Dissertação de Mestrado

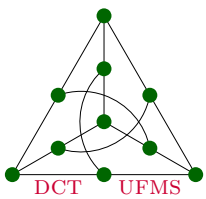
O Problema dos Uns Consecutivos Utilizando Arquiteturas Reconfiguráveis

Adriano Genovez Idalgo

Orientação: Profª. Drª. Nahri Balesdent Moreano

Área de Concentração: Arquitetura de Computadores

Durante o desenvolvimento deste trabalho o autor recebeu apoio financeiro do CNPq



Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul
Novembro de 2007

O Problema dos Uns Consecutivos
Utilizando Arquiteturas Reconfiguráveis

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Adriano Genovez Idalgo e aprovada pela comissão julgadora.

Campo Grande/MS, Novembro, 2007.

Banca Examinadora:

- Profª. Drª. Nahri Balesdent Moreano (Orientadora) (DCT-UFMS)
- Prof. Dr. Ricardo Pezzuol Jacobi (CIC-UnB)
- Prof. Dr. Ricardo Ribeiro dos Santos (EC-UCDB)

Agradecimentos

Em primeiro lugar, gostaria de agradecer à minha orientadora, Nahri Balesdent Moreano, pela paciência e apoio técnico, sempre respondendo minhas dúvidas e fornecendo materiais didáticos para que eu aprendesse e implementasse as soluções deste trabalho.

Agradeço também ao professor Guido Araújo do Instituto de Computação da Unicamp por ceder a placa multimídia que contém a FPGA, sem a qual este trabalho não poderia seguir adiante. Também agradeço ao professor Henrique Mongelli pelo auxílio em relação à bolsa financiada pelo CNPQ.

Também quero agradecer à minha família, pelo apoio dado durante a realização deste trabalho e pelo incentivo a nunca desistir quando problemas surgiam.

Aos amigos do mestrado, um agradecimento especial, pelas horas de estudo em conjunto, pela ajuda nas disciplinas relacionadas ao mestrado e pelas alegrias e aflições vivenciadas juntos em decorrência delas.

Escreveria inúmeras páginas apenas com agradecimentos àqueles que me auxiliaram direta ou indiretamente durante o tempo em que realizei este trabalho, e portanto peço desculpas a estes que omiti por falta de espaço.

Adriano Genovez Idalgo

Resumo

As arquiteturas reconfiguráveis possibilitam que a função do hardware seja implementada pelo usuário. Por causa de suas características, estas arquiteturas têm sido usadas em muitas áreas, inclusive a Bioinformática. Muitos problemas em Bioinformática podem ser representados por modelos matemáticos que, por sua vez, podem ser resolvidos por métodos computacionais. O problema dos uns consecutivos é um exemplo destes problemas, e trata da obtenção de uma permutação de colunas em uma matriz binária, de modo que todos os uns em cada linha sejam consecutivos. Esta matriz representa informações sobre fragmentos de DNA e sondas, os quais permitem a identificação da ordem relativa entre os fragmentos e, assim, auxiliam a determinação da ordem das bases nitrogenadas que formam o DNA original.

Nesta dissertação são descritos alguns conceitos sobre arquiteturas reconfiguráveis e os principais dispositivos de lógica programável. Também são revisados o problema dos uns consecutivos e um algoritmo para resolvê-lo. São apresentadas diversas implementações, em hardware reconfigurável, de partes do algoritmo para resolução do problema dos uns consecutivos de modo a obter um melhor desempenho em sua execução. Também são apresentados e discutidos os resultados obtidos através de experimentos realizados com estas implementações. Finalmente, são descritas as conclusões deste trabalho e mostrados os trabalhos futuros que podem expandir as soluções apresentadas.

Palavras-chave: Mapeamento Físico de DNA, Arquiteturas Reconfiguráveis, Problema dos Uns Consecutivos, Particionamento Software/Hardware.

Abstract

Reconfigurable architectures enable the hardware function to be implemented by the user. Due to its characteristics, these architectures have been used in many areas, including Bioinformatics. Many problems in Bioinformatics can be represented by mathematical models that, in turn, can be solved by computational methods. The consecutive ones problem is an example of such problems. Its goal is to find a permutation of columns in a binary matrix, in such a way that all the ones in each row are consecutive. This matrix represents information about DNA fragments and probes, which allow the identification of the relative order among the fragments and, thus, assist the determination of the order of the nitrogened bases that form the original DNA.

This work describes the concepts of reconfigurable architectures and the main programmable logic devices. The consecutive ones problem and an algorithm to solve it are also revised. It is also presented several implementations, in a reconfigurable hardware, of sections of the algorithm for solving the ones consecutive problem, in order to achieve a better performance in its execution. The results obtained through experiments performed with these implementations are presented and analyzed. Finally, the conclusions of this work are described and the future work that can expand the presented solutions are shown.

Keywords: DNA Physical Mapping, Reconfigurable Architectures, Consecutive Ones Problem, Software/Hardware Partitioning.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

A computação reconfigurável tem sido objeto de estudo e vem ganhando destaque dentro da área de Arquitetura de Computadores. Ela caracteriza o hardware que pode ser reconfigurado, de maneira que a lógica implementada seja criada e modificada pelo usuário final e não pelo fabricante. Deste modo, surgem inúmeras possibilidades de aplicações que não poderiam ser pensadas antes, ao utilizar um hardware com funcionalidade fixa e pré-definida. Tais possibilidades incluem a capacidade de realizar correções na lógica do circuito sem os custos de fabricá-lo novamente, a capacidade de atualizar a lógica do circuito de forma a implementar novas versões da aplicação utilizada, o melhor aproveitamento do hardware devido à possibilidade de projetar a lógica que implemente exatamente a função necessária, o uso de múltiplas unidades funcionais explorando paralelismo da aplicação em vários níveis de granularidade, entre outras. Aliado a essas possibilidades, há o potencial de melhoria de desempenho da aplicação implementada no hardware reconfigurável, em relação à sua implementação em software.

Dentre os dispositivos de lógica programável merece destaque a FPGA (*Field Programmable Gate Array*) [BP00, CH02], que tem sido o dispositivo mais usado para implementar lógica reconfigurável. Uma FPGA contém blocos lógicos e interconexões programáveis, sendo utilizada juntamente com ferramentas CAD (*Computer Aided Design*), que auxiliam na sua configuração para implementar o hardware desejado. Seu uso vai desde o auxílio no projeto de circuitos integrados específicos para a aplicação, prototipando o projeto do circuito final, até a utilização em dispositivos inteligentes que se adaptam em resposta a eventos externos.

Em uma outra área de pesquisa, a Bioinformática, o mapeamento de DNA ganhou recentemente destaque na mídia, devido às conquistas realizadas utilizando algoritmos e técnicas computacionais para auxiliar na tarefa de decifrar a ordem das informações contidas em suas moléculas. Devido a limitações técnicas, o DNA não pode ser seqüenciado (processo de identificação e definição da ordem das bases nitrogenadas que o compõem) de maneira direta. Assim, é necessário que a molécula seja quebrada em pedaços menores para possibilitar a sua leitura. Neste processo, são feitas inúmeras cópias destes fragmentos. Entretanto, o problema está no momento de ordenar os fragmentos para se obter o mapa (a seqüência de bases) do DNA. Diversas técnicas foram propostas, utilizando modelos matemáticos e algoritmos para resolver o problema.

Uma das abordagens representa, como uma matriz binária, as informações sobre os fragmentos de DNA e sondas[1], que chegam a ser bilhões, visto que o genoma humano, por exemplo, possui aproximadamente três bilhões de bases nitrogenadas. É necessário permutar as colunas da matriz de forma que todos os uns em cada linha fiquem consecutivos para encontrar uma ordenação de sondas que possibilite reconhecer a ordem relativa entre os diversos fragmentos. Tal propriedade é denominada *propriedade dos uns consecutivos*, e alguns algoritmos foram propostos para resolver este problema.

Assim, o problema de mapeamento de DNA, além de ter grande aplicação prática, requer a manipulação de grandes volumes de dados. Em específico, o problema dos uns consecutivos manipula dados binários. Tais características motivam o estudo e implementação, em dispositivos reconfiguráveis, de operações muito freqüentes em algoritmos para os referidos problemas, com o objetivo de obter um melhor desempenho na execução destes algoritmos.

1.2 Objetivos do Trabalho

Este trabalho tem o propósito de estudar os dispositivos de lógica reconfigurável e a sua utilização na implementação de soluções para problemas da área de Bioinformática. Mais especificamente, pretende-se utilizar uma FPGA para implementar operações que ocorrem com grande freqüência em um algoritmo para o problema dos uns consecutivos.

Também pretende-se comparar a execução do algoritmo com e sem o auxílio da FPGA, de maneira a elicitar as vantagens e desvantagens encontradas. Em particular, deseja-se comparar e analisar o desempenho das diversas implementações.

[1]Sonda é uma seqüência de bases nitrogenadas complementares às bases de uma única subseqüência do cromossomo. Assim, as sondas combinam-se a fragmentos que contêm a subseqüência correspondente.

1.3 Organização do Texto

O restante do texto está organizado em seis capítulos. O Capítulo 2 mostra uma visão geral das arquiteturas reconfiguráveis, seus benefícios e critérios para sua classificação. Também descreve a organização e características dos principais tipos de dispositivos de lógica programável disponíveis no mercado. No Capítulo 3, são descritas algumas implementações, encontradas na literatura, de problemas em Bioinformática utilizando arquiteturas reconfiguráveis. O Capítulo 4 introduz o problema dos uns consecutivos e apresenta um algoritmo que o soluciona. Isto é feito através de exemplos, visando uma melhor compreensão. No Capítulo 5, é apresentada a solução software/hardware do problema dos uns consecutivos e suas diversas implementações, incluindo a descrição dos módulos que compõem o circuito do hardware reconfigurável e o modelo de comunicação entre FPGA e processador. No Capítulo 6, são apresentados os resultados obtidos através de experimentos realizados com as diferentes implementações e discutidos os seus aspectos e escolhas de projeto. Por fim, no Capítulo 7, são feitas as conclusões deste trabalho e indicados os trabalhos futuros que podem expandir as soluções apresentadas.

Capítulo 2

Computação Reconfigurável

Este capítulo apresenta os conceitos que definem a computação reconfigurável, os principais dispositivos concebidos ao longo de sua história, e uma visão do cenário atual com exemplos de soluções comerciais disponíveis.

2.1 Arquiteturas Reconfiguráveis

Os processadores de propósito geral provêem uma plataforma de computação flexível, sendo capazes de executar uma grande variedade de aplicações. A funcionalidade de cada componente do microprocessador é fixa. Deste modo, os programas devem ser desenvolvidos utilizando o conjunto de instruções disponibilizado pela arquitetura e operando em dados armazenados na hierarquia de memória. Assim, instruções mais complexas ou específicas, não disponíveis no conjunto de instruções do microprocessador, devem ser mapeadas em instruções existentes que, executadas na ordem correta, produzem o resultado desejado. Entretanto, a decodificação e execução seqüencial de instruções, o gargalo no acesso à memória e a arquitetura de controle fixa limitam o desempenho que pode ser alcançado usando estes processadores [BP02].

Circuitos integrados específicos para a aplicação (*Application Specific Integrated Circuits* – ASICs) fornecem uma alternativa para questões de desempenho em relação aos processadores de propósito geral. ASICs são projetados para uma aplicação em particular, possuindo funcionalidade fixa e desempenho superior em relação aos processadores convencionais [BP00]. Deste modo, ASICs restringem a flexibilidade da arquitetura e excluem quaisquer otimizações e atualizações pós-projeto nas características e algoritmos da aplicação.

O paradigma da computação reconfigurável oferece um compromisso intermediário entre desempenho e flexibilidade. A computação reconfigurável utiliza um hardware que pode ser adaptado, podendo explorar paralelismo de granularidade fina e grossa disponíveis na aplicação. A adaptação do hardware para computações específicas permite que funções complexas possam ser mapeadas na arquitetura, reduzindo o gargalo da busca e execução das instruções. Assim, uma arquitetura reconfigurável pode proporcionar ganhos significativos de desempenho quando comparada a processadores convencionais e de flexibilidade quando comparada aos ASICs. A Figura 2.1 ilustra o compromisso entre desempenho e flexibilidade dos dispositivos de lógica digital descritos.

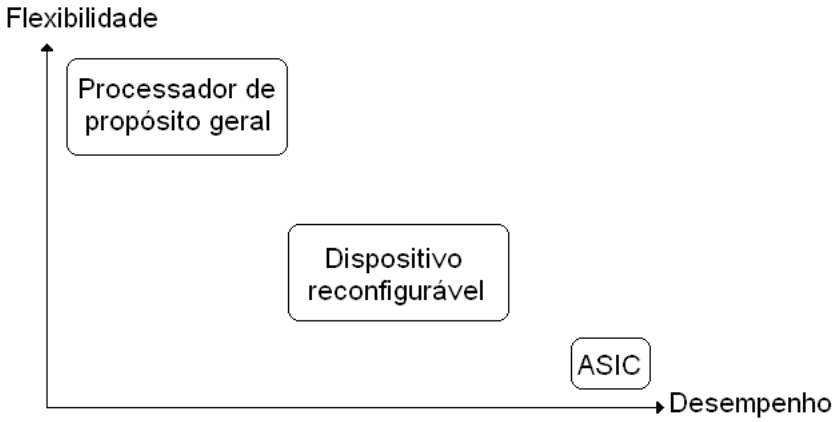


Figura 2.1: Compromisso entre desempenho e flexibilidade

De modo geral, um dispositivo reconfigurável possui blocos lógicos e interconexões programáveis. Tal estrutura permite que a lógica implementada seja alterada através da reconfiguração, sendo realizada com o envio de bits de configuração ao dispositivo, que determinarão o comportamento dos blocos lógicos e interconexões.

O uso de dispositivos reconfiguráveis tem aumentado em decorrência da queda dos preços e do aumento de sua capacidade com o passar dos anos. Isto é, houve um aumento do número de portas lógicas e interconexões disponíveis nos dispositivos. Além disto, uma das principais vantagens em se utilizar um hardware reconfigurável é que basta realizar o carregamento dos bits de configuração para que o hardware do sistema possua novas capacidades e efetue novas tarefas. Sendo assim, é possível que a correção de erros no projeto, melhorias e atualizações sejam aplicadas da mesma forma que as alterações feitas em software.

A reconfiguração dinâmica vai mais longe, permitindo que o hardware mude em resposta a necessidades externas enquanto o sistema continua funcionando. Com esta

capacidade, projetistas podem implementar um hardware maior do que pode caber no dispositivo reconfigurável[1], dado que apenas algumas partes da execução estão ativas simultaneamente. Um exemplo de aplicação é um telefone celular inteligente que permite o uso de múltiplos protocolos de dados e de comunicação, mas apenas um por vez. Quando o telefone passa de uma região com um tipo de protocolo para outra região servida por um protocolo diferente, o hardware é automaticamente reconfigurado [Bar98].

Podem ser destacadas, dentre outras, as seguintes diferenças fundamentais entre as arquiteturas de lógica reconfigurável e os processadores de propósito geral [BP02]:

- Computação espacial: Os dados são processados por diversos blocos lógicos ao mesmo tempo, distribuindo a computação de forma espacial. Isto contrapõe-se ao seqüenciamento temporal da computação através da utilização de unidades funcionais compartilhadas. A implementação espacial permite que a computação explore paralelismo para alcançar maior *throughput* e menor tempo de execução [DW99];
- *Datapath* configurável: A funcionalidade dos blocos lógicos e a rede de interconexão podem ser adaptadas em tempo de execução pelo uso de um mecanismo de reconfiguração;
- Controle distribuído: Os blocos lógicos processam dados baseados em uma configuração local, ao invés do envio simultâneo de uma instrução para todas as unidades funcionais;
- Recursos distribuídos: Os recursos necessários para a computação, tais como unidades funcionais e memória, são distribuídos através do dispositivo em vez de serem localizados em apenas uma de suas partes.

Existem diversos fatores que possibilitam classificar as arquiteturas reconfiguráveis, dentre os quais podem ser citados: granularidade, acoplamento ao processador hospedeiro, modo de reconfiguração e organização da memória [BP02].

A granularidade diz respeito ao tamanho do menor bloco lógico que é endereçado pelas ferramentas que mapeiam a lógica do usuário no dispositivo reconfigurável. Menor granularidade proporciona mais flexibilidade na adaptação do hardware para a estrutura da computação. Isto resulta, porém, em menor desempenho devido ao maior atraso quando se constrói módulos de tamanhos maiores usando blocos lógicos menores.

[1] Apenas uma parte da lógica do circuito está configurada e ativa no hardware. O restante da lógica (na forma de arquivo de configuração) está armazenado em uma memória.

Para executar uma aplicação de forma mais eficiente em um sistema reconfigurável, as partes do programa que não podem ser facilmente mapeadas no dispositivo reconfigurável são executadas em um processador hospedeiro (processador convencional). Tal processador pode também realizar funções de controle para configurar o dispositivo reconfigurável, operações de entrada e saída de dados, entre outros. Entretanto, as partes da aplicação de maior demanda computacional e que podem se beneficiar da implementação em hardware são mapeadas no dispositivo reconfigurável. O nível de acoplamento com o processador hospedeiro interfere nos tempos de reconfiguração e de acesso a dados. O grau de acoplamento pode ser dividido em quatro classes [CH02]. Na primeira, o hardware reconfigurável assume a forma de unidades funcionais reconfiguráveis acopladas ao *datapath* do processador hospedeiro, e ambos estão em um mesmo *chip*, resultando no acoplamento mais forte. Na segunda classe, o dispositivo reconfigurável é usado como um co-processador, capaz de realizar computações sem a supervisão do processador hospedeiro e ambos podem executar instruções simultaneamente. Na terceira classe, o dispositivo reconfigurável se comporta como um processador adicional em um sistema multiprocessado, onde a comunicação entre eles é feita através da memória principal, resultando num maior tempo de comunicação. Finalmente, no acoplamento mais fraco o hardware reconfigurável é uma unidade de processamento independente, comunicando-se raramente com o processador hospedeiro, de forma similar a uma rede de computadores.

A configuração de um dispositivo reconfigurável é feita pela transferência de uma seqüência de bits para o dispositivo. Esta transferência é feita através de alguma interface do dispositivo e o tempo de configuração é diretamente proporcional ao tempo de transferência. Em certas arquiteturas o tempo de configuração pode ser muito grande, e como possível solução utiliza-se a reconfiguração parcial e/ou dinâmica do dispositivo, ou até mesmo uma memória cache de configurações.

A computação realizada no dispositivo reconfigurável necessita acessar dados da memória, e resultados intermediários precisam ser armazenados na memória antes que o dispositivo seja reconfigurado para realizar a próxima computação. A organização da memória afeta o seu tempo de acesso, que é uma fração significativa do tempo de execução. Desta forma, muitas arquiteturas reconfiguráveis incluem memória dentro dos dispositivos reconfiguráveis. Esta memória pode ser implementada como um grande bloco de memória, ou como diversos blocos menores de memória distribuída.

Podem ser destacadas como vantagens da computação reconfigurável em relação a dispositivos com funcionalidade fixa [Bar98]:

- Menor custo do sistema, uma vez que a "fabricação" do hardware pode ser feita pelo usuário e diversas vezes. Se erros são encontrados, não é necessário produzir novos circuitos com a lógica correta, basta apenas reprogramar o dispositivo. Deste modo, o hardware reconfigurável pode ter sua funcionalidade atualizada com o tempo, levando muito mais tempo para se tornar obsoleto que o hardware com funcionalidade fixa;

- O tempo de desenvolvimento é menor, uma vez que não há ciclos de desenvolvimento e prototipação do circuito integrado. O projeto se mantém flexível mesmo depois de pronto o dispositivo, permitindo que projetos incrementais sejam desenvolvidos;

- Em particular para dispositivos dinamicamente reconfiguráveis, é possível alcançar maior funcionalidade com um hardware mais simples. Como nem toda lógica precisa estar presente no dispositivo ao mesmo tempo, o custo de permitir características adicionais é reduzido para o custo da memória necessária para armazenar a lógica.

2.2 Dispositivos de Lógica Programável

Esta seção descreve os principais dispositivos de lógica programável disponíveis comercialmente. Tais dispositivos variam em tamanho e complexidade e são (ou foram) as arquiteturas configuráveis ou reconfiguráveis de uso mais difundido.

Os dispositivos de lógica programável (*Progammable Logic Devices* – PLDs) contêm circuitos que podem ser configurados pelo usuário para implementar uma grande quantidade de funções lógicas. Tais dispositivos têm uma estrutura muito geral e incluem uma coleção de blocos lógicos e interconexões configuráveis que permitem que o circuito interno seja configurado de várias maneiras diferentes [BR96]. O projetista pode implementar as funções necessárias para uma aplicação em particular através da escolha de uma configuração apropriada para os blocos e interconexões. Os blocos e interconexões são configurados pelo usuário final, e não quando o PLD é fabricado.

Vários PLDs atuais podem ser configurados múltiplas vezes. Esta capacidade possui a vantagem de possibilitar ao projetista a realização de mudanças no hardware implementado no PLD, como correções, novas funcionalidades, etc. Estas mudanças são realizadas pela reconfiguração do PLD, tornando o processo de projeto de hardware muito mais flexível. O custo de reconfiguração pode ser melhorado através da reconfiguração parcial e dinâmica. Na reconfiguração parcial, é possível modificar a configuração de parte do dispositivo, enquanto a configuração da parte restante se mantém intacta.

Na reconfiguração dinâmica, os dispositivos permitem a reconfiguração parcial enquanto outros blocos lógicos estão realizando computações.

2.2.1 PLA, PAL e CPLD

Um dos primeiros dispositivos programáveis desenvolvido especificamente para implementar circuitos lógicos foi o PLA (*Programmable Logic Array*) [BV00]. Ele consiste de dois níveis de portas lógicas: um plano AND programável, seguido por um plano OR também programável. A estrutura do PLA permite que qualquer subconjunto das suas entradas (ou os seus complementos) seja encaminhado a qualquer porta AND do plano AND. Assim, cada saída do plano AND pode corresponder a qualquer "e-lógico" das entradas e seus complementos. Similarmente, usuários podem configurar o plano OR de forma que cada saída produza o "ou-lógico" de qualquer subconjunto das saídas do plano AND. Com esta estrutura, PLAs são bem apropriados para implementar funções lógicas da forma "soma de produtos". Eles também são razoavelmente versáteis, uma vez que ambos os termos AND e OR podem ter muitas entradas [BR96]. A Figura 2.2 mostra um PLA programado para implementar o circuito de duas funções lógicas.

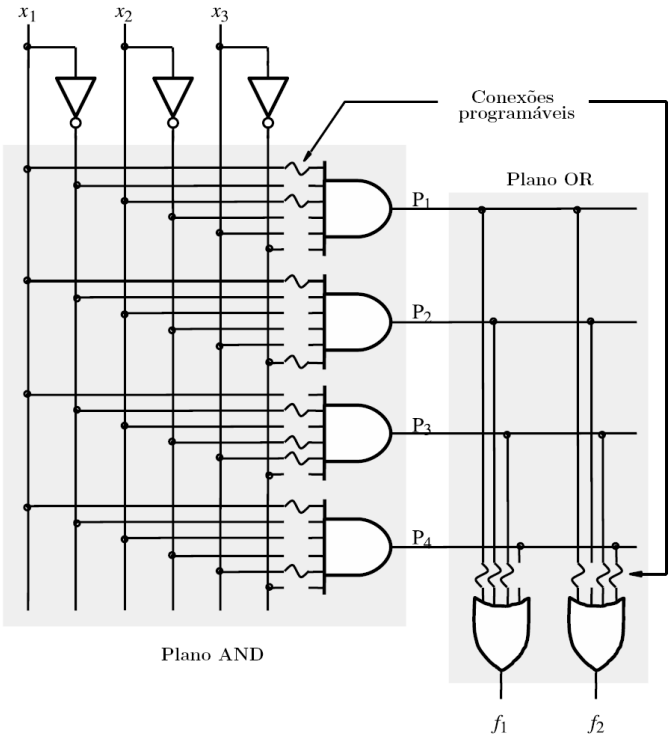


Figura 2.2: PLA implementando $f_1 = x_1 \cdot x_2 + x_1 \cdot \overline{x_3} + \overline{x_1} \cdot \overline{x_2} \cdot x_3$ e $f_2 = x_1 \cdot x_2 + \overline{x_1} \cdot \overline{x_2} \cdot x_3 + x_1 \cdot x_3$ [BV00]

Quando foram introduzidos, no início dos anos 70, os PLAs possuíam como principal ponto negativo seu alto custo de fabricação e baixo desempenho. Ambas as desvantagens resultavam dos dois níveis de lógica configurável: planos lógicos programáveis eram difíceis de produzir e introduziam atrasos significativos na propagação dos sinais. Para resolver estes problemas foram desenvolvidos os dispositivos PAL (*Programmable Array Logic*).

Os dispositivos PAL possuem apenas um nível de programabilidade, consistindo de um plano AND programável que alimenta portas OR fixas. Para compensar a falta de generalidade causada pelo plano OR fixo, PALs possuem variações, com diferentes números de entradas e saídas, e portas OR com diferentes números de entradas. Em muitos PALs, circuitos extras são adicionados na saída de cada porta OR para proporcionar maior flexibilidade [BV00]. Por exemplo, PALs geralmente contêm flip-flops conectados às saídas das portas OR, para implementar circuitos seqüenciais.

Os pequenos dispositivos programáveis, incluindo PLAs, PALs e similares, podem ser agrupados em uma única categoria de dispositivos simples de lógica programável (*Simple Programmable Logic Devices* – SPLDs), nos quais as características mais importantes eram o baixo custo, o alto desempenho, pouco número de entradas e saídas e baixa capacidade [BR96, BV00].

A dificuldade em aumentar a capacidade de uma arquitetura SPLD é que a estrutura dos planos AND/OR programáveis cresce muito rapidamente com o aumento do número de entradas. O único modo razoável de fornecer dispositivos de grande capacidade, baseados em arquiteturas SPLD, é interconectar vários SPLDs, através de uma rede de interconexões, em um único *chip*. Os produtos com esta estrutura básica são conhecidos como dispositivos complexos de lógica programável (*Complex Programmable Logic Devices* – CPLDs) [BR96, BV00].

A rede de interconexões entre os blocos (SPLDs) contém *switches* programáveis que são usados para conectar os blocos. Os *switches* providos para conexões entre os fios visam proporcionar flexibilidade no mapeamento dos circuitos [BV00].

2.2.2 FPGA

Uma FPGA (*Field Programmable Gate Array*) é um PLD que permite a implementação de circuitos lógicos relativamente grandes. FPGAs não contêm planos AND ou OR programáveis, como PLAs e PALs, mas contêm blocos lógicos para a implementação da função requisitada. A estrutura geral de uma FPGA consiste de três componentes principais: blocos lógicos (*Configurable Logic Blocks* – CLBs), blocos de

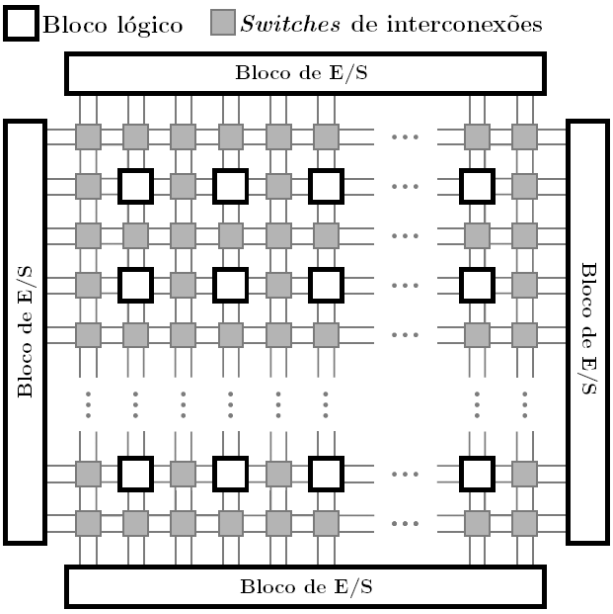


Figura 2.3: Estrutura geral de uma FPGA [BV00]

entrada/saída para conectar os pinos do *chip*, e fios e *switches* que formam uma rede de interconexões [BV00]. Os blocos lógicos são organizados em uma matriz de duas dimensões, e as interconexões são organizadas como canais verticais e horizontais de roteamento entre linhas e colunas de blocos lógicos. Os canais de roteamento contêm fios e *switches* programáveis que permitem que os blocos lógicos sejam interconectados de várias maneiras. *Switches* programáveis também existem entre os blocos de E/S e as interconexões, como mostrado na Figura 2.3.

A capacidade de configuração, na maioria das FPGAs atuais, é alcançada usando memórias SRAM (*Static Random Access Memory*). Elas controlam as configurações dos blocos lógicos e interconexões [BP00, BP02]. Os bits de memória são conectados aos pontos de configuração na FPGA, e a escrita nestes bits configura a FPGA. Assim, a FPGA pode ser reprogramada diretamente pelo envio de diferentes bits de configuração para as células de memória SRAM.

Cada bloco lógico geralmente tem um pequeno número de entradas e saídas, e diferentes FPGAs apresentam diferentes tipos de blocos lógicos, tais como: blocos com planos AND/OR (como PALs), blocos baseados em multiplexadores, e até mesmo blocos com funções fixas tais como portas NAND e XOR [CH02]. O bloco lógico mais usado é baseado em pequenas memórias denominadas *lookup-tables* (LUTs) [CH02, BV00, TB01].

Uma LUT contém células de armazenamento que são usadas para implementar uma função lógica pequena. Cada célula é capaz de armazenar um único valor lógico, 0 ou

1. O valor armazenado é produzido como saída da célula de armazenamento. Dentro de cada LUT existem multiplexadores que permitem escolher qual célula de armazenamento será escolhida como saída, e tais multiplexadores têm como sinais de controle os sinais de entrada da LUT. A Figura 2.4 mostra uma LUT de duas entradas, configurada para implementar uma função lógica. Existem LUTs de vários tamanhos, onde o tamanho é definido pelo número de entradas. Assim como em PALs, FPGAs podem ter circuitos extras junto às LUTs em cada bloco lógico.

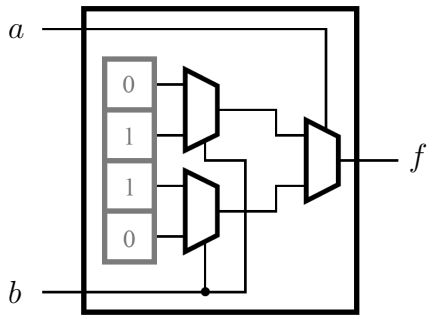


Figura 2.4: LUT de duas entradas implementando $f = a\ XOR\ b$

Para um circuito lógico do usuário ser implementado em uma FPGA, o circuito é automaticamente traduzido para o formato necessário através do uso de ferramentas CAD (*Computer Aided Design*) [BV00]. Quando a FPGA é configurada para realizar o circuito, os blocos lógicos são programados para realizar as funções necessárias, e os canais de roteamento são programados para fazer as interconexões necessárias entre os blocos lógicos.

Blocos lógicos tipicamente contêm LUTs, flip-flops, lógica combinacional adicional e células de memória SRAM para controlar a configuração do bloco lógico. A rede de interconexão pode ser reconfigurada pela mudança das conexões entre os blocos lógicos e os fios da rede, e também pela configuração dos *switches* que conectam diferentes fios. Tais *switches* da rede de interconexão também são controlados por células de memória SRAM [BV00].

As FPGAs atuais possuem dezenas e centenas de milhares de CLBs. Como exemplo, a FPGA Virtex-5 da Xilinx é organizada em uma matriz de 240×108 CLBs, onde cada CLB possui oito LUTs de seis entradas e oito *flip-flops* [Xil07b]. Já a FPGA Stratix III da Altera possui em seus blocos lógicos elementos adaptativos chamados ALM (*Adaptive Logic Modules*), e cada ALM consiste de uma LUT de oito entradas (que pode ser subdividida para implementar funções menores), dois somadores e dois *flip-flops* [Alt07].

A Figura 2.5 ilustra os elementos básicos das duas FPGAs descritas e a Figura 2.6 mostra parte das LUTs e *flip-flops* que compõe um CLB da Virtex-5.

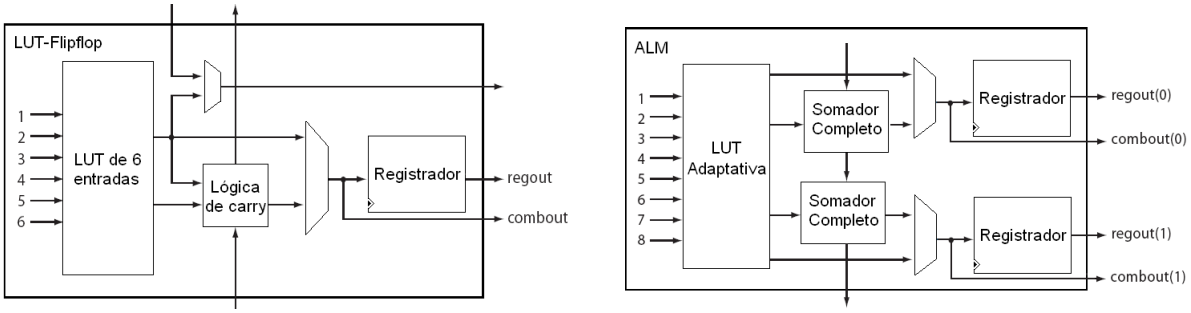


Figura 2.5: Elementos que formam o CLB nas FPGAs Virtex-5 da Xilinx (à esquerda), e Stratix III da Altera (à direita) [Alt06]

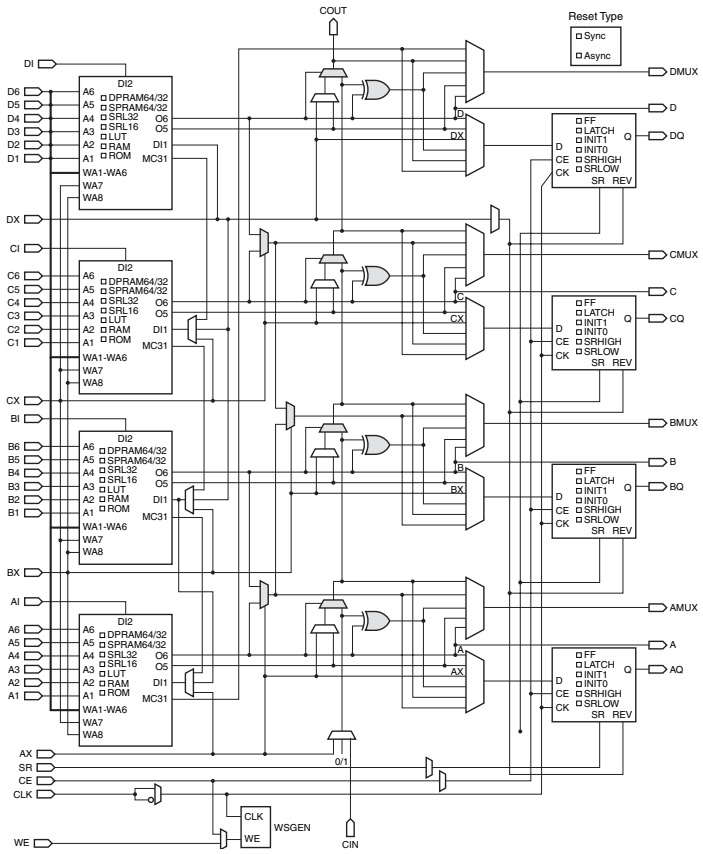


Figura 2.6: Parte de um CLB da FPGA Virtex-5 [Xil07c]

2.3 Arquiteturas Híbridas

Vários fabricantes utilizam FPGAs para construir soluções que atendam diversas áreas de aplicação tais como prototipação de circuitos integrados, armazenamento de dados, transmissão e manipulação de dados, vídeo digital, processamento de imagem, multimídia, navegação, processamento de sinais, telecomunicações, entre outros.

Uma arquitetura híbrida possui um processador de propósito geral e um componente reconfigurável (por exemplo uma FPGA) acoplado a ele [BP00, GS98].

Nesta seção são apresentados alguns sistemas híbridos atuais disponíveis comercialmente. Uma lista contendo os fabricantes e seus dispositivos pode ser encontrada em [Fre07].

2.3.1 Processador e FPGA no Mesmo Chip

As FPGAs da Xilinx, Virtex II PRO e Virtex 4 FX, possuem até dois processadores IBM PowerPC 405 integrados dentro do mesmo chip da FPGA [Xil06d, Xil06e]. Desta forma, a lógica implementada pelo usuário no dispositivo reconfigurável se comunica com o processador através de um barramento, funcionando como um processador adicional. Os caches de dados e instruções do PowerPC não são visíveis pela unidade reconfigurável. Existem diversos *kits* de desenvolvimento disponíveis utilizando essas FPGAs, tais como ML300 Evaluation Platform [Xil06a] e PowerPC & MicroBlaze Development Kit Virtex-4 FX12 Edition [Xil06c].

2.3.2 Processador e FPGA na Mesma Placa

Este tipo de sistema oferece em uma mesma placa de circuito impresso um ou mais processadores de propósito geral e uma ou mais FPGAs. Estes sistemas tiram proveito da proximidade entre processadores e FPGAs com o uso de um barramento embutido na placa. Porém, a comunicação presente nestes sistemas é mais lenta que a comunicação realizada pelo processador convencional integrado na FPGA.

Em [Gen07] é apresentado um sistema contendo duas FPGAs e um processador convencional, destinado para sistemas de defesa e aplicações aeroespaciais. No processador convencional utiliza-se o sistema operacional de tempo real LynxOS, baseado em Unix, proporcionando gerenciamento do sistema e capacidades de cálculos de ponto flutuante para complementar a funcionalidade dos dispositivos reconfiguráveis.

2.3.3 FPGAs em Placas Expansoras

Existem empresas que confeccionam placas de circuito impresso de expansão com FPGAs para computadores desktop e servidores. Tais placas utilizam as interfaces PCI, PCI-Express, PMC (*PCI Mezzanine Card*), entre outras. Estas placas são utilizadas para processamento de sinais digitais, e outras operações que requerem processamento em tempo real.

Os sistemas híbridos formados por estas soluções apresentam grau de acoplamento, entre a FPGA e o processador de propósito geral, mais fraco e com comunicação mais lenta em relação aos processadores embutidos na mesma placa de circuito impresso da FPGA. Porém, o uso de barramentos do tipo PCI proporciona maior velocidade de comunicação do que a encontrada em sistemas híbridos compostos de um PC convencional e uma placa de desenvolvimento interconectados pela interface de rede. Fabricantes de tais soluções podem ser encontrados em [Alp07, Acq07, CES07, Nal07].

Capítulo 3

Bioinformática em Arquiteturas Reconfiguráveis

Neste capítulo são descritos alguns trabalhos que envolvem o uso de arquiteturas reconfiguráveis para solução de problemas de Bioinformática. Através dos trabalhos, podem ser notadas vantagens obtidas na implementação destas soluções em arquiteturas reconfiguráveis, em comparação à execução delas por software em processadores convencionais.

3.1 Varredura de Banco de Dados de Seqüências Biológicas com FPGAs

Varrer um banco de dados de seqüências de proteínas é uma tarefa comum e repetida diversas vezes em biologia molecular. A necessidade de acelerar esta tarefa se origina do crescimento exponencial dos bancos de seqüências biológicas. A operação de varredura consiste em encontrar similaridades entre uma seqüência de busca e todas as seqüências do banco. Esta atividade leva a identificar similaridades entre uma seqüência de funcionalidade desconhecida e as seqüências, com funcionalidade conhecida, que estão no banco. A dissimilaridade entre duas seqüências (ou distância de edição) pode ser definida como o menor número de operações necessárias para transformar uma seqüência na outra, dentre todos os alinhamentos possíveis entre elas. O algoritmo de Smith-Waterman [SW81] computa as similaridades entre duas seqüências, utilizando a técnica de Programação Dinâmica [CLR90].

Em [OSM05a] esse algoritmo é implementado em uma FPGA, através da utilização de elementos de processamento, que operam em paralelo e formam uma estrutura linear. Basicamente, cada elemento de processamento armazena um caractere da seqüência de busca e realiza os cálculos necessários a medida que as outras seqüências (que serão comparadas) passam pela estrutura linear. Como geralmente as seqüências são maiores que o número de elementos de processamento, a computação é dividida em vários passos. Em cada passo, uma parte da seqüência de busca é processada, tal que o número de caracteres desta parte seja igual ao número de elementos de processamento.

Além disso, os elementos de processamento são configurados dinamicamente de forma a serem especializados para a consulta sendo executada. Nos experimentos realizados, a implementação em FPGA apresentou ganhos de desempenho de até 170 vezes em relação à implementação por software em um processador convencional.

Em [RBMR05], o algoritmo de Smith-Waterman também é utilizado e é implementado em uma rede de três FPGAs. Novamente, o algoritmo é mapeado em uma estrutura linear de elementos de processamento, que processam cada caractere da seqüência de busca. Os resultados apresentados mostram uma redução do tempo de execução de mais de 200 vezes, comparado a um sistema com dois processadores de propósito geral.

3.2 Busca Antecipada Adaptativa em Banco de Dados de Seqüências Biológicas com FPGA

A implementação em hardware reconfigurável de consultas a banco de dados de seqüências biológicas pode proporcionar ganhos significativos de desempenho em comparação com implementações em processadores convencionais. Entretanto, o *overhead* de comunicação entre FPGA e processador hospedeiro na implementação híbrida deste tipo de aplicação (com grande volume de dados) limita o desempenho.

Em [MC07] é implementado um esquema de busca antecipada adaptativa para a execução das buscas em banco de dados com o algoritmo de Smith-Waterman. O tempo de execução total da busca de seqüências biológicas pode ser dividido em três partes: tempo de carregamento das seqüências do banco de dados na máquina hospedeira, tempo de processamento do algoritmo de Smith-Waterman na FPGA, e outros tempos incluindo inicialização da FPGA, definição de dados, latência de comunicação e impressão dos resultados.

A transferência de dados necessita de baixa latência e alta largura de banda para

assegurar que a comunicação não afete o tempo de processamento. A placa contendo a FPGA utilizada nos experimentos possui interface PCI e, desta forma, é necessário que haja um *buffer*, na aplicação do processador hospedeiro, que armazene as seqüências do banco de dados e que transfira estes dados para a FPGA quando necessário.

Assim, foi realizada uma implementação com dois *buffers* que operam em paralelo, um implementado na FPGA e o outro no software executado no processador hospedeiro. Quando o processamento termina no *buffer* da FPGA, os resultados são enviados ao processador hospedeiro e a FPGA recebe novas seqüências (utilizando o acesso direto à memória – DMA). São usadas várias *threads* no processador hospedeiro para gerenciar a comunicação e carregar no *buffer* do software as seqüências do banco de dados, sobrepondo o processamento com a comunicação para diminuir de forma eficiente a latência da recuperação das seqüências do banco de dados. Além disto, o *buffer* de busca antecipada possui tamanho adaptativo baseado no tamanho da seqüência de busca, de modo a manter a proporção do tempo de recuperação de seqüências do banco de dados com o tempo de processamento da FPGA para as diferentes seqüências de busca. Com isto, os benefícios da implementação são mantidos independentemente do tamanho da seqüência de busca.

Foram realizados experimentos com várias seqüências de busca, alcançando desempenho 42% superior comparado-se com uma implementação com FPGA sem o esquema de busca antecipada, e *speedup* de 110 em relação à implementação que utiliza apenas software.

3.3 Alinhamento de Múltiplas Seqüências em FPGA

Biologistas moleculares freqüentemente computam alinhamentos de múltiplas seqüências (*Multiple Sequence Alignments* – MSAs) para identificar regiões similares em famílias de proteínas. A programação dinâmica é freqüentemente usada para computar o alinhamento ótimo de um par de seqüências. Entretanto, a utilização da programação dinâmica para o alinhamento simultâneo de múltiplas seqüências é impraticável, dado que as complexidades de tempo e espaço são da ordem do produto do comprimento das seqüências. O problema se agrava ainda mais com o crescimento das bases de dados de seqüências. Assim, muitas heurísticas para computar MSAs em tempo razoável têm sido desenvolvidas, como por exemplo, o *Alinhamento Progressivo* [FD87].

Geralmente, métodos de alinhamento progressivo consistem de três etapas. Primeiro, a dissimilaridade entre cada par de seqüências é computada (usando o algoritmo Smith-Waterman), formando uma matriz. Segundo, uma árvore filogenética é construída

baseando-se nesta matriz. Finalmente, o alinhamento de várias seqüências é feito seguindo-se a ordem de ramificações na árvore filogenética para formar o MSA final.

Em [OSM+05b] uma abordagem para MSA é apresentada, onde somente a primeira etapa do alinhamento progressivo é implementada em uma FPGA, em decorrência da influência predominante desta etapa no tempo de execução. São utilizados elementos de processamento similares aos implementados em [OSM05a] e descritos na Seção 3.1.

Experimentos foram realizados para comparar o tempo de processamento da implementação da primeira etapa do algoritmo na FPGA com o tempo de execução desta mesma etapa em um computador tradicional. Nos resultados mostrados, o ganho de desempenho utilizando a FPGA foi de até $50,9$ vezes.

3.4 Alinhamento de Seqüências em FPGA

Em [JRC+04] é apresentado um protótipo de uma arquitetura sistólica reconfigurável, utilizado para a implementação do método de programação dinâmica para resolver diferentes problemas, dentre eles o de alinhamento de seqüências. Esta arquitetura possui elementos de processamento que calculam em paralelo os valores das diagonais da matriz de programação dinâmica. O *array* sistólico é implementado em uma FPGA acoplada a um computador hospedeiro. Os elementos de processamento também determinam ponteiros, que permitem que os alinhamentos gerados para as seqüências sejam recuperados por software (no computador hospedeiro) em um passo posterior.

Nos resultados apresentados, comparou-se o tempo de execução estimado da arquitetura sistólica usando várias FPGAs em cadeia com um *cluster* de estações de trabalho com 1, 2, 4, e 8 processadores. Por exemplo, para processar duas seqüências de tamanho 80 K, a cadeia de FPGAs levaria $0,003277s$, enquanto que o *cluster* com 8 processadores consumiria $2162,82s$.

3.5 Construção de Árvores Filogenéticas e Cálculo da Probabilidade Máxima em FPGA

Uma árvore filogenética representa a história da evolução de diferentes organismos. Ela ilustra as subdivisões taxonômicas dos animais ou vegetais e determina também as diferenciações na evolução, segundo o tamanho dos ramos. Diferentes espécies são

representadas em uma árvore com topologia e tamanho de ramos específicos.

Algoritmos e modelos matemáticos foram desenvolvidos para construir árvores filogenéticas a partir de um conjunto de seqüências de nucleotídeos. A abordagem da Probabilidade Máxima (*Maximum Likelihood*) permite comparações quantitativas entre diferentes árvores, e pode ser combinada a um Algoritmo Genético para buscar a árvore ótima de forma iterativa [ML03].

Em [ML03] é apresentado um sistema híbrido de *Software/Hardware* (SW/HW) para a reconstrução da árvore filogenética usando o Algoritmo Genético para Probabilidade Máxima (*Genetic Algorithm for Maximum Likelihood* – GAML). O algoritmo genético é implementado em SW e a avaliação da função de probabilidade máxima é implementada em HW.

O sistema SW/HW foi implementado em um computador PC (que executa o SW) e uma FPGA (que implementa o HW). O PC e a FPGA comunicam-se através de uma porta paralela, formando assim um sistema fracamente acoplado, conseqüentemente com grande *overhead* de comunicação.

Para reduzir o *overhead* de comunicação, em [ML04] o sistema de SW/HW é implementado em uma plataforma contendo um processador e uma FPGA na mesma placa de circuito impresso. Este sistema fortemente acoplado permite um melhor desempenho na comunicação entre ambos.

O programa, executado no processador, comunica as topologias das árvores filogenéticas para a FPGA, que, por sua vez, retorna os valores de probabilidade de cada árvore para o software, no processador. A avaliação da probabilidade de cada árvore realiza muitas multiplicações de probabilidades condicionais e estas operações são realizadas em paralelo na FPGA.

Nos experimentos foram constatados ganhos de desempenho. A tarefa realizada passou de $5,4h$ para a execução de 100 iterações do algoritmo genético totalmente em SW em um PC convencional, para $10min$ no sistema SW/HW fracamente acoplado, e para $1,87s$ no sistema com FPGA integrada na mesma placa do processador. A operação de avaliação de probabilidade passou de $80s$ para $0,21s$ no primeiro sistema SW/HW implementado, e para $5,77ms$ no sistema fortemente acoplado.

3.6 Comparação de Seqüências de DNA no Sistema Mercury

Em [KBC+04] o programa BLASTN, uma variante do programa BLAST (*Basic Local Alignment Search Tool*), é usado para comparar seqüências de DNA.

O programa BLASTN é implementado no sistema *Mercury* [Cha03], uma arquitetura com um sistema robusto de armazenamento em discos e com hardware reconfigurável. A lógica reconfigurável (implementada em uma FPGA) é associada ao controlador do disco, de forma a oferecer capacidade computacional muito próxima ao fluxo de dados de saída dos discos. A funcionalidade da aplicação é dividida em duas partes, uma executando na FPGA (obtém os dados do disco, os processa e, por fim, os envia para a memória principal do processador), e a outra no processador principal.

O programa BLASTN é composto de 3 etapas: casamento de palavras entre a seqüência de consulta e o banco de dados, alinhamento entre a seqüência de consulta e o banco de dados sem modificação (pode ocorrer casamento de bases diferentes, mas as seqüências com poucos casamentos corretos são descartadas e não passam para a próxima etapa), e alinhamento com modificação, onde são permitidas inserções e remoções de bases. No final da terceira etapa, os alinhamentos com número suficiente de casamentos de bases são retornados para o usuário.

O sistema *Mercury* implementa na FPGA a primeira etapa. Foram obtidos *speedups* de até 90,8, comparando-se apenas a implementação da primeira etapa no sistema *Mercury* com a implementação totalmente em software em um PC convencional. Para todas as etapas do programa, foram estimados *speedups* de até 7,42.

Capítulo 4

O Problema dos Uns Consecutivos

Neste capítulo são abordados o problema dos uns consecutivos e um algoritmo para sua resolução. Tal problema origina-se da necessidade de obtenção da ordem relativa entre os fragmentos de uma molécula de DNA, visando recuperar a seqüência completa das bases nitrogenadas. É utilizada uma matriz binária para modelar as informações pertinentes sobre os fragmentos. Deste modo, o algoritmo baseia-se na manipulação desta matriz binária para a obtenção da solução do problema.

4.1 Definição do Problema

Dentro do contexto de mapeamento do DNA, existem diversas técnicas que possibilitam a identificação da ordem das bases nitrogenadas de um cromossomo, processo denominado seqüenciamento do DNA. Como o cromossomo é geralmente muito grande para ser analisado como um todo, é necessário quebrá-lo em partes menores, seqüenciá-las, e tentar obter a ordem das seqüências de forma a ter um mapa completo do cromossomo. Para obter tal mapa são necessárias várias cópias da molécula que se deseja seqüenciar, denominada DNA alvo. Cada cópia é quebrada em vários fragmentos usando enzimas. Por fim, o mapeamento é feito comparando cada fragmento, mais especificamente observando as sobreposições. Em geral, cada fragmento é ainda muito grande para ser seqüenciado, portanto não se pode determinar sobreposições pelo seqüenciamento e comparação de fragmentos. A informação de sobreposição é obtida usando impressões digitais dos fragmentos.

As impressões digitais descrevem uma parte da informação contida num fragmento de uma maneira única. Assim, com uma impressão digital é possível identificar o único

fragmento que a possui. Dois modos de se obter impressões digitais são a *Restriction Site Analysis* e a hibridização [SM97]. No primeiro, o objetivo é encontrar locais específicos onde uma dada enzima quebra um DNA alvo. Tais locais são marcadores (seqüências pequenas e precisamente definidas) que possibilitam identificar locais comuns entre dois fragmentos. A técnica usada para localização dos marcadores é baseada na medida do comprimento do fragmento, que é a sua impressão digital [SM97].

No mapeamento por hibridização é verificado se certas seqüências pequenas, denominadas sondas, combinam (ou hibridizam) com os fragmentos, denominados clones (após a quebra, de diferentes maneiras, de cada cópia do DNA alvo, cada fragmento resultante é replicado através de uma técnica denominada clonagem). O subconjunto destas pequenas seqüências que combinam com o fragmento forma a sua impressão digital. Verifica-se, pela comparação de impressões digitais, se os fragmentos possuem sobreposições e assim determina-se a ordem relativa dos fragmentos [SM97].

Um possível modelo matemático correspondente à estratégia de mapeamento por hibridização usa como impressão digital o conjunto de sondas que combinam com o clone. Neste modelo, é considerado que não há erros, que todos os experimentos de hibridização de clones com as sondas foram feitos e que o complemento da seqüência de cada sonda ocorre apenas uma vez ao longo do DNA alvo (sondas são únicas).

Considerando que existem n clones e m sondas, é construída uma matriz binária M, $n \times m$, onde a posição M_{ij} diz se a sonda j hibridizou ($M_{ij} = 1$) ou não ($M_{ij} = 0$) com o clone i. Assim, pode-se definir a propriedade dos uns consecutivos e, em seguida, o problema dos uns consecutivos.

Definição 1 *Dada a matriz M, $n \times m$, M é dita possuir a propriedade dos uns consecutivos para as linhas (Consecutive 1s Property – C1P), se todos os uns em cada linha são consecutivos.*

Definição 2 *Dada a matriz M, $n \times m$, obter um mapa físico através desta matriz se torna o problema de encontrar uma permutação de colunas (sondas) tal que todos os uns em cada linha (clone) sejam consecutivos. Este problema é denominado problema dos uns consecutivos.*

Mesmo que uma permutação que tenha a C1P exista, não se pode afirmar que tal permutação é a verdadeira, pois podem existir várias permutações com a C1P para uma dada matriz. Pode-se dizer que todas as permutações que fazem todos os 1s serem

consecutivos são candidatas para a verdadeira permutação. Por isto, um algoritmo para este problema deveria encontrar todas as possíveis soluções.

Verificar se a matriz possui essa propriedade e então encontrar uma permutação válida é um problema bem conhecido para o qual um algoritmo de complexidade de tempo polinomial existe [SM97]. Na seção seguinte será descrito um algoritmo para este problema, sendo outras soluções encontradas em [Hsu92, BL76, AS95, CY91].

Porém, problemas NP-difíceis são produzidos ao se usar generalizações do problema, como por exemplo, ao invés de requerer um bloco de 1s consecutivos por linha, tentar encontrar uma permutação de colunas tal que cada linha tenha no máximo k blocos de 1s consecutivos ou se a consideração de que as sondas são únicas for relaxada.

4.2 Algoritmo de Fulkerson e Gross

Fulkerson e Gross [FG65] propuseram um método simples para processar uma matriz binária M, $n \times m$, para verificar se ela possui a propriedade dos uns consecutivos para as linhas. A meta do algoritmo é encontrar uma permutação de colunas tal que em cada linha todos os uns sejam consecutivos. Para simplificar, supõe-se que todas as linhas são diferentes, ou seja, nenhum clone tem a mesma impressão digital, e nenhuma linha é toda preenchida com zeros, ou seja, cada clone hibridiza com pelo menos uma sonda. Tais hipóteses simplificadoras não limitam a utilização do algoritmo, dado que é possível eliminar as linhas com apenas zeros e linhas repetidas, ou desconsiderá-las, no processamento, e incluí-las novamente na matriz final (a linha repetida possuirá a mesma permutação da linha que permaneceu na matriz).

O algoritmo utiliza certas relações entre as linhas para agrupá-las em componentes e permutá-las, definidas a seguir.

Definição 3 *Para cada linha i de M, seja S_i o conjunto de colunas k onde $M_{ik} = 1$. Dadas duas linhas i e j três situações podem ocorrer:*

1. $S_i \cap S_j = \emptyset$;
2. $S_i \subseteq S_j$ *ou* $S_j \subseteq S_i$;
3. $S_i \cap S_j \neq \emptyset$, *e nenhum deles é subconjunto do outro.*

Claramente, pode-se lidar separadamente com as linhas que se enquadram na primeira situação, pois a permutação das colunas correspondentes aos elementos de S_i não interfere

com a permutação das colunas correspondentes aos elementos de S_j. Em tal caso, as duas linhas pertencerão a componentes separados e serão processadas separadamente. As linhas cujos conjuntos não possuem interseção vazia podem ser inicialmente agrupadas em um componente. Suponha que haja uma linha k neste componente tal que S_k ou é um subconjunto de S_i ou é disjunto de S_i, para todo $i \neq k$ no componente. Pode-se deixar k (e todas as linhas com a mesma propriedade) fora deste componente em decorrência da disjunção de seu conjunto com os conjuntos de outras linhas. Neste caso, a permutação das colunas das linhas com conjuntos disjuntos será independente.

A partir das regras acima, determina-se o modo que as linhas são agrupadas em componentes. Para isto, constrói-se um grafo não direcionado G_C usando a matriz M. Em G_C cada vértice será uma linha de M. Há uma aresta entre os vértices i e j se $S_i \cap S_j$ é diferente de $\emptyset$ e nenhum deles é subconjunto do outro. Os componentes de linhas serão os componentes conexos de G_C. Assim, cada componente é uma sub-matriz de M com o mesmo número de colunas, podendo ter um número menor de linhas. Como um exemplo [SM97], a Figura 4.2 mostra o grafo G_C para a matriz M da Figura 4.1.

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
l_1	1	1	0	1	1	0	1	0	1
l_2	0	1	1	1	1	1	1	1	1
l_3	0	1	0	1	1	0	1	0	1
l_4	0	0	1	0	0	0	0	1	0
l_5	0	0	1	0	0	1	0	0	0
l_6	0	0	0	1	0	0	1	0	0
l_7	0	1	0	0	0	0	1	0	0
l_8	0	0	0	1	1	0	0	0	1

Figura 4.1: Uma matriz binária M

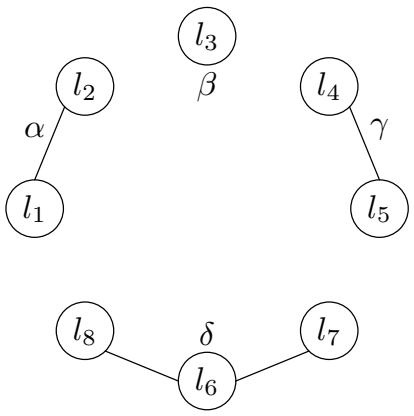


Figura 4.2: Grafo G_C correspondente à matriz M da Figura 4.1 (componentes conexos indicados por letras gregas)

Se cada componente de M possuir a C1P, a matriz M também possuirá esta propriedade. Se um componente possui até duas linhas, claramente, este componente possui a propriedade, pois a permutação das suas colunas pode ser feita de maneira simples e direta. Assim, deve-se colocar consecutivamente as colunas que contêm 1 apenas na primeira linha, a partir do início da matriz. As colunas com 1 nas duas linhas devem ser colocadas em seguida. Então, devem ser colocadas as colunas que contêm 1 apenas na segunda linha. Por fim, são colocadas as colunas com zero em ambas as linhas.

Agora, um algoritmo pode ser esquematizado: deve-se separar as linhas em componentes, como descrito anteriormente, deve-se permutar as colunas de cada componente de modo que cada linha possua todos os 1s consecutivos, e então deve-se juntar os componentes. Isto leva a dois subproblemas: como encontrar a permutação correta para um componente, e como unir os componentes.

Na Figura 4.3 é apresentado o algoritmo para resolução do problema dos 1s consecutivos, descrito em alto nível.

Entrada: Matriz binária M, $n \times m$.
Saída: Matriz binária M', $n \times m$, com todos os 1s em cada linha dispostos em colunas consecutivas, ou uma mensagem dizendo que M não possui a C1P.
Início
→ Cria grafo G_C:
- Compara as linhas de M entre si
- Separa as linhas de M em componentes (componentes conexos de G_C)

→ Para cada componente c de M
- Permuta colunas do componente c, através da busca em profundidade em G_C começando por um vértice de grau 1
  - A linha de M (vértice de G_C), encontrada na busca em profundidade, é inserida, com seus 1s consecutivos, no componente permutado correspondente
  - Após inserção, se o número de 1s nas mesmas colunas entre duas linhas inseridas não for igual ao observado em M, M não tem a C1P e o algoritmo termina
- Gera conjunto de 1s de cada coluna do componente c

→ Cria grafo G_M:
- Cada vértice de G_M corresponde a um componente conexo de G_C e as arestas direcionadas indicam os componentes que possuem linhas com conjuntos que contêm os conjuntos das linhas dos componentes apontados

→ Ordena topologicamente G_M
→ Une os componentes permutados de M seguindo a ordem definida na ordenação topológica, e forma a matriz binária final com colunas permutadas
Fim

Figura 4.3: Algoritmo de Fulkerson e Gross

4.2.1 Permutação de Colunas em Cada Componente

Para permutar as colunas de um componente de forma que cada linha possua todos os 1s consecutivos, essas linhas são inseridas no componente uma de cada vez. Isto é feito após a construção de G_C. Se a permutação for feita simultaneamente à criação de G_C pode ocorrer que dois componentes ainda em formação, com as colunas de suas linhas permutadas, sejam unidos. Isto fará com que um dos componentes tenha que ser permutado novamente para que as interseções entre todos os conjuntos de colunas das linhas do novo componente sejam iguais às interseções observadas em M.

A matriz da Figura 4.4 será utilizada como exemplo [SM97] para descrever os passos do algoritmo. Inicialmente, para inserir uma primeira linha (l_1) no componente, as colunas são permutadas de forma que todos os 1s fiquem consecutivos. Note que se há k 1s nesta linha, haverá $k!/2$ possíveis permutações. Permutações invertidas não são consideradas como soluções distintas, porque elas são as mesmas permutações obtidas, onde muda-se apenas a ordem de leitura das sondas. Pode-se codificar todas as soluções pela associação de um conjunto de possíveis colunas a cada elemento igual a 1 em uma dada linha, como mostrado na Figura 4.5. Nessa figura, cada coluna que possui 1 pode representar tanto a coluna 2, como a 7, ou a coluna 8 de M. A escolha da ordem destas três colunas define a permutação escolhida (como há apenas uma linha, qualquer permutação das três colunas mantém os 1s consecutivos).

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
l_1	0	1	0	0	0	0	1	1
l_2	0	1	0	0	1	0	1	0
l_3	1	0	0	1	0	0	1	1

Figura 4.4: Uma segunda matriz binária M

$$l_1 \quad \rightarrow \quad \ldots \quad 0 \quad \overset{\{2,7,8\}}{1} \quad \overset{\{2,7,8\}}{1} \quad \overset{\{2,7,8\}}{1} \quad 0 \quad \ldots$$

Figura 4.5: Primeira linha (l_1) de M

Para a segunda linha (l_2) do mesmo componente, permuta-se as colunas com 1s de l_2 em relação às colunas com 1s de l_1. Como l_2 está no mesmo componente de l_1, l_2 possui 1s em algumas colunas onde l_1 também possui, e outros 1s onde l_1 não possui. Para colocar l_2 de uma maneira consistente em relação a l_1, há duas escolhas: colocar os 1s que pertencem apenas a l_2 à esquerda dos 1s de l_1, ou colocá-los à direita. Colocando as colunas à esquerda da permutação de l_1 resultará num conjunto de permutações. Se as

colunas fossem colocadas à direita, resultaria no mesmo conjunto, só que com a ordem inversa, e esta é uma observação chave. Na Figura 4.6, o 1 exclusivo de l_2 foi colocado à esquerda dos 1s de l_1.

				$\{5\}$	$\{2,7\}$	$\{2,7\}$	$\{8\}$		
l_1	$\rightarrow$	$\ldots$	0	0	1	1	1	0	$\ldots$
l_2	$\rightarrow$	$\ldots$	0	1	1	1	0	0	$\ldots$

Figura 4.6: Inserção da segunda linha (l_2) de M

Repare que os conjuntos de cada coluna devem refletir as sobreposições dos 1s entre l_1 e l_2. A linha l_2 possui 1s nas colunas 2, 5 e 7, formando o conjunto $\{2, 5, 7\}$. Quando esta linha é inserida, as colunas que se sobrepõem com as colunas de l_1 formam um conjunto contendo a interseção entre S_1 e S_2, ou seja, $\{2, 7\}$. A única coluna onde l_2 possui 1 e l_1 possui 0 é coluna 5, e tal informação é expressa pelo conjunto $\{5\}$ na coluna à esquerda dos 1s da interseção entre S_1 e S_2 (pois l_2 foi escolhida para ser posta à esquerda de l_1). Na coluna 8, entretanto, l_1 possui 1 e l_2 possui 0. Como a coluna 8 pertence a S_1, ela deve ser colocada na direção oposta àquela em que as colunas exclusivas de S_2 foram colocadas.

Se l_2 fosse colocada à direita de l_1, seria formada a configuração exposta na Figura 4.7. Note que a ordem das colunas foi apenas invertida, e a permutação de colunas permanece válida.

				$\{8\}$	$\{2,7\}$	$\{2,7\}$	$\{5\}$		
l_1	$\rightarrow$	$\ldots$	0	1	1	1	0	0	$\ldots$
l_2	$\rightarrow$	$\ldots$	0	0	1	1	1	0	$\ldots$

Figura 4.7: Inserção, à direita de l_1, da segunda linha (l_2) de M

Considere agora a inserção de uma terceira linha (l_3) nesse mesmo componente. Pelo fato dela estar no mesmo componente que l_1 e l_2, sabe-se que em G_C pode-se encontrar uma aresta (l_3, l_1), ou (l_3, l_2), ou ambas. Deve-se colocar l_3 com relação a l_2, mas agora levando em consideração a relação entre l_3 e l_1. O modo de fazê-lo é considerando o número de elementos na interseção entre S_1, S_2 e S_3. Se $|S_1 \cap S_3| < min(|S_1 \cap S_2|, |S_2 \cap S_3|)$, l_3 precisa estar na mesma direção que l_2 foi colocada em relação a l_1; caso contrário, l_3 irá na direção oposta a l_2.

Tal intuição decorre do modo em que as colunas podem ser dispostas. Por exemplo, suponha que l_2 está colocada à direita de l_1, e $|S_1 \cap S_3| < min(|S_1 \cap S_2|, |S_2 \cap S_3|)$. Uma vez que S_3 possui menos interseções com S_1 do que $|S_1 \cap S_2|$, é coerente colocar l_3 (em relação a l_2) na mesma direção em que l_2 foi posta (em relação a l_1), pois tal ação deixa os 1s de l_3 mais distantes dos 1s de l_1. Assim, seguindo o exemplo, l_3 fica à direita de l_2 e

mais à direita ainda de l_1, diminuindo o número de interseções entre S_3 e S_1 e respeitando o número de interseções entre S_3 e S_2. Considere agora que l_2 esteja colocada à direita de l_1, e $|S_1 \cap S_3| > min(|S_1 \cap S_2|, |S_2 \cap S_3|)$. Como existem mais interseções entre S_3 e S_1, colocar l_3 à esquerda de l_2 (na direção oposta) resulta em fazer com que mais 1s de l_3 formem interseções com 1s de l_1 (repare que os 1s de l_1 também estão à esquerda dos 1s de l_2). Como resultado, haverá mais interseções entre l_3 e l_1 do que as outras interseções, refletindo as informações contidas na matriz M.

Tudo o que deve ser feito para inserir uma nova linha k é encontrar duas linhas previamente colocadas, i e j, tais que existam arestas (k, i) e (i, j) em G_C, e proceder como feito com l_3. No exemplo da matriz da Figura 4.4, $|S_1 \cap S_3| = 2$, que é maior que $min(|S_1 \cap S_2|, |S_2 \cap S_3|) = 1$. Assim, l_3 será colocada na direção oposta daquela em que l_2 foi colocada em relação a l_1, ou seja, l_3 irá para a direita em relação a l_2, como mostrado na Figura 4.8.

				$\{5\}$	$\{2\}$	$\{7\}$	$\{8\}$	$\{1,4\}$	$\{1,4\}$		
l_1	$\rightarrow$	...	0	0	1	1	1	0	0	0	...
l_2	$\rightarrow$	...	0	1	1	1	0	0	0	0	...
l_3	$\rightarrow$	...	0	0	0	1	1	1	1	0	...

Figura 4.8: Inserção da terceira linha (l_3) de M

Os vértices k, i, j, de G_C interligados por arestas (k, i) e (i, j) são denominados tripla rígida, e a permutação das colunas é feita caminhando em qualquer ordem pela tripla rígida. Assim, o primeiro vértice da tripla rígida, que teve suas colunas permutadas, serve de referência para os outros dois vértices que terão suas colunas permutadas. A partir da permutação das colunas da terceira linha do componente será sempre necessário verificar a relação entre a linha a ter suas colunas permutadas e as outras duas linhas, já permutadas, que formam sua tripla rígida. Desta forma, é interessante que os componentes conexos de G_C sejam árvores, propiciando a fácil identificação e processamento das triplas rígidas. Pode-se, por exemplo, realizar uma busca em profundidade em cada componente de G_C para determinar a ordem em que as colunas de cada linha de M serão permutadas.

Teste da C1P

Ao permutar as colunas de cada linha de um componente, deve-se verificar se o número de interseções de 1s entre a nova linha e a primeira linha da sua tripla rígida é o mesmo que o número de interseções existentes em M, antes da permutação. Se o número de interseções for diferente então o componente não possui a C1P e, conseqüentemente, M

também não tem a C1P. Este teste deve ser feito a partir da permutação de colunas da terceira linha do componente.

Pode-se citar, como um exemplo de uma matriz sem a C1P, a matriz M' da Figura 4.9. Seu grafo G_C, sem ciclos, possui apenas um componente, como ilustrado na Figura 4.10.

	c_1	c_2	c_3	c_4	c_5	c_6
l_1	1	0	0	0	0	1
l_2	0	1	1	0	1	1
l_3	1	0	1	1	1	0

Figura 4.9: Matriz binária M' sem a C1P

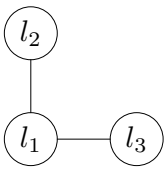


Figura 4.10: Grafo G_C correspondente a M'

É realizada a busca em profundidade no grafo G_C da Figura 4.10, a partir de l_2. A permutação das linhas l_2 e l_1 é realizada conforme descrito anteriormente (colocando l_1 à esquerda de l_2, por exemplo), sendo mostrada na Figura 4.11. As interseções destas duas linhas permanecem as mesmas, onde só uma coluna (c_6) possui 1 nas duas linhas.

		$\{1\}$	$\{6\}$	$\{2,3,5\}$	$\{2,3,5\}$	$\{2,3,5\}$
l_2	$\rightarrow$	0	1	1	1	1
l_1	$\rightarrow$	1	1	0	0	0

Figura 4.11: Permutação das colunas das linhas l_2 e l_1 de M'

Por fim, é inserida l_3 verificando-se a inequação $|S_3 \cap S_2| < min(|S_1 \cap S_3|, |S_1 \cap S_2|)$. Dado que $|S_3 \cap S_2| = 2$ e $min(|S_1 \cap S_3|, |S_1 \cap S_2|) = 1$, l_3 deve ser colocada à direita (direção oposta) de l_1, resultando na matriz permutada da Figura 4.12 com os conjuntos correspondentes. Note que as colunas com 1 que não correspondem a nenhuma coluna de M' possuem conjunto vazio.

		$\{\,\}$	$\{\,\}$	$\{3,5\}$	$\{3,5\}$	$\{3,5\}$
l_2	$\rightarrow$	0	1	1	1	1
l_1	$\rightarrow$	1	1	0	0	0
l_3	$\rightarrow$	0	1	1	1	1

Figura 4.12: Matriz binária M' permutada pelo do algoritmo de Fulkerson e Gross

Os conjuntos S_3 e S_1 possuem uma interseção, portanto o 1 na linha l_3 e na segunda coluna da matriz da Figura 4.12 teve de ser inserido, e os 1s restantes de l_3 foram postos consecutivamente à direita do 1 da interseção. Porém esta configuração alterou o número de interseções entre S_2 e S_3, mostrando que a matriz da Figura 4.9 não possui a propriedade dos uns consecutivos.

O problema da matriz M' da Figura 4.9 é que as colunas onde l_1 possui 1 têm a configuração mostrada na Figura 4.13. Esta configuração impõe que, para que os 1s de l_1 sejam consecutivos, os 1s de l_2 fiquem de um lado, e os 1s de l_3 fiquem do outro lado (o lado em que os 1s de cada linha serão colocados depende da ordem entre c_1 e c_6). Entretanto, l_2 e l_3 possuem interseções, e como conseqüência, é impossível permutar as colunas de tal forma que em cada linha todos os 1s sejam consecutivos.

	c_1	c_6
l_1	1	1
l_2	0	1
l_3	1	0

Figura 4.13: Colunas com os 1s de l_1 de M'

4.2.2 União de Componentes

Para realizar a junção de componentes é necessário outro grafo, o grafo direcionado G_M. Neste grafo, cada componente da matriz original M será um vértice. Uma aresta dirigida existirá do vértice α para o vértice β se, para toda linha i do componente β, o conjunto S_i está contido em pelo menos um conjunto S_j do componente α. O grafo G_M correspondente aos componentes da matriz M da Figura 4.1 é mostrado na Figura 4.14.

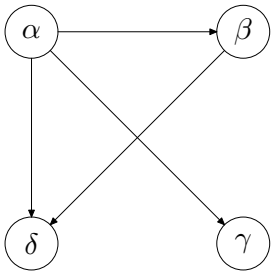


Figura 4.14: Grafo G_M correspondente aos componentes da matriz M da Figura 4.1

A união de componentes depende de como os conjuntos em um componente contêm ou estão contidos nos conjuntos de outro componente. Intuitivamente, devem ser

processados primeiro os componentes que possuem conjuntos não contidos em nenhum outro componente. Visto que a relação de inclusão é dada pela direção das arestas, deve-se processar G_M em ordem topológica [CLR90].

Depois de ter permutado corretamente as colunas de todos os componentes individualmente, são selecionados todos os vértices de G_M sem arestas incidentes, e suas colunas são fixadas. A seguir, é tomado o próximo vértice na ordem topológica. Tomando-se, por exemplo, a aresta (α, β), isto significa que é necessário encontrar algum tipo de "coluna de referência" em α que indicará como colocar as linhas de β. Deve-se escolher a linha l de β que possui o 1 mais à esquerda e denominar a coluna respectiva de c_β. Sabe-se que S_l está contido em alguns conjuntos S_i de α, mas não está contido em outros conjuntos de α. Deve-se encontrar todas as linhas de α que contêm S_l, e encontrar a coluna mais à esquerda onde todas estas linhas possuem 1, chamando esta coluna de c_α. Esta é a coluna de referência procurada, uma vez que se pode fazer c_α e c_β serem a mesma coluna. Assim, une-se os dois componentes copiando, da esquerda para a direita, as colunas de β para α, a partir da coluna c_α.

Para ilustrar esse processo serão usados a matriz da Figura 4.1 e o grafo da Figura 4.14, e seus componentes serão considerados em ordem topológica. A ordenação seguida é α, β, δ, γ. O componente α já com suas colunas permutadas é mostrado na Figura 4.15. Depois o componente β, mostrado na Figura 4.16, deve ser unido ao componente α.

Procura-se a coluna com o 1 mais à esquerda em β, que neste caso é a primeira coluna (esta coluna é o c_β desejado). O conjunto S_l da linha que contém esta coluna é $\{2, 4, 5, 7, 9\}$, e ele está contido nas duas linhas de α. Como a segunda coluna de α é a coluna mais à esquerda que possui 1s em todas as linhas que contêm S_l, esta coluna é portanto c_α. Assim, a união entre os componentes é feita copiando os 1s de l_3 para α, a partir de c_α e colocando 0 nas outras colunas que não foram preenchidas, como mostrado na Figura 4.17.

			$\{1\}$	$\{2,4,5,7,9\}$				$\{3,6,8\}$			
l_1	$\rightarrow$	$\ldots$	1	1	1	1	1	0	0	0	$\ldots$
l_2	$\rightarrow$	$\ldots$	0	1	1	1	1	1	1	1	$\ldots$

Figura 4.15: Linhas do componente α, com 1s consecutivos

			$\{2,4,5,7,9\}$					
l_3	$\rightarrow$	$\ldots$	1	1	1	1	1	$\ldots$

Figura 4.16: Linha do componente β, com 1s consecutivos

			{1}	{2,4,5,7,9}					{3,6,8}			
l_1	$\rightarrow$	...	1	1	1	1	1	1	0	0	0	...
l_2	$\rightarrow$	...	0	1	1	1	1	1	1	1	1	...
l_3	$\rightarrow$	...	0	1	1	1	1	1	0	0	0	...

Figura 4.17: União de l_3 com as linhas de α

O próximo componente é δ, mostrado na Figura 4.18, já com suas colunas permutadas. É possível ver que neste componente, l_8 possui o 1 mais à esquerda (nas colunas 5 e 9). No componente α ambas as linhas l_1 e l_2 contêm l_8 (note que são analisadas as linhas do componente α apenas, e não as linhas de outros componentes que foram inseridas em passos anteriores), e a segunda coluna, representando um leque de colunas possíveis, é a que possui todos os 1s. Então esta segunda coluna deve coincidir com a coluna 5 ou 9 do componente δ, resultando na Figura 4.19.

			{5,9}		{4}	{7}	{2}	
l_6	$\rightarrow$	...	0	0	1	1	0	...
l_7	$\rightarrow$	...	0	0	0	1	1	...
l_8	$\rightarrow$	...	1	1	1	0	0	...

Figura 4.18: Linhas do componente δ, com 1s consecutivos

			{1}	{5,9}		{4}	{7}	{2}	{3,6,8}			
l_1	$\rightarrow$	...	1	1	1	1	1	1	0	0	0	...
l_2	$\rightarrow$	...	0	1	1	1	1	1	1	1	1	...
l_3	$\rightarrow$	...	0	1	1	1	1	1	0	0	0	...
l_6	$\rightarrow$	...	0	0	0	1	1	0	0	0	0	...
l_7	$\rightarrow$	...	0	0	0	0	1	1	0	0	0	...
l_8	$\rightarrow$	...	0	1	1	1	0	0	0	0	0	...

Figura 4.19: União do componente δ

Finalmente, o componente γ mostrado na Figura 4.20 é unido. A linha com o 1 mais à esquerda é l_5, e no componente α apenas l_2 contém l_5. A segunda coluna de l_2 é a coluna mais à esquerda, tal que todas as linhas que contém l_5, possuem o valor 1. Porém, nesta coluna, e da terceira a sexta colunas, os conjuntos não contêm os elementos 3 e 6 de l_5. Portanto deve-se utilizar como c_α a coluna mais à esquerda cujo conjunto de elementos tenha relação com algum conjunto de l_5[1]. Esta coluna é a sétima, com o conjunto $\{3,6,8\}$ e ela é utilizada como c_α, obtendo a matriz final com todos os 1s consecutivos, mostrada na Figura 4.21, correspondente à matriz M da Figura 4.1.

[1]Se a segunda coluna fosse a escolhida, indicaria que l_1 também conteria l_5. Portanto, a coluna c_α escolhida deve ter 0 em l_1.

			{6}	{3}	{8}	
l_4	$\rightarrow$	...	0	1	1	...
l_5	$\rightarrow$	...	1	1	0	...

Figura 4.20: Linhas do componente γ, com 1s consecutivos

		{1}	{5,9}		{4}	{7}	{2}	{6}	{3}	{8}
l_1	$\rightarrow$	1	1	1	1	1	1	0	0	0
l_2	$\rightarrow$	0	1	1	1	1	1	1	1	1
l_3	$\rightarrow$	0	1	1	1	1	1	0	0	0
l_6	$\rightarrow$	0	0	0	1	1	0	0	0	0
l_7	$\rightarrow$	0	0	0	0	1	1	0	0	0
l_8	$\rightarrow$	0	1	1	1	0	0	0	0	0
l_4	$\rightarrow$	0	0	0	0	0	0	0	1	1
l_5	$\rightarrow$	0	0	0	0	0	0	1	1	0

Figura 4.21: União do componente γ e obtenção da matriz final

Deve-se notar que um dos conjuntos de colunas não é único, indicando que foram encontradas duas soluções. Neste caso em particular, a razão é que as colunas 5 e 9 são idênticas. Mas em geral podem haver múltiplas soluções que não envolvem apenas permutações de colunas idênticas. Por exemplo, $\{1, 2, 7, 4, 9, 5, 8, 3, 6\}$ é outra solução para a mesma matriz. Múltiplas soluções podem existir pois G_M pode permitir diferentes ordenações topológicas, e porque cada componente pode ter várias soluções e cada uma delas pode ser usada de dois modos (a permutação e sua reversa).

A complexidade do algoritmo de Fulkerson e Gross para o problema dos uns consecutivos é $O(n \times m)$, dado que este é composto pela criação de G_C que leva tempo $O(n \times m)$ (é necessário ordenar as linhas, pelo número de 1s de cada linha, usando *radix-sort* e gerar uma matriz auxiliar que contém informação para definir a relação entre as linhas), pelo processamento de cada componente que leva tempo $O(n \times m)$, pela ordenação topológica que leva tempo $O(n + m)$ (é possível pré-processar as entradas de M de modo que as consultas necessárias quando se percorre G_M tomem tempo constante, este pré-processamento pode ser feito em tempo $O(n \times m)$) e pela união dos componentes que leva tempo $O(n \times m)$ [SM97].

Capítulo 5

Solução Software/Hardware do Problema dos Uns Consecutivos

Este capítulo descreve um conjunto de implementações híbridas do algoritmo de Fulkerson e Gross que soluciona o problema dos 1s consecutivos para o mapeamento físico de DNA. Em tais soluções, parte do algoritmo é realizada em software (SW) e operações específicas do mesmo são executadas em hardware (HW). Desta forma, é necessário resolver o particionamento SW/HW, definindo quais operações serão executadas no processador e quais serão mapeadas para a FPGA. As operações que possuem controle e estruturas de dados complexas serão executadas no processador e as operações que podem ser aceleradas pelo hardware reconfigurável são identificadas e mapeadas na FPGA [BP00].

5.1 Implementação Híbrida

As implementações híbridas consistem de um componente de propósito geral, o processador convencional, e um componente reconfigurável (a FPGA), no qual é mapeado um componente específico para a aplicação.

O algoritmo tem suas operações divididas, onde as áreas do código que não podem ser mapeadas facilmente em lógica reconfigurável são executadas no PC hospedeiro, e as áreas com grande densidade de computação e que podem se beneficiar da implementação em hardware são executadas na FPGA. O objetivo, com a implementação, é obter um ganho de desempenho através da execução em hardware de trechos do algoritmo dos 1s consecutivos que antes eram realizados em software. Desta forma, pretende-se superar as limitações encontradas pelo programa em software, que utiliza o controle seqüencial e o

conjunto de instruções genérico do processador convencional.

A implementação híbrida do algoritmo para o problema dos 1s consecutivos foi desenvolvida utilizando-se um *kit* de desenvolvimento composto por uma placa contendo uma FPGA com uma interface de rede. Assim, alguns trechos do código do algoritmo em software foram substituídos por chamadas a funções que realizam a comunicação com a FPGA e definem vários parâmetros necessários para o processamento no hardware configurável.

Devido às características da placa que contém a FPGA, o acoplamento entre o processador hospedeiro e a FPGA é fraco e, portanto, a comunicação mais rápida disponível é realizada através das interfaces de rede de ambos os dispositivos. O modelo da arquitetura utilizado na implementação híbrida é mostrado na Figura 5.1. Com esta arquitetura, o tempo de comunicação pode causar um grande impacto no tempo de execução final do programa, e por isto, a estratégia de implementação escolhida para a comunicação é muito importante.

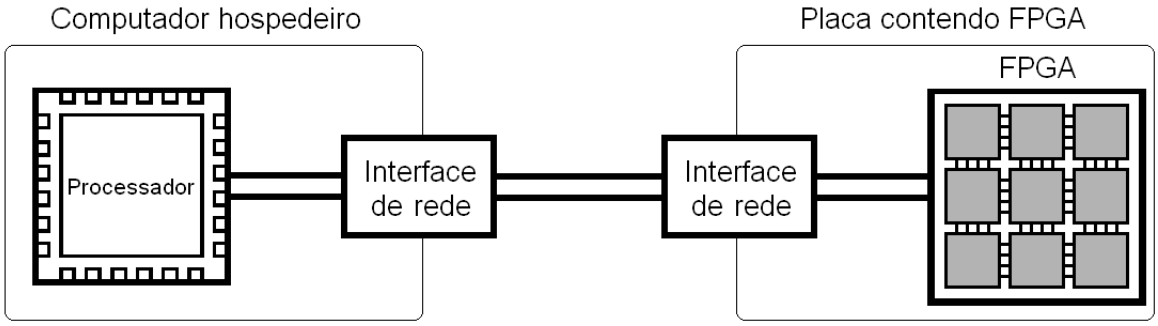


Figura 5.1: Modelo da arquitetura híbrida utilizado

A Figura 5.2 ilustra o processo de substituição dos trechos do código do programa em software (solução em SW) por chamadas a funções que realizam a comunicação com a FPGA, obtendo assim uma solução híbrida (solução SW/HW). Tal solução implementa portanto o particionamento SW/HW projetado.

Início

→ Cria grafo G_C usando M $(n \times m)$

Para cada par de linhas i e j de M **Faça**

Para k de 1 **Até** m **Faça**

Se $M_{ik} = 1$ **E** $M_{jk} = 1$ **Então**

Linha i e j tem interseção

Senão Se $M_{ik} = 1$ **E** $M_{jk} = 0$ **Então**

Linha i não é contida por linha j

Senão Se $M_{ik} = 0$ **E** $M_{jk} = 1$ **Então**

Linha j não é contida por linha i

Fim Se

FimPara

Se Todas condições satisfeitas **Então**

Adiciona aresta (i, j) em G_C

Fim Se

Fim Para

→ Para cada componente c de M

- Permuta colunas do componente c
- Gera conjunto de cada coluna de c

→ Cria e ordena topologicamente G_M

→ Une os componentes

Fim

(a) Solução em SW

Início

→ Cria grafo G_C usando M $(n \times m)$

Para cada par de linhas i e j de M **Faça**

Comunica_FPGA(i, j, condições)

Se Todas condições satisfeitas **Então**

Adiciona aresta (i, j) em G_C

Fim Se

Fim Para

→ Para cada componente c de M

- Permuta colunas do componente c
- Gera conjunto de cada coluna de c

→ Cria e ordena topologicamente G_M

→ Une os componentes

Fim

(b) Solução SW/HW

Figura 5.2: Mapeamento de instruções da solução em SW em comunicação com FPGA na solução SW/HW

Uma implementação híbrida do algoritmo do problema dos 1s consecutivos é ilustrada na Figura 5.3. Como mostrado nesta figura, o módulo reconfigurável é capaz de realizar em paralelo várias comparações que eram executadas seqüencialmente no algoritmo em SW, proporcionando ganhos de desempenho que não são possíveis apenas com a otimização do código equivalente em software.

Processador

Início
→ Cria grafo G_C usando M $(n \times m)$
Para cada par de linhas i e j de M **Faça**
Comunica_FPGA(i, j, condições)
Se Todas condições satisfeitas **Então**
Adiciona aresta (i, j) em G_C
Fim Se
Fim Para
→ Para cada componente c de M
• Permuta colunas do componente c
• Gera conjunto de cada coluna de c
→ Cria e ordena topologicamente G_M
→ Realiza união dos componentes
Fim

Componente Reconfigurável

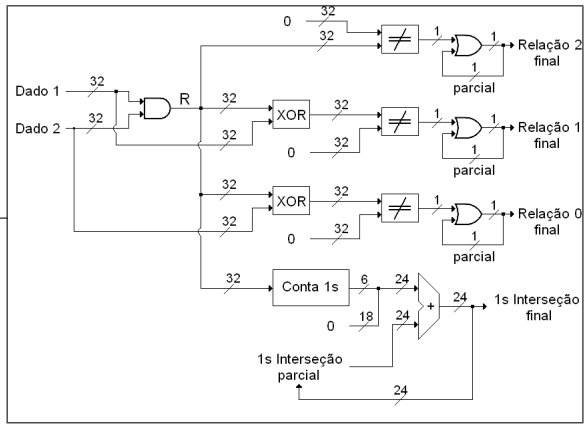


Figura 5.3: Implementação híbrida

Através dos recursos oferecidos pela FPGA foi possível o desenvolvimento de várias soluções que exploram diferentes formas de implementação das operações realizadas e suas interações com o código executado no processador hospedeiro. Com isto, foi possível comparar estas diversas implementações e compreender os impactos que determinadas escolhas de projeto acarretam para o desempenho do programa.

Foram identificados, através de um *profile* da implementação em SW, os trechos do algoritmo que consomem a maior parte do tempo de execução. Assim, no particionamento SW/HW foram selecionadas para implementação em hardware duas operações críticas para o tempo de execução total do algoritmo, que serão descritas nas seções seguintes. A primeira operação realizada é a comparação de clones, onde são feitas operações que indicam se dois clones pertencem ao mesmo componente e qual o número de sondas em comum entre estes dois clones. A segunda operação realiza a construção de conjuntos de colunas de cada componente, após a permutação de suas colunas.

A implementação do circuito de HW da solução híbrida é composta por alguns módulos principais, cada um realizando uma função específica:

- Controle: Responsável pelo gerenciamento dos demais módulos e pelo controle da execução das operações implementadas em HW.
- Compara linhas: Responsável pela comparação entre duas linhas e contagem do número de interseções de ambas.

- Constrói conjuntos: Responsável pela construção dos conjuntos de colunas de um componente.
- Recebe dados da rede: Recebe da rede de comunicação os dados oriundos da parte em SW do programa e os repassa para o controle.
- Envia dados pela rede: Envia pela rede de comunicação, para a parte em SW do programa, os resultados da comparação de linhas ou da criação de conjuntos.

Além disso, a implementação em HW pode utilizar bancos de memória da FPGA aumentando sua capacidade de processamento. A estrutura geral do relacionamento entre os módulos que compõem a implementação do HW da solução híbrida é mostrada na Figura 5.4. Nas seções seguintes cada um destes módulos é descrito em detalhes.

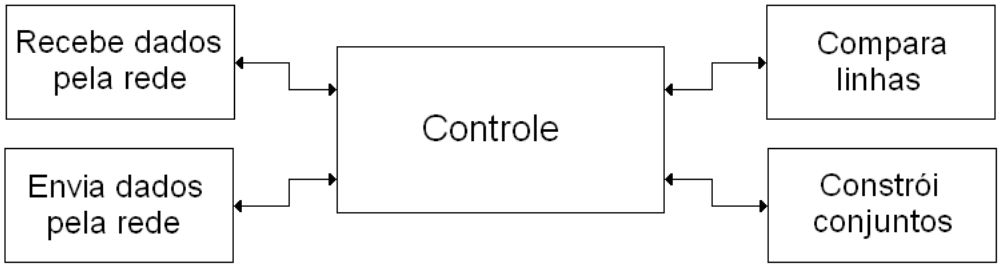


Figura 5.4: Visão geral dos módulos do hardware da implementação híbrida

5.2 Comparação de Clones

A comparação de linhas de M (clones) determina se duas linhas pertencem ao mesmo componente e também determina quantas interseções (o número de 1s nas mesmas colunas das duas linhas) tais linhas possuem. Para que duas linhas pertençam ao mesmo componente, como descrito no Capítulo 4, é necessário que ambas tenham 1 na mesma coluna e que possuam 1 em alguma coluna onde a outra linha possui 0. O número de interseções é utilizado no momento da permutação das linhas de cada componente, para determinar a direção que a nova linha, com suas colunas permutadas, será colocada em relação às linhas que já tiveram suas colunas permutadas e para orientar a posição de colocação desta nova linha de modo que ela mantenha o número de interseções com a linha, já permutada, que é a sua vizinha no grafo G_C.

A implementação em hardware desta operação consiste em realizar as comparações e contagens das interseções por blocos. Cada linha é convertida em uma seqüência de bits

pelo programa em software e enviada para a FPGA. O hardware realiza comparações de blocos de cada linha em um único ciclo de *clock*, até que o par de linhas sendo comparadas seja operado por completo, conforme ilustrado na Figura 5.5. Assim, no 1º ciclo o bloco 1 do clone 1 é comparado com o bloco 1 do clone 2; no 2º ciclo o bloco 2 do clone 1 é comparado com o bloco 2 do clone 2, e assim sucessivamente. Desta forma, o número de ciclos gastos nesta operação será o tamanho da linha de M (o número de colunas) dividido pelo tamanho do bloco (em bits). As implementações desenvolvidas usam blocos de 32 bits.

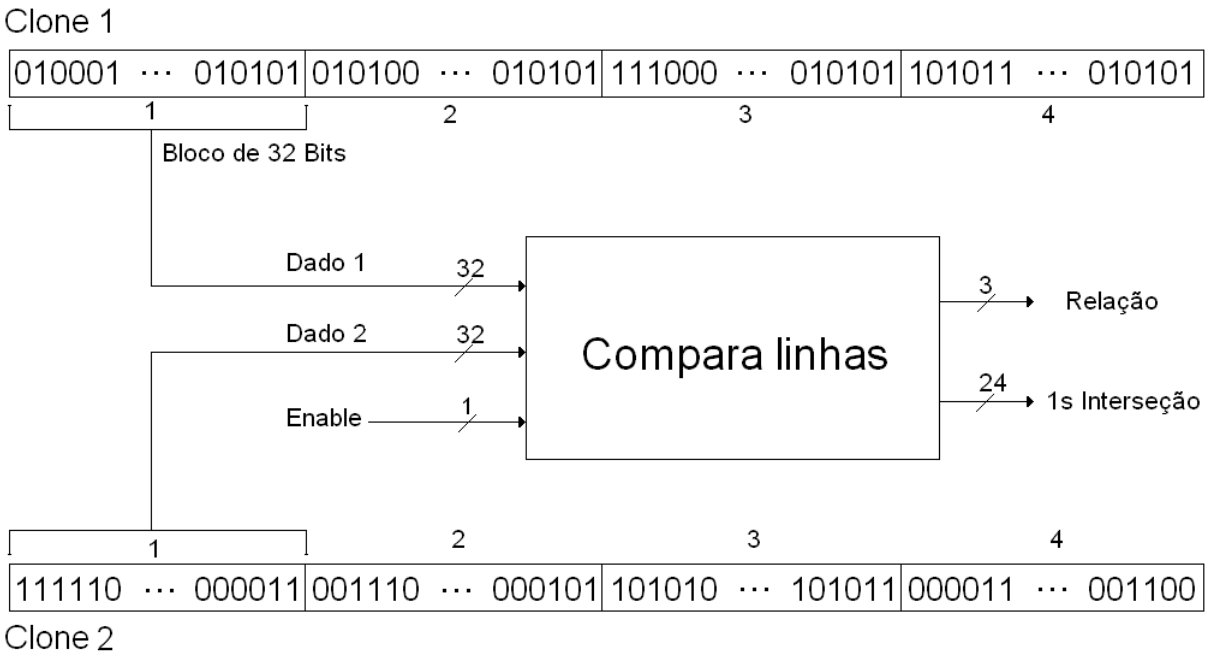


Figura 5.5: Comparação das linhas em blocos

O resultado da relação entre os operandos (operandos, neste caso, são os blocos de cada linha), obtido com a comparação, é codificado em 3 bits. O bit mais significativo (Relação 2) indica se houve ou não (1 ou 0, respectivamente) interseção entre os dois operandos. O bit seguinte (Relação 1) indica se o primeiro operando possui 1 em alguma coluna onde o segundo operando possui 0, ou seja, se a primeira linha não é contida pela segunda. O bit menos significativo (Relação 0) indica o mesmo que o bit do meio, só que considerando o segundo operando em relação ao primeiro. Os possíveis valores do resultado da relação entre as linhas são mostrados na Tabela 5.1.

No mesmo ciclo, o comparador de linhas realiza as seguintes operações:

- É feita uma operação AND entre os dois operandos, obtendo um resultado intermediário R de 32 bits.

- Em paralelo são feitas as seguintes operações:
  - R é comparado com uma seqüência de 32 bits toda em 0 ($ZERO$). Se R for diferente de $ZERO$, os operandos possuem interseção, e o bit Relação 2 é definido em 1.
  - É feita uma operação XOR entre o primeiro operando e R, e comparado este resultado com $ZERO$. Se o resultado deste XOR for diferente de $ZERO$, indica que o primeiro operando não é contido pelo segundo. Assim, o bit Relação 1 é definido em 1.
  - É feita uma operação XOR entre o segundo operando e R, e comparado este resultado com $ZERO$. Se o resultado deste XOR for diferente de $ZERO$, indica que o segundo operando não é contido pelo primeiro. Assim, o bit Relação 0 é definido em 1.
  - É chamada a função que conta os 1s de R, e o resultado parcial é acumulado em um dado de 24 bits.
- São colocados nas portas de saída, o resultado da relação obtido e o valor acumulado do número de interseções.

Valor do resultado da Relação	Relação entre as duas linhas
011	Os dois operandos não possuem interseção, e desta forma um operando não é contido pelo outro.
100	Os dois operandos possuem interseção, mas um é contido pelo outro e vice-versa. Desta forma, os dois operandos são iguais.
101	Os dois operandos possuem interseção, e o segundo contém o primeiro.
110	Os dois operandos possuem interseção, e o primeiro contém o segundo.
111	Os dois operandos possuem interseção, e nenhum é contido pelo outro. Esta resposta indica que as duas linhas de M estão contidas no mesmo componente.

Tabela 5.1: Resultados possíveis da relação entre duas linhas

Note que a cada comparação de blocos, na definição da relação das duas linhas, se qualquer uma das condições para definir os três bits Relação não for atendida, o respectivo bit Relação não será mudado e continuará com o valor obtido quando esta condição foi atendida. Assim, para a comparação das duas linhas seria necessário processar os blocos até que os três bits fossem definidos em 1. Porém, como é preciso contar o número de

interseções, é necessário comparar todos os blocos. Após cada bit Relação passar de 0 para 1, seu valor não mudará até o término das comparações de blocos.

O circuito gerado para a realização da comparação de linhas é simples e, ao observar as operações descritas acima, pode-se projetá-lo da forma mostrada na Figura 5.6.

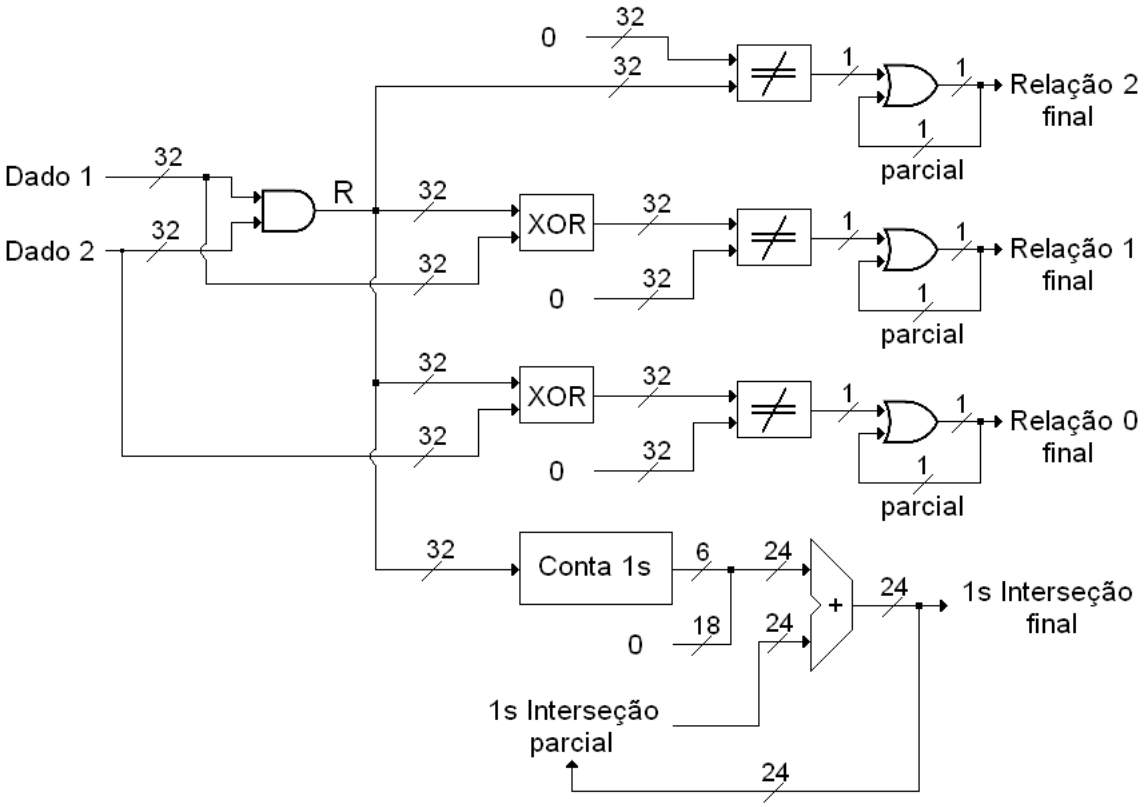


Figura 5.6: Circuito para comparação de linhas em blocos

A contagem de interseções é acumulada em um dado de 24 bits e sua saída é lida pelo controle ao final da comparação de todos os blocos das duas linhas. A função, que conta o número de interseções do par de linhas com operandos de 32 bits, usa somadores organizados em uma árvore binária. Primeiro, os bits de R são somados aos pares (R_0 com R_1, R_2 com R_3, ... , R_{30} com R_{31}) obtendo 16 resultados de dois bits cada. Em seguida, soma estes resultados aos pares e obtém oito resultados de três bits. Estes oito resultados são somados aos pares, obtendo quatro resultados de quatro bits. Novamente, é feita a soma destes quatro resultados em pares e são obtidos dois resultados de cinco bits. Por fim, os dois resultados de cinco bits são somados formando um resultado de seis bits. Este resultado é unido com 18 bits em 0, para formar a soma final em 24 bits. Do modo como foi implementada, esta soma gera uma estrutura de somadores organizados como uma árvore, com 16 somadores de um bit nas folhas e um somador de cinco bits na raiz.

Os somadores de um bit das folhas da árvore podem ser implementados de forma muito simples, como mostra a Figura 5.7, e a árvore de somadores é mostrada na Figura 5.8

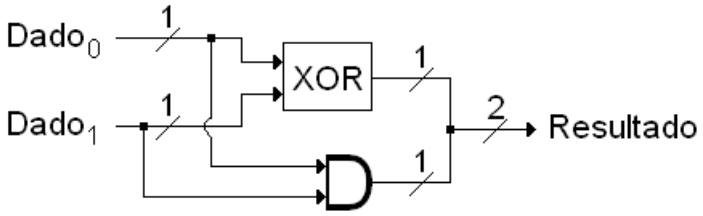


Figura 5.7: Implementação do somador de um bit

O algoritmo de Fulkerson e Gross, ao construir o grafo G_C, não realiza a comparação de todas as linhas entre si, pois ele pode encontrar duas linhas que ainda não foram comparadas mas que já pertencem ao mesmo componente. Isto acontece quando são feitas primeiro as comparações entre as linhas (vértices) que formam o caminho em G_C que une as duas linhas não comparadas. As variações da implementação híbrida da operação de comparação de clones são apresentadas em seguida.

5.2.1 Envio e Comparação de Clones por Demanda

Nesta implementação são realizadas apenas as comparações necessárias entre linhas, isto é, a parte em SW da solução envia o pedido de realização desta operação para a parte em HW a medida que determina quais pares de linhas precisam ser comparados. Ou seja, a comparação de linhas no HW é feita por demanda. Além disso, para cada operação solicitada, o par de linhas a ser comparado é enviado para a FPGA, intercalando-se os bits de cada linha. Os primeiros 32 bits são da primeira linha, os próximos 32 bits são da segunda linha, e assim sucessivamente.

O módulo da FPGA que recebe os pacotes, recebe quatro bits por vez da porta de rede e espera receber um bloco de 32 bits para passá-lo ao módulo de controle, que o coloca em uma das portas do comparador. O primeiro bloco é colocado na porta do primeiro operando do comparador, e o bloco seguinte na porta do segundo operando do comparador. No ciclo seguinte, o comparador compara os dois operandos, determina a relação entre eles e conta suas interseções. Este processo é repetido até que o módulo de recebimento termine de receber os bits da porta de rede.

Quando os blocos das duas linhas acabam, o controle lê os resultados das portas de saída do comparador de linhas e os encaminha para o módulo de envio de pacotes, que

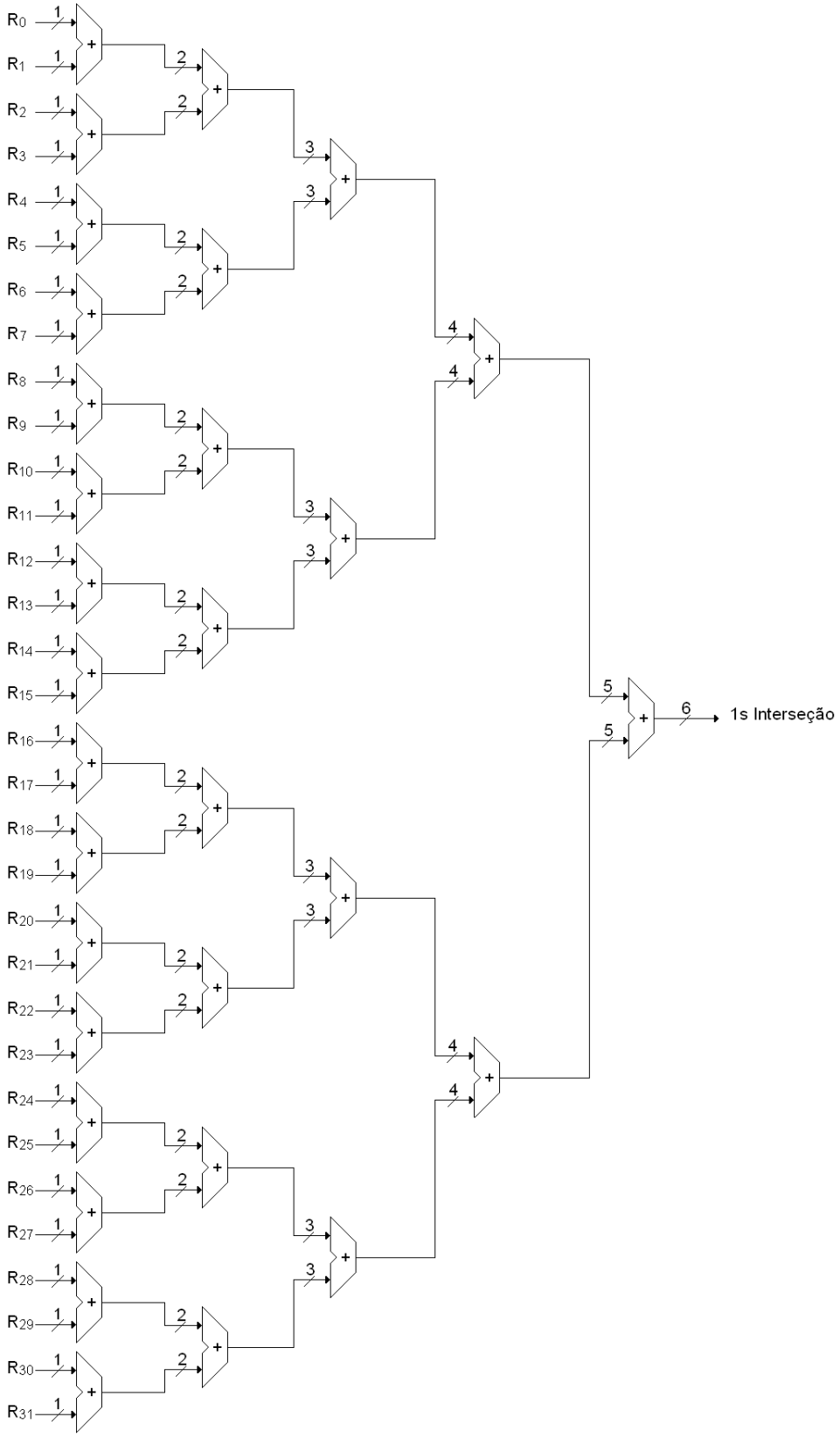


Figura 5.8: Circuito para contagem do número de 1s da interseção das linhas

monta o pacote e o envia pela interface de rede.

Desta forma, para cada comparação de linhas a ser realizada, é necessário o envio do par de linhas e o recebimento da resposta contendo a relação entre as linhas e o número de interseções entre elas.

5.2.2 Envio Total e Comparação de Clones por Demanda

O envio de pares de linhas para comparação pode ser requerido milhares de vezes pelo programa híbrido e uma mesma linha pode ser enviada várias vezes, quando comparada com diferentes linhas. Assim, foi implementada uma melhoria em relação à primeira implementação, utilizando a memória RAM disponível na placa que contém a FPGA.

A memória contida na placa está dividida em cinco bancos de 2 MB cada, numerados de 0 a 4. Nesta segunda implementação, o software envia toda a matriz M para a FPGA e ela é armazenada no banco 1 de memória. Os pedidos de comparação são feitos sob demanda, assim como na solução anterior, enviando-se o índice das duas linhas, que será usado para acessar a memória. Deste modo, a comunicação fica mais rápida pois o pacote contendo os dois índices é bem menor que o pacote que contém as duas linhas da primeira implementação.

A memória armazena 32 bits por endereço. Desta forma, são necessários dois acessos para ler os blocos que serão comparados. O tempo de acesso para ler os dois blocos é bem menor que o tempo despendido na espera dos dois blocos oriundos do módulo de recebimento (que acontecia na primeira implementação). Isto acontece porque a memória está trabalhando na mesma freqüência de *clock* que o comparador e o controle, e o módulo de recebimento trabalha à metade desta freqüência.

O tempo necessário para enviar e armazenar M é compensado pelo envio de cada linha uma única vez. Uma linha pode ser comparada até $n-1$ vezes. Na implementação anterior, esta linha poderia ser comunicada integralmente este número de vezes. Nesta nova implementação, a linha é comunicada apenas uma vez, e seu índice de 32 bits é que pode ser comunicado $n-1$ vezes, gerando um menor *overhead* de comunicação que aquele da implementação anterior.

5.2.3 Envio e Comparação Totais de Clones

Dependendo da matriz de entrada, podem ser feitas muitas comparações de linhas, e muitos pacotes podem ser trocados para que essas comparações sejam realizadas. A proposta desta terceira implementação é receber M, como na implementação anterior, e realizar a comparação de todas as linhas de M entre si, sem a necessidade do programa em SW requisitar a comparação dos pares de linhas necessários. O número de operações realizadas nesta implementação é dado pelo somatório $\sum_{i=1}^{n-1} i = (n-1) \times (\frac{n}{2})$.

Esta mudança pode produzir dois efeitos contrários dependendo das características da matriz de entrada. Se são necessárias muitas comparações de linhas, havendo muitos pedidos de comparações, a realização da comparação total diminuirá o *overhead* de comunicação. Isto ocorre pois a comunicação, neste caso, é composta apenas pelo número total de pacotes retornados pela FPGA para o SW. Com a comparação por demanda, a comunicação é composta pelos pacotes de pedidos do SW mais os pacotes retornados pela FPGA para o SW. Assim, o número de pacotes comunicados nesta implementação é menor ou equivalente ao número de pacotes da implementação que compara por demanda. Neste contexto, esta mudança proporciona ganho de desempenho. Caso sejam necessárias poucas comparações de linhas, a comparação total realiza muitas operações desnecessárias, levando a um tempo de processamento maior que o apresentado na implementação anterior.

Como podem ser realizadas milhares, e até milhões de comparações, o resultado destas comparações é armazenado no banco de memória 0. O controle, após receber a matriz M no banco 1, inicia a comparação de linhas mantendo contadores de endereço que são incrementados a medida que as posições do banco de memória 1 são lidas. Após ler todos os blocos de 32 bits das duas linhas e realizar as comparações, o resultado é armazenado no banco de memória 0. Cada resultado é armazenado em uma posição do banco 0, onde 8 bits são o resultado da relação entre as duas linhas de M e os outros 24 bits são o número de interseções entre elas. Quando o banco de memória 0 é totalmente preenchido, é feita uma rodada de comunicação enviando todas as posições do banco 0 para o SW. Se ainda houver comparações a serem feitas, elas serão feitas até que o banco de memória 0 seja totalmente preenchido ou que as comparações necessárias terminem. No momento da rodada de comunicação, não são feitas comparações de linhas pois os resultados de tais comparações podem sobrescrever resultados ainda não enviados.

5.2.4 Envio e Comparação Totais de Clones com Paralelismo

Para melhorar o desempenho do processamento no hardware reconfigurável, foi acrescentado um novo comparador à implementação anterior. Este comparador também contém duas portas de entrada de 32 bits cada para receber os blocos lidos do banco de memória. Com isto, foi utilizado mais um banco de memória, o banco 2, de forma que o primeiro comparador continua recebendo blocos do banco 1, enquanto que o segundo comparador recebe blocos do banco 2.

A comparação de linhas de M é feita em blocos, e cada bloco de cada linha pode ser comparado em qualquer ordem, desde que seja comparado com o bloco correspondente da outra linha. Assim, ao receber M, os blocos passados do módulo de recebimento para o controle são armazenados intercaladamente nos bancos de memória 1 e 2, fazendo com que cada linha tenha metade de seus bits em cada banco. A Figura 5.9 ilustra um exemplo do armazenamento de duas linhas com quatro blocos em dois bancos de memória da placa da FPGA.

No momento do processamento, o endereço de memória a ser acessado em cada banco é o mesmo, pois agora são lidos dois blocos de uma mesma linha que ocupam a mesma posição nos dois bancos. São feitas duas leituras em cada banco, a primeira leitura acessa os dois blocos da primeira linha, e a segunda leitura acessa os dois blocos da segunda linha. A Figura 5.10 mostra como os dados são fornecidos para os comparadores para que sejam feitas as comparações dos dois primeiros blocos de cada linha do exemplo da Figura 5.9.

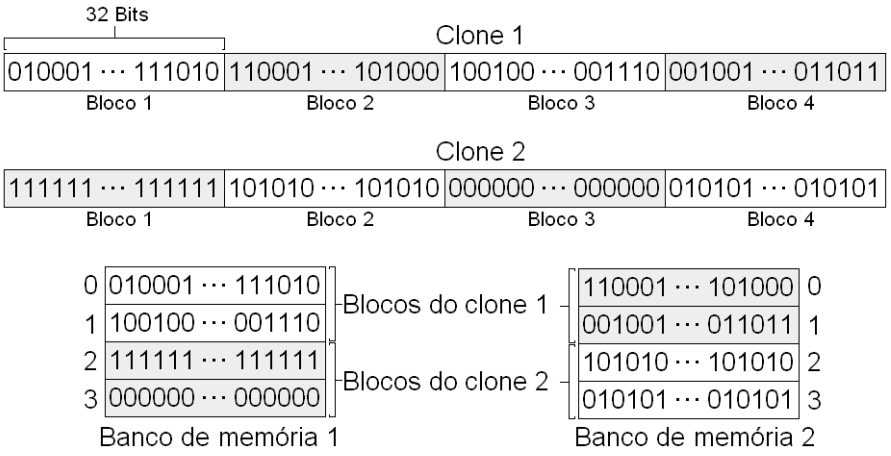


Figura 5.9: Esquema de armazenamento dos clones em dois bancos de memória

Ao final do processamento de todos os blocos, o número de interseções resultante em

cada comparador é somado pelo controle e é feita uma operação OR com os dois resultados da relação entre as linhas. Estes resultados são armazenados no banco de memória 0, utilizado pelo módulo de envio de dados, da mesma forma que a implementação anterior.

Os dois comparadores tem o mesmo tempo de acesso aos dados que a implementação anterior, com o dobro de capacidade de armazenamento. Ambos os comparadores, utilizando dois bancos de memória, proporcionam um desempenho equivalente ao uso de um único comparador de 64 bits com uma memória que armazena 64 bits por posição.

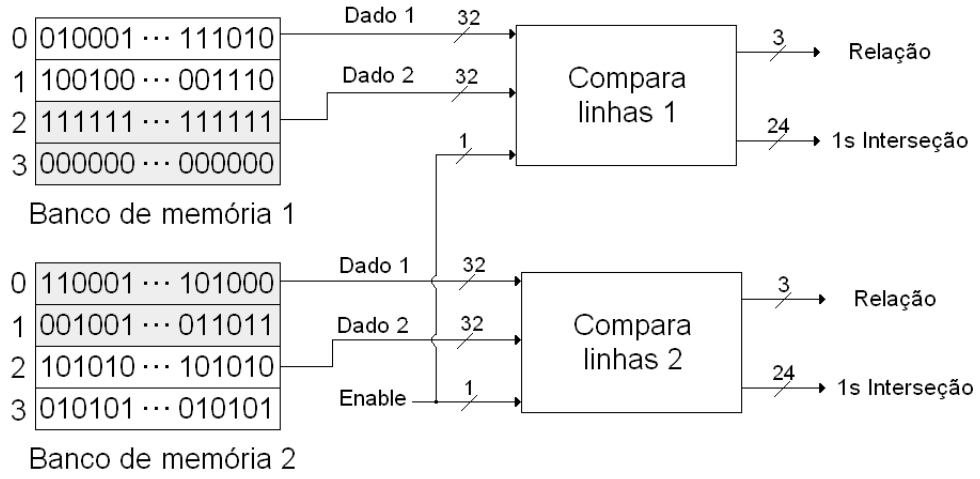


Figura 5.10: Comparação das primeiras duas partes dos clones da Figura 5.9

5.3 Construção de Conjuntos de Colunas

A construção dos conjuntos de colunas de cada componente é realizada após a permutação das suas colunas. Estes conjuntos são utilizados para orientar a união dos componentes no passo final do algoritmo e são escritos na saída do programa, indicando quais colunas da matriz original são representadas por cada coluna da matriz final. Desta forma, os conjuntos de colunas codificam a ordem em que as colunas podem ser organizadas de modo a gerar a matriz de entrada com todos os 1s consecutivos em cada linha. Por exemplo, a Figura 5.11 mostra um componente (já permutado) com 4 linhas e os seus conjuntos de colunas: $\{2\}$, $\{4\}$, $\{1\}$, $\{6\}$, $\{5\}$, $\{3, 7\}$.

	{2}	{4}	{1}	{6}	{5}	{3, 7}	
l_9	0	0	1	1	1	0	0
l_2	0	0	0	1	1	1	1
l_0	0	1	1	1	1	0	0
l_7	1	1	1	1	0	0	0

Figura 5.11: Componente com 1s consecutivos

Para realizar a criação do conjunto de uma coluna, de um componente, é necessário percorrê-la, ou seja, os componentes são processados por colunas. Neste percurso, se a posição encontrada desta coluna for 1, indica que algum elemento do conjunto de 1s da respectiva linha[1] pertence ao conjunto de 1s desta coluna. Se este é o primeiro 1 encontrado ao percorrer a coluna, o conjunto de 1s da respectiva linha é copiado para o conjunto de 1s desta coluna. Senão, apenas as colunas pertencentes a ambos os conjuntos (da linha e da coluna) farão parte do conjunto da coluna atual. Caso a posição na coluna do componente seja 0, as colunas pertencentes ao conjunto da respectiva linha não pertencem ao conjunto desta coluna. Portanto, deve ser retirado do conjunto da coluna qualquer elemento que esteja no conjunto desta linha que possui 0 nesta coluna do componente. Definido este raciocínio, são realizadas estas comparações e manipulações de conjuntos percorrendo todas as colunas do componente. A Figura 5.12 ilustra este raciocínio para a quinta coluna do componente da Figura 5.11.

A operação de construção dos conjuntos de colunas realizada em hardware utiliza um ou dois bancos de memória contendo a matriz M, enviada pelo software para a FPGA. São utilizados mais dois bancos de memória: o banco 4 contém o componente permutado armazenado por colunas, e o banco 3 contém os índices das linhas de M, no banco 1, que pertencem ao componente. O construtor de conjuntos utiliza a mesma estratégia do comparador de linhas em dividir os dados a serem processados e o resultado em blocos. Neste caso, todos os dados utilizados são divididos em blocos 32 bits. Por poder conter todas as colunas de M, um conjunto de uma coluna de um componente possui o número de bits igual ao número de colunas de M.

[1] Representa todas as colunas onde a linha possui 1 na matriz M

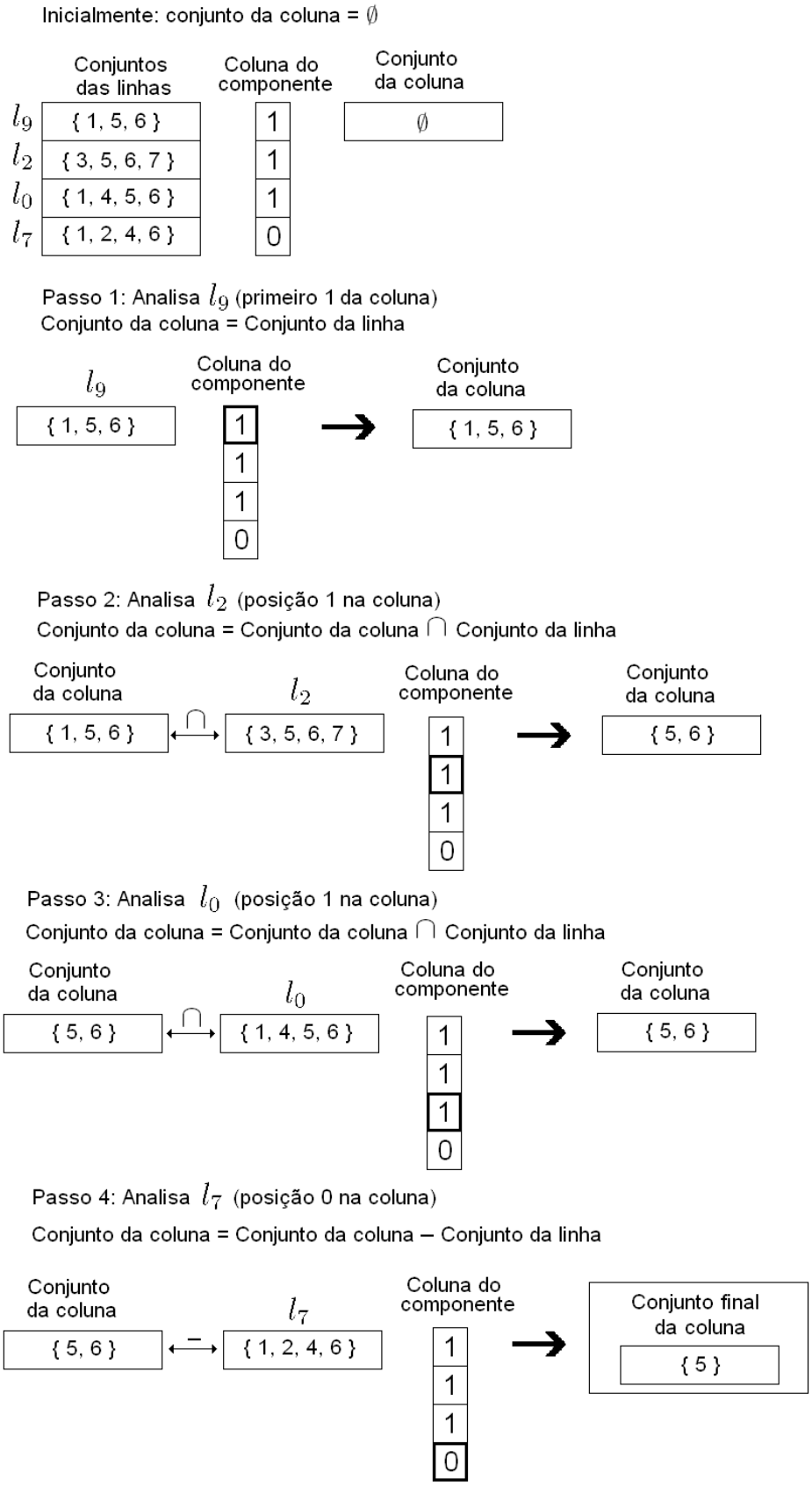


Figura 5.12: Construção do conjunto de colunas para a quinta coluna do componente da Figura 5.11

As entradas do construtor de conjuntos são o bloco da coluna do componente, o bloco do conjunto de colunas da linha e os sinais de controle. A saída é o bloco do conjunto da coluna do componente. O construtor mantém dois dados, um com o resultado parcial das computações (P) de 32 bits, e outro que representa o índice da posição da coluna a ser acessada. O construtor recebe um bloco da coluna a cada 32 ciclos do *clock* e um bloco de uma linha a cada ciclo.

Após receber um bloco de uma linha, o construtor verifica a posição atual da coluna. Se a posição da coluna for 0 (indicando que as colunas do conjunto da linha não pertencem ao conjunto da coluna), o bloco do conjunto da linha é negado através da operação NOT e é feita uma operação AND deste resultado com P. Se a posição da coluna for 1, será feita diretamente a operação AND entre o bloco do conjunto da linha e P. O resultado é armazenado em P, e o índice da coluna é decrementado. O circuito da Figura 5.13 mostra como pode ser implementada esta operação em hardware.

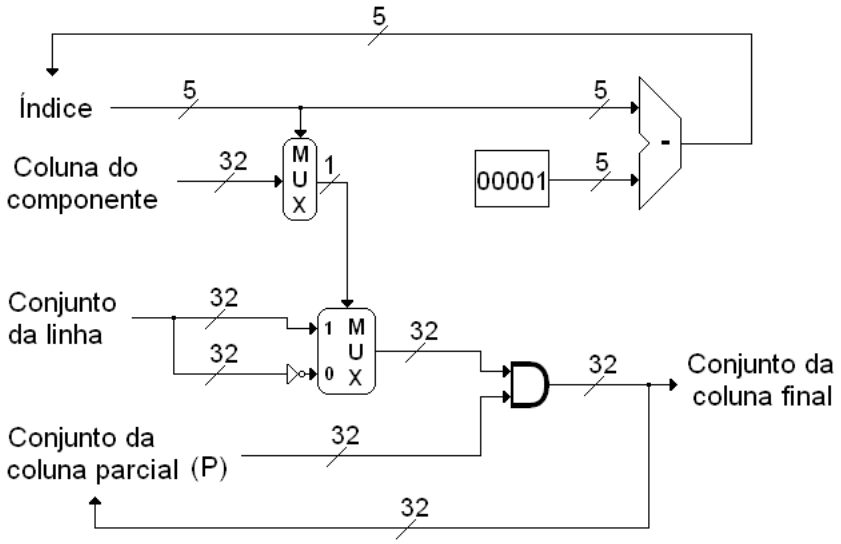


Figura 5.13: Circuito para construção de conjuntos de colunas

O resultado parcial P é inicializado com 1 em todos os seus bits, indicando que todas as colunas do bloco da linha processada pertencem ao conjunto da coluna. Quando a posição da coluna é 0, o bloco do conjunto da linha é negado para que onde haja 1 seja convertido em 0 e estas posições são passadas para 0 em P. Assim, as colunas do conjunto desta linha nunca aparecerão no conjunto final desta coluna do componente, o que é correto, pois, havendo 0 nesta coluna do componente, indica que nenhuma coluna do conjunto da linha pertence ao conjunto desta coluna. Quando a posição da coluna é 1, a operação AND faz com que permaneçam em P apenas as colunas comuns entre o resultado parcial

e o bloco do conjunto da linha de entrada. Se P continuar todo 1 quando for realizada a operação AND, os 1s do bloco da linha serão os 1s de P, e os 0s do conjunto da linha serão passados para P, fazendo com que o conjunto de P seja igual ao conjunto da linha.

Este processamento requer que haja pelo menos uma posição da coluna do componente que seja 1, pois senão, serão retiradas as colunas de P que ocorrem em cada linha, mas as colunas que não ocorrem em nenhuma linha vão permanecer, e P conterá 1s que não existem, pois o conjunto correto neste caso seria todo 0. Deste modo, o software envia as colunas do componente permutado enquanto elas não tiverem apenas 0s[2]. Os conjuntos das colunas com apenas 0s são definidos como vazios.

Foram desenvolvidas duas implementações para a construção de conjuntos. A primeira utiliza apenas um construtor, e a segunda utiliza dois construtores trabalhando em paralelo.

5.3.1 Construção Simples de Conjuntos

Nesta implementação, é utilizado um construtor de conjuntos, que utiliza três bancos de memória para armazenar os dados para processamento, e um banco de memória para armazenar o resultado que será enviado pela rede.

O controle inicia a leitura dos índices das linhas, pertencentes ao componente, que estão armazenados no banco de memória 3. Quando o primeiro índice é recuperado, ele é usado para acessar o banco de memória 1, que armazena M, e o primeiro bloco da linha correspondente é lido. Também é lido o primeiro bloco da coluna do componente, armazenado no banco de memória 4. Assim, o bloco da linha e o bloco da coluna chegam no mesmo ciclo para o comparador, que pode inciar o seu processamento. A leitura dos índices é feita de maneira circular, ou seja, ao alcançar o índice da última linha do componente, o próximo índice a ser lido será o da primeira linha. A cada 32 blocos de linhas lidos, um novo bloco da coluna é lido, pois o processamento é feito olhando uma posição da coluna por ciclo e utilizando todo o bloco de 32 bits de uma linha em cada ciclo.

Ao terminar de ler todos os blocos da coluna, o bloco seguinte de cada linha deverá ser lido, sendo feita a releitura dos blocos da coluna. Quando todos os blocos das linhas forem processados, passa-se para a próxima coluna do componente, para gerar seu conjunto de colunas. Quando o processamento de todos os blocos da coluna com os blocos de todas as

[2]Pode haver colunas que possuem apenas 0 em um componente, pois os 1s destas colunas na matriz de entrada pertencem às linhas que são de outros componentes.

linhas termina, P contém o bloco do conjunto final desta coluna. Assim, P é colocado na saída do construtor e este valor é armazenado no banco de memória 0, que será usado pelo módulo que envia os pacotes pela rede. A escrita dos resultados neste banco de memória é seqüencial, uma vez que cada bloco do conjunto de uma coluna é gerado de maneira seqüencial, formando no final o conjunto completo gravado no banco.

A Figura 5.14 ilustra o processo descrito. Neste exemplo, M possui 300 linhas e 96 colunas e, portanto, suas linhas são compostas de três blocos no banco 1. O componente possui 192 linhas, onde sua primeira linha é a 2ª linha de M e a última linha é a 250ª linha de M (índices na memória começam em 0). Após ler o primeiro endereço do banco 3, sabe-se que a primeira linha do componente começa na posição 3 do banco 1. Esta posição é lida juntamente com a primeira posição do banco 4, que contém o primeiro bloco de 32 bits da primeira coluna. Após ler 32 linhas do banco 1, todas as posições do bloco da coluna lida foram avaliadas, neste momento é feita uma nova leitura do segundo bloco da coluna (endereço 1 do banco 4), juntamente com o endereço inicial da 33ª linha, conforme mostra a Figura 5.14.

Como dito anteriormente, para a formação do conjunto de uma coluna é preciso utilizar a informação contida nos conjuntos de colunas de todas as linhas do componente, descendo pelas linhas de cada coluna. Porém, os conjuntos de cada linha (a linha contida no banco 1 pode ser considerada o seu conjunto de colunas) são divididos em blocos e, deste modo, o conjunto da coluna a ser gerado também é criado por blocos. No exemplo, para cada coluna teremos três rodadas (linhas divididas em três blocos). Na primeira rodada, é lido apenas a posição inicial de cada linha, e a coluna do banco 4 é lida completamente (todos os seus seis blocos).

Após esta rodada, o primeiro bloco do conjunto da coluna estará pronto e será colocado no banco 0, e o construtor de conjuntos é reiniciado. Como há mais dois blocos de cada linha do banco 1 a serem processados, o endereço a ser lido no banco 4 volta pra o início da sua coluna. Também neste momento, o banco 3 de memória já leu a última posição com o índice da última linha do componente e lerá o índice da primeira linha.

Para acessar a segunda posição de cada linha, é usado um contador que é somado com o índice fornecido pelo banco 3, gerando assim o endereço. Este contador possuía o valor 0 na primeira rodada e agora possui o valor 1. Na segunda rodada são lidos novamente os blocos da coluna do banco 4 e o segundo bloco das linhas do banco 1 indicados pelo banco 3 e o contador. No final da rodada, são realizados os mesmos passos que a rodada anterior, e é feita a terceira rodada.

Após a escrita do terceiro e último bloco do conjunto e reinício do contador de blocos,

passa-se para a próxima coluna do componente, onde serão criados os três blocos de seu respectivo conjunto. Note que, para esta segunda coluna, o endereço inicial no banco 4 é 6, e ao fim de suas duas primeiras rodadas volta-se para este endereço.

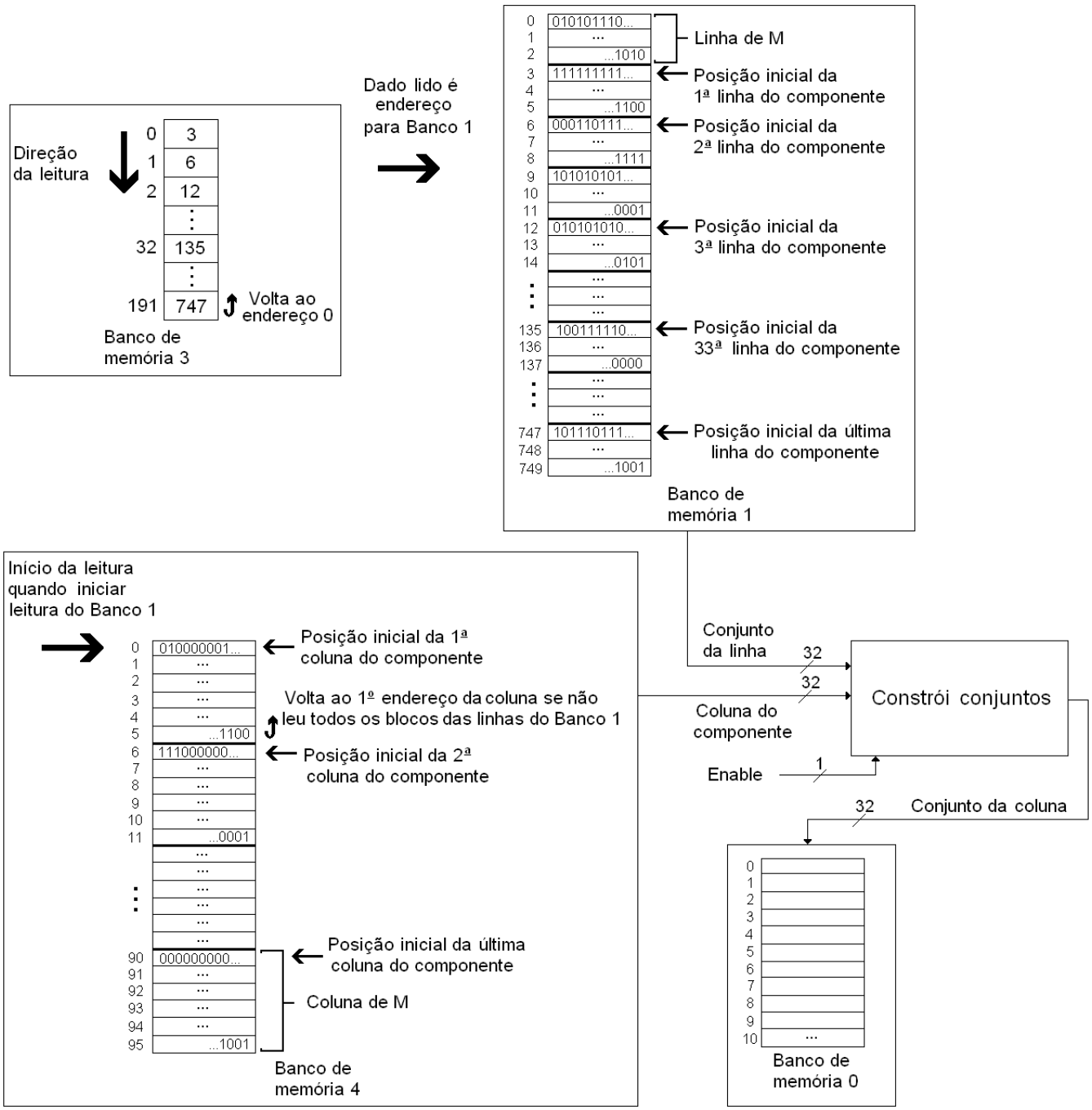


Figura 5.14: Seqüência de passos para construção dos conjuntos de colunas

5.3.2 Construção de Conjuntos em Paralelo

Esta segunda implementação utiliza dois construtores de conjuntos, que processam dois blocos consecutivos do conjunto de uma coluna. Para isto, aproveitando o que foi feito na comparação de linhas com dois comparadores, utiliza-se os bancos de memória 1 e 2 para armazenar as linhas de M de forma alternada em cada banco. Assim, todos os cinco bancos de memória da placa que contém a FPGA são usados: os bancos 1 e 2 armazenam as linhas de M, o banco 4 armazena o componente permutado, o banco 3 armazena os índices de linhas de M que compõe o componente, e o banco 0 armazena os resultados e é utilizado pelo módulo de envio de pacotes.

Os dois construtores recebem o mesmo bloco da coluna lida pelo controle, e os dois blocos lidos da linha estão na mesma posição de memória dos bancos que contém M. Com isto, o controle fica simplificado, utilizando um único endereço para acessar os bancos 1 e 2, e utilizando o mesmo bloco da coluna do componente. O resto do controle funciona da mesma forma que a implementação da construção de conjuntos anterior.

Assim, são produzidos dois blocos do conjunto da coluna atual ao final da leitura dos blocos de cada linha e da coluna inteira do componente. Porém, quando o componente possui apenas uma linha, a geração dos dois blocos acontece a cada ciclo, mas a capacidade de armazenamento dos resultados é de apenas um bloco por ciclo. Neste caso, é necessário uma pausa de um ciclo para que não se perca nenhum resultado. Com componentes com duas ou mais linhas o processamento ocorre sem pausas.

5.4 Controle

O módulo de controle é aquele que possui a maior complexidade dentro da implementação híbrida, uma vez que gerencia o fluxo de dados necessário para as implementações feitas na FPGA. Com as várias versões de implementações das duas operações, o controle também foi modificado para atender às necessidades de cada implementação. Foram feitas cinco versões da solução híbrida, aumentando a cada versão as funcionalidades e a complexidade do controle.

5.4.1 Envio e Comparação de Clones por Demanda

Esta versão conta apenas com a primeira implementação da operação de comparação de linhas, onde não há uso de bancos de memória e os dados são processados em um fluxo

enquanto são recebidos através da porta de rede.

O controle consiste de uma máquina de estados. Inicialmente, a máquina espera o módulo de recebimento indicar o recebimento do par de blocos a serem comparados. Em seguida, passa-se para um estado de espera da operação de comparação e contagem de ciclos gastos. O próximo estado é o que avalia o resultado. Após o processamento dos dois blocos, se houver mais dois blocos para serem processados, o autômato volta para o estado de espera de blocos. Caso contrário, este estado de avaliação de resultados coloca na porta do módulo de envio de pacotes os resultados calculados e o autômato passa para o estado de envio.

No estado de envio, é iniciado o envio do pacote de resposta e o autômato passa para o estado de reinício, para que esteja pronto para um novo processamento. Voltando, por fim, ao estado de espera de blocos. A Figura 5.15 mostra os estados e transições de estados descritos.

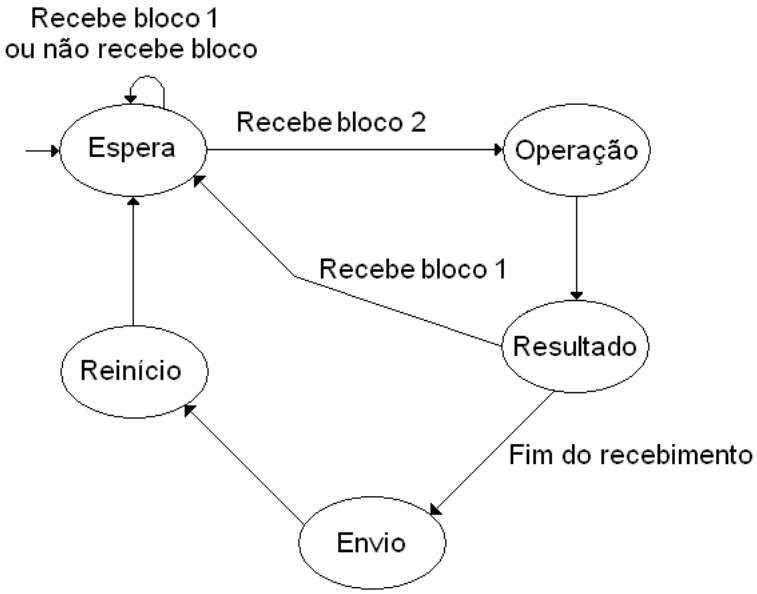


Figura 5.15: Máquina de estados do controle da primeira implementação

5.4.2 Envio Total e Comparação de Clones por Demanda

Esta segunda implementação utiliza os bancos de memória 0 e 1 para executar apenas a operação de comparação de linhas. Assim, o autômato é modificado para possibilitar o tratamento dos pacotes que contém M, cujos dados precisam ser armazenados no banco 1. São incluídos códigos de operação para diferenciar os pacotes que contém dados que serão escritos na memória, dos pacotes que contêm os índices das linhas a serem comparadas.

Há também um pacote que contém o número de blocos de cada linha (possuindo seu código exclusivo).

O estado inicial espera o código de operação (primeiro byte do pacote) ser recebido, e então passa para o estado que trata os dados do pacote. Quando os 32 bits vindos do módulo de recebimento chegam, o estado que trata os dados fará o armazenamento destes dados no banco 1, se o código de operação recebido anteriormente for o de escrita na memória. Enquanto houver dados no pacote, o autômato se mantém neste estado, voltando para o estado inicial quando terminar. Caso o código de operação indique que o pacote contém o tamanho de cada linha em blocos de 32 bits, este valor é armazenado e o autômato permanece no estado inicial. Caso o código de operação indique a comparação de linhas com os índices presentes no pacote, passa-se para o estado de espera da leitura do bloco da linha indicada pelo índice recebido. Este estado armazena os índices e inicia o acesso à primeira posição de memória que contém o primeiro bloco da primeira linha a ser comparada.

Além do autômato, existe um pequeno controlador da memória, que tomará conta dos endereços a serem acessados (alternando o endereço do bloco da primeira linha e o endereço do bloco da segunda linha). Este controlador é iniciado no ciclo seguinte ao pedido de leitura do primeiro bloco e se encarregará de pedir os próximos blocos. Assim, o autômato só controla o pedido dos primeiros 32 bits da primeira linha a ser comparada e o controlador adicional manda fazer uma leitura de cada bloco, de cada linha, por ciclo.

No estado de espera da memória do autômato são contados ciclos de *overhead* enquanto os blocos das duas linhas não estão prontos. Um outro pequeno controlador coloca cada bloco que chega da memória na sua respectiva porta do comparador de linhas e ativa a comparação destes blocos quando o bloco da segunda linha chega, fazendo com que a cada três ciclos haja uma comparação. Neste momento, o autômato passa do estado de espera da memória para o estado de operação, que conta o ciclo de operação e passa para o estado de resultado. No estado de resultado, se não foram comparados todos os blocos, o autômato volta para o estado de operação.

Note que, quando o autômato está no estado de operação e cada linha possui dois ou mais blocos, o bloco seguinte da primeira linha chega da memória, e quando passa-se para o estado de resultado, o bloco seguinte da segunda linha chega da memória e o processamento será feito no próximo ciclo. Por isto, o autômato passa do estado de resultado para o de operação quando há mais blocos a serem processados, sem precisar esperar.

Quando todos os blocos são comparados, o estado de resultado manda gravar os

resultados (relação entre linhas e número de interseções) no banco de memória 0 (quem controla o banco 0 é o módulo de envio e, portanto, o estado de resultado pede para o módulo de envio gravar os resultados no banco 0) e passa para o estado de envio. A ativação do módulo de envio é feita no estado de envio, que passa em seguida para o estado de reinício. Por fim, o autômato volta ao estado inicial de espera de código de operação do módulo de recebimento. A Figura 5.16 mostra os estados e transições de estados descritos.

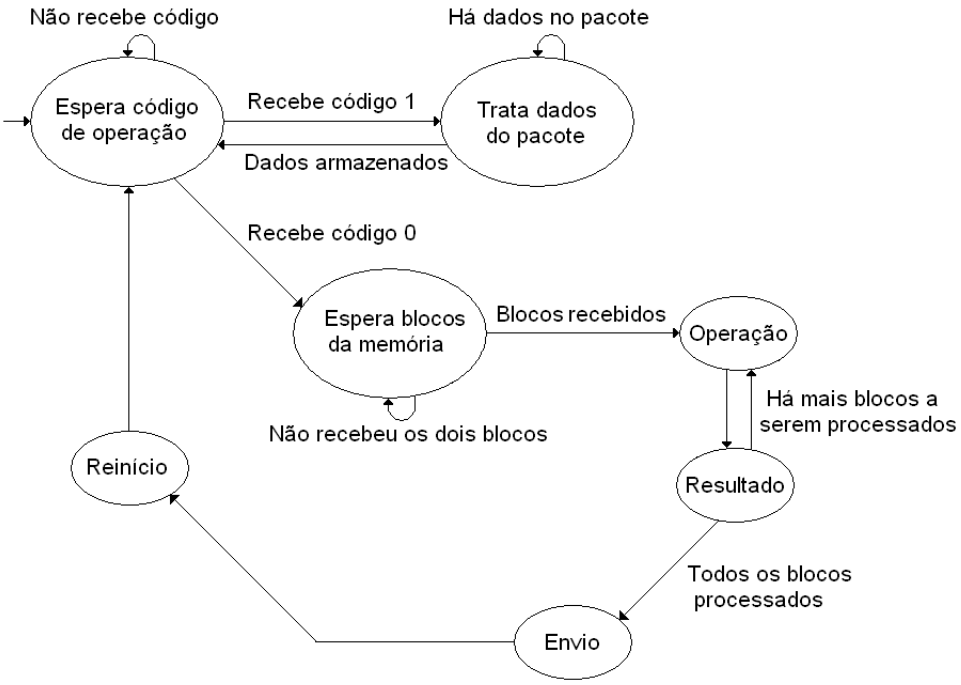


Figura 5.16: Máquina de estados do controle da segunda implementação

5.4.3 Envio Total, Comparação de Clones por Demanda e Construção Simples de Conjuntos

Esta implementação inclui, além da comparação de linhas, a operação de construção de conjuntos de colunas do componente. Para isto, é necessário o uso dos bancos de memória 3 e 4. O estado que trata os dados do pacote foi desmembrado em três estados que gerenciam o armazenamento dos dados nos bancos 1, 3, e 4 (cada estado comanda um banco de memória). Há também a inclusão de dois estados que controlam a construção de conjuntos.

Para a comparação de linhas, há dois pequenos controladores adicionais que realizam a leitura e iniciam cada comparação de blocos das linhas. Assim como na implementação

anterior, é feito um pedido à memória de um bloco de cada linha a cada ciclo, alternando o pedido do bloco da primeira linha com o da segunda linha. A memória entrega alternadamente um bloco de cada linha por ciclo, com a realização de uma comparação a cada três ciclos, e isto gera um fluxo de dados semelhante a um pipeline. Os estados de espera dados da memória, operação, resultado, e envio também funcionam da mesma maneira que na implementação anterior.

A construção de conjuntos possui um estado que é ativado quando um pacote contendo o código de operação correspondente é recebido, iniciando a leitura do banco 3 (contém os índices iniciais das linhas que pertencem ao componente). Em seguida, o autômato passa para o estado de operação da construção de conjuntos. Neste estado são contados os ciclos de *overhead* e de operação do construtor. Quando um bloco do conjunto da coluna for produzido (processando os blocos correspondentes de todas as linhas e todos os blocos da coluna) é feito o armazenamento deste bloco do conjunto da coluna no banco de memória 0 (pedindo ao módulo de envio para gravar o bloco no banco). A Figura 5.17 mostra os estados e transições de estados descritos.

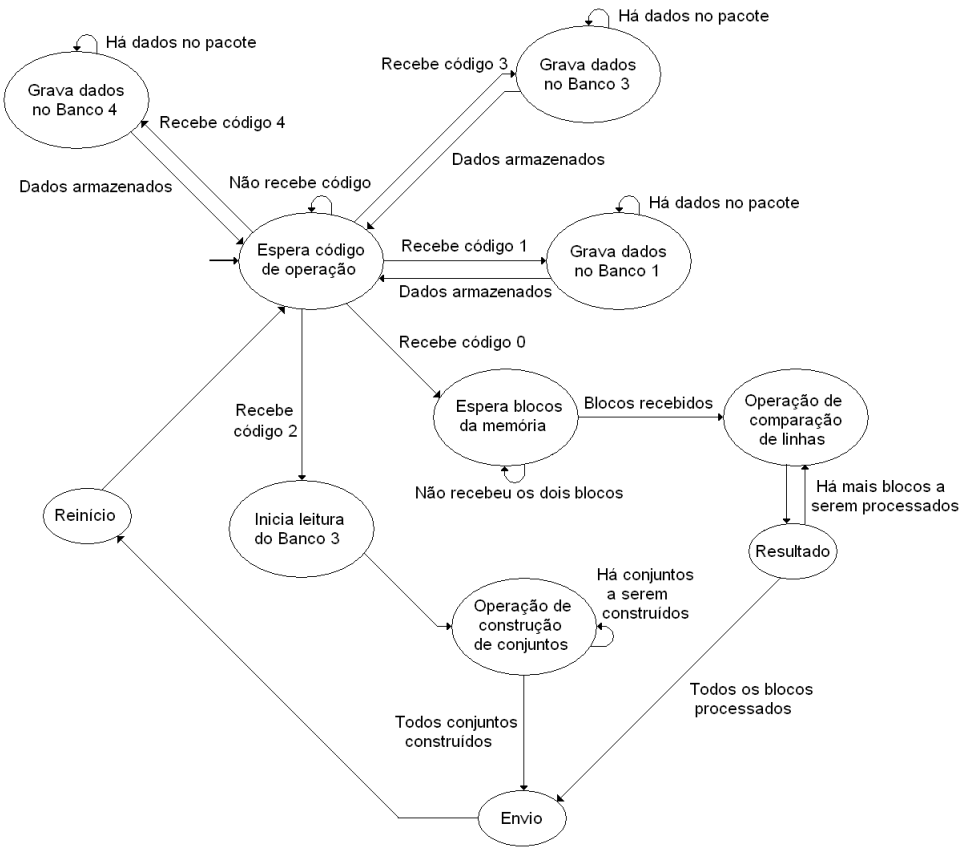


Figura 5.17: Máquina de estados do controle da terceira implementação

Para a construção de conjuntos são utilizados três pequenos controladores que funcionam em cooperação com o autômato, e proporcionam um fluxo de dados contínuo entre os bancos de memória e o construtor de conjuntos. Desta forma, a cada ciclo o construtor de conjuntos realiza uma operação, e seu processamento só é cessado quando todos os conjuntos forem criados. Estes controladores adicionais são responsáveis pelas operações descritas na Seção 5.3.1.

O primeiro controlador adicional, utilizado na construção de conjuntos, faz a leitura dos índices que estão contidos no banco 3 e serão usados para acessar o banco 1, que contém as linhas de M. Esta leitura é feita de maneira seqüencial e circular, como ilustrado no exemplo da Figura 5.14. Quando o primeiro índice lido do banco 3 é entregue, o segundo controlador inicia a leitura do dado contido no banco 1 utilizando como endereço este índice mais o valor do contador (conforme descreve a Seção 5.3.1). Também neste momento, o terceiro controlador inicia a leitura do bloco da primeira coluna do banco 4. Assim, no mesmo ciclo, o construtor de conjuntos terá o bloco da coluna e o bloco do conjunto da primeira linha pertencente ao componente, e poderá iniciar o seu processamento.

O terceiro controlador (do banco 4) lê um bloco da coluna e fica em espera. Quando o segundo controlador (do banco 1) lê o bloco da 32ª linha do componente, este controlador ativa o terceiro controlador que, no próximo ciclo, fará a leitura do próximo bloco da coluna junto com a leitura do bloco da 33ª linha do componente, feita pelo segundo controlador.

Quando todos os blocos da coluna são lidos e não foram processados todos os blocos das linhas, o terceiro controlador reiniciará seu contador de endereço para acessar novamente o primeiro bloco da coluna. Caso contrário, este controlador passa para a próxima coluna, e o segundo controlador reinicia o seu contador de blocos, para acessar novamente o primeiro bloco de cada linha para a nova coluna, como discutido anteriormente.

Este processo é realizado de forma síncrona e sem interrupções, com um pedido de leitura por ciclo. Desta forma, o construtor de conjuntos, após uma espera pelos primeiros dados, faz comparações a cada ciclo. Assim como na comparação de linhas, também é criado um pipeline na construção de conjuntos com a estratégia de controle utilizada.

5.4.4 Comparação Total e Construção Simples de Conjuntos

As comparações realizadas são de todas as linhas entre si, sem a necessidade do software enviar os índices de cada par de linhas a serem comparadas. A construção

de conjuntos não é alterada. Portanto, seus estados do autômato e seus controladores menores não são modificados em relação à implementação anterior. Assim, as modificações tratadas aqui dizem respeito apenas aos estados e controladores auxiliares que coordenam a comparação de linhas.

No autômato, os três estados que armazenam os dados de M, o componente, e os índices das linhas que pertencem ao componente nos bancos 1, 4, e 3 respectivamente, continuam os mesmos da implementação anterior. O estado que espera os dados da memória para o início da operação de comparação de linhas foi modificado para, além de iniciar a leitura do banco 1, armazenar o endereço do primeiro bloco da última linha de M (calculado pelo software e enviado no pacote que indica o início da comparação), necessário para que os controladores auxiliares saibam o momento do fim da leitura do banco 1.

Os estados de espera da memória e de operação da comparação de linhas permanecem inalterados. O estado de resultado é modificado para que mande armazenar o resultado no banco 0 (pedindo ao módulo de envio) e volte para o estado de espera para a comparação do próximo par de linhas. Caso o banco 0 fique cheio, ou todas as comparações sejam feitas, o autômato passa então para o estado de envio, comunicando para o módulo de envio o número de posições do banco 0 a enviar. O estado de envio também foi modificado para verificar se todas as comparações foram feitas. Neste caso, após enviar os resultados, o autômato passa para o estado de reinício. Senão, o autômato passa para um novo estado, que espera o envio dos resultados do banco 0. Neste novo estado, após o módulo de envio indicar ao controle que enviou todos os dados pela rede, são feitas algumas reinicializações (não todas) e é ativada a leitura da memória do ponto onde parou. O autômato passa então para o estado de espera da memória recomeçando o processo descrito neste parágrafo. A Figura 5.18 mostra os estados e transições de estados descritos.

Os dois controladores adicionais ao autômato, para a comparação de linhas, também são modificados. O primeiro controlador (que realiza e controla os pedidos de leitura ao banco 1), além de sua função que faz o pedido do bloco de cada linha intercaladamente, também controla o momento de interromper a leitura devido ao preenchimento completo do banco 0, ou quando são feitas todas as comparações das linhas de M entre si.

O primeiro controlador coloca uma pausa de 1 ciclo na leitura para indicar, ao segundo controlador, que foram lidos todos os blocos das duas linhas que estão sendo comparadas. O segundo controlador (que recebe os dados lidos da memória, os coloca na porta do comparador de linhas e ativa o comparador), ao ver uma pausa nos dados lidos da memória, avisa o autômato (que está no estado de resultado) que uma escrita no banco 0

deve ser feita pois os resultados da comparação das duas linhas estão prontos.

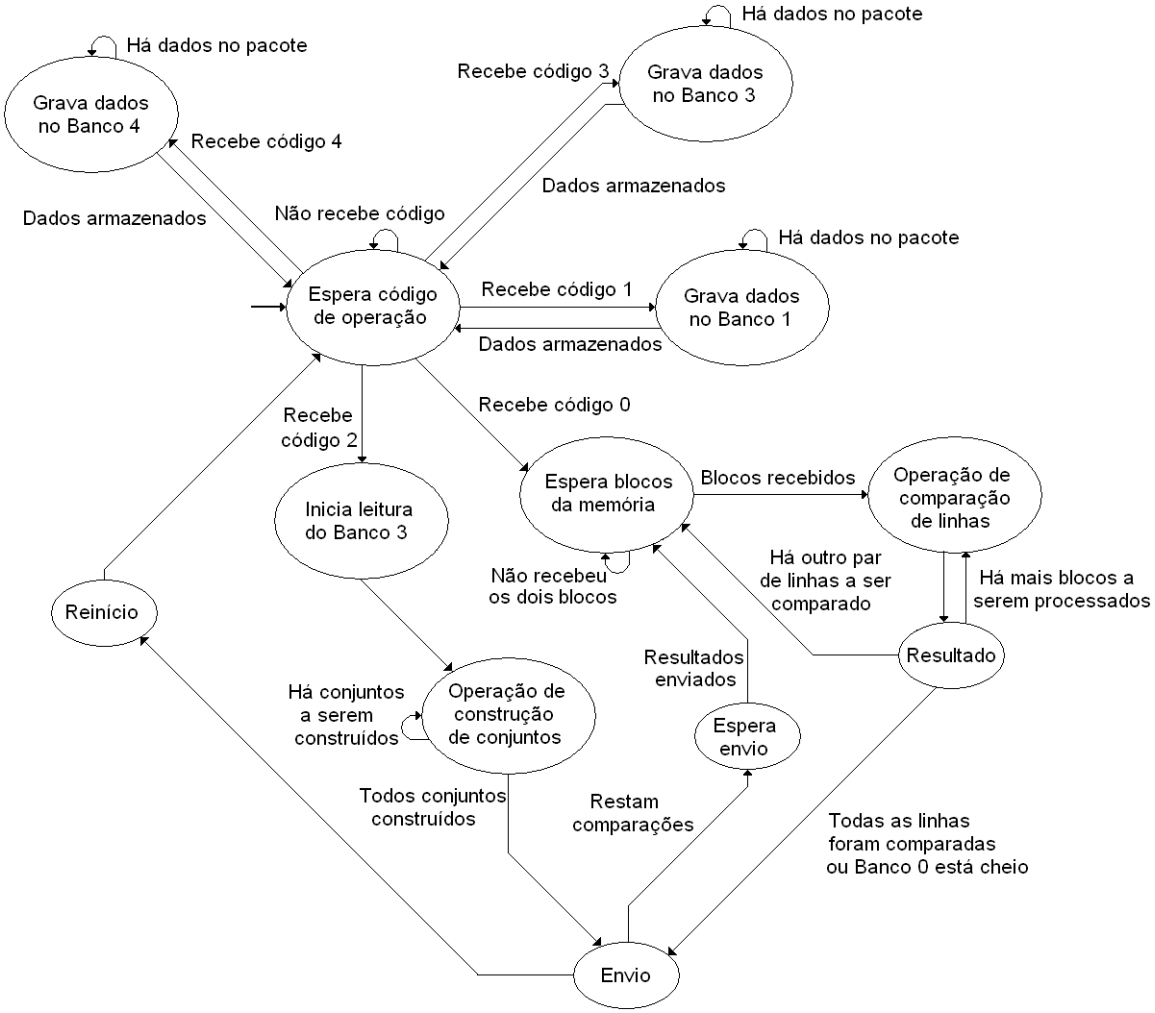


Figura 5.18: Máquina de estados do controle da quarta implementação

A Figura 5.19 mostra, mais detalhadamente, a interação do módulo controle com os demais módulos que compõem o circuito em hardware da solução híbrida.

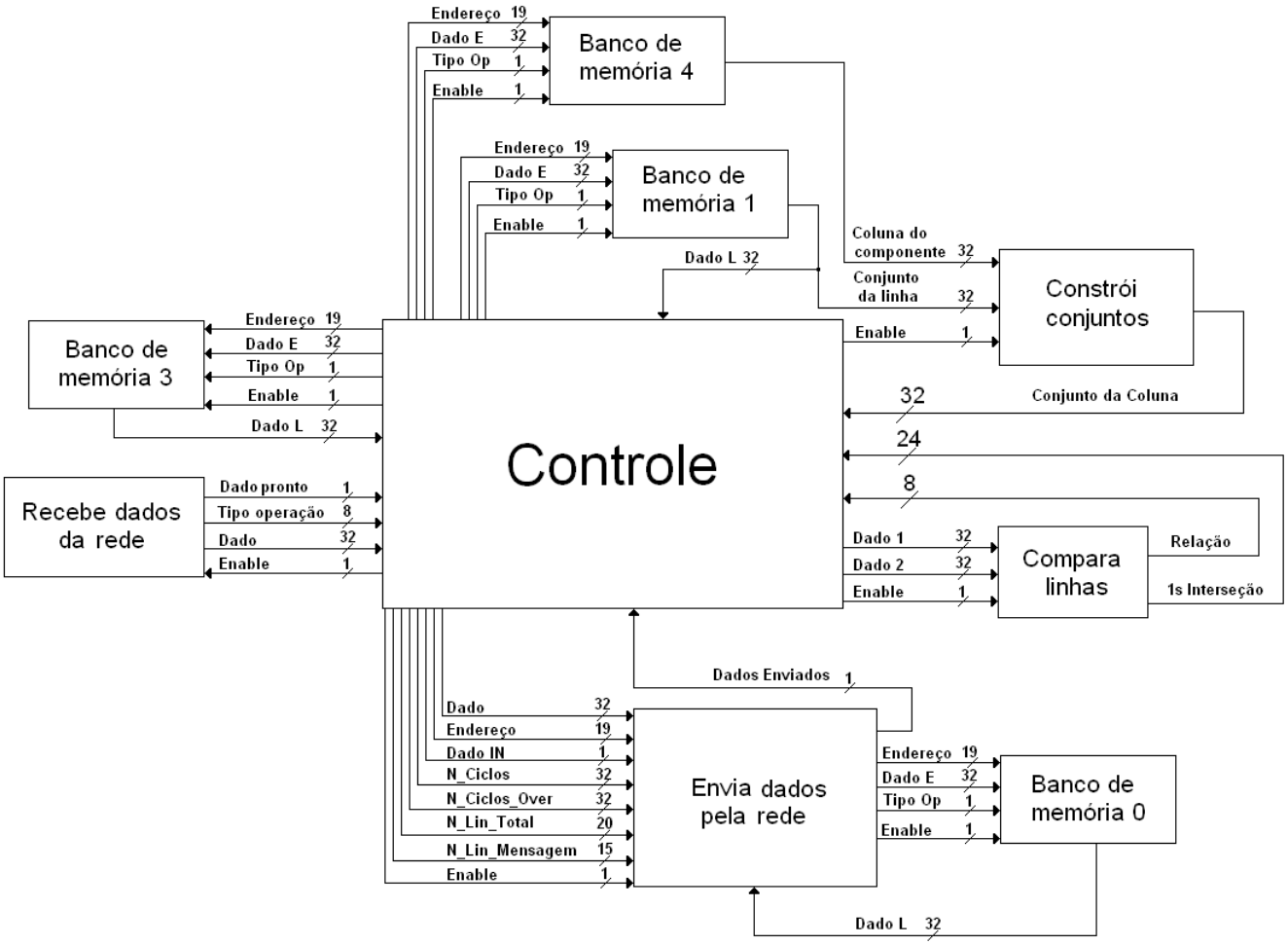


Figura 5.19: Módulos do hardware da implementação híbrida e interações entre eles

5.4.5 Comparação Total em Paralelo e Construção de Conjuntos em Paralelo

Esta implementação introduz o uso de dois comparadores de linhas e dois construtores de conjuntos, utilizando todos os bancos de memória. Agora o banco 2 armazena metade dos dados contidos no banco 1, e isto requer que o estado do autômato que preenche o banco 1 faça isto com o banco 2, intercalando qual banco recebe os dados.

Nos estados do autômato que controlam a comparação de linhas, é feita uma modificação no estado de resultados para somar as interseções calculadas, e realizar a operação OR no resultado das comparações realizados pelos dois comparadores, antes de passar estes resultados para o módulo de envio, que se encarregará de gravá-los no banco 0. Nos controladores adicionais à máquina de estados, são adicionados os pedidos de

leitura (no primeiro controlador) do banco 2, e o encaminhamento dos dados do banco 2 para o segundo comparador (no segundo controlador).

No estado que controla a escrita dos resultados da construção de conjuntos no banco 0, são incluídas operações para que se faça a escrita do bloco do conjunto calculado pelo primeiro construtor e, em seguida, a escrita do bloco do conjunto calculado pelo segundo construtor. Nos três controladores adicionais ao autômato, que tratam do apoio à construção dos conjuntos, são feitas modificações para que sejam lidos os dados do banco 2 e que sejam encaminhados para o segundo construtor.

Como a leitura dos índices das linhas do componente, feita pelo primeiro controlador adicional que controla o banco 3, define a operação ou não dos outros dois controladores adicionais, é feita neste controlador a pausa de um ciclo na leitura caso o componente possua apenas uma linha. Isto é feito para que haja tempo suficiente para a escrita, no banco 0, do bloco criado pelo segundo construtor de conjuntos.

O segundo controlador adicional, que controla agora os bancos 1 e 2, é alterado para mandar ler o bloco da linha contida no banco 2 no mesmo endereço que o utilizado para acessar o banco 1. Assim, serão processados dois blocos consecutivos de cada linha por vez pelos construtores de conjuntos. É realizada a pausa quando há apenas uma linha no componente, pois o segundo controlador só faz a leitura caso haja dado lido do banco 3, e como o primeiro controlador pausa por um ciclo esta leitura, este controlador acaba pausando também a sua leitura.

O terceiro controlador adicional, que controla o banco 4, permanece inalterado, e assim como o segundo controlador, realiza a pausa quando o componente possui apenas uma linha, em decorrência de seu funcionamento depender da disponibilidade do dado lido do banco 3.

5.5 Comunicação

A comunicação entre o algoritmo em software e o circuito implementado em hardware é feita utilizando-se a interface de rede disponível no PC e na placa que contém a FPGA. Em cada implementação híbrida do programa (descritas na Seção 5.4) foram utilizados pacotes com diferentes modos de organização de seus dados. No circuito em hardware foram criados dois módulos, um para recebimento e outro para envio dos pacotes, como mostra a Figura 5.19.

O módulo de recebimento consiste de uma máquina de estados que recebe quatro bits

por vez da rede, retira os cabeçalhos e campos de controle do pacote e encaminha os dados para o módulo de controle, 32 bits por vez. Já o módulo de envio precisa calcular o *checksum* e o CRC (*Cyclic Redundancy Check*) do pacote a ser enviado e manipular o banco de memória 0 nas implementações que usam a memória. Como os *clocks* do módulo de recebimento e de envio trabalham com metade do *clock* utilizado no resto do hardware (devido à limitação imposta pelo padrão IEEE 802.3) é necessário fazer a sincronização entre estes dois domínios de *clock*.

No envio e comparação de clones por demanda, os dados são enviados pelo software intercalando 32 bits da primeira linha com 32 bits da segunda linha. O módulo de recebimento espera receber os 32 bits para repassá-los para o módulo de controle. Os dados de resultado contêm um byte com a relação entre as linhas e 24 bits com o número de interseções entre as duas linhas de M.

No envio total e comparação de clones por demanda, o software envia a matriz M por linhas utilizando o número mínimo de pacotes possível. O software calcula quantas linhas de M cabem em um pacote[3] junto com o código da operação de um byte. O hardware apenas armazena seqüencialmente as linhas de M em seu banco de memória 1. Para pedir as comparações de linhas, o software monta o pacote com o código da operação e os endereços do primeiro bloco das duas linhas, ambos com 32 bits (são utilizados apenas 19 bits para o endereço, mas como o tamanho dos dados passados do módulo de recebimento para o controle é de 32 bits, o endereço do primeiro bloco de cada linha é completado com 0s). O software também envia um pacote contendo o tamanho das linhas de M para orientar as comparações de linhas. O resultado da comparação das linhas é armazenado no banco 0, controlado pelo módulo de envio. Para isto, o controle principal pede ao módulo de envio para escrever os dados na memória, e em seguida será feito o envio destes dados. O pacote de retorno tem o mesmo formato da primeira implementação.

Quando a construção de conjuntos é realizada, o SW envia ao HW os componentes e os endereços do primeiro bloco (em relação ao banco 1) das linhas que pertencem ao componente. O HW retorna os conjuntos calculados ao SW. Cada endereço (ou índice) do primeiro bloco possui 32 bits no pacote, e é colocado o número máximo possível de índices em cada pacote. O hardware armazena seqüencialmente estes dados no banco 3. Em seguida, o software envia o componente permutado por colunas, com o máximo de colunas por pacote. O hardware recebe estas colunas e as armazena seqüencialmente no banco 4. Também são enviados pacotes que indicam o tamanho das linhas em blocos, o tamanho das colunas em blocos, e quando deve-se iniciar a construção de conjuntos. Após o cálculo de todos os conjuntos, o módulo de envio realiza a comunicação de um conjunto

[3] O tamanho máximo de um pacote é de 1500 bytes, incluindo os campos de cabeçalho e CRC

de coluna por pacote, até que todos os dados do banco 0 tenham sido enviados.

Com a comparação total, como são feitas as comparações de todas as linhas entre si, o software envia apenas M, e espera o hardware retornar os resultados. O resultado de cada comparação é armazenado inteiramente em 32 bits, onde os primeiros 8 bits são o tipo de relação entre as linhas, e os outros 24 bits são o número de interseções entre o par de linhas. Desta forma, acontecem rodadas de comunicação, onde o hardware preenche o banco 0. Cada posição deste banco contém a resposta da comparação de um par de linhas. O hardware envia o número máximo de posições da memória, por pacote, para o software. Após uma rodada de comunicação, com o envio de vários pacotes, o software volta a esperar a próxima rodada, caso haja comparações a serem feitas, e o hardware volta a comparar as linhas até que o banco 0 seja preenchido e haja uma nova rodada de comunicação. Também é comunicado ao hardware o endereço do primeiro bloco da última linha de M, para que o hardware saiba quando terminar suas comparações. A comunicação na construção de conjuntos permanece da mesma forma, com uma otimização. Anteriormente, um pacote retornado pelo hardware continha apenas um conjunto do componente. Agora, é colocado o número máximo de conjuntos que cabe em um pacote (um conjunto possui o mesmo número de blocos que uma linha de M).

O uso de comparadores de linhas e construtores de conjuntos em paralelo não requer muitas modificações, há apenas a mudança no tamanho das linhas, enviado pelo software ao hardware. Como as linhas de M são divididas em 2 bancos, o tamanho utilizado para os cálculos também deve ser dividido para o correto funcionamento.

5.5.1 Detalhamento da Comunicação

A placa que acomoda a FPGA possui um *chip* encarregado de tratar as modulações e codificações da camada física do modelo de rede empregado. Porém, não é feito nenhum controle no nível da camada de enlace e posteriores (considerando-se o modelo OSI para redes de computadores [Tan88]). De uma maneira simplificada, o padrão *Ethernet* (IEEE 802.3) [IEE05] especifica que a transmissão e recepção da mensagem entre as camadas de rede e enlace sejam feitas utilizando 4 sinais (4 bits desta forma) para envio e 4 sinais para recepção, havendo 2 sinais auxiliares, um para indicar que o dado pode ser enviado e outro para indicar que um dado foi recebido. Também é especificado que devem ser utilizados 2 *clocks* que temporizam o envio e recepção dos 4 bits de cada canal de comunicação.

Assim, foi necessário implementar na FPGA módulos para controle de envio e recebimento de dados. A comunicação é feita com trocas de pacotes UDP (*User Datagram*

Protocol), um protocolo da camada de transporte, não orientado a conexão e sem confirmação de recebimento.

Na FPGA, o módulo de recebimento de dados teve sua implementação simplificada, devido ao PC e a placa estarem conectados diretamente por um único cabo. Assim, verificações realizadas pelo receptor (CRC, *checksum*, ip de destino) de um pacote UDP puderam ser ignoradas, uma vez que os pacotes recebidos são sempre do PC, e o cabo é muito curto para sofrer atenuações, ruídos e outros problemas que possam corromper os dados.

O módulo de envio de dados para o PC precisou seguir as regras de encapsulamento e enquadramento, pois o programa que roda no PC hospedeiro só receberá mensagens UDP cujos pacotes estejam de acordo com o padrão *Ethernet*. Portanto, foi necessário construir e enviar em um pacote campos de cabeçalho, *checksum*, e CRC junto com os dados, seguindo a especificação IEEE 802.3. Assim, este módulo é composto de quatro sub-módulos: um módulo que calcula o *checksum* do cabeçalho UDP, um módulo que calcula o CRC do pacote, um módulo que realiza o envio dos 4 bits de dados por vez e um módulo que realiza o controle.

Capítulo 6

Resultados Experimentais

Este capítulo descreve os resultados obtidos com cada implementação do algoritmo para o problema dos 1s consecutivos. Descreve também os dados utilizados nestes experimentos e a plataforma em que tais soluções foram desenvolvidas.

6.1 Plataforma de Desenvolvimento e Implementação

Como dito em capítulos anteriores, a plataforma de desenvolvimento consiste de um computador pessoal contendo um processador de propósito geral, e um *kit* de desenvolvimento composto por uma placa multimídia, contendo FPGA, bancos de memória, portas de comunicação, conectores de áudio e vídeo, entre outros. A maneira de comunicação mais eficaz e com menor latência disponibilizada por este conjunto foi através da interface de rede de ambos os dispositivos, utilizando um cabo de par trançado do tipo cruzado (*crossover*).

O sistema utilizado para realizar os testes é composto de um computador equipado com um processador Athlon 64 a 2.2 GHz com 2 GB de memória RAM, placa de rede com conexão a 100 Mb/s, e uma placa multimídia da Xilinx [Xil05] equipada com uma FPGA Virtex-II XCV2000-FF896, com cinco bancos de memória, totalizando 10 MB e operando com a freqüência de *clock* de 50 MHz.

O computador hospedeiro opera com o sistema operacional Windows XP. A implementação totalmente em SW, assim como as partes em SW da solução híbrida foram desenvolvidas na linguagem de programação C, utilizando o compilador Borland C++

5.02 [Bor07] com suas opções de otimização padrão, definidas pelo programa. As partes em HW das soluções híbridas foram desenvolvidas utilizando a linguagem de descrição de hardware VHDL. As ferramentas ISE e ModelSim Xilinx Edition da Xilinx [Xil06b, Xil07a] foram utilizadas para simulação, testes e síntese do circuito em HW.

6.2 Clones e Sondas Utilizados nos Experimentos

Foram utilizados cromossomos de dois seres vivos para gerar as matrizes utilizadas nos experimentos. O primeiro foi o cromossomo 5 da *Arabidopsis thaliana* (uma planta da família da mostarda), e o segundo foi um dos *contigs*[1] que formam o cromossomo 2 do *Homo sapiens*, obtidos no NCBI (*National Center for Biotechnology Information*) [NCB07].

Tendo a seqüência de cada cromossomo, foi utilizado o conjunto de programas denominado *Genfrag* [EB93, EB94] para gerar os clones e as sondas. Foram geradas as posições de início e fim de cada clone ou sonda, e com estas informações uma matriz binária foi construída. O número de fragmentos, o tamanho médio de cada fragmento, a variação deste tamanho médio, o tamanho da cobertura e o número usado como semente do gerador de números aleatórios permitem ao *Genfrag* gerar um conjunto de fragmentos aleatório, mas de forma determinística, ou seja, com os mesmos valores de entrada, é gerado o mesmo conjunto de fragmentos.

Para o primeiro cromossomo, foram gerados 4.096 sondas e 3.285 clones com conjuntos de sondas distintos, assim, não houve linhas repetidas na matriz. Os clones possuem tamanho de 8.436.282 pares de bases, e as sondas possuem tamanho de 659 pares de bases, enquanto que o cromossomo possui um total de 26.992.728 pares de bases.

Para o *contig* do segundo cromossomo, foram gerados 4.096 sondas e 2.881 clones com conjuntos de sondas distintos, deste modo, também não houve linhas repetidas na matriz. Os clones possuem tamanho de 27.068.901 pares de bases, as sondas possuem tamanho de 30 pares de bases e o *contig* do cromossomo possui 84.213.157 pares de bases.

[1] Pedaço do cromossomo que foi totalmente seqüenciado. Quando o cromossomo inteiro é seqüenciado, existe apenas um *contig*, formado pelo cromossomo completo.

6.3 Resultados das Implementações

Para cada matriz de entrada, foram executados o algoritmo totalmente em SW, e as cinco versões do programa híbrido, descritas no Capítulo 5 e listadas a seguir:

- SW/HW 1: Envio e Comparação de Clones por Demanda;
- SW/HW 2: Envio Total e Comparação de Clones por Demanda;
- SW/HW 3: Envio Total e Comparação de Clones por Demanda com Construção Simples de Conjuntos;
- SW/HW 4: Comparação Total com com Construção Simples de Conjuntos;
- SW/HW 5: Comparação Total em Paralelo com Construção de Conjuntos em Paralelo.

O cálculo dos tempos de execução foi realizado no código presente no PC, utilizando os ciclos contados na FPGA e os tempos marcados pela parte em software. Desta forma, cada pacote enviado pela FPGA pode conter, após os dados de resposta da operação solicitada, o número de ciclos gastos no processamento e o número de ciclos de *overhead* de controle. Os resultados obtidos pelas implementações híbridas foram comparados com a implementação feita totalmente em software e submetidos a testes para determinar sua correção.

6.3.1 Freqüência do *Clock*, Área da FPGA e Tempo de Síntese

Através da ferramenta de síntese utilizada, foi possível coletar medidas de porcentagem de área ocupada da FPGA, de freqüência máxima do *clock* que cada circuito permite e de tempo de síntese, mostradas na Tabela 6.1, para cada implementação híbrida.

Implementação	Área ocupada	Freqüência do *clock*	Tempo de síntese (s)
SW/HW 1	20%	68.538 MHz	118
SW/HW 2	21%	68.646 MHz	119
SW/HW 3	32%	65.514 MHz	175
SW/HW 4	31%	62.210 MHz	174
SW/HW 5	34%	62.210 MHz	177

Tabela 6.1: Área ocupada da FPGA, freqüência do *clock* e tempo de síntese

Embora a freqüência máxima do *clock* do circuito sintetizado de cada implementação seja maior que 50 MHz, o gerador de *clock* da placa fornece dois *clocks* para a FPGA, um de 25 MHz e outro de 50 MHz. Portanto, foi utilizada para todas as implementações a freqüência de 50 MHz.

Estes resultados mostram que a cada versão, a complexidade do controle aumenta, e novos módulos (de acesso aos bancos de memória, processamento de conjunto de colunas) são adicionados, refletindo no aumento da área consumida. O envio e comparação de clones por demanda tem o controle mais simples e não utiliza o controlador de memória, fazendo com que ocupe menos área que as demais implementações. Na implementação seguinte, é necessário adicionar ao controle o circuito para manipular o armazenamento e recuperação dos dados no banco de memória e, desta forma, o controle se tornou mais complexo, aumentando a área necessária na FPGA. Com a inclusão da construção simples de conjuntos, o controle precisa de mais lógica para manipular os bancos de memória adicionais, e é inserido o novo módulo para construir os conjuntos de colunas, justificando o aumento da área e diminuição da freqüência máxima do *clock* (o período que um sinal leva para percorrer a lógica é maior, neste caso). Já na comparação total, o controle fica menor, pois não possui a parte que trata cada pedido de comparação de linhas (faz as comparações de uma vez), porém são acrescentados mecanismos para controle do armazenamento dos resultados (para evitar que a memória fique cheia e mais dados sejam gravados nela). A última versão acrescenta um comparador de linhas e um construtor de conjuntos, aumentando a área utilizada. Entretanto, o controle não sofre muitas alterações, uma vez que estas unidades adicionais trabalham de forma síncrona com as unidades originais e compartilham muitos dos sinais de controle.

Esta baixa ocupação da área da FPGA possibilitaria o aumento do número de comparadores de linhas e construtores de conjunto que operam em paralelo, se não houvesse a limitação do número de bancos de memória, ou se houvesse um acoplamento mais forte entre processador hospedeiro e FPGA. O volume de dados que ocupa um banco de memória, torna inviável a implementação da memória dentro da FPGA, utilizando sua área livre para implementar um elemento de armazenamento.

6.3.2 Clones e Sondas de *Arabidopsis thaliana*

A Tabela 6.2 mostra os tempos de execução apenas da operação de comparação de linhas para as diferentes implementações, utilizando a matriz formada com os dados do cromossomo 5 da *Arabidopsis thaliana*. Mostra também o número de pares de linhas comparados em cada implementação do problema dos 1s consecutivos. O tempo sem

comunicação é o tempo gasto com processamento na FPGA para realizar a comparação de linhas. O tempo total é todo o tempo consumido pela operação de comparação de linhas, incluindo tempo de comunicação, tempo de processamento na FPGA e tempo de processamento no SW. Com uma arquitetura com acoplamento mais forte o custo de comunicação pode ser reduzido, e por isto é feita a distinção dos tempos sem comunicação e total.

Implementação	Comparação de linhas		
	Número de comparações	Tempo sem comunicação	Tempo total
Algoritmo em SW	7.879	–	0,70
SW/HW 1	7.879	0,04	1,30
SW/HW 2	7.879	0,04	0,67
SW/HW 3	7.879	0,04	0,63
SW/HW 4	5.393.970	27,72	29,95
SW/HW 5	5.393.970	13,92	16,14

Tabela 6.2: Tempos (em segundos) da comparação de linhas para a primeira matriz

Os resultados mostram que, para esta matriz, a comparação de linhas é necessária poucas vezes, e por isto a comparação de todas as linhas entre si não oferece benefícios, pois realiza um número muito maior de comparações, em relação ao volume de operações realmente necessárias. Mesmo considerando o tempo total da operação de comparação de linhas (que inclui o *overhead* de comunicação), as implementações SW/HW 2 e 3, que realizam a comparação por demanda utilizando índices, foram mais rápidas que a implementação em SW. Mesmo assim, a diferença de tempo é insignificante para esta matriz. Estes resultados indicam que a matriz possui poucos componentes, uma vez que são necessárias poucas comparações para colocar todas as linhas em seus componentes.

A Tabela 6.3 mostra os tempos de execução apenas da operação de construção de conjuntos, para esta mesma matriz. As duas primeiras implementações híbridas realizam a mesma construção de conjuntos que o algoritmo em software, isto é, não realizam esta operação em HW.

Implementação	Construção de conjuntos	
	Tempo sem comunicação	Tempo total
Algoritmo em SW	–	571,45
SW/HW 1	–	589,34
SW/HW 2	–	572,59
SW/HW 3	34,45	35,17
SW/HW 4	34,45	35,16
SW/HW 5	17,22	17,94

Tabela 6.3: Tempos (em segundos) da construção de conjuntos para a primeira matriz

É possível ver uma grande diferença de tempo entre as implementações em SW e em HW da construção de conjuntos. O *speedup* entre a versão que possui construtores em paralelo (SW/HW 5) chega a 31,9 em relação à implementação em software. As implementações SW/HW 3 e 4 realizam a construção simples de conjuntos e seu desempenho é o mesmo. Através destes dados pode-se constatar que a construção em paralelo dos conjuntos (implementação SW/HW 5) possui ganho de 100% no tempo sem comunicação em relação à construção simples. Isto ocorreu porque cada construtor pode trabalhar independentemente e em paralelo com metade dos dados, havendo apenas um overhead mínimo para a escrita dos resultados no banco de memória (que exige uma escrita por vez).

A Tabela 6.4 mostra o tempo total das operações de comparação de linhas e construção de conjuntos. Mostra também o tempo de execução total de cada implementação (incluindo todos os passos do algoritmo de Fulkerson e Gross) e o *speedup* de cada implementação híbrida em relação à implementação em software. As duas primeiras implementações híbridas (SW/HW 1 e 2) realizam a construção de conjuntos em software, da mesma forma que o algoritmo em SW.

Implementação	Comparação de linhas	Construção de conjuntos	Tempo total	*Speedup*
Algoritmo em SW	0,70	571,45	573,11	–
SW/HW 1	1,30	589,34	591,45	0,97
SW/HW 2	0,67	572,59	574,11	1,00
SW/HW 3	0,63	35,17	36,67	15,63
SW/HW 4	29,95	35,16	65,89	8,70
SW/HW 5	16,14	17,94	34,86	16,44

Tabela 6.4: Tempos de execução (em segundos) de cada implementação para dados de *Arabidopsis thaliana*

Os tempos medidos indicam que a construção de conjuntos é responsável por grande parte do tempo total de execução, para esta matriz. A primeira e segunda versões híbridas não realizam a construção de conjuntos em hardware e, portanto, o uso da FPGA não causou nenhuma melhoria em relação ao tempo do algoritmo em software (pois são feitas muito poucas operações na FPGA). Em relação à comparação de linhas destas duas versões, o envio de uma linha é feito até 3.284 vezes na implementação SW/HW 1, e nas implementações SW/HW 2 e 3, cada linha é enviada apenas uma vez. Esta mudança na abordagem da comunicação refletiu em menor tempo de comunicação.

Comparando-se as duas últimas implementações, a introdução de um novo comparador de linhas e um novo construtor de conjuntos proporcionaram um ganho de desempenho em torno de 100% para cada tipo de operação. Desta forma, a versão que utiliza comparadores

e construtores em paralelo (SW/HW 5) foi mais rápida até que a versão que realiza envio total e comparação por demanda (SW/HW 3), mesmo realizando 5.386.091 comparações a mais de linhas. O *speedup* obtido com a melhor implementação híbrida (com comparadores em paralelo) para esta matriz, em relação ao algoritmo em software, foi de 16,44. Este resultado foi alcançado com a FPGA rodando a uma freqüência de *clock* 44 vezes menor que o processador de propósito geral, ou seja, se uma FPGA mais moderna e ágil for utilizada, maiores ganhos de desempenho podem ser obtidos.

Por fim, os resultados mostram que a melhor implementação híbrida (SW/HW 5), alcançou um desempenho 1544% melhor que a versão totalmente em software, para esta primeira matriz que possui características não muito favoráveis para as soluções em hardware.

6.3.3 Clones e Sondas de *Homo sapiens*

Esta segunda matriz, formada com os clones e sondas gerados a partir do *contig* do cromossomo 2 do *Homo sapiens*, foi processada da mesma maneira que a matriz de *Arabidopsis thaliana*. Os resultados da operação de comparação de linhas estão presentes na Tabela 6.5.

Implementação	Comparação de linhas		
	Número de comparações	Tempo sem comunicação	Tempo total
Algoritmo em SW	2.080.334	–	214,83
SW/HW 1	2.080.334	10,65	351,53
SW/HW 2	2.080.334	11,07	137,08
SW/HW 3	2.080.334	11,15	124,12
SW/HW 4	4.148.640	21,32	23,06
SW/HW 5	4.148.640	10,70	12,44

Tabela 6.5: Tempos (em segundos) da comparação de linhas para a segunda matriz

Para esta matriz, a execução da comparação de linhas representa uma grande parcela do tempo de execução total do algoritmo em software. Assim, a implementação desta operação em hardware possibilitou a todas as implementações híbridas (exceto à primeira) um ganho substancial de desempenho em relação à implementação em software.

O número de comparações de linhas necessárias para o algoritmo é um pouco maior que a metade do número de comparações de todas as linhas entre si. Desta forma, o tempo sem comunicação das implementações SW/HW 1, 2, e 3 foi praticamente a metade do tempo sem comunicação da implementação SW/HW 4.

Considerando a comunicação, o envio das linhas em cada mensagem utilizado na

implementação SW/HW 1 produz um grande impacto no tempo de comunicação (fazendo com que o tempo total da operação seja maior que o tempo da implementação em software). Este tempo de comunicação foi reduzido com o envio da matriz M uma única vez e o envio apenas dos índices das linhas para as comparações, nas implementações SW/HW 2 e 3. Assim, o ganho de desempenho, com o envio de índices, foi em média de 169% em relação à primeira implementação híbrida e de 65% em relação à implementação em software.

As duas implementações que realizam todas as comparações (SW/HW 4 e 5) e conseqüentemente não precisam do envio de índices, reduziram ainda mais o tempo de comunicação, ganhando em média 636% em relação às implementações que realizam o envio de índices. Isto mostra que, quando o número de comparações é significativo, é melhor realizar todas as comunicações para que não sejam enviados pacotes com pedidos de comparação. Desta forma, a implementação com comparadores em paralelo (SW/HW 5) obteve *speedup* de 17,27, ou seja, foi 1627% melhor que a implementação em SW.

A Tabela 6.6 mostra os tempos de execução da operação de construção de conjuntos, para as diferentes implementações. Novamente, as implementações híbridas SW/HW 1 e 2 realizam a construção de conjuntos da mesma forma que o algoritmo em software, isto é, não utilizam o hardware.

Implementação	Construção de conjuntos	
	Tempo sem comunicação	Tempo total
Algoritmo em SW	–	272,42
SW/HW 1	–	280,55
SW/HW 2	–	248,28
SW/HW 3	15,11	15,61
SW/HW 4	15,11	15,59
SW/HW 5	7,55	8,03

Tabela 6.6: Tempos (em segundos) da construção de conjuntos para a segunda matriz

Assim como os resultados da operação de construção de conjuntos para a primeira matriz, os resultados obtidos mostram ganhos significativos de desempenho para as implementações que realizam na FPGA a construção de conjuntos. O *speedup* da implementação da construção de conjuntos em paralelo (SW/HW 5) em relação à execução em software foi de 33,92, significando um ganho de desempenho de 3292%. A comunicação representou pouco impacto, uma vez que a maior parte dos dados é reutilizada da comparação de linhas, e assim é necessária a comunicação de poucos dados adicionais. Assim como aconteceu com a primeira matriz, o ganho de desempenho com o uso de dois construtores de conjuntos operando em paralelo no hardware foi em torno de 100% (em relação às soluções SW/HW 3 e 4), devido às características inerentes da implementação.

A Tabela 6.7 mostra o tempo total de execução do algoritmo, utilizando esta segunda matriz, para cada implementação.

Implementação	Comparação de linhas	Construção de conjuntos	Tempo total	*Speedup*
Algoritmo em SW	214,83	272,42	487,73	–
SW/HW 1	351,53	280,55	633,56	0,77
SW/HW 2	137,08	248,28	386,47	1,26
SW/HW 3	124,12	15,61	145,53	3,35
SW/HW 4	23,06	15,59	39,02	12,50
SW/HW 5	12,44	8,03	20,84	23,40

Tabela 6.7: Tempo de execução (em segundos) de cada implementação para dados de *Homo sapiens*

O envio e comparação de clones por demanda (SW/HW 1) foi a única implementação cujo tempo total de execução foi maior que o algoritmo em software. Como mostra a Tabela 6.5, a comunicação para a comparação de linhas foi o fator preponderante para este resultado, uma vez que o tempo da operação de comparação na FPGA sem considerar a comunicação foi cerca de 11 segundos.

A inclusão da operação de construção de conjuntos, na comparação por demanda (SW/HW 3) também possibilitou um ganho de desempenho significativo, onde seu desempenho foi 235% melhor que a implementação em software.

A utilização de comparadores de linhas e construtores de conjuntos em paralelo (SW/HW 5) produziu novamente a implementação híbrida com os melhores resultados, uma vez que as duas operações realizadas se beneficiam do paralelismo, pois seus dados podem ser particionados e processados independentemente. Desta forma, esta implementação híbrida obteve ganho de desempenho de 2240% e *speedup* de 23,4, em relação à implementação em software. Este ganho é maior que o obtido com a matriz da *Arabidopsis thaliana* pois as duas operações que são realizadas pela FPGA são responsáveis pela maior parte do tempo de execução do algoritmo.

Capítulo 7

Conclusão

O estudo da computação reconfigurável, de seus dispositivos e, em particular, das características da FPGA, mostrou que esta área possui grande relevância para a computação atual, permitindo que uma nova abordagem seja adotada no desenvolvimento de programas. O fato de permitir que o circuito se adapte às necessidades da aplicação e, desta forma, acelerar sua execução é um dos principais fatores que tornaram as arquiteturas reconfiguráveis objeto de estudo deste trabalho.

Para que fosse medido e comprovado o ganho de desempenho que uma arquitetura reconfigurável pode proporcionar, foi discutido o problema dos 1s consecutivos e implementado um algoritmo para solucioná-lo. Assim como outras soluções para problemas de bioinformática, como construção de árvores filogenéticas e alinhamento e comparação de seqüências de DNA, o algoritmo para o problema dos 1s consecutivos possui grande aplicação prática e contém operações que são executadas de maneira eficiente se realizadas no hardware reconfigurável.

Deste modo, foram realizadas várias implementações híbridas para o algoritmo do problema dos 1s consecutivos, explorando abordagens diferentes tanto em relação à comunicação entre processador convencional e FPGA, quanto às estratégias de execução, tais como paralelismo e *pipeline*.

Não foram encontrados na literatura trabalhos com soluções para o problema dos uns consecutivos em hardware, seja com lógica fixa ou reconfigurável. Assim, esta dissertação traz uma importante contribuição, por apresentar um solução híbrida para este problema.

7.1 Resultados

Os resultados obtidos com cada implementação híbrida, quando comparados com a implementação em software, mostram a capacidade do dispositivo reconfigurável em executar as duas operações que mais contribuem para o tempo total do algoritmo. Estas operações (a comparação de linhas e a construção de conjuntos) realizam tarefas repetitivas e que exigem um fluxo de dados contínuo, características estas que são beneficiadas pela execução no dispositivo reconfigurável.

Um dos maiores problemas da arquitetura utilizada foi a comunicação entre as partes do programa que executam no software e as partes implementadas no hardware. Devido às limitações impostas pela plataforma utilizada, foi necessário propor várias abordagens de comunicação para que os benefícios da execução no hardware reconfigurável não fossem anulados pelo *overhead* de comunicação. Assim, em uma arquitetura com acoplamento mais forte o custo da comunicação pode ser reduzido, e as vantagens da utilização da FPGA podem se tornar mais evidentes.

Mesmo com as limitações de acoplamento da arquitetura utilizada, pode-se ver o ganho que cada operação realizada na FPGA proporcionou, e o ganho no tempo de execução total do algoritmo. Os ganhos foram expressivos para os dois conjuntos de clones e sondas utilizados (obtidos a partir dos cromossomos reais de *Arabidopsis thaliana* e *Homo sapiens*), utilizando uma abordagem com menos comunicação e com o uso de *pipeline* e paralelismo. O *speedup* da implementação híbrida em relação à implementação apenas em software chegou a 23,4, equivalendo a um ganho de desempenho de 2240%. Um resultado indiscutivelmente melhor que a implementação em software.

7.2 Trabalhos Futuros

A área de bioinformática possui um vasto campo de problemas que podem ser beneficiados pelas arquiteturas reconfiguráveis. Como trabalhos futuros são apresentadas três abordagens que podem produzir resultados positivos para esta área de estudo.

Uma maneira de aprimorar ainda mais os resultados obtidos neste trabalho é mudar o modelo da arquitetura híbrida, utilizando um acoplamento maior. Com isto pode-se reduzir o tempo de comunicação nas implementações que comparam linhas por demanda e assim eliminar a necessidade de realizar todas as comparações de linhas.

Com este acoplamento maior, pode-se também eliminar a limitação do tamanho da

memória e, com isto, a implementação híbrida pode processar matrizes maiores e explorar mais paralelismo. Desta forma, é possível utilizar uma das arquiteturas híbridas descritas no Capítulo 2, onde grau de acoplamento, dentre outros fatores, definirá a estratégia a ser utilizada na implementação.

Outra alternativa, para melhorar o desempenho do algoritmo em SW, é realizar uma implementação utilizando *multithreading* e, com isto, aproveitar os recursos de um processador de propósito geral que possua vários núcleos.

Por fim, outro trabalho importante é a exploração de outras soluções para o problema dos 1s consecutivos, [Hsu92, BL76, AS95, CY91], utilizando uma arquitetura híbrida, como realizado neste trabalho.

Referências Bibliográficas

[Acq07] Acqiris. Acqiris – Data Conversion Instruments, Setembro 2007. Disponível em: http://www.acqiris.com.

[Alp07] Alpha Data. Alpha Data: FPGA Based Solutions for High End Applications, 2007. Disponível em: http://www.alpha-data.com.

[Alt06] Altera. Stratix III FPGAs vs. Xilinx Virtex-5 Devices: Architecture and Performance Comparison. Relatório técnico, Altera, Novembro 2006.

[Alt07] Altera. Stratix III Overview, 2007. Disponível em: http://www.altera.com.

[AS95] F. S. Annexstein e R. P. Swaminathan. On Testing Consecutive-Oones Property in Parallel. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, páginas 234–243. Julho 1995.

[Bar98] M. Barr. A Reconfigurable Computing Primer. *Multimedia Systems Design*, páginas 44–47, Setembro 1998.

[BL76] K. S. Booth e G. S. Lueker. Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-tree Algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, Dezembro 1976.

[Bor07] Borland. Borland: Aberta a seus Processos, Ferramentas e Plataformas, Novembro 2007. Disponível em: http://www.borland.com/br/.

[BP00] K. Bondalapati e V. K. Prasanna. Reconfigurable Computing: Architectures, Models and Algorithms. *Current Science: Special Section on Computational Science*, 78(7):828–837, Abril 2000.

[BP02] K. Bondalapati e V. K. Prasanna. Reconfigurable Computing Systems. *Proceedings of the IEEE*, 90(7):1201–1217, Julho 2002.

[BR96] S. Brown e J. Rose. FPGA and CPLD Architectures: A Tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, Junho 1996.

[BV00] S. Brown e Z. Vranesic. *Fundamentals of Digital Logic with VHDL Design*. McGraw Hill, 2000.

[CES07] CESYS. CESYS GmbH FPGA boards, 2007. Disponível em: http://www.cesys.com.

[CH02] K. Compton e S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, Junho 2002.

[Cha03] R. D. Chamberlain. The Mercury system: Exploiting truly fast hardware for data search. In *Int'l Workshop on Storage Network Architecture and Parallel I/Os*, páginas 65–72. Setembro 2003.

[CLR90] T. H. Cormen, C. E. Leiserson, e R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CY91] L. Chen e Y. Yesha. Parallel Recognition of the Consecutive Ones Property with Applications. *Journal of Algorithms*, 12(3):375–392, Setembro 1991.

[DW99] A. DeHon e J. Wawrzynek. Reconfigurable Computing: What, Why, and Implications for Design Automation. In *Proceedings of the 36th Annual Design Automation Conference (DAC)*, páginas 610–615. Junho 1999.

[EB93] M.L. Engle e C. Burks. Artificially Generated Data Sets for Testing DNA Fragment Assembly Algorithms. *Genomics*, 16:286–288, 1993.

[EB94] M.L. Engle e C. Burks. Genfrag 2.1: New Features for More Robust Fragment Assembly Benchmarks. *CABIOS*, 10:567–568, 1994.

[FD87] D.F. Feng e R.F. Doolittle. Progressive Sequence Alignment as a Prerequisite to Correct Phylogenetic Trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.

[FG65] D. Fulkerson e O. Gross. Incidence Matrices and Interval Graphs. *Pacific Journal of Mathematics*, 15(3):835–855, Abril 1965.

[Fre07] P. Freidin. FPGA Boards and Systems at FPGA, Junho 2007. Disponível em: http://www.fpga-faq.com/FPGA_Boards.shtml.

[Gen07] General Eletric. Rugged 6U VME320 Dual Virtex-4 FX100 & PowerPC 7448 Processor with one PMC/XMC Site, 2007. Disponível em: http://www.gefanucembedded.com/products/2046.

[GS98] M.B. Gokhale e J.M. Stone. NAPA C: Compiling for a Hybrid RISC/FPGA Architecture. In *IEEE 6th Symposium on Field-Programmable Custom Computing Machines,*, volume 15, páginas 126–135. Abril 1998.

[Hsu92] W. L. Hsu. A Simple Test for the Consecutive Ones Property. In *Third International Symposium on Algorithms and Computation*, volume 650 de *Lecture Notes in Computer Science*, páginas 459–468. Dezembro 1992.

[IEE05] IEEE Computer Society. IEEE Standard for Information Technology – Telecommunications and Information Exchange Between Systems – Local and Metropolitan Area Networks – Specific Requirements. Relatório técnico, IEEE, Dezembro 2005.

[JRC+04] R. P. Jacobi, M. A. Rincón, L. G. A. Carvalho, C. H. Llanos, e R. W. Hartenstein. Reconfigurable Systems for Sequence Alignment and for General Dynamic Programming. In *III Brazilian Workshop on Bioinformatics*, páginas 25–32. Outubro 2004.

[KBC+04] P. Krishnamurthy, J. Buhler, R. D. Chamberlain, M. A. Franklin, K. Gyang, A. Jacob, e J. Lancaster. Biosequence Similarity Search on the Mercury System. In *Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, páginas 365–375. Setembro 2004.

[MC07] X. Meng e V. Chaudhary. An Adaptive Data Prefetching Scheme For Biosequence Database Search on Reconfigurable Platforms. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, páginas 140–141. 2007.

[ML03] T. S. T. Mak e K. P. Lam. High Speed GAML-based Phylogenetic Tree Reconstruction Using HW/SW Codesign. In *Proceedings of The 2nd IEEE Computer Society Bioinformatics Conference*, páginas 470–473. Agosto 2003.

[ML04] T. S. T. Mak e K. P. Lam. Embedded Computation of Maximum-Likelihood Phylogeny Inference Using Platform FPGA. In *Proceedings of The 3rd International IEEE Computer Society Computational Systems Bioinformatics Conference*, páginas 512–514. Agosto 2004.

[Nal07] Nallatech. High Performance FPGA Computing Solutions for Defense and HPC - Nallatech, 2007. Disponível em: http://www.nallatech.com.

[NCB07] NCBI – National Center for Biotechnology Information. NCBI HomePage, 2007. Disponível em: http://www.ncbi.nlm.nih.gov/.

[OSM05a] T. Oliver, B. Schmidt, e D. Maskell. Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, páginas 229–237. Janeiro 2005.

[OSM+05b] T. Oliver, B. Schmidt, D. L. Maskell, D. Nathan, e R. Clemens. Multiple Sequence Alignment on an FPGA. In *Proceedings of The 11th International Conference on Parallel and Distributed Systems*, páginas 326–330. Julho 2005.

[RBMR05] K. Regester, J. H. Byun, A. Mukherjee, e A. Ravindran. Implementing Bioinformatics Algorithms on Nallatech-Configurable Multi-FPGA Systems. *Xcell Journal Online*, (53):100–103, Março 2005.

[SM97] J. C. Setubal e J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[SW81] T. F. Smith e M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[Tan88] A. S. Tanenbaum. *Computer networks: 2nd edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-162959-X.

[TB01] R. Tessier e W. Burleson. Reconfigurable Computing for Digital Signal Processing: A Survey. *Journal of VLSI Signal Processing*, 28(1):7–27, Junho 2001.

[Xil02a] Xilinx. *MicroBlaze and Multimedia Development Board User Guide*, 2002.

[Xil02b] Xilinx. *Virtex-II Data Sheet*, Setembro 2002.

[Xil05] Xilinx. Xilinx Multimedia Board, Maio 2005. Disponível em: http://www.xilinx.com/products/boards/multimedia.

[Xil06a] Xilinx. Evaluation Platform, Virtex-II Pro ML300, 2006. Disponível em: http://www.xilinx.com.

[Xil06b] Xilinx. *ISE Overview*, Março 2006. Disponível em: http://www.xilinx.com.

[Xil06c] Xilinx. PowerPC & MicroBlaze Development kit Virtex-4 FX12 Edition, 2006. Disponível em: http://www.xilinx.com.

[Xil06d] Xilinx. PowerPC 405 Processor, Fevereiro 2006. Disponível em: http://www.xilinx.com.

[Xil06e] Xilinx. Virtex-4 Capabilities, Novembro 2006. Disponível em: http://www.xilinx.com.

[Xil07a] Xilinx. ModelSim Xilinx Edition, Setembro 2007. Disponível em: http://www.xilinx.com.

[Xil07b] Xilinx. Virtex-5 Overview, 2007. Disponível em: http://www.xilinx.com.

[Xil07c] Xilinx. *Virtex-5 User Guide*, Fevereiro 2007. Disponível em: http://www.xilinx.com.