j <= n_nodes ; j ++)
      {
            printf("%3d   ", j) ;

            for (i = 0 ; i < ALPHABET_SIZE ; i ++)
            {
                  printf(" %8d", em_M[j][i]) ;
            }
            printf("\n") ;
      }

      printf("\nI emission probabilities\n      ") ;

      for (i = 0 ; i < ALPHABET_SIZE ; i ++)
      {
            printf(" %8c", alphabet[i]) ;
      }
      printf("\n") ;

      for (j = 0 ; j <= n_nodes ; j ++)
      {
            printf("%3d   ", j) ;

            for (i = 0 ; i < ALPHABET_SIZE ; i ++)
            {
                  printf(" %8d", em_I[j][i]) ;
            }
            printf("\n") ;
      }

      printf("\nTransition probabilities\n") ;
      printf("  N->B   N->N   E->C   C->T   C->C\n") ;
      printf("%6d %6d %6d %6d %6d\n", tr_N_B, tr_N_N, tr_E_C, tr_C_T,
tr_C_C) ;

      printf("\n      M->M   M->I   M->D   I->M   I->I   D->M   D->D   B-
>M   M->E\n") ;

      for (j = 0 ; j <= n_nodes ; j ++)
      {
            printf("%3d %6d %6d %6d %6d %6d %6d %6d %6d %6d\n", j,
tr_M_M[j], tr_M_I[j], tr_M_D[j], tr_I_M[j],
                        tr_I_I[j], tr_D_M[j], tr_D_D[j], tr_B_M[j],
tr_M_E[j]) ;
      }

} /* print_hmm */

/*--------------------------------------------------------------------*/

void print_dp(int Ms[MAX_LEN_SEQ+1][MAX_NODES+1], int
Is[MAX_LEN_SEQ+1][MAX_NODES+1],
              int Ds[MAX_LEN_SEQ+1][MAX_NODES+1],
```

```
              int XBs[MAX_LEN_SEQ+1], int XEs[MAX_LEN_SEQ+1], int
XCs[MAX_LEN_SEQ+1])
{
      int i, j ;

      printf("------------------------------------------------------------
-------------------------------------------\n") ;
      printf("M\n") ;

      printf("     ") ;
      for (j = 0 ; j <= n_nodes ; j ++)
      {
            printf(" %8d", j) ;
      }
      printf("\n") ;

      for (i = 0 ; i <= len_seq ; i ++)
      {
            if (i == 0)
            {
                  printf("%3d -", i) ;
            }
            else
            {
                  printf("%3d %c", i, seq[i]) ;
            }

            for (j = 0 ; j <= n_nodes ; j ++)
            {
                  if (Ms[i][j] > -INFTY)
                  {
                        printf(" %8d", Ms[i][j]) ;
                  }
                  else
                  {
                        printf("   -INFTY") ;
                  }
            }
            printf("\n") ;
      }

      printf("------------------------------------------------------------
-------------------------------------------\n") ;
      printf("I\n") ;

      printf("     ") ;
      for (j = 0 ; j <= n_nodes ; j ++)
      {
            printf(" %8d", j) ;
      }
      printf("\n") ;

      for (i = 0 ; i <= len_seq ; i ++)
      {
            if (i == 0)
            {
                  printf("%3d -", i) ;
            }
            else
            {
                  printf("%3d %c", i, seq[i]) ;
```

```
            }

            for (j = 0 ; j < n_nodes ; j ++)
            {
                  if (Is[i][j] > -INFTY)
                  {
                        printf(" %8d", Is[i][j]) ;
                  }
                  else
                  {
                        printf("   -INFTY") ;
                  }
            }
            printf("\n") ;
      }

      printf("-------------------------------------------------------------
-----------------------------------------\n") ;
      printf("D\n") ;

      printf("     ") ;
      for (j = 0 ; j <= n_nodes ; j ++)
      {
            printf(" %8d", j) ;
      }
      printf("\n") ;

      for (i = 0 ; i <= len_seq ; i ++)
      {
            if (i == 0)
            {
                  printf("%3d -", i) ;
            }
            else
            {
                  printf("%3d %c", i, seq[i]) ;
            }

            for (j = 0 ; j <= n_nodes ; j ++)
            {
                  if (Ds[i][j] > -INFTY)
                  {
                        printf(" %8d", Ds[i][j]) ;
                  }
                  else
                  {
                        printf("   -INFTY") ;
                  }
            }
            printf("\n") ;
      }

      printf("-------------------------------------------------------------
-----------------------------------------\n") ;
      printf("XB, XE, XC\n") ;

      printf("            XB       XE       XC\n") ;

      for (i = 0 ; i <= len_seq ; i ++)
      {
            if (i == 0)
```

```
            {
                  printf("%3d -", i) ;
            }
            else
            {
                  printf("%3d %c", i, seq[i]) ;
            }

            if (XBs[i] > -INFTY)
            {
                  printf(" %8d", XBs[i]) ;
            }
            else
            {
                  printf("   -INFTY") ;
            }

            if (XEs[i] > -INFTY)
            {
                  printf(" %8d", XEs[i]) ;
            }
            else
            {
                  printf("   -INFTY") ;
            }

            if (XCs[i] > -INFTY)
            {
                  printf(" %8d", XCs[i]) ;
            }
            else
            {
                  printf("   -INFTY") ;
            }

            printf("\n") ;
      }
} /* print_dp */

/*-------------------------------------------------------------------------*/

void print_dp_path(int Mp[MAX_LEN_SEQ+1][MAX_NODES+1], int
Ip[MAX_LEN_SEQ+1][MAX_NODES+1], int Dp[MAX_LEN_SEQ+1][MAX_NODES+1])
{
      int i, j ;

      printf("-------------------------------------------------------------
-----------------------------------------\n") ;
      printf("M/I/D path\n\n") ;

      printf("         ") ;
      for (j = 0 ; j <= n_nodes ; j ++)
      {
            printf(" %4d", j) ;
      }
      printf("\n") ;

      for (i = lin_ini ; i <= lin_fin ; i ++)
      {
            if (i == 0)
            {
```

```
                printf("%3d    - ", i) ;
            }
            else
            {
                printf("%3d    %c ", i, seq[i]) ;
            }

            for (j = 0 ; j <= n_nodes ; j ++)
            {
                printf("  %d/%d/%d", Mp[i][j], Ip[i][j], Dp[i][j]) ;
            }

            printf("\n") ;
        }
} /* print_dp_path */

/*-----------------------------------------------------------------------*/

void print_trace(int len_tr, t_trc tr[2*MAX_LEN_SEQ])
{
      int t, i, j ;
      char state, last_state ;

      //printf("-------------------------------------------------------------
-------------------------------------------\n") ;
      //printf("Trace: (Symbol/State)\n\n") ;

      state = ' ' ;

      for (t = 0 ; t < len_tr ; t ++)
      {
            last_state = state ;
            state      = tr[t].state ;
            i          = tr[t].i ;
            j          = tr[t].j ;

            switch (state)
            {
                  case 'M' :
                  case 'I' :
                        printf("(%c/%c%d) ", seq[i], state, j) ;
                  break ;
                  case 'D' :
                        printf("(-/%c%d) ", state, j) ;
                  break ;
                  case 'E' :
                        printf("(-/%c) ", state) ;
                  break ;
                  case 'C' :
                        if (last_state != 'C')
                        {
                              printf("(-/%c) ", state) ;
                        }
                        else
                        {
                              printf("(%c/%c) ", seq[i], state) ;
                        }
                  break ;
                  case 'T' :
                        printf("(-/%c) ", state) ;
                  break ;
```

```
                  case 'B' :
                        printf("(-/%c) ", state) ;
                  break ;
                  case 'N' :
                        if (last_state != 'N')
                        {
                              printf("(-/%c) ", state) ;
                        }
                        else
                        {
                              printf("(%c/%c) ", seq[i], state) ;
                        }
                  break ;
                  case 'S' :
                        printf("(-/%c) ", state) ;
                  break ;
            }
      }
      printf("\n") ;

} /* print_trace */

/*--------------------------------------------------------------------*/

void print_divergence(void)
{
      int i, j ;

      printf("-------------------------------------------------------------
-----------------------------------------\n") ;
      printf("M LI/DI/DS\n\n") ;

      printf(" ") ;
      for (j = 0 ; j <= n_nodes ; j ++)
      {
            printf("  %11d", j) ;
      }
      printf("\n") ;

      for (i = 0 ; i <= len_seq ; i ++)
      {
            if (i == 0)
            {
                  printf("%3d -", i) ;
            }
            else
            {
                  printf("%3d %c", i, seq[i]) ;
            }

            for (j = 0 ; j <= n_nodes ; j ++)
            {
                  printf("  %3d/%3d/%3d", M_lin_ini[i][j],
M_div_inf[i][j], M_div_sup[i][j]) ;
            }
            printf("\n") ;
      }

      printf("-------------------------------------------------------------
-----------------------------------------\n") ;
      printf("I LI/DI/DS\n\n") ;
```

```
      printf(" ") ;
      for (j = 0 ; j <= n_nodes ; j ++)
      {
            printf("  %11d", j) ;
      }
      printf("\n") ;

      for (i = 0 ; i <= len_seq ; i ++)
      {
            if (i == 0)
            {
                  printf("%3d -", i) ;
            }
            else
            {
                  printf("%3d %c", i, seq[i]) ;
            }

            for (j = 0 ; j <= n_nodes ; j ++)
            {
                  printf("  %3d/%3d/%3d", I_lin_ini[i][j],
I_div_inf[i][j], I_div_sup[i][j]) ;
            }
            printf("\n") ;
      }

      printf("-------------------------------------------------------------
-------------------------------------------\n") ;
      printf("D LI/DI/DS\n\n") ;

      printf(" ") ;
      for (j = 0 ; j <= n_nodes ; j ++)
      {
            printf("  %11d", j) ;
      }
      printf("\n") ;

      for (i = 0 ; i <= len_seq ; i ++)
      {
            if (i == 0)
            {
                  printf("%3d -", i) ;
            }
            else
            {
                  printf("%3d %c", i, seq[i]) ;
            }

            for (j = 0 ; j <= n_nodes ; j ++)
            {
                  printf("  %3d/%3d/%3d", D_lin_ini[i][j],
D_div_inf[i][j], D_div_sup[i][j]) ;
            }
            printf("\n") ;
      }

      printf("-------------------------------------------------------------
-------------------------------------------\n") ;
      printf("XE       LI/ DI/ DS ,   XC LI/ LF/ DI/ DS\n\n") ;
```

```
      for (i = 0 ; i <= len_seq ; i ++)
      {
            if (i == 0)
            {
                  printf("%3d -", i) ;
            }
            else
            {
                  printf("%3d %c", i, seq[i]) ;
            }

            printf("   %3d/%3d/%3d       %3d/%3d/%3d/%3d\n",
XE_lin_ini[i], XE_div_inf[i], XE_div_sup[i],
                  XC_lin_ini[i], XC_lin_fin[i], XC_div_inf[i],
XC_div_sup[i]) ;
      }

} /* print_divergence */

/*--------------------------------------------------------------------------*/
```

A.4 – READ.C

```
/*--------------------------------------------------------------------------*/

int symbol_index(char symbol)
{
      int index ;

      index = (strchr(alphabet, toupper(symbol)) - alphabet) ;

      return index ;
} /* symbol_index */

/*--------------------------------------------------------------------------*/

int read_line(FILE *file, char *line)
{
      char ch ;
      int len, end_line ;

      len = 0 ;
      end_line = 0 ;

      while ((! end_line) && (! feof(file)))
      {
            ch = fgetc(file) ;

            if (ch == '\n')
            {
                  end_line = 1 ;
            }
            else if (ch == EOF)
            {
                  end_line = 1 ;
            }
            else
            {
                  line[len] = (char) ch ;
```

```
                len ++ ;
            }
        }

        line[len] = '\0' ;

        return len ;

} /* read_line */

/*-------------------------------------------------------------------------*/

void read_hmm_file(char *filename)
{
        int len, i, j ;
        char line[MAX_LEN_LINE], *tk ;
        FILE *f_hmm ;

        /* Open HMM file (hmmn format) */
        f_hmm = fopen(filename, "rt") ;

        if (f_hmm == NULL)
        {
                printf("Error opening hmm file\n") ;
                exit(0) ;
        }

        len = read_line(f_hmm, line) ; /* Skip comment line */

        /* Read HMM number of nodes */
        len = read_line(f_hmm, line) ;
        tk = strtok(line, " ") ;
        n_nodes = atoi(tk) ;

        len = read_line(f_hmm, line) ; /* Skip comment line */
        len = read_line(f_hmm, line) ; /* Skip comment line */

        /* Initialize M emission probabilities of node 0 and I emission
probabilities of node 0 and node n_nodes */
        /* For each symbol of alphabet */
        for (i = 0 ; i < ALPHABET_SIZE ; i ++)
        {
                em_M[0][i]       = -INFTY ;
                em_I[0][i]       = -INFTY ;
                em_I[n_nodes][i] = -INFTY ;
        }

        /* Read M emission probabilities */
        /* For each node of HMM */
        for (j = 1 ; j <= n_nodes ; j ++)
        {
                len = read_line(f_hmm, line) ;
                tk = strtok(line, " ") ;

                /* For each symbol of alphabet */
                for (i = 0 ; i < ALPHABET_SIZE ; i ++)
                {
                        tk = strtok(NULL, " ") ;
                        em_M[j][i] = atoi(tk) ;
                }
        }
```

```
	len = read_line(f_hmm, line) ; /* Skip comment line */
	len = read_line(f_hmm, line) ; /* Skip comment line */

	/* Read I emission probabilities */
	/* For each node of HMM */
	for (j = 1 ; j < n_nodes ; j ++)
	{
		len = read_line(f_hmm, line) ;
		tk = strtok(line, " ") ;

		/* For each symbol of alphabet */
		for (i = 0 ; i < ALPHABET_SIZE ; i ++)
		{
			tk = strtok(NULL, " ") ;
			em_I[j][i] = atoi(tk) ;
		}
	}

	len = read_line(f_hmm, line) ; /* Skip comment line */
	len = read_line(f_hmm, line) ; /* Skip comment line */
	len = read_line(f_hmm, line) ; /* Skip comment line */

	/* Read transition probabilities */
	len = read_line(f_hmm, line) ;
	tk = strtok(line, " ") ;
	tr_N_B = atoi(tk) ;
	tk = strtok(NULL, " ") ;
	tr_N_N = atoi(tk) ;
	tk = strtok(NULL, " ") ;
	tr_E_C = atoi(tk) ;
	tk = strtok(NULL, " ") ;
	tk = strtok(NULL, " ") ;
	tr_C_T = atoi(tk) ;
	tk = strtok(NULL, " ") ;
	tr_C_C = atoi(tk) ;

	len = read_line(f_hmm, line) ; /* Skip comment line */

	/* Initialize transition probabilities of node 0 and node n_nodes
*/
	tr_M_M[0] = -INFTY ;
	tr_M_I[0] = -INFTY ;
	tr_M_D[0] = -INFTY ;
	tr_I_M[0] = -INFTY ;
	tr_I_I[0] = -INFTY ;
	tr_D_M[0] = -INFTY ;
	tr_D_D[0] = -INFTY ;
	tr_B_M[0] = -INFTY ;
	tr_M_E[0] = -INFTY ;

	tr_M_M[n_nodes] = -INFTY ;
	tr_M_I[n_nodes] = -INFTY ;
	tr_M_D[n_nodes] = -INFTY ;
	tr_I_M[n_nodes] = -INFTY ;
	tr_I_I[n_nodes] = -INFTY ;
	tr_D_M[n_nodes] = -INFTY ;
	tr_D_D[n_nodes] = -INFTY ;
	tr_B_M[n_nodes] = -INFTY ;
	tr_M_E[n_nodes] = -INFTY ;
```

```
	/* Read transition probabilities */
	/* For each node of HMM */
	for (j = 1 ; j < n_nodes ; j ++)
	{
		len = read_line(f_hmm, line) ;

		tk = strtok(line, " ") ;
		tr_M_M[j] = atoi(tk) ;
		tk = strtok(NULL, " ") ;
		tr_M_I[j] = atoi(tk) ;
		tk = strtok(NULL, " ") ;
		tr_M_D[j] = atoi(tk) ;
		tk = strtok(NULL, " ") ;
		tr_I_M[j] = atoi(tk) ;
		tk = strtok(NULL, " ") ;
		tr_I_I[j] = atoi(tk) ;
		tk = strtok(NULL, " ") ;
		tr_D_M[j] = atoi(tk) ;
		tk = strtok(NULL, " ") ;
		tr_D_D[j] = atoi(tk) ;
		tk = strtok(NULL, " ") ;
		tr_B_M[j] = atoi(tk) ;
		tk = strtok(NULL, " ") ;
		tr_M_E[j] = atoi(tk) ;
	}

	len = read_line(f_hmm, line) ;
	tk = strtok(line, " ") ;
	tk = strtok(NULL, " ") ;
	tk = strtok(NULL, " ") ;
	tk = strtok(NULL, " ") ;
	tk = strtok(NULL, " ") ;
	tk = strtok(NULL, " ") ;
	tk = strtok(NULL, " ") ;
	tk = strtok(NULL, " ") ;
	tr_B_M[n_nodes] = atoi(tk) ;
	tk = strtok(NULL, " ") ;
	tr_M_E[n_nodes] = atoi(tk) ;

	fclose(f_hmm) ;

} /* read_hmm_file */

/*--------------------------------------------------------------------------*/

FILE *open_sequence_file(char *filename)
{
	int len ;
	char line[MAX_LEN_LINE] ;
	FILE *f_seq ;

	/* Open sequences file (FASTA format) */
	f_seq = fopen(filename, "rt") ;

	if (f_seq == NULL)
	{
		printf("Error opening sequence file\n") ;
		exit(0) ;
	}

	/* Skip comment line of first sequence */
```

```
      len = read_line(f_seq, line) ;

      if ((len == 0) || (line[0] != '>'))
      {
            printf("Error reading sequence file\n") ;
            exit(0) ;
      }

      return f_seq ;

} /* open_sequence_file */

/*---------------------------------------------------------------------------*/

int read_sequence(FILE* f_seq, char *seq)
{
      int end_f_seq, end_seq ;
      int len ;
      char line[MAX_LEN_LINE] ;

      end_f_seq = 0 ;
      end_seq   = 0 ;

      strcpy(seq, " ") ;

      while (! (end_f_seq || end_seq))
      {
            len = read_line(f_seq, line) ;

            if (len == 0)
            {
                  end_f_seq = 1 ;
            }
            else if (line[0] == '>')
            {
                  end_seq = 1 ;
            }
            else if (feof(f_seq))
            {
                  end_f_seq = 1 ;
                  strcat(seq, line) ;
            }
            else
            {
                  strcat(seq, line) ;
            }
      }

      if (strlen(seq) <= 1)
      {
            printf("Error reading sequence file\n") ;
            exit(0) ;
      }

      return end_f_seq ;

} /* read_sequence */

/*---------------------------------------------------------------------------*/
```

B - PROGRAMAS DE EDIÇÃO E VIZUALISAÇÃO DE RESULTADOS

Vários programas foram codificados para obter uma melhor formatação dos dados e preparar eles para a sua posterior visualização. Foi escrito um programa em Java para a divisão do arquivo de HMMs (700MB) obtido da PFAM em arquivos individuais, a fim de poder alimentar eles a um *Script* feito em Linux que executa o programa *hmmsearch* para a obtenção dos tempos de execução do HMMER (Algoritmo de Viterbi e total). Foi modificado o programa *hmmsearch* do HMMER para obter o tempo de processamento para o Algoritmo de Viterbi, deixando fora os processos de Entrada/Saída de dados e o Algoritmo de *traceback*. Foi codificado um *Script* de Linux que faz a comparação de um arquivo de sequências com todos os HMMs obtidos do arquivo de HMMs de PFAM e captura o tempo de processamento do Algoritmo de Viterbi e o tempo total de execução do *hmmsearch.* Foi escrito o programa em Java para fazer a escolha randômica das sequências de teste da base de dados Uniprot.Sprot, para a entrada do *testbench* foi necessário tirar o texto dos arquivos para os HMMs e separar as probabilidades de transição e emissão em dois arquivos, o que foi feito também em Java. Finalmente foi codificado um programa em Matlab para visualizar os dados de desempenho obtidos para os tempos de execução do HMMER.

B.1 – DIVISÃO DO ARQUIVO DE HMMS PFAM

```
//divide the pfam HMM generated with HMMER
   public void PFAM_DIVIDE(){
      String PfamFile = "C:\\Documents and Settings\\Juan\\Escritorio\\viterbi\\Pfam-A.hmm";
      String destPath = "C:\\Documents and Settings\\Juan\\Escritorio\\viterbi\\PfamHMM\\";

      try{
         //Opening the file
         FileReader inputFileReader   = new FileReader(PfamFile);
```

```
        BufferedReader inputStream   = new BufferedReader(inputFileReader);

        //String buffer to read each line
        String inline = null;

        //File processing
        while((inline=inputStream.readLine())!=null){
           //initial lines
           String header=inline;
           String name=inputStream.readLine();
           String acc=inputStream.readLine();

           //HMM filename
           String fileName=acc.substring(3).trim();

           //opening the destination file
           FileOutputStream destination = new
FileOutputStream(destPath+fileName+".hmm");
           BufferedOutputStream BufoutStream = new BufferedOutputStream(destination);
           DataOutputStream outStream = new DataOutputStream(BufoutStream);

           //writing the initial 3 lines
           outStream.writeBytes(header);
           outStream.write(10);
           outStream.writeBytes(name);
           outStream.write(10);
           outStream.writeBytes(acc);
           outStream.write(10);

           //writing the rest of the HMM
           while(!(inline=inputStream.readLine()).equals("//")){
              outStream.writeBytes(inline);
              outStream.write(10);
           }
```

```
            outStream.writeBytes(inline);
            //closing all
            outStream.flush();
            outStream.close();
            destination.close();
          }
          //closing all
          inputFileReader.close();
          inputStream.close();
        }catch(IOException e){
          System.out.println(e.getMessage());
        }
      }
```

B.2 – *HMMSEARCH* MODIFICADO

```
/* Function: main_loop_serial()
 * Date:     SRE, Wed Sep 23 10:20:49 1998 [St. Louis]
 *
 * Purpose:  Search an HMM against a sequence database.
 *           main loop for the serial (non-PVM, non-threads)
 *           version.
 *
 *           In:   HMM and open sqfile, plus options
 *           Out:  histogram, global hits list, domain hits list, nseq.
 *
 * Args:     hmm        - the HMM to search with.
 *           sqfp       - open SQFILE for sequence database
 *           thresh     - score/evalue threshold info
 *           do_forward - TRUE to score using Forward()
 *           do_null2   - TRUE to use ad hoc null2 score correction
 *           do_xnu     - TRUE to apply XNU mask
 *           histogram  - RETURN: score histogram
 *           ghit       - RETURN: ranked global scores
 *           dhit       - RETURN: ranked domain scores
 *           ret_nseq   - RETURN: actual number of seqs searched
 *
 * Returns:  (void)
 */
static void
main_loop_serial(struct plan7_s *hmm, SQFILE *sqfp, struct threshold_s
*thresh, int do_forward,
             int do_null2, int do_xnu,
             struct histogram_s *histogram,
             struct tophit_s *ghit, struct tophit_s *dhit, int *ret_nseq)
{
  struct dpmatrix_s *mx;        /* DP matrix, growable
*/
```

```
  struct p7trace_s *tr;         /* traceback
*/
  char   *seq;                  /* target sequence
*/
  unsigned char   *dsq;         /* digitized target sequence
*/
  SQINFO sqinfo;        /* optional info for seq                    */
  float  sc;                 /* score of an HMM search
*/
  double pvalue;        /* pvalue of an HMM score                   */
  double evalue;        /* evalue of an HMM score                   */
  int    nseq;               /* number of sequences searched
*/

  /* Create a DP matrix; initially only two rows big, but growable;
   * we overalloc by 25 rows (L dimension) when we grow; not growable
   * in model dimension, since we know the hmm size
   */
  mx = CreatePlan7Matrix(1, hmm->M, 25, 0);

  nseq = 0;
  float totalTime=0;

  while (ReadSeq(sqfp, sqfp->format, &seq, &sqinfo))
    {
      /* Silently skip length 0 seqs.
       * What, you think this doesn't occur? Welcome to genomics,
       * young grasshopper.
       */
      if (sqinfo.len == 0) continue;

      nseq++;
      dsq = DigitizeSequence(seq, sqinfo.len);

      if (do_xnu && Alphabet_type == hmmAMINO) XNU(dsq, sqinfo.len);

      /* 1. Recover a trace by Viterbi.
       *    In extreme cases, the alignment may be literally impossible;
       *    in which case, the score comes out ridiculously small (but
not
       *    necessarily <= -INFTY, because we're not terribly careful
       *    about underflow issues), and tr will be returned as NULL.
       */
      if (P7ViterbiSpaceOK(sqinfo.len, hmm->M, mx)){
      //Esta funcion es la que hace el computo del algoritmo de Viterbi,
tengo que contar cuanto tiempo lleva su ejecucion para las
        //sequencias de prueba
      float time=clock();
      sc = P7Viterbi(dsq, sqinfo.len, hmm, mx, &tr);
      totalTime=totalTime+clock()-time;
      }
      else
      sc = P7SmallViterbi(dsq, sqinfo.len, hmm, mx, &tr);

      /* 2. If we're using Forward scores, calculate the
       *    whole sequence score; this overrides anything
       *    PostprocessSignificantHit() is going to do to the per-seq
score.
       */
      if (do_forward) {
      sc  = P7Forward(dsq, sqinfo.len, hmm, NULL);
```

```
      if (do_null2)   sc -= TraceScoreCorrection(hmm, tr, dsq);
      }

#if DEBUGLEVEL >= 2
      P7PrintTrace(stdout, tr, hmm, dsq);
#endif

      /* 2. Store score/pvalue for global alignment; will sort on score,
       *    which in hmmsearch is monotonic with E-value.
       *    Keep all domains in a significant sequence hit.
       *    We can only make a lower bound estimate of E-value since
       *    we don't know the final value of nseq yet, so the list
       *    of hits we keep in memory is >= the list we actually
       *    output.
       */
      pvalue = PValue(hmm, sc);
      evalue = thresh->Z ? (double) thresh->Z * pvalue : (double) nseq *
pvalue;
      if (sc >= thresh->globT && evalue <= thresh->globE)
      {
        sc = PostprocessSignificantHit(ghit, dhit,
                                tr, hmm, dsq, sqinfo.len,
                                sqinfo.name,
                                sqinfo.flags & SQINFO_ACC  ? sqinfo.acc  :
NULL,
                                sqinfo.flags & SQINFO_DESC ? sqinfo.desc :
NULL,
                                do_forward, sc,
                                do_null2,
                                thresh,
                                FALSE); /* FALSE-> not hmmpfam mode,
hmmsearch mode */
      }
      SQD_DPRINTF2(("AddToHistogram: %s\t%f\n", sqinfo.name, sc));
      AddToHistogram(histogram, sc);
      FreeSequence(seq, &sqinfo);
      P7FreeTrace(tr);
      free(dsq);
    }

  FreePlan7Matrix(mx);
  *ret_nseq = nseq;
  //Imprimiendo el tiempo gastado!!!
  printf("VitTime:%f",(totalTime/CLOCKS_PER_SEC));

  return;
}
```

B.3 - SCRIPT DE EXECUÇÃO DO HMMSEARCH PARA LINUX

```
#!/bin/bash
#echo "Enter the path to the HMMs"
HMMSPATH=/home/juan/Viterbi/secuencias_de_prueba/HMMtabla
#echo "Enter the sequence File"
SEQFILE=rand_seqs_712734.fasta

array=(`ls $HMMSPATH`)
len=${#array[*]}
```

```
echo "There are $len profileHMMs in the directory, processing..."

i=0
while [ $i -lt $len ]; do
        cat $HMMSPATH"/"${array[$i]} | grep ACC >> SalidaHmms_712734.txt
        cat $HMMSPATH"/"${array[$i]} | grep LENG >> SalidaHmms_712734.txt
      /usr/bin/time -a -o SalidaHmms_712734.txt -f "\nP_time:%U"
hmmsearch $HMMSPATH"/"${array[$i]} $SEQFILE >> SalidaHmms_712734.txt
      let i++
done
```

B.4 - ESCOLHA RANDOMICA DAS SEQUÊNCIAS DE TESTE

```
public static void main(String[] args) {
    String seqFile = "C:\\Documents and Settings\\Juan\\Escritorio\\viterbi\\uniprot_sprot.fasta";
    String destFile = "C:\\Documents and Settings\\Juan\\Escritorio\\viterbi\\rand_seqs.fasta";

    //Number of sequences to be chosed
    int numSeqs=2000;

    //Number of total elements of the chosen sequences
    long residues=0;

    //Auxiliar Variables
    int cuenta=0;
    int[] randSeqs;
    Random generator = new Random();

    //Read the sequence file
    try{
      //Random index array
      randSeqs = new int[numSeqs];

      //Opening the file
      FileReader inputFileReader  = new FileReader(seqFile);
      BufferedReader inputStream  = new BufferedReader(inputFileReader);
```

```
//String buffer to read each line
String inline = null;

//File processing
while((inline=inputStream.readLine())!=null){
   if(inline.contains(">")){
      cuenta++;
   }
}

System.out.println("Total de secuencias:"+cuenta);

//closing the seq file
inputFileReader.close();
inputStream.close();

//generating the random sequences to be picked from the file
randSeqs[0]=generator.nextInt(cuenta);

for(int i=1;i<numSeqs;i++){
   randSeqs[i]=generator.nextInt(cuenta);
   for(int j=0;j<i;j++){
      //System.out.println(i+"   "+j+"   "+randSeqs[j]+"   "+temp);
      if(randSeqs[j]==randSeqs[i]){
         i--;
      }
   }
}

//Sorting the generated random numbers
for(int i=0;i<numSeqs;i++){
   for(int j=0;j<numSeqs;j++){
      if(randSeqs[j]>randSeqs[i]){
```

```
            int aux=randSeqs[i];
            randSeqs[i]=randSeqs[j];
            randSeqs[j]=aux;
        }
    }
}

//getting the chosen sequences

//Opening the sequence file
FileReader inputFileReader2   = new FileReader(seqFile);
BufferedReader inputStream2   = new BufferedReader(inputFileReader2);
inline=null;

//opening the destination file
FileOutputStream destination = new FileOutputStream(destFile);
BufferedOutputStream BufoutStream = new BufferedOutputStream(destination);
DataOutputStream outStream = new DataOutputStream(BufoutStream);

//Auxiliar variable to keep track of how many seqs have passed
int indexSeq=-1;
//Auxiliar Variable to keep track of index of the random number array
int indexArr=0;
boolean obtain=false;

while((inline=inputStream2.readLine())!=null){

    if(inline.contains(">")){
        indexSeq++;
        //System.out.println(indexSeq+"   "+randSeqs[indexArr]+"   "+indexArr);
        if(indexSeq==randSeqs[indexArr]){
            outStream.writeBytes(inline);
            outStream.write(10);
            obtain=true;
```

```
                if(indexArr<numSeqs-1)
                    indexArr++;
                }else{
                    obtain=false;
                }
            }else{
                if(obtain==true){
                    residues=residues+inline.length();
                    outStream.writeBytes(inline);
                    outStream.write(10);
                }
            }
        }

        //closing all
        inputFileReader2.close();
        inputStream2.close();
        outStream.flush();
        outStream.close();
        destination.close();

        System.out.println("total de residuos:"+residues);

    }catch(IOException e){
        System.out.println(e.getMessage());
    }
}
```

B.5 – VISUALIZAÇÃO DO DESEMPENHO DO HMMER EM MATLAB

```
function final=graph_times(lengths,viterbi,proces,seq_elem)

index=1;

for i=0:max(lengths)
```

```
    found=find(lengths==i);
    totVit=0;
    totProc=0;
    if ~isempty(found)
        for j=1:length(found)
            totVit=totVit+viterbi(found(j));
            totProc=totProc+proces(found(j));
        end
        final(index,1)=i;
        final(index,2)=[totVit/length(found)];
        final(index,3)=[totProc/length(found)];
        index=index+1;
    end
end

%final(:,1)=final(:,1)*seq_elem;

diff=final(:,3)-final(:,2);

figure(1);
plot(final(:,1),final(:,2));
hold;
reg=polyfit(final(:,1),final(:,2),2);
aprox=polyval(reg,final(:,1));
plot(final(:,1),aprox,'r');
plot(final(:,1),final(:,3),'g');
plot(final(:,1),diff,'k');
```

B.6 – DIVISÃO DE ARQUIVOS .HMM Y .SEQ PARA O TESTBENCH

```
public class Main{
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        String rawHMM = null;
        String parsedHMM = null;
        String rawSeqFile = null;
        int PEnumber =0;
        boolean passed1 = false;
        boolean passed2 = false;
        boolean passed3 = false;
        boolean passed4 = false;

        //rawHMM =
"D:\\MAESTRIA\\Arquiteituras_VLSI\\Proyectos_quartus\\viterbi_divergencia
s_7_2\\modelsim\\PF00465.hmm";

        for(int i=0;i<args.length;i++){
            if(args[i].equals("--source")){
                rawHMM = args[i+1];
                passed1 = true;
            }else if(args[i].equals("--destination")){
                parsedHMM = args[i+1];
                passed2=true;
            }else if(args[i].equals("--help")){
```

```
                System.out.println("Usage: java Main.class --source <HMM
2.3.2 generated HMM file> --sequence <FASTA format sequence file> --
destination <destination path> --PEnum <number of PEs>");
            }else if(args[i].equals("--sequence")){
                rawSeqFile=args[i+1];
                passed3=true;
            }else if(args[i].equals("--PEnum")){
                PEnumber=Integer.valueOf(args[i+1]);
                passed4=true;
            }
        }

        //Argument error checking
        if(!passed1 | !passed2 | !passed3 | !passed4){
            System.out.println("You must provide the path for the un-
parsed HMM, the un-parsed sequence file AND the destination directory");
            System.out.println("Usage: java Main.class --source <HMM
2.3.2 generated HMM file> --sequence <FASTA format sequence file> --
destination <destination path> --PEnum <number of PEs>");
            System.exit(0);
        }

        System.out.println("HMM to parse: "+rawHMM);
        System.out.println("Input Sequence to parse: "+rawSeqFile);
        System.out.println("VHDL code PE number: "+PEnumber);
        System.out.println("Destination path: "+parsedHMM);

        //Reading in the HMM and parsing the file
        HMM readHMM = HMM.HMM_read(rawHMM);
        readHMM.HMMWriteToFile(parsedHMM,PEnumber);
        readHMM.seqWriteToFile(parsedHMM,rawSeqFile);

//readHMM.HMMWriteToFile("D:\\MAESTRIA\\Arquiteituras_VLSI\\Proyectos_qua
rtus\\viterbi_divergencias_7_2\\modelsim",100);

//readHMM.seqWriteToFile("D:\\MAESTRIA\\Arquiteituras_VLSI\\Proyectos_qua
rtus\\viterbi_divergencias_7_2\\modelsim",
"D:\\MAESTRIA\\Arquiteituras_VLSI\\Proyectos_quartus\\viterbi_divergencia
s_7_2\\modelsim\\PF01761_full.txt");
    }
}

public class HMM {
    private int emM[][]=null;
    private int emI[][]=null;
    private int trM_M[]=null;
    private int trM_I[]=null;
    private int trM_D[]=null;
    private int trI_M[]=null;
    private int trI_I[]=null;
    private int trD_M[]=null;
    private int trD_D[]=null;
    private int trB_M[]=null;
    private int trM_E[]=null;
    private int trN_N,trN_B,trC_C,trE_C;
    private int length;
    private int passes;

    //constructor for an empty HMM, must have the two parameters in order
to work
    HMM(int alphabet_size,int hmm_length){
```

```
        emM = new int[hmm_length][alphabet_size];
        emI = new int[hmm_length][alphabet_size];
        trM_M = new int[hmm_length];
        trM_I = new int[hmm_length];
        trM_D = new int[hmm_length];
        trI_M = new int[hmm_length];
        trI_I = new int[hmm_length];
        trD_M = new int[hmm_length];
        trD_D = new int[hmm_length];
        trB_M = new int[hmm_length];
        trM_E = new int[hmm_length];
        length=hmm_length;
    }

   //read the given HMM from the file and return a new HMM filled
   public static HMM HMM_read(String sourcePath){
       //Creating an empty HMM to read into
       HMM readHMM = null;
       //read parameters from file
       int length = 0;

       try{
            //Opening the file
            FileReader inputFileReader   = new FileReader(sourcePath);
            BufferedReader inputStream   = new
BufferedReader(inputFileReader);

            //String buffer to read each line
            String inline = null;

            //Parsing process (Specific to HMMER 2.3.2 generated HMMs)
            //first line (HMM version checking)
            if((inline=inputStream.readLine())!=null &
inline.contains("HMMER2.0  [2.3.2]")){
                //next 21 lines, mostly junk, but also especial
probabilities and model length
                for(int i=0;i<21;i++){
                    inline=inputStream.readLine();
                    if(inline.contains("LENG")){ //length
                        length =
Integer.parseInt(inline.substring(4).trim());
                        //Creating the empty HMM with the length
parameter
                        readHMM = new HMM(20,length);
                    }
                    if(inline.contains("XT")){  //Special transitions
                        inline=inline.substring(3).trim();
                        String[] tempTrans = new String[8];
                        for(int j=0;j<7;j++){

tempTrans[j]=inline.substring(0,inline.indexOf(" ")).trim();
                            inline = inline.substring(inline.indexOf("
")).trim();
                        }

readHMM.setSpecialTransitions(Integer.parseInt(tempTrans[1]),Integer.pars
eInt(tempTrans[0]),Integer.parseInt(tempTrans[5]),Integer.parseInt(tempTr
ans[2]));
                    }
                }
```

```
                //now we read as-is the transition and emision
probabilities into the created HMM to fill in the model
                for(int i=0;i<length;i++){
                    String temp;
                    //M state emission probabilities
                    inline=inputStream.readLine().trim();
                    inline=inline.substring(inline.indexOf("
"),inline.lastIndexOf(" ")).trim();
                    //20 emission probs
                    for(int j=0;j<19;j++){
                        temp = inline.substring(0,inline.indexOf("
")).trim();
                        if(!temp.equals("*")){
                            readHMM.emM[i][j]=Integer.parseInt(temp);
                        }else{
                            readHMM.emM[i][j]=-32768;
                        }
                        inline = inline.substring(inline.indexOf("
")).trim();
                    }
                    if(!inline.equals("*")){
                        readHMM.emM[i][19]=Integer.parseInt(inline);
                    }else{
                        readHMM.emM[i][19]=-32768;
                    }

                    //I state emission probabilities
                    inline=inputStream.readLine().trim();
                    inline=inline.substring(inline.indexOf(" ")).trim();
                    //20 emission probs
                    for(int j=0;j<19;j++){
                        temp = inline.substring(0,inline.indexOf("
")).trim();
                        if(!temp.equals("*")){
                            readHMM.emI[i][j]=Integer.parseInt(temp);
                        }else{
                            readHMM.emI[i][j]=-32768;
                        }
                        inline = inline.substring(inline.indexOf("
")).trim();
                    }
                    if(!inline.equals("*")){
                        readHMM.emI[i][19]=Integer.parseInt(inline);
                    }else{
                        readHMM.emI[i][19]=-32768;
                    }

                    //transition probabilities
                    inline=inputStream.readLine().trim();
                    inline=inline.substring(inline.indexOf(" ")).trim();
                    //trM_M
                    temp = inline.substring(0,inline.indexOf("
")).trim();
                    if(!temp.equals("*")){
                        readHMM.trM_M[i]=Integer.parseInt(temp);
                    }else{
                        readHMM.trM_M[i]=-32768;
                    }
                    inline = inline.substring(inline.indexOf("
")).trim();
```

```
                    //trM_I
                    temp = inline.substring(0,inline.indexOf("
")).trim();
                    if(!temp.equals("*")){
                        readHMM.trM_I[i]=Integer.parseInt(temp);
                    }else{
                        readHMM.trM_I[i]=-32768;
                    }
                    inline = inline.substring(inline.indexOf("
")).trim();

                    //trM_D
                    temp = inline.substring(0,inline.indexOf("
")).trim();
                    if(!temp.equals("*")){
                        readHMM.trM_D[i]=Integer.parseInt(temp);
                    }else{
                        readHMM.trM_D[i]=-32768;
                    }
                    inline = inline.substring(inline.indexOf("
")).trim();

                    //trI_M
                    temp = inline.substring(0,inline.indexOf("
")).trim();
                    if(!temp.equals("*")){
                        readHMM.trI_M[i]=Integer.parseInt(temp);
                    }else{
                        readHMM.trI_M[i]=-32768;
                    }
                    inline = inline.substring(inline.indexOf("
")).trim();

                    //trI_I
                    temp = inline.substring(0,inline.indexOf("
")).trim();
                    if(!temp.equals("*")){
                        readHMM.trI_I[i]=Integer.parseInt(temp);
                    }else{
                        readHMM.trI_I[i]=-32768;
                    }
                    inline = inline.substring(inline.indexOf("
")).trim();

                    //trD_M
                    temp = inline.substring(0,inline.indexOf("
")).trim();
                    if(!temp.equals("*")){
                        readHMM.trD_M[i]=Integer.parseInt(temp);
                    }else{
                        readHMM.trD_M[i]=-32768;
                    }
                    inline = inline.substring(inline.indexOf("
")).trim();

                    //trD_D
                    temp = inline.substring(0,inline.indexOf("
")).trim();
                    if(!temp.equals("*")){
                        readHMM.trD_D[i]=Integer.parseInt(temp);
                    }else{
```

```
                        readHMM.trD_D[i]=-32768;
                    }
                    inline = inline.substring(inline.indexOf("
")).trim();

                    //trB_M
                    temp = inline.substring(0,inline.indexOf("
")).trim();
                    if(!temp.equals("*")){
                        readHMM.trB_M[i]=Integer.parseInt(temp);
                    }else{
                        readHMM.trB_M[i]=-32768;
                    }
                    inline = inline.substring(inline.indexOf("
")).trim();

                    //trM_E
                    if(!inline.equals("*")){
                        readHMM.trM_E[i]=Integer.parseInt(inline);
                    }else{
                        readHMM.trM_E[i]=-32768;
                    }
                }
            }else{
                readHMM=null;
            }
        }catch(IOException e){
            System.out.println("Error while reading the input
file"+e.getMessage());
        }
       //returning the read HMM
       return readHMM;
   }

   //Set the special transitions of the calling HMM
   public void setSpecialTransitions(int trN_N,int trN_B,int trC_C,int
trE_C){
        this.trN_N=trN_N;
        this.trN_B=trN_B;
        this.trC_C=trC_C;
        this.trE_C=trE_C;
   }

   //Take the read HMM and write the file according to the format
   public void HMMWriteToFile(String DestinationBaseName,int PE_number){
        int pass_number = this.length/PE_number;

        if(this.length > pass_number*PE_number){
            pass_number++;
        }

        this.passes = pass_number;

        try{
        //emission file writing
        FileOutputStream destination = new
FileOutputStream(DestinationBaseName+"\\emissions.hmm");
        BufferedOutputStream BufoutStream = new
BufferedOutputStream(destination);
        DataOutputStream outStream = new DataOutputStream(BufoutStream);
```

```
        for(int i=0;i<this.length;i++){
            //M emission writing
            for(int j=0;j<20;j++){
                outStream.writeBytes(String.valueOf(this.emM[i][j])+"
");
            }
            outStream.write('\n');
            //I emission writing
            for(int j=0;j<20;j++){
                outStream.writeBytes(String.valueOf(this.emI[i][j])+"
");
            }
            outStream.write('\n');
        }

        //Filling the rest of the PEs transition probabilities with 0s
        for(int i=this.length;i<pass_number*PE_number;i++){
            //M emission writing
            for(int j=0;j<20;j++){
                outStream.writeBytes(String.valueOf(0)+"    ");
            }
            outStream.write('\n');
            //I emission writing
            for(int j=0;j<20;j++){
                outStream.writeBytes(String.valueOf(0)+"    ");
            }
            if(i<(pass_number*PE_number)-1)
                outStream.write('\n');
        }
        //closing all
        outStream.flush();
        outStream.close();
        destination.close();

        //transition file writing (must take care for the index as the
VHDL testbench read the text file in order)
        FileOutputStream destination2 = new
FileOutputStream(DestinationBaseName+"\\transitions.hmm");
        BufferedOutputStream BufoutStream2 = new
BufferedOutputStream(destination2);
        DataOutputStream outStream2 = new
DataOutputStream(BufoutStream2);

        //first write special transitions
        outStream2.writeBytes(String.valueOf(this.trN_B)+"   ");
        outStream2.writeBytes(String.valueOf(this.trN_N)+"   ");
        outStream2.writeBytes(String.valueOf(this.trC_C)+"   ");
        outStream2.writeBytes(String.valueOf(this.trE_C)+"   ");
        outStream2.write('\n');

        for(int i=0;i<this.length;i++){
            //transitions writing
            if((i-1)>=0){
                outStream2.writeBytes(String.valueOf(this.trM_M[i-1])+"
");
            }else{
                outStream2.writeBytes(String.valueOf(-32768)+"    ");
            }
            outStream2.writeBytes(String.valueOf(this.trM_I[i])+"    ");
            if((i-1)>=0){
```

```
                outStream2.writeBytes(String.valueOf(this.trM_D[i-1])+"
");
             }else{
                outStream2.writeBytes(String.valueOf(-32768)+"    ");
            }
            if((i-1)>=0){
                outStream2.writeBytes(String.valueOf(this.trI_M[i-1])+"
");
             }else{
                outStream2.writeBytes(String.valueOf(-32768)+"    ");
            }
            outStream2.writeBytes(String.valueOf(this.trI_I[i])+"    ");
            if((i-1)>=0){
                outStream2.writeBytes(String.valueOf(this.trD_M[i-1])+"
");
             }else{
                outStream2.writeBytes(String.valueOf(-32768)+"    ");
            }
            if((i-1)>=0){
                outStream2.writeBytes(String.valueOf(this.trD_D[i-1])+"
");
             }else{
                outStream2.writeBytes(String.valueOf(-32768)+"    ");
            }
            outStream2.writeBytes(String.valueOf(this.trB_M[i])+"    ");
            outStream2.writeBytes(String.valueOf(this.trM_E[i])+"    ");
            outStream2.write('\n');
        }

        for(int i=this.length;i<pass_number*PE_number;i++){
             for(int j=0;j<9;j++){
                outStream2.writeBytes(String.valueOf(-32768)+"    ");
             }
            /*outStream2.writeBytes(String.valueOf(0)+"    ");
//M->M
            outStream2.writeBytes(String.valueOf(-32768)+"    ");   //M-
>I
            outStream2.writeBytes(String.valueOf(-32768)+"    ");   //M-
>D
            outStream2.writeBytes(String.valueOf(-32768)+"    ");   //I-
>M
            outStream2.writeBytes(String.valueOf(0)+"    ");        //I-
>I
            outStream2.writeBytes(String.valueOf(-32768)+"    ");   //D-
>M
            outStream2.writeBytes(String.valueOf(0)+"    ");        //D-
>D
            outStream2.writeBytes(String.valueOf(-32768)+"    ");   //B-
>M
            outStream2.writeBytes(String.valueOf(-32768)+"    ");   //M-
>E*/

            if(i<(pass_number*PE_number)-1)
                outStream2.write('\n');
        }
        //closing all
        outStream2.flush();
        outStream2.close();
        destination2.close();

        }catch(IOException e){
```

```
            System.out.println("Error while writing files
"+e.getMessage());
        }
   }

   //Take the raw sequence and adapt it to the VHDL testbench required
format
   public void seqWriteToFile(String DestinationBaseName,String
rawSeqFile){
       try{
           //Opening the file
           FileReader inputFileReader   = new FileReader(rawSeqFile);
           BufferedReader inputStream   = new
BufferedReader(inputFileReader);

           //String buffer to read each line
           String inline = null;

           //Sequence file writing variables
           FileOutputStream destination = new
FileOutputStream(DestinationBaseName+"\\sequence.seq");
           BufferedOutputStream BufoutStream = new
BufferedOutputStream(destination);
           DataOutputStream outStream = new
DataOutputStream(BufoutStream);

           //auxiliar flag
           String seqString = "";

           while((inline=inputStream.readLine())!=null){
                if(inline.contains(">")){
                    if(!seqString.equals("")){
                        if(this.passes==1){
                            outStream.writeBytes(seqString+"@\n");
                            outStream.flush();
                        }else{
                            for(int i=0;i<this.passes-1;i++){
                                outStream.writeBytes(seqString+"*\n");
                                outStream.flush();
                            }
                            outStream.writeBytes(seqString+"@\n");
                            outStream.flush();
                        }
                        seqString="";
                    }
                    outStream.writeBytes(inline+'\n');
                    outStream.flush();
                    //System.out.println(inline);
                }else{
                    inline = inline.replace("U","S");
                    inline = inline.replace("B","ND");
                    inline = inline.replace("Z","QE");
                    inline = inline.replace("X","ACDEFGHIKLMNPQRSTVWY");
                    seqString = seqString + inline;
                }
           }

           //Last sequence output
           if(!seqString.equals("")){
                if(this.passes==1){
                    outStream.writeBytes(seqString+"@\n");
```

```
                    outStream.flush();
                }else{
                    for(int i=0;i<this.passes-1;i++){
                        outStream.writeBytes(seqString+"*\n");
                        outStream.flush();
                    }
                    outStream.writeBytes(seqString+"@\n");
                    outStream.flush();
                }
                seqString="";
             }

            //Close all
            inputFileReader.close();
            inputStream.close();
            destination.close();
            BufoutStream.close();
            outStream.close();

        }catch(Exception e){
             System.out.println("Error while writing files
"+e.getMessage());
        }
    }
}
```

C - CÓDIGO VHDL DO PROJETO

O código VHDL do projeto esta dividido de maneira hierárquica em arquivos .vhd separados que contem um componente do sistema completo cada um. Existem dois arquivos especiais no código que são o arquivo de configuração do Arranjo Sistólico (config.vhd) e o arquivo pacote de VHDL que contém as definições de componentes e dos tipos de dados utilizados no desenvolvimento da arquitetura.

C.1 – SOMADOR SATURADO (saturated_adder.vhd)

```
-- synthesis library viterbi
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

LIBRARY viterbi;
USE viterbi.config.all;

entity saturatedAdder is
generic(
        WORD_SIZE :         integer :=W_SIZE
);
port(
        in1     :               in      std_logic_vector(WORD_SIZE-1 downto 0);
        in2     :               in      std_logic_vector(WORD_SIZE-1 downto 0);
        sum     :               out     std_logic_vector(WORD_SIZE-1 downto 0)
);
end;

architecture RTL of saturatedAdder is
        signal overflowPos  : std_logic;
        signal overflowNeg  : std_logic;
```

```
        signal sumS                 : std_logic_vector(WORD_SIZE-1 downto 0);
begin
--Adition
sumS<=in1+in2;

--overflow conditions
overflowPos<=(not(in1(WORD_SIZE-1)) and not(in2(WORD_SIZE-1)) and
sumS(WORD_SIZE-1));
overflowNeg<=(in1(WORD_SIZE-1) and in2(WORD_SIZE-1) and
not(sumS(WORD_SIZE-1)));

--Saturation implementation
sum(WORD_SIZE-1)<=sumS(WORD_SIZE-1) when overflowPos='0' and
overflowNeg='0' else
        '0' when overflowPos='1' else
        '1' when overflowNeg='1' else
        '0';

sum(WORD_SIZE-2 downto 0)<=sumS(WORD_SIZE-2 downto 0) when overflowPos='0'
and overflowNeg='0' else
        (others=>'1') when overflowPos='1' else
        (others=>'0') when overflowNeg='1' else
        (others=>'0');
end;
```

C.2 – MÍNIMO DE DUAS ENTRADAS (min2.vhd)

```
-- synthesis library viterbi
LIBRARY ieee;
USE ieee.std_logic_1164.all;

library viterbi;
use viterbi.VITERBI_PKG.comparador16bits;
use viterbi.config.all;

entity min2 is
generic(
      WORD_SIZE   :     integer     := W_SIZE
);
```

```
port(
      in1   :     in          std_logic_vector(WORD_SIZE-1 downto 0);
      in2   :     in          std_logic_vector(WORD_SIZE-1 downto 0);
      lesser:     out         std_logic;
      min   :     out         std_logic_vector(WORD_SIZE-1 downto 0)
);
end min2;

architecture RTL of min2 is

--input selection signal
signal selector:std_logic;

begin

min<=in1 when selector='1' else
     in2;

comp1:comparador16bits
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      a=>in1,
      b=>in2,
      greater=>selector
);
lesser<=selector;
end;
```

C.3 – MÁXIMO DE DUAS ENTRADAS (max2.vhd)

```
-- synthesis library viterbi
LIBRARY ieee;
USE ieee.std_logic_1164.all;

library viterbi;
use viterbi.VITERBI_PKG.comparador16bits;
use viterbi.config.all;

entity max2 is
generic(
      WORD_SIZE   :     integer     := W_SIZE
);
port(
      in1   :     in          std_logic_vector(WORD_SIZE-1 downto 0);
      in2   :     in          std_logic_vector(WORD_SIZE-1 downto 0);
      greater:    out         std_logic;
      max   :     out         std_logic_vector(WORD_SIZE-1 downto 0)
);
end;

architecture RTL of max2 is

--input selection signal
signal selector:std_logic;

begin
```

```
max<=in1 when selector='0' else
     in2;

comp1:comparador16bits
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      a=>in1,
      b=>in2,
      greater=>selector
);

greater<=selector;
end;
```

C.4 – MÁXIMO DE QUATRO ENTRADAS (max4.vhd)

```
-- synthesis library viterbi
LIBRARY ieee;
USE ieee.std_logic_1164.all;

library viterbi;
use viterbi.VITERBI_PKG.comparador16bits;
use viterbi.config.all;

entity max4 is
generic(
      WORD_SIZE   :     integer     := W_SIZE
);
port(
      in1   :     in          std_logic_vector(WORD_SIZE-1 downto 0);
      in2   :     in          std_logic_vector(WORD_SIZE-1 downto 0);
      in3   :     in          std_logic_vector(WORD_SIZE-1 downto 0);
      in4   :     in          std_logic_vector(WORD_SIZE-1 downto 0);
      greater:    out         std_logic_vector(1 downto 0);
      max   :     out         std_logic_vector(WORD_SIZE-1 downto 0)
);
end;

architecture RTL of max4 is

--input selection signal
signal greater1,greater2,greater3,greater4,greater5,greater6 : std_logic;

begin

--process(in1,in2)
--begin
--    if(in1>in2)then
--          max<=in1;
--          greater<='0';
--    else
--          max<=in2;
--          greater<='1';
--    end if;
--end process;

max<=in1 when greater1='0' and greater2='0' and greater3='0' else
```

```
     in2 when greater1='1' and greater4='0' and greater5='0' else
     in3 when greater2='1' and greater4='1' and greater6='0' else
     in4;

greater<="00" when greater1='0' and greater2='0' and greater3='0' else
       "01" when greater1='1' and greater4='0' and greater5='0' else
       "10" when greater2='1' and greater4='1' and greater6='0' else
       "11";

comp1:comparador16bits
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      a=>in1,
      b=>in2,
      greater=>greater1
);

comp2:comparador16bits
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      a=>in1,
      b=>in3,
      greater=>greater2
);

comp3:comparador16bits
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      a=>in1,
      b=>in4,
      greater=>greater3
);

comp4:comparador16bits
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      a=>in2,
      b=>in3,
      greater=>greater4
);

comp5:comparador16bits
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      a=>in2,
      b=>in4,
      greater=>greater5
);

comp6:comparador16bits
generic map(
```

```
      WORD_SIZE=>WORD_SIZE
)
port map(
      a=>in3,
      b=>in4,
      greater=>greater6
);

end;
```

C.5 – BANCO DE REGISTRADORES DE TRANSIÇÃO

```
----synthesis library viterbi
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library viterbi;
use viterbi.config.all;
use viterbi.VITERBI_PKG.transitionInputs;


--Transition probabilities register bank, as you can see from the code,
--the register bank has data inputs coming from the block ram storing the
--transitions for the entire HMM and is only capable of storing the
--transitions for one pass (to reduce area), so it must be loaded with
the
--probabilities every time we are switching passes.
--This loading process is not time consuming as it can be done in only 5
--clock cycles due to the dual port nature of the transition
--probabilities RAM storage.

entity trRegisters is
generic(
      WORD_SIZE   :     integer := W_SIZE
);
port(
      data_1:     in  std_logic_vector(WORD_SIZE-1  downto  0);    --Data
coming from dual port RAM
      data_2      :     in  std_logic_vector(WORD_SIZE-1  downto  0);   --
Data coming from dual port RAM
      regAddr     :     in std_logic_vector(3 downto 0);  --16 registers
maximum (actually we have only 10)
      clk         :     in std_logic;
      w_en  :     in std_logic;     --Register bank write enabler
      reset :     in std_logic;
      tr          :     out  transitionInputs--Transition  outputs  to  the
PE

);
end trRegisters;

architecture RTL of trRegisters is
      type  regType  is  array(9  downto  0)  of  std_logic_vector(WORD_SIZE-1
downto 0);
```

```
      signal registers : regType;
begin

process(clk,reset)
begin
if(reset='1')then
      registers(0)<=(others=>'0');
      registers(1)<=(others=>'0');
      registers(2)<=(others=>'0');
      registers(3)<=(others=>'0');
      registers(4)<=(others=>'0');
      registers(5)<=(others=>'0');
      registers(6)<=(others=>'0');
      registers(7)<=(others=>'0');
      registers(8)<=(others=>'0');
      registers(9)<=(others=>'0');
elsif(rising_edge(clk))then
      if(w_en='1')then
            registers(conv_integer(regAddr))<=data_1;
            registers(conv_integer(regAddr+1))<=data_2;
      end if;
end if;
end process;

tr.tr_Mj1_Mj<=registers(0);
tr.tr_Mj_Ij<=registers(1);
tr.tr_Mj1_Dj<=registers(2);
tr.tr_Ij1_Mj<=registers(3);
tr.tr_Ij_Ij<=registers(4);
tr.tr_Dj1_Mj<=registers(5);
tr.tr_Dj1_Dj<=registers(6);
tr.tr_B_Mj<=registers(7);
tr.tr_Mj_E<=registers(8);

end;
```

C.6 – MULTIPLEXADOR DE 2 PARA 1 (genericMux2_1.vhd)

```
-- synthesis library viterbi
library ieee;
use ieee.std_logic_1164.all;

library viterbi;
use viterbi.config.all;

--2-to-1 generic word size mux
entity genericMux2_1 is
generic(
      WORD_SIZE   :     integer := W_SIZE);
port(
      in1         :     in std_logic_vector(WORD_SIZE-1 downto 0);
      in2         :     in std_logic_vector(WORD_SIZE-1 downto 0);
      sel         :     in std_logic;
      outp  :     out std_logic_vector(WORD_SIZE-1 downto 0));
end;

architecture behavioral of genericMux2_1 is
begin
```

```
outp<=in1 when sel='0' else
        in2;

end behavioral;
```

C.7 – MEMORIA RAM GENERICA (MEMORY.vhd)

```
--synthesis library viterbi
library ieee;
use ieee.std_logic_1164.all;

library viterbi;
use viterbi.config.all;

entity memory is
      generic
      (
            DATA_WIDTH : natural := W_SIZE;
            ADDR_WIDTH : natural := ADDR_WIDTH
      );
      port
      (
            clk         : in std_logic;
            addr_a      : in natural range 0 to (2**ADDR_WIDTH) - 1;
            addr_b      : in natural range 0 to (2**ADDR_WIDTH) - 1;
            data_a      : in std_logic_vector((DATA_WIDTH-1) downto 0);
            data_b      : in std_logic_vector((DATA_WIDTH-1) downto 0);
            we_a  : in std_logic;
            we_b  : in std_logic;
            q_a         : out std_logic_vector((DATA_WIDTH -1) downto 0);
            q_b         : out std_logic_vector((DATA_WIDTH -1) downto 0)
      );

end memory;

architecture rtl of memory is
      -- Build a 2-D array type for the RAM
      subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
      type memory_t is array((2**ADDR_WIDTH) - 1 downto 0) of word_t;

      -- Declare the RAM signal.
      signal ram : memory_t:=(others=>(others=>'Z'));
begin
      -- Port A
      process(clk)
      begin
      if(rising_edge(clk)) then
            if(we_a = '1') then
                  ram(addr_a) <= data_a;

                  -- Read-during-write on the same port returns NEW data
                  q_a <= data_a;
            else
                  -- Read-during-write on the mixed port returns OLD data
                  q_a <= ram(addr_a);
            end if;
      end if;
      end process;
```

```
	-- Port B
	process(clk)
	begin
	if(rising_edge(clk)) then
		if(we_b = '1') then
			ram(addr_b) <= data_b;

			-- Read-during-write on the same port returns NEW data
			q_b <= data_b;
		else
			-- Read-during-write on the mixed port returns OLD data
			q_b <= ram(addr_b);
		end if;
	end if;
	end process;

end rtl;
```

C.8- COMPARADOR (comparador16bits.vhd)

```
-- synthesis library viterbi
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

library viterbi;
use viterbi.config.all;


--Generic two input comparator which will help in the implementation of
the MAX and MIN
--functions
ENTITY comparador16bits IS
	generic(
		WORD_SIZE   :     integer     := W_SIZE
	);
	port
	(
		a :  IN  STD_LOGIC_VECTOR(WORD_SIZE-1 downto 0);
		b :  IN  STD_LOGIC_VECTOR(WORD_SIZE-1 downto 0);
		greater :  OUT  STD_LOGIC
	);
END comparador16bits;

ARCHITECTURE bdf_type OF comparador16bits IS
BEGIN
process(a,b)
begin
	if(a>=b)then
		greater<='0';
	else
		greater<='1';
	end if;
end process;
END;
```

C.9 – MEMORIA DE EMISSÕES (emitMem.vhd)

```
-- synthesis library viterbi
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library viterbi;
use viterbi.VITERBI_PKG.all;
use viterbi.config.all;

--Emission memory storage, it is based on a dual port RAM memory with a
--controller to choose between the two provided addresses (loading
--procedure and execution).

--This memory recognizes two special control characters:
--*(Sin=30): This character indicates the beginning of a new pass, and
--increments the pass counter

--@(Sin=31): This character indicates the beginning of a new sequence
--and clears the pass counter

--The pass counter indicates the offset in which are located the
--emission memories, for amino acids, the offset is 2*AMINO_NUMBER,
--indicating that there are two emision probabilities for each amino acid
--one for the Match State and one for the Insert State
entity emitMem is
generic(
      WORD_SIZE : integer := W_SIZE;
      AMINO_NUMBER : integer := AMINO_NUMBER;
      EMIT_ADDR_WIDTH : integer := EMIT_ADDR_WIDTH
);
port(
      clk         : in std_logic;
      reset : in std_logic;
      --write port
      addr  : in natural range 0 to (2**EMIT_ADDR_WIDTH) - 1;
      data1 : in std_logic_vector((WORD_SIZE-1) downto 0);
      data2 : in std_logic_vector((WORD_SIZE-1) downto 0);
      we          : in std_logic;
      --read port
      Sin         : in natural range 0 to AMINO_NUMBER+11;
      emM         : out std_logic_vector((WORD_SIZE -1) downto 0);
      emI         : out std_logic_vector((WORD_SIZE -1) downto 0);
      --Sequence element propagation output
      Sout    : out natural range 0 to AMINO_NUMBER+11
);
end emitMem;

architecture behavioral of emitMem is
--Address mux to chose between the writing address input (addr) and the
reading address input (Sout)
signal addressS : natural range 0 to (2**EMIT_ADDR_WIDTH) - 1;
signal addressS2 : natural range 0 to (2**EMIT_ADDR_WIDTH) - 1;

signal pass    : std_logic_vector(W_SIZE-1 downto 0);

begin

--pass signal controller
```

```
process(clk,Sin,reset,we)
begin
if(reset='1' or we='1')then               --Reset or New HMM
  pass<=(others=>'0');
elsif(rising_edge(clk))then
  if(Sin=AMINO_NUMBER+10)then     --New pass
    pass<=pass+1;
  elsif(Sin=AMINO_NUMBER+11)then  --New sequence
    pass<=(others=>'0');
  else
    pass<=pass;
  end if;
end if;
end process;

--Signal to generate the second address for the memory, this address
correspond to the
--Insert state emision
addressS2<=addressS+AMINO_NUMBER;

emMem:memory
generic map(
      DATA_WIDTH =>WORD_SIZE,
      ADDR_WIDTH =>EMIT_ADDR_WIDTH
)
port map(
      clk=>clk,
      addr_a=>addressS,
      addr_b=>addressS2,
      data_a=>data1,
      data_b=>data2,
      we_a=>we,
      we_b=>we,
      q_a=>emM,
      q_b=>emI
);

--Address mux
addressS<=addr when we='1' else
             Sin+AMINO_NUMBER*2*conv_integer(pass) when we='0' else
             0;
--Synchronous write logic
writeProcess:process(clk,reset)
begin
if(reset='1')then
      Sout<=0;
elsif(rising_edge(clk))then
      Sout<=Sin;
end if;
end process;
end behavioral;
```

C.10 – MEMORIA DE TRANCISÕES (transMem.vhd)

```
--synthesis library viterbi
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

```
library viterbi;
use viterbi.config.all;
use viterbi.VITERBI_PKG.all;

entity TransMem is
      generic
      (
            DATA_WIDTH : natural := W_SIZE;
            ADDR_WIDTH : natural := ADDR_WIDTH
      );
      port
      (
            clk         : in std_logic;
            reset       : in std_logic;
            addr_bus    :in natural range 0 to (2**ADDR_WIDTH) - 1;
            data_a      : in std_logic_vector((DATA_WIDTH-1) downto 0);
            data_b      : in std_logic_vector((DATA_WIDTH-1) downto 0);
            we          : in std_logic;
            Sin         : in natural range 0 to AMINO_NUMBER+11;
            reg_we      : out std_logic;
            regAddr     : out std_logic_vector(3 downto 0);
            q_a         : out std_logic_vector((DATA_WIDTH -1) downto 0);
            q_b         : out std_logic_vector((DATA_WIDTH -1) downto 0)
      );

end TransMem;

architecture rtl of TransMem is
--Signals to control the addres inputs of the dual port block ram memory
  signal addr : natural range 0 to 2**ADDR_WIDTH - 1;
  signal addr_Aux : natural range 0 to 2**ADDR_WIDTH - 1;

--Pass number acumulator signal
  signal pass : std_logic_vector(W_SIZE-1 downto 0);
  --Address generator states and signals
  type addGenStates is (IDLE,LOADING_HMM,LOADING_PROB);
  signal state : addGenStates;
  signal addrCount : natural range 0 to 2**ADDR_WIDTH - 1:=0;
  signal regCount : natural range 0 to 2**ADDR_WIDTH - 1:=0;
begin

--pass signal controller and address generator
process(clk,Sin,reset,we)
begin
if(reset='1')then               --Reset
  pass<=(others=>'0');
  state<=IDLE;
  addrCount<=0;
  regCount<=0;
  reg_we<='0';
elsif(we='1')then
  pass<=(others=>'0');
  addrCount<=0;
  regCount<=0;
  state<=LOADING_HMM;
  reg_we<='0';
elsif(rising_edge(clk))then
  case(state)is
  when IDLE=>
    reg_we<='0';
```

```
    addrCount<=0;
        regCount<=0;
    if(Sin=AMINO_NUMBER+10)then     --New pass
      pass<=pass+1;
      state<=LOADING_HMM;
    elsif(Sin=AMINO_NUMBER+11)then  --New sequence
      if(pass/=0)then
        pass<=(others=>'0');
        state<=LOADING_HMM;
      else
        state<=IDLE;
        pass<=pass;
      end if;
    else
      reg_we<='0';
      addrCount<=0;
          regCount<=0;
      pass<=pass;
      state<=state;
    end if;
  when LOADING_HMM=>
        if(we='0')then
              reg_we<='1';
            state<=LOADING_PROB;
              addrCount<=addrCount+2;
              regCount<=0;
    else
              reg_we<='0';
              addrCount<=0;
              regCount<=0;
              state<=LOADING_HMM;
    end if;
    pass<=pass;
  when LOADING_PROB=>
    reg_we<='1';
        if(addrCount<8)then
              addrCount<=addrCount+2;
              regCount<=regCount+2;
              state<=state;
        else
              regCount<=regCount+2;
              addrCount<=0;
              state<=IDLE;
        end if;
        pass<=pass;
  when others=>
    reg_we<='0';
    state<=IDLE;
    pass<=(others=>'0');
        addrCount<=0;
        regCount<=0;
  end case;
end if;
end process;

--Muxing the addresses (one for the programming process and one for the
loading process)
addr<=addr_bus when we='1' else
      addrCount+10*conv_integer(pass) when we='0' else
      0;
```

```
--Transition register bank controls
regAddr<=conv_std_logic_vector(regCount,4);

--generate the other memory address
addr_Aux<=addr+1;

Mem_i:memory
generic map(
      DATA_WIDTH =>W_SIZE,
      ADDR_WIDTH =>ADDR_WIDTH
)
port map(
      clk=>clk,
      addr_a=>addr,
      addr_b=>addr_Aux,
      data_a=>data_a,
      data_b=>data_b,
      we_a=>we,
      we_b=>we,
      q_a=>q_a,
      q_b=>q_b
      );
end rtl;
```

C.11 – UNIDADE DE CÁLCULO DO VETOR B (xbCalc.vhd)

```
-- synthesis library viterbi
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY viterbi;
use viterbi.VITERBI_PKG.saturatedAdder;
use viterbi.config.all;

entity xbCalc is
generic(
      WORD_SIZE   :     integer     := W_SIZE
);
port(
      dataLoad    :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      trN_N       :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      load        :     in    std_logic;
      reset       :     in    std_logic;
      enable            :     in    std_logic;
      clk               :     in    std_logic;
      Sin             : in natural range 0 to AMINO_NUMBER+11;
      Bi                :     out   std_logic_vector(WORD_SIZE-1    downto
0)
);
end;

architecture RTL of xbCalc is

--Signals
signal BiS : std_logic_vector(WORD_SIZE-1 downto 0);
```

```
signal adderOut : std_logic_vector(WORD_SIZE-1 downto 0);

begin

--Flip-Flop Data with load and enable
process(clk,load,reset)
begin
if(reset='1')then
      BiS<=(others=>'0');
elsif(rising_edge(clk))then
      if(load='1' or Sin=AMINO_NUMBER+10 or Sin=AMINO_NUMBER+11)then
            BiS<=dataLoad;
      elsif(enable='1')then
            BiS<=adderOut;
      end if;
end if;
end process;

--Transition probabilities adder with saturation arithmetic
add1:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>BiS,
      in2=>trN_N,
      sum=>adderOut
);

--Output assignment
Bi<=BiS;

end;
```

C.12 – UNIDADE DE CÁLCULO DO VETOR C (xcCalc.vhd)

```
-- synthesis library viterbi
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY viterbi;
use viterbi.VITERBI_PKG.all;
use viterbi.config.all;

entity xcCalc is
generic(
      WORD_SIZE   :     integer     := W_SIZE
);
port(
      load        :     in    std_logic;
      reset       :     in    std_logic;
      enable            :     in    std_logic;
      clk               :     in    std_logic;
      dataLoad    :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      XEin        :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      trC_C       :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      trE_C       :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      i_in        :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      Sin         : in natural range 0 to AMINO_NUMBER+11;
```

```
      --Divergence Signals
      XEDIin            :     in    std_logic_vector(WORD_SIZE-1    downto
0);
      XEDSin            :     in    std_logic_vector(WORD_SIZE-1    downto
0);
      XELIin            :     in    std_logic_vector(WORD_SIZE-1    downto
0);
      --Output Signals
      XCDI        :     out   std_logic_vector(WORD_SIZE-1 downto 0);
      XCDS        :     out   std_logic_vector(WORD_SIZE-1 downto 0);
      XCLI        :     out   std_logic_vector(WORD_SIZE-1 downto 0);
      XCLF        :     out   std_logic_vector(WORD_SIZE-1 downto 0);
      Ci                :     out   std_logic_vector(WORD_SIZE-1    downto
0)
);
end;

architecture RTL of xcCalc is

--Signals
signal Cireg            : std_logic_vector(WORD_SIZE-1 downto 0);
signal adder_1_Out : std_logic_vector(WORD_SIZE-1 downto 0);
signal adder_2_Out : std_logic_vector(WORD_SIZE-1 downto 0);
signal maxXCOut   : std_logic_vector(WORD_SIZE-1 downto 0);

--signals to calculate the divergence
signal ctlXC      : std_logic;
signal XCDISig          : std_logic_vector(WORD_SIZE-1 downto 0);
signal XCDSSig          : std_logic_vector(WORD_SIZE-1 downto 0);
signal XCLISig          : std_logic_vector(WORD_SIZE-1 downto 0);
signal XCLFSig          : std_logic_vector(WORD_SIZE-1 downto 0);
signal ctlXC_reg  : std_logic;


--Intermediate divergence registers
signal XCDIi1_reg       : std_logic_vector(WORD_SIZE-1 downto 0);
signal XCDSi1_reg       : std_logic_vector(WORD_SIZE-1 downto 0);
signal XCLIi1_reg       : std_logic_vector(WORD_SIZE-1 downto 0);
signal XCLFi1_reg       : std_logic_vector(WORD_SIZE-1 downto 0);

begin

--Flip-Flop Data with load and enable
process(clk,load,reset)
begin
if(reset='1')then
      Cireg<=(others=>'0');
      XCDIi1_reg<=(others=>'0');
      XCDSi1_reg<=(others=>'0');
      XCLIi1_reg<=(others=>'0');
      XCLFi1_reg<=(others=>'0');
      ctlXC_reg<='0';
elsif(load='1')then
      Cireg<=dataLoad;
      XCDIi1_reg<=(others=>'0');   --!!!OJO ESTO HAY QUE PREGUNTARLO
      XCDSi1_reg<=(others=>'0');   --!!!OJO ESTO HAY QUE PREGUNTARLO
      XCLIi1_reg<=(others=>'0');
      XCLFi1_reg<=(others=>'0');
      ctlXC_reg<='0';
elsif(rising_edge(clk))then
      if(Sin=AMINO_NUMBER+10 or Sin=AMINO_NUMBER+11)then
```

```
        Cireg<=dataLoad;
        XCDIi1_reg<=(others=>'0');   --!!!OJO ESTO HAY QUE PREGUNTARLO
        XCDSi1_reg<=(others=>'0');   --!!!OJO ESTO HAY QUE PREGUNTARLO
        XCLIi1_reg<=(others=>'0');
        XCLFi1_reg<=(others=>'0');
        ctlXC_reg<='0';
      elsif(enable='1')then
            Cireg<=maxXCOut;
            XCDIi1_reg<=XCDISig;
            XCDSi1_reg<=XCDSSig;
            XCLIi1_reg<=XCLISig;
            XCLFi1_reg<=XCLFSig;
            ctlXC_reg<=ctlXC;
      else
        Cireg<=Cireg;
            XCDIi1_reg<=XCDIi1_reg;
            XCDSi1_reg<=XCDSi1_reg;
            XCLIi1_reg<=XCLIi1_reg;
            XCLFi1_reg<=XCLFi1_reg;
            ctlXC_reg<=ctlXC_reg;
      end if;
end if;
end process;

--Transition probabilities adder with saturation arithmetic
add1:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>Cireg,
      in2=>trC_C,
      sum=>adder_1_Out
);

--Transition probabilities adder with saturation arithmetic
add2:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>XEin,
      in2=>trE_C,
      sum=>adder_2_Out
);

--Maximum to obtain the XC output
maxXC:max2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>adder_1_Out,
      in2=>adder_2_Out,
      greater=>ctlXC,
      max=>maxXCOut
);

--Divergence signals
XCDISig <= XCDIi1_reg when ctlXC_reg='0'else
              XEDIin when ctlXC_reg='1'else
```

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

XCDSSig <= XCDSi1_reg when ctlXC_reg='0'else
               XEDSin when ctlXC_reg='1'else
               (others=>'0');

XCLISig <= XCLIi1_reg when ctlXC_reg='0'else
               XELIin when ctlXC_reg='1'else
               (others=>'0');

XCLFSig <= XCLFi1_reg when ctlXC_reg='0'else
               i_in when ctlXC_reg='1'else
               (others=>'0');

--Output assignment
Ci<=Cireg;
XCDI<=XCDIi1_reg;
XCDS<=XCDSi1_reg;
XCLI<=XCLIi1_reg;
XCLF<=XCLFi1_reg;

end;
```

C.13 – FIFOS DE RESULTADOS INTERMEDÍARIOS (fifo.vhd)

```
-- synthesis library viterbi
LIbrary IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

Library viterbi;
use viterbi.config.all;
use viterbi.VITERBI_PKG.memory;

--altera fifo, remove for generality
LIBRARY lpm;
USE lpm.lpm_components.all;

Entity fifo is
      generic(
            WORD_SIZE   :     integer := W_SIZE;
            FIFO_DEPTH  :     integer := FIFO_DEPTH;
            ADDR_WIDTH  :     integer := FIFO_ADDR_WIDTH);
      Port (
            Clk               :     in std_logic;
            Reset             :     in std_logic;
            WriteEnable       :     in std_logic;
            ReadEnable        :     in std_logic;
            DataIn            :     in        std_logic_vector(WORD_SIZE-1
downto 0);
            DataOut           :     out       std_logic_vector(WORD_SIZE-1
downto 0);
      FifoEmpty         :     buffer std_logic;
            FifoFull          :     buffer std_logic
```

```
        );
END fifo;

architecture fifo_altera of fifo is
begin

fifo:LPM_FIFO
generic map(
      LPM_WIDTH=>WORD_SIZE,
      LPM_WIDTHU=>ADDR_WIDTH,
      LPM_NUMWORDS=>FIFO_DEPTH
)
port map(
      DATA=>DataIn,
      CLOCK=>clk,
      WRREQ=>WriteEnable,
      RDREQ=>ReadEnable,
      ACLR=>reset,
  Q=>DataOut,
  FULL=>FifoFull,
  EMPTY=>FifoEmpty
);
end fifo_altera;
```

C.14 – PACOTE DE DEFINIÇÕES (viterbiPackage.vhd)

```
-- synthesis library viterbi
library ieee;
use ieee.std_logic_1164.all;

library viterbi;
use viterbi.config.all;

package VITERBI_PKG is

------------------------------------------------------------------------
------
-- DATA TYPES DEFINITION IN ORDER TO MAKE THE CODE MORE COMPACT AND
READABLE
------------------------------------------------------------------------
------

--Permanent Inputs to the INDIVIDUAL PEs
type PEInputs is record
      reset       :     std_logic;
      load        :     std_logic;
      dataLoad    :     std_logic_vector(W_SIZE-1 downto 0);
      clk               :     std_logic;
      enable            :     std_logic;
      loadStall   :     std_logic;    --this signal is wired to the write
enable of the transition register bank
      Sin               :   natural range 0 to AMINO_NUMBER+11;
      --Score calculator data
      Mi_j1       :     std_logic_vector(W_SIZE-1 downto 0);
      Ii_j1       :     std_logic_vector(W_SIZE-1 downto 0);
      Di_j1       :     std_logic_vector(W_SIZE-1 downto 0);
      XBi1_in           :     std_logic_vector(W_SIZE-1 downto 0);
```

```
      XEi_j1            :     std_logic_vector(W_SIZE-1 downto 0);
      --Divergence calculator data
      MDIi_j1           :     std_logic_vector(W_SIZE-1 downto 0);
      MDSi_j1           :     std_logic_vector(W_SIZE-1 downto 0);
      MLIi_j1           :     std_logic_vector(W_SIZE-1 downto 0);
      IDIi_j1           :     std_logic_vector(W_SIZE-1 downto 0);
      IDSi_j1           :     std_logic_vector(W_SIZE-1 downto 0);
      ILIi_j1           :     std_logic_vector(W_SIZE-1 downto 0);
      DDIi_j1           :     std_logic_vector(W_SIZE-1 downto 0);
      DDSi_j1           :     std_logic_vector(W_SIZE-1 downto 0);
      DLIi_j1           :     std_logic_vector(W_SIZE-1 downto 0);
      XEDIi_j1    :     std_logic_vector(W_SIZE-1 downto 0);
      XEDSi_j1    :     std_logic_vector(W_SIZE-1 downto 0);
      XELIi_j1    :     std_logic_vector(W_SIZE-1 downto 0);
      i_in        :     std_logic_vector(W_SIZE-1 downto 0);
      --pass         :   std_logic_vector(W_SIZE-1 downto 0);  generated
internally
end record;

type transitionInputs is record
      --Transition Inputs
      tr_Mj1_Mj         :     std_logic_vector(W_SIZE-1 downto 0);  --M
      tr_Ij1_Mj         :     std_logic_vector(W_SIZE-1 downto 0);
      tr_Dj1_Mj         :     std_logic_vector(W_SIZE-1 downto 0);
      tr_B_Mj                 :     std_logic_vector(W_SIZE-1 downto 0);
      tr_Mj_Ij          :     std_logic_vector(W_SIZE-1 downto 0);  --I
      tr_Ij_Ij          :     std_logic_vector(W_SIZE-1 downto 0);
      tr_Mj1_Dj         :     std_logic_vector(W_SIZE-1 downto 0);  --D
      tr_Dj1_Dj         :     std_logic_vector(W_SIZE-1 downto 0);
      tr_Mj_E                 :     std_logic_vector(W_SIZE-1  downto  0);
--E
end record;

type emisionInputs is record
      --Emision Inputs
      EmMj_Si                 :     std_logic_vector(W_SIZE-1 downto 0);
      EmIj_Si                 :     std_logic_vector(W_SIZE-1 downto 0);
end record;

type PEOutputs is record
      --PE score outputs
      Mi1_j       :     std_logic_vector(W_SIZE-1 downto 0);
      Ii1_j       :     std_logic_vector(W_SIZE-1 downto 0);
      Di1_j       :     std_logic_vector(W_SIZE-1 downto 0);
      XBi1        :     std_logic_vector(W_SIZE-1 downto 0);
      XEi1_j            :     std_logic_vector(W_SIZE-1 downto 0);
      --PE divergence outputs
      MDIi1_j           :     std_logic_vector(W_SIZE-1 downto 0);
      MDSi1_j           :     std_logic_vector(W_SIZE-1 downto 0);
      MLIi1_j           :     std_logic_vector(W_SIZE-1 downto 0);
      IDIi1_j           :     std_logic_vector(W_SIZE-1 downto 0);
      IDSi1_j           :     std_logic_vector(W_SIZE-1 downto 0);
      ILIi1_j           :     std_logic_vector(W_SIZE-1 downto 0);
      DDIi1_j           :     std_logic_vector(W_SIZE-1 downto 0);
      DDSi1_j           :     std_logic_vector(W_SIZE-1 downto 0);
      DLIi1_j           :     std_logic_vector(W_SIZE-1 downto 0);
      XEDIi1_j    :     std_logic_vector(W_SIZE-1 downto 0);
      XEDSi1_j    :     std_logic_vector(W_SIZE-1 downto 0);
      XELIi1_j    :     std_logic_vector(W_SIZE-1 downto 0);
      i_out       :     std_logic_vector(W_SIZE-1 downto 0);
end record;
```

```
--This types are especifically designed to automatically replicate the
transition inputs in
--the design of the array of PEs and to connect them with a generate
estatement!!!!
type PEArrayTransitionInputs is array(NUMBER_OF_PES-1 downto 0) of
transitionInputs;
type PEArrayEmisionInputs is array(NUMBER_OF_PES-1 downto 0) of
emisionInputs;
type RamData is array(NUMBER_OF_PES-1 downto 0) of
std_logic_vector(W_SIZE-1 downto 0);

--These are the inputs to the array of PEs, they are very similar to
those of the individual PEs
--and have the aditional inputs of the control unit and the transition
and emission inputs
type PEArrayDataInputs is record
      --data loading
      dataLoad    :     std_logic_vector(W_SIZE-1 downto 0);
      --Special transitions that have to be delivered to the XC and XB
modules
      --and that do not change between passes
      trN_N       :     std_logic_vector(W_SIZE-1 downto 0);
      trN_B       :     std_logic_vector(W_SIZE-1 downto 0);
      trC_C       :     std_logic_vector(W_SIZE-1 downto 0);
      trE_C       :     std_logic_vector(W_SIZE-1 downto 0);
      --Score calculator data
      Min               :     std_logic_vector(W_SIZE-1 downto 0);
      Iin               :     std_logic_vector(W_SIZE-1 downto 0);
      Din               :     std_logic_vector(W_SIZE-1 downto 0);
      Sin               :     natural range 0 to (2**EMIT_ADDR_WIDTH);
      XEin        :     std_logic_vector(W_SIZE-1 downto 0);
      --Divergence calculator data
      MDIin       :     std_logic_vector(W_SIZE-1 downto 0);
      MDSin       :     std_logic_vector(W_SIZE-1 downto 0);
      MLIin       :     std_logic_vector(W_SIZE-1 downto 0);
      IDIin       :     std_logic_vector(W_SIZE-1 downto 0);
      IDSin       :     std_logic_vector(W_SIZE-1 downto 0);
      ILIin       :     std_logic_vector(W_SIZE-1 downto 0);
      DDIin       :     std_logic_vector(W_SIZE-1 downto 0);
      DDSin       :     std_logic_vector(W_SIZE-1 downto 0);
      DLIin       :     std_logic_vector(W_SIZE-1 downto 0);
      XEDIin            :     std_logic_vector(W_SIZE-1 downto 0);
      XEDSin            :     std_logic_vector(W_SIZE-1 downto 0);
      XELIin            :     std_logic_vector(W_SIZE-1 downto 0);
      i_in        :     std_logic_vector(W_SIZE-1 downto 0);
end record;

type PEArrayControlInputs is record
      reset       :     std_logic;
      load        :     std_logic; --Load dataLoad to the individual PEs
      clk               :     std_logic;
      enable            :     std_logic;  --Enable the operation of the
individual PEs
      WriteEnable :     std_logic; --Enable signals for the Score Fifos
      ReadEnable  :     std_logic;
      DivWriteEnable :  std_logic;  --Enable signals for the Divergence
Fifos
      DivReadEnable  :  std_logic;
      --Transition/emision memories addresses,enables and data inputs
(implemented as block RAMs)
```

```
      we_trans    :     std_logic_vector(NUMBER_OF_PES-1 downto 0);
      addr_bus    :     natural range 0 to 2**ADDR_WIDTH - 1;
      data_a            :     std_logic_vector(W_SIZE-1 downto 0);
      data_b            :     std_logic_vector(W_SIZE-1 downto 0);
      --Emision memories write enable
      EmitEn            :   std_logic_vector(NUMBER_OF_PES-1 downto 0);
      --Selection mux control signal
      passMuxSel              :     std_logic;
end record;

type PEArrayDataOutputs is record
      --Mout              :     std_logic_vector(W_SIZE-1 downto 0);
      --Iout              :     std_logic_vector(W_SIZE-1 downto 0);
      --Dout              :     std_logic_vector(W_SIZE-1 downto 0);
      Sout        :     natural range 0 to AMINO_NUMBER+11;
      Iout        :     std_logic_vector(W_SIZE-1 downto 0);
      --XBout             :     std_logic_vector(W_SIZE-1 downto 0);
      --XEout             :     std_logic_vector(W_SIZE-1 downto 0);
      XCout       :     std_logic_vector(W_SIZE-1 downto 0);
      --Divergence calculator data
      --MDIout            :     std_logic_vector(W_SIZE-1 downto 0);
      --MDSout            :     std_logic_vector(W_SIZE-1 downto 0);
      --MLIout            :     std_logic_vector(W_SIZE-1 downto 0);
      --XEDIout           :     std_logic_vector(W_SIZE-1 downto 0);
      --XEDSout           :     std_logic_vector(W_SIZE-1 downto 0);
      --XELIout           :     std_logic_vector(W_SIZE-1 downto 0);
      --XC calculator outputs
      XCDIout           :     std_logic_vector(W_SIZE-1 downto 0);
      XCDSout           :     std_logic_vector(W_SIZE-1 downto 0);
      XCLIout           :     std_logic_vector(W_SIZE-1 downto 0);
      XCLFout           :     std_logic_vector(W_SIZE-1 downto 0);
end record;

type PEArrayControlOutputs is record
      FifoEmpty         :     std_logic;  -- Fifo->ControlUnit outputs
      FifoFull          :     std_logic;
      DivFifoEmpty      :     std_logic;  -- Fifo->ControlUnit outputs
      DivFifoFull       :     std_logic;
end record;

------------------------------------------------------------------------
------
-- ACTUAL HARDWARE COMPONENTS USED
------------------------------------------------------------------------
------

--2 to 1 multiplexor
component genericMux2_1 is
generic(
      WORD_SIZE   :     integer := W_SIZE);
port(
      in1         :     in std_logic_vector(WORD_SIZE-1 downto 0);
      in2         :     in std_logic_vector(WORD_SIZE-1 downto 0);
      sel         :     in std_logic;
      outp  :     out std_logic_vector(WORD_SIZE-1 downto 0));
end component;

--Comparator
component comparador16bits IS
generic(
      WORD_SIZE   :     integer     := W_SIZE);
```

```
port
(
      a :  IN  STD_LOGIC_VECTOR(WORD_SIZE-1 downto 0);
      b :  IN  STD_LOGIC_VECTOR(WORD_SIZE-1 downto 0);
      greater :  OUT  STD_LOGIC);
END component;

--Maximum between 4 numbers
component max4 is
generic(
      WORD_SIZE   :     integer     := W_SIZE
);
port(
      in1         :           in          std_logic_vector(WORD_SIZE-1
downto 0);
      in2         :           in          std_logic_vector(WORD_SIZE-1
downto 0);
      in3         :           in          std_logic_vector(WORD_SIZE-1
downto 0);
      in4         :           in          std_logic_vector(WORD_SIZE-1
downto 0);
      greater     :           out         std_logic_vector(1 downto 0);
      max         :           out         std_logic_vector(WORD_SIZE-1
downto 0)
);
end component;

--Maximum between 2 numbers
component max2 is
generic(
      WORD_SIZE   :     integer     := W_SIZE);
port(
      in1         :           in          std_logic_vector(WORD_SIZE-1
downto 0);
      in2         :           in          std_logic_vector(WORD_SIZE-1
downto 0);
      greater     :           out         std_logic;
      max         :           out         std_logic_vector(WORD_SIZE-1
downto 0));
end component;

--Minimum between 2 numbers
component min2 is
generic(
      WORD_SIZE   :     integer     := W_SIZE
);
port(
      in1         :           in          std_logic_vector(WORD_SIZE-1
downto 0);
      in2         :           in          std_logic_vector(WORD_SIZE-1
downto 0);
      lesser      :           out         std_logic;
      min         :           out         std_logic_vector(WORD_SIZE-1
downto 0)
);
end component;

--Saturates adition between 2 16-bit numbers
component saturatedAdder is
generic(
      WORD_SIZE   :     integer     := W_SIZE);
```

```
port(
      in1         :           in    std_logic_vector(WORD_SIZE-1    downto
0);
      in2         :           in    std_logic_vector(WORD_SIZE-1    downto
0);
      sum         :           out   std_logic_vector(WORD_SIZE-1    downto
0));
end component;

--Processing element for the calculus of the viterbi algorithm
component PE is
generic(
      BASE_J            :     integer :=0;
      WORD_SIZE   :     integer     := W_SIZE);
port(
      PEinp       :     in    PEInputs;
      PEtrans           :   in  transitionInputs;
      PEemit            :     in  emisionInputs;
      PEout       :     out PEOutputs);
end component;

--Fifo implementation
component fifo is
      generic(
            WORD_SIZE   :     integer := W_SIZE;
            FIFO_DEPTH  :     integer := FIFO_DEPTH;
            ADDR_WIDTH  :     integer := FIFO_ADDR_WIDTH);
      Port (
            Clk               :     in std_logic;
            Reset             :     in std_logic;
            WriteEnable       :     in std_logic;
            ReadEnable        :     in std_logic;
            DataIn            :     in        std_logic_vector(WORD_SIZE-1
downto 0);
            DataOut           :     out       std_logic_vector(WORD_SIZE-1
downto 0);
      FifoEmpty         :     out std_logic;
            FifoFull          :     out std_logic);
END component;

--XB SCORE GENERATOR
component xbCalc is
generic(
      WORD_SIZE   :     integer     := W_SIZE
);
port(
      dataLoad    :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      trN_N       :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      load        :     in    std_logic;
      reset       :     in    std_logic;
      enable            :     in    std_logic;
      clk               :     in    std_logic;
      Sin         : in natural range 0 to AMINO_NUMBER+11;
      Bi                :     out   std_logic_vector(WORD_SIZE-1    downto
0)
);
end component;

--XC SCORE GENERATOR
component xcCalc is
generic(
```

```
      WORD_SIZE   :     integer     := W_SIZE
);
port(
      load        :     in    std_logic;
      reset       :     in    std_logic;
      enable            :     in    std_logic;
      clk               :     in    std_logic;
      dataLoad    :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      XEin        :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      trC_C       :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      trE_C       :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      i_in        :     in    std_logic_vector(WORD_SIZE-1 downto 0);
      Sin         : in natural range 0 to AMINO_NUMBER+11;
      --Divergence Signals
      XEDIin            :     in    std_logic_vector(WORD_SIZE-1    downto
0);
      XEDSin            :     in    std_logic_vector(WORD_SIZE-1    downto
0);
      XELIin            :     in    std_logic_vector(WORD_SIZE-1    downto
0);
      --Output Signals
      XCDI        :     out   std_logic_vector(WORD_SIZE-1 downto 0);
      XCDS        :     out   std_logic_vector(WORD_SIZE-1 downto 0);
      XCLI        :     out   std_logic_vector(WORD_SIZE-1 downto 0);
      XCLF        :     out   std_logic_vector(WORD_SIZE-1 downto 0);
      Ci                :     out   std_logic_vector(WORD_SIZE-1    downto
0)
);
end component;

--DUAL PORT MEMORY GENERATOR
component memory is
      generic
      (
            DATA_WIDTH : natural := W_SIZE;
            ADDR_WIDTH : natural := 6
      );
      port(
            clk         : in std_logic;
            addr_a      : in natural range 0 to 2**ADDR_WIDTH - 1;
            addr_b      : in natural range 0 to 2**ADDR_WIDTH - 1;
            data_a      : in std_logic_vector((DATA_WIDTH-1) downto 0);
            data_b      : in std_logic_vector((DATA_WIDTH-1) downto 0);
            we_a  : in std_logic := '1';
            we_b  : in std_logic := '1';
            q_a         : out std_logic_vector((DATA_WIDTH -1) downto 0);
            q_b         : out std_logic_vector((DATA_WIDTH -1) downto 0)
      );
end component;

--TRANSITION MEMORY STORAGE
component TransMem is
      generic
      (
            DATA_WIDTH : natural := W_SIZE;
            ADDR_WIDTH : natural := ADDR_WIDTH
      );
      port
      (
            clk            : in std_logic;
            reset      : in std_logic;
```

```
            addr_bus: in natural range 0 to (2**ADDR_WIDTH) - 1;
            data_a       : in std_logic_vector((DATA_WIDTH-1) downto 0);
            data_b       : in std_logic_vector((DATA_WIDTH-1) downto 0);
            we         : in std_logic;
            Sin         : in natural range 0 to AMINO_NUMBER+11;
            reg_we  : out std_logic;
            regAddr : out std_logic_vector(3 downto 0);
            q_a            : out std_logic_vector((DATA_WIDTH -1) downto
0);
            q_b            : out std_logic_vector((DATA_WIDTH -1) downto
0)
      );

end component;

--MODEL REGISTER BANK
component trRegisters is
generic(
      WORD_SIZE   :     integer := W_SIZE
);
port(
      data_1      :     in std_logic_vector(WORD_SIZE-1 downto 0);
      data_2      :     in std_logic_vector(WORD_SIZE-1 downto 0);
      regAddr     :     in std_logic_vector(3 downto 0);   --16 registers
maximum (actually we have only 10)
      clk         :     in std_logic;
      w_en  :     in std_logic;
      reset :     in std_logic;
      tr          :     out transitionInputs
);
end component;

--READ MEMORY IMPLEMENTED TO STORE THE EMISION PROBABILITIES
--FOR THE M AND I STATE
component emitMem is
generic(
      WORD_SIZE : integer := W_SIZE;
      AMINO_NUMBER : integer := AMINO_NUMBER;
      EMIT_ADDR_WIDTH : integer := EMIT_ADDR_WIDTH
);
port(
      clk         : in std_logic;
      reset : in std_logic;
      --write port
      addr  : in natural range 0 to (2**EMIT_ADDR_WIDTH) - 1;
      data1 : in std_logic_vector((WORD_SIZE-1) downto 0);
      data2 : in std_logic_vector((WORD_SIZE-1) downto 0);
      we          : in std_logic;
      --read port
      Sin         : in natural range 0 to AMINO_NUMBER+11;
      emM         : out std_logic_vector((WORD_SIZE -1) downto 0);
      emI         : out std_logic_vector((WORD_SIZE -1) downto 0);
      --Sequence element propagation output
      Sout    : out natural range 0 to AMINO_NUMBER+11
);
end component;

------------------------------------------------------------------------
------
-- COMPLETE PE ARRAY
```

```
------------------------------------------------------------------------
------
component PEArray is
generic(
      WORD_SIZE         :     integer := W_SIZE;
      NUMBER_OF_PES     :     integer := NUMBER_OF_PES;
      FIFO_DEPTH        :     integer := FIFO_DEPTH;
      FIFO_ADDR_WIDTH : integer := FIFO_ADDR_WIDTH;
      ADDR_WIDTH        :     integer := ADDR_WIDTH;
      AMINO_NUMBER      :     integer := AMINO_NUMBER;
      EMIT_ADDR_WIDTH :       integer     := EMIT_ADDR_WIDTH);
port(
      PEArrayDataIn           :     in    PEArrayDataInputs;
      PEArrayControlIn  :     in    PEArrayControlInputs;
      PEArrayDataOut          :     out   PEArrayDataOutputs;
      PEArrayControlOut :     out   PEArrayControlOutputs
);
end component;

end package;
```

C.15 – ARQUIVO DE CONFIGURAÇÕES (config.vhd)

```
--CONFIGURATION PACKAGE FOR THE VITERBI ARCHITECTURE IMPLEMENTATION
--This file has the bus sizes, the number of PEs, the number of block
--and distributed RAM, and the other parameters that constitue the entire
--architecture

-- synthesis library viterbi
library ieee;
use ieee.std_logic_1164.all;

package config is

--Number of PEs in the array
CONSTANT NUMBER_OF_PES : integer :=85;
--Word Size for all the components
CONSTANT W_SIZE : integer :=16;
--Addres width for the transition memory (10 for 1024 words)
CONSTANT ADDR_WIDTH : integer := 8;
--Fifo depth
CONSTANT FIFO_DEPTH : integer := 8192;
CONSTANT FIFO_ADDR_WIDTH : integer := 13;
--Number of amino acids of which the protein is composed of
CONSTANT AMINO_NUMBER :       integer :=20;
--Emision probabilities memory depth
CONSTANT EMIT_ADDR_WIDTH : integer := 10;

end package;
```

C.16 – ELEMENTO DE PROCESSAMENTO (PE.vhd)

```
-- synthesis library viterbi
```

```
--------------------------------------------------------------------------
-- Processing element for the viterbi algorithm
--
-- Note: the transition order is as especified into the trRegisters
--assignment, not as in the transitionInputs type definition, when it
--comes to feed the block RAMs.
--------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE ieee.std_logic_arith.all;

library Viterbi;
use Viterbi.VITERBI_PKG.all;
USE viterbi.config.all;

entity PE is
generic(
      BASE_J            :     integer :=1;
      WORD_SIZE   :     integer :=W_SIZE
);
port(
      --Permanent Inputs to the PE
      PEinp       :     in    PEInputs;
      --Transition Inputs
      PEtrans           :   in  transitionInputs;
      --Emision Inputs
      PEemit            :     in  emisionInputs;
      --PE outputs
      PEout       :     out PEOutputs
);
end;

architecture RTL of PE is
--Maximum Component
--Mnemonics:
--maxM1=>   M:State
--                number: for the M state, we need to choose between 4
numbers so we have a maximum
--                            calculator tree.

--Saturated Adder component
--Mnemonics:
--adderMM_M=>     first M:original State
--                second M:destination State
--                    third M: Component score (M,I or D)

--PE registered signals - equivalent to data Flip-Flops
--Mnemonics:
--reg<State><index i><index j>
--Example: regMi1_j1 its the signal for the register output equivalent to
the M(i-1,j-1) component
--             of the calculation.
signal regMi1_j1  :     std_logic_vector(WORD_SIZE-1  downto  0);    --M(i-
1,j-1)
signal regIi1_j1  :     std_logic_vector(WORD_SIZE-1  downto  0);    --I(i-
1,j-1)
signal regDi1_j1  :     std_logic_vector(WORD_SIZE-1  downto  0);    --D(i-
1,j-1)
signal regMi1_j         :     std_logic_vector(WORD_SIZE-1 downto 0);  --
M(i-1,j)
```

```
signal regIi1_j         :     std_logic_vector(WORD_SIZE-1 downto 0);   --
I(i-1,j)
signal regDi1_j         :     std_logic_vector(WORD_SIZE-1 downto 0);   --
D(i-1,j)
signal regEi1_j         :     std_logic_vector(WORD_SIZE-1 downto 0);   --
E(i-1,j)
signal regXBi1          :     std_logic_vector(WORD_SIZE-1 downto 0);   --
XB(i)

--Divergence registers
--Divergence for the M state
signal regMDIi1_j1            :     std_logic_vector(W_SIZE-1 downto 0);
signal regMDIi1_j       :     std_logic_vector(W_SIZE-1 downto 0);
signal regMDSi1_j1            :     std_logic_vector(W_SIZE-1 downto 0);
signal regMDSi1_j       :     std_logic_vector(W_SIZE-1 downto 0);
signal regMLIi1_j1            :     std_logic_vector(W_SIZE-1 downto 0);
signal regMLIi1_j       :     std_logic_vector(W_SIZE-1 downto 0);
--Divergence for the I state
signal regIDIi1_j1            :     std_logic_vector(W_SIZE-1 downto 0);
signal regIDIi1_j       :     std_logic_vector(W_SIZE-1 downto 0);
signal regIDSi1_j1            :     std_logic_vector(W_SIZE-1 downto 0);
signal regIDSi1_j       :     std_logic_vector(W_SIZE-1 downto 0);
signal regILIi1_j1            :     std_logic_vector(W_SIZE-1 downto 0);
signal regILIi1_j       :     std_logic_vector(W_SIZE-1 downto 0);
--Divergence for the D state
signal regDDIi1_j1            :     std_logic_vector(W_SIZE-1 downto 0);
signal regDDIi1_j       :     std_logic_vector(W_SIZE-1 downto 0);
signal regDDSi1_j1            :     std_logic_vector(W_SIZE-1 downto 0);
signal regDDSi1_j       :     std_logic_vector(W_SIZE-1 downto 0);
signal regDLIi1_j1            :     std_logic_vector(W_SIZE-1 downto 0);
signal regDLIi1_j       :     std_logic_vector(W_SIZE-1 downto 0);

signal regi                   :     std_logic_vector(W_SIZE-1 downto 0);
signal regXEDIi1_j            :     std_logic_vector(W_SIZE-1 downto 0);
signal regXEDSi1_j            :     std_logic_vector(W_SIZE-1 downto 0);
signal regXELIi1_j            :     std_logic_vector(W_SIZE-1 downto 0);

--PE interconection signals
--Mnemonics:
--sig<State><index i>_<index j>
--sig<component>_<component signal>

--M state
signal sigAdderMM_M_out       :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigAdderIM_M_out       :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigAdderDM_M_out       :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigAdderBM_out         :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigAdderEmitM_out      :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigMaxM1_out                 :     std_logic_vector(WORD_SIZE-1
downto 0);
signal sigMaxM2_out                 :     std_logic_vector(WORD_SIZE-1
downto 0);
signal sigMaxM3_out                 :     std_logic_vector(WORD_SIZE-1
downto 0);

--I state
```

```
signal sigAdderMM_I_out      :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigAdderII_I_out      :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigAdderEmitI_out     :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigMaxI_out                 :     std_logic_vector(WORD_SIZE-1
downto 0);

--D state
signal sigAdderDD_D_out      :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigAdderMD_D_out      :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigMaxD_out                 :     std_logic_vector(WORD_SIZE-1
downto 0);

--E state
signal sigAdderME_E_out      :     std_logic_vector(WORD_SIZE-1    downto
0);
signal sigMaxE_out                 :     std_logic_vector(WORD_SIZE-1
downto 0);

--Divergence calculation signals
signal ctlM       :     std_logic_vector(1 downto 0);
signal g1,g2,g3 :       std_logic;    --greater signals from the M state
max units
signal ctlXE      :     std_logic;
signal ctlI       :     std_logic;
signal ctlD       :     std_logic;
signal ActualJ    :     std_logic_vector(WORD_SIZE-1   downto   0);     --
BASE_J*PASS_NUMBER_REGISTER (PRE-LOADED)
signal  i_minus_j  :  std_logic_vector(WORD_SIZE-1  downto  0);  --i_in  -
ActualJ

-------------------------------------------------------------------------
-------------------------------------------------------------------------
-- TEST !!!!!
-------------------------------------------------------------------------
-------------------------------------------------------------------------
signal ctlM_reg    :    std_logic_vector(1 downto 0);
signal ctlXE_reg  :     std_logic;
signal ctlI_reg    :    std_logic;
signal ctlD_reg    :    std_logic;


--Maximum MDI and XEDI signals signals
signal sigMax_MDI_1_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigMax_MDI_2_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigMDI_i_j : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigXEDI_i_j : std_logic_vector(WORD_SIZE-1 downto 0);

--Minimum MDS and XEDS signals Signals
signal sigMin_MDS_1_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigMin_MDS_2_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigMDS_i_j : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigXEDS_i_j : std_logic_vector(WORD_SIZE-1 downto 0);

--Maximum IDI signals
signal sigMax_IDI_1_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigMax_IDI_2_out : std_logic_vector(WORD_SIZE-1 downto 0);
```

```
signal sigIDI_i_j : std_logic_vector(WORD_SIZE-1 downto 0);

--Minimum IDS Signals
signal sigMin_IDS_1_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigMin_IDS_2_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigIDS_i_j : std_logic_vector(WORD_SIZE-1 downto 0);

--Maximum DDI signals
signal sigMax_DDI_1_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigMax_DDI_2_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigDDI_i_j : std_logic_vector(WORD_SIZE-1 downto 0);

--Minimum DDS signals
signal sigMin_DDS_1_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigMin_DDS_2_out : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigDDS_i_j : std_logic_vector(WORD_SIZE-1 downto 0);

--MLI,ILI, DLI and XELI signals
signal sigMLI_i_j : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigILI_i_j : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigDLI_i_j : std_logic_vector(WORD_SIZE-1 downto 0);
signal sigXELI_i_j : std_logic_vector(WORD_SIZE-1 downto 0);

--Pass number acumulator signal
signal pass : std_logic_vector(W_SIZE-1 downto 0);
signal passChange : std_logic;

begin
--Actual J PE number calculation
ActualJ                                                 <=                0-
conv_std_logic_vector((conv_integer(pass)*NUMBER_OF_PES+BASE_J),WORD_SIZE
);
i_minus_j <= PEinp.i_in+ActualJ;

--Component declaration and port binding
--MaxM1:max2 --M state
--generic map(
--    WORD_SIZE=>WORD_SIZE
--)
--port map(
--    in1=>sigAdderMM_M_out,
--    in2=>sigAdderIM_M_out,
--    max=>sigMaxM1_out,
--    greater=>g1
--);
--
--MaxM2:max2 --M state
--generic map(
--    WORD_SIZE=>WORD_SIZE
--)
--port map(
--    in1=>sigAdderDM_M_out,
--    in2=>sigAdderBM_out,
--    max=>sigMaxM2_out,
--    greater=>g2
--);
--
--MaxM3:max2 --M state
--generic map(
--    WORD_SIZE=>WORD_SIZE
--)
```

```
--port map(
--    in1=>sigMaxM1_out,
--    in2=>sigMaxM2_out,
--    max=>sigMaxM3_out,
--    greater=>g3
--);

--Divergence control from M state
--ctlM(1)<=g3;
--ctlM(0)<=(g2 and g3) or (g1 and not(g3));

MaxM:max4
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>sigAdderMM_M_out,
    in2=>sigAdderIM_M_out,
    in3=>sigAdderDM_M_out,
    in4=>sigAdderBM_out,
    greater=>ctlM,
    max=>sigMaxM3_out
);

MaxI:max2 --I state
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>sigAdderMM_I_out,
      in2=>sigAdderII_I_out,
      max=>sigMaxI_out,
      greater=>ctlI
);

maxD:max2  --D state
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>sigAdderMD_D_out,
      in2=>sigAdderDD_D_out,
      max=>sigMaxD_out,
      greater=>ctlD
);

maxE:max2  --E state
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEinp.XEi_j1,
      in2=>sigAdderME_E_out,
      max=>sigMaxE_out,
      greater=>ctlXE
);

AdderME_E:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
```

```
port map(
      in1=>sigAdderEmitM_out,
      in2=>PEtrans.tr_Mj_E,
      sum=>sigAdderME_E_out
);

AdderMM_M:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>regMi1_j1,
      in2=>PEtrans.tr_Mj1_Mj,
      sum=>sigAdderMM_M_out
);

AdderIM_M:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>regIi1_j1,
      in2=>PEtrans.tr_Ij1_Mj,
      sum=>sigAdderIM_M_out
);

AdderDM_M:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>regDi1_j1,
      in2=>PEtrans.tr_Dj1_Mj,
      sum=>sigAdderDM_M_out
);

AdderBM_M:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEinp.XBi1_in,
      in2=>PEtrans.tr_B_Mj,
      sum=>sigAdderBM_out
);

AdderMM_I:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>regMi1_j,
      in2=>PEtrans.tr_Mj_Ij,
      sum=>sigAdderMM_I_out
);

AdderII_I:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
```

```
      in1=>regIi1_j,
      in2=>PEtrans.tr_Ij_Ij,
      sum=>sigAdderII_I_out
);

AdderDD_D:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEinp.Di_j1,
      in2=>PEtrans.tr_Dj1_Dj,
      sum=>sigAdderDD_D_out
);

AdderMD_D:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEinp.Mi_j1,
      in2=>PEtrans.tr_Mj1_Dj,
      sum=>sigAdderMD_D_out
);

AdderEmitM:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEemit.EmMj_Si,
      in2=>sigMaxM3_out,
      sum=>sigAdderEmitM_out
);

AdderEmitI:saturatedAdder
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEemit.EmIj_Si,
      in2=>sigMaxI_out,
      sum=>sigAdderEmitI_out
);

------------------------------------------------------------------------
---
--Divergence calculations for the M state
------------------------------------------------------------------------
---
Max_MDI_1:max2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>i_minus_j,
      in2=>regIDIi1_j1,
      max=>sigMax_MDI_1_out
);

Max_MDI_2:max2
```

```
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>i_minus_j,
      in2=>regDDIi1_j1,
      max=>sigMax_MDI_2_out
);

--M_DI(i,j)
sigMDI_i_j <=   regMDIi1_j1 when ctlM_reg="00" else
                        sigMax_MDI_1_out when ctlM_reg="01" else
                        sigMax_MDI_2_out when ctlM_reg="10" else
                        i_minus_j when ctlM_reg="11" else
                        (others=>'0');

--XE_DI(i,j)
sigXEDI_i_j <= sigMDI_i_j when ctlXE_reg='1' else
                     PEinp.XEDIi_j1 when ctlXE_reg='0' else
                     (others=>'0');

Min_MDS_1:min2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>regIDSi1_j1,
      in2=>i_minus_j,
      min=>sigMin_MDS_1_out
);

Min_MDS_2:min2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>regDDSi1_j1,
      in2=>i_minus_j,
      min=>sigMin_MDS_2_out
);

--MDS(i,j) output assignment
sigMDS_i_j <= regMDSi1_j1 when ctlM_reg="00" else
                    sigMin_MDS_1_out when ctlM_reg="01" else
                    sigMin_MDS_2_out when ctlM_reg="10" else
                    i_minus_j when ctlM_reg="11" else
                    (others=>'0');



--XE_DI(i,j) output assignment
sigXEDS_i_j <=    sigMDS_i_j when ctlXE_reg='1' else
                        PEinp.XEDSi_j1 when ctlXE_reg='0' else
                        (others=>'0');

--MLI(i,j) output assingment
sigMLI_i_j <= regMLIi1_j1 when ctlM_reg="00" else
                    regILIi1_j1  when ctlM_reg="01" else
                    regDLIi1_j1 when ctlM_reg="10" else
                    PEinp.i_in when ctlM_reg="11" else
                    (others=>'0');
```

```
--XELI(i,j) output assignment
sigXELI_i_j <=    PEinp.XELIi_j1 when ctlXE_reg='0'else
                        sigMLI_i_j when ctlXE_reg='1'else
                        (others=>'0');

-------------------------------------------------------------------------
---
--Divergence calculations for the I state
-------------------------------------------------------------------------
---
--I_DI(i,j)
Max_IDI_1:max2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>i_minus_j,
      in2=>regMDIi1_j,
      max=>sigMax_IDI_1_out
);

Max_IDI_2:max2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>i_minus_j,
      in2=>regIDIi1_j,
      max=>sigMax_IDI_2_out
);

sigIDI_i_j <=   sigMax_IDI_1_out when ctlI_reg='0' else
                                  sigMax_IDI_2_out when ctlI_reg='1' else
                                  (others=>'0');
--IDS(i,j)
Min_IDS_1:min2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>regMDSi1_j,
      in2=>i_minus_j,
      min=>sigMin_IDS_1_out
);

Min_IDS_2:min2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>regIDSi1_j,
      in2=>i_minus_j,
      min=>sigMin_IDS_2_out
);

sigIDS_i_j <=   sigMin_IDS_1_out when ctlI_reg='0' else
                                  sigMin_IDS_2_out when ctlI_reg='1' else
                                  (others=>'0');
```

```
--ILI(i,j)
sigILI_i_j <= regMLIi1_j when ctlI_reg='0' else
              regILIi1_j when ctlI_reg='1' else
              (others=>'0');

-------------------------------------------------------------------------
---
--Divergence calculations for the D state
-------------------------------------------------------------------------
---
--D_DI(i,j)
Max_DDI_1:max2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>i_minus_j,
      in2=>PEinp.MDIi_j1,
      max=>sigMax_DDI_1_out
);

Max_DDI_2:max2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>i_minus_j,
      in2=>PEinp.DDIi_j1,
      max=>sigMax_DDI_2_out
);

sigDDI_i_j <=   sigMax_DDI_1_out when ctlD_reg='0' else
                                  sigMax_DDI_2_out when ctlD_reg='1' else
                                  (others=>'0');
--DDS(i,j)
Min_DDS_1:min2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEinp.MDSi_j1,
      in2=>i_minus_j,
      min=>sigMin_DDS_1_out
);

Min_DDS_2:min2
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEinp.DDSi_j1,
      in2=>i_minus_j,
      min=>sigMin_DDS_2_out
);

sigDDS_i_j <=   sigMin_DDS_1_out when ctlD_reg='0' else
                                  sigMin_DDS_2_out when ctlD_reg='1' else
                                  (others=>'0');

--DLI(i,j)
```

```
sigDLI_i_j <= PEinp.MLIi_j1 when ctlD_reg='0' else
              PEinp.DLIi_j1 when ctlD_reg='1' else
              (others=>'0');

--pass signal controller
process(PEinp.clk,PEinp.Sin,PEinp.reset,PEinp.load)
begin
if(Peinp.reset='1' or PEinp.load='1')then          --Reset or New HMM
  pass<=(others=>'0');
  passChange<='0';
elsif(rising_edge(PEinp.clk))then
  if(PEinp.Sin=AMINO_NUMBER+10)then     --New pass
    pass<=pass+1;
    passChange<='1';
  elsif(PEinp.Sin=AMINO_NUMBER+11)then  --New sequence
    pass<=(others=>'0');
    passChange<='1';
  else
    pass<=pass;
    passChange<='0';
  end if;
end if;
end process;

--Registration of the signals done to make the software to infer the FF
required for the
--algorithm, also the reset sigal initializates to 0. The load signal
causes the signals
--to go to the value dataLoad (tipically MIN_INT), this to begin the
calculation
--of the score for a new sentence, the enable signal serves to the
control unit to control
--the calculations in the case where the control unit needs to stall the
execution.
registers:process(PEinp.reset,PEinp.enable,PEinp.clk,PEinp.load)
begin
if(PEinp.reset='1')then
      regMi1_j1<=(others=>'0');
      regDi1_j1<=(others=>'0');
      regIi1_j1<=(others=>'0');
      regMi1_j<=(others=>'0');
      regIi1_j<=(others=>'0');
      regDi1_j<=(others=>'0');
      regEi1_j<=(others=>'0');
      regXBi1<=(others=>'0');
elsif(rising_edge(PEinp.clk))then
      if(PEinp.load='1' or passChange='1')then
            regMi1_j1<=PEinp.dataLoad;
            regDi1_j1<=PEinp.dataLoad;
            regIi1_j1<=PEinp.dataLoad;
            regMi1_j<=PEinp.dataLoad;
            regIi1_j<=PEinp.dataLoad;
            regDi1_j<=PEinp.dataLoad;
            regEi1_j<=PEinp.dataLoad;
            regXBi1<=PEinp.dataLoad;
      elsif(PEinp.enable='1' and PEinp.loadStall='0')then
            regMi1_j1<=PEinp.Mi_j1;
            regDi1_j1<=PEinp.Di_j1;
            regIi1_j1<=PEinp.Ii_j1;
            regMi1_j<=sigAdderEmitM_out;
            regIi1_j<=sigAdderEmitI_out;
```

```
            regDi1_j<=sigMaxD_out;
            regEi1_j<=sigMaxE_out;
            regXBi1<=PEinp.XBi1_in;
      else
            regMi1_j1<=regMi1_j1;
            regDi1_j1<=regDi1_j1;
            regIi1_j1<=regIi1_j1;
            regMi1_j<=regMi1_j;
            regIi1_j<=regIi1_j;
            regDi1_j<=regDi1_j;
            regEi1_j<=regEi1_j;
            regXBi1<=regXBi1;
      end if;
end if;
end process;

DivRegs1:process(PEinp.reset,PEinp.enable,PEinp.clk,PEinp.load)
begin
if(PEinp.reset='1')then
      regMDIi1_j1<=(others=>'0');
      regMDIi1_j<=(others=>'0');
      regMDSi1_j1<=(others=>'0');
      regMDSi1_j<=(others=>'0');
      regMLIi1_j1<=(others=>'0');
      regMLIi1_j<=(others=>'0');
      regIDIi1_j1<=(others=>'0');
      regIDIi1_j<=(others=>'0');
      regIDSi1_j1<=(others=>'0');
      regIDSi1_j<=(others=>'0');
      regILIi1_j1<=(others=>'0');
      regILIi1_j<=(others=>'0');
      regDDIi1_j1<=(others=>'0');
      regDDIi1_j<=(others=>'0');
      regDDSi1_j1<=(others=>'0');
      regDDSi1_j<=(others=>'0');
      regDLIi1_j1<=(others=>'0');
      regDLIi1_j<=(others=>'0');
      regXEDIi1_j<=(others=>'0');
      regXEDSi1_j<=(others=>'0');
      regXELIi1_j<=(others=>'0');
      regi<=(others=>'0');
      --test
      ctlM_reg<=(others=>'0');
      ctlI_reg<='0';
      ctlD_reg<='0';
      ctlXE_reg<='0';
elsif(rising_edge(PEinp.clk))then
      if(PEinp.load='1' or passChange='1')then
            --M STATE
            regMDIi1_j1<=ActualJ+1;
            regMDIi1_j<=ActualJ;
            regMDSi1_j1<=ActualJ+1;
            regMDSi1_j<=ActualJ;
            regMLIi1_j1<=(others=>'0');
            regMLIi1_j<=(others=>'0');
            --I STATE
            regIDIi1_j1<=ActualJ+1;
            regIDIi1_j<=ActualJ;
            regIDSi1_j1<=ActualJ+1;
            regIDSi1_j<=ActualJ;
            regILIi1_j1<=(others=>'0');
```

```
      regILIi1_j<=(others=>'0');
      --D STATE
      regDDIi1_j1<=ActualJ+1;
      regDDIi1_j<=ActualJ;
      regDDSi1_j1<=ActualJ+1;
      regDDSi1_j<=ActualJ;
      regDLIi1_j1<=(others=>'0');
      regDLIi1_j<=(others=>'0');
      --XE output register initialization
      regXEDIi1_j<=(others=>'0');
      regXEDSi1_j<=(others=>'0');
      regXELIi1_j<=(others=>'0');
      regi<=(others=>'0');
      --test
  ctlM_reg<=(others=>'0');
  ctlI_reg<='0';
  ctlD_reg<='0';
  ctlXE_reg<='0';
elsif(PEinp.enable='1' and PEinp.loadStall='0')then
      --M STATE
      regMDIi1_j1<=PEinp.MDIi_j1;
      regMDIi1_j<=sigMDI_i_j;
      regMDSi1_j1<=PEinp.MDSi_j1;
      regMDSi1_j<=sigMDS_i_j;
      regMLIi1_j1<=PEinp.MLIi_j1;
      regMLIi1_j<=sigMLI_i_j;
      --I STATE
      regIDIi1_j1<=PEinp.IDIi_j1;
      regIDIi1_j<=sigIDI_i_j;
      regIDSi1_j1<=PEinp.IDSi_j1;
      regIDSi1_j<=sigIDS_i_j;
      regILIi1_j1<=PEinp.ILIi_j1;
      regILIi1_j<=sigILI_i_j;
      --D STATE
      regDDIi1_j1<=PEinp.DDIi_j1;
      regDDIi1_j<=sigDDI_i_j;
      regDDSi1_j1<=PEinp.DDSi_j1;
      regDDSi1_j<=sigDDS_i_j;
      regDLIi1_j1<=PEinp.DLIi_j1;
      regDLIi1_j<=sigDLI_i_j;
      --E STATE
      regXEDIi1_j<=sigXEDI_i_j;
      regXEDSi1_j<=sigXEDS_i_j;
      regXELIi1_j<=sigXELI_i_j;
      regi<=PEinp.i_in;
      --test
  ctlM_reg<=ctlM;
  ctlI_reg<=ctlI;
  ctlD_reg<=ctlD;
  ctlXE_reg<=ctlXE;
else
      regMDIi1_j1<=regMDIi1_j1;
      regMDIi1_j<=regMDIi1_j;
      regMDSi1_j1<=regMDSi1_j1;
      regMDSi1_j<=regMDSi1_j;
      regMLIi1_j1<=regMLIi1_j1;
      regMLIi1_j<=regMLIi1_j;
      regIDIi1_j1<=regMDIi1_j1;
      regIDIi1_j<=regMDIi1_j;
      regIDSi1_j1<=regMDSi1_j1;
      regIDSi1_j<=regMDSi1_j;
```

```
            regILIi1_j1<=regMLIi1_j1;
            regILIi1_j<=regMLIi1_j;
            regDDIi1_j1<=regMDIi1_j1;
            regDDIi1_j<=regMDIi1_j;
            regDDSi1_j1<=regMDSi1_j1;
            regDDSi1_j<=regMDSi1_j;
            regDLIi1_j1<=regMLIi1_j1;
            regDLIi1_j<=regMLIi1_j;
            regXEDIi1_j<=regXEDIi1_j;
            regXEDSi1_j<=regXEDSi1_j;
            regXELIi1_j<=regXELIi1_j;
            regi<=regi;
            --test
        ctlM_reg<=ctlM_reg;
        ctlI_reg<=ctlI_reg;
        ctlD_reg<=ctlD_reg;
        ctlXE_reg<=ctlXE_reg;
      end if;
end if;
end process;

--output assignment
PEout.Mi1_j<=regMi1_j;
PEout.Ii1_j<=regIi1_j;
PEout.Di1_j<=regDi1_j;
PEout.XBi1<=regXBi1;
PEout.XEi1_j<=regEi1_j;
PEout.i_out<=regi;

--divergence output assignment
PEout.MDIi1_j<=regMDIi1_j;
PEout.MLIi1_j<=regMLIi1_j;
PEout.MDSi1_j<=regMDSi1_j;
PEout.IDIi1_j<=regIDIi1_j;
PEout.ILIi1_j<=regILIi1_j;
PEout.IDSi1_j<=regIDSi1_j;
PEout.DDIi1_j<=regDDIi1_j;
PEout.DLIi1_j<=regDLIi1_j;
PEout.DDSi1_j<=regDDSi1_j;
PEout.XEDIi1_j<=regXEDIi1_j;
PEout.XEDSi1_j<=regXEDSi1_j;
PEout.XELIi1_j<=regXELIi1_j;

end;
```

C.17 – ARRANJO SISTÓLICO (PRArray.vhd)

```
-- synthesis library viterbi
library ieee;
use ieee.std_logic_1164.all;

library viterbi;
use viterbi.VITERBI_PKG.all;
use viterbi.config.all;

entity PEArray is
generic(
      WORD_SIZE         :     integer := W_SIZE;
      NUMBER_OF_PES     :     integer := NUMBER_OF_PES;
```

```
      FIFO_DEPTH        :     integer := FIFO_DEPTH;
      FIFO_ADDR_WIDTH : integer := FIFO_ADDR_WIDTH;
      ADDR_WIDTH        :     integer := ADDR_WIDTH;
      AMINO_NUMBER      :     integer := AMINO_NUMBER;
      EMIT_ADDR_WIDTH :       integer     := EMIT_ADDR_WIDTH);
port(
      PEArrayDataIn           :     in    PEArrayDataInputs;
      PEArrayControlIn  :     in    PEArrayControlInputs;
      PEArrayDataOut          :     out   PEArrayDataOutputs;
      PEArrayControlOut :     out   PEArrayControlOutputs
);
end;

architecture RTL of PEArray is

--Architecture signals
--Mnemonics
--sig_<Fromcomponent>_<FromPin>_<toComponent>_<toPin>
signal sig_FifoM_dataOut_MuxM_in2         :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoI_dataOut_MuxI_in2         :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoD_dataOut_MuxD_in2         :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoXE_dataOut_MuxXE_in2       :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoMDI_dataOut_MuxMDI_in2     :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoMDS_dataOut_MuxMDS_in2     :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoMLI_dataOut_MuxMLI_in2     :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoIDI_dataOut_MuxIDI_in2     :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoIDS_dataOut_MuxIDS_in2     :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoILI_dataOut_MuxILI_in2     :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoDDI_dataOut_MuxDDI_in2     :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoDDS_dataOut_MuxDDS_in2     :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoDLI_dataOut_MuxDLI_in2     :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoXEDI_dataOut_MuxXEDI_in2   :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoXEDS_dataOut_MuxXEDS_in2   :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_FifoXELI_dataOut_MuxXELI_in2   :
      std_logic_vector(WORD_SIZE-1 downto 0);

--XB and XC component signals
signal sig_U_XB_Bi_PE1_XBi1_in            :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_U_XC_Ci_XCOut                  :
      std_logic_vector(WORD_SIZE-1 downto 0);
signal sig_U_XC_XCDI_XCDIOut        :     std_logic_vector(WORD_SIZE-1
downto 0);
signal sig_U_XC_XCDS_XCDSOut        :     std_logic_vector(WORD_SIZE-1
downto 0);
```

```
signal sig_U_XC_XCLI_XCLIOut       :     std_logic_vector(WORD_SIZE-1
downto 0);
signal sig_U_XC_XCLF_XCLFOut       :     std_logic_vector(WORD_SIZE-1
downto 0);

--Especial signals used to recurrenlty connect the PEs
type PEInputsArray is array(NUMBER_OF_PES-1 downto 0) of PEInputs;
signal inputs : PEInputsArray;

type PEOutputsArray is array(NUMBER_OF_PES-1 downto 0) of PEOutputs;
signal outputs : PEOutputsArray;

--Transition signals to connect the PE Array and the transition memory to
the register banks
signal PEArrayTransitions : PEArrayTransitionInputs;
type   regAddrArray_type   is   array   (NUMBER_OF_PES   downto   0)   of
std_logic_vector(3 downto 0);
signal regAddrArray : regAddrArray_type;
signal regWeArray : std_logic_vector(NUMBER_OF_PES downto 0);

--Emission signals to connect the PE Array to the emision memories
signal PEArrayEmisions : PEArrayEmisionInputs;

--Signal to interconnect the Sout signal of the emision memory with the
Sin
--signal of the next emision memory (the first is already connected to
the input).
--The reason we leave one extra element is to avoid the connection error
in the last
--PE.
type  S_Interconnect_type  is  array(NUMBER_OF_PES  downto  0)  of  natural
range 0 to AMINO_NUMBER+11; --0 TO 31
signal Sin_interconnect : S_Interconnect_type;

--Signals to connect the memory data ports to the input data ports of the
register banks
signal RamData_a : RamData;
signal RamData_b : RamData;

begin

-------------------------------------------------------------------------
-- THIS MUXES CHOOSE BETWEEN THE CONTROL UNIT INPUT OR THE FIFO INPUT --
-------------------------------------------------------------------------
MuxM:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.Min,
      in2=>sig_FifoM_dataOut_MuxM_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).Mi_j1
);

MuxI:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.Iin,
```

```
      in2=>sig_FifoI_dataOut_MuxI_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).Ii_j1
);

MuxD:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.Din,
      in2=>sig_FifoD_dataOut_MuxD_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).Di_j1
);

MuxXE:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.XEin,
      in2=>sig_FifoXE_dataOut_MuxXE_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).XEi_j1
);

MuxMDI:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.MDIin,
      in2=>sig_FifoMDI_dataOut_MuxMDI_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).MDIi_j1
);

MuxMDS:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.MDSin,
      in2=>sig_FifoMDS_dataOut_MuxMDS_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).MDSi_j1
);

MuxMLI:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.MLIin,
      in2=>sig_FifoMLI_dataOut_MuxMLI_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).MLIi_j1
);

MuxIDI:genericMux2_1
```

```
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.IDIin,
      in2=>sig_FifoIDI_dataOut_MuxIDI_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).IDIi_j1
);

MuxIDS:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.IDSin,
      in2=>sig_FifoIDS_dataOut_MuxIDS_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).IDSi_j1
);

MuxILI:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.ILIin,
      in2=>sig_FifoILI_dataOut_MuxILI_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).ILIi_j1
);

MuxDDI:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.DDIin,
      in2=>sig_FifoDDI_dataOut_MuxDDI_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).DDIi_j1
);

MuxDDS:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.DDSin,
      in2=>sig_FifoDDS_dataOut_MuxDDS_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).DDSi_j1
);

MuxDLI:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.DLIin,
      in2=>sig_FifoDLI_dataOut_MuxDLI_in2,
```

```
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).DLIi_j1
);

MuxXEDI:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.XEDIin,
      in2=>sig_FifoXEDI_dataOut_MuxXEDI_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).XEDIi_j1
);

MuxXEDS:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.XEDSin,
      in2=>sig_FifoXEDS_dataOut_MuxXEDS_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).XEDSi_j1
);

MuxXELI:genericMux2_1
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      in1=>PEArrayDataIn.XELIin,
      in2=>sig_FifoXELI_dataOut_MuxXELI_in2,
      sel=>PEArrayControlIn.passMuxSel,
      outp=> inputs(0).XELIi_j1
);

-------------------------------------------------------------------------
--                    XB SCORE CALCULATOR                              --
-------------------------------------------------------------------------
U_XB:xbCalc
generic map(
      WORD_SIZE => WORD_SIZE
)
port map(
      dataLoad => PEArrayDataIn.trN_B,
      trN_N  => PEArrayDataIn.trN_N,
      load   => regWeArray(0),--PEArrayControlIn.load,
      reset  => PEArrayControlIn.reset,
      enable       => PEArrayControlIn.enable,
      clk          => PEArrayControlIn.clk,
      Sin  => Sin_interconnect(1),
      Bi           => sig_U_XB_Bi_PE1_XBi1_in
);

-------------------------------------------------------------------------
--                    XC SCORE CALCULATOR                              --
-------------------------------------------------------------------------
U_XC:xcCalc
generic map(
      WORD_SIZE => WORD_SIZE
```

```
)
port map(
      load  => regWeArray(NUMBER_OF_PES-1),--PEArrayControlIn.load,
      reset => PEArrayControlIn.reset,
      enable      => PEArrayControlIn.enable,
      clk         => PEArrayControlIn.clk,
      dataLoad=> PEArrayDataIn.dataLoad,
      XEin  => outputs(NUMBER_OF_PES-1).XEi1_j,
      trC_C => PEArrayDataIn.trC_C,
      trE_C => PEArrayDataIn.trE_C,
      i_in  => outputs(NUMBER_OF_PES-1).I_OUT,
      Sin   => Sin_interconnect(NUMBER_OF_PES),
      --Divergence
      XEDIin      => outputs(NUMBER_OF_PES-1).XEDIi1_j,
      XEDSin      => outputs(NUMBER_OF_PES-1).XEDSi1_j,
      XELIin      => outputs(NUMBER_OF_PES-1).XELIi1_j,
      --Output
      XCDI  => sig_U_XC_XCDI_XCDIOut,
      XCDS  => sig_U_XC_XCDS_XCDSOut,
      XCLI  => sig_U_XC_XCLI_XCLIOut,
      XCLF  => sig_U_XC_XCLF_XCLFOut,
      Ci          => sig_U_XC_Ci_XCOut
);

-------------------------------------------------------------------------
--                                  FIRST                                PE
--
-------------------------------------------------------------------------
inputs(0).reset<=PEArrayControlIn.reset;
inputs(0).load<=PEArrayControlIn.we_trans(0);
inputs(0).enable<=PEArrayControlIn.enable;
inputs(0).clk<=PEArrayControlIn.clk;
inputs(0).dataLoad<=PEArrayDataIn.dataLoad;
inputs(0).XBi1_in<=sig_U_XB_Bi_PE1_XBi1_in;
inputs(0).i_in<=PEArrayDataIn.i_in;
inputs(0).loadStall<=regWeArray(0);
inputs(0).Sin<=Sin_interconnect(0);

PE_1 : PE
generic map(
      BASE_J => 1,
      WORD_SIZE => WORD_SIZE
)
port map(
      PEinp   => inputs(0),
      PEtrans => PEArrayTransitions(0),
      PEemit  => PEArrayEmisions(0),
      PEout   => outputs(0)
);

-------------------------------------------------------------------------
--                                  OTHER                                PEs
--
-------------------------------------------------------------------------
GEN_PES:for i in 1 to (NUMBER_OF_PES-1) generate
      inputs(i).reset<=PEArrayControlIn.reset;
      inputs(i).load<=PEArrayControlIn.we_trans(i);
      inputs(i).enable<=PEArrayControlIn.enable;
      inputs(i).clk<=PEArrayControlIn.clk;
      inputs(i).dataLoad<=PEArrayDataIn.dataLoad;
      inputs(i).loadStall<=regWeArray(i);
```

```
      inputs(i).Sin<=Sin_interconnect(i);

      --This inputs are connected to the previous PE output
      inputs(i).Mi_j1<=outputs(i-1).Mi1_j;
      inputs(i).Ii_j1<=outputs(i-1).Ii1_j;
      inputs(i).Di_j1<=outputs(i-1).Di1_j;
      inputs(i).XBi1_in<=outputs(i-1).XBi1;
      inputs(i).XEi_j1<=outputs(i-1).XEi1_j;
      --divergence
      inputs(i).MDIi_j1<=outputs(i-1).MDIi1_j;
      inputs(i).MDSi_j1<=outputs(i-1).MDSi1_j;
      inputs(i).MLIi_j1<=outputs(i-1).MLIi1_j;
      inputs(i).IDIi_j1<=outputs(i-1).IDIi1_j;
      inputs(i).IDSi_j1<=outputs(i-1).IDSi1_j;
      inputs(i).ILIi_j1<=outputs(i-1).ILIi1_j;
      inputs(i).DDIi_j1<=outputs(i-1).DDIi1_j;
      inputs(i).DDSi_j1<=outputs(i-1).DDSi1_j;
      inputs(i).DLIi_j1<=outputs(i-1).DLIi1_j;
      inputs(i).XEDIi_j1<=outputs(i-1).XEDIi1_j;
      inputs(i).XEDSi_j1<=outputs(i-1).XEDSi1_j;
      inputs(i).XELIi_j1<=outputs(i-1).XELIi1_j;
      inputs(i).i_in<=outputs(i-1).i_out;

      PE_i : PE
      generic map(
          BASE_J => i+1,
              WORD_SIZE => WORD_SIZE
      )
      port map(
            PEinp   => inputs(i),
            PEtrans => PEArrayTransitions(i),
            PEemit  => PEArrayEmisions(i),
            PEout   => outputs(i)
      );
end generate;

-------------------------------------------------------------------------
--                          M,I,D,XE FIFOS                             --
-------------------------------------------------------------------------
FifoM:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.WriteEnable,
      ReadEnable =>PEArrayControlIn.ReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).Mi1_j,
      DataOut => sig_FifoM_dataOut_MuxM_in2
);

-- I
FifoI:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
```

```
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.WriteEnable,
      ReadEnable =>PEArrayControlIn.ReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).Ii1_j,
      DataOut => sig_FifoI_dataOut_MuxI_in2
);

-- D
FifoD:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.WriteEnable,
      ReadEnable =>PEArrayControlIn.ReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).Di1_j,
      DataOut =>  sig_FifoD_dataOut_MuxD_in2
);

--XE
--Note: As all fifos have the same data, only the XE signals for control
are taken
--into account (fifoEmpty and FifoFull)
FifoXE:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.WriteEnable,
      ReadEnable =>PEArrayControlIn.ReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).XEi1_j,
      DataOut => sig_FifoXE_dataOut_MuxXE_in2,
    FifoEmpty => PEArrayControlOut.FifoEmpty,
    FifoFull => PEArrayControlOut.FifoFull
);

------------------------------------------------------------------------
--        DIVERGENCE  (MDI,MDS,MLI,IDI,IDS,ILI,DDI,DDS,DLI,XEDI,XEDS,XELI)
FIFOS                  --
------------------------------------------------------------------------
--MDI
FifoMDI:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
```

```
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).MDIi1_j,
      DataOut => sig_FifoMDI_dataOut_MuxMDI_in2,
      FifoEmpty => PEArrayControlOut.DivFifoEmpty,
  FifoFull => PEArrayControlOut.DivFifoFull
);

--MDS
FifoMDS:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).MDSi1_j,
      DataOut => sig_FifoMDS_dataOut_MuxMDS_in2
);

--MLI
FifoMLI:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).MLIi1_j,
      DataOut => sig_FifoMLI_dataOut_MuxMLI_in2
);

--IDI
FifoIDI:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).IDIi1_j,
      DataOut => sig_FifoIDI_dataOut_MuxIDI_in2
);

--IDS
FifoIDS:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
```

```
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).IDSi1_j,
      DataOut => sig_FifoIDS_dataOut_MuxIDS_in2
);

--ILI
FifoILI:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).ILIi1_j,
      DataOut => sig_FifoILI_dataOut_MuxILI_in2
);

--DDI
FifoDDI:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).DDIi1_j,
      DataOut => sig_FifoDDI_dataOut_MuxDDI_in2
);

--DDS
FifoDDS:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).DDSi1_j,
      DataOut => sig_FifoDDS_dataOut_MuxDDS_in2
);

--DLI
FifoDLI:fifo
generic map(
```

```
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).DLIi1_j,
      DataOut => sig_FifoDLI_dataOut_MuxDLI_in2
);

--XEDI
FifoXEDI:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).XEDIi1_j,
      DataOut => sig_FifoXEDI_dataOut_MuxXEDI_in2
);

--XEDS
FifoXEDS:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).XEDSi1_j,
      DataOut => sig_FifoXEDS_dataOut_MuxXEDS_in2
);

--XELI
FifoXELI:fifo
generic map(
      WORD_SIZE  =>WORD_SIZE,
      FIFO_DEPTH =>FIFO_DEPTH,
      ADDR_WIDTH =>FIFO_ADDR_WIDTH
)
port map(
      Clk =>PEArrayControlIn.clk,
      Reset =>PEArrayControlIn.reset,
      WriteEnable =>PEArrayControlIn.DivWriteEnable,
      ReadEnable =>PEArrayControlIn.DivReadEnable,
      DataIn => outputs(NUMBER_OF_PES-1).XELIi1_j,
      DataOut => sig_FifoXELI_dataOut_MuxXELI_in2
);
```

```
-------------------------------------------------------------------------
--                REGISTER BANK, EMISION MEMORIES AND RAM IMPLEMENTATION
--
-------------------------------------------------------------------------

--connection of the Sin input with Sin_interconnect(0)
Sin_interconnect(0) <= PEArrayDataIn.Sin;

genRegRamEmit:for i in 0 to (NUMBER_OF_PES-1) generate
--Generate and conect the register banks
regBank_i:trRegisters
generic map(
      WORD_SIZE=>WORD_SIZE
)
port map(
      data_1=>RamData_a(i),
      data_2=>RamData_b(i),
      regAddr=>regAddrArray(i),--PEArrayControlIn.regAddr, now comes from
the transition memories
      clk=>PEArrayControlIn.clk,
      w_en=>regWeArray(i),--PEArrayControlIn.w_en,  now  comes  from  the
transition memories
      reset=>PEArrayControlIn.reset,
      tr=>PEArrayTransitions(i)
);


--Generate and conect the ram and register components
--these are the transition rams, the emision memories are next!!
TransMem_i:TransMem
generic map(
      DATA_WIDTH =>WORD_SIZE,
      ADDR_WIDTH =>ADDR_WIDTH
)
port map(
      clk=>PEArrayControlIn.clk,
      reset=>PEArrayControlIn.reset,
      addr_bus=>PEArrayControlIn.addr_bus,
      data_a=>PEArrayControlIn.data_a,
      data_b=>PEArrayControlIn.data_b,
      we=>PEArrayControlIn.we_trans(i),
      Sin=>Sin_interconnect(i),
      reg_we=>regWeArray(i),
      regAddr=>regAddrArray(i),
      q_a=>RamData_a(i),
      q_b=>RamData_b(i)
      );

--Generate the emision memories
EmisionMem_i:emitMem
generic map(
      WORD_SIZE=>WORD_SIZE,
      AMINO_NUMBER=>AMINO_NUMBER,
      EMIT_ADDR_WIDTH=>EMIT_ADDR_WIDTH
)
port map(
      clk=>PEArrayControlIn.clk,
      reset=>PEArrayControlIn.reset,
      addr=>PEArrayControlIn.addr_bus,
      data1=>PEArrayControlIn.data_a,
      data2=>PEArrayControlIn.data_b,
```

```
      we=>PEArrayControlIn.EmitEn(i),
      Sin=>Sin_interconnect(i),
      emM=>PEArrayEmisions(i).EmMj_Si,
      emI=>PEArrayEmisions(i).EmIj_Si,
      Sout=>Sin_interconnect(i+1)
);

end generate;

-------------------------------------------------------------------------
--Viterbi score outputs
-------------------------------------------------------------------------
PEArrayDataOut.XCDIout <= sig_U_XC_XCDI_XCDIOut;
PEArrayDataOut.XCDSout <= sig_U_XC_XCDS_XCDSOut;
PEArrayDataOut.XCLIout <= sig_U_XC_XCLI_XCLIOut;
PEArrayDataOut.XCLFout <= sig_U_XC_XCLF_XCLFOut;
end RTL;
```

C.18 – TEST BENCH (viterbi_tb.vhd)

```
--synthesis library viterbi

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_SIGNED.all;
use std.textio.all;

library viterbi;
use viterbi.VITERBI_PKG.all;
use viterbi.config.all;

entity viterbi_tb is
--just a test bench
generic(
  HMM_trans_file : string := "../HMM_files/transitions.hmm";
  HMM_emit_file : string := "../HMM_files/emissions.hmm";
  seq_file : string := "../HMM_files/sequence.seq";
  out_file : string := "../HMM_files/out_general.txt";
  out_file2 : string := "../HMM_files/out_scores.txt";
  clk_period : time := 14.2857 ns;
  WORD_SIZE             :     integer := W_SIZE;
      NUMBER_OF_PES     :     integer := NUMBER_OF_PES;
      FIFO_DEPTH        :     integer := FIFO_DEPTH;
      FIFO_ADDR_WIDTH : integer := FIFO_ADDR_WIDTH;
      ADDR_WIDTH        :     integer := ADDR_WIDTH;
      AMINO_NUMBER      :     integer := AMINO_NUMBER;
      EMIT_ADDR_WIDTH :       integer     := EMIT_ADDR_WIDTH
);
end viterbi_tb;

architecture test_bench of viterbi_tb is
--Signals to count the total clock cycles spent to read in the HMM, then
to compare
--X number of sequences
```

```
signal cycle_count : integer range 0 to 700000000 :=0;
signal begin_count   : std_logic := '0';

--signal to count how many sequences were processed by the Array with the
given HMM
--file and the given seq file
signal sequence_count : integer range 0 to 30000 :=0;

--PEArray connection signals
signal PEArrayControlIn   :   PEArrayControlInputs;
signal PEArrayDataIn             : PEArrayDataInputs;
signal PEArrayControlOut      :    PEArrayControlOutputs;
signal PEArrayDataOut           :  PEArrayDataOutputs;

begin

--Process to generate the system clock with the desired frequency
clk_proc:process
begin
  PEArrayControlIn.clk<='0'; wait for clk_period/2;
  PEArrayControlIn.clk<='1'; wait for clk_period/2;
end process;

--Process to count the clock cycles spent during a test run
count_proc:process(PEArrayControlIn.clk)
begin
  if(rising_edge(PEArrayControlIn.clk))then
    if(begin_count='1')then
      cycle_count<=cycle_count+1;
    end if;
  end if;
end process;

--Process to read in the HMM and to program the PEArray to make the
calculations
proc_HMM:process
  --HMM files to read
  file HMM_trans : text open read_mode is HMM_trans_file;
  file HMM_emit  : text open read_mode is HMM_emit_file;
  file sequences : text open read_mode is seq_file;
  file outFile   : text open write_mode is out_file;

  --Variable to read each line of the file
  variable lin_in1 : line;
  variable lin_in2 : line;

  --Variable to write messages to console or to a file
  variable lin_out : line;

  --variable to hold signed intermediary values
  variable   temp_value   :   integer   range   -(2**(WORD_SIZE-1))   to
(2**(WORD_SIZE-1))-1;

  --AUXILIAR VARIABLES
  variable passVarTrans,passVarEm : integer range 0 to (2**WORD_SIZE)-1;
--In which pass are we??
  variable seqElement         : character;
  variable iterations         : integer;
  variable messages           : string(1 to 40);
  variable elementCount       : integer:=0;
```

```
  --function to parse a sequence character into a number to address the
emission
  --memories
  function parseSeqChar(char:character)return natural is
  begin
    if(char='A')then
      return 0;
    end if;
    if(char='C')then
      return 1;
    end if;
    if(char='D')then
      return 2;
    end if;
    if(char='E')then
      return 3;
    end if;
    if(char='F')then
      return 4;
    end if;
    if(char='G')then
      return 5;
    end if;
    if(char='H')then
      return 6;
    end if;
    if(char='I')then
      return 7;
    end if;
    if(char='K')then
      return 8;
    end if;
    if(char='L')then
      return 9;
    end if;
    if(char='M')then
      return 10;
    end if;
    if(char='N')then
      return 11;
    end if;
    if(char='P')then
      return 12;
    end if;
    if(char='Q')then
      return 13;
    end if;
    if(char='R')then
      return 14;
    end if;
    if(char='S')then
      return 15;
    end if;
    if(char='T')then
      return 16;
    end if;
    if(char='V')then
      return 17;
    end if;
    if(char='W')then
      return 18;
```

```
    end if;
    if(char='Y')then
      return 19;
    end if;
    if(char='*')then
      return 30;
    end if;
    if(char='@')then
      return 31;
    end if;
  end parseSeqChar;

begin
  wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

  ------------------------------------------------------------------------
----
  --Initializing all to zero
  ------------------------------------------------------------------------
----
  PEArrayControlIn.reset<='0';                    PEArrayDataIn.dataLoad
<="1000000000000000";
  PEArrayControlIn.load<='0';
PEArrayDataIn.trN_N<=(others=>'0');
  PEArrayControlIn.enable<='0';
PEArrayDataIn.trN_B<=(others=>'0');
  PEArrayControlIn.passMuxSel<='0';
PEArrayDataIn.trC_C<=(others=>'0');
  PEArrayControlIn.WriteEnable<='0';
PEArrayDataIn.trE_C<=(others=>'0');
  PEArrayControlIn.ReadEnable<='0';
PEArrayDataIn.Min<="1000000000000000";
  PEArrayControlIn.DivReadEnable<='0';
PEArrayDataIn.Iin<="1000000000000000";
  PEArrayControlIn.DivWriteEnable<='0';
PEArrayDataIn.Din<="1000000000000000";
  PEArrayControlIn.we_trans<=(others=>'0');
PEArrayDataIn.XEin<="1000000000000000";
  PEArrayControlIn.addr_bus<=0;               PEArrayDataIn.Sin<=0;
  PEArrayControlIn.data_a<=(others=>'0');
PEArrayDataIn.MDIin<=(others=>'0');
  PEArrayControlIn.data_b<=(others=>'0');
PEArrayDataIn.MDSin<=(others=>'0');
  PEArrayControlIn.EmitEn<=(others=>'0');
PEArrayDataIn.MLIin<=(others=>'0');

PEArrayDataIn.IDIin<=(others=>'0');

PEArrayDataIn.IDSin<=(others=>'0');

PEArrayDataIn.ILIin<=(others=>'0');

PEArrayDataIn.DDIin<=(others=>'0');

PEArrayDataIn.DDSin<=(others=>'0');

PEArrayDataIn.DLIin<=(others=>'0');

PEArrayDataIn.XEDIin<=(others=>'0');

PEArrayDataIn.XEDSin<=(others=>'0');
```

```
PEArrayDataIn.XELIin<=(others=>'0');

PEArrayDataIn.i_in<=(others=>'0');
  wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

  ------------------------------------------------------------------------
----
  --Reseting the system and enabling the cycle count
  ------------------------------------------------------------------------
----
  PEArrayControlIn.reset<='1';
  begin_count<='1';
  wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';
  PEArrayControlIn.reset<='0';

  --REPORTING TO THE TEXT OUTPUT
  messages:="Beginning test bench operation          ";
  write(lin_out,messages);
  writeline(outFile,lin_out);
  messages:="Cycle count:                            ";
  write(lin_out,messages);
  write(lin_out,cycle_count);
  writeline(outFile,lin_out);

  ------------------------------------------------------------------------
----
  --   SPECIAL PROBABILITIES LOADING INTO THE PEs MEMORY
  ------------------------------------------------------------------------
----
  readline(HMM_trans,lin_in1);
  read(lin_in1,temp_value);
  PEArrayDataIn.trN_B<=conv_std_logic_vector(temp_value,WORD_SIZE);
  wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

  read(lin_in1,temp_value);
  PEArrayDataIn.trN_N<=conv_std_logic_vector(temp_value,WORD_SIZE);
  wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

  read(lin_in1,temp_value);
  PEArrayDataIn.trC_C<=conv_std_logic_vector(temp_value,WORD_SIZE);
  wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

  read(lin_in1,temp_value);
  PEArrayDataIn.trE_C<=conv_std_logic_vector(temp_value,WORD_SIZE);
  wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

  --REPORTING TO THE TEXT OUTPUT
  messages:="Finished Special probs loading          ";
  write(lin_out,messages);
  writeline(outFile,lin_out);
  messages:="Cycle count:                            ";
  write(lin_out,messages);
  write(lin_out,cycle_count);
  writeline(outFile,lin_out);

  ------------------------------------------------------------------------
----
  --   TRANSITION PROBABILITIES LOADING INTO THE PEs MEMORY
  ------------------------------------------------------------------------
----
```

```
  passVarTrans:=0;
  PEArrayControlIn.we_trans(0)<='1';     --initializing the transition's
memory write signal
  PEArrayControlIn.we_trans(NUMBER_OF_PES-1 downto 1)<=(others=>'0');

   --reading the transitions probability file from the parser output
  while(not(endfile(HMM_trans)))loop
    --read one line (9 transitions)
    readline(HMM_trans,lin_in1);

    PEArrayControlIn.addr_bus<=0+10*passVarTrans;
    read(lin_in1,temp_value);
    PEArrayControlIn.data_a<=conv_std_logic_vector(temp_value,WORD_SIZE);
    read(lin_in1,temp_value);
    PEArrayControlIn.data_b<=conv_std_logic_vector(temp_value,WORD_SIZE);
    wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

    PEArrayControlIn.addr_bus<=2+10*passVarTrans;
    read(lin_in1,temp_value);
    PEArrayControlIn.data_a<=conv_std_logic_vector(temp_value,WORD_SIZE);
    read(lin_in1,temp_value);
    PEArrayControlIn.data_b<=conv_std_logic_vector(temp_value,WORD_SIZE);
    wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

    PEArrayControlIn.addr_bus<=4+10*passVarTrans;
    read(lin_in1,temp_value);
    PEArrayControlIn.data_a<=conv_std_logic_vector(temp_value,WORD_SIZE);
    read(lin_in1,temp_value);
    PEArrayControlIn.data_b<=conv_std_logic_vector(temp_value,WORD_SIZE);
    wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

    PEArrayControlIn.addr_bus<=6+10*passVarTrans;
    read(lin_in1,temp_value);
    PEArrayControlIn.data_a<=conv_std_logic_vector(temp_value,WORD_SIZE);
    read(lin_in1,temp_value);
    PEArrayControlIn.data_b<=conv_std_logic_vector(temp_value,WORD_SIZE);
    wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

    PEArrayControlIn.addr_bus<=8+10*passVarTrans;
    read(lin_in1,temp_value);
    PEArrayControlIn.data_a<=conv_std_logic_vector(temp_value,WORD_SIZE);
    wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';

    --passVar is used to generate the transition memory address
acorrdingly to the number
    --of PEs
    if(PEArrayControlIn.we_trans(NUMBER_OF_PES-1)='1')then
      passVarTrans:=passVarTrans+1;
      PEArrayControlIn.we_trans(0)<='1';          --initializing   the
transition's memory write signal
      PEArrayControlIn.we_trans(NUMBER_OF_PES-1 downto 1)<=(others=>'0');
    else
      passVarTrans:=passVarTrans;
      --incrementing the transition's memories enable signal
      PEArrayControlIn.we_trans(NUMBER_OF_PES-1                    downto
1)<=PEArrayControlIn.we_trans(NUMBER_OF_PES-2 downto 0);
      PEArrayControlIn.we_trans(0)<='0';
    end if;
  end loop;
  --Clearing the transition meories enable signals to avoid erroneus
writings
```

```
  PEArrayControlIn.we_trans<=(others=>'0');

  --REPORTING TO THE TEXT OUTPUT
  messages:="Finished transition probs loading       ";
  write(lin_out,messages);
  writeline(outFile,lin_out);
  messages:="Cycle count:                            ";
  write(lin_out,messages);
  write(lin_out,cycle_count);
  writeline(outFile,lin_out);

  ------------------------------------------------------------------------
--
  --   EMISSION PROBABILITIES LOADING INTO THE PEs MEMORY
  ------------------------------------------------------------------------
--
  passVarEm:=0; --Initializing the pass number

  PEArrayControlIn.EmitEn(0)<='1';   --initializing the EMISSION's memory
write signal
  PEArrayControlIn.EmitEn(NUMBER_OF_PES-1 downto 1)<=(others=>'0');

  --reading the emissions probability file from the parser output
  while(not(endfile(HMM_emit)))loop
    --read one line (20 M-emissions)
    readline(HMM_emit,lin_in1);
    --read one line (20 I-emissions)
    readline(HMM_emit,lin_in2);

    for i in 0 to 19 loop
      PEArrayControlIn.addr_bus<=i+40*passVarEm;
      read(lin_in1,temp_value);

PEArrayControlIn.data_a<=conv_std_logic_vector(temp_value,WORD_SIZE);
      read(lin_in2,temp_value);

PEArrayControlIn.data_b<=conv_std_logic_vector(temp_value,WORD_SIZE);
      wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';
    end loop;

    --passVar  is  used  to  generate  the  transition  memory  address
acorrdingly to the number
    --of PEs
    if(PEArrayControlIn.EmitEn(NUMBER_OF_PES-1)='1')then
      passVarEm:=passVarEm+1;
      PEArrayControlIn.EmitEn(0)<='1';    --initializing the transition's
memory write signal
      PEArrayControlIn.EmitEn(NUMBER_OF_PES-1 downto 1)<=(others=>'0');
    else
      passVarEm:=passVarEm;
      --incrementing the transition's memories enable signal
      PEArrayControlIn.EmitEn(NUMBER_OF_PES-1                        downto
1)<=PEArrayControlIn.EmitEn(NUMBER_OF_PES-2 downto 0);
      PEArrayControlIn.EmitEn(0)<='0';
    end if;

  end loop;

  --Clearing  the  transition  memories  enable  signals  to  avoid  erroneus
writings
  PEArrayControlIn.EmitEn<=(others=>'0');
```

```
  --Checking if the files are parsed correctly
  ASSERT(passVarTrans=passVarEm)
    report "Number of passes different for the transmition and emission
files!!"
    severity ERROR;

  --REPORTING TO THE TEXT OUTPUT
  messages:="Number of PEs:                          ";
  write(lin_out,messages);
  write(lin_out,NUMBER_OF_PES);
  writeline(outFile,lin_out);

  messages:="Number of passes:                       ";
  write(lin_out,messages);
  write(lin_out,passVarTrans);
  writeline(outFile,lin_out);

  messages:="Finished emission probs loading         ";
  write(lin_out,messages);
  writeline(outFile,lin_out);
  messages:="Cycle count:                            ";
  write(lin_out,messages);
  write(lin_out,cycle_count);
  writeline(outFile,lin_out);

-------------------------------------------------------------------------
-------
-- READING THE INPUT SEQUENCE FILE, APLYING THE SEQUENCES AND GETTING THE
SCORES
-------------------------------------------------------------------------
-------
  while(not(endfile(sequences)))loop
    readline(sequences,lin_in1); --First Line of the sequence (contains
the description)
    writeline(outFile,lin_in1);

    --Making all the necesary passes to process the sequence
    for i in 0 to passVarTrans-1 loop
       --actual sequence line (must be replicated in the parser)
      readline(sequences,lin_in2);

      --inserting the first sequence element one clock before enabling
the operation
      --in order to provide one cycle delay to enable the emission
memories to address
      --data correctly
      read(lin_in2,seqElement);
      PEArrayDataIn.Sin<=parseSeqChar(seqElement);

      --if it is not the first pass, the read enable signal for the
intermediary fifos
      --must be activated to read intermediary data
      if(not(i=0))then
        PEArrayControlIn.ReadEnable<='1';
      else
        PEArrayControlIn.ReadEnable<='0';
      end if;
      wait until PEArrayControlIn.clk'event and PEArrayControlIn.clk='1';
```

```
      --if it is not the first pass, the read enable signal for the
intermediary fifos
      --must be activated to read intermediary divergences
      if(not(i=0))then
        PEArrayControlIn.DivReadEnable<='1';
      else
        PEArrayControlIn.DivReadEnable<='0';
      end if;

      --Selection signal
      if(not(i=0))then
        PEArrayControlIn.passMuxSel<='1';
      else
        PEArrayControlIn.passMuxSel<='0';
      end if;

      --determining how many sequence elements i am going to insert
      iterations:=lin_in2'length;

      --REPORT SEQUENCE SIZE TO THE TEXT OUTPUT
      if(i=0)then
        messages:="Sequence Size:                          ";
        write(lin_out,messages);
        write(lin_out,iterations+1);
        elementCount:=elementCount+iterations+1;
        writeline(outFile,lin_out);
      end if;

      --inserting the other sequence elements and setting the other data
signals
      for j in 0 to iterations-1 loop
        --ENABLE PE ARRAY OPERATION
        PEArrayControlIn.Enable<='1';

        --EXTERNAL CONTROL PROVIDED INPUTS
        PEArrayDataIn.i_in<=conv_std_logic_vector(j,WORD_SIZE);
        PEArrayDataIn.MDIin<=conv_std_logic_vector(j,WORD_SIZE);
        PEArrayDataIn.MDSin<=conv_std_logic_vector(j,WORD_SIZE);
        PEArrayDataIn.MLIin<=conv_std_logic_vector(j,WORD_SIZE);
        PEArrayDataIn.IDIin<=conv_std_logic_vector(j,WORD_SIZE);
        PEArrayDataIn.IDSin<=conv_std_logic_vector(j,WORD_SIZE);
        PEArrayDataIn.ILIin<=conv_std_logic_vector(j,WORD_SIZE);
        PEArrayDataIn.DDIin<=conv_std_logic_vector(j,WORD_SIZE);
        PEArrayDataIn.DDSin<=conv_std_logic_vector(j,WORD_SIZE);
        PEArrayDataIn.DLIin<=conv_std_logic_vector(j,WORD_SIZE);

        --READ EACH SEQUENCE ELEMENT
        read(lin_in2,seqElement);

        --DETERMINE WHEN TO ENABLE THE INTERMEDIARY RESULT STORAGE (FIFO)
        --Score
        if(j>=NUMBER_OF_PES AND not(i=passVarTrans-1))then
          PEArrayControlIn.WriteEnable<='1';
        elsif(PEArrayDataOut.Sout=30 or PEArrayDataOut.Sout=31)then
          PEArrayControlIn.WriteEnable<='0';
        end if;
        --Divergences
        if(j>NUMBER_OF_PES AND not(i=passVarTrans-1))then
          PEArrayControlIn.DivWriteEnable<='1';
        elsif(PEArrayDataOut.Sout=30 or PEArrayDataOut.Sout=31)then
          PEArrayControlIn.DivWriteEnable<='0';
```

```
        end if;

        if(parseSeqChar(seqElement)=30                                    or
parseSeqChar(seqElement)=31)then
          --LAST ITERATION I_IN, MDIin,MDSin,MLIin
          wait        until        PEArrayControlIn.clk'event        and
PEArrayControlIn.clk='1';
          PEArrayControlIn.ReadEnable<='0';
          PEArrayControlIn.DivReadEnable<='0';

PEArrayDataIn.i_in<=conv_std_logic_vector(iterations,WORD_SIZE);

PEArrayDataIn.MDIin<=conv_std_logic_vector(iterations,WORD_SIZE);

PEArrayDataIn.MDSin<=conv_std_logic_vector(iterations,WORD_SIZE);

PEArrayDataIn.MLIin<=conv_std_logic_vector(iterations,WORD_SIZE);

PEArrayDataIn.IDIin<=conv_std_logic_vector(iterations,WORD_SIZE);

PEArrayDataIn.IDSin<=conv_std_logic_vector(iterations,WORD_SIZE);

PEArrayDataIn.ILIin<=conv_std_logic_vector(iterations,WORD_SIZE);

PEArrayDataIn.DDIin<=conv_std_logic_vector(iterations,WORD_SIZE);

PEArrayDataIn.DDSin<=conv_std_logic_vector(iterations,WORD_SIZE);

PEArrayDataIn.DLIin<=conv_std_logic_vector(iterations,WORD_SIZE);
          PEArrayDataIn.Sin<=parseSeqChar(seqElement);
        else
          PEArrayDataIn.Sin<=parseSeqChar(seqElement);
        end if;

        --WAIT ONE CLOCK CYCLE PER ITERATION
        wait         until         PEArrayControlIn.clk'event         and
PEArrayControlIn.clk='1';
      end loop;


      --must  zero  the  residue  input  for  5  cycles  waiting  for  the
transition probs
      --to load into the register bank ojo hacer la espera si se tiene
mas de una pasada
      --solamente!!!
      if(passVarTrans/=1)then
        PEArrayDataIn.Sin<=0;
        wait         until         PEArrayControlIn.clk'event         and
PEArrayControlIn.clk='1';
        wait         until         PEArrayControlIn.clk'event         and
PEArrayControlIn.clk='1';
        wait         until         PEArrayControlIn.clk'event         and
PEArrayControlIn.clk='1';
        wait         until         PEArrayControlIn.clk'event         and
PEArrayControlIn.clk='1';
        wait         until         PEArrayControlIn.clk'event         and
PEArrayControlIn.clk='1';
        --wait        until        PEArrayControlIn.clk'event        and
PEArrayControlIn.clk='1';
      end if;
```

```
--
--      PEArrayControlIn.Enable<='0';
--        para varias pasadas se activa el write enable hasta un ciclo
despues!!!!
--                      wait    until    PEArrayControlIn.clk'event    and
PEArrayControlIn.clk='1';
--      PEArrayControlIn.WriteEnable<='0';

    end loop; --end passes

  end loop; --end of sequences file

  --Zeroing the residue input to prevent the test bench to register wrong
scores
  PEArrayDataIn.Sin<=0;

  messages:="Total Residues Count:                    ";
  write(lin_out,messages);
  write(lin_out,elementCount);
  writeline(outFile,lin_out);

  wait;
end process;

--Process to get the output from the array, this process only gets the
output
--when the entire sequence has been processed (the @ (31) character has
appeared)
--
--Must get the result on the falling edge of the clock due to design
constraints
process(PEArrayDataOut.Sout,PEArrayControlIn.clk)
  --File and line variables to write to the text output
  file outFile2    : text open write_mode is out_file2;
  variable lin_out : line;
  variable messages: string(1 to 61);
  variable tab : character:=HT;

  --Current sequence number
  variable seq_num : integer range 0 to 5000000:=1;
begin
  if(rising_edge(PEArrayControlIn.clk))then
    if(PEArrayDataOut.Sout=31)then
      if(seq_num=1)then
        messages:="Seq    Length    XCout    LinIni/LinFin    DivInf/DivSup
cycles";
        write(lin_out,messages);
        writeline(outFile2,lin_out);
      end if;
        write(lin_out,seq_num);
        write(lin_out,tab);
        write(lin_out,conv_integer(PEArrayDataOut.Iout));
        write(lin_out,tab);
        write(lin_out,conv_integer(PEArrayDataOut.XCout));
        write(lin_out,tab);
        write(lin_out,conv_integer(PEArrayDataOut.XCLIout));
        write(lin_out,tab);
        write(lin_out,conv_integer(PEArrayDataOut.XCLFout));
        write(lin_out,tab);
        write(lin_out,conv_integer(PEArrayDataOut.XCDIout));
        write(lin_out,tab);
```

```
        write(lin_out,conv_integer(PEArrayDataOut.XCDSout));
        write(lin_out,tab);
        write(lin_out,cycle_count);
        writeline(outFile2,lin_out);
        seq_num:=seq_num+1;
    end if;
  end if;
end process;


--------------------------------------------------------------------------
--PE ARRAY COMPONENT DECLARATION TO CONNECT TO THE TEST BENCH SIGNALS
--------------------------------------------------------------------------
PEArray1:PEArray
generic map(
  WORD_SIZE             => WORD_SIZE,
      NUMBER_OF_PES     => NUMBER_OF_PES,
      FIFO_DEPTH        => FIFO_DEPTH,
      FIFO_ADDR_WIDTH => FIFO_ADDR_WIDTH,
      ADDR_WIDTH        => ADDR_WIDTH,
      AMINO_NUMBER      => AMINO_NUMBER,
      EMIT_ADDR_WIDTH => EMIT_ADDR_WIDTH
)
port map(
  PEArrayDataIn      =>PEArrayDataIn,
      PEArrayControlIn   =>PEArrayControlIn,
      PEArrayDataOut            =>PEArrayDataOut,
      PEArrayControlOut  =>PEArrayControlOut
);

end test_bench;
```

A HMMER Hardware Accelerator using Divergences

Juan Fernando Eusse Giraldo
Department of Electrical Engineering
University of Brasilia
Brasília/DF, Brazil
eusse@microe.udea.edu.co

Nahri Moreano
Department of Statistics and Computation
University of Mato Grosso do Sul
Campo Grande/MS, Brazil
nahri @dct.ufms.br

Ricardo Pezzuol Jacobi
Department of Computer Science
University of Brasilia
Brasília/DF, Brazil
Jacobi@unb.br

Alba Cristina Magalhães Alves de Melo
Department of Computer Science
University of Brasilia
Brasília/DF, Brazil
albamm@cic.unb.br

Abstract - As new protein sequences are discovered on an everyday basis and protein databases continue to grow exponentially with time, computational tools take more and more time to search protein databases to discover the common ancestors of them. HMMER is among the most used tools in protein search and comparison and multiple efforts have been made to accelerate its execution by using dedicated hardware prototyped on FPGAs. In this paper we introduce a novel algorithm called the Divergence Algorithm, which not only enables the FPGA accelerator to reduce execution time, but also enables further acceleration of the alignment generation algorithm of the HMMER programs by reducing the number of cells of the Dynamic Programming matrices it has to calculate. We also propose a more accurate performance measurement strategy that considers all the execution times while doing protein searches and alignments, while other works only considered hardware execution times and did not included alignment generation times. By the inclusion of the hardware accelerator and the Divergence Algorithm, we were able to achieve gains up to 182x when compared to the unaccelerated HMMER software running on a general purpose CPU.

Keywords – Protein sequence; Hidden Markov Models; HMMER; Hardware accelerator; FPGA; Divergences.

I. INTRODUCTION

Due to the constant increase in the size of bio-sequence databases [1-2], a lot of effort has been put into optimizing and accelerating bio-sequence analysis algorithms. In the field of protein comparison, the Plan7-HMM Viterbi algorithm and the HMMER software developed by Eddy et al. [3] are among the most used tools for profile Hidden Markov Model (HMM) database search. Several optimization strategies for HMMER have been proposed over the years. MPI-HMMER [4] explores parallel execution in a cluster as well as software optimizations via the Intel-SSE2 instruction set. Other approaches like SledgeHMMER [5] and "HMMER on the Sun Grid" [6] provide web based search interfaces to either an optimized version of HMMER running on a web server or the Sun Grid, respectively. Other approaches such as ClawHMMER [7] and GPU-HMMER [8] implement GPU parallelization of the Viterbi algorithm, while achieving a better cost/benefit relation than the cluster approach.

Studies have shown that most of the processing time of the HMMER software is spent into processing non significant sequences [9]. Therefore, most authors have found useful to apply a first phase filter in order to discard poor scoring sequences prior to full processing. Some works apply heuristics, but the mainstream focuses in the use of FPGA-based accelerators [9-14] as this first phase filter. The filter retrieves the sequence score and if it is acceptable, full reprocessing of the sequence is done in software.

Our work proposes further acceleration of the algorithm by using the concept of divergence in which full reprocessing of the sequence after the FPGA accelerator is not needed, since the alignment only appears in specific parts of both the profile HMM model and the sequence. The proposed accelerator outputs the similarity score and the limits of the area of the Dynamic Programming (DP) matrices that contains the optimal alignment. The software then calculates only that small area of the DP matrices for the Viterbi algorithm and returns the same alignment as the unaccelerated software.

The rest of this work is organized as follows. In Section 2 we introduce the concept of protein families, profile HMMs, the Viterbi algorithm and the HMMER suite. Section 3 shows the related work in FPGA accelerators for the HMMER suite. In Section 4, we present the concept of divergences and explain its two main phases. Section 5 presents the proposed hardware architecture. In Section 6, we explain the VHDL implementation of the accelerator. In Section 7 we show the synthesis and performance results of the accelerator. Finally, in Section 8 we summarize the results of our work and present some future works.

II. PROFILE HMMs AND THE PLAN7-VITERBI ALGORITHM

As explained in [15], databases usually group similar proteins into protein families. These families have structural and/or functional similarities and are used to construct probabilistic models for the family in order to be able to identify and align new members to it. One of the most accepted probabilistic models is based on HMMs. It is called profile HMM because it groups the evolutionary statistics for all the family members "profiling" it. A profile HMM models the common similarities among all the sequences in a protein family as discrete states, each one corresponding to an evolutionary possibility such as amino acid insertions, deletions or matches between them.

The traditional profile HMM architecture proposed by Krogh et. al. [15] only consisted of Insert (I), Delete (D) and Match (M) states with transition probabilities between states and emission probabilities for each amino acid element of the sequence. HMMER [3] uses a modified architecture that in addition to the traditional M, I and D states includes flanking states that enable the algorithm to produce global or local alignments, with respect to the model or to the sequence, and also multiple hit alignments. Given a HMM modeling a protein family and a query sequence, HMMER computes the probability that the sequence belongs to the family, as a similarity score, and generates the resulting alignment in case the score is sufficiently good. To do so, it implements a well-

known DP algorithm called the Viterbi algorithm [16]. This algorithm calculates a set of DP matrices (corresponding to states M, I, and D) and vectors (corresponding to states N, B, E, C, and J) by means of a set of recurrence equations found in [10,12,13] and shown in Figure 1. Each state has transition probabilities to other states and the M and I states have emission probabilities in order to model the statistical behavior of the protein family. As a result, it finds the best (most probable) alignment and its score for the query sequence with the given model.

$$M(i,0) = I(i,0) = D(i,0) = -\infty \quad \forall\, 1 \le i \le n$$
$$M(0,j) = I(0,j) = D(0,j) = -\infty \quad \forall\, 0 \le j \le k$$

$$M(i,j) = em(M_j, s_i) + max \begin{cases} M(i-1,j-1) + tr(M_{j-1}, M_j) \\ I(i-1,j-1) + tr(I_{j-1}, M_j) \\ D(i-1,j-1) + tr(D_{j-1}, M_j) \\ B(i-1) + tr(B, M_j) \end{cases} \quad \forall\, 1 \le i \le n,$$

$$I(i,j) = em(I_j, s_i) + max \begin{cases} M(i-1,j) + tr(M_j, I_j) \\ I(i-1,j) + tr(I_j, I_j) \end{cases} \quad \forall\, 1 \le j \le k$$

$$D(i,j) = max \begin{cases} M(i,j-1) + tr(M_{j-1}, D_j) \\ D(i,j-1) + tr(D_{j-1}, D_j) \end{cases}$$

$$N(0) = 0$$
$$N(i) = N(i-1) + tr(N,N) \quad \forall\, 1 \le i \le n$$
$$B(0) = tr(N,B)$$
$$B(i) = max \begin{cases} N(i) + tr(N,B) \\ J(i) + tr(J,B) \end{cases} \quad \forall\ 1 \le i \le n$$
$$E(i) = max_{\,1 \le j \le k} (M(i,j) + tr(M_j, E)) \quad \forall\ 1 \le i \le n$$
$$J(0) = -\infty$$
$$J(i) = max \begin{cases} J(i-1) + tr(J,J) \\ E(i) + tr(E,J) \end{cases} \quad \forall\ 1 \le i \le n$$
$$C(0) = -\infty$$
$$C(i) = max \begin{cases} C(i-1) + tr(C,C) \\ E(i) + tr(E,C) \end{cases} \quad \forall\ 1 \le i \le n$$

$$similarity\_score = C(n) + tr(C,T)$$

Figure 1. Plan7-Viterbi algorithm recurrence equations, for a profile HMM with *k* nodes and sequence *s* of length *n.* In the figure *tr(state1,state2)* corresponds to the transition probability from state1 to state 2 and em($state_j$,S_i) corresponds to the probability of amino acid S_i to appear in $state_j$.

As the routine that implements the Viterbi algorithm in HMMER is the most time consuming of the search and comparison programs, most works [9-14,17] focus in accelerating its execution by proposing a pre-filter phase which only calculates the similarity score for the algorithm. Then, if the similarity score is good, the entire query sequence is reprocessed to produce the alignment. This paper presents first the related work in order to introduce the advances made in such acceleration and then proposes a new technique which not only accelerates the execution by pre-filtering the query sequences, but also reduces the number of calculations required to generate the alignment of the query sequence to the profile HMM.

III. RELATED WORK

In general, FPGA-based accelerators for the Viterbi algorithm are composed of processing elements (PEs), connected together in a systolic array to exploit either the algorithm parallelism, by eliminating the J state of the Plan7. As the typical profile HMMs would be too long for its implementation to fit into an FPGA, the entire sequence processing is divided into several passes [9-11,14]. First-In First-Out memories are included inside the FPGA implementation to store the necessary intermediary data between passes. Transition and emission probabilities for all the passes of the HMM are pre-loaded into block memories inside the FPGA to hide model turn around (transition probabilities reloading) when switching between passes. These memory requirements impose restrictions on the maximum PE number that can fit into the device, the maximum HMM size and the maximum sequence size.

Benkrid et al. [10] propose an array of 90 PEs, capable of comparing a 1024 element sequence with a 1440 nodes profile HMM. They eliminate the J state dependencies in order to exploit the dynamic programming parallelism and report a maximum performance of 9 GCUPS (Giga Cell Updates per Second). Maddimsetty et al. [9] enhances accuracy by reducing the precision error induced by the elimination of the J state, and proposes a two-pass architecture to detect and correct false negatives. Based on technology assumptions, they report an estimated maximum size of 50 PEs estimating at a clock frequency of 200MHz and a performance of 5 to 20 GCUPS. Jacob et al. [14] divide the PE into 4 pipeline stages estimating a maximum clock frequency of 180MHz and a throughput of 10 GCUPS. Their work also eliminates the J state. The proposed architecture was implemented in a Virtex 2 6000 FPGA and supports up to 68 PEs, a HMM with maximum length of 544 nodes and a maximum sequence size of 1024 amino acids. In Derrien et al. [13], a methodology to implement a pipeline inside the PE is outlined, based on the mathematical representation of the algorithm. Then a design space exploration is made for a Xilinx Spartan 3 4000, with maximum PE clock frequency of 66MHz and a maximum performance of about 1.3 GCUPS. Oliver et al. [11] implement the typical PE array without taking into account the J and the B state when calculating the score. They obtain an array of 72 PEs working at a clock rate of 74MHz, and an estimated performance of 3.95 GCUPS. In [17] a special functional unit is introduced to detect when the J state feedback loop is taken. Then a control unit updates the value for state B and instructs the PEs to recalculate the inaccurate values. The implementation was made in a Xilinx Virtex 5 110-T FPGA with a maximum of 25 PEs and operating at 130MHz. The reported performance is 3.2GCUPS. No maximum HMM length or pass number is reported in the article.

Walters et al. [12] implement a complete Plan7-Viterbi algorithm (including the J State) in hardware, by exploiting the inherent parallelism in processing different sequences against the same HMM at the same time. They include hardware acceleration into a cluster of computers, in order to further enhance the speedup. The implementation was made in a Xilinx Spartan 3 1500 board with a maximum of 10 PEs per node and a maximum HMM profile length of 256. The maximum clock speed for each PE is 70MHz and the complete system yields a performance of 700 MCUPS per cluster node.

Like all the designs discussed in this section, our design does not calculate the alignment in hardware, providing the score as output. Nevertheless, unlike the previous FPGA

proposals, our design also provides information that can be used by the software to significantly reduce the number of cells contained in the DP matrices that need to be recalculated. Therefore, besides the score, our design outputs also the Divergence Algorithm information, that will be used by the software to determine a band in the DP matrices where the actual alignment occurs. In this way, the software reprocessing time can be reduced and better overall speedups can be attained. This work also proposes the use of a more accurate performance measurement that includes not only the time spent calculating the sequence score and divergence, but also takes into account the time spent while reprocessing the sequences of interest found, which gives a clearer idea of the real gain achieved when integrating the accelerator to HMMER.

IV. DIVERGENCE ALGORITHM

The divergence concept was first introduced in [18] for pairwise DNA alignment in order to both enhance processing speed and reduce memory requirements. It reduces the number of DP cells that the software has to calculate by computing limits that instruct the algorithm where the alignment starts and ends. We adapted the divergence concept to the Plan7-Viterbi algorithm. Our Divergence Algorithm (DA) works in two main phases. The first phase is inserted into a simplified version of the Viterbi algorithm which eliminates data dependencies induced by the J state. We calculate the limits of the alignment expressed by its initial and final lines and superior and inferior divergences (IL, FL, SD and ID, respectively). These limits are computed as new DP matrices, by means of a new set of recurrence equations, for the M, I, D, E, and C states of the Viterbi algorithm. Figure 2 shows the main idea behind the divergence concept. The superior and inferior divergences represent how far the alignment departs from the main diagonal, in up and down directions, respectively. The DA calculates the limits of the alignment while it computes the similarity score leaving no space to error and generating the same results as the unbound calculations.

The DA's first phase was thought to be implemented in hardware because its implementation in software would increase the memory requirements and processing time as it introduces new DP matrices. Besides, the alignment limits computation does not create new data dependencies inside the Viterbi algorithm and can be performed in parallel with the similarity score calculation.The second phase takes the output data coming from the first phase. If the sequence's score is significant enough, it produces the alignment by processing only a small portion of the DP matrices, called the alignment region (AR), saving memory space and processing time. It is necessary to initialize only the cells above and to the left of the AR and to calculate only the cells inside it. Figure 2 shows the initialization process and the calculated cells inside the AR. In the next section we propose a hardware implementation of the first phase of DA. This implementation not only speeds-up the execution of the Viterbi algorithm by acting as a pre-filtering phase, but also minimizes the alignment generation time by reducing the number of DP cells that the software has to calculate for significant sequences as it returns the alignment limits for them to the second phase of the DA. The second phase of the DA is implemented as a modification inside HMMER's *hmmpfam* and *hmmsearch* programs.

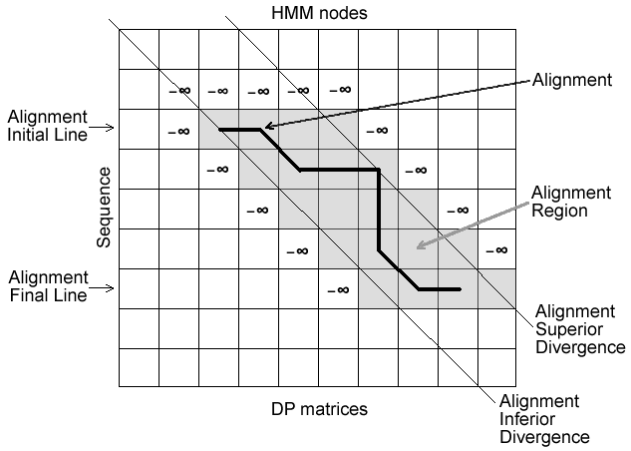


Figure 2. Divergence concept: alignment limits, initialized cells (with -∞) and calculated cells (AR)

V. PROPOSED ARCHITECTURE

In this section we describe the architecture and implementation of a simplified Plan7 Viterbi algorithm, without the J state, and the insertion of the DA's first phase in order to further enhance speed up of the whole system when compared to the unaccelerated software. The architecture consists of an array of interconnected PEs, each one calculating one column of the dynamic programming matrices. The system also has RAM memories to store emission and transition probabilities and FIFOs to store intermediate results when dealing with long profile HMMs similar to those analyzed in Section 3. Figure 3 shows the main building blocks of the system.

The PE consists of a score stage which calculates the M, I, D and E scores, and a divergence stage which calculates the alignment limits for the current sequence. A pass controller [14] is included inside each PE, programmed to recognize two especial characters (one for a pass end and another for a sequence end) and increment or clear a pass register. Additional modules are included for the B and C vector calculations as well. Block RAM memories are inserted to store emission probabilities, transition probabilities and intermediate results between passes. A pass controller was included inside each memory in order to control address offsets and to keep track of the current pass.

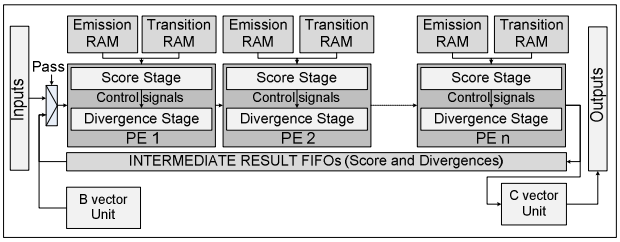


Figure 3. Block diagram of the proposed system: inputs are the protein sequence elements and outputs are the alignment score and the alignment limits

A. SCORE STAGE

This stage calculates the scores for the simplified Viterbi algorithm. Each PE represents an individual HMM node, and

calculates the scores as each element of the sequence passes through. The PE's inputs are the scores calculated for the current element in a previous HMM node, and the PE's outputs are the scores for the current sequence element in the current node. Figure 4 shows the score calculation stage of the PE. The signals ctlM, ctlI, ctlD and ctlXE produced in this stage are used in the divergence stage and represent which input was chosen by the maximum operator in the M, I, D and E score calculations, respectively.

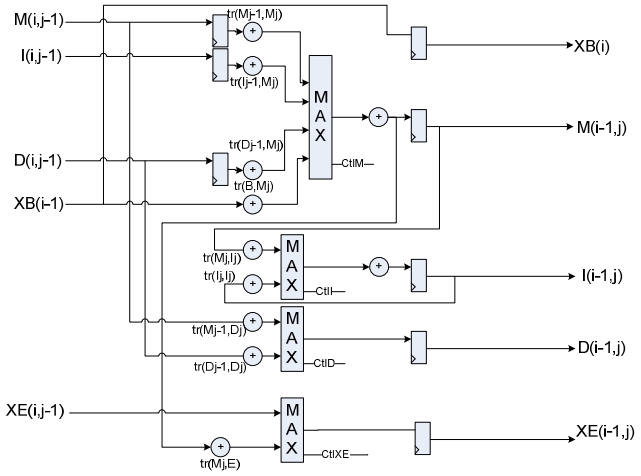


Figure 4. Score calculating stage of the PE

B. *DIVERGENCE STAGE*

This stage calculates the alignment limits for the current sequence element. The stage inputs are the previous node alignment limits for the current sequence element and the outputs are the calculated alignment limits for the current element. The outputs depend directly on the score stage of the PE and are controlled by the ctlM, ctlI, ctlD and ctlXE signals. The divergence stage also requires the current sequence element index, in order to calculate the alignment limits. Figures 5 and 6 show the DA implementation for the I and D states, respectively. Figure 7 shows the DA implementation for the M and E states. The Base parameter is the position of the PE in the systolic array and the #PE parameter is the total number of PEs in the current system implementation. These parameters are used to initialize the divergence stage registers according to the current pass and ensure the limits are calculated correctly.

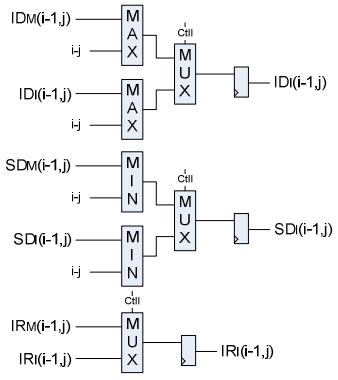


Figure 5. Divergence calculating stage for I state

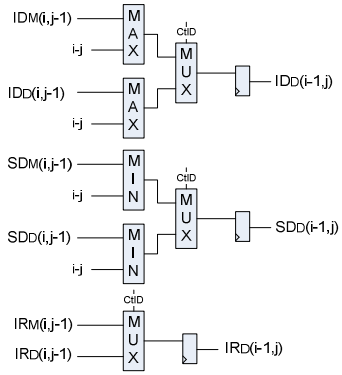


Figure 6. Divergence calculating stage for D state

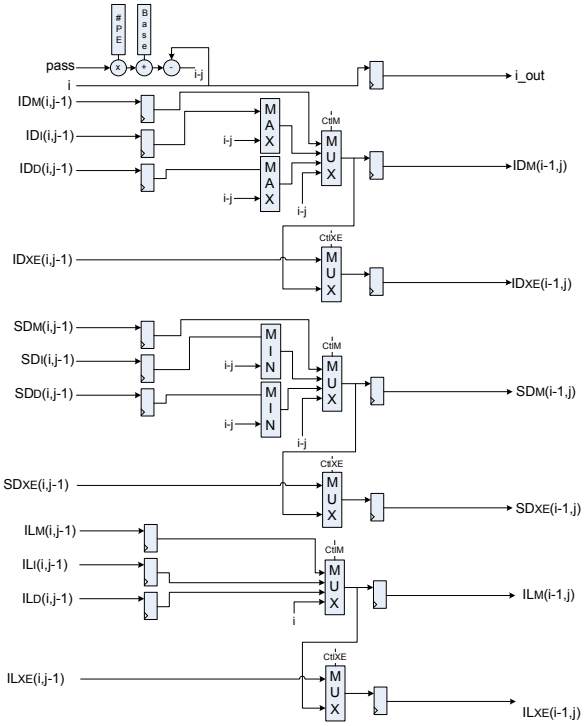


Figure 7. Divergence calculating stage for M and E states

The C vector module must include the computation of the output alignment limits that will feed the second phase of the DA. The C module is shown in Figure 8.

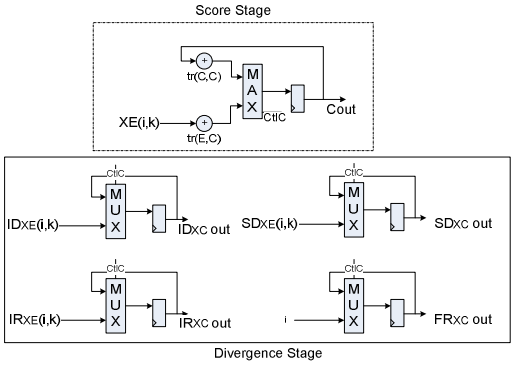


Figure 8. Modified C module to include the DA. The CtlC line is generated by the Score Stage to control the output of the Divergence Stage.

TABLE 1. AREA AND SPEED SYNTHESIS RESULTS

#PEs	Max passes	Max HMM nodes	Max sequence size	Combinational ALUs	Dedicated registers	Memory bits	% Logic	Max clock frequency (MHz)
25	25	625	8192	31738	18252	2609152	25	71
50	25	1250	8192	59750	35294	3121152	49	71
75	25	1875	8192	93132	52520	3663152	75	69
85	25	2125	8192	103940	59285	3837952	84	67

VI. HARDWARE IMPLEMENTATION

The entire system is organized as a linear array of PEs, each one connected to its two immediate neighbors (left and right). Each PE represents one node of the profile HMM and computes the scores for the current sequence element in the current node as well as the alignment limits. Emission and transition probabilities are stored in Dual Port RAM memories, loaded at the beginning of the execution with the profile HMM we are comparing the query sequence against to. As nine transition probabilities must be available all the time for each PE, the transition memories are connected to a register bank. This register bank has two input ports and nine output ports, and thus can be fully loaded in only 5 clock cycles as its input ports come from the Dual Port RAM memory holding the transition probabilities. Transition memories also have a small controller inside that allows the memory to recognize the end of pass and end of sequence character, and automatically load the register bank with the right transition probabilities for each pass. As intermediate results must be stored to support multiple passes, FIFO memories for each DP matrix are present. The B and C state score units are placed outside from the first and last PEs in order to have an easily modifiable and homogeneous PE.

The complete system was implemented in VHDL and mapped to an Altera Stratix II 180 device. Several configurations were explored to maximize the number of HMM nodes, the number of PEs and the maximum sequence length. In order to do design space exploration, we developed a parametrable VHDL code, in which we can modify the PE word size, the number of PEs of the array and the memories size. For the current implementation, we obtained a maximum frequency of 67MHz after constraining the design time requirements in the Quartus II tool to optimize the synthesis for speed instead of area. Further works will include pipelining the PE to achieve better performance in terms of clock frequency. Table 1 shows the synthesized configurations and their resource utilization.

VII. PERFORMANCE RESULTS

The proposed architecture not only enhances software execution by applying a first phase pre-filter to the HMMER software, but also provides a means to limit data processing to a small portion of the DP matrices. Because of this, the speedup of the solution must be measured taking into account the performance achieved by the hardware pre-filter as well as the saved software processing time by only recalculating the scores inside the alignment region. Execution time is measured separately for hardware by measuring its real throughput rate (including loading time and inter-pass delays), and for software by computing the savings when calculating the scores of the divergence-limited region of the DP matrices. Experimental tests were conducted over all the 10340 profile HMMs for the PFam-A protein database [2]. Searches were made using 4 sets of 2000 randomly sampled protein sequences from the Uniprot.Sprot protein database [1] and only significantly scoring sequences were considered to DA second phase (reprocessing in software). To find out which sequences from the sequence set were significant, we utilized a user defined threshold and relaxed it to include the greatest possible number of sequences.

A. HMMER'S PERFORMANCE

To obtain HMMER performance in a typical work environment we used a platform composed of an Intel Centrino Duo processor running at 1.8GHz, 4GB of RAM memory and a 250GB Hard Drive and HMMER was compiled to optimize execution time. We also modified the hmmsearch program in order to get the execution times only for the Viterbi algorithm, as our main goal was to accelerate it. From the experiments we obtained a performance of 23.157 MegaCUPS. We also obtained the execution times shown in Table 2 for one of the sets of random sequences and 6 PFam-A profile HMMs.

TABLE 2. HMMER'S EXECUTION TIMES

Sequence set elements (entire set)	HMM nodes	HMMER execution time (sec)	Viterbi execution time (sec)
700218	788	24.40	23.37
	10	0.42	0.38
	226	6.76	6.72
	337	10.26	9.84
	2295	81.23	62.25
	901	27.41	26.33

A. HARDWARE'S PERFORMANCE

We formulated the exact equation for the proposed accelerator performance, taking into account all possible delays, including systolic array data filling and consuming, profile HMM probabilities loading into RAM memories and probabilities reloading delays when switching between passes. In order to validate this equation we developed a *testbench* to execute all the test sets. Data transmission is not considered into the formula due to the fact that we will be using at a minimum a PCIe interface [19], whose data transmission rates are well above the maximum required for the system (130MBps).Let m_i be the number of nodes of the current HMM, S_j be the size of the sequence being processed, n be the number of PEs in hardware, and f be the maximum system frequency. Then the throughput T_{hw} of the system (measured in CUPS) and the time $t_{h(i,j)}$ the accelerator takes to process one sequence are fully described by Equations (1) and (2), respectively. In these equations, $25n\lceil m_i / n \rceil$ are the number of cycles spent loading the current HMM into memory, 2n are the

cycles spent filling and emptying the array of PEs, $(S_j+6)\lceil m_i/n \rceil$ are the cycles spent while processing the current sequence, 3 are the cycles spent loading the special transitions, and $S_j m_i$ are the number of cells that the unaccelerated algorithm has to calculate to process the current sequence with the current HMM.Table 3 shows the obtained throughputs and the execution times for a hardware accelerator with 85 PEs.

$$T_{hw} = \frac{\sum_{i=1}^{\#HMMs}\sum_{j=1}^{\#Seqs} S_j m_i}{\sum_{i=1}^{\#HMMs}\left[\left(\sum_{j=1}^{\#Seqs}(S_j+6)\left\lceil\frac{m_i}{n}\right\rceil\right)+25n\left\lceil\frac{m_i}{n}\right\rceil+3+2n\right]} * f \quad (1)$$

$$t_{h_{(i,j)}} = \frac{(S_j+6)\left\lceil\frac{m_i}{n}\right\rceil+25n\left\lceil\frac{m_i}{n}\right\rceil+3+2n}{f} \quad (2)$$

TABLE 3. EXECUTION TIME AND PERFORMANCE FOR THE HARDWARE ACCELERATOR

#PEs	Sequence set elements (entire set)	HMM nodes	T_{hw} (GigaCUPS)	t_h (sec)
85	700218	788	5.4206	0.0989
		10	0.6877	0.0099
		226	5.1818	0.0297
		337	5.7953	0.0396
		2295	5.8472	0.2671
		901	5.6345	0.1088

A. DIVERGENCE'S SECOND PHASE PERFORMANCE

After simulating the hardware accelerator, we fed the modified HMMER software with the score and alignment limits data for the test HMMs and the significant sequences in the sequence sets. Table 4 shows the total HMMER processing time and the execution time obtained when only calculating the cells inside the AR and obtaining the resulting alignment.

TABLE 4. EXECUTION TIME FOR THE MODIFIED HMMER SOFTWARE PROCESSING THE AR

Sequence set elements (entire set)	HMM nodes	HMMER execution time (sec)	AR's processing time (sec)
700218	788	24.40	0.0537
	10	0.42	0.0009
	226	6.76	0.0149
	337	10.26	0.0226
	2295	81.23	0.1787
	901	27.41	0.0603

B. SYSTEM'S PERFORMANCE

To obtain the system performance we added the execution time of the accelerator and the reprocessing time for the alignment regions of the significant sequences and compared it to the total HMMER processing time for the same set. We obtained performances up to 5.8 GCUPS when considering only accelerator throughput, which means a gain of up to 254 times the performance of the unaccelerated HMMER. When including also the reprocessing times, we obtained a gain of up to 182 times the performance of the unaccelerated software, which represents a significant reduction in execution time.

VIII. CONCLUSIONS

In this paper we introduced the Divergence Algorithm that enables the implementation of a hardware accelerator for the *hmmsearch* and *hmmpfam* programs of the HMMER suite. We proposed an accelerator architecture which acts as a pre-filtering phase and uses the divergence concept to avoid the full reprocessing of the sequence in software. We also introduced a more accurate performance measurement strategy when evaluating HMMER hardware accelerators, which not only includes the time spent on the pre-filtering phase or the hardware throughput, but also includes reprocessing times for the significant sequences found in the process. We implemented our accelerator in VHDL, obtaining performance gains of up to 182 times the performance of the HMMER software. For future works we also intend to implement the Divergence Algorithm in a complete version of the Plan7-Viterbi algorithm, to observe the performance gains achieved when doing so.

ACKNOWLEDGMENTS

The authors would like to acknowledge the National Council for Technologic and Scientific development (CNPq), the National Microelectronics Program (PNM), the Studies and Projects Financial Fund (FINEP) and the Brazilian Millennium Institute (NAMITEC) for funding this work.

REFERENCES

[1] The Universal Protein Resource (UniProt. http://www.uniprot.org. Last access: June 2009.
[2] Sanger's Institute PFAM Protein Sequence Database. http://pfam.sanger.ac.uk/. Last access: May 2009.
[3] HMMER: Biosequence analysis using profile hidden Markov models. http://hmmer.janelia.org, 2006.
[4] Darole, R., Walters, J. P., and Chaudhary, V. (2008) "Improving MPI-HMMER's Scalability With Parallel I/O." Technical Report. Department of Computer Science and Engineering. University of Buffalo.
[5] Chukkapalli, G., Guda, C., and Subramaniam, S. (2004) "SledgeHMMER: A Web Server for Batch searching the PFam Database", Nucleic Acids Research, 32, 542-544.
[6] HMMER on the Sun Grid Project. https://hmmer.dev.java.net/. Last access, July 2009.
[7] Horn, D. R., Houston, M., and Hanrahan, P. (2005) "ClawHMMER: A Streaming HMMer-Search Implementation." In: Conference on Supercomputing, 11-19.
[8] GPU-HMMER. http://mpihmmer.org/userguideGPUHMMER.htm. Last Access, July 2009.
[9] Maddimsetty, R. P., Buhler, J., Chamberlain, R. D., Franklin, M. A., Harris B. (2006) "Accelerator design for protein sequence HMM search." In: Conference on Supercomputing.
[10] Benkrid, K., Velentzas, P., and Kasap, S. (2008) "A High Performance Reconfigurable Core for Motif Searching Using Profile HMM." In: Conference on Adaptive Hardware Systems, 285-292.
[11] Oliver, T., Schmidt, B., Jakop, Y., and Maskell, D. "High Speed Biological Sequence Analysis with Hidden Markov Models on Reconfigurable Platforms." In: IEEE Transactions on Information Technology in Biomedicine, in press.
[12] Walters, J. P., Meng, X., Chaudhary, V., Oliver, T., Yeow, L. Y., Schmidt, B., Nathan, D., and Landman, J. (2007) "MPI-HMMER-Boost: Distributed FPGA Acceleration." In: Journal of VLSI Signal Processing Systems, 48(3), 223-238.
[13] Derrien, S. and Quinton, P. (2007) "Parallelizing HMMER for Hardware Acceleration on FPGAs." In: Conference on Application-specific Systems, Architectures and Processors, 10-17.
[14] Jacob, A. C., Lancaster, J. M., Buhler, J. D., and Chamberlain, R. D. (2007) "Preliminary results in accelerating profile HMM search on FPGAs." In: International Symposium on Parallel and Distributed Processing.
[15] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. (2008) Biological Sequence Analysis Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press.
[16] Rabiner, L. R. (1989) "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition". In: Proceedings of the IEEE, 77(2), 257-286.
[17] Sun, Y., Li, P., Gu, G., Wen, Y., Liu, Y., and Liu, D. (2009) "HMMer acceleration using systolic array based reconfigurable architecture." In: Symposium on Field Programmable Gate Arrays, 282-282.
[18] Batista, R. B., Boukerche, A., and Melo, A. C. (2008) "A parallel strategy for biological sequence alignment in restricted memory space." In: Journal of Parallel and Distributed Computing, 68(4), 548-561.
[19] Dell PCI Express Technology. http://www.dell.com/content/topics/global.aspx/vectors/en/2004_pciexpress. Last access, July 2009.

A Protein Sequence Analysis Hardware Accelerator based on Divergences

Juan Fernando Eusse, Nahri Balesdent Moreano, Alba Cristina Magalhaes Alves de Melo, *Senior Member, IEEE*, and Ricardo Pezzuol Jacobi, *Member, IEEE*

Abstract— Hidden Markov Models are a powerful tool for protein organization and identification because they allow identifying and classifying highly representative structures and functional units inside the amino acid chains that form them. The Viterbi algorithm is one of the most used Dynamic Programming algorithms in protein comparison and identification. It uses Hidden Markov Models (HMMs), and is implemented inside the widely used HMMER software suite. Due to the exponential growth in the size of protein databases, the necessity to accelerate software execution to reduce comparison and search times rose. Most of the works focus on the implementation of a hardware accelerator that acts as a pre-filter stage in the comparison process. This stage discards poorly aligned sequences with a low similarity score and forwards sequences with good similarity scores to software, where they are reprocessed to generate the sequence alignment. This work presents the concept of divergence, in which the region of interest of the Dynamic Programming matrices is delimited, reducing the number of calculations needed to perform a comparison. Therefore, full reprocessing of the sequence is not needed after the pre-filter stage, as only the interest region of the Dynamic Programming matrices is needed to be reprocessed in order to generate the sequence alignment with the HMM. Furthermore, this paper proposes and evaluates the implementation of a hardware accelerator, which not only calculates the similarity score of a query sequence, but also generates the divergence information. With the implementation of our accelerator including the divergence algorithm, we got maximum gain of up to 182x when compared to unaccelerated software. Also, a new performance measurement strategy is introduced in this work, which not only takes into account the acceleration achieved by the hardware, but also the re-processing stage that follows hardware and that is required to generate the alignment.

Index Terms—Keywords should be taken from the taxonomy (http://www.computer.org/mc/keywords/keywords.htm). Keywords should closely reflect the topic and should optimally characterize the paper. Use about four key words or phrases in alphabetical order, separated by commas.

1 INTRODUCTION

Protein Sequence comparison and analysis is a repetitive task in the field of Molecular Biology, as is needed by biologists to predict or determine the function, structure and evolutional characteristics of newly discovered Protein Sequences. During the last decade, technological advances had made possible the identification of a vast number of new proteins that have been introduced to the existing protein databases [1], [2]. With the exponential growth of these databases, the execution times of the protein comparison algorithms also grew exponentially [3], and the necessity to accelerate the existing software rose in order to speedup research.

- *Juan Fernando Eusse is with the Electrical Engineering Department, University of Brasilia, Brasilia/DF, Brazil. E-mail: eusse@microe.udea.edu.co.*
- *Nahri Balesdent Moreano is with the Department of Statistics and Computation, University of Mato Grosso do Sul, Campo Grande/MS, Brazil. E-mail: nahri@dct.ufms.br.*
- *Alba Cristina Magalhaes Alves de Melo is with the Computer Science Department, University of Brasilia, Brasilia/DF, Brazil. E-mail: albamm@cic.unb.br.*
- *Ricardo Pezzuol Jacobi is with the Computer Science Department, University of Brasilia, Brasilia/DF, Brazil. E-mail: jacobi@unb.br.*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

The HMMER program suite [4] is among one of the most used programs for sequence comparison. HMMER takes multiple sequence alignments of similar protein sequences grouped into protein families and builds Hidden Markov Models (HMMs) [5] of them. This is done to estimate statistically the evolutionary relations that exist between different members of the protein family, and to ease the identification of new family members with a similar structure or function. HMMER then takes unknown input sequences and compares them against the generated HMM models of protein families (profileHMM) via the Viterbi algorithm (See Section 2), to generate both a similarity score and an alignment for the input sequences.

As the Viterbi routine is the most time consuming part of the HMMER programs, multiple attempts to optimize and accelerate it have been made. MPI-HMMER [6] explores parallel execution in a cluster as well as software optimizations via the Intel-SSE2 instruction set. Other approaches like SledgeHMMER [7] and "HMMER on the Sun Grid" [8] provide web based search interfaces to either an optimized version of HMMER running on a web server or the Sun Grid, respectively. Other approaches such as ClawHMMER [9] and GPU-HMMER [10] implement GPU parallelization of the Viterbi algorithm, while achieving a better

cost/ benefit relation than the cluster approach.

Studies have also shown that most of the processing time of the HMMER software is spent into processing poor scoring (non significant) sequences [11], and most authors have found useful to apply a first phase filter in order to discard poor scoring sequences prior to full processing. Some works apply heuristics [12], but the mainstream focuses in the use of FPGA-based accelerators [3], [11], [13], [14], [15], [16] as this first phase filter. The filter retrieves the sequence's similarity score and if it is acceptable instructs the software to reprocess the sequence in order to generate the corresponding alignment.

Our work proposes further acceleration of the algorithm by using the concept of divergence in which full reprocessing of the sequence after the FPGA accelerator is not needed, since the alignment(s) only appear in specific parts of both the profile HMM model and the sequence. The proposed accelerator outputs the similarity score and the coordinates of a band that defines the area of the Dynamic Programming (DP) matrices that contains the optimal alignment. The software then calculates only that small of the DP matrices for the Viterbi algorithm and returns the same alignment as the unaccelerated software.

This work is organized as follows. In Section 2 we clarify some of the concepts of protein sequences, protein families and profileHMMs. In Section 3 we present the related work in FPGA based HMMER accelerators. Section 4 introduces the concept of divergences and their use in the acceleration of the Viterbi Algorithm. Section 5 shows the proposed hardware architecture. Section 6 presents the metrics used to analyze the performance of the system. In section 7 we show implementation and performance results and we compare the obtained results with the existing works. Finally, in Section 8 we summarize the results of the work and show the future works to be done in the field.

2 Protein Sequence Comparison

2.1 Protein Sequences, Protein Families and Profile HMMs

Proteins are basic elements that are present in every living organism. They may have several important functions such as: catalyzing chemical reactions and signaling if a gene must be expressed, among others. Essentially, a protein is a chain of amino acids. In the nature, there are 20 different amino acids, that compose the alphabet Σ={A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y} [17].

A protein family is defined to be a set of proteins that have similar function, or have similar 2D/ 3D structure or have a common evolutionary history [17]. Therefore, a newly sequenced protein is often compared to several known protein families, in search of similarities. This comparison usually aligns the protein sequence to the representation of a protein family. This representation can be a profile, a consensus sequence or a signature [18]. In this paper, we will only deal with profile representations, which are based on multiple sequence alignments.

Given a multiple sequence alignment, a profile specifies, for each column, the frequency that each amino acid appears in the column. If the comparison sequence-profile results in high similarity, the protein sequence is usually identified to be member of the family. This identification is a very important step towards determining the function or/ and structure of the protein sequence.

One of the most accepted probabilistic models to do sequence-profile comparisons is based on Hidden Markov Models (HMMs). It is called profile HMM because it groups the evolutionary statistics for all the family members "profiling" it. A profile HMM models the common similarities among all the sequences in a protein family as discrete states, each one corresponding to an evolutionary possibility such as amino acid insertions, deletions or matches between them. The traditional profile HMM architecture proposed by Krogh et. al. [5] consisted of Insert (I), Delete (D) and Match (M) states.

The HMMER suite, developed by Eddy [4], is a widely used software implementation of profile HMMs for biological sequence analysis, composed of several programs. In particular, the program hmmsearch searches a sequence database for matches to an HMM, while the program hmmpfam searches an HMM database for matches to a query sequence.

HMMER uses a modified HMM architecture that in addition to the traditional M, I and D states includes flanking states that enable the algorithm to produce global or local alignments, with respect to the model or to the sequence, and also multiple hit alignments. The Plan7 architecture used by HMMER is shown in Fig. 1.

Usually, there is one match state for each consensus column in the multiple alignment. Each M state aligns to (emits) a single residue, with a probability score that is determined by the frequency that residues have been observed in the corresponding column of the multiple alignment. Therefore, each M state has 20 probabilities for scoring the 20 amino acids.

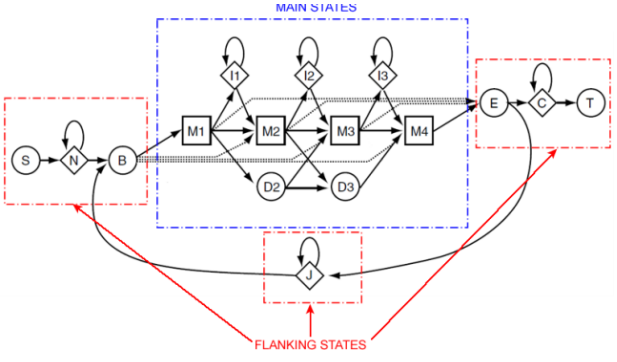


Fig. 1: Plan7 architecture used by HMMER [3]

The insertion (I) and deletion (D) states model gapped alignments, that is, alignments including residue insertions and deletions. Each I state also has 20 probabilities for scoring the 20 amino acids. The group of M, I, and D states corresponding to the same position in the multiple alignment is called a node of the HMM. Besides the emission probabilities, there are transition probabilities associated to each transition from one state to another.

2.2 Viterbi Algorithm

Given a HMM modeling a protein family and a query sequence, HMMER computes the probability that the sequence belongs to the family, as a similarity score, and generates the resulting alignment in case the score is sufficiently good. To do so, it implements a well-known DP algorithm called the Viterbi algorithm [19]. This algorithm calculates a set of DP score matrices (corresponding to states M, I, and D) and vectors (corresponding to states N, B, E, C, and J) by means of a set of recurrence. As a result, it finds the best (most probable) alignment and its score for the query sequence with the given model.

Fig. 2 shows the Viterbi algorithm recurrence equations for aligning a sequence of length n to a profile HMM with k nodes. In these equations, M(i,j) is the score of the best path aligning subsequence s1…si to the submodel up to state Mj, and I(i,j) and D(i,j) are defined similarly. The emission probability of the amino acid si at state1 is denoted by em(state1,si), while tr(state1,state2) represents the transition cost from state1 to state2. The similarity score of the best alignment is given by C(n)+ tr(C,T).

$$M(i,0) = I(i,0) = D(i,0) = -\infty \quad \forall\, 1 \le i \le n$$
$$M(0,j) = I(0,j) = D(0,j) = -\infty \quad \forall\, 0 \le j \le k$$

$$M(i,j) = em(M_j, s_i) + max \begin{cases} M(i-1,j-1) + tr(M_{j-1}, M_j) \\ I(i-1,j-1) + tr(I_{j-1}, M_j) \\ D(i-1,j-1) + tr(D_{j-1}, M_j) \\ B(i-1) + tr(B, M_j) \end{cases} \quad \forall\, 1 \le i \le n,$$

$$I(i,j) = em(I_j, s_i) + max \begin{cases} M(i-1,j) + tr(M_j, I_j) \\ I(i-1,j) + tr(I_j, I_j) \end{cases} \quad \forall\, 1 \le j \le k$$

$$D(i,j) = max \begin{cases} M(i,j-1) + tr(M_{j-1}, D_j) \\ D(i,j-1) + tr(D_{j-1}, D_j) \end{cases}$$

$$N(0) = 0$$
$$N(i) = N(i-1) + tr(N,N) \quad \forall\, 1 \le i \le n$$
$$B(0) = tr(N,B)$$
$$B(i) = max \begin{cases} N(i) + tr(N,B) \\ J(i) + tr(J,B) \end{cases} \quad \forall\ 1 \le i \le n$$
$$E(i) = max_{\,1 \le j \le k} (M(i,j) + tr(M_j, E)) \quad \forall\ 1 \le i \le n$$
$$J(0) = -\infty$$
$$J(i) = max \begin{cases} J(i-1) + tr(J,J) \\ E(i) + tr(E,J) \end{cases} \quad \forall\ 1 \le i \le n$$
$$C(0) = -\infty$$
$$C(i) = max \begin{cases} C(i-1) + tr(C,C) \\ E(i) + tr(E,C) \end{cases} \quad \forall\ 1 \le i \le n$$
$$similarity\_score = C(n) + tr(C,T)$$

Fig. 2: Plan7-Viterbi algorithm recurrence equations, for a profile HMM with k nodes and sequence s of length n.

The function that implements the Viterbi algorithm in the HMMER suite is the most time consuming of the *hmmsearch* and *hmmpfam* programs of the suite. Therefore, most works [3], [11], [13], [14], [15], [16], [20] focus in accelerating its execution by proposing a pre-filter phase which only calculates the similarity score for the algorithm. Then, if the similarity score is good, the entire query sequence is reprocessed to produce the alignment. This paper presents first the related work in order to introduce the advances made in such acceleration and then proposes a new technique which not only accelerates the execution by pre-filtering the query sequences, but also reduces the number of calculations required to generate the alignment of the query sequence to the profile HMM.

Fig. 3 shows a profile HMM with 4 nodes representing a multiple sequence alignment. The transition scores are shown in the figure, labeling the state transitions. The emission scores for the M and I states are shown in Table 1.

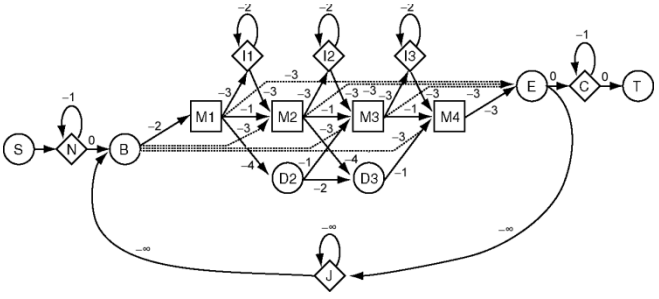


Fig. 3: A profile HMM with 4 nodes and the transition scores

TABLE 1
EMISSION SCORES OF AMINO ACIDS FOR MATCH AND INSERT STATES OF PROFILE HMM OF FIG. 3.

State	A	C	D	E	F, I, L, M, V, W	G, K, P, S	H, Q, R, T	Y
M_1	7	–1	–1	1	–1	2	1	–1
M_2	–1	9	–1	1	–1	2	1	–1
M_3	–1	–1	8	2	–1	2	1	–1
M_4	–1	–1	3	9	–1	2	1	–1
I_1	–1	–1	0	1	–1	0	1	2
I_2	–1	–1	0	1	–1	0	1	2
I_3	–1	–1	0	1	–1	0	1	2

Fig. 4 shows the score matrices and vectors computed by the Viterbi algorithm, while aligning the query sequence ACYDE to the profile HMM given in Fig. 3. The best alignment has the similarity score of 25 and corresponds to the path (S,–) → (N,–) → (B,–) → (M1,A) → (M2,C) → (I2,Y) → (M3,D) → (M4,E) → (E,–) → (C,–) → (T,–).

3 RELATED WORK

In general, FPGA-based accelerators for the Viterbi algorithm are composed of processing elements (PEs), connected together in a systolic array to exploit either the algorithm parallelism, by eliminating the J state of the Plan7. As the typical profile HMMs would be too long for its implementation to fit into an FPGA, the

	N	B	M					I					D					E	J	C
			0	1	2	3	4	0	1	2	3	4	0	1	2	3	4			
–	0	0	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞
A	−1	−1	−∞	−5	−4	−4	−4	−∞	−∞	−∞	−∞	−∞	−∞	−∞	1	−1	−3	2	−∞	2
C	−2	−2	−∞	−4	13	−1	−3	−∞	1	−8	−8		−∞	−∞	−8	9	7	10	−∞	10
Y	−3	−3	−∞	−5	−3	11	7	−∞	1	12	2		−∞	−∞	−9	−7	7	8	−∞	9
D	−4	−4	−∞	−6	−3	17	13	−∞	−1	10	8		−∞	−∞	−10	−7	13	14	−∞	14
E	−5	−5	−∞	−5	−3	9	25	−∞	−2	9	15		−∞	−∞	−9	−7	5	25	−∞	25

Fig. 4: Score matrices and vectors of the Viterbi algorithm for the comparison of the sequence ACYDE against the profile HMM of Fig. 3.

entire sequence processing is divided into several passes [3], [11], [13], [14]. First-In First-Out memories are included inside the FPGA implementation to store the necessary intermediary data between passes. Transition and emission probabilities for all the passes of the HMM are pre-loaded into block memories inside the FPGA to hide model turn around (transition probabilities reloading) when switching between passes. These memory requirements impose restrictions on the maximum PE number that can fit into the device, the maximum HMM size and the maximum sequence size.

Benkrid et al. [13] propose an array of 90 PEs, capable of comparing a 1024 element sequence with a 1440 nodes profile HMM. They eliminate the J state dependencies in order to exploit the dynamic programming parallelism, and calculate one cell element per clock cycle in each PE, reporting a maximum performance of 9 GCUPS (Giga Cell Updates per Second). Their systolic array was synthesized into a Virtex 2 Pro FPGA with a 100MHz clock frequency.

Maddimsetty et al. [11] enhances accuracy by reducing the precision error induced by the elimination of the J state, and proposes a two-pass architecture to detect and correct false negatives. Based on technology assumptions, they report an estimated maximum size of 50 PEs estimating at a clock frequency of 200MHz, and supposing a performance of 5 to 20 GCUPS.

Jacob et al. [3] divide the PE into 4 pipeline stages, in order to increase the maximum clock frequency up to 180MHz and the throughput up to 10 GCUPS. Their work also eliminates the J state. The proposed architecture was implemented in a Xilinx Virtex 2 6000 and supports up to 68 PEs, a HMM with maximum length of 544 nodes and a maximum sequence size of 1024 amino acids.

In Derrien et al. [16], a methodology to implement a pipeline inside the PE is outlined, based on the mathematical representation of the algorithm. Then a design space exploration is made for a Xilinx Spartan 3 4000, with maximum PE clock frequency of 66MHz and a maximum performance of about 1.3 GCUPS.

Oliver et al. [14] implement the typical PE array without taking into account the J state when calculating the score. They obtain an array of 72 PEs working at a clock rate of 74MHz, and an estimated performance of 3.95 GCUPS.

In [20] a special functional unit is introduced to detect when the J state feedback loop is taken. Then a control unit updates the value for state B and instructs the PEs to recalculate the inaccurate values. The implementation was made in a Xilinx Virtex 5 110-T FPGA with a maximum of 25 PEs and operating at 130MHz. The reported performance is 3.2GCUPS. No maximum HMM length or pass number is reported in the article.

Walters et al. [15] implement a complete Plan7-Viterbi algorithm in hardware, by exploiting the inherent parallelism in processing different sequences against the same HMM at the same time. Their PE is slightly more complex than those of other works as it includes the J state in the score calculation process. They include hardware acceleration into a cluster of computers, in order to further enhance the speedup. The implementation was made in a Xilinx Spartan 3 1500 board with a maximum of 10 PEs per node and a maximum HMM profile length of 256. The maximum clock speed for each PE is 70MHz and the complete system yields a performance of 700 MCUPS per cluster node, in a cluster comprised of 10 worker nodes. As any of the other analyzed works, its only output is the sequence score, and for the trace back, a complete reprocessing of the sequence has to be done in software.

Like all the designs discussed in this section, our design does not calculate the alignment in hardware, providing the score as output. Nevertheless, unlike the previous FPGA proposals, our design also provides information that can be used by the software to significantly reduce the number of cells contained in the DP matrices that need to be recalculated. Therefore, besides the score, our design outputs also the Divergence Algorithm information, that will be used by the software to determine a band in the DP matrices where the actual alignment occurs. In this way, the software reprocessing time can be reduced and better overall speedups can be attained.

This work also proposes the use of a more accurate performance measurement that includes not only the time spent calculating the sequence score and divergence, but also takes into account the time spent while reprocessing the sequences of interest found, which gives a clearer idea of the real gain achieved when integrating the accelerator to HMMER.

4 Divergence Algorithm

The divergence concept was first introduced by Batista et. al. [21], and is an exact variation of the Smith-Waterman algorithm for pairwise local alignment of DNA sequences. Their goal was to obtain the alignment of huge biological sequences, handling the quadratic space and time complexity of the Smith-Waterman algorithm. Therefore, they used parallel processing in a

cluster of processors to reduce execution time, and exploited the divergence concept to reduce memory requirements.

Initially, the whole similarity matrix is calculated in linear space. This phase of the algorithm outputs the highest similarity score and the coordinates in the similarity matrix that define the area that contains de optimal alignment. These coordinates were called superior and inferior divergences. To obtain the alignment itself using limited memory space, they recalculate the similarity matrix, but this time only the cells inside the limited area need to be computed and stored.

We adapted the divergence concept to the Plan7-Viterbi algorithm. Given the DP matrices of the Viterbi algorithm, the limits of the best alignment are expressed by its initial and final rows and superior and inferior divergences (IR, FR, SD, and ID, respectively). The initial and final rows indicate the row of the matrices where the alignment starts and ends. The superior and inferior divergences represent how far the alignment departs from the main diagonal, in up and down directions, respectively. The main diagonal has divergence 0, the diagonal immediately above it has divergence −1, the next one −2, and so on. Analogously, the diagonals below the main diagonal have divergences +1, +2, and so on. These divergences are calculated as the difference i − j between the row (i) and column (j) coordinates of the matrix cell.

Fig. 5 shows the main ideas behind the divergence concept. Given a profile HMM with k nodes and a query sequence of length n, the figure shows the DP matrices (represented as only one matrix, for clarity) of the Viterbi algorithm. The best alignment of the sequence to the HMM is a path (shown in a thick line) along the cells of the matrices. The initial and final rows of the alignment are 3 and 7, respectively, while the alignment superior and inferior divergences are −3 and 0, respectively. These limits determine what we define as the alignment region (AR), shown in shadow in the figure.

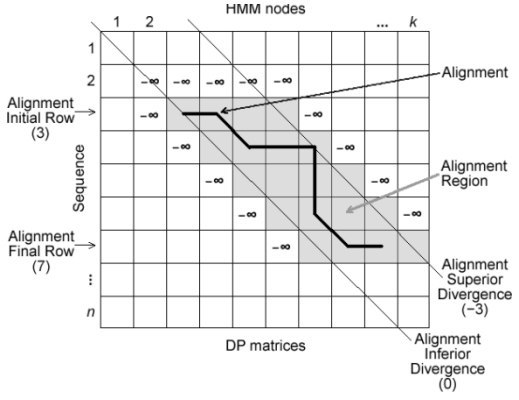


Fig. 5: Divergence concept: alignment limits, initialized cells (with -∞) and alignment region, for a HMM with k nodes and a sequence of length n.

The AR contains the cells of the score matrices M, I, and D that must be computed in order to obtain the best alignment. The other Viterbi DP vectors are also limited by IR and FR, as well. The alignment limits are calculated precisely, leaving no space to error, in a sense that computing only the cells inside the AR will produce the same best alignment as the unbounded (not limited to the AR) computation of the whole matrices.

Our Divergence Algorithm (DA) works in two main phases. The first phase is inserted into a simplified version of the Viterbi algorithm which eliminates data dependencies induced by the J state. In this phase, we compute the similarity score of the best alignment of the sequence against the profile HMM, but we do not obtain the alignment itself. We also calculate the limits of the alignment, while computing the similarity score. These limits are computed as new DP matrices and vectors, by means of a new set of recurrence equations. The alignment limits IR, SD, and ID are computed for the M, I, D, E, and C states. The limit FR is computed only for the C state.

The Viterbi algorithm in Fig. 2 has (1) for the M state score computation:

$$M(i,j) = em\left(M_j, s(i)\right) + max\begin{cases} M(i-1,j-1) + tr(M_{j-1}, M_j) \\ I(i-1,j-1) + tr(I_{j-1}, M_j) \\ D(i-1,j-1) + tr(D_{j-1}, M_j) \\ B(i-1) + tr\left(B, M_j\right) \end{cases} \quad (1)$$

Let SelM assume the values 0, 1, 2 or 3, depending on the result of the maximum operator in (1). If the argument selected by the maximum operator is the first, second, third, or fourth one, then SelM will assume the value 0, 1, 2, or 3, respectively. Then, the alignment limits IR, SD, and ID, concerning the score matrix M, are defined by the recurrence equations in Fig. 6.

$$IR_M(i,j) = \begin{cases} IR_M(i-1,j-1), & if\ Sel_M = 0 \\ IR_I(i-1,j-1), & if\ Sel_M = 1 \\ IR_D(i-1,j-1), & if\ Sel_M = 2 \\ i, & if\ Sel_M = 3 \end{cases}$$

$$SD_M(i,j) = \begin{cases} SD_M(i-1,j-1), & if\ Sel_M = 0 \\ \min\left(i-j, SD_I(i-1,j-1)\right), & if\ Sel_M = 1 \\ \min\left(i-j, SD_D(i-1,j-1)\right), & if\ Sel_M = 2 \\ i-j, & if\ Sel_M = 3 \end{cases}$$

$$ID_M(i,j) = \begin{cases} ID_M(i-1,j-1), & if\ Sel_M = 0 \\ \max\left(i-j, ID_I(i-1,j-1)\right), & if\ Sel_M = 1 \\ \max\left(i-j, ID_D(i-1,j-1)\right), & if\ Sel_M = 2 \\ i-j, & if\ Sel_M = 3 \end{cases}$$

Fig. 6: Recurrence equations for the alignment limits IL, SD, and ID, concerning the score matrix M, for $1 \le i \le n$ and $1 \le j \le k$.

The alignment limits IR, SD, and ID, related to the score matrices I and D and vector E are defined analogously, based on the value of SelI, SelD, and SelE, determined by the result of the maximum operator of the Viterbi algorithm recurrence equation for the I, D, and E states, respectively.

Given the recurrence equation for the C state score computation in the Viterbi algorithm in Fig. 2, let SelC assume the values 0 or 1, depending on the result of the

maximum operator in this equation. Fig. 7 shows the recurrence equations which define the alignment limits IR, FR, SD, and ID, concerning the score vector C.

$$IR_C(i) = \begin{cases} IR_C(i-1), & if\ Sel_C = 0 \\ IR_E(i), & if\ Sel_C = 1 \end{cases}$$

$$FR_C(i) = \begin{cases} FR_C(i-1), & if\ Sel_C = 0 \\ i, & if\ Sel_C = 1 \end{cases}$$

$$SD_C(i) = \begin{cases} SD_C(i-1), & if\ Sel_C = 0 \\ SD_E(i), & if\ Sel_C = 1 \end{cases}$$

$$ID_C(i) = \begin{cases} ID_C(i-1), & if\ Sel_C = 0 \\ ID_E(i), & if\ Sel_C = 1 \end{cases}$$

Fig. 7: Recurrence equations for the alignment limits IL, FL, SD, and ID, concerning the score vector C, for $1 \leq i \leq n$.

The first phase of the Divergence Algorithm was thought to be implemented in hardware because its implementation in software would increase the memory requirements and processing time as it introduces new DP matrices. Besides, the alignment limits computation does not create new data dependencies inside the Viterbi algorithm and can be performed in parallel to the similarity score calculation, what can be easily implemented in hardware.

The second phase of the DA uses the output data coming from the first one. If the alignment score (computed in phase 1) is significant enough, the second phase generates the alignment itself. To achieve this, we execute the Viterbi algorithm again, this time keeping track of the information required to obtain the alignment. Nevertheless, it is not necessary to compute the whole DP matrices of the Viterbi algorithm again. We use the alignment limits produced by the first phase, in order to calculate only the cells inside the AR of the DP matrices, saving memory space and execution time. Fig. 8 shows the high-level structure of the DA and the interaction between its two phases.

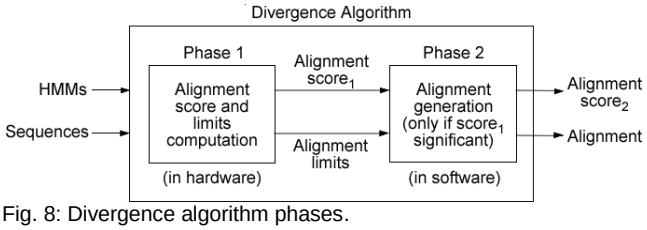


Fig. 8: Divergence algorithm phases.

The DA's second phase is implemented in software as a modification inside HMMER's Viterbi function used by hmmpfam and hmmsearch programs. In this function, we need to initialize with $-\infty$ only the cells above and to the left of the AR, as shown in Fig. 3. The main loops are also modified in order to calculate only the cells inside the AR, using the alignment limits IR, FR, SD, and ID.In the next section we propose a hardware implementation of the first phase of DA. This implementation accelerates the analysis, through the Viterbi algorithm, of a sequence database with a set of profile HMMs in two ways. It acts as a pre-filtering phase, since only the sequences with significant similarity score proceed to the DA's second phase in software. Besides, the first phase also minimizes the alignment generation time of phase 2, since it reduces the number of DP matrices cells that the software has to calculate for the significant sequences, by supplying the alignment limits of these sequences to the software.

5 HMMER-ViTDiV Architecture

The proposed architecture, called HMMER-ViTDiV, consists of an array of interconnected Processing Elements (PEs) that implements a simplified version of the Viterbi Algorithm [3], [11], [13], [14], [16] including the necessary modifications to calculate the divergence algorithm presented in Section 4. The architecture is projected to be integrated to the system as a pre-filter stage that returns the similarity score and the divergence values for a query sequence with a specific profileHMM. If the similarity score for the query sequence is significant enough, then the software uses the divergence data calculated for the sequence inside the architecture and generates the alignment using the divergence algorithm. Each PE calculates the score for the *j* column of the dynamic programming (DP) matrices of the Viterbi Algorithm and the Divergence information for the same column. Fig. 9 shows the DP matrices anti-diagonals and their relationship with each one of the PEs when the number of profileHMM nodes is equal to the number of implemented PEs inside the architecture. In the figure, the arrows show the DP matrices anti-diagonals, cells marked with I correspond to idle PEs and shaded cells correspond to DP cells that are being calculated by their corresponding PE. The Systolic Array is filled gradually as the sequence elements are inserted until there are no idle PEs left, and then when sequence elements are exiting, it empties until there are no more DP cells to calculate.

As the size of commercial FPGAs is currently limited, today we cannot implement a system with a number of PEs that is equal to one of the largest profileHMM in sequence databases (2295) [2]. We implemented a system that divides the computation into various passes, each one computing a band of size N of the DP matrices, where N is the maximum number of PEs that fits into the target FPGA. In each pass the entire sequence is fed across the array of PEs and the DP scores are calculated for the band N. Then the output of the last PE of the array is stored inside FIFOs, as it is the input to the next pass and will be consumed by the first PE. Fig. 10 presents the concept of band division and multiple passes.

As shown in Figure 10, in each pass the PE acts as a different node of the profileHMM, and has to be loaded with the corresponding transition and emission probabilities that are required by the calculations. Also, we note that the system does not have to wait for the entire sequence to be out of the array in one pass to start the next pass, and the PEs can be in different passes at a given time.

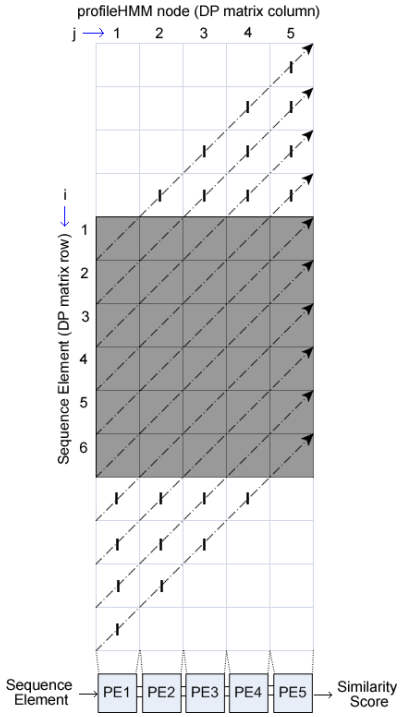


Fig. 9: PE to DP matrices correspondence when the profileHMM node number is less or equal to the implemented PEs.

Two modified RAM memories per PE are included inside the architecture to store and provide the transition and emission probabilities for all passes. Two special sequence elements are included in the design to ease the identification of the end of a pass (@) and the end of the sequence processing (*). A controller is implemented inside each PE to identify these two characters, increment or clear the pass number and signal the transition and emission RAM memories as their address offset depends directly on the pass number.

An input multiplexer had to be included to choose between initialization data for the first pass and intermediate data coming from the FIFOs for the other passes. A transition register bank had to be included to store the 9 transition probabilities are used concurrently by the PE. This bank is loaded in 5 clock cycles by a small controller inside the transition block RAM memory. Fig. 11 shows the transition and emission memories address scheme and Fig. 12 shows a general diagram of the architecture.

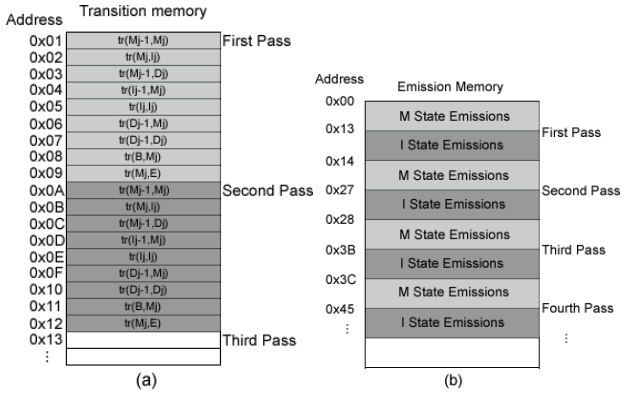


Fig. 11: a)Transition memory organization. b)Emission memory organization.

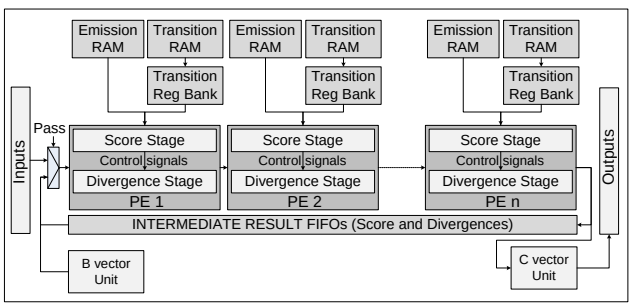


Fig. 12: General Block Diagram of the architecture.

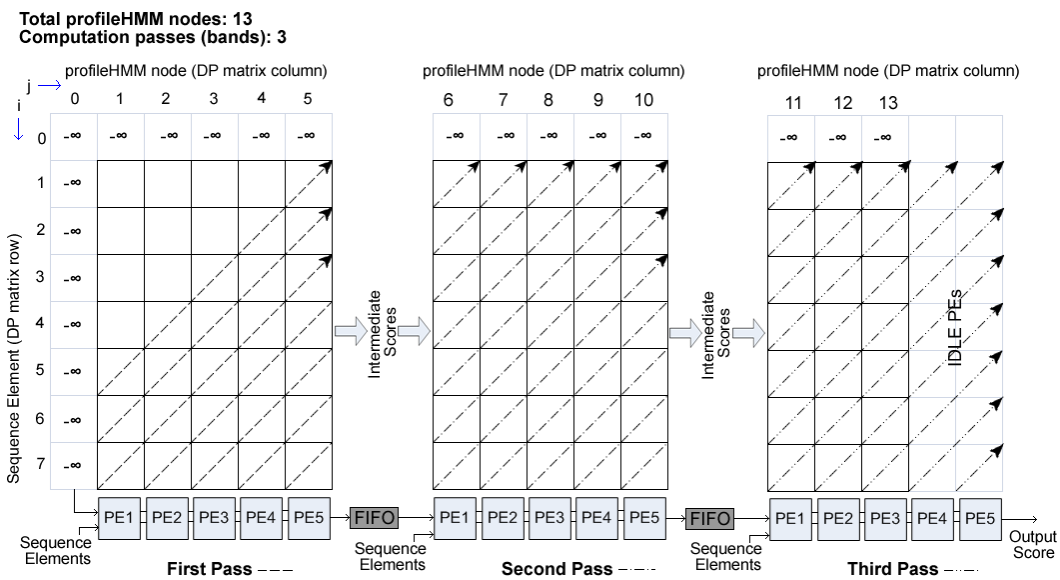


Fig. 10: PE to DP matrices correspondence for profileHMMs that have more nodes than PEs (Band division and multiple passes).

As illustrated in Fig. 12, the PE consists of a score stage which calculates the M, I, D and E scores, and a divergence stage which calculates the alignment limits for the current sequence. Additional modules are included for the B and C vector calculations which were placed outside from the PE array in order to have an easily modifiable and homogeneous design.

5.1 Score Stage

This stage calculates the scores for the M, I, D and E states of the simplified Viterbi algorithm (without the J state). Each PE represents an individual HMM node, and calculates the scores as each element of the sequence passes through. The PE's inputs are the scores calculated for the current element in a previous HMM node, and the PE's outputs are the scores for the current sequence element in the current node. The score stage of the PE uses a) 16-bit saturated adders which detect and avoid overflow or underflow errors by saturating the result either to 32767 or to -32768 and b) modified maximum units which not only return the maximum of its inputs but also the index of which of them was the chosen one. Finally, the score stage consists also of 8 16-bit registers used to store the data required by the DP algorithm to calculate the next cell of the matrix. Fig. 13 shows the operator diagram of the score stage. The 4-input maximum unit was implemented in parallel in order to reduce the critical path of the system and thus increment the operating frequency.

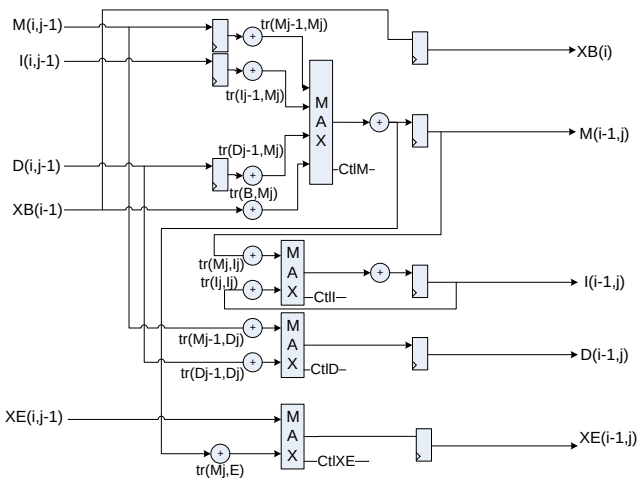


Fig. 13: Score Stage for the architecture's PE.

5.2 Divergence Stage

This stage calculates the alignment limits for the current query sequence element. The stage inputs are the previous node alignment limits for the current query sequence element and the outputs are the calculated alignment limits for the current element. The outputs depend directly on the score stage of the PE and are controlled by the ctlM, ctlI, ctlD and ctlXE signals. The divergence stage also requires the current sequence element index, in order to calculate the alignment limits.

Fig. 14 shows the Divergence Algorithm (DA) implementation for the M and E states. Figs. 15 and 16 show the DA implementation for the I and D states, respectively. The Base parameter is the position of the PE in the systolic array and the #PE parameter is the total number of PEs in the current system implementation. These parameters are used to initialize the divergence stage registers according to the current pass and ensure the limits are calculated correctly. The divergence stage is composed by a) 2-input maximum and minimum operators. b) 2-input multiplexers, which selection line are the control signals and c) 16-bit registers, which serve as temporal storage for the DP data that is needed to calculate the current divergence DP cell.

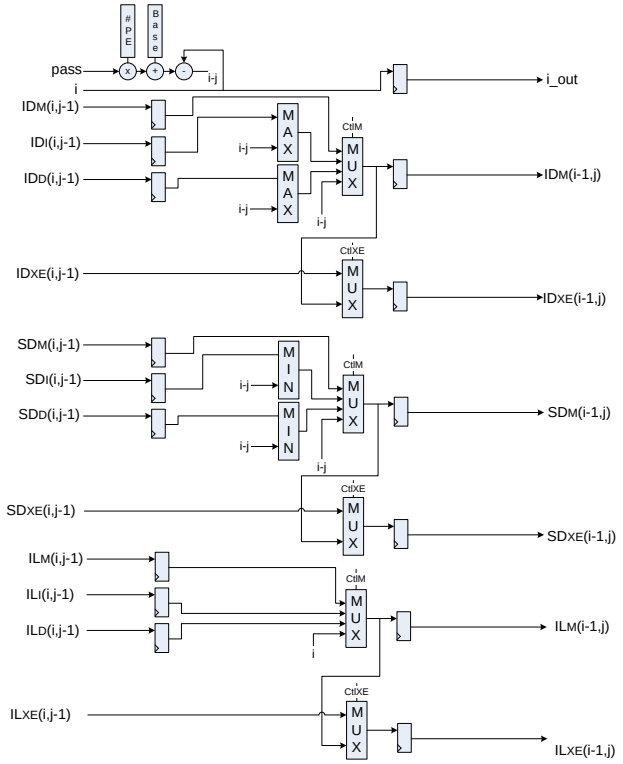


Fig. 14: Divergence calculating stage for M and E states.

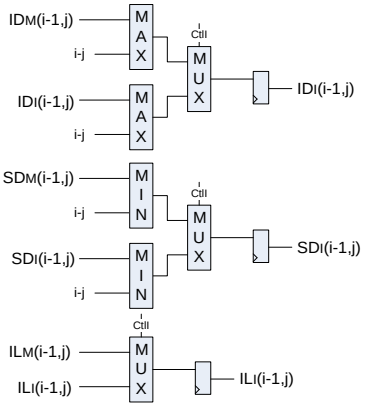


Fig. 15: I state Divergence calculating stage.

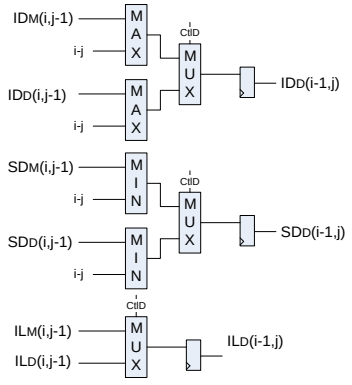


Fig. 16: D state Divergence calculating stage.

5.3 B and C Vector Calculation Units

The B Vector calculation unit is in charge of feeding the PE array with the B vector DP values. This module is placed left of the first PE and it is connected to the XB(i-1) input of it. It has to be initialized for the first iteration with the tr(N,B) transition probability for the current profileHMM by the control software. For other iterations, it adds the tr(N,N) probability to the previous values and feeds the output to the first PE. As discussed in Sections 2 and 4, the divergence algorithm does not generate modifications to the B calculation unit. Fig. 17 shows its hardware implementation.

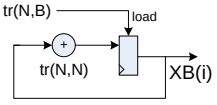


Fig. 17: B vector calculation unit (See Section 2).

The C calculation unit is in charge of consuming the XE output provided by the last PE of the array and generating the output similarity score for the current element of the query sequence (the score for the best alignment up to this sequence element). As the divergence algorithm introduces the calculation of the limits for the best alignment in this state of the Viterbi Algorithm, the score stage of the C unit also delivers the control signal for the multiplexers of the divergence stage. Fig. 18 shows the C vector calculation unit, including the score and divergence stages.

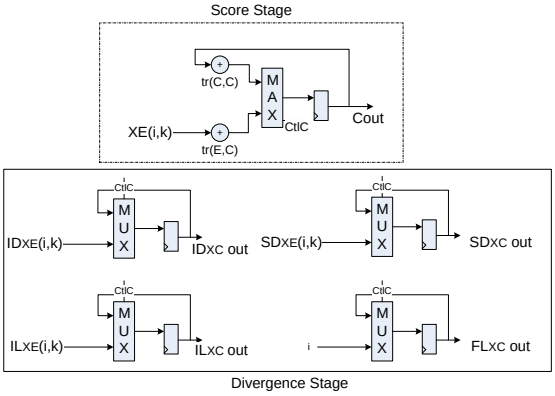


Fig. 18: C vector calculation unit.

6 Proposed Performance Measurement

In order to assess the proposed architecture's performance we used two approaches. The first uses the Cell Updates per Second (CUPS) metric, which is utilized by the majority of the previous works [3], [11], [13], [14], [15], [16], [20] and measures the quantity of DP matrix cells that the proposed architecture is capable of calculate in one second. We choose this metric in order to compare the performance of our system to the other proposed accelerators. The weakness of the CUPS approach is that it does not consider the reprocessing time, and therefore the alignment generation for unaccelerated software, providing an unrealistic measure of the achieved acceleration when integrating the hardware to HMMER.

The second approach measures the execution times of the unaccelerated software when executing a predefined set of sequence comparisons. Then compares it to the execution time of the accelerated system when executing the same set of experiments, to obtain the real gain when integrating a hardware accelerator and the Divergence Algorithm.

Let S_t be the total number of query sequences in the test set, P_t be the total number of profileHMMs in the test set, $t_{s(i,j)}$ the time the unaccelerated *hmmsearch* takes to compare the query sequence S_i to the profileHMM P_j, $t_{rep(i,j)}$ the time the divergence algorithm takes to reprocess (generate the alignment) for the significant query sequence S_i and the profileHMM P_j, $t_{con(i,j)}$ the time spent in communication and control tasks inside the accelerated system, $t_{h(i,j)}$ the time the hardware accelerator takes to execute the comparison between the query sequence S_i and the profileHMM P_j. Then (2), (3) and (4) show the total time spent by unaccelerated HMMER (T_{ss}), the total time spent by the accelerated system (T_{sa}) and the achieved performance gains (G). The times $t_{s(i,j)}$, $t_{h(i,j)}$, $t_{rep(i,j)}$ and $t_{con(i,j)}$ are obtained directly from HMMER, the implemented accelerator and the software implementing the divergence algorithm, and will be shown in the following sections.

$$T_{ss} \leftarrow \sum_{i=1}^{S_t} \sum_{j=1}^{P_t} t_{s_{(i,j)}} \quad (2)$$

$$T_{sa} \leftarrow \sum_{i=1}^{S_t} \sum_{j=1}^{P_t} (t_{h_{(i,j)}} + t_{rep_{(i,j)}} + t_{con_{(i,j)}}) \quad (3)$$

$$G \leftarrow \frac{T_{ss}}{T_{sa}} \quad (4)$$

7 Experimental Results

The proposed architecture not only enhances software execution by applying a pre-filter to the HMMER software, but also provides a means to limit the area of the DP matrices that needs to be reprocessed, by the software, in the case of significant sequences. Because of this, the speedup of the solution must be measured by taking into account the performance achieved by the hardware pre-filter as well as the saved software processing time by only recalculating the scores inside

the alignment region. Execution time is measured separately for the hardware by measuring its real throughput rate (including loading time and inter-pass delays), and for software by computing the savings when calculating the scores and the alignment of the divergence-limited region of the DP matrices (Fig. 5).

Experimental tests were conducted over all the 10340 profile HMMs for the PFam-A protein database [2]. Searches were made using 4 sets of 2000 randomly sampled protein sequences from the Uniprot.Sprot protein database [1] and only significantly scoring sequences were considered to be reprocessed in software. To find out which sequences from the sequence set were significant, we utilized a user defined threshold and relaxed it to include the greatest possible number of sequences [11]. The experiments were done several times to guarantee the repeatability of them and the stability of the obtained data.

7.1 Implementation and Synthesis Results

The complete system was implemented in VHDL and mapped to an Altera Stratix II EP2S180F1508C3 device. Several configurations were explored to maximize the number of HMM nodes, the number of PEs and the maximum sequence length. In order to do design space exploration, we developed a parametrable VHDL code, in which we can modify the PE word size, the number of PEs of the array and the size of the memories. For the current implementation, we obtained a maximum frequency of 67MHz after constraining the design time requirements in the Quartus II tool to optimize the synthesis for speed instead of area. Further works will include pipelining the PE to achieve better performance in terms of clock frequency. Table 2 shows the synthesized configurations and their resource utilization.

7.2 Unaccelerated HMMER Performance

To measure the *hmmsearch* performance in a typical work environment we used a platform composed of an Intel Centrino Duo processor running at 1.8GHz, 4GB of RAM memory and a 250GB Hard Drive. HMMER was compiled to optimize execution time inside a Kubuntu 8.10 Linux distribution. We also modified the hmmsearch program in order to obtain the execution times only for the Viterbi algorithm, as our main goal was to accelerate it. The characterization of HMMER was done by executing the entire set of tests (4 sets of 2000 randomly sampled sequences compared against 10340 profileHMMs) in the modified hmmsearch program. This was done to obtain an exact measure of the execution times of the unaccelerated software and to make its characterization when executing in our test platform. Fig. 19 shows the obtained results for the experiments. The line with triangular markers represents the total execution time of the *hmmsearch* program including the alignment generation times, the line with circular markers represents the execution time only for the Viterbi algorithm, the line with square markers represents the time consumed by the program when generating the alignments, and the line with the plus sign markers corresponds to the expected execution times obtained via the characterization expression shown in (5).

Let l_i be the number of amino acids in sequence S_i and let m_j be the number of nodes in the profileHMM P_j. Then the time to make the comparison between the profileHMM and the query sequence ($t_{s(i,j)}$) was found to be accurately represented by (5) which was found by making a least-squares regression on the data plotted in the circle marked line of Fig. 19.

$$t_{s_{i,j}} \leftarrow -1.3684 * 10^{-18}(m_j l_i)^2 + 4.3208 * 10^{-8}(m_j l_i) - 0.1160 \quad (5)$$

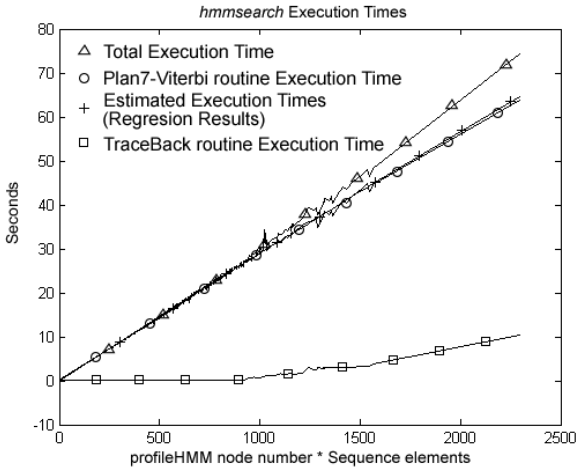


Fig. 19: Unaccelerated *hmmsearch* performance when executing our test set.

Even though we ran our tests with all the profileHMMs in the PFam-A database [2], we chose to shows results only for 6 representative profileHMMs that include the smallest and the largest of the database, due to space limitations. Table 3 shows the estimated execution time obtained with (5) and its error percentage when compared to the real execution times. We obtained a performance of 23.157 Mega Cell Updates per Second (CUPS) for HMMER executing on the test platform.

TABLE 2
AREA AND SPEED SYNTHESIS RESULTS

#PEs	Max passes	Max HMM nodes	Max sequence size	Combinational ALUs	Dedicated registers	Memory bits	% Logic	Max clock frequency (MHz)
25	25	625	8192	31738	18252	2609152	25	71
50	25	1250	8192	59750	35294	3121152	49	71
75	25	1875	8192	93132	52520	3663152	75	69
85	27	2295	8192	103940	59285	5230592	84	67

TABLE 3
MODIFIED HMMSEARCH PERFORMANCE RESULTS.

Sequence set elements	Number of profileHMM nodes	Real Time (Total)	Real Time (Viterbi only)	Estimated Time (Viterbi only)	Error (%)
687406	788	23.40	23.28	22.871	1.75
	10	0.40	0.35	0.1810	48.2
	226	6.55	6.47	6.5635	1.42
	337	9.85	9.8	9.8199	0.2
	2295	74.49	64.09	64.6425	0.8
	901	26.32	26.25	26.1199	0.4
697407	788	24.34	23.48	23.2158	1.12
	10	0.47	0.41	0.1853	54.1
	226	6.68	6.66	6.6602	0.003
	337	9.88	9.83	9.9634	1.35
	2295	78.87	62.89	65.5334	4.2
	901	27.55	25.96	26.4938	2.05
700218	788	24.40	23.37	23.3082	0.264
	10	0.42	0.38	0.1865	50.92
	226	6.76	6.72	6.6873	0.486
	337	10.26	9.84	10.0037	1.663
	2295	81.23	62.25	65.7849	5.678
	901	27.41	26.33	26.5989	1.021
712734	788	25.42	24.03	23.7193	1.293
	10	0.42	0.37	0.1919	48.13
	226	6.82	6.77	6.8083	0.565
	337	10.09	10.01	10.1832	1.730
	2295	81.07	63.81	66.8985	4.840
	901	27.46	26.82	27.0665	0.919

*Execution times are all expressed in seconds.

7.3 Hardware Performance

We formulated an equation for performance prediction of the proposed accelerator, taking into account the possible delays, including systolic array data filling and consuming, profile HMM probabilities loading into RAM memories and probability reloading delays when switching between passes. In order to validate this equation we developed a testbench to execute all the test sets. I/ O Data transmission delays from/ to the PC host were not considered into the formula due to the fact that we will be using at a minimum a PCIe interface [22], whose data transmission rates are well above the maximum required for the system (130MBps).

Let m_i be the number of nodes of the current HMM, S_j be the size of the current query sequence being processed, n be the number of PEs in hardware, f be the maximum system frequency, T_{hw} the throughput of the system (measured in CUPS) and $t_{h(i,j)}$ is the time the accelerator takes to process one sequence set. Then T_{hw} and $t_{h(i,j)}$ are fully described by (6) and (7), where $25n\left\lceil\frac{m_i}{n}\right\rceil$ are the number of cycles spent loading the current HMM into memory, n are the array filling number of cycles, $(S_j+6)\left\lceil\frac{m_i}{n}\right\rceil$ are the cycles spent while processing the current sequence, 3 are the cycles spent loading the special transitions, and $S_j m_i$ are the number of cells that the unaccelerated algorithm will have to calculate to process the current sequence with the current HMM.

$$T_{hw} = \frac{\sum_{i=1}^{\#HMMs}\sum_{j=1}^{\#Seqs} S_j m_i}{\sum_{i=1}^{\#HMMs}\left[\left(\sum_{j=1}^{\#Seqs}(S_j+6)\left\lceil\frac{m_i}{n}\right\rceil\right)+25n\left\lceil\frac{m_i}{n}\right\rceil+n-2\right]} * f \quad (6)$$

$$t_{h_{(i,j)}} = \frac{(S_j+6)\left\lceil\frac{m_i}{n}\right\rceil+25n\left\lceil\frac{m_i}{n}\right\rceil+n-2}{f} \quad (7)$$

We made the performance evaluation for the 4 proposed systolic PE arrays (25, 50, 75 and 85 PEs) and found out that the two characteristics that greatly influence the performance of the array are the quantity of PEs implemented in the array and the number of nodes of the profileHMM we are comparing the sequences against. Table 4 shows the obtained performances for all the array variations when executing the comparisons for our 4 sets of sequences against the 6 profileHMM subset. The best result for each case is shown in bold.

From the table we can see that performance increases significantly with the number of implemented PEs. Also we can observe that the system has better performance for profileHMMs whose node number is an exact multiple of the array node number. This is due to the fact that when a PE does not correspond to a node inside the profileHMM, its transition and emission probabilities are set to minus infinity in order to stop that PE to modify the previously calculated result and only forward that result, thus wasting a clock cycle and affecting performance. Figs. 20 and 21 show the variations in the accelerator performance with the implemented PE number and the profileHMM node number, as seen from the experimental results. From Fig. 21 we can see that as the performance varies according with profileHMM node number, there is an envelope curve around the performance data which shows the maximum and minimum performances of the array when varying the number of the profileHMM nodes.

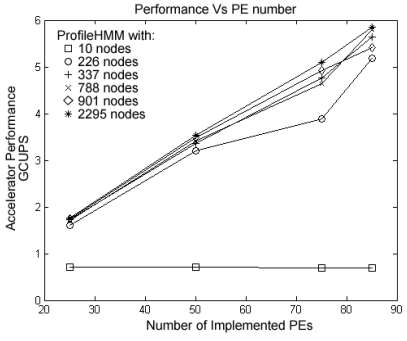


Fig. 20: Performance Vs PE number relation.

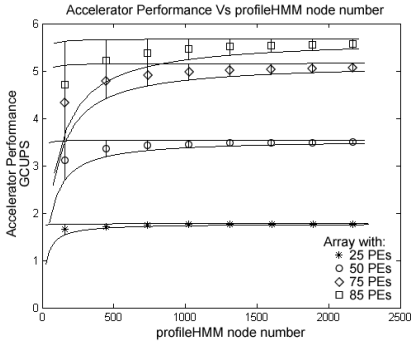


Fig. 21: Performance Vs profileHMM node number graph envelope curves.

TABLE 4
HARDWARE PERFORMANCE RESULTS.

Sequence set elements	Number of profileHMM nodes	T_{hw} (GCUPS) 25 PEs	t_h (sec) 25 PEs	T_{hw} (GCUPS) 50 PEs	t_h (sec) 50 PEs	T_{hw} (GCUPS) 75 PEs	t_h (sec) 75 PEs	T_{hw} (GCUPS) 85 PEs	t_h (sec) 85 PEs
687406	788	1.7468	0.3101	3.4903	0.1552	4.9293	0.1068	5.4203	0.0971
	10	0.7093	0.0097	0.7086	0.0097	0.6880	0.0097	0.6877	0.0097
	226	1.6031	0.0969	3.2033	0.0485	3.8876	0.0388	5.1815	0.0291
	337	1.7075	0.1357	3.4118	0.0679	4.6377	0.0485	5.7949	0.0388
	2295	1.7695	0.8915	3.5358	0.4462	5.0942	0.3010	5.8468	0.2622
	901	1.7274	0.3586	3.3607	0.1843	4.7691	0.1262	5.6341	0.1068
697407	788	1.7468	0.3146	3.4904	0.1574	4.9295	0.1083	5.4205	0.0985
	10	0.7093	0.0098	0.7086	0.0098	0.6880	0.0099	0.6877	0.0099
	226	1.6031	0.0983	3.2033	0.0492	3.8878	0.0394	5.1817	0.0296
	337	1.7075	0.1376	3.4119	0.0689	4.6379	0.0492	5.7952	0.0394
	2295	1.7695	0.9045	3.5359	0.4527	5.0944	0.3053	5.8471	0.2660
	901	1.7274	0.3638	3.3608	0.1870	4.7693	0.1280	5.6344	0.1084
700218	788	1.7468	0.3159	3.4905	0.1581	4.9296	0.1088	5.4206	0.0989
	10	0.7093	0.0099	0.7086	0.0099	0.6880	0.0099	0.6877	0.0099
	226	1.6031	0.0987	3.2034	0.0494	3.8878	0.0396	5.1818	0.0297
	337	1.7075	0.1382	3.4120	0.0692	4.6379	0.0494	5.7953	0.0396
	2295	1.7695	0.9081	3.5359	0.4545	5.0945	0.3066	5.8472	0.2671
	901	1.7274	0.3652	3.3608	0.1877	4.7693	0.1286	5.6345	0.1088
712734	788	1.7468	0.3215	3.4906	0.1609	4.9298	0.1107	5.4209	0.1007
	10	0.7093	0.0100	0.7086	0.0101	0.6880	0.0101	0.6878	0.0101
	226	1.6031	0.1005	3.2035	0.0503	3.8880	0.0403	5.1821	0.0302
	337	1.7075	0.1407	3.4121	0.0704	4.6382	0.0503	5.7956	0.0403
	2295	1.7695	0.9244	3.5360	0.4626	5.0947	0.3120	5.8475	0.2719
	901	1.7274	0.3718	3.3609	0.1911	4.7696	0.1308	5.6348	0.1108

7.4 Re-processing Stage Performance (with Divergence Algorithm)

When aligning different sequences with profileHMMs is unlikely to find two alignments that are equal. Due to this fact, we cannot predict beforehand what will be the performance of the reprocessing stage as the divergence limits for every alignment are likely to be different. To make an estimate of the performance of the second stage, we made a study in which we executed the comparison of the 20 top profileHMMs from the PFam-A [2] database with our 4 sets of query sequences to obtain both the similarity score and the divergence data for them. Then we built a graph plotting the similarity score threshold and the number of sequences with a similarity score greater than the threshold. From this graph we learned that less than 1% of the sequences were considered significant, even relaxing the threshold to include very bad alignments. With this information, we plotted the percentage of the DP matrices that the second stage of the system will have to reprocess in order to find out the worst case situation and make our estimations with it. From Fig. 22 we can see that, for the experimental data considered, in the worst case the divergence region only corresponds to 22% of the DP matrices.

To obtain the second stage performance estimations for HMMER ($t_{rep(i,j)}$ in Eq. 3), we obtained the percentage of significant sequences (p_s), multiplied it by the worst case percentage of the DP matrices that the second stage has to reprocess in order to generate the alignment (p_c) and then we multiplied it by the time the program *hmmsearch* takes to do the whole query sequence (S_i) comparison with a profileHMM (P_j). Equation (8) shows the expression used to estimate the performance for the second stage. Table 5 presents the obtained results and also shows the comparison between the times the second stage will spend reprocessing the significant sequences with and without the divergence algorithm. As shown in table 5, we obtained a performance gain up to 5 times only in the reprocessing stage.

$$t_{reg_{i,j}} = t * p_s * p_c \tag{8}$$

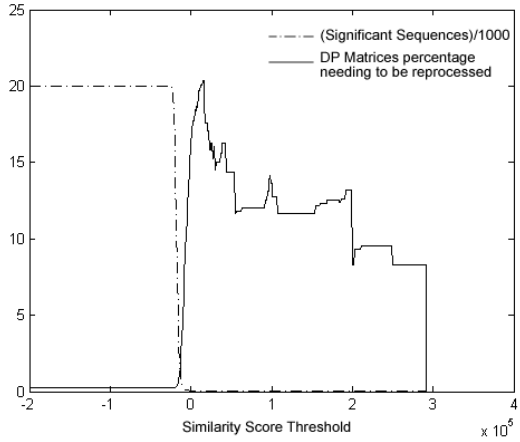


Fig. 22: Number of significant sequences and DP percentage that is required to reprocess in software Vs Similarity Score Threshold.

TABLE 5
SECOND STAGE PERFORMANCE ESTIMATIONS.

Sequence set elements	Number of profileHMM nodes	Entire sequence set processing time in unaccelerated HMMER	Significant sequences reprocessing times with a pre-filter and unaccelerated HMMER	Divergence accelerated significant sequences reprocessing times
687406	788	23.40	0.234	0.0515
	10	0.40	0.004	0.0009
	226	6.55	0.0655	0.0144
	337	9.85	0.0985	0.0217
	2295	74.49	0.7449	0.1639
	901	26.32	0.2632	0.0579
697407	788	24.34	0.2434	0.0535
	10	0.47	0.0047	0.001
	226	6.68	0.0668	0.0147
	337	9.88	0.0988	0.0217
	2295	78.87	0.7887	0.1735
	901	27.55	0.2755	0.0606
700218	788	24.40	0.244	0.0537
	10	0.42	0.0042	0.0009
	226	6.76	0.0676	0.0149
	337	10.26	0.1026	0.0226
	2295	81.23	0.8123	0.1787
	901	27.41	0.2741	0.0603
712734	788	25.42	0.2542	0.0559
	10	0.42	0.0042	0.0009
	226	6.82	0.0682	0.015
	337	10.09	0.1009	0.0222
	2295	81.07	0.8107	0.1784
	901	27.46	0.2746	0.0604

7.5 Total System Performance

In Section 6, we proposed two approaches to evaluate the performance for the system. For the first approach based in CUPS, we obtained a maximum system performance of up to 5.8 GCUPS when implementing a system composed by 85 PEs. This gives us a maximum gain of 254 times over the performance of unaccelerated HMMER software. For the second approach, as we obtained the individual processing times for every stage of the execution, we can determine the overall system performance by applying (3) in the results obtained in Tables 4 and 5. As mentioned before, we did not take into account communication times, as our interface is capable of providing data at a far superior rate than required. When including the divergence reprocessing stage, we got a maximum gain of up to 182 times the unaccelerated software, which still means a significant gain when comparing to unaccelerated HMMER. Table 6 presents the total execution time of the system and shows the obtained performance gains.

Table 7 shows a brief comparison of this work with the ones found in the literature. From the table, we can observe that despite the area that the divergence stage adds to the PE, we have sufficient PEs and memory resources in order to implement the largest profileHMM in the entire database. Also we note that we not only obtained performance gains inside the pre-filter stage but also in the second stage and that we can implement the divergence concept into a pipelined architecture or a full implementation of the plan7 architecture to have even better results.

TABLE 6
TOTAL SYSTEM PERFORMANCE AND OBTAINED PERFORMANCE GAINS.

Sequence set elements	ProfileHMM node number	Pre-Filter Hardware Execution Time (sec)	Divergence Second Stage Execution Time (sec)	Total Time ($t_{sa(i,j)}$)	Unaccelerated HMMER execution time (sec)	Obtained Gain
687406	788	0.0971	0.0515	0.1486	23.40	157.4697
	10	0.0097	0.0009	0.0106	0.40	37.7358
	226	0.0291	0.0144	0.0435	6.55	150.5747
	337	0.0388	0.0217	0.0605	9.85	162.8099
	2295	0.2622	0.1639	0.4261	74.49	174.8181
	901	0.1068	0.0579	0.1647	26.32	159.8057
697407	788	0.0985	0.0535	0.152	24.34	160.1316
	10	0.0099	0.001	0.0109	0.47	43.1193
	226	0.0296	0.0147	0.0443	6.68	150.7901
	337	0.0394	0.0217	0.0611	9.88	161.7021
	2295	0.2660	0.1735	0.4395	78.87	179.4539
	901	0.1084	0.0606	0.169	27.55	163.0178
700218	788	0.0989	0.0537	0.1526	24.40	159.8952
	10	0.0099	0.0009	0.0108	0.42	38.8889
	226	0.0297	0.0149	0.0446	6.76	151.5695
	337	0.0396	0.0226	0.0622	10.26	164.9518
	2295	0.2671	0.1787	0.4458	81.23	182.2118
	901	0.1088	0.0603	0.1691	27.41	162.0934
712734	788	0.1007	0.0559	0.1566	25.42	162.3244
	10	0.0101	0.0009	0.011	0.42	38.1818
	226	0.0302	0.015	0.0452	6.82	150.885
	337	0.0403	0.0222	0.0625	10.09	161.44
	2295	0.2719	0.1784	0.4503	81.07	180.0355
	901	0.1108	0.0604	0.1712	27.46	160.3972

TABLE 7
RELATED WORK AND COMPARISON.

ref	# PEs	# HMM nodes	Max Seq. Size	Plan7-VA complete	Clock (MHz)	Performance (GCUPS)	Gain	FPGA
[3]	68	544	1024	N	180	10	190	Xilinx Virtex II 6000
[13]	90	1440	1024	N	100	9	247	Xilinx 2VP100
[11]	50	---	---	N	200	5 to 20	---	Not Synthesized
[14]	72	1440	8192	N	74	3.95	195	XC2V6000
[15]	10	256	---	Y	70	7	300	XC3S1500
[16]	50	---	---	N	66	1.3	50	Xilinx Spartan 3 4000
[20]	25	---	---	Y	130	3.2	56.8	Xilinx Virtex 5 110-T
Our	85	2295	8192	N	67	5.8	254 (182*)	Altera Stratix II EP2S180F1508C3

*Including significant sequences reprocessing times.

8 CONCLUSIONS

In this paper we introduced the Divergence Algorithm that enables the implementation of a hardware accelerator for the hmmsearch and hmmpfam programs of the HMMER suite. We proposed an accelerator architecture which acts as a pre-filtering phase and uses the divergence concept to avoid the full reprocessing of the sequence in software. We also introduced a more accurate performance measurement strategy when evaluating HMMER hardware accelerators, which not only includes the time spent on the pre-filtering phase or the hardware throughput, but also includes reprocessing times for the significant sequences found in the process.

We implemented our accelerator in VHDL, obtaining performance gains of up to 182 times the performance of the HMMER software. We also made a comparison of the present work with those found in the literature and found that, despite the increased area, we managed to fit a considerable amount of PEs inside the FPGA, which are capable of comparing query sequences with even the largest profileHMM present in the PFam-A database.

For future works we also intend to implement the Divergence Algorithm in a complete version of the Plan7-Viterbi algorithm and make a pipelined version of the PEs, in order to further augment the performance gains achieved when doing so.

ACKNOWLEDGMENTS

The authors would like to acknowledge the National Council for Technologic and Scientific development (CNPq), the National Microelectronics Program (PNM), the Studies and Projects Financial Fund (FINEP) and the Brazilian Millennium Institute (NAMITEC) for funding this work.

REFERENCES

[1] The Universal Protein Resource - UniProt. http:/ / www.uniprot.org. Last access: June 2009.

[2] Sanger's Institute PFAM Protein Sequence Database. http:/ / pfam.sanger.ac.uk/ . Last access: May 2009.

[3] A. C. Jacob, J. M. Lancaster, J. D. Buhler and R. D. Chamberlain, "Preliminary results in accelerating profile HMM search on FPGAs," *Proc. IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2007)*, pp. 1-8, Mar. 2007, doi 10.1109/ IPDPS.2007.370447.

[4] HMMER: Biosequence analysis using profile hidden Markov models. http:/ / hmmer.janelia.org, 2006.

[5] R. Durbin, S. Eddy, A. Krogh and G. Mitchison, *Biological Sequence Analysis Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, New York, 2008.

[6] R. Darole, J. P. Walters and V. Chaudhary, "Improving MPI-HMMER's Scalability With Parallel I/ O." Technical Report 2008-11, Department of Computer Science and Engineering, University of Buffalo, 2008.

[7] G. Chukkapalli, C. Guda and S. Subramaniam, "SledgeHMMER: A Web Server for Batch searching the PFam Database," *Nucleic Acids Research*, no. 32 (Web Server issue), pp. 542-544, March 2004.

[8] HMMER on the Sun Grid Project. https:/ / hmmer.dev.java.net/ . Last access, July 2009.

[9] D. R. Horn, M. Houston and P. Hanrahan, "ClawHMMER: A Streaming HMMer-Search Implementation," *Proc. Of the ACM/IEEE Conference on Supercomputing (SC 2005)*, pp. 11-19, Nov. 2005, doi 10.1109/ SC.2005.18.

[10] GPU-HMMER. http:/ / mpihmmer.org/ userguideGPUHMMER.htm. Last Access, July 2009.

[11] R. P. Maddimsetty, J. Buhler, R. D. Chamberlain, M. A. Franklin and B. Harris, "Accelerator design for protein sequence HMM search," *Proc. of the 20th annual international conference on Supercomputing*, pp. 288-296, 2006, doi 10.1145/ 1183401.1183442.

[12] BLAST: Basic Local Alignment Search Tool. http:/ / blast.ncbi.nlm.nih.gov/ . Last Access, September 2009.

[13] K. Benkrid, P. Velentzas, and S. Kasap, "A High Performance Reconfigurable Core for Motif Searching Using Profile HMM," *Proc. of the 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 285-292, 2008, doi 10.1109/ AHS.2008.16.

[14] T. Oliver, B. Schmidt, Y. Jakop and D. Maskell, "High Speed Biological Sequence Analysis with Hidden Markov Models on Reconfigurable Platforms," *IEEE Transactions on Information Technology in Biomedicine*, vol. 13, no. 5, pp. 740-746, Sep. 2009.

[15] J. P. Walters, X. Meng, V. Chaudhary, T. Oliver, L. Y. Yeow, B. Schmidt, D. Nathan and J. Landman, "MPI-HMMER-Boost: Distributed FPGA Acceleration," *Journal of VLSI Signal Processing Systems*, vol. 48, no. 3, pp. 223-238, 2007, doi 10.1007/ s11265-007-0062-9.

[16] S. Derrien and P. Quinton, "Parallelizing HMMER for Hardware Acceleration on FPGAs," *Proc. International Conference on Application-specific Systems, Architectures and Processors (ASAP 2007)*, pp. 10-17, July 2007, doi 10.1109/ ASAP.2007.4429951.

[17] L. Hunter, *Artificial Intelligence and Molecular Biology*. MIT Press, 1st edition, 1993.

[18] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, New York, 1997.

[19] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proc. of the IEEE*, vol. 77, no. 2, pp. 257-286, 1989, doi 10.1109/ 5.18626.

[20] Y. Sun, P. Li, G. Gu, Y. Wen, Y. Liu, and D. Liu, "HMMer acceleration using systolic array based reconfigurable architecture," *Proc. of the ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 282-282, 2009, doi 10.1145/ 1508128.1508193.

[21] R. B. Batista, A. Boukerche, and A. C. Melo, "A parallel strategy for biological sequence alignment in restricted memory space," *Journal of Parallel and Distribituted Computing*, vol. 68, no. 4, pp. 548-561, 2008, doi 10.1016/ j.jpdc.2007.08.007.

[22] Dell PCI Express Technology. http:/ / www.dell.com/ content/ topics/ global.aspx/ vectors/ en/ 2004_pciexpress. Last access, July 2009.