WALDIR DENVER MUNIZ MEIRELES FILHO

VIRTUALIZAÇÃO E EXECUÇÃO DE ALGORITMOS EM FPGA: UM ALGORITMO DE *MODULO SCHEDULING* PARA ARRANJOS RECONFIGURÁVEIS DE GRÃO GROSSO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS – BRASIL
2014

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
M514v
2014

Meireles Filho, Waldir Denver Muniz, 1983-
Virtualização e execução de algoritmos em FPGA : um algoritmo de *modulo scheduling* para arranjos reconfiguráveis de grão grosso / Waldir Denver Muniz Meireles Filho. – Viçosa, MG, 2014.
xiii, 93f. : il. ; 29 cm.

Orientador: Ricardo dos Santos Ferreira.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Inclui bibliografia.

1. Programação (Matemática). 2. Algoritmos. 3. Modulo Scheduling. 4. Software Pipeline. 5. Arquitetura Reconfigurável. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-graduação em Ciência da Computação. II. Título.

CDD 22. ed. 005.711

WALDIR DENVER MUNIZ MEIRELES FILHO

VIRTUALIZAÇÃO E EXECUÇÃO DE ALGORITMOS EM FPGA: UM ALGORITMO DE MODULO SCHEDULING PARA ARRANJOS RECONFIGURÁVEIS DE GRÃO GROSSO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae.*

APROVADA: 12 de março de 2014.

José Augusto Miranda Nacif
(Coorientador)

Luiz Fernando Etrusco Moreira

Ricardo dos Santos Ferreira
(Orientador)

Agradecimentos

Agradeço primeiramente a Deus por ter me dado mais esta oportunidade. Agradeço aos meus pais Waldir e Sirléa pelo amor e apoio incondicional em todos os momentos. Agradeço à minha namorada Dolores pela paciência e compreensão. Agradeço aos meus irmãos Jonas, Leonel e Lais pela boa convivência e pelo apoio. Agradeço à Universidade Federal de Viçosa pela oportunidade de estudar e a todos os docentes do Departamento de Informática, por compartilhar seus conhecimentos durante esses anos de formação, em especial ao meu orientador Ricardo por todo apoio e orientação para conclusão deste trabalho. À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa. Agradeço a todos os colegas, do mestrado e da graduação, que me incentivaram, ajudaram e contribuíram direta ou indiretamente para eu chegar até aqui.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

MEIRELES FILHO, Waldir Denver Muniz, M.Sc., Universidade Federal de Viçosa, Março de 2014. **Virtualização e execução de algoritmos em FPGA: um algoritmo de *modulo scheduling* para arranjos reconfiguráveis de grão grosso.** Orientador: Ricardo dos Santos Ferreira. Coorientador: José Augusto Miranda Nacif.

Em muitas aplicações para sistemas embarcados, os laços mais internos dominam o tempo total de execução dos algorítimos. As arquiteturas reconfiguráveis de grão grosso (CGRA) alcançam um alto desempenho executando estes laços através de uma abordagem de *software pipelining*, mais especificamente através da técnica de *modulo scheduling*. Esta técnica melhora o desempenho fazendo a sobreposição de diversas iterações do laço em intervalos constantes. Os CGRA são energeticamente eficientes em termos de operações/watts. Por outro lado, as abordagens anteriores de *modulo scheduling* tem um tempo de execução elevado. Além disso os CGRAs e ferramentas comercias não estão disponíveis no mercado. Este trabalho propõe um CGRA virtual implementado como uma camada sobre um FPGA comercial. Além disso, apresentamos a primeira implementação de um algoritmo de *modulo scheduling* em *hardware*. O CGRA baseia-se em uma rede de interconexão global, através do uso de redes crossbar ou redes multiestágio implementadas em um FPGA comercial. As arquiteturas propostas foram comparadas com uma arquitetura CGRA 2D em malha, uma vez que a maioria dos trabalhos anteriores são baseados neste tipo de CGRA. Os resultados experimentais demonstram que a arquitetura com interconexão global proposta tem área/atraso comparável com a arquitetura 2D em malha. Além disso, uma vez que a abordagem global simplifica o algoritmo de escalonamento, apresentamos também a implementação do *modulo scheduling* em hardware em um FPGA. Esta implementação em hardware é cerca de 4× a 7× mais rápida do que uma abordagem em software-core executando em um FPGA. A área ocupada pelo módulo de escalonamento em hardware é menor que 10% da área ocupada pelo CGRA. Embora a implementação seja baseada em um algoritmo guloso, os resultados experimentais demonstram que o algoritmo pode mapear aplicações multimídia no CGRA virtual implementado no sobre um FPGA comercial. As arquiteturas

CGRA avaliadas possuem 16, 32 e 64 unidades de processamento de 32 bits. Por fim, mostramos que a abordagem de escalonamento proposta é rápida o suficiente para mapear o código de um laço dinamicamente para o acelerador CGRA com um rede global.

Abstract

MEIRELES FILHO, Waldir Denver Muniz, M.Sc., Universidade Federal de Viçosa, March 2014. **Virtualization and implementation of algorithms in FPGA: an modulo scheduling algorithm for coarse-grained reconfigurable architecture.** Adviser: Ricardo dos Santos Ferreira. Co-Adviser: José Augusto Miranda Nacif.

In most stream applications for embedded systems, the innermost loop codes dominate the total execution time. Coarse-grained reconfigurable architecture (CGRA) could achieve high performance to implement innermost loop codes by using software pipelining approach. More specifically, modulo scheduling which is a software pipelining approach are used. This technique improve the performance by overlapping the loop iteration execution. In additional, CGRA are energetically efficient in terms of operations/watts. However, the previous modulo scheduling approaches are time-consuming and commercial CGRA and tools are not available in the market. This work proposes to implement a Virtual CGRA as a layer in a commercial FPGA, and moreover, we present the first hardware-based implementation of a modulo scheduling algorithm. The CGRA is based on a global interconnection network. An interconnection design exploration is performed by using crossbar and multistage interconnection network in a commercial target FPGA. The proposed architectures are compared to a baseline 2D-mesh CGRA mesh, since most previous work are based on it. The experimental results demonstrate that the proposed global interconnection architecture has area/delay comparable to the 2D-mesh. In addition, since the global approach simplify the modulo scheduling algorithm, we present the first FPGA hardware-based implementation. The modulo scheduling hardware implementation is from 4x up to 7x faster than a software core approach. Moreover, the area overhead of our modulo scheduling hardware is less than 10% of the CGRA area. Although the implementation is based on a greedy modulo scheduling algorithm, the experimental results demonstrate that the algorithm could mapped a multimedia benchmark set on a target Virtual CGRA implemented on the top of a commercial FPGA. The CGRA evaluated architectures have 16, 32 and 64 execution units 32 bit width. Finally, we show that the proposed approach is

fast enough to be used in dynamic framework to map loop code to a target CGRA accelerator in run-time.

1. Introdução

A demanda por desempenho computacional vem crescendo a décadas. Para algumas aplicações, apenas utilizar hardwares genéricos (como os processadores de uso geral) e softwares não é suficiente, pois estes podem não satisfazer a demanda de desempenho necessária. Tradicionalmente usar um *hardware* para uma aplicação específica na forma de ASIC (*Application Specific Integrated Circuit*) tem sido adotado nos núcleos de computação intensiva para atender a demanda de desempenho com eficiência energética.

No entanto, a crescente convergência de diferentes funcionalidades combinada aos custos elevados envolvidos na fabricação de um ASIC, levaram os desenvolvedores a utilizar soluções programáveis. Dentre os hardwares reconfiguráveis, podemos destacar o Arranjo de Portas Programável por Campo, FPGA (*Field Programmable Gate Array*). Os FPGAs possuem os benefícios do alto desempenho de soluções feitas em hardware com a flexibilidade das soluções em software. Diferente dos ASICs, os FPGAs podem ser reprogramados depois de fabricados. Além disso, os FPGAs são dispositivos que permitem um alto grau de paralelismo, pois permitem a implementação de computação espacial e temporal na qual as operações podem ser simultaneamente executadas em várias unidades funcionais e podem ser sobrepostas no tempo com uso de pipeline.

Os FPGAs são arquiteturas de grão fino, nos quais as operações são executadas no nível de bits, o que lhe confere muita flexibilidade, mas que também implica em uma sobrecarga gerada pela memória de configuração. Ademais, a complexidade do posicionamento e do roteamento nos FPGAs é alta [Hartenstein, 2001a], que faz com que o tempo de reconfiguração seja elevado. Além disso, o preço a ser pago pela alta flexibilidade é o grande espaço de solução a ser explorado para mapear uma aplicação no FPGA. O mapeamento da solução pode demandar um longo tempo de compilação e síntese. Outro problema é a sobrecarga da área de roteamento, que é consequência do grande número de fios de interconexão chegando a cerca de 90% da área total [Hauck & DeHon, 2007].

Outro tipo de arquitetura reconfigurável são os CGRAs (*Coarse grained reconfigurable architectures*), que tentam superar as desvantagens das soluções de computação baseados em FPGA reduzindo a complexidade e os tempos de escalonamento, posicionamento e roteamento. Os CGRAs estão se tornando alternativas

atraentes porque oferecem uma capacidade de realizar operações com redução do custo/energia. Alguns exemplos de sistemas CGRAs que visam o processamento multimídia são ADRES [Mei et al., 2005b], MorphoSys [Lu et al., 1999] e Silicon Hive [Quax et al., 2004].

CGRAs geralmente são compostos por uma matriz com unidades funcionais (UFs) interligadas por uma rede de interconexão. As UFs podem executar operações a nível de palavra, incluindo adição, subtração e multiplicação. Em contraste com FPGAs, os CGRAs sacrificam a flexibilidade para aumentar a eficiência de hardware. Como resultado eles têm baixo tempo de reconfiguração, maior velocidade de execução e baixo consumo de energia.

A eficiência energética tornou-se uma das métricas de projeto mais importantes em muitos domínios computacionais. A computação de alto desempenho é extremamente limitada pela potência e pelos fatores térmicos de tal forma que um maior desempenho só é possível através do aumento da eficiência energética [Shrivastava et al., 2011]. Além disso, a eficiência energética é a métrica mais importante para determinar a usabilidade em dispositivos eletrônicos portáteis, tais como telefones celulares, tabletes, reprodutores de MP3, etc. O CGRA é uma solução promissora para o processamento intensivo que exige baixo consumo de energia, pois um CGRA pode ser visto como um processador com várias unidades que executam em paralelo.

A configuração de um CGRA apresenta alguns desafios e um compilador eficaz é essencial para explorar os recursos de computação disponíveis em um CGRA. Para fazer uso eficiente dos recursos do CGRA, o *modulo scheduling*, que é uma técnica de *software pipelining*, pode ser usado [Rau, 1994]. Esta técnica proporciona a oportunidade de explorar o paralelismo de instruções presente nos laços internos de computação intensiva das aplicações, utilizando de forma mais eficiente os recursos de um CGRA.

Para lidar com o problema de roteamento, o compilador DRESC (*Dynamically Reconfigurable Embedded System Compiler*) [Mei et al., 2002] propõe um algoritmo de *modulo scheduling* que começa com um posicionamento aleatório das operações nas UFs, que pode ou não resultar em um escalonamento válido. As operações são em seguida alternadas entre as UFs até que um escalonamento válido seja alcançado com a meta heurística *Simulated Annealing* (SA). O compilador DRESC alcança bons resultados, próximos da solução ótima, no entanto, a técnica de posicionamento com a SA pode resultar em um longo tempo de compilação.

Abordagens posteriores para o problema buscaram reduzir o tempo de compilação, porém os melhores resultados ainda gastam em média de 1 a 100 segundos para escalonar os laços [Yoon et al., 2008; Chen & Mitra, 2012; Hamzeh et al., 2012].

Para laços mais complexos, as soluções propostas na literatura [Yoon et al., 2008; Chen & Mitra, 2012; Hamzeh et al., 2012] podem gastar tempos superiores a 30 minutos.

A reconfiguração e compilação dinâmica são importantes para garantir a portabilidade em sistemas embarcados [Ferreira et al., 2011a]. Portanto, propor uma arquitetura CGRA aliada a uma heurística de escalonamento que seja simples para ser usada em tempo de execução é um desafio onde a simplificação nos passos de posicionamento e roteamento dos CGRAs podem ser cruciais para que a reconfiguração dinâmica do hardware em tempo de execução seja alcançada.

1.1 Motivação

Um CGRA gasta pouca energia na execução de uma operação. CGRAs podem atingir a eficiência energética de 10-100 GOps/W [Singh et al., 2000] que é cerca de 2 vezes maior que a eficiência de processador Intel Core i7.

Em aplicações multimídia normalmente existem muitos laços que dominam o tempo de execução. Os CGRAs possuem muitos elementos de processamento que podem acelerar estes laços utilizando técnicas *software pipeline* (SWP), sobrepondo as execuções de diferentes iterações.

Park [Park et al., 2009] analisou três aplicações multimídia importantes (*3D graphics rendering*, *ACC decoder* e *H.264 decoder*) e detectou a quantidade de laços que podem ser paralelizados com *software pipelining* (SWP). Os resultados são mostrados na Tabela 1.1.

Algoritmo	Total de laços	laços SWP
3D	102	36
AAC	260	83
H.264	269	81

Tabela 1.1. Total de laços e laços paralelizáveis com *software pipelining*. Resultados extraídos de [Park et al., 2009].

Os estudos de [Park et al., 2009], também mediram a parcela do tempo de execução gasta nos laços SWP e no restante do código. Na Figura 1.1, cada barra representa a distribuição do tempo de execução gasto nas regiões SWP e no resto da aplicação. A barra à esquerda de cada algoritmo representa a parcela de tempo de execução gasto quando apenas um processador VLIW (*Very Long Instruction Word*) é usado para toda a aplicação, sem usar *software pipelining*. A barra da

direita mostra a nova parcela de tempo se as regiões SWP forem executadas com *pipelining* pelo CGRA.

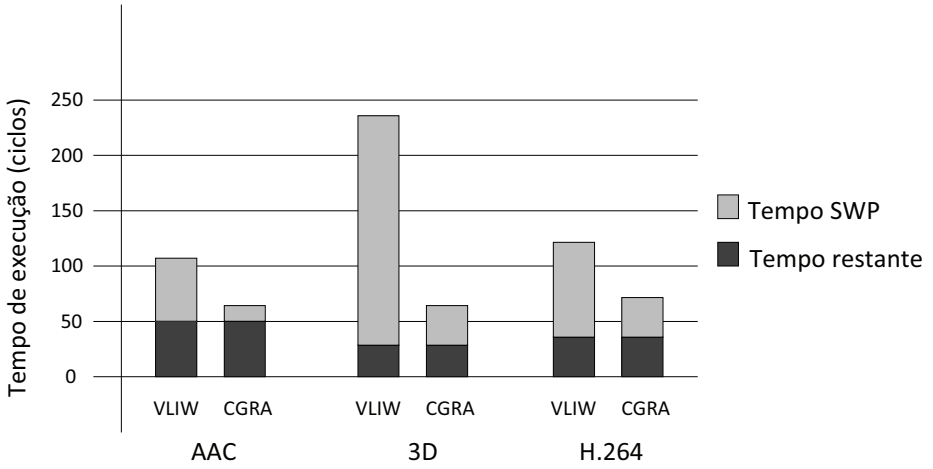


Figura 1.1. Tempo de execução para laços paralelizáveis com *software pipelining* e o restante do *software*. Resultados extraídos de [Park et al., 2009].

A Tabela 1.1 e a Figura 1.1 mostram que há muitas oportunidades para explorar o paralelismo. Nos *benchmarks* analisados, em média 35% dos laços são paralelizáveis com *software pipelining* e 71% do tempo de execução é gasto em regiões SWP. Quando o CGRA é usado para mapear os laços SWP, obtiveram-se ganhos de $1,76\times$, $3,25\times$ e $1,48\times$ para o AAC, 3D e H.264, respectivamente. Portanto, o paralelismo das aplicações multimídia pode ser explorado com CGRAs, que são plataformas eficazes neste contexto.

Como já mencionado, o *software pipelining* nos CGRAs utiliza a técnica de *modulo scheduling* [Rau, 1994] e muitos algoritmos foram propostos [Park et al., 2006, 2009; Hatanaka & Bagherzadeh, 2007; Mei et al., 2003; Oh et al., 2009]. No entanto, o tempo de compilação destes algoritmos ainda é elevado para um cenário dinâmico em tempo de execução.

A flexibilidade da arquitetura pode ser aumentada caso o algoritmo possa escalonar tarefas em tempo de execução e a reconfiguração dinâmica pode permitir que o CGRA execute multitarefas. Um suporte para multitarefas em CGRAs não só aumentará a utilização dos recursos do CGRA, mas também irá melhorar o desempenho de comunicação entre as tarefas e ajudar aliviar os gargalos de memória [Shrivastava et al., 2011].

No entanto, esta é uma tarefa difícil devido ao tempo de compilação das abordagens onde se aplicam técnicas como as de [Mei et al., 2002]. O principal requisito para a execução multitarefas em CGRAs é a capacidade de remapear os núcleos de

computação dinamicamente e em tempo de execução. Isto demanda que o processo de configuração seja rápido, de tal forma que o benefício de desempenho obtido pela execução multitarefa não seja mascarado pelo tempo de configuração envolvido.

Várias abordagens vêm sendo propostas para reduzir o tempo, porém, ao se propor uma solução deve se considerar a complexidade do algoritmo de mapeamento em função dos recursos de interconexão para prover reconfiguração em tempo de execução.

Os trabalhos anteriores [Lopes, 2013; Mucida, 2013] são extensões da proposta inicial apresentada em [Ferreira et al., 2011a]. O objetivo principal é a redução do tempo de compilação. Esta dissertação complementa estes trabalhos ao implementar em FPGA as arquiteturas apresentadas [Lopes, 2013] e [Mucida, 2013]. Os resultados gerados em conjunto com as duas dissertações supracitadas foram publicadas em [Filho & Ferreira, 2012; Mucida et al., 2012; Ferreira et al., 2013].

1.2 Objetivos

O objetivo deste trabalho é implementar uma versão do algoritmo *Modulo Scheduling* [Lopes, 2013; Mucida, 2013] em hardware para otimizar a execução de laços de computação intensiva de aplicações multimídia, fazendo o posicionamento e o roteamento em uma arquitetura CGRA. Além disso, diversas alternativas de arquiteturas CGRAs virtuais serão exploradas e mapeadas em FPGA, comparando a área em hardware, tamanho da memória de configuração, atraso e flexibilidade de mapeamento, de modo viabilizar o uso da solução em circuitos comerciais.

1.3 Organização do texto

Esta dissertação está organizada em capítulos, como descrito a seguir:

- O capítulo 1 introduz a motivação e os objetivos do trabalho.

- O capítulo 2 apresenta uma revisão bibliográfica das arquiteturas reconfiguráveis de grão fino (FPGA) e de grão grosso (CGRA), abordando questões sobre virtualização de CGRAs em FPGAs. Também são apresentadas seções sobre *software pipelining* e *Modulo Scheduling*.

- O capítulo 3 apresenta variações das arquiteturas e seus componentes exploradas nesta dissertação. Uma simplificação da arquitetura ADRES, desenvolvida para comparação, também é apresentado neste capítulo.

- O capítulo 4 mostra a implementação em *hardware* dos algoritmos de *modulo scheduling* propostos em [Mucida, 2013; Mucida et al., 2012]. Dois algoritmos são apresentados. O primeiro é mais simples, onde o CGRA que possui um número de unidades maior que a quantidade de nós do laço. O segundo, que faz uso de partições temporais, realizando mais de uma configuração para o CGRA, de modo que este seja capaz de executar um laço com mais operações que a quantidade de unidades de processamento disponíveis.

- O capítulo 5 apresenta a avaliação da solução proposta. Foram realizados experimentos que avaliaram o tempo de execução, o tamanho das arquiteturas, desempenho e a ocupação da implementação do algoritmo *Modulo Scheduling* em *hardware*.

- O capítulo 6 apresenta a conclusão do estudo, propondo algumas sugestões de melhorias para as soluções propostas na dissertação, para que possam ser aplicadas em trabalhos futuros.

2. Revisão bibliográfica

Neste capítulo é apresentada a revisão bibliográfica desta dissertação. Primeiramente serão apresentadas as arquiteturas reconfiguráveis CGRA e FPGA, listando suas características e os componentes que compõem cada uma. Em seguida é feito um resumo sobre virtualização de arquiteturas especializadas sobre FPGAs comerciais. As seções seguintes tratam das técnicas de *software pipelining* e algoritmos de *Modulo Scheduling* para configuração de arquiteturas reconfiguráveis, que são o foco principal desta dissertação.

2.1 Arquiteturas reconfiguráveis

Hoje em dia muitas aplicações, principalmente em telecomunicações e multimídia, exigem multitarefa e um processamento de fluxo de dados eficiente. Além disso, sistemas com diferentes capacidades de processamento exigem eficiência em termos de desempenho e potência.

A computação reconfigurável surge como alternativa para atingir parcialmente o desempenho dos circuitos de aplicação específica *(Application Specific Integrated Circuits* ou ASIC) mantendo a flexibilidade dos processadores de propósito geral. O surgimento do FPGA (*Field Programmable Gate Array*) em meados dos anos 80 abriu caminho para a implementação da computação reconfigurável.

Um FPGA tem uma granularidade de reconfiguração muito pequena que lhe confere grande flexibilidade, porém com uma grande complexidade de mapeamento das aplicações para arquitetura. Uma alternativa é aumentar a granularidade das células base para unidades lógica aritméticas que implementam operações a nível de palavra de 16 ou 32 bits. As arquiteturas reconfiguráveis de grão grosso (*Coarse Grain Reconfigurable Architecture* ou CGRA) implementam operações a nível de palavra e vêm se destacando nos últimos anos, pois podem aliar o desempenho dos ASICs com a flexibilidade dos processadores de finalidade geral, mostrado que podem fornecer tanto a eficiência energética como aceleração em hardware [Hamzeh et al., 2012].

A seguir é apresentado um resumo sobre as arquiteturas FPGA e CGRA, listando as características e os componentes de cada uma.

2.2 FPGA

O FPGA é um tipo de circuito integrado (CI) que pode ser programado para diferentes algoritmos após sua fabricação. Dispositivos FPGA modernos podem conter milhões de células lógicas que são configuradas para implementar uma infinidade de algoritmos. Embora o fluxo de projeto de um FPGA tradicional seja mais parecido com de um circuito integrado simples do que de um processador, um FPGA oferece muitas vantagens em comparação com os CIs, oferecendo o mesmo nível de desempenho em vários casos. A vantagem do FPGA, quando comparado ao CI é a sua capacidade de ser reconfigurado. Este processo, que é o mesmo que o carregamento de um programa em um processador, pode envolver todos os recursos de um FPGA ou parte dos recursos disponíveis.

O trabalho apresentado nesta dissertação utiliza a ferramenta de síntese para FPGA da Xilinx para família Virtex 6 [Xilinx, 2010]. Neste capítulo apresentamos um resumo da arquitetura FPGA e seus principais componentes, tomando como referência a arquitetura utilizada na família Virtex 6 da Xilinx .

2.2.1 Arquitetura de um FPGA

A estrutura básica de um FPGA é composta pelos seguintes elementos:

- ***Look-up table* (LUT)**: Elemento que executa as operações lógicas;
- **Flip-Flop (FF)**: Elemento que armazena o resultado da LUT;
- **Fios**: Elementos utilizados para fazer a interconexão entre os demais elementos;
- **blocos de Entradas/saídas**: Portas fisicamente disponíveis para troca de informações do FPGA com circuitos externos;
- **Memória de configuração**: Configuração das operações realizadas pelas LUTs, configuração da rede de roteamento e outras funcionalidades.

A combinação destes elementos resulta na arquitetura básica de um FPGA, mostrada na Figura 2.1. Os CLBs (*Configuration Logical Blocks*) são circuitos idênticos entre si, construídos por *flip-flops* e lógica combinacional utilizando LUTs. Embora esta estrutura seja suficiente para a implementação de vários algoritmos, a eficiência de execução resultante é limitada em termos da máxima frequência.

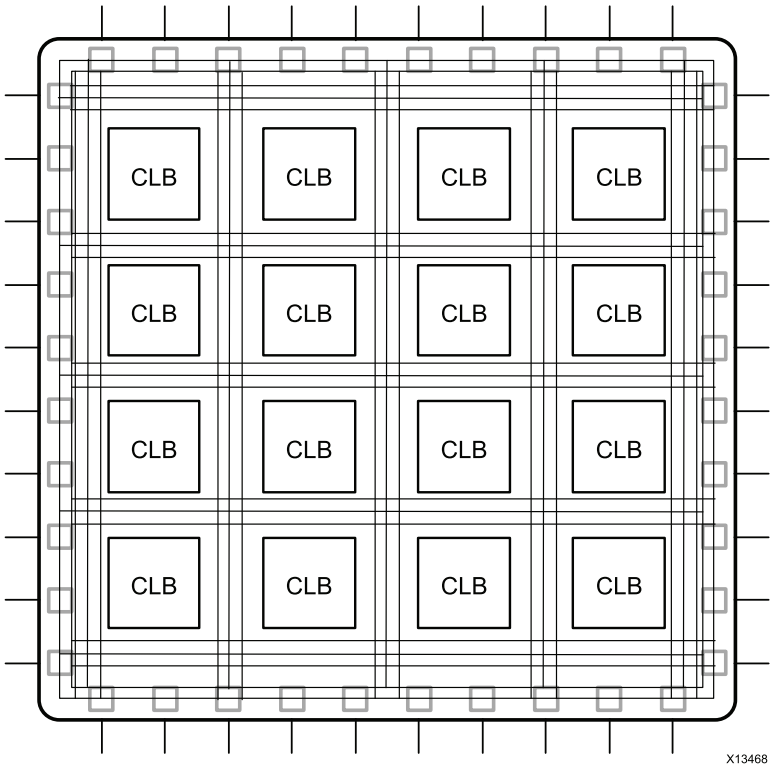


Figura 2.1. Arquitetura básica de um FPGA

As arquiteturas FPGA atuais, como a Virtex 6 [Xilinx, 2010], incorporam aos elementos básicos já citados, blocos de processamento adicionais e blocos de armazenamento de dados, aumentando a capacidade computacional e a eficiência do dispositivo. Estes elementos adicionais podem ser:

- Memórias embarcadas para armazenamento de dados distribuídos;
- *Phase-locked loops*(PLLs) para oferecer diferentes frequências de relógio.
- Transreceptores seriais de alta velocidade para interface entrada/saída.
- Controladores de memória externa;
- Blocos multiplicadores/acumuladores;

A combinação destes elementos proporciona ao FPGA maior flexibilidade e eficiência para várias aplicações que demandam multiplicações com DSPs e/ou memória distribuída. O FPGA resultante da combinação destes elementos é ilustrado na Figura 2.2.

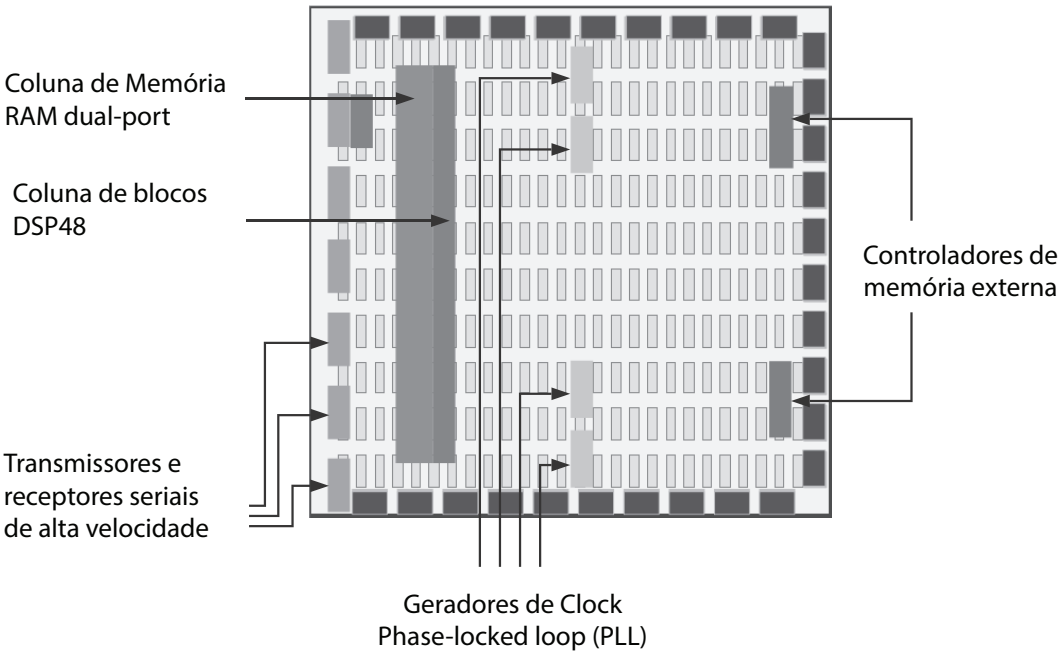


Figura 2.2. Atual arquitetura de um FPGA

2.2.1.1 LUT

A LUT é o bloco básico de construção de um FPGA, sendo capaz de implementar qualquer função lógica booleana de N variáveis. Essencialmente, este elemento é uma tabela verdade, na qual diferentes combinações de entradas implicam em diferentes valores de saída. Para uma LUT com N entradas[1], teremos 2^N posições de memória acessíveis na LUT que permitem que ela implemente 2^{N^N} funções.

A implementação de uma LUT em hardware pode ser imaginada como uma coleção de células de memória de 1 bit ligadas a um conjunto de multiplexadores. A Figura 2.3 mostra esta representação da LUT. Desta forma, as entradas da LUT são os bits seletores dos multiplexadores e a saída o valor resultante da célula de memória selecionada. Esta estrutura permite que uma LUT funcione tanto como uma estrutura que retorna o resultado de uma função quanto um elemento de armazenamento de dados.

2.2.1.2 Flip-Flop

O flip-flop é a unidade básica de armazenamento de uma arquitetura FPGA. Este elemento é sempre emparelhado à uma LUT para implementar lógica com *pipeline* e armazenamento de dados. A estrutura básica de um flip-flop inclui uma entrada

[1]O valor típico de N para um FPGA Xilinx da família Virtex é 6

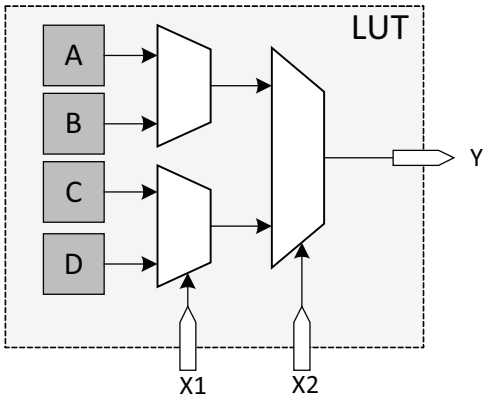


Figura 2.3. Representação funcional de uma LUT como uma coleção de células de memória

de dados, entrada de relógio, sinal de habilitação, *reset*, e saída de dados. Durante seu funcionamento, o valor na porta de entrada é retido e passado para a saída a cada pulso do relógio.

2.2.1.3 Blocos DSP

O bloco de cálculo mais complexos disponível em um FPGA Virtex 6 Xilinx é o bloco DSP48, o qual é mostrado na Figura 2.4. O bloco DSP48 é uma unidade lógica aritmética (ULA) incorporada na arquitetura do FPGA, composto por uma cadeia de três blocos diferentes. Conforme ilustra a Figura 2.4, a cadeia computacional do DSP48 é composto de uma unidade de adição/subtração ligada a um multiplicador e em seguida ligado a um somador/subtrador/acumulador. Esta cadeia permite que uma única unidade DSP48 implemente funções da seguinte forma:

$$p = a \times (b + d) + c \tag{2.1}$$

ou

$$p = p + a \times (b + d) \tag{2.2}$$

2.2.1.4 BRAM e outras memórias

Na arquitetura do FPGA também estão inclusos elementos de memória que podem ser usados como memórias de acesso aleatório (RAM), memórias só de leitura (ROM), ou registros de deslocamento. Estes elementos são blocos RAMs (BRAMS), LUTs e registradores de deslocamento.

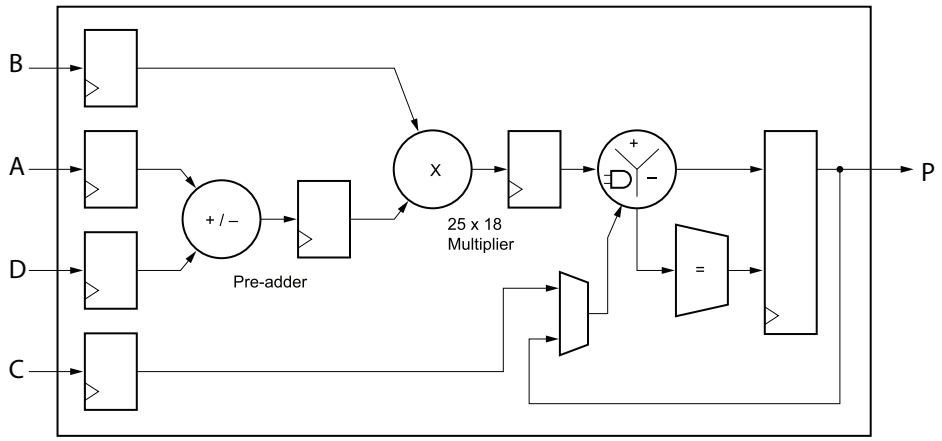


Figura 2.4. Estrutura de um bloco de DSP48

Uma BRAM é um módulo de memória RAM *dual-port*[2] existente na arquitetura interna do FPGA Virtex 6 para fornecer armazenamento a um conjunto relativamente grande de dados. Os dois tipos de memórias BRAM disponíveis em um dispositivo pode conter 18k ou 36k bits e o número destas memórias disponíveis é específica para cada dispositivo FPGA.

Como discutido anteriormente, a LUT é uma pequena memória na qual os valores de uma tabela verdade são gravados durante a configuração do dispositivo. Devido à flexibilidade de uma LUT, estas podem ser usadas como memórias e são comumente referidas como memórias distribuídas. Este é o tipo mais rápido de memória disponível no dispositivo FPGA, porque pode ser instanciada em qualquer parte do FPGA melhorando o desempenho do circuito implementado.

2.3 CGRA

Uma arquitetura reconfigurável de grão grosso(CGRA)[3], assim como um FPGA possui componentes lógicos que podem ser configurados e que estão conectados uns aos outros por uma rede de interconexão.

O CGRA surgiu com objetivo principal de reduzir a complexidade e o tempo de configuração exigido por um FPGA, fornecendo uma largura de bits mais compatível com a aplicação. Para isso, as operações lógica de um CGRA são executada por unidades de processamento (UP), que executam operações mais complexas, se comparadas com as unidades lógicas de processamento dos FPGAs. Então, enquanto

[2]Permite acesso simultâneo em dois locais diferentes da memória
[3]do inglês *Coarse-Grained Reconfigurable Arrays*

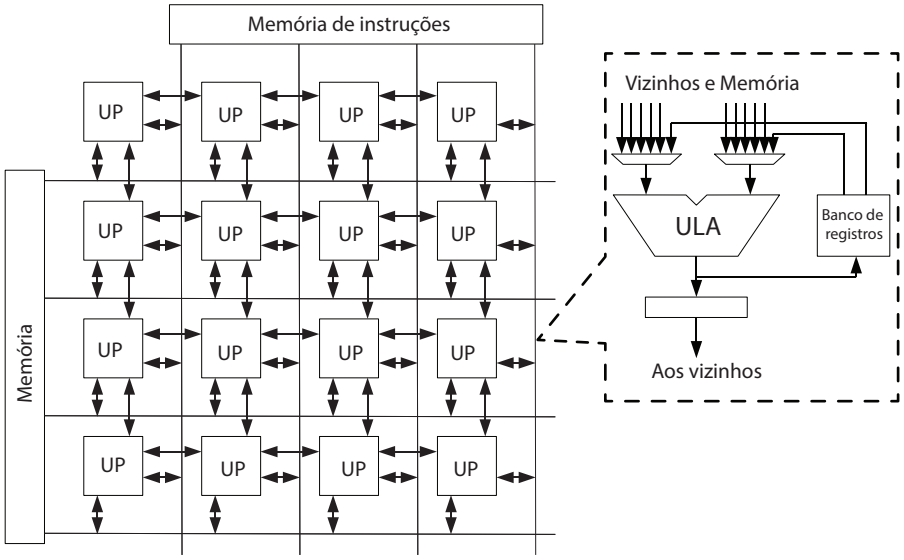


Figura 2.5. Exemplo de uma arquitetura CGRA 4x4, com as UPs conectadas em malha. Cada UP é constituída de dois multiplexadores, um registrador, uma ULA e de um banco de registradores.

os FPGAs operam no nível de um único bit, os CGRAS operam a nível de palavras [Hartenstein, 2001b].

Um CGRA é composto basicamente pelos seguintes elementos:

- **Unidade de processamento**: elemento que executa as operações lógicas e aritméticas;
- **Rede de Interconexão**: elemento utilizado para fazer a interconexão entre as Unidades de processamento;
- **Memória de configuração**: para a configuração das operação realizadas pelas UPs e o roteamento pela rede de interconexão;

A Figura 2.5 ilustra um típico CGRA, detalhando a constituição interna de sua unidade de processamento. Note que neste CGRA, cada UP é constituída de dois multiplexadores, um registrador, uma ULA e de um banco de registradores.

Os CGRAs se diferenciam principalmente pelo tipo de rede e tipo de Unidade de Processamento e cada uma destas característica influenciará no custo de implementação do circuito, no algoritmo de mapeamento e na capacidade de processamento.

Em um CGRA, as Unidades de Processamento podem ser homogêneas ou heterogêneas. Quando são homogêneas, todas as UPs são iguais e realizam qualquer

operação de um determinado conjunto de instruções. Quando são heterogêneos, as UPs realizam operações específicas, onde algumas UPs podem, por exemplo, realizar soma de dois números e outras realizam *load/store* ou qualquer outro tipo de operação lógica ou aritmética.

Um CGRA se diferencia também pela sua granularidade, que significa o número de bits transportado pela rede de interconexão e manipulados pelas UPs. Quanto menor a granularidade de uma arquitetura reconfigurável, mais flexível ela será, porém uma granularidade maior reduz a complexidade e diminui a memória de configuração.

As UPs de um CGRA podem estar conectadas de várias formas e algumas das conexões mais comuns estão ilustradas na Figura 2.6.

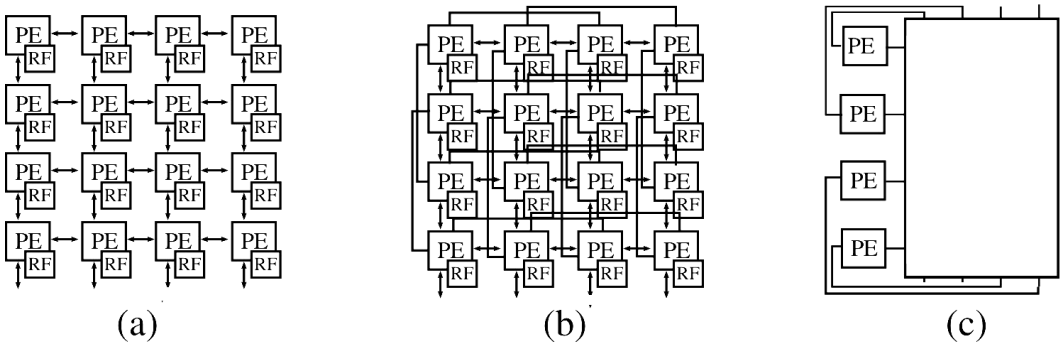


Figura 2.6. Exemplo de conectividade entre as UPs de um CGRA. (a) Rede em malha. (b) Malha plus, (c) Rede Global

Na Figura 2.6(a) é mostrado um CGRA 4x4 no qual cada UP se conecta diretamente com seus 4 vizinhos. Além disso, cada UP também tem um banco de registros para armazenar valores temporários. A Figura 2.6(b) mostra uma interconexão em malha melhorada, que adiciona novas conexões entre as UPs. Isso melhora a capacidade de roteamento e simplifica o algoritmo de posicionamento e roteamento. A Figura 2.6(c) mostra um CGRA com conexão completa entre as unidades de processamento. Esta interconexão global simplifica o trabalho de roteamento, pois todas as UPs podem se conectar diretamente e simultaneamente a qualquer outra, sem que haja restrição no posicionamento das operação nestas unidades de processamento.

As redes de interconexão tem impacto direto no custo da implementação física, na frequência de operação e na complexidade dos algoritmos de mapeamento. Nesta dissertação é apresentada no capítulo 3 algumas implementações baseadas no CGRA proposto por [Ferreira et al., 2011a], que utilizam uma rede de interconexão global Multiestágio como mostrado na Figura 2.6(c). A rede global foi implementada com uma rede multiestágio em [Ferreira et al., 2011a]. Nesta dissertação, além das

redes multiestágio, também foram utilizadas as redes *crossbar* como elemento de interconexão nos CGRAs avaliados.

2.3.1 Rede Multiestágio

As redes de conexão multiestágio (MINs)[4] tem como objetivo conectar um dispositivo de entrada a um dispositivo de saída através de estágios de elementos de chaveamento. A capacidade de roteamento de uma MIN é determinada pelo número de estágios e pelo padrão de ligação entre os estágios. Em uma rede Multiestágio cada saída pode ser alcançada a partir de uma entrada, porém em redes multiestagios com menos de $2log_n$ estágios, podem acontecer conflitos de roteamento quando várias entradas e saídas são conectadas simultaneamente.

A Figura 2.7 mostra um exemplo de uma rede Multiestágio com 8 entradas/saídas e log_n estágios (onde podem ocorrer conflitos), na qual existe um roteamento da entrada 0 à saída 4. Note que nesta rede existe apenas um caminho possível para tal rota.

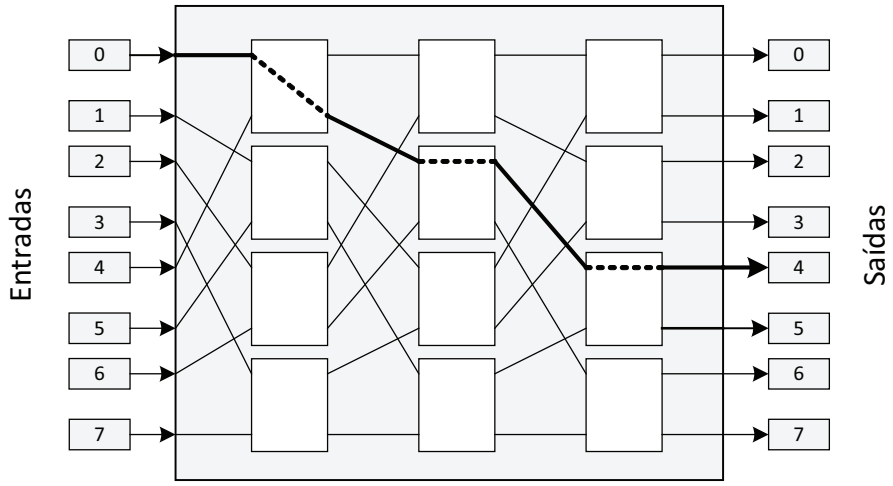


Figura 2.7. Rede multiestágio ômega 8x8. Conexão da entrada 0 à saída 4.

Como dito, algumas redes multiestágios não permitem que qualquer conjunto de conexões entre entradas e saídas seja estabelecido. Portanto, podem ocorrer conflitos de roteamento para um subconjunto de permutações. A Figura 2.8 mostra uma tentativa de dois roteamentos simultâneos, entre $0 \rightarrow 4$ e $6 \rightarrow 5$, porém um conflito é gerado no segundo comutador da segunda coluna. A primeira saída do comutador já está sendo utilizada pelo roteamento $0 \rightarrow 4$ e por isso a ligação $6 \rightarrow 5$ não pode ser realizada.

[4]do inglês *Multistage Interconnection Networks*

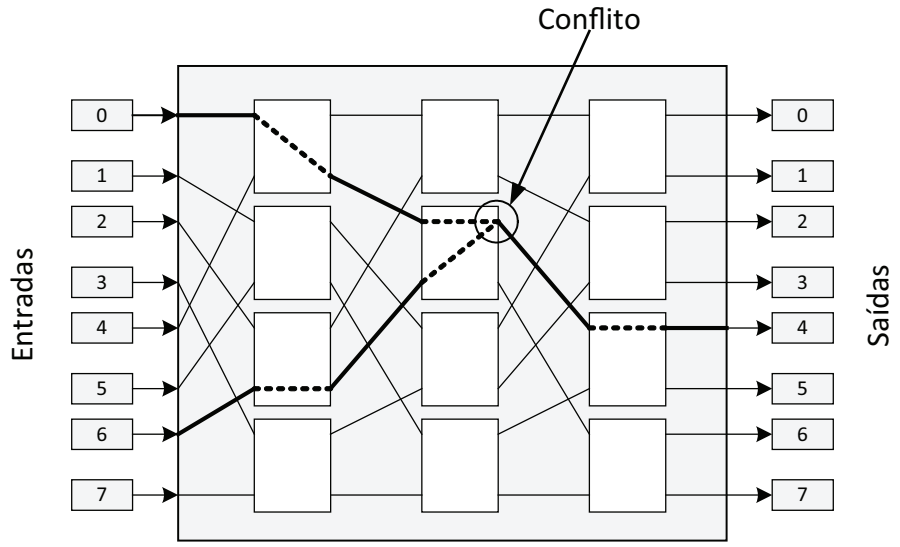


Figura 2.8. Rede multiestágio com conflito. O roteamento $0 \rightarrow 4$ e $6 \rightarrow 5$ não foi possível

Nas redes multiestágio, estágios extras são uma alternativa para melhorar o roteamento. Cada estágio extra insere uma coluna de comutadores na rede e aumenta em 2 vezes o roteamento entre os endereços de entrada e saída [Ferreira et al., 2011a]. Isso acontece porque para cada par de entradas e saídas, o número de caminhos é dado por R^x, onde R é o RADIX do comutador utilizado e x é o número de estágios extras. A Figura 2.9 mostra como um estágio extra resolve o problema de conflito no roteamento anterior. Porém, o atraso irá aumentar com o acréscimo de mais estágios. Além disso, o número de bits de configuração e o calculo do roteamento também aumentam.

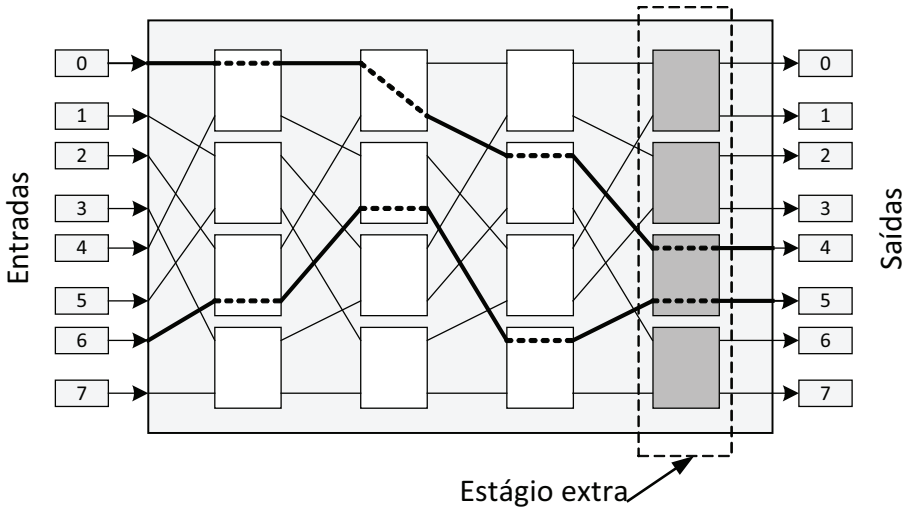


Figura 2.9. Estágio extra para resolver conflito de roteamento.

O comutador da rede pode ser simples, como um comutador 2x2 (Figura 2.7) ou mais complexo. Quanto maior o comutador, menor será o número de estágios. Para comutadores 2x2 ou radix-2, a rede terá no mínimo Lg_2N estágios, porém em uma rede com comutadores 4x4 ou radix-4, o número de estágios será Lg_4N. Ou seja, existe um custo benefício entre a complexidade do comutador e o número de estágios, pois o número reduz, mas a complexidade do comutador aumenta.

Nesta dissertação foram utilizadas redes multiestágios com comutadores RADIX 4, como mostrado na Figura 2.10. Os comutadores radix-4 se adaptam melhor no FPGA Virtex 6, com LUT de 6 entradas [Vendramini & Ferreira, 2010; Ferreira et al., 2011b]. Note que, para as 16 entradas e saídas da rede, são necessários apenas 2 colunas com 4 comutadores em cada uma delas.

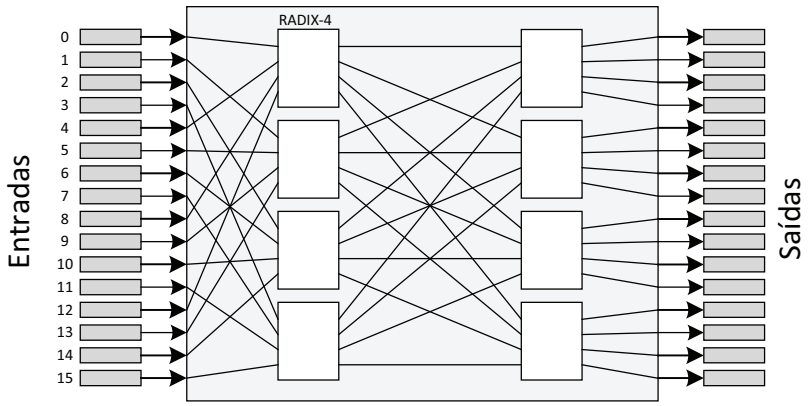


Figura 2.10. Rede Multiestágio com comutadores Radix 4

2.3.2 Rede Crossbar

Enquanto as redes multiestágio com menos de $2log_n$ só podem realizar o roteamento para um subconjunto de permutações, as redes crossbar são capazes de realizar qualquer permutação de entrada/saída, porém seu custo em comutadores é $O(N^2)$ para N entradas/saídas.

A Figura 2.11 mostra um exemplo de uma rede de interconexão Crossbar que possui N^2 comutadores e realiza dois roteamentos simultâneos ($5 \rightarrow 3$ e $2 \rightarrow 2$).

2.4 CGRA e Virtualização

Os FPGAs têm demonstrado um significativo desempenho e vantagens energéticas em comparação com microprocessadores [DeHon, 2000]. Contudo, apesar das vantagens da execução de aplicações em FPGAs, o uso do FPGA é limitado devido ao

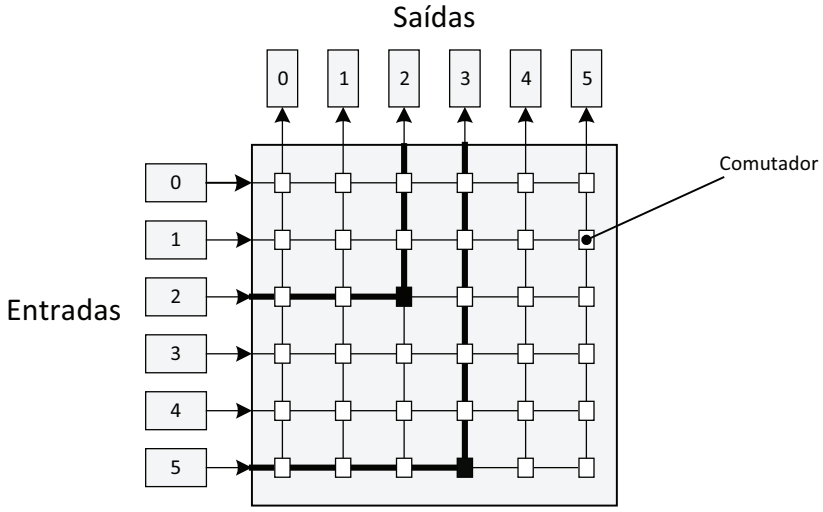


Figura 2.11. Rede Crossbar de 6 entradas/saídas com 6^2 comutadores. Roteamento da entrada 5 à saída 3 e roteamento da entrada 2 à saída 2.

aumento da complexidade do projeto em comparação com os projetos de softwares, devido principalmente aos desafios do projeto em termos de fluxo de sinais (RTL)[5].

Vários estudos têm aumentado o nível de abstração com ferramentas de síntese de alto nível (HLS)[6]. Mas embora as ferramentas de HLS possam melhorar a produtividade, o uso do FPGA é ainda, em grande parte, limitado a especialistas de hardware. Estudos recentes identificaram outros gargalos de produtividade importantes que incluem longas iterações de projeto, portabilidade limitada e reutilização de projetos. O aumento do tempo de execução do posicionamento e roteamento (P&R) é um problema emergente que pode exigir horas ou dias para grandes circuitos. Por isso, o tempo de execução do P&R nos FPGAs também representa um gargalo significativo no tempo de projeto, o que consequentemente dificulta a depuração e a verificação, reduzindo a produtividade e aumentando os custos.

Alguns dos problemas apresentados no uso dos FPGAs não acontecem em CGRAs. No entanto, embora diversos CGRAs tenham sido propostos durante as últimas décadas, poucos deles são de código aberto ou são comerciais ou até mesmo chegaram a ser implementados fisicamente [Baumgarte et al., 2003].

Uma possibilidade é criação de uma camada virtual sintetizada sobre um FPGA. Em outras palavras é uma arquitetura implementada diretamente no FPGA que possui um padrão de configuração diferente do utilizado no FPGA. Desta forma a nova arquitetura poderá ser reconfigurada em tempo de execução a partir de uma configuração específica, sem que seja preciso que uma nova aplicação passe por todo

[5]do inglês *Register Transfer Level*
[6]do inglês *high-level synthesis*

o processo de síntese, mapeamento e roteamento que uma aplicação convencional necessitaria para executar no FPGA. Por isso, com intenção de promover os CGRAs, surgiram propostas de virtualizações de CGRAs em FPGAs [Coole & Stitt, 2010; Brant & Lemieux, 2012].

Arquiteturas como [Coole & Stitt, 2010; Ferreira et al., 2011a] e outras abordagens semelhantes [Eguro & Hauck, 2006; Fung et al., 2004] diminuíram significativamente o tempo de P&R em FPGAs comerciais através da virtualização de uma arquitetura especializada. As principais vantagens de se virtualizar um CGRA, é a compatibilidade dos *bitstreams* entre os diferentes fornecedores de FPGA e um menor tempo de configuração. Porém, a virtualização diminui um pouco a flexibilidade do FPGA além de limitar a área e a frequência de operação causada pela implementação de recursos de roteamento da arquitetura virtual sobre o FPGA.

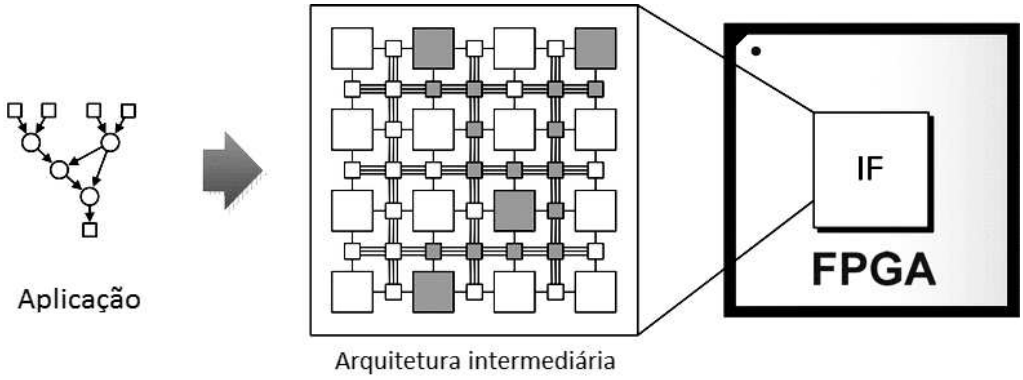


Figura 2.12. Arquitetura intermediária (Intermediate fabrics (IF)) virtual implementada em um FPGAs, que permite maior portabilidade e menor tempo de posicionamento e roteamento

Em 2010, foi proposto por [Coole & Stitt, 2010] um CGRA virtual sobre um FPGA comercial (veja Figura 2.12). Esta arquitetura é baseada no modelo de ilha, similar a arquitetura de um FPGA. Os resultados experimentais mostraram um aumento de velocidade do P&R em até três vezes em comparação com as ferramentas de síntese dos FPGAs comerciais. No entanto, nessa abordagem o escalonamento era feito manualmente.

Apesar de alguns resultados significativos, as estratégias usadas para CGRAs virtuais nunca foram usadas para permitir uma reconfiguração dinâmica em tempo de execução, isso porque os algoritmos de *modulo scheduling* anteriores não são adequados para implementação em hardware.

Recentemente foi proposta por [Ferreira et al., 2011a] uma arquitetura CGRA virtual implementada sob um FPGA e um algoritmo de mapeamento. O objetivo foi

produzir um algoritmo com tempo de execução baixo que pudesse ser utilizado por ferramentas dinâmicas para hardware reconfigurável, como compiladores *Just-In-Time* ou reconfiguração em tempo de execução. Essa arquitetura tem um conjunto de UPs conectadas a uma rede global multiestágio que permite que uma UP seja conectado a qualquer outra, simplificando o processo de posicionamento e roteamento.

2.5 Configuração de Arquiteturas Reconfiguráveis

A computação reconfigurável pode ser dividida em dois modos de configuração: estática e dinâmica [Sanchez et al., 1999]. Na configuração estática a arquitetura é configurada e se mantem inalterada durante a execução da aplicação. Isso permite que a aplicação atinja um alto desempenho, porém não provê grande flexibilidade. Na configuração dinâmica mais de uma configuração pode ser utilizada ao longo da execução da aplicação. Isso possibilita que vários segmentos da aplicação sejam mapeados na arquitetura. Configurações estáticas ou dinâmicas podem ser geradas por um compilador de maneira estática ou durante a execução. Estes compiladores são conhecidos na literatura como *Just-In-Time* - JIT - ou *on-the-fly compilation*.

Uma arquitetura que é dinamicamente reconfigurável pode ter sua configuração gerada estaticamente. O fato de esta arquitetura ser dinamicamente reconfigurável não significa que sua configuração seja gerada durante a execução. Porém, compilar dinamicamente proporciona a possibilidade de usar informação que só estarão disponíveis em tempo de execução.

2.6 Software Pipeline

O paralelismo temporal, ou *pipeline*, implica na execução de eventos sobrepostos no tempo. Uma determinada tarefa é subdividida em uma sequência de sub-tarefas, e cada uma delas é executada por um estágio especializado. *Software Pipeline* é uma técnica que permite aumentar o nível de paralelismo da implementação em hardware de um algoritmo preservando as dependências de dados. Várias iterações são executadas em paralelo, no mesmo ciclo de relógio. A única diferença é a fonte de dados para cada operação. Cada etapa do cálculo recebe os seus valores do resultado calculado do ciclo anterior. Por exemplo, para calcular a seguinte função teremos um multiplicador e dois somadores:

$$y = (a \times b) + c + d \tag{2.3}$$

A Figura 2.13 mostra o resultado da estrutura de computação para a Equação 2.3 e os efeitos do pipeline. A implementação da esquerda é o caminho de dados necessários para calcular o resultado sem pipeline. Esta implementação funciona da mesma forma que uma função correspondente em C/C++, onde todos os valores de entrada devem ser conhecidos no início do cálculo, no qual apenas um único resultado de y podem ser calculados de cada vez. A implementação à direita mostra a versão em pipeline do mesmo circuito.

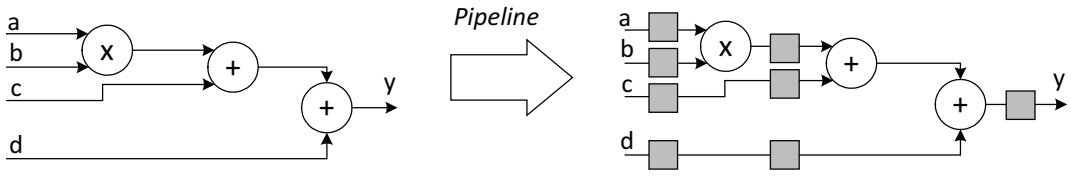


Figura 2.13. Efeitos do pipeline no caminho de dados.

As caixas no caminho de dados na Figura 2.13 representam os registradores. A cada ciclo do relógio o valor armazenado em cada registrador é atualizado pelo valor da sua entrada. Os registradores funcionam com uma barreira, dividindo o cálculo em etapas e desta forma, os dados calculados em cada etapa são passados para a etapa seguinte a cada ciclo do relógio. Por isso, na versão em pipeline da Figura 2.13 o resultado do cálculo para cada y levará três ciclos do relógio resultando em uma latência 3, porem a cada ciclo um novo resultado é gerado, ou seja, a vazão e o calculo de uma equação completa por ciclo. A Figura 2.14 mostra o cálculo do fluxo de dados através do tempo.

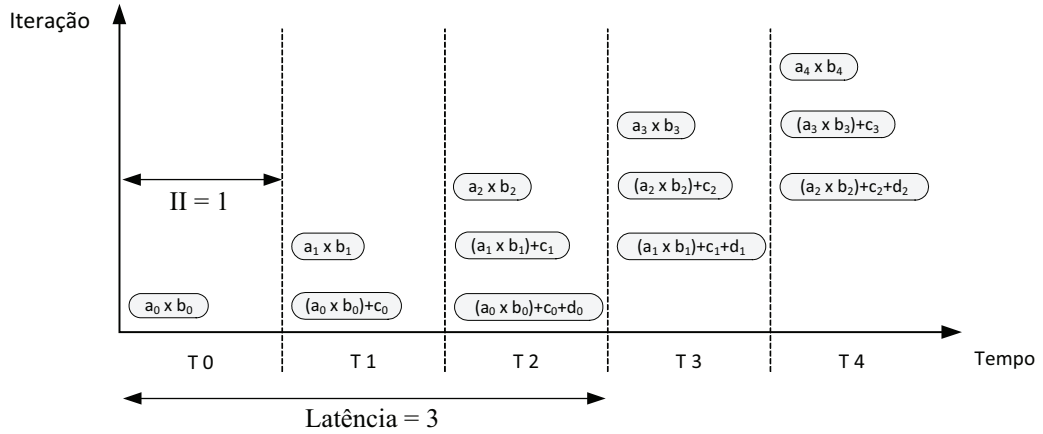


Figura 2.14. Paralelismo espacial com sobreposição temporal das operações

O tempo constante para iniciar cada nova iteração, respeitando as dependências e restrições, é denominado Intervalo de Inicialização (II). Quanto maior for o

valor de II, maior o tempo gasto para executar cada iteração do laço e consequentemente, menor será a vazão de instruções/dados da aplicação. O II é a métrica usada para medir a eficiência do software pipeline.

2.7 Modulo Scheduling

O escalonamento é o processo de identificação das dependências de dados e ordenação das operações para determinar quando cada uma irá executar. *Modulo Scheduling* (MS) é uma técnica de *Software Pipelining* que busca sobrepôr diferentes iterações da execução de um laço para explorar um maior grau de paralelismo de instruções (ILP)[7], de modo que cada iteração possa começar a sua execução antes do término das iterações anteriores [Rau, 1994] em intervalos regulares (II constante). Essa técnica tenta obter o escalonamento ótimo para uma iteração do laço de tal modo que quando o escalonamento é repetido, nenhuma restrição de dependência ou de recursos é violada.

As restrições de dependência podem ser internas ou externas. As dependências internas são aquelas em que os nós dentro de uma mesma iteração são dependentes entre si e a ordem de execução deve ser respeitada. As dependências externas são aquelas na qual os nós de uma determinada iteração dependem do resultado da iteração anterior, não permitindo que estas duas sejam sobrepostas. A restrição de recursos está relacionada com disponibilidade de recursos da arquitetura, como o tipo e a quantidade de unidades funcionais e a disponibilidade de roteamento da rede de interconexão.

Como já mencionado, o intervalo constante entre o início de iterações sucessivas é chamado de Intervalo de Iniciação (II). O intervalo inicial mínimo é chamado de MII, que é determinado pelo algoritmo de acordo as dependências de operações existentes no grafo. Contudo, não se pode garantir que o algoritmo irá conseguir sempre um II igual ao MII devido a restrições de recursos da arquitetura (unidades e roteamento).

2.7.1 Modulo Scheduling em CGRAs

Algoritmos de *Modulo Scheduling* foram usados em processadores VLIW (*Very Long Instruction Word*) [Colwell et al., 1988] para resolver problemas de escalonamento [Yun & Kim, 2001; Sánchez & González, 2000; Guo, 2005; Zalamea et al., 2001].

[7] Do inglês *Instruction Level Parallelism*

Porém, o uso de MS em CGRAs apresenta novos desafios devido à diferença entre essas arquiteturas [Park et al., 2008].

Em uma arquitetura VLIW o roteamento é feito de maneira implícita, pelo banco de registradores centralizado, diferente de um CGRAs, onde é preciso especificar explicitamente o roteamento entre as operações. Além disso, nos CGRAs que possuem unidades de processamento heterogêneas, as operações com maior custo de implementação como multiplicação, load e store podem estar disponíveis apenas em um subconjunto de UPs e por isso é preciso evitar alocar uma operação simples em uma UP mais complexa, de modo obter um melhor aproveitamento da arquitetura.

Em um CGRA, o *Modulo Scheduling* executa 3 passos:

- **Escalonamento**: No processo de escalonamento serão identificadas as precedências entre as operações e em seguida definido em qual tempo cada operação será executada.
- **Posicionamento**: Na etapa de posicionamento é definida em qual unidade de processamento será posicionada cada operação.
- **Roteamento**: O roteamento conecta as UPs, gerando uma cadeia de operações conforme a cadeia de dependências definida pelas aresta no grafo de fluxo de dados.

Um exemplo de mapeamento de um fluxo de dados em um CGRA é apresentado na Figura 2.15. Primeiramente é definido o escalonamento de cada nó (Figura 2.15 (b)), para que assim seja possível determinar a precedência entre as operações. Em seguida é feito o posicionamento dos nós do grafo (Figura 2.15 (a)) na arquitetura (Figura 2.15(c)) onde os nodos a, b, c, d, e e são alocados nas unidades UP_1, UP_4, UP_2, UP_3 e UP_6 respectivamente. No próximo passo é feito o roteamento conectando as UPs vizinhas para criar o caminho do fluxo de dados (Figura 2.15(d)).

Os valores calculados por cada UPs serão entregues para as unidades seguintes a cada ciclo do relógio. Note que na unidade UP_5 do exemplo não foi posicionado nenhum nó. Neste caso, a UP_5 é usada para receber e armazenar o valor da computação pelo nó b, ou seja, realiza um *store-and-forward* para equilibrar o fluxo de dados.

No exemplo da Figura 2.15, o número de UPs disponíveis foi suficiente para posicionar todas as operações do grafo. No entanto, na maioria dos casos, o número de UPs é menor que o número de operações a serem mapeadas, sendo necessário o uso de partições temporais que implicará em uma vazão menor que 1.

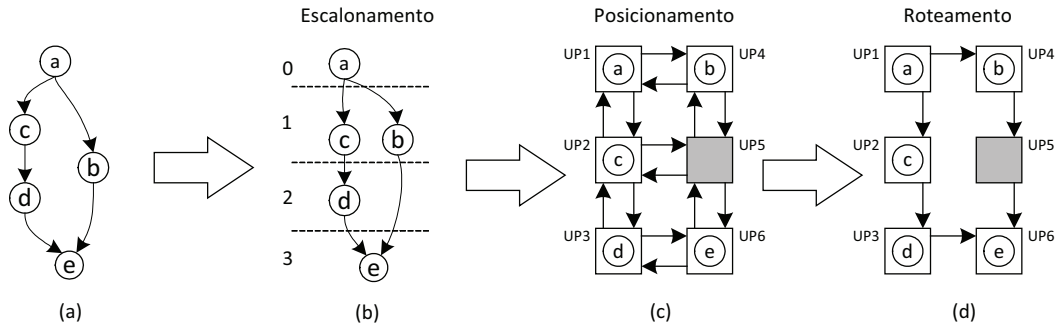


Figura 2.15. Escalonamento posicionamento e roteamento de um fluxo de dados

Na Figura 2.16 é ilustrado o mesmo grafo com 5 nós do exemplo anterior, sendo mapeado em um CGRA com apenas 4 UPs (Figura 2.16(b)). Assim, o particionamento temporal é usado e duas configurações para o CGRA são criadas ((Figura 2.16(c))), de modo que a cada ciclo estas configurações vão se alternando.

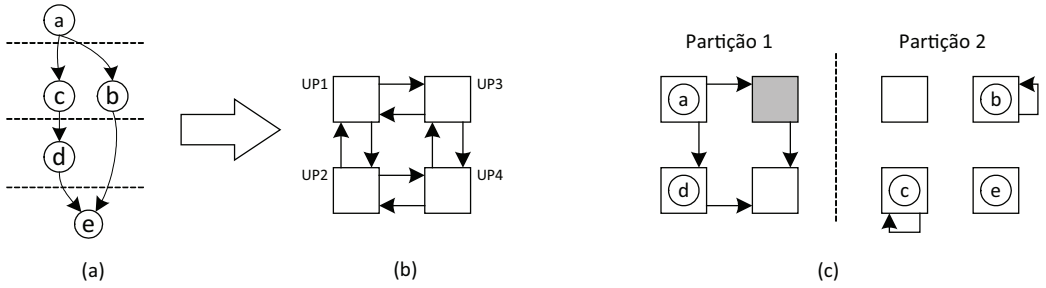


Figura 2.16. Mapeamento em um CGRA com número de UPS menor que a quantidade de nós do grafo.

O processo de execução do exemplo da Figura 2.16 está ilustrado na Figura 2.17 e acontece da seguinte forma:

- **Tempo 0**: A UP_1 na partição 1 recebe os dados de entrada e o resultado da operação é passado para a UP_2 e UP_3 da partição 2, onde estão posicionados os nós c e b respectivamente.
- **Tempo 1**: O dado calculado por UP_2 é passado para a própria UP_2 da partição 1, onde está posicionado o nó d. O mesmo acontecerá com a UP_3 que passará o valor para a própria UP_3, da partição 1, para armazenamento.
- **Tempo 2**: O dado calculado pelo nó d na UP_2 e o dado armazenado na UP_3 são passados para a UP_4 da partição 2, onde está posicionado o nó e. Ainda no tempo 2 um novo valor é computado por a na UP_1, começando um novo ciclo.

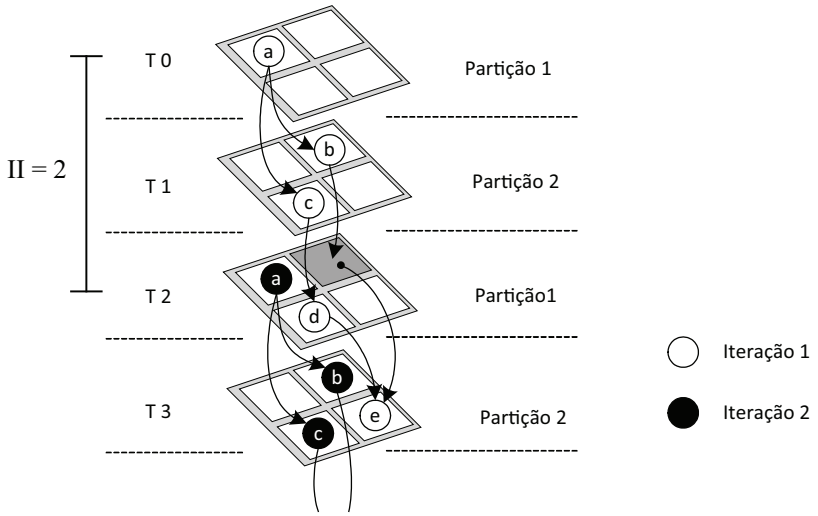


Figura 2.17. Exemplo de execução do CGRA com o número de UPS menor que a quantidade de nós de um grafo. (a) Grafo que será mapeado. (b) CGRA com 4 UPs. (c) Duas configurações geradas para duas partições temporais.

- **Tempo 3**: O resultado final da primeira iteração é calculado pelo nó e na UP_4, enquanto que os dados da segunda iteração são calculados pelos nós c e b.

Neste exemplo o resultado da última operação do *loop* para a primeira iteração que começou no ciclo 1, foi calculado pelo nó e no tempo 3, portanto a latência será 3. Como existe uma sobreposição das execuções das iterações um novo resultado é calculado a cada 2 ciclos do relógio, fazendo a vazão de dados igual a 1/5 ciclo, que corresponde ao $II = 2$. Como são executadas 5 operações em 2 ciclos, o paralelismo a nível de instrução é $ILP = 2,5$.

2.8 Trabalhos Relacionados

Este trabalho faz parte da linha de pesquisa em escalonamento, posicionamento e roteamento de fluxos de dados em tempo de execução na arquitetura proposta por [Ferreira et al., 2011a]. Duas linhas diferentes foram desenvolvidas, MSG [Lopes, 2013] e EPR [Mucida, 2013]. Ambas são baseadas em heurísticas gulosas que percorrem o grafo de fluxo de dados, que representa o laço, apenas uma vez. O nosso objetivo é fornecer uma solução complementar na forma de módulo que possa ser inserido juntamente com o CGRA virtualizado sobre um FPGA.

A Figura 2.18 apresenta uma visão geral das duas soluções. Em ambas, a execução das aplicações é realizada pelo processador de propósito geral *softcore* e

pelo acelerador CGRA de laços paralelizáveis, ambos programados no FPGA. Um compilador externo gera as instruções para o *softcore*, que em tempo de execução, executa o algoritmo de mapeamento para o CGRA.

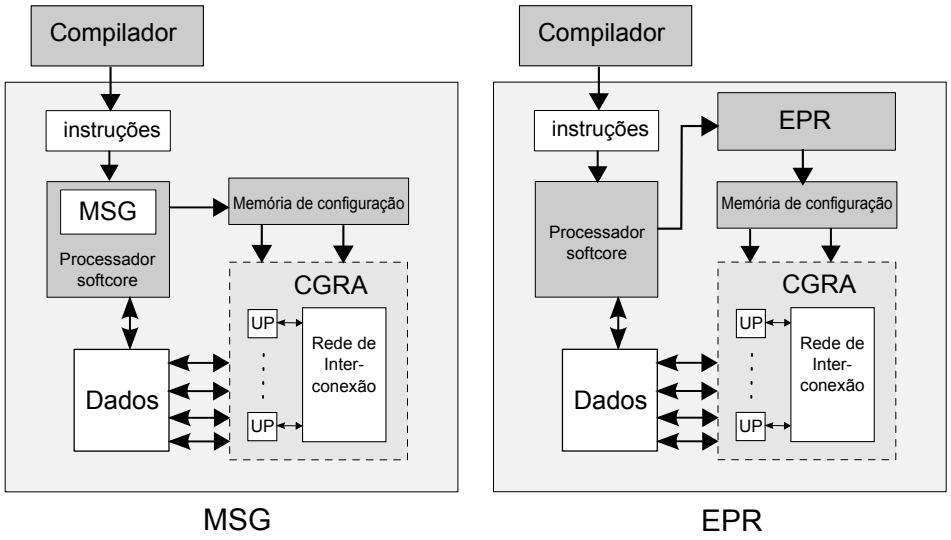


Figura 2.18. Ambiente de aceleração de laços com computação intensiva, em duas abordagens diferentes: à esquerda com o algoritmo de mapeamento MSG implementado para executar sequencialmente no processador softcore e à direita com o algoritmo EPR implementado em hardware (EPR).

A abordagem MSG, proposta por [Lopes, 2013], faz o processamento do grafo por vértice e foi implementada para compilação *Just-in-Time*. O algoritmo de mapeamento foi implementado em linguagem de programação C, para ser executado em um *softcore*, sintetizado juntamente com o CGRA no FPGA. A abordagem MSG também realizou uma redução no número de registradores com *multicasting* utilizados no mapeamento e propôs uma modificação na arquitetura CGRA, que possibilita o melhor uso dos registros de cada elemento de processamento. Os testes realizados pela abordagem MSG foram feitos utilizando um ambiente simulado. Esta dissertação dá continuidade ao trabalho MSG com a implementação e avaliação das arquiteturas em FPGA, com resultados publicados em conjunto em [Ferreira et al., 2013].

A abordagem EPR foi proposta por [Mucida, 2013] e visa uma implementação em *hardware* do algoritmo de mapeamento. Para essa abordagem, o grafo é descrito como uma lista de aresta que simplifica a implementação em *hardware*. O algoritmo foi desenvolvido em linguagem C em forma de máquina de estados. A implementação em hardware deste algorítimo é validada nesta dissertação e foi implementada em VHDL e será apresentado no capítulo 4. Os resultados foram publicados em conjunto

em [Mucida et al., 2012] e [Filho & Ferreira, 2012].

Desta forma, esta dissertação complementa as duas dissertações [Lopes, 2013; Mucida, 2013] e mostra dois pontos fundamentais: Primeiro, qual é o custo em área, bits de configuração e atraso para se implantar os CGRAs com rede global como uma camada virtual em um FPGA comercial. Segundo, como o algoritmo EPR pode ser implementado em hardware e qual é o seu custo e seu desempenho. Apesar de esta dissertação apresentar apenas a implementação em *hardware* do algorítimo EPR, a mesma abordagem pode ser usada para o MSG.

3. Arquiteturas

Neste capítulo apresentamos variações do CGRA proposto inicialmente em [Ferreira et al., 2011a]. Nesta dissertação propomos variações na forma de implementação das unidades de processamento e suas conexões com a rede Global.

O intuito do trabalho não é o desenvolvimento de um dispositivo CGRA, mas um CGRA que possa ser construído como uma camada virtual sobre um FPGA. Dessa forma, buscou-se avaliar o custo da virtualização aliado ao uso dos recursos já disponíveis em alguns FPGAs comercias, como DSPs e Memórias embarcadas. No capítulo 5 será apresentada uma comparação dos recursos utilizados pelas as diferentes variações bem como o desempenho de cada uma delas.

Um CGRA é constituído basicamente de três partes: Unidades de Processamento, Rede de Interconexão e Memória de Configuração. A seguir, iremos descrever cada uma destas partes e explicar as variações propostas.

3.1 Unidades de Processamento

Em um CGRA, as unidades de processamento são responsáveis pela execução das operações descrita pelos nós de um Grafo de Fluxo de dados[1]. Em geral são operações logicas e/ou aritméticas como adição e multiplicação. Se a arquitetura é homogênea, ou seja, todas as unidades de processamento são iguais, cada uma destas unidades deve ser capaz de executar todas as operações. Se forem heterogêneas, as unidades de processamento são divididas em grupos, nas quais as unidades de um determinado grupo realizam operações diferentes das unidades de outros grupos, o que pode reduzir os custos da arquitetura. Neste trabalho, todas as arquiteturas construídas utilizam um conjunto de unidades de processamento homogêneas. Porém, como as implementações são parametrizadas, é possível derivar arquiteturas heterogêneas.

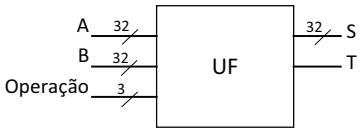
A unidade de processamento de um CGRA é basicamente construída a partir de uma Unidade Funcional (UF), alguns multiplexadores e registradores. A UF é o componente fundamental de uma unidade de processamento (UP).

[1]Grafo que descreve o fluxo de dados e a dependência entre as operações para implementação de uma aplicação ou parte dela.

3.1.1 Unidades Funcionais

Neste trabalho foram utilizados 3 diferentes tipos de UFs. O objetivo da variação dos tipos das UFs é avaliar o custo em função da complexidade da unidade, bem como o atraso e o impacto em relação ao custo das interconexões.

A Figura 3.1 apresenta a UF construída na primeira etapa do desenvolvimento das arquiteturas. Esta é uma UF simples com 3 bits de configuração, capaz de executar 8 operações conforme ilustra a Figura 3.1(b). A descrição em VHDL desta FU utiliza um código parametrizado em função do tamanho de palavra. Os resultados de síntese mostrados na Figura 3.1(b) são para uma UF que foi sintetizada para trabalhar como palavras de 32 bits.



(a)

	Operação
0 0 0	A + B
0 0 1	| A – B |
0 1 0	A **and** B
0 1 1	A **or** B
1 0 0	A **xor** B
1 0 1	**not** A
1 1 0	**not** B
1 1 1	A

(b)

Recursos	
Número de **Slice Registers**	25
Número de **Slice LUTs**	194
Número de **occupied Slices**	70
Número de **LUT Flip Flop pairs**	210
Número de **RAMB36E1/FIFO36E1s**	0
Número de **DSP48E1s**	0

(c)

Figura 3.1. (a) Unidade Funcional 1. (b) Tabela de operações. (c) Resultado dos recursos utilizados na síntese de uma UF de 32 bits para o FPGA Virtex6 XC6VLX240T

O foco do trabalho é avaliar a implementação de um CGRA para ser construído como uma camada em um FPGA, por isso é importante que se busque aproveitar ao máximo os recursos disponíveis em um FPGA.

Os FPGAs mais modernos expandem suas capacidades incluindo novos componentes e funcionalidades diretamente no silício, reduzindo a área e aumentando o desempenho. Exemplos são as memórias embutidas e os blocos de DSPs genéricos.

Um bloco DSP (*Digital Signal Processing*) é uma unidade projetada especialmente para o processamento de sinais digitais. As arquiteturas mencionadas neste trabalho foram avaliadas em um FPGA Virtex6 XC6VLX240T que possui 768 módulos DSPs distribuídos em várias partes do FPGA.

Os FPGAs da família Virtex6 possuem DSPs do tipo DSP48E1. Este DSP suporta diversas funções independentes. Essas funções incluem multiplicação, multiplicar e acumular (MACC), multiplicar e somar, soma com 3 entradas, deslocamento de bits, multiplexadores para grande barramentos, comparador de magnitude, fun-

ção lógica bit a bit, detector de padrões e contador grande. O desenho da estrutura interna de um DSP48E1 é mostrado na Figura 3.2.

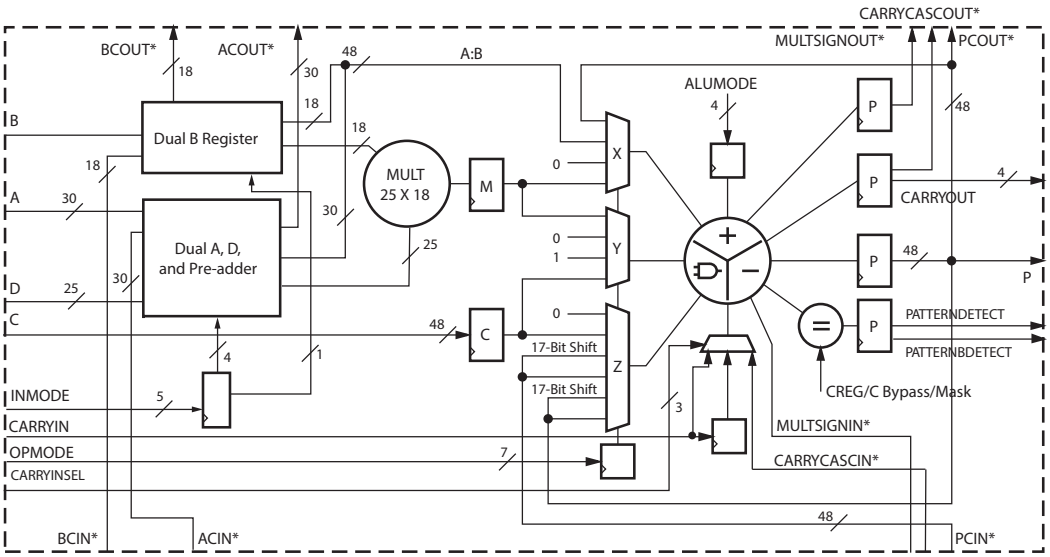


Figura 3.2. Virtex-6 FPGA DSP48E1

A segunda UF avaliada foi especificada com o intuito utilizar os DSPs disponíveis no FPGA. O objetivo é substituir a UF anterior por uma UF especificada a partir de um único DSP sem que isso afete as outras partes do CGRA. Então o tamanho das palavras manipuladas por esta nova UF e os bits de configurações de operações devem permanecer os mesmos. Para isso foi necessário restringir as funcionalidades do DSP deixando alguns parâmetros de configurações pré-definidos. Na figura 3.3 é apresentada a UF especificada a partir de um DSP48E1. Desta forma o único recurso do FPGA utilizado para construção de uma UF é um dos DSPs disponíveis.

É importante ressaltar que nesta abordagem, a UF não possui os mesmos conjuntos de instruções da UF apresentada anteriormente, mas neste trabalho, o objetivo principal foi avaliar a redução da área ocupada e a variação da frequência de relógio da arquitetura. É importante observar também que os DSPs possuem registradores internos que interferem na latência da arquitetura.

Os CGRAs, que utilizam as duas UFs apresentadas em suas unidades de processamento, não possuem a capacidade de executar operações mais complexas, como multiplicações. Porém esta é uma operação fundamental. Por isso, foi avaliada também a influência de uma unidade lógica aritmética capaz de executar multiplicação. Por isso a terceira UF foi especificada, semelhante à UF 1, porém substituindo uma

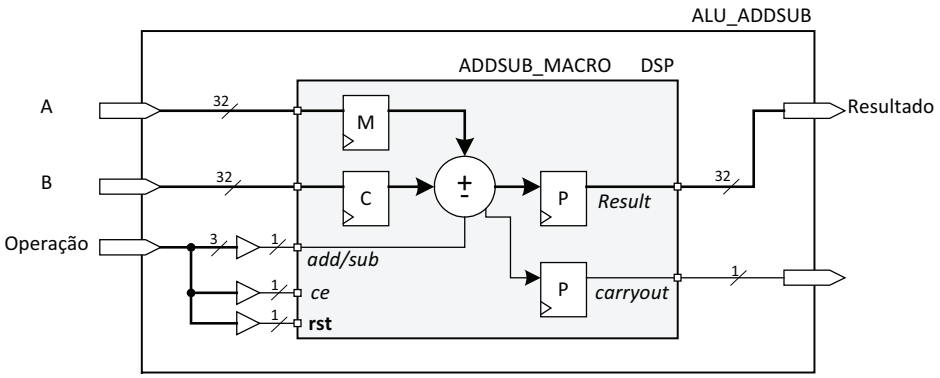


Figura 3.3. UF sintetizada utilizando um DSP disponível no FPGA

de suas operações por uma operação de multiplicação. Na Figura 3.4(a) é apresentada a lista de operações dá UF e na Figura 3.4(b) podemos ver o resultado de sua síntese. Note que se comparado com o resultado da síntese da UF 1, na Figura 3.1(c), o número de LUTs utilizado foi menor, porém neste caso, o sintetizador da arquitetura capturou 3 DSPs para ajudar compor a nova UF.

	Operação
0 0 0	A + B
0 0 1	|A - B|
0 1 0	A **and** B
0 1 1	A **or** B
1 0 0	A **xor** B
1 0 1	**not** A
1 1 0	A **x** B
1 1 1	A

Operação de multiplicação adicionada

(a)

Recursos	UF3	UF2	UF1
Número de *Slice Registers*	**66**	0	25
Número de *Slice LUTs*	**147**	0	194
Número de *occupied Slices*	**64**	0	70
Número de *LUT Flip Flop pairs*	**180**	0	210
Número de *RAMB36E1/FIFO36E1s*	**0**	0	0
Número de *DSP48E1s*	**3**	1	0

(b)

Figura 3.4. UF com multiplicação. (a)Tabela de operações. (b)Recursos em hardware da UF de 32 bits com multiplicação e destaque, comparada com as UF anteriores.

3.2 CGRA com rede global de interconexão

A Figura 3.5 apresenta o modelo geral da arquitetura com rede global proposta em [Ferreira et al., 2011a], que será a base das variações avaliadas nesta dissertação. Este CGRA é constituído por unidades de processamento, multiplexadores, uma rede global de interconexão e uma memória de configuração. As UPs são responsáveis pela execução dos cálculos e cada uma destas UPs possui a saída conectada à rede global. Em geral, como cada UP tem duas entradas e uma saída, para manter a

simetria cada saída da UP é conectada a duas entradas da rede. A rede global permite conectar diretamente qualquer par de UP. Para conectar a UP_0 na UP_1 por exemplo, basta solicitar o roteamento de uma conexão da entrada 0 da rede para a saída 1. Desta forma, o resultado calculado por qualquer UP pode ser repassado para qualquer uma das duas entradas de qualquer outra UP no próximo ciclo do relógio.

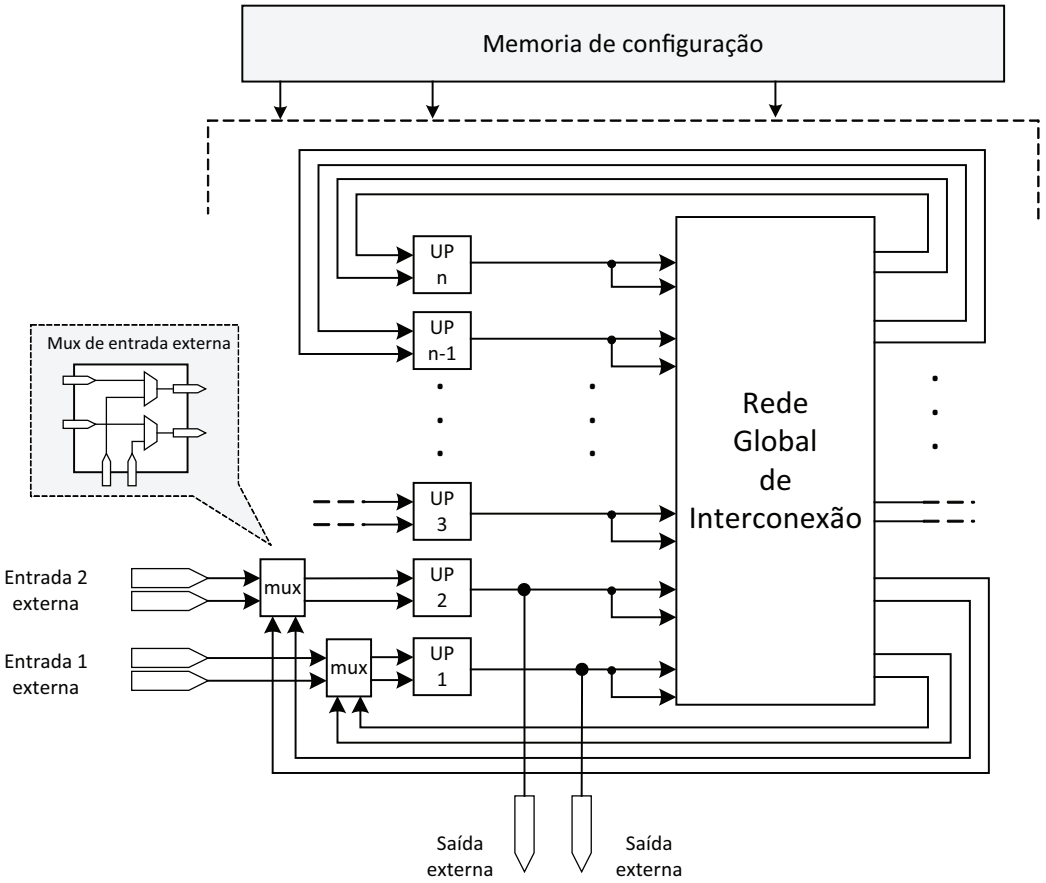


Figura 3.5. Modelo geral do CGRA

Algumas das unidades de processamento podem receber dados externos, por isso essas unidades possuem um multiplexador de entrada externa. Desta forma, uma UP de entrada pode receber os dados de outras UPs através da rede ou dados externos ao CGRA.

A memória de configuração armazena as informações que determinam qual operação cada UP irá executar, como será feito o roteamento na rede global e o controle das entradas externas. A largura da memória irá depender do tamanho da arquitetura, do número de unidades que suportam entrada de dados, do tipo das unidades de processamento e do número de operações suportados pela UF. Cada

linha da memória armazena uma configuração. Então, toda configuração do CGRA é alterada simplesmente mudando o endereço de acesso à memória.

3.3 Tipos de interconexões

A rede de interconexão é responsável pela comunicação entre as unidades de processamento do CGRA. Ela tem impacto no custo da implementação física e também na complexidade do algoritmo de mapeamento. Diferente de outras arquiteturas, uma arquitetura que utiliza uma rede global como elemento de conexão entre as unidades, é capaz de fazer o roteamento entre quaisquer unidades de processamento diretamente, já que uma unidade pode alcançar qualquer outra unidade diretamente.

Neste trabalho, dois tipos de redes foram avaliados: As Redes *CrossBar* e as Redes Multiestágios.

3.3.1 Rede Crossbar

Redes *crossbar* são redes capazes de realizar qualquer permutação de entrada e saída, isso significa que não ocorre conflitos de roteamento, porém seu custo em comutadores é $O(N^2)$ para N entradas e saídas.

A implementação de uma rede *crossbar* é muito simples. Em uma rede de oito entradas, por exemplo, teremos oito multiplexadores com oito entradas. Então, a saída de cada multiplexador será uma saída da rede e cada uma das entradas destes multiplexadores estarão ligados em paralelo às entradas equivalentes dos outros multiplexadores e conectadas à uma entrada da rede. A Figura 3.6 ilustra a construção de uma rede *crossbar* com oito entradas/saídas.

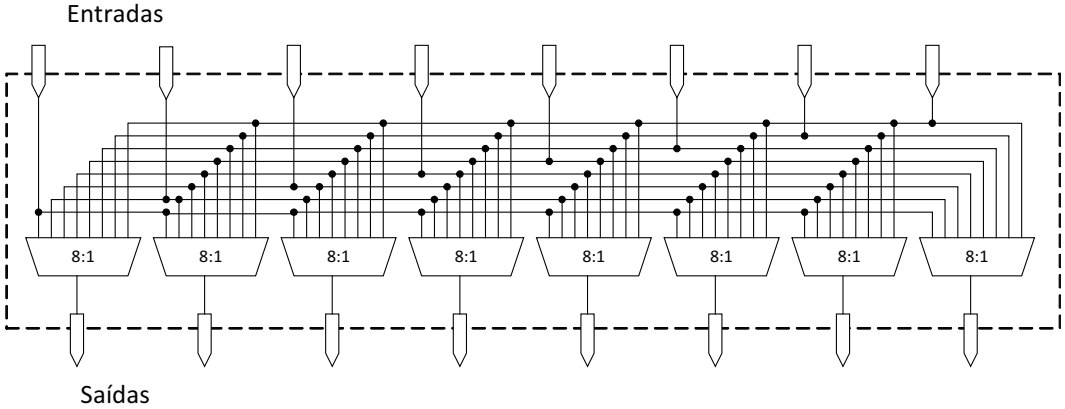


Figura 3.6. Construção de uma rede crossbar com 8 entradas/saídas.

3.3.2 Rede Multiestágio

Uma das motivações para o uso das redes multiestágio nas arquiteturas testadas nesta dissertação foi um estudo do espaço ocupado pelas MINs mapeados em uma arquitetura FPGA. Este estudo foi desenvolvido no trabalho de [Vendramini, 2012] e pelo trabalho desenvolvido por [Ferreira et al., 2011a]. Foi analisado o custo em área da rede MIN em função da frequência de relógio e do número de LUTs (*loockUP tables*).

O número de entradas e saídas de uma rede é a potência do radix do comutador para redes homogêneas. Para radix-4, que foi utilizado neste trabalho, temos 4^N. O número de colunas é Lg_4N e o número de comutadores utilizados é $N * Lg_4N$ que é muito menor que o custo $O(N^2)$ de comutadores em uma rede crossbar. A Figura 3.7 (a) mostra uma rede radix-4 com 16 entradas e saídas.

Um comutador com radix-N pode ser visto como uma pequena rede crossbar com N entradas e N saídas. Se aumentarmos o tamanho do radix, até o número de entradas da rede, teremos uma rede *crossbar*. A Figura 3.7(b) ilustra um comutador radix-4.

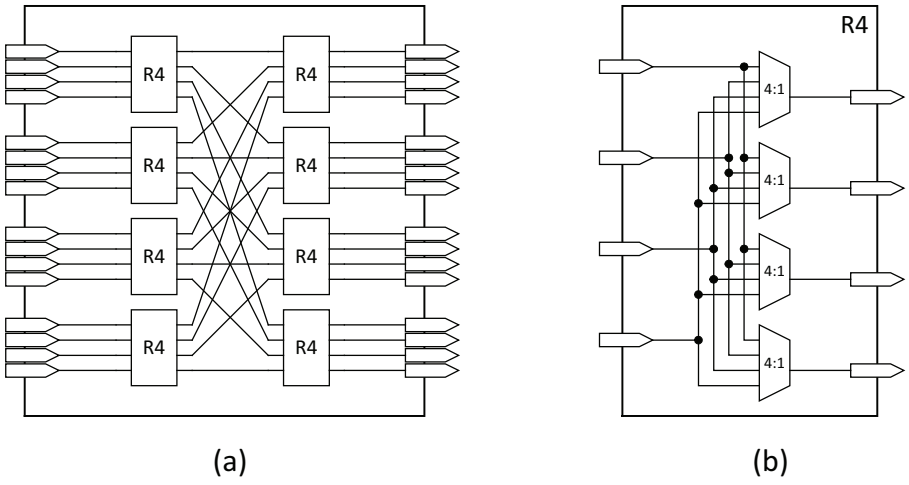


Figura 3.7. (a) Rede multiestágio com 16 entradas. (b) Comutador radix-4

3.3.3 Redes no CGRA

Como já foi dito, um dos objetivos neste trabalho é avaliar o uso das redes multiestágios como uma rede de interconexão entre as unidades de processamento de um CGRA. Neste trabalho foram desenvolvidas arquiteturas de diferentes tamanhos. As arquiteturas com redes multiestágio utilizam 16, 32 e 64 unidades de proces-

samento. Como já mostrado anteriormente, cada UP possui duas entradas com etapas de roteamento independentes conectadas à rede global, e por isso, o número de entrada/saída da rede é o dobro da quantidade de UPs.

Diferentemente de uma rede *crossbar*, em uma rede multiestágio o número de entradas não pode ser simplesmente escolhido indiscriminadamente. Isso porque o número de entradas da rede é uma potência do radix do comutador desta rede. Na rede multiestagio radix-4, as arquiteturas com 16 e 64 UPs, necessitam de 32 e 128 entradas/saídas respectivamente. Como cada etapa de roteamento é independente, o roteamento poderá ser feito por redes separadas. Já em uma rede com 32 unidades a solução será usar uma única rede de 64 entradas, (porque 32 não é potência de 4) e conectar cada uma das duas saídas das UPs nessa rede. A Figura 3.8(a) ilustra uma rede global com 32 entadas/saídas, para um CGRA com 16 UPs e na Figura 3.8(b) uma rede global de 64 entradas para um CGRA com 32 UPs.

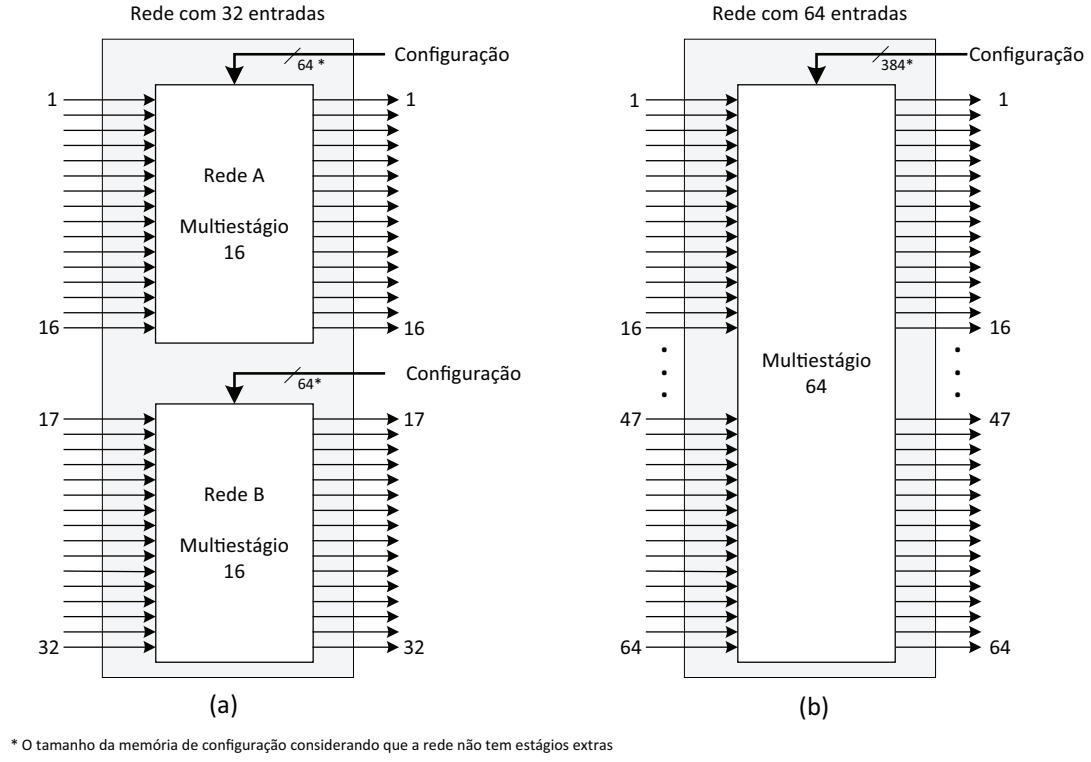


Figura 3.8. (a) Rede de interconexão de 32 entradas para uma CGRA com 16 UPs . (b) Rede de interconexão com 64 entradas para um CGRA com 32 UPs

3.4 Arquiteturas

As arquiteturas apresentadas neste trabalho são baseadas na arquitetura proposta por [Ferreira et al., 2011a]. Na arquitetura original, cada elemento de processamento possui uma UF e dois registradores: A e B. A saída da UF é conectado à duas redes multiestágios e a saída equivalente na rede é conectada de volta a cada um dos dois registradores A e B. Isso permite que as entradas A e B possam receber os dados de diferentes UFs da partição temporal anterior.

Neste trabalho, diversas variações da arquitetura base, proposta por [Ferreira et al., 2011a] foram feitas, com o objetivo de avaliar em cada uma delas as variação no uso dos recursos do FPGA, o desempenho de execução e melhorar a ocupação do CGRA.

Foi implementado também uma arquitetura para comparação baseado no modelo ADRES [Mei et al., 2005a], compatível em termos de processamento e complexidade com as outras arquiteturas deste trabalho. O objetivo é comparar os resultados com trabalhos correlatos para *Modulo Scheduling* que em sua grande maioria são baseados nas arquitetura em malha tipo ADRES. Além disso, avaliar o custo de uma arquitetura com rede global em comparação com uma arquitetura com rede local como a ADRES.

3.4.1 Arquitetura Mono1R

A arquitetura Mono1R(monotarefa com um registrador) é a mais simples. Cada unidade de processamento possui apenas uma UF e um registrador conectado à saída da UF, conforme ilustra a Figura 3.9. O registrador é necessário para tornar a arquitetura síncrona, ou seja, garantir que a cada ciclo do relógio o dado calculado na saída de cada UF seja entregue para as UFs no próximo ciclo.

As redes utilizadas nessa estrutura podem ser redes *Crossbar* ou redes multiestágio e o tamanho da arquitetura em função do número de UPs pode variar.

Algumas UPs são usadas para entrada de dados no CGRA. Por isso, em cada uma destas UPs possui dois multiplexadores, para selecionar a origem de entrada de dados.

3.4.2 Arquitetura Mono2R

A arquitetura Mono2R(monotarefa com dois registradores) é semelhante à arquitetura Mono1R. No entanto o registrador que antes estava na saída, passa agora para

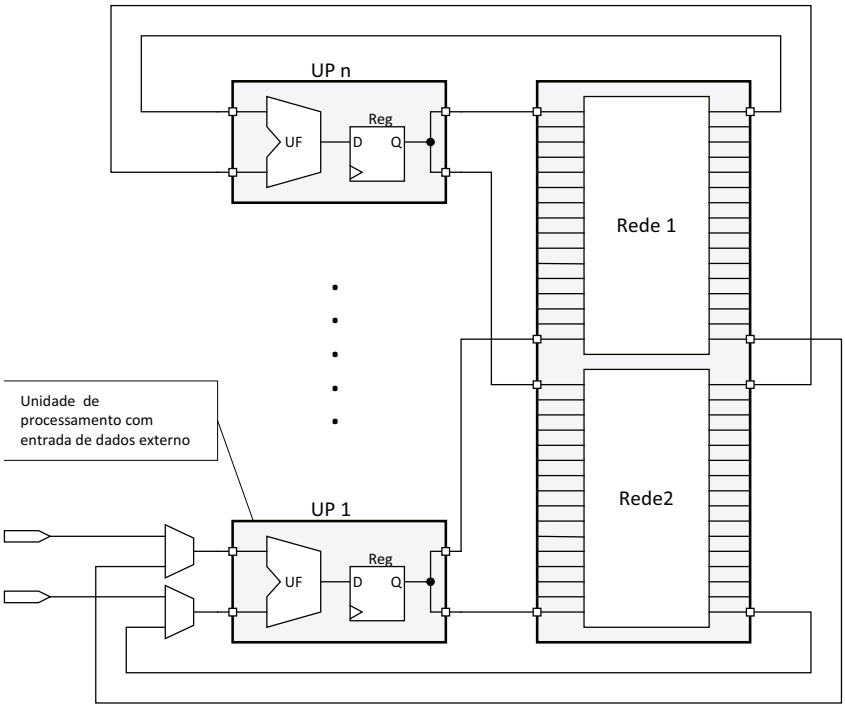


Figura 3.9. Arquitetura Mono1R

a entrada desta UF. O efeito no funcionamento é o mesmo, porém, como cada UF possui duas entradas, serão necessários dois registradores por unidade de processamento. Então na arquitetura Mono2Rcada UP possui uma UF e dois registradores: A e B. A saída da UF é conectada diretamente nas duas redes. Isso permite que as entradas A e B recebam dados de diferentes UFs. A Figura 3.10 ilustra a arquitetura com dois registradores para cada UF. A desvantagem desta abordagem em comparação com a Arquitetura Mono1Ré o aumento no custo da UPs, já que antes era necessário apenas em registrador e agora são necessários dois para que o CGRA funcione corretamente. Porém desta forma cada UP será capaz de armazenar dois valores temporários. Outro objetivo desta mudança é avaliar a variação na frequência do ciclo de relógio de uma arquitetura para a outra.

3.4.3 Arquitetura Mult2R3M

A arquitetura Mult2R3M(multitarefa com dois registradores e três multiplexadores) possui alguns registradores e multiplexadores adicionais, conforme ilustra a Figura 3.11. Esta abordagem é uma alternativa proposta por [Ferreira et al., 2013] e [Lopes, 2013] para melhorar a utilização da arquitetura. Na arquitetura Mono2R, quando uma unidade de processamento é usada como um registro de balanceamento,

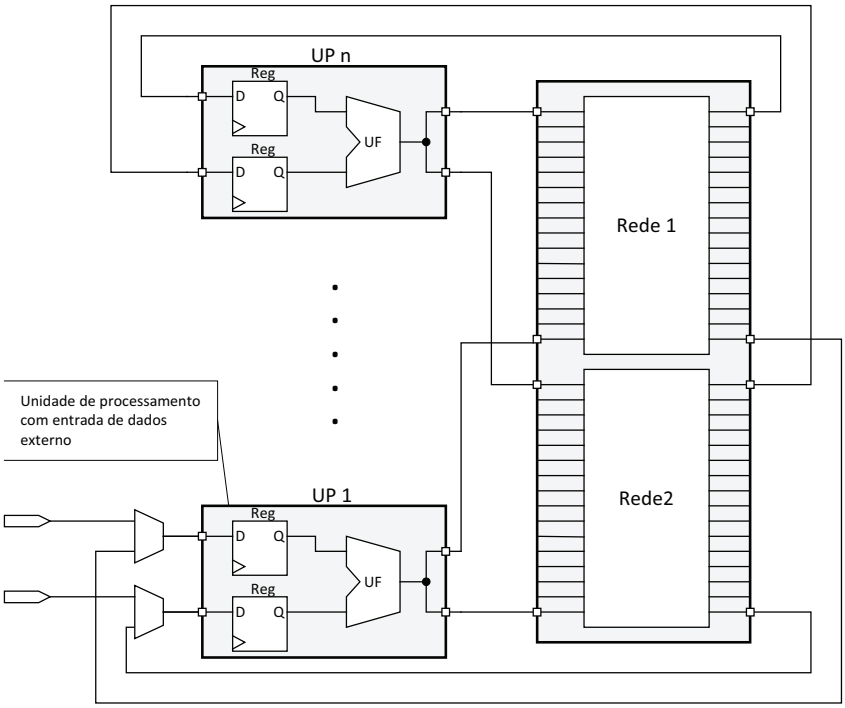


Figura 3.10. Arquitetura Mono2R

o registro A é utilizado, o Registro B é ignorado e a UF passa o valor de A diretamente para a saída.

Considere o Grafo do fluxo de dados ilustrado na Figura 3.12(a) que será mapeado para um CGRA. Na arquitetura Mono2R, conforme ilustra a Figura 3.12 (b), apenas o registrador A da UP_5 e o registrador A da UP_6 são utilizados para mapear os registradores de $R1$ e $R2$ respectivamente. Como na arquitetura Mono2Rcada UP já possui dois registradores locais, é possível fazer o uso de ambos multiplexadores com inclusão de 1 multiplexador por UP, como mostra a Figura 3.12(c), utilizado pela arquitetura Mult2R3M.

3.4.4 Arquitetura Mult2R

A arquitetura Mult2R(multitarefa com dois registradores) é ilustrada na Figura 3.13. Esta é uma tentativa de utilizar melhor os registradores disponíveis em cada UP, assim como a arquitetura Mult2R3M. Cada UP da arquitetura possui dois registradores, A e B ligados à entrada da UF. Além disso, o registrador B possui um desvio que o conecta diretamente à rede. Desta forma cada UP possui duas conexões com a rede. Na UP_i, a UF_i é conectada à $entrada_i$ da rede, enquanto que o registrador B_i é conectado à $entrada_{i+32}$ da rede.

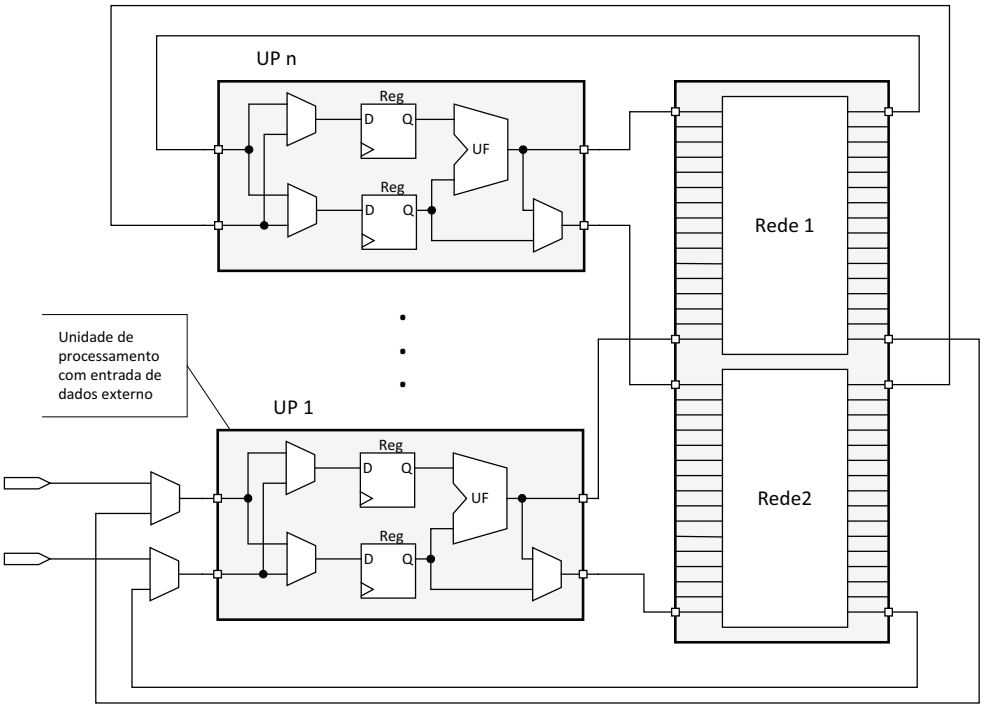


Figura 3.11. Arquitetura Mult2R3M

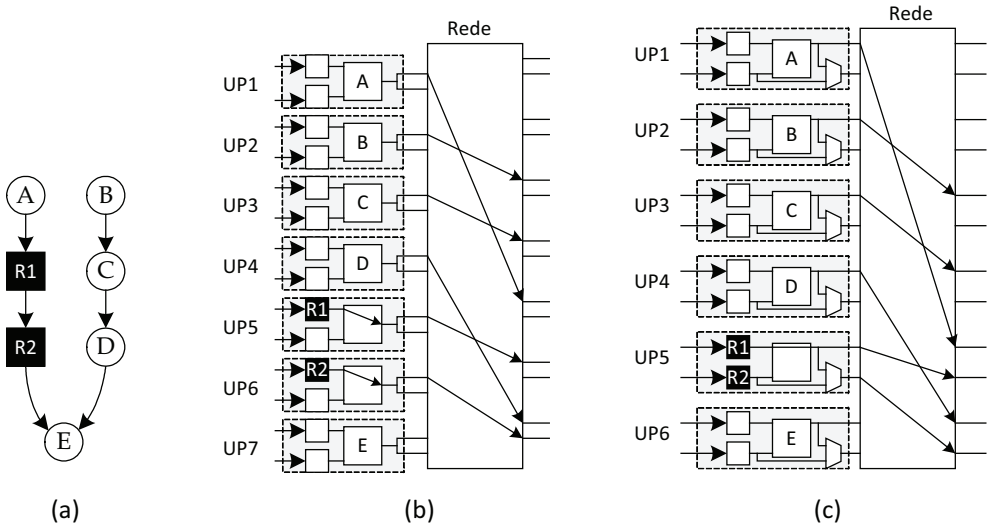


Figura 3.12. (a) Grafo do fluxo de dados. (b) Grafo mapeado na arquitetura Mono2R. (c) Grafo mapeado na arquitetura Mult2R3M

Por causa do padrão de conexão da UF à rede e o desvio que conecta o registrador B diretamente com a rede, será necessária uma rede de interconexão global com capacidade de roteamento completo. Isso significa que qualquer entrada deverá ser capaz de se conectar com qualquer saída, o que não é possível em uma rede composta por duas sub-redes separadas. Por isso, diferente das arquiteturas anteri-

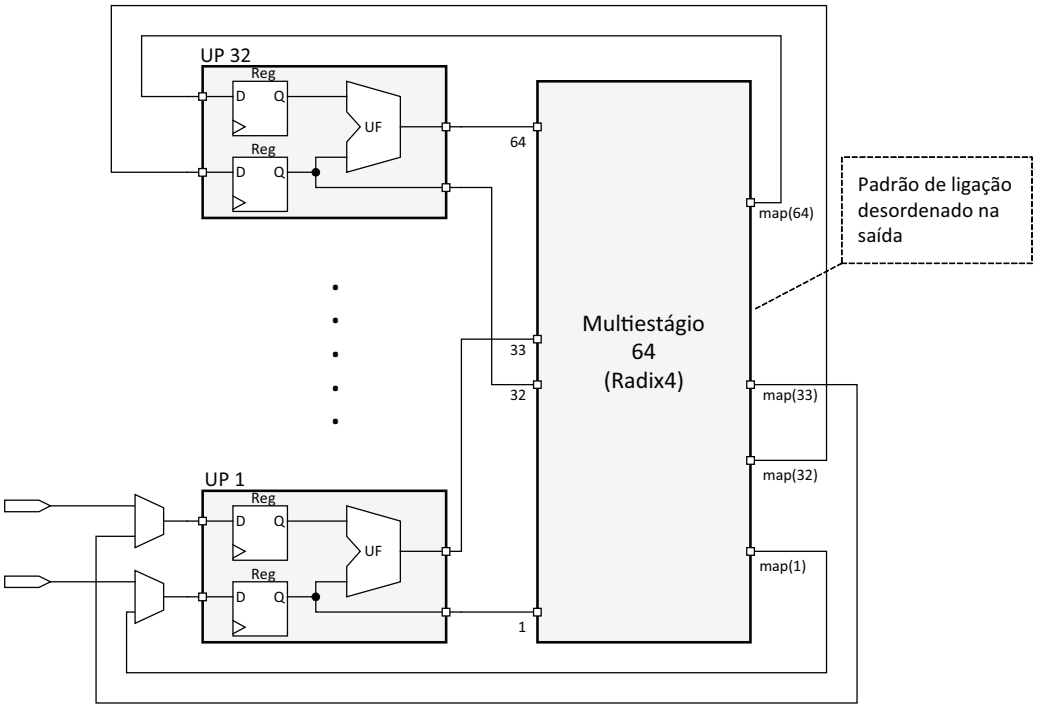


Figura 3.13. Arquitetura Mult2R

ores, a arquitetura Mult2Rutiliza apenas uma rede para prover todo o roteamento entre as UPs. A rede utilizada neste caso é uma rede multiestágio radix-4 com 64 entradas, suficiente para um CGRA com 32 UPs. Como agora existe apenas uma rede sendo usada para rotear para as duas entradas de cada UFs, o número de conflitos de roteamento aumentará. Por isso na arquitetura Mult2Ra rede multiestágio possui um estagio extra, aumentando seu poder de roteamento.

Mesmo com um estágio extra na rede multiestágio, conflitos no roteamento poderão acontecer. No entanto, conforme testes efetuados com *Benchmarks* [Mucida, 2013], observou-se que um padrão de conexão de saída aleatório pra este CGRA, melhora o roteamento, diminuindo os conflitos na rede, quando a heurística gulosa de *Modulo Scheduling* é executada conforme descrito em [Mucida et al., 2012]. Por isso, diferente das arquiteturas anteriores, o padrão de conexão de saída da rede não segue a mesma ordem do padrão de entrada. Porém, para que seja possível fazer um roteamento correto entre as Unidades de Processamento, este padrão de conexão deverá ser conhecido previamente pelo algoritmo de roteamento do Fluxo de dados no CGRA. Então, a relação de conexão entre a saída da rede e a UPs é definido por uma função $map(i)$, que possui um vetor com o padrão de conexão que é utilizado também no roteamento pelo algoritmo de escalonamento.

3.4.5 Arquitetura ADRES

A arquitetura ADRES proposta por [Mei et al., 2005a], foi usada para comparação com as arquitetura experimentadas neste trabalho. Esta arquitetura em malha é mais escalável pois as unidades de processamento são equipadas com uma unidade funcional e um banco de registradores. Diferente das arquiteturas anteriores, esta arquitetura não possui uma rede global. Cada UP é conectado aos UPs vizinhos e a saída desta UP é acessada por estes vizinhos no ciclo seguinte.

Neste trabalho, buscou-se comparar uma arquitetura que utiliza uma rede global à uma arquitetura em malha. Então, para que a arquitetura ADRES pudesse ser usada como uma arquitetura de comparação foi preciso construir uma simplificação desta arquitetura, compatível em termos de funcionalidades com as outras arquiteturas. Na Figura 3.14 é ilustrada a forma geral da arquitetura Adres 4x4 construída.

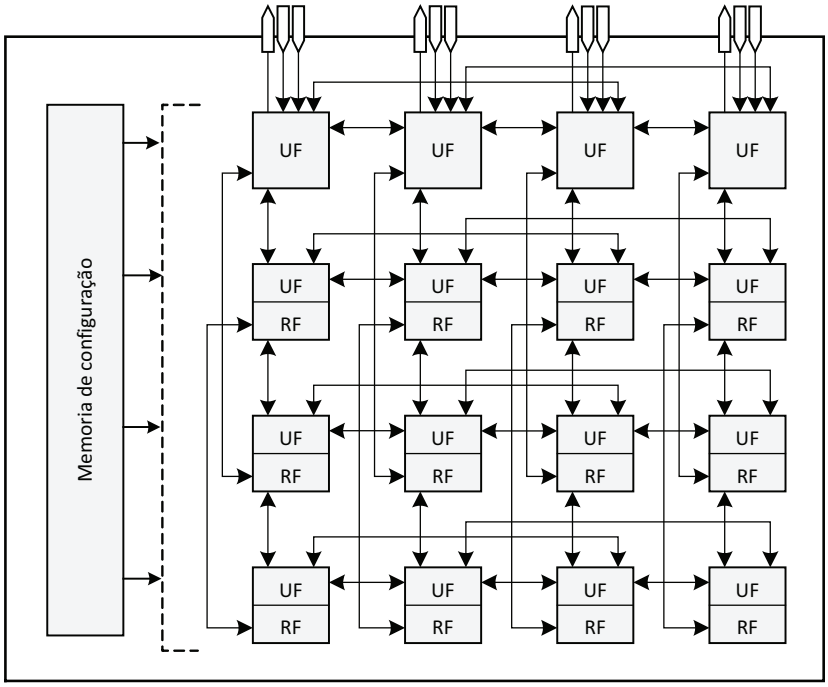


Figura 3.14. Arquitetura ADRES simplificada

Na arquitetura ADRES, somente as UPs da primeira linha são usadas como entradas. Estas UPs também são mais simples, já que não possuem um banco de registradores, como nas demais UPs. É importante observar que nesta arquitetura, as unidades funcionais também são homogêneas, ou seja, todas as UPs realizam o mesmo conjunto de operações. Por isso, as UFs utilizadas nesta arquitetura são as mesmas UFs utilizada nas demais arquiteturas construídas apresentadas na Se-

ção 3.1.1. Então, basicamente a diferença entre as arquiteturas está na rede de interconexão e na inclusão do banco de registradores em algumas das UPs da arquitetura ADRES. Assim como nas demais arquiteturas, um registrador na saída da UF é utilizado para garantir o tempo na troca de informações.

Nesta arquitetura, a rede é formada pela comunicação entre as UPs vizinhas. Por isso, cada UP conta também com dois Multiplexadores, um para cada entrada da UF, para que seja possível estabelecer a conexão entre elas. Como as UPs que estão na borda do CGRA fazem menos conexão do que as UPs centrais, os multiplexadores das UPs possuem tamanhos diferentes. Na Figura 3.15 é mostrado em detalhes parte da arquitetura ADRES construída.

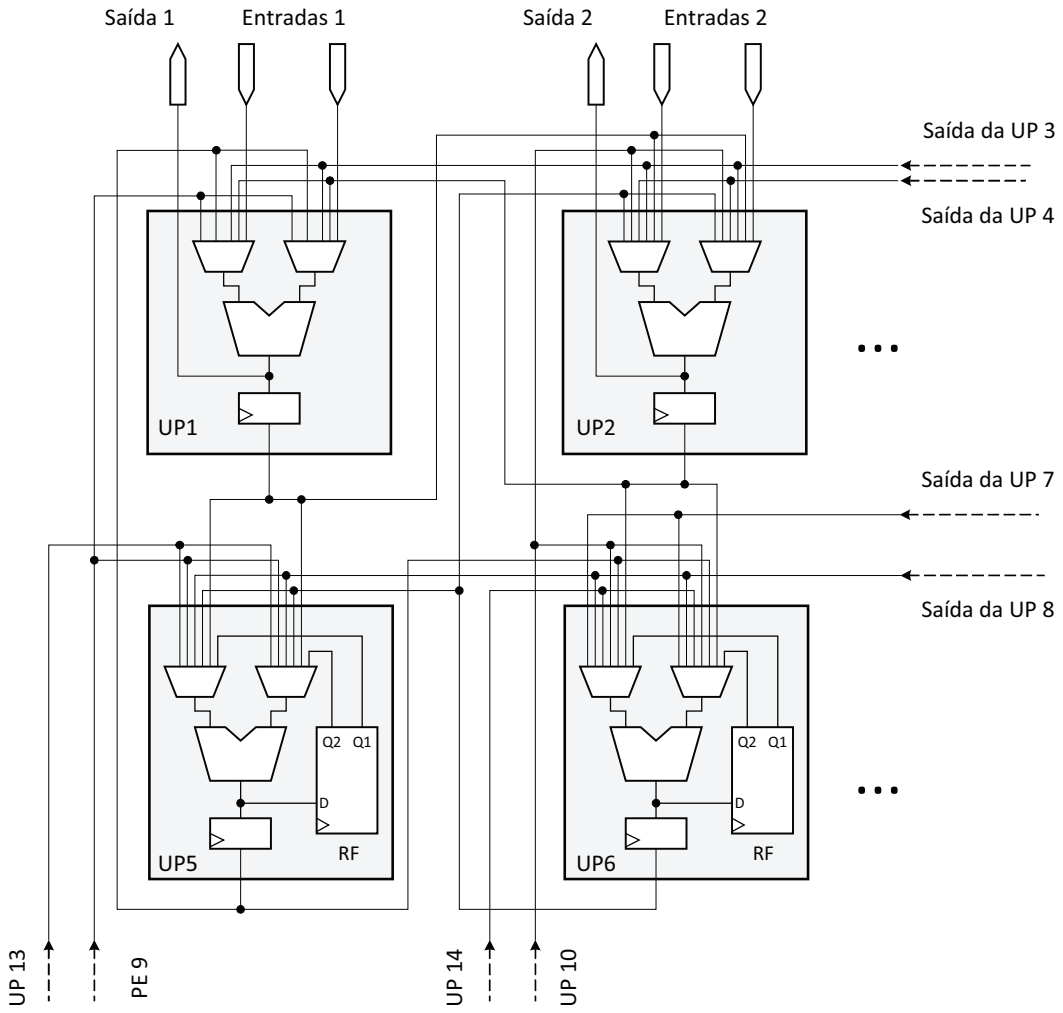
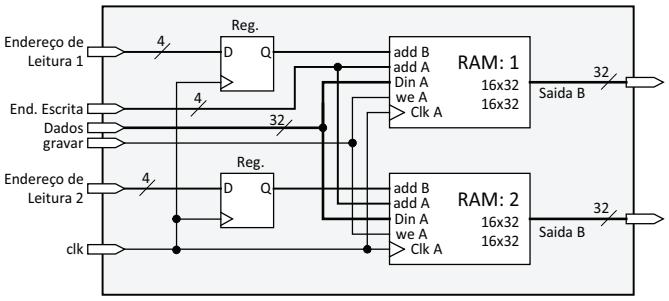


Figura 3.15. Detalhe da implementação da arquitetura ADRES

Nas arquiteturas anteriores, quando se faz necessário o balanceamento do Grafo[2], algumas UPs são utilizadas apenas como registradores. Porém na arqui-

[2]O Balanceamento do grafo de operações que será explicado em detalhes no capítulo 4

tetura ADRES, temos um banco de registradores locais para resolver a questão do balanceamento do grafo. Então, os resultados de uma UF podem ser escrito no banco de registro local ou poderão ser encaminhados para outras UFs. Este Banco de registradores possui a capacidade que permite a leitura em duas posições de memória simultaneamente. Na Figura 3.16 é mostrada em detalhes a síntese e o custo em FPGA do banco de registro usado localmente em cada UP. Estes resultados são para um banco de registradores de 32 bits com 16 posições.



Recursos	
Número de *Slice Registers*	8
Número de *Slice LUTs*	44
Número de *occupied Slices*	13
Número de *LUT Flip Flop pairs used*	52
Número de *RAMB36E1/FIFO36E1s*	0

Figura 3.16. Resultado da síntese do banco de registradores utilizado nas unidades de processamento da arquitetura ADRES simplificada para o FPGA Virtex6 XC6VLX240T.

A comparação entre os recursos do FPGA gastos em cada uma das arquiteturas e a velocidade de máxima de execução depende do número de unidades de processamento, do tipo de UF, do número de unidades que suportam entradas de dados e do tipo de rede. A comparação entre as arquiteturas será apresentada no capítulo 5 desta dissertação.

4. Modulo Scheduling

Este capítulo apresenta a implementação em *hardware* do algoritmo de *Modulo Scheduling*[Mucida et al., 2012; Mucida, 2013] que executa o escalonamento, posicionamento e roteamento de um grafo de operações em uma arquitetura reconfigurável com rede de interconexão *crossbar*. O objetivo é obter uma implementação que seja eficiente e rápida, o suficiente para ser utilizada em tempo de execução. O algoritmo foi descrito como uma máquina de estados para facilitar sua descrição em linguagem de *hardware*.

O algoritmo terá uma área no FPGA, na qual estará implementado como um módulo de configuração. A Figura 4.1 mostra todo o sistema, composto por três componentes principais: a unidade de *modulo scheduling*, a memória de configuração e o CGRA. O módulo de configuração recebe o grafo de entrada, executa o algoritmo em tempo de execução e armazena o resultado gerado na memória de configuração. No CGRA, um número limitado de UPs tem a capacidade de operar sobre valores externos, chamados de UPs de entrada, indicados na figura. Uma vez que a configuração é carregada, o CGRA começa a processar os fluxos de entrada. As UPs realizam a operação para a qual foram configuradas, passando o resultado da operação para a rede de interconexão que repassa o valor para os registradores das UPs de destino.

O algoritmo base [Mucida, 2013] faz o mapeamento em arquiteturas que utilizam uma rede de interconexão *CrossBar*. Essa é uma heurística gulosa derivada do algoritmo de [Ferreira et al., 2011a]. No algoritmo guloso, que realiza o escalonamento, posicionamento e roteamento (EPR), cada aresta do grafo é visitada apenas uma vez, diferentemente do trabalho de [Ferreira et al., 2011a], que durante o escalonamento, os nós eram visitados mais de uma vez. Além disso, enquanto o trabalho em [Ferreira et al., 2011a] focava compiladores JIT (*just-in-time*), a heurística gulosa EPR, foi desenvolvida para ser implementada em hardware. Por isso, ela foi originalmente descrita como um modelo de máquina de estados de Moore utilizando linguagem C.

O algoritmo assume que o ponto de partida é um grafo de operações, similar a outras abordagens [Chen & Mitra, 2012; Park et al., 2008; Hamzeh et al., 2012; Oh et al., 2009; Hatanaka & Bagherzadeh, 2007]. Todas as arestas são percorridas em ordem de busca em largura e cada aresta é processada apenas uma única vez.

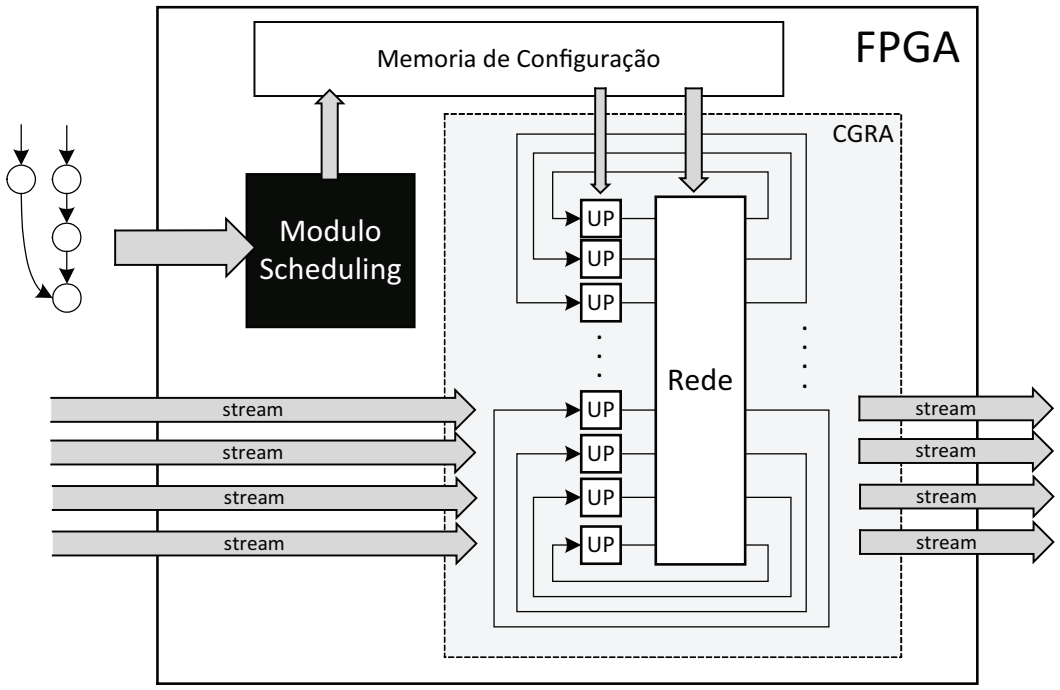


Figura 4.1. Algoritmo de *Modulo Scheduling*em *hardware* junto à um CGRA virtualizado no mesmo FPGA

Inicialmente, para que seja possível realizar um escalonamento do grafo de operações, o grafo deve ter todas as arestas balanceadas. Porém, o grafo recebido pelo algoritmo, pode possuir os caminhos desbalanceados. Os caminhos serão balanceados durante o escalonamento, diferente do trabalho anterior [Ferreira et al., 2011a].

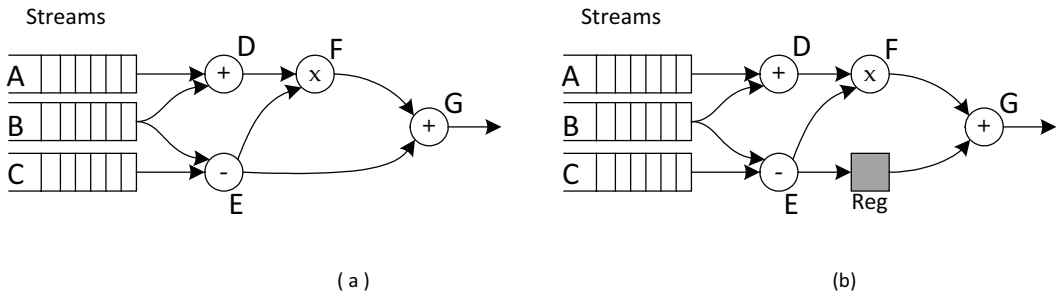


Figura 4.2. (a) Grafo de fluxo de dados, (b) Grafo balanceado

O grafo ilustrado na Figura 4.2 (a) mostra o contexto de aplicações na quais os fluxos de dados são recebidos a cada ciclo do relógio. Na arquitetura alvo, que irá processar o fluxo de dados, as unidades de processamento devem trabalhar de maneira síncrona, por isso, as unidades de processamento do CGRA possuem registradores que fazem com que a informação calculada seja passada para a próxima

unidade apenas a cada ciclo do relógio. Desta forma, os dados são processados em pipeline e por isso os caminhos entre os operadores devem ser equilibrados para que a arquitetura funcione corretamente. Para o exemplo mostrado na Figura 4.2(a), os valores de E chegariam mais cedo do que os dados de F, para serem calculados em G. Para resolver este problema, o balanceamento é feito entre os caminhos através da inserção de alguns registros entre as arestas do grafo, conforme mostra a Figura 4.2(b).

Conforme ilustra a Figura 4.3(a), cada aresta do grafo de fluxo de dados representa uma dependência entre duas operações. Cada Nó é um operador como um multiplicador ou um somador que será posicionado em uma UF dedicada do CGRA. No CGRA, como já mencionado, cada UF tem as saídas e entradas conectadas à rede global e por isso, a saída de qualquer UF poderá ser conectada a uma das entradas de qualquer outra UF, inclusive as dela mesma. Então, as arestas do grafo são implementadas pela rede de interconexão, conforme ilustra a Figura 4.3(b).

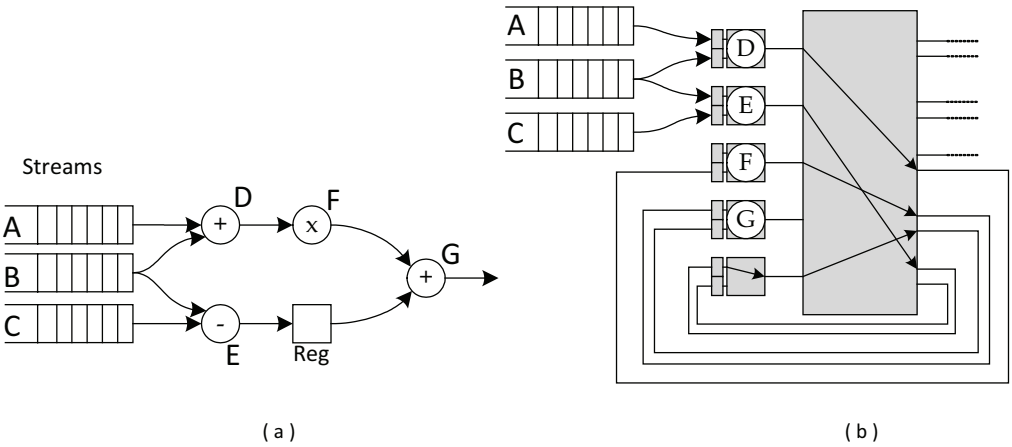


Figura 4.3. (a) Grafo de fluxo de dados balanceado, (b) Mapeamento do Grafo em uma arquitetura com 5 unidades de processamento.

Para manter o sincronismo entre as operações, cada unidade de processamento possui dois registradores, um para cada uma das entradas da UF. Estes registradores também podem ser usados para fazer o balanceamento das arestas. Isso é possível porque a UF possui uma operação que simplesmente passa o valor do primeiro registrador, conforme é mostrado pela última unidade da Figura 4.3(b).

Algumas UFs podem ser usadas como entradas de dados externos e por isso, os *Streams* de dados deverão ser conectados apenas nestas UFs. Na Figura 4.3(b), podemos ver os *streams* A, B e C conectados às UFs que estão executando os nós D e E.

A seguir será explicado com mais detalhes o funcionamento do algoritmo guloso de *Modulo Scheduling*e as estruturas de dados necessárias para execução do algoritmo. Em seguida será apresentado a implementação em **hardware** e um exemplo do mapeamento de um grafo em uma arquitetura reconfigurável.

4.1 Algoritmo EPR guloso

Como já mencionado, o grafo é percorrido em busca em largura e cada aresta processada apenas uma vez. Enquanto as arestas são lidas e os nós vão sendo posicionados e escalonados, podem ocorrer quatro casos, conforme ilustra a Figura 4.4. Para cada um dos casos o algoritmo terá um comportamento distinto.

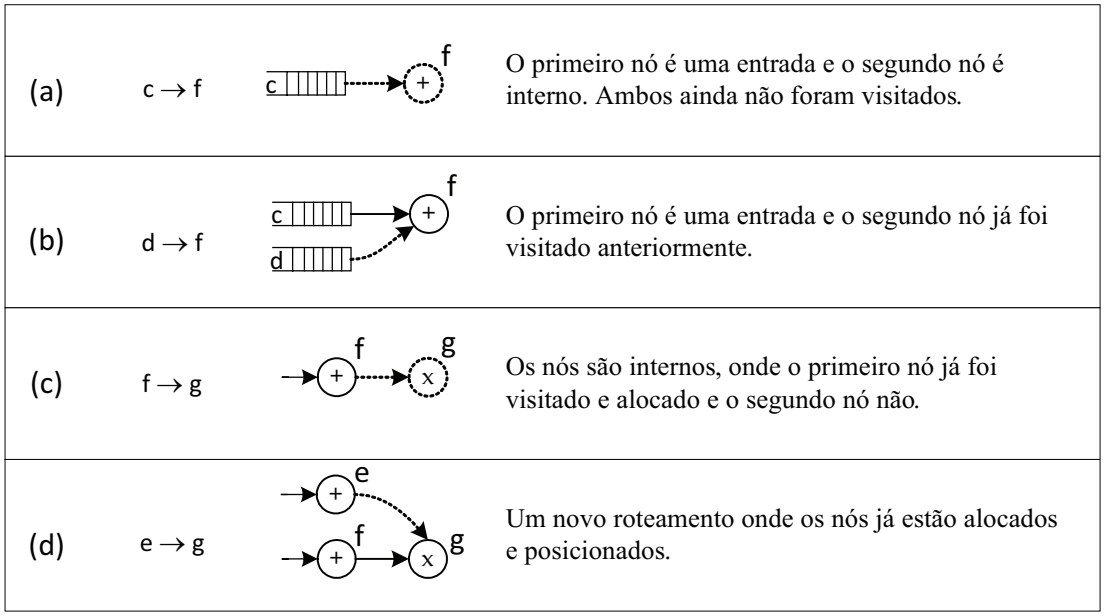


Figura 4.4. Quatro situações encontradas em um grafo de fluxo de dados

O algoritmo EPR possui complexidade $O(n)$ e é composto apenas por estruturas de dados simples, como registros e vetores:

- **Vetor Posicionamento**: mapeia os nós do grafo em uma UF do CGRA. Dado um nó o vetor retorna em qual UF ele está alocado. Caso o nó dado não esteja ainda posicionado, um valor sinaliza que a posição está vazia.
- **Vetor Escalonamento**: mantém o controle do escalonamento temporal de cada operação. Dado um nó, retorna em qual tempo ele está escalonado.
- **Vetor Partição**: armazena em qual partição temporal está cada operação. Dado um nó, retorna em qual partição temporal ele está.

- **Vetor Unidade_Livre**: usado para apontar para a próxima unidade funcional livre em cada partição temporal. Dado uma partição temporal, informa qual a primeira UF livre dessa partição.

- **Vetor Roteamento**: armazena o roteamento entre as UFs do CGRA. São dois vetores bidimensionais, **Roteamento A** e **Roteamento B**, pois cada UF do CGRA possui duas entradas e pode receber valores calculados por outras duas UFs e além disso, cada partição temporal exige um roteamento diferente. Então, em cada um dos dois vetores, dada uma UF e uma partição temporal, informa para qual UF será o roteamento.

O pseudocódigo do algoritmo é mostrado no Algoritmo 1. Para cada arco $X \rightarrow Y$ o algoritmo será repetido. X é o nó de origem, que pode ser tanto um fluxo de entrada ou um operador. O nó Y é o nó de destino, que será sempre um operador.

Conforme mostra as linhas $2 - 8$ do Algoritmo 1, se X é uma entrada e Y ainda não foi visitado, uma nova UF livre será alocada para Y. Neste passo será realizado o posicionamento e o escalonamento para o nó Y. Esta situação está ilustrada na Figura 4.4(a). Caso contrário, se uma UF já foi alocada para Y, o nó X será alocada para essa UF, através da atribuição de entrada, conforme mostra a linha 8 do algoritmo. Este é o caso da Figura 4.4(b).

O código das linhas $10 - 14$ do algoritmo é executado quando X não é uma entrada de dados externa. Nesta implementação, consideramos uma unidade de atraso por operação, por isso, como no caso da Figura 4.4 (c), se X não é uma entrada, então Y deve ser escalonado para um ciclo de relógio após o escalonamento de X e então posicionado na primeira UF livre da próxima partição temporal. Em seguida será feito o roteamento da aresta $X \rightarrow Y$, mostrado na linha 14 do algoritmo.

A última situação acontece quando X e Y são nós internos e já estão posicionados. Esta situação pode ser vista na Figura 4.4(d). Nesta etapa, o algoritmo irá executar as linhas 16-28. Aqui podem ocorrer duas situações: Se o tempo de escalonamento de Y for maior em uma unidade do tempo de X, o roteamento de X para Y será feito e o algoritmo recomeçará, para a leitura de um novo arco. Caso contrário, se o tempo de escalonamento de Y for menor ou igual ao tempo de X, significará que o caminho $X \rightarrow Y$ está desbalanceado. Para balancear o caminho, um registro deverá ser inserido entre o caminho $X \rightarrow Y$. Dessa forma, um registro será colocado na atual posição de Y e Y movido para a primeira UF livre da próxima partição e escalonado com uma unidade de tempo maior. Este ciclo, correspondente

Algorithm 1 Pseudo Código *Modulo Scheduling*

```
1: for cada aresta X → Y do
2:   if X é uma entrada externa de dados then
3:     if Posicionamento[Y] está vazio then
4:       UF = Posicionamento[Y] = Unidade_Livre[Pt0]++;
5:       Escalonamento[Y] = 1;
6:       Partição[Y] = Pt0;
7:     end if
8:     AtribuiEntrada(X,UF);
9:   else if Posicionamento[Y] está vazio then
10:    Pt = Partição[X];
11:    Escalonamento[Y] = Escalonamento[X]+1;
12:    UF = Place[Y] = Unidade_Livre[Pt +1]++;
13:    Partição[Y] = Pt +1;
14:    RoteamentoA[UF][CFG] = Posicionamento[X];
15:  else
16:    corrente = Escalonamento[X]+1;
17:    UF = Posicionamento[Y];
18:    Pt = Partição[Y];
19:    while Escalonamento[Y] < corrente do
20:      UF[Pt].op = Registro;
21:      Posicioanamento[Y] = Unidade_Livre[Pt +1]++;
22:      Escalonamento[Y]++;
23:      RoteamentoA[Posicioanamento[Y]][Pt] = UF;
24:      Pt++;
25:      UF = Posicionamento[Y];
26:      Partição[Y] = Pt;
27:    end while
28:    RoteamentoB[Posicioanamento[Y]][Pt-1] = Posicionamento[X];
29:  end if
30: end for
```

às linhas 20-26 do algoritmo será repetido até que o tempo de escalonamento de Y seja uma unidade maior que o escalonamento de X

A complexidade do algoritmo descrito é $O(N)$, onde N é o número de arestas, pois cada aresta é processada apenas uma vez. Os passos de escalonamento, posicionamento e roteamento são $O(1)$, uma vez que são simples atribuições nos vetores. A complexidade do espaço de memória ocupada pelo algoritmo é $O(N + U \times C)$, onde N é o número de nós do grafo, U é o número de UF na arquitetura alvo, e C, o número de configurações.

4.2 EPR guloso implementado em *hardware*

Para implementar o algoritmo, uma descrição no formato de máquina de estados foi o ponto de partida. A unidade de *Modulo Scheduling*baseado em FPGA consiste de uma máquina de estados Finita (FSM) e de memórias distribuídas. Os modernos FPGAs comerciais, incluem módulos de memórias embutidas, que permitem reduzir o custo de implementação, uma vez que a FSM ocupa poucas LUTs.

O algoritmo EPR em *hardware* foi dividido em 3 módulos principais: Uma máquina de estados, um módulo para cálculo do escalonamento e um módulo para o cálculo e armazenamento das informações de posicionamento e roteamento. Um módulo de memória é utilizado para armazenar o arquivo de fluxo de dados que será processado. A estrutura geral do algoritmo pode ser vista na Figura 4.5.

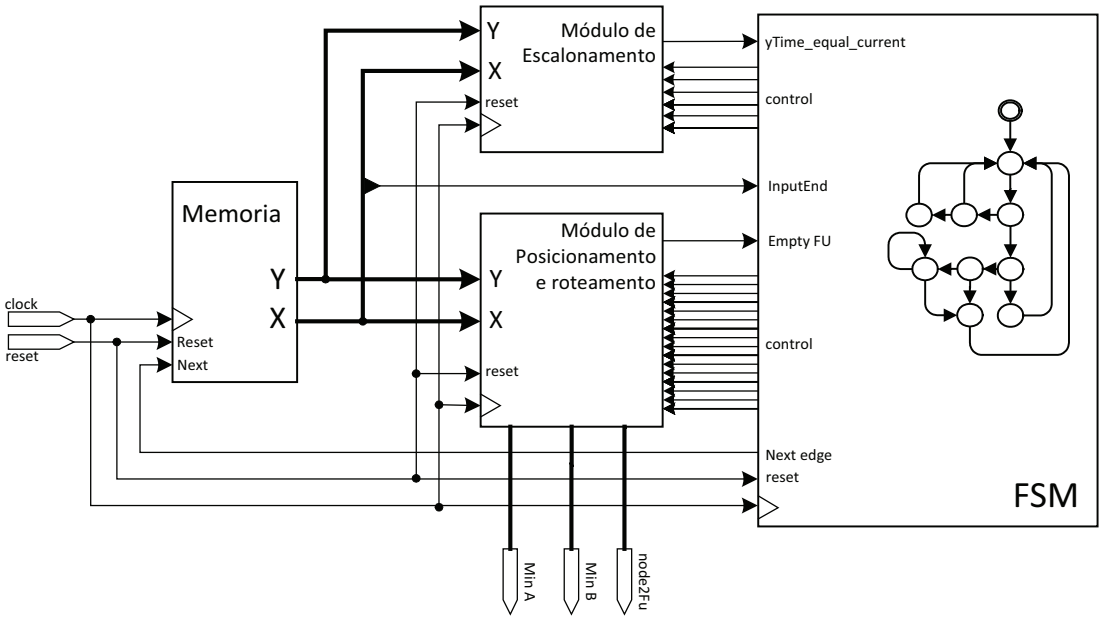


Figura 4.5. EPR em hardware. FSM, módulo de escalonamento e módulo de posicionamento e roteamento

Em cada linha da memória é armazenado um arco do grafo na forma $X \rightarrow Y$. Quando a entrada *Next* da memória, controlada pela FSM, passa do estado baixo para alto, um novo arco $X \rightarrow Y$ será entregue nos barramentos X e Y pela memória. O fim do arquivo é sinalizado na entrada *InputEnd* da FSM, utilizando o bit mais significativo de X. O sinal de*reset* do *Modulo Scheduling*fará a FSM voltar para o estado inicial e e reiniciará a posição de leitura na memória que armazena o grafo. Além disso, o *reset* também fará com que os registradores e as memórias dos dois

módulos (módulo de escalonamento e módulo de Posicionamento e roteamento) sejam limpos.

Os Sinais *Empty UF* e *Current_ equal_ bTime* são sinais utilizados como condições de transição pela máquina de estado. O sinal *Empty UF* do módulo de Posicionamento e roteamento, indica se uma UF selecionada já está alocada. Esta informação pode ser vista no pseudocódigo do Algoritmo 1, nas linhas 2, 3 e 9. A informação *Current_ equal_ bTime*, do módulo de escalonamento, indica se o tempo de escalonamento de Y é igual ao tempo corrente. Esta informação é usada pelo algoritmo e foi descrita no pseudocódigo do Algoritmo 1 na linha 19 (**while** Escalonamento[B] $<$ corrente **do**).

4.2.1 FSM

Uma visão geral da máquina de estados, utilizada no controle do EPR é mostrada na Figura 4.6. Em média, três ou quatro estados da máquina são percorridos para cada aresta do fluxo de dados.

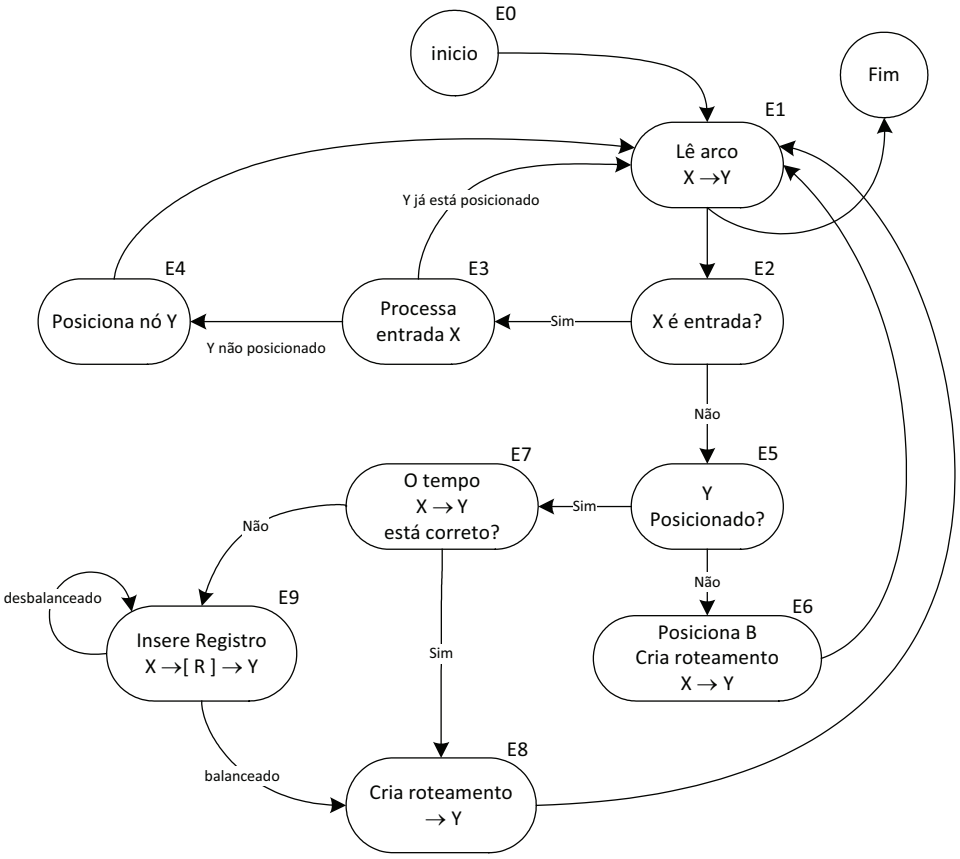


Figura 4.6. Máquina de estados finitos do algoritmo EPR

Por causa das restrições de leitura e escrita nas memórias e registadores, o algoritmo foi dividido em vários estados, com cinco caminhos possíveis. A FSM possui 11 estados, incluindo o estado inicial e o estado final. O comportamento de cada estado é listado a seguir:

- O estado $E0$ é o estado inicial. Nele acontece o *reset* dos registradores e a limpeza das memórias.
- No estado $E1$ acontece a leitura dos arcos do grafo de fluxo de dados. A cada iteração um novo arco $X \rightarrow Y$ é lido. Quando o arquivo de leitura com o grafo chega ao fim, a FSM segue para o estado final e o algoritmo termina.
- O estado $E2$ verifica se o Nó A já foi alocado. Caso contrário, significa que esse Nó deverá ser posicionado como uma entrada e neste caso o fluxo da máquina seguirá para o estado $E3$. Se o Nó já estiver sido alocado o fluxo seguirá para o estado $E5$. Neste estado, também será verificado se o arquivo de leitura chegou ao fim.
- O estado $E3$ é responsável pelo processamento do Nó de entrada.
- No estado $E4$ ocorre o posicionamento do nó Y, que neste caso será um Nó que receberá uma das entradas do fluxo de dados. Após isso, a sequência de máquina retorna ao estado inicial $E1$ e uma nova aresta será processada.
- O estado $E5$ é responsável em verificar se o nó Y já foi posicionado anteriormente. Caso verdadeiro, o fluxo segue para o estado $E7$. Se não, a máquina vai para o estado $E6$.
- No estado $E6$ corre o posicionamento do nó Y como sendo um nó interno. Neste caso, ocorre também o roteamento entre o Nó X e o Nó Y ($X \rightarrow Y$).
- No estado $E7$ é feita uma verificação do balanceamento dos caminhos do grafo de fluxo de dados. Esta etapa é necessária caso um Nó interno Y já esteja posicionado. Então, se o arco $X \rightarrow Y$ estiver balanceado, ou seja, se o tempo de escalonamento de Y for uma unidade maior que o tempo de X, a máquina seguirá para o estado $E8$ para o roteamento, caso contrário, seguira para o estado $E9$.
- O estado $E8$ realiza o roteamento para o Nó Y e em seguida retorna para estado $E1$, recomeçando o processo para uma nova aresta.

- No estado $E9$ acontece o balanceamento das arestas. Como dito antes, caso o tempo de escalonamento do nó Y não for uma unidade maior que o tempo de X, um roteamento direto não poderá ser realizado, e assim um registro deverá ser adicionado entre os nós X e Y para balancear o fluxo. Este estado será repetido e um novo registro será inserido enquanto for necessário. Quando um registro é inserido, é realizado o roteamento entre o nó X e o primeiro registro ou um roteamento entre os registros adicionados. O roteamento final, ou seja, entre o ultimo registro adicionado e o Nó Y será realizado no estado $E8$.

4.2.2 Mapeamento da FSM para *hardware*

A heurística EPR foi desenvolvida para ser implementada em hardware mas a sua validação, apresentada em [Mucida, 2013] foi descrita utilizando linguagem C, como um modelo de máquina de estados de Moore, que tem a saída exclusivamente em função do estado atual. Na implementação da FSM em linguagem C, cada estado é implementado por uma cláusula **case** que executa uma função sobre as variáveis e vetores do algorítimo. Neste tipo de implementação, as variáveis devem ser vistas como registradores e os vetores como memórias distribuídas.

Como o objetivo é conseguir um algorítimo que possa ser convertido para uma implementação em VHDL, deve-se ter em mente que todas as operações em cada estado serão executas em hardware paralelamente. Por isso, deve-se garantir que apenas uma das duas operações (leitura e escrita) seja feita em cada estado para um determinado registrador ou memória.

A simulação do algorítimo EPR feita por [Mucida, 2013] apresentou ganhos em relação à proposta JIT de [Ferreira et al., 2011a], porém esta simulação, utilizando linguagem C considera apenas o número de ciclos executados, desconsiderando o tempo do ciclo. Por isso o mapeamento da heurística EPR em C para VHDL foi realizado neta dissertação para complementar os resultados apresentados por [Mucida, 2013] e validar a solução.

Se as restrições citadas anteriormente nesta seção forem satisfeitas para a implementação do algoritmo, a tradução de C para VHDL, poderá ser feita conforme o exemplo da Figura 4.7(a) que descreve uma máquina com dois estados ($E0$ e $E1$) que opera sobre quatro variáveis (a, b, c e d) e um vetor (M). Quatro registradores e uma memória são necessários para este exemplo, conforme ilustra a Figura 4.7(b). Um multiplexador é necessário na entrada de a, já que este registrador recebe o valor de b no estado $E0$ e o valor de c no estado $E1$. Neste exemplo, a memória M sempre recebe os valores de d e o registrador b sempre recebe os valores de M, por

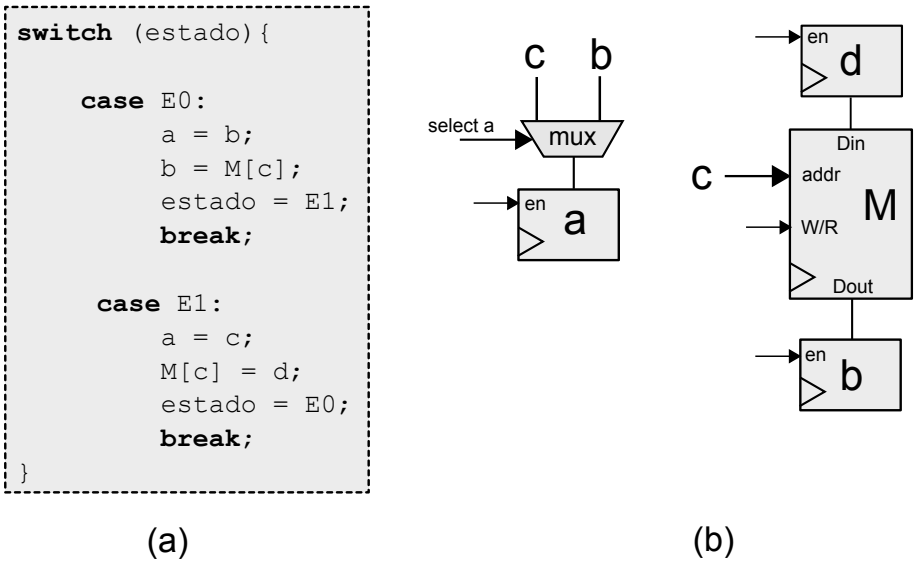


Figura 4.7. Mapeamento do código C para *hardware*

isso, a conexão entre estes componentes é feita diretamente.

Na Figura 4.7(b), as setas de entrada ligadas em cada componente (no **select** do multiplexador, no **en** dos registradores e no **W/R** da memória), representam o controle da FSM sob a estrutura. Em cada estado, um determinado conjunto de componentes estarão habilitados para gravação, dependendo das operações necessárias no estado.

A conversão do algoritmo EPR em C para VHDL dividiu a estrutura de dados necessária em dois módulos diferentes (módulo de escalonamento e módulo de posicionamento e roteamento), conforme dito anteriormente.

4.2.3 Módulo de escalonamento

Os detalhes do módulo de escalonamento estão ilustrados na Figura 4.8. O módulo possui dois registradores, uma memória, dois somadores, dois multiplexadores e um comparador. O número de bits dos registradores, dos somadores e o tamanho da memória podem variar dependendo do tamanho máximo dos grafos que se pretende manipular.

A memória **NodeTime** armazena o tempo de escalonamento é usada apenas no decorrer do algoritmo, ou seja, os valores finais armazenados não serão usados para configurar o CGRA e por isso não há necessidade de um meio de acesso externo à essa memória.

A saída **yTime_equal_current** do módulo, é usada pela FSM como uma

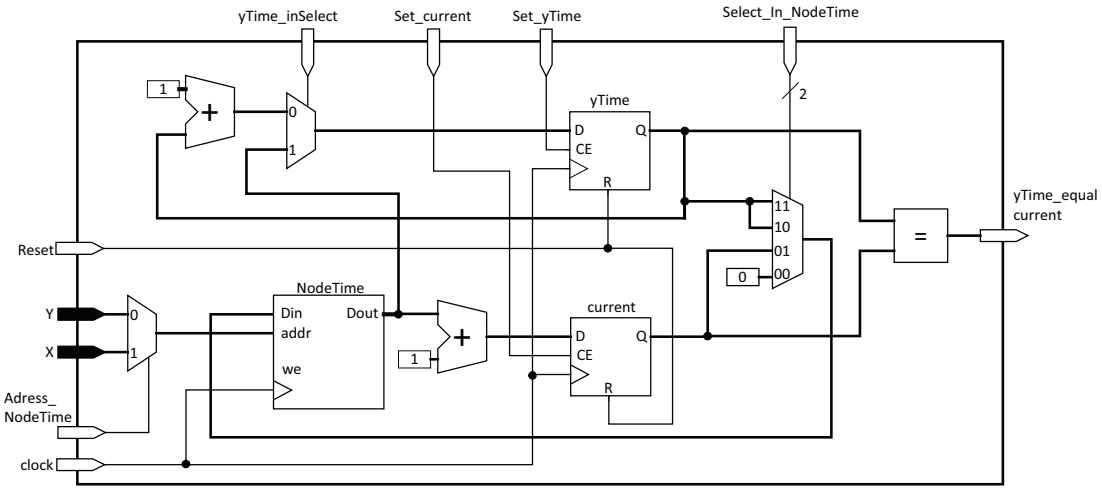


Figura 4.8. Módulo de Escalonamento

condição de transição e indica se **y_time** é igual à **current**, retornando nível alto caso seja verdade. As entradas de controle (bTime_inSelect, Set_current, Set_bTime, select_in_NodeTime e Adress_NodeTime) são conectadas à saída de controle da FSM e são usadas por ela para selecionar como os dados internos ao módulo são manipulados.

4.2.4 Duas implementações do algoritmo

O *Modulo Scheduling*apresentado foi dividido em duas implementações. A primeira não trabalha com partições temporais, por isso, o mapeamento só é possível se o CGRA possuir unidades suficientes para comportar todos os nós do grafo. A segunda implementação é capaz de mapear um grafo maior que a arquitetura e faz isso criando partições temporais.

Para cada um dos dois algoritmos, o módulo de escalonamento permanece o mesmo, porém, as funções de controle da FSM e os módulos de posicionamento de roteamento são diferentes.

Nas próximas seções será explicado com mais detalhes o funcionamento de cada um dos **módulos de posicionamento e roteamento** e a diferença nos sinais de controle das máquinas de estados nos dois algoritmos.

4.2.4.1 Implementação sem partições temporais

As operações executados pelo módulo em cada estado da FSM são mostradas na Figura 4.9.

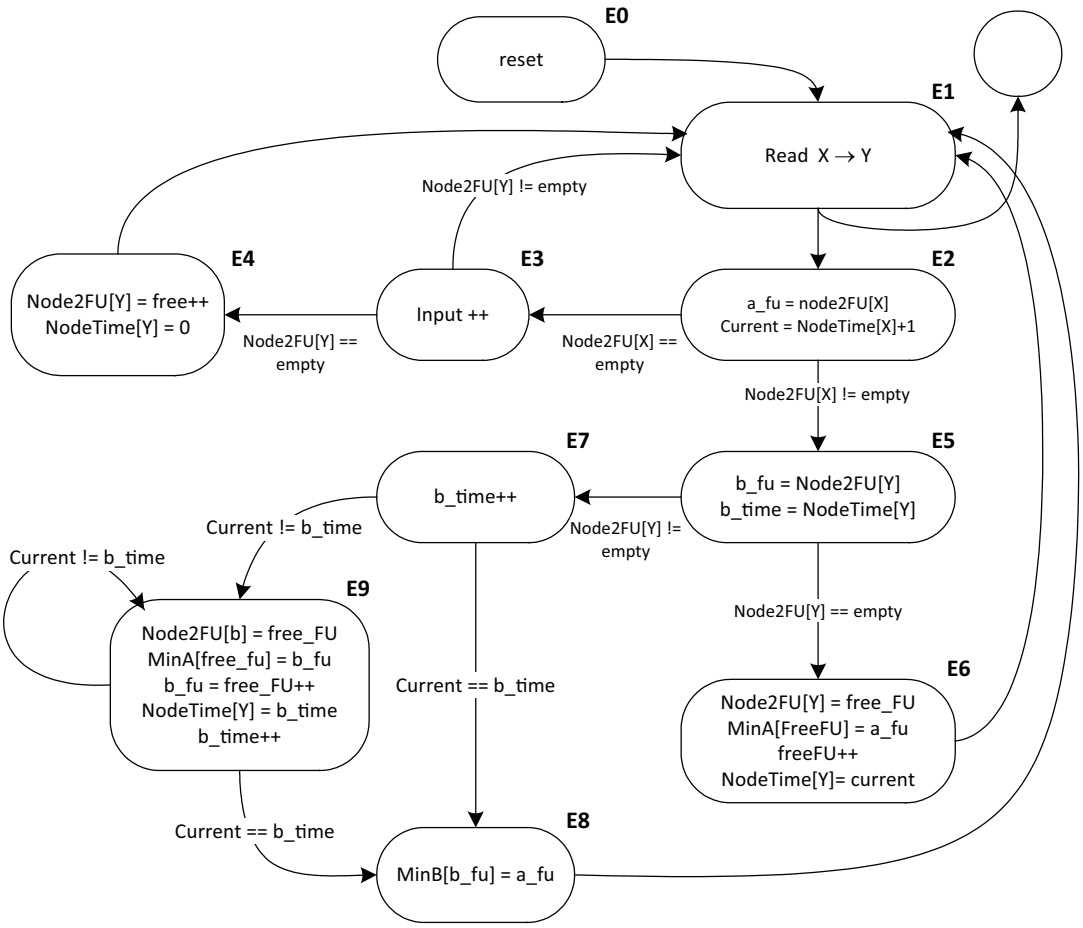


Figura 4.9. Operações na máquina de estados do algoritmo que não faz uso de partições temporais

A máquina de estados (FSM) não trabalha com partições temporais e por isso as memórias utilizadas no módulo de posicionamento e roteamento são unidimensionais. O desenho das estruturas interna do módulo pode ser visto na Figura 4.10.

O módulo utiliza 3 registradores simples com *reset*, um somador, 3 multiplexadores e 3 módulos de memória.

As memórias **Min A** e **Min B**, uma para cada entrada da UF, são usadas para a etapa de roteamento. Elas podem ser acessadas externamente e seus valores são utilizados na configuração de cada uma das redes do CGRA.

A memória **Node2FU** armazena a tabela de posicionamento, onde um Nó é mapeado para uma UF. No início do funcionamento do algoritmo, ou quando o módulo é reiniciado, é necessário que todas as posições da memória estejam marcadas como vazias, ou desocupadas. Essa informação é usada pela FSM através da saída **FU_empty** do módulo, como uma condição de transição para decidir o caminho do algoritmo na máquina de estados.

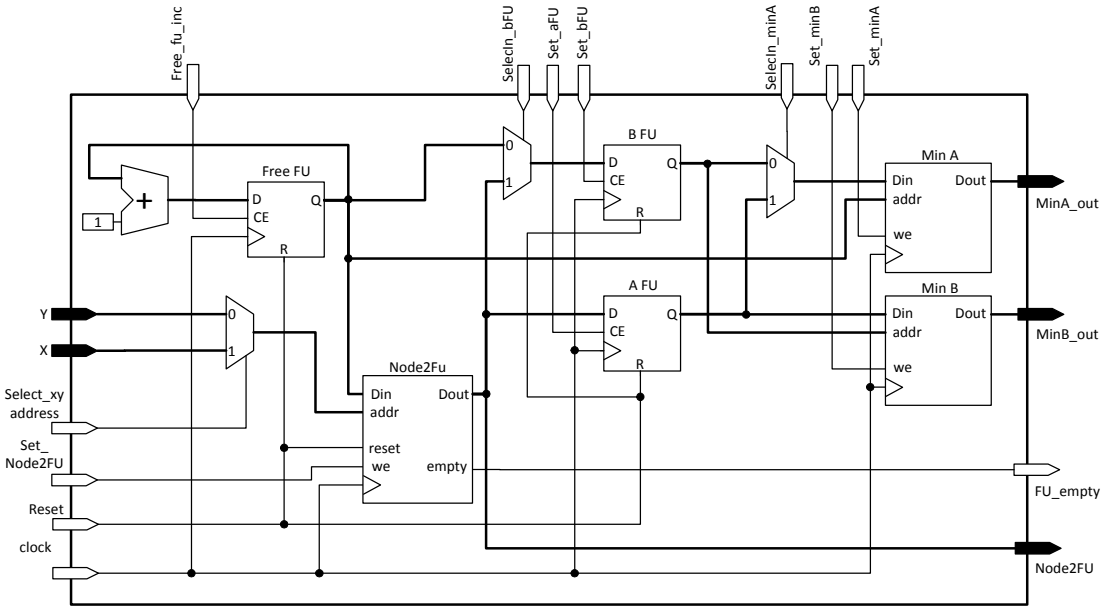


Figura 4.10. Módulo de posicionamento e roteamento para o algoritmo que não usa partições temporais

Quando um componente de memória é necessário, o FPGA utiliza blocos de memórias disponíveis na sua arquitetura. Porém, para que o algoritmo funcione com eficiência, é importante que o *reset* da memória seja feito em um único ciclo de relógio. No entanto, memórias que possuem a característica de *reset*, feitos a partir de um único ciclo de relógio são muito caras e não são possíveis de serem sintetizadas a partir dos blocos de memória disponíveis no FPGA. Por isso, a solução para o problema foi sintetizar uma única memória a partir de dois tipos de memórias diferentes.

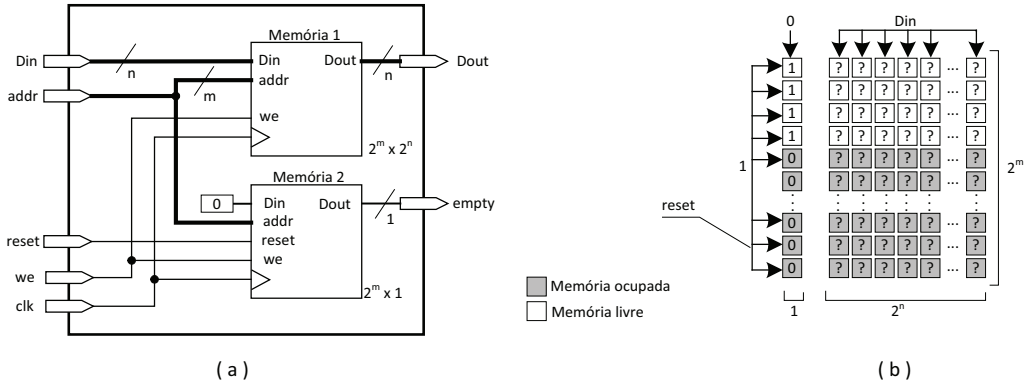


Figura 4.11. Detalhe da construção da memória com Reset.

A Figura 4.11(a) ilustra em detalhes o módulo de memória **Node2FU** Note que existem dois módulos de memórias: Memória 1 e 2. A **Memória 1** é um

módulo de memória comum, sintetizado de modo a aproveitar alguns dos módulos de memórias disponíveis no FPGA. A **Memória 2** possui um *reset*, capaz de modificar todos os valores da memória para 1 em um único *clock*. Note que a **Memória 2** possui apenas uma coluna. Desta forma, ela será usada apenas para informar quais linhas da **memória 1** estão livres. Então se a $linha_i$ da **Memória 2** estiver marcada com 1, significa que a $linha_i$ da **Memória 1** está desocupada, conforme ilustrado na Figura 4.11(b).

As duas memórias estão ligadas em paralelo, com o mesmo valor de endereço, sinal de relógio e sinal para gravação. Assim quando um valor, a partir da entrada **Din** é gravado em um endereço na **Memória 1**, o valor 0 é gravado no mesmo endereço na **Memória 2**, indicando que agora aquela posição de memória está ocupada.

4.2.4.2 Implementação com Múltiplas configurações

Na seção anterior, a implementação assume que a arquitetura tem unidades suficientes para mapear todos os operadores de um fluxo de dados na arquitetura em uma única configuração. Porém, um algoritmo de *Modulo Scheduling*deve ser capaz de mapear grafos que sejam maiores que a arquitetura alvo. Vamos considerar grafo fluxo de dados representado na Figura 4.12(a), onde A, B e C são fluxos de dados de entradas, D e E são operações que utilizam os dados do fluxo de dados, F e G são operações que recebem dados dos operadores anteriores. O registro será incluído no decorrer do algoritmo. Além disso, cada nó tem um registro de saída que não está mostrado para facilitar a explicação.

O grafo citado será a entrada do algoritmo e será percorrido em largura. A arquitetura utilizada no exemplo possui apenas 4 unidades de processamento. Por isso, pelo menos, duas partições temporais serão necessárias (veja Figura 4.12(b)).

As entradas das UFs são posicionadas na primeira partição P_0. Ao processar uma aresta como a $D \rightarrow F$, como a UF_D pertence a P_0, uma UF em P_1 será alocada para F. Então, quando a partições de processamento P_i, o operador alvo será posicionado na partição $P_i + 1 = (P_i + 1) mod P$, onde p é o número máximo de partições.

Como o grafo é percorrido em largura, as operações D e E são posicionadas primeiro. Em seguida F é posicionado na segunda configuração e serão feitos os roteamentos $E \rightarrow F$ e $D \rightarrow F$. O posicionamento do operador G, em um primeiro momento, pode acontecer na segunda configuração, caso o arco $E \rightarrow G$ aconteça primeiro. Porém o processamento do último arco, $F \rightarrow G$ pelo algoritmo, fará com

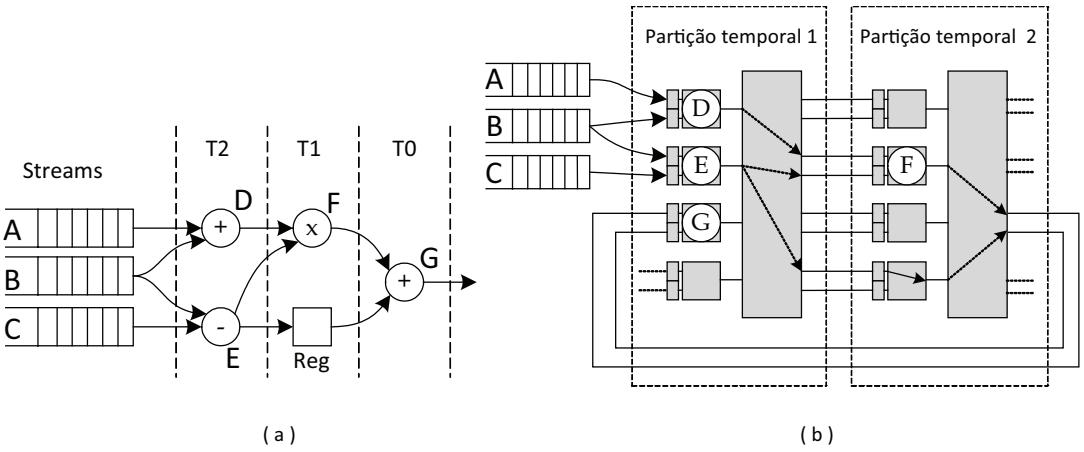


Figura 4.12. Mapeamento com Múltiplas partições temporais

que o operador G seja movido para uma unidade da primeira configuração. A antiga unidade ocupada por G, será agora usada como registro para balancear o caminho entre os operadores E e G.

Ao final do mapeamento, os nós D, E e G são mapeados na primeira partição temporal (Partição temporal 1). O nó F e o registro são mapeados na segunda partição temporal (Partição temporal 2), conforme ilustra a Figura 4.12(b). Desta forma, durante a execução do CGRA, as duas partições temporais vão se alternando. Por isso, o intervalo inicial (II) para inserir o *stream* de dados será 2, ou seja, a cada 2 ciclos, novos dados são injetados no CGRA. O primeiro resultado levará 3 ciclos, ou seja, a latência será 3. O ILP será de pelo menos 2.

Diferentes UFs na mesma configuração poderão processar os dados em tempos diferentes. Por exemplo, o nó D computa os dados no tempo T_2 (o terceiro elemento do fluxo), e o nó G posicionado na mesma configuração computa os dados do tempo T_0 (primeiro elemento) como mostrado na Figura 4.12(a).

Este simples exemplo, é utilizado para demostrar o comportamento do algoritmo quando o tamanho do grafo de fluxo de dados é maior que o CGRA. Se o CGRA tem 64 UFs, Será possível mapear 100 nós do grafo de fluxo de dados em duas partições temporal. O II será de 2 e a ILP 50 e mais UFs serão usadas.

Podemos notar então que o algoritmo poderá lidar com mais de uma partição temporal usando poucos recursos de hardware. A FSM é semelhante à implentação que não faz uso de partições temporais, porém alguns operações a mais em cada estados da FSM foram adicionadas, conforme ilustrado na Figura 4.13.

O escalonamento utiliza um módulo de memória maior para armazenar a partição temporal ou de configuração. Por causa das partições temporais, os registrador

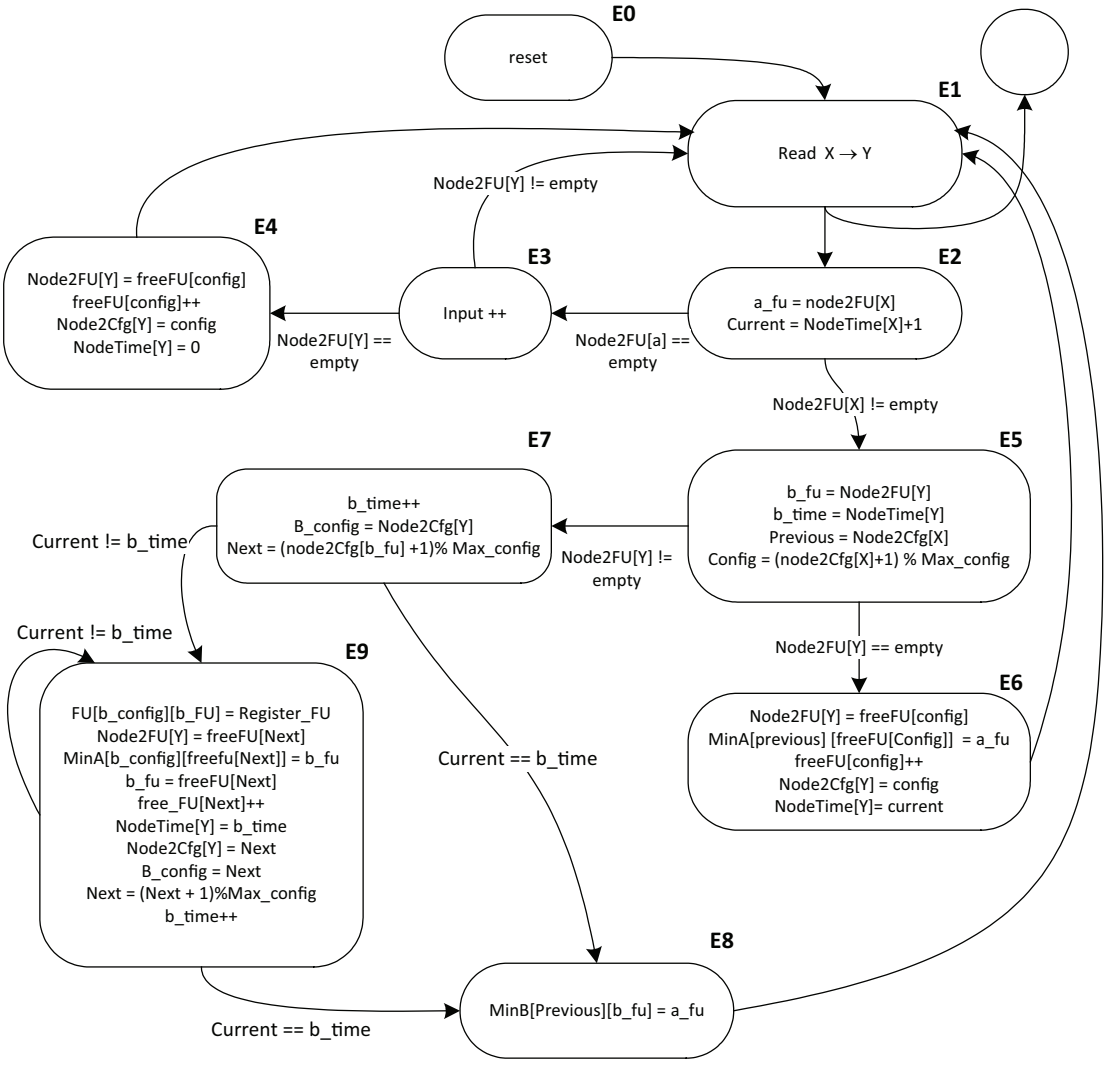


Figura 4.13. Operações na máquina de estados do algoritmo que faz uso de partições temporais

FreeFU, da implementação anterior, usado para apontar para a próxima UF livre, foi substituído por uma memória que armazena a unidade livre por partição temporal. Uma nova memória também foi adicionada, para informar em qual configuração está cada Nó. Além disso, novos registradores também tiveram que ser acrescentados.

O roteamento precisa de dois módulos de memórias para cada partição temporal ou uma memória bidimensional. A Figura 4.14 ilustra os recursos do módulo de posicionamento e roteamento. As memórias **Node2Fu** e **Node2Cfg** armazenam o posicionamento e a configuração por nó. A **FreeFU** é usada durante o posicionamento e duas memoria de duas dimensões são usadas para armazenar a tabela de roteamento (**Net A** e **B**).

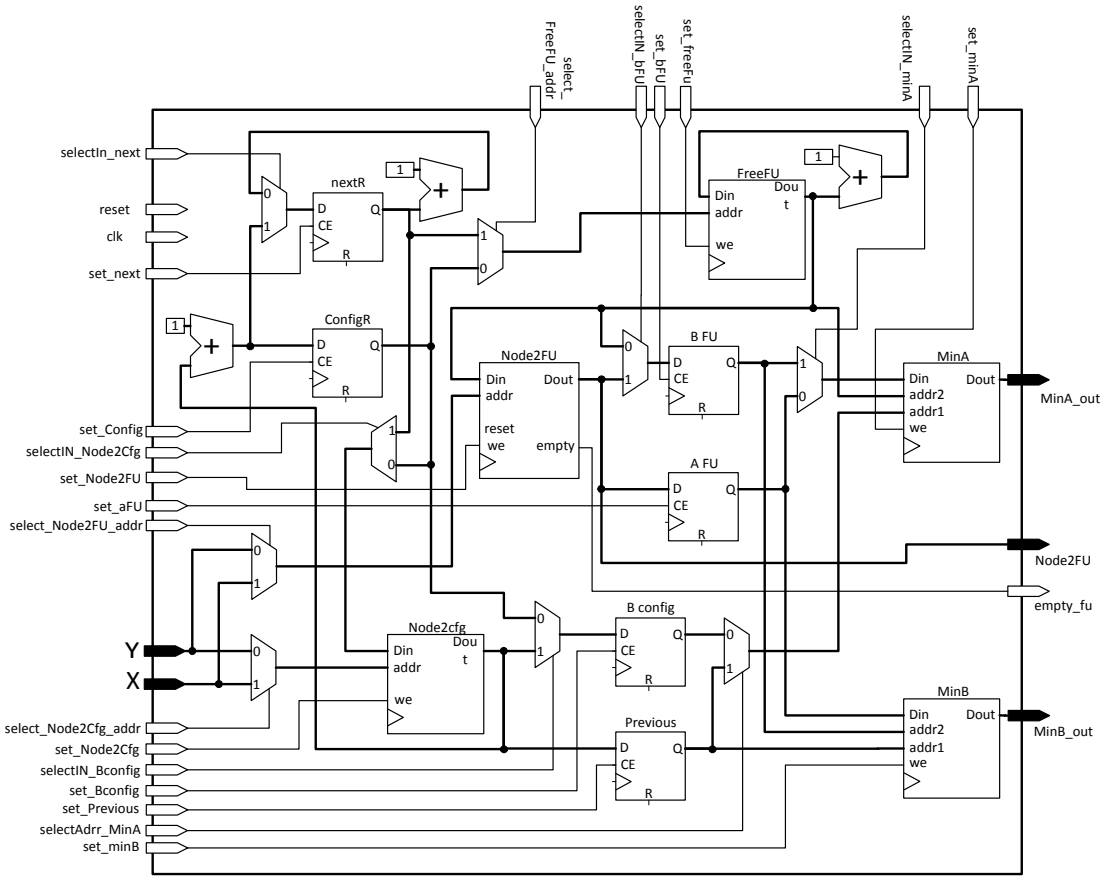


Figura 4.14. Módulo de posicionamento e roteamento para múltiplas configurações

4.2.5 Exemplo de mapeamento de um grafo

O exemplo a seguir mostra a execução do mapeamento de um grafo de fluxo de dados para um CGRA. Para facilitar a explicação, vamos considerar aqui que o CGRA possui unidades suficientes para comportar o grafo, inclusive os registros necessários para fazer o balanceamento. Por isso, neste exemplo usaremos como base a implementação que não faz uso de partições temporais. Para começar, temos o grafo ilustrado na Figura 4.15(a), onde A , B, C e D são fluxos de dados de entrada e E, F, G e H são operadores. Este grafo é usado como entrada da FSM por um arquivo onde cada aresta é armazenada na forma de um par de nós, conforme ilustrado na Figura 4.15(b). Note que este é um grafo desbalanceado e por isso dois registros serão necessários para fazer o balanceamento entre as arestas. A Figura 4.15(b) ilustra o resultado do balanceamento do grafo que será feito no decorrer do algoritmo.

Neste exemplo passo a passo, considere a Figura 4.16. A descrição de entrada

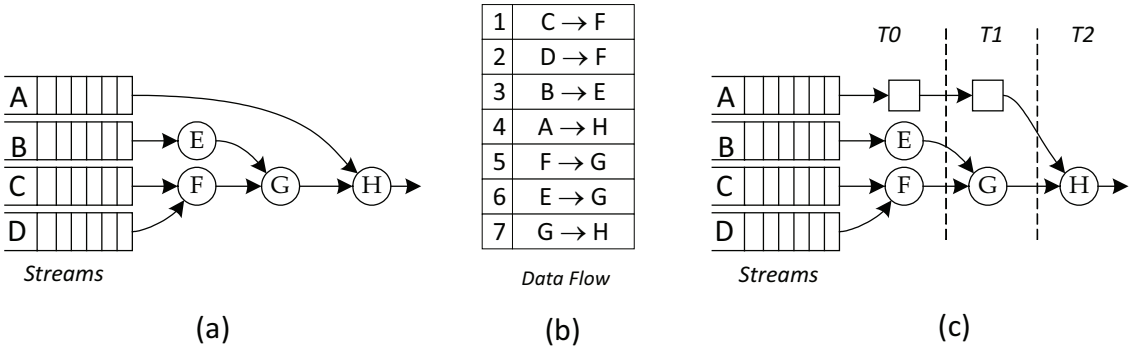


Figura 4.15. (a) Grafo utilizado no exemplo de mapeamento. (b) Arquivo que representa o grafo. (c) Resultado do balanceamento de grafo que será feito pelo algoritmo

mostrada na Figura 4.16(a), terá todas as aresta percorrida. Neste caso, a primeira aresta processada é a aresta ($C \rightarrow F$), marcada em preto. O algoritmo fará a leitura da aresta no estado E_1 e em seguida, verifica no estado E_2 que o nó C não foi posicionado e que por isso, trata-se de uma entrada de dados. No estado E_3, é verificado que o nó F também não está posicionado e por isso, no estado E_4, será feito o posicionamento do nó F na primeira UF livre, apontada pelo apontador **Free_FU**. A Figura 4.16(b) ilustra o caminho $E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4$, percorrido pela máquina de estados para a atual aresta. O resultado final da memória de escalonamento (**NodeTime**) e de posicionamento (**Node2FU**), está ilustrado na Figura 4.16(c). A partir das memórias podemos ver que o nó F foi escalonado para o tempo 0 e posicionado na UF_1. Neste momento, ainda não foi feito nenhum roteamento e por isso, as memórias **MinA** e **MinB** permanecem vazias. O CGRA utilizado neste exemplo possui 4 unidades de processamento e está ilustrado na Figura 4.16(d) que apresenta o resultado do posicionamento do nó F para este primeiro passo.

A Figura 4.17 ilustra o resultado do processamento do arco $D \rightarrow F$. Neste caso, foi é verificado no estado E_2 que o nó D é uma entrada, pois este nó ainda não foi posicionado. Em seguida, no estado E_3 é verificado que o nó F já está posicionado e por isso, apenas a entrada D será computada. Portanto, o apontador de unidade livre e as memórias permanecem da mesma forma e o caminho percorrido pela FSM será $E_1 \rightarrow E_2 \rightarrow E_3$.

O processamento da aresta $B \rightarrow E$ e da aresta $A \rightarrow H$ é semelhante ao processamento da primeira aresta do exemplo ($C \rightarrow F$). Então, conforme ilustra a Figura 4.18, que apresentado o resultado do processamento destas duas arestas, o

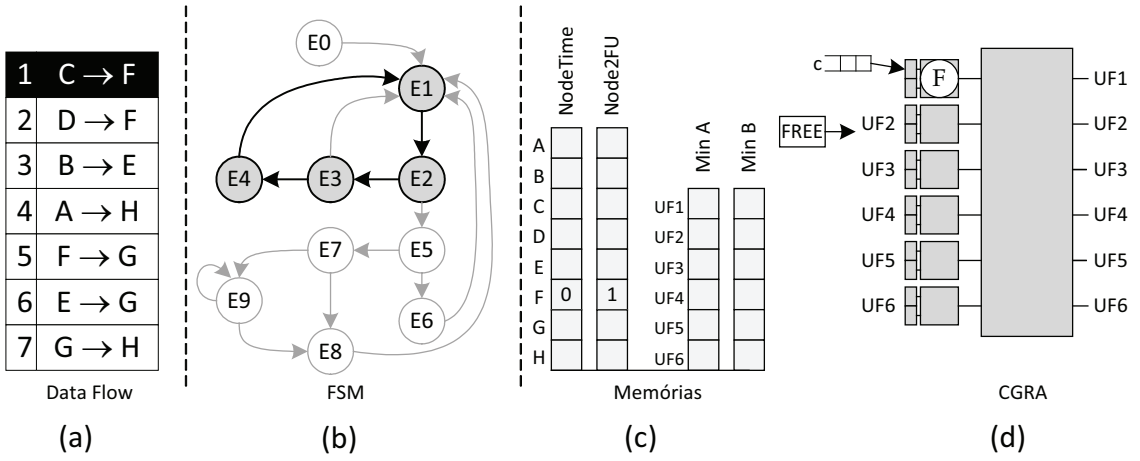


Figura 4.16. Exemplo de mapeamento. Passo 1

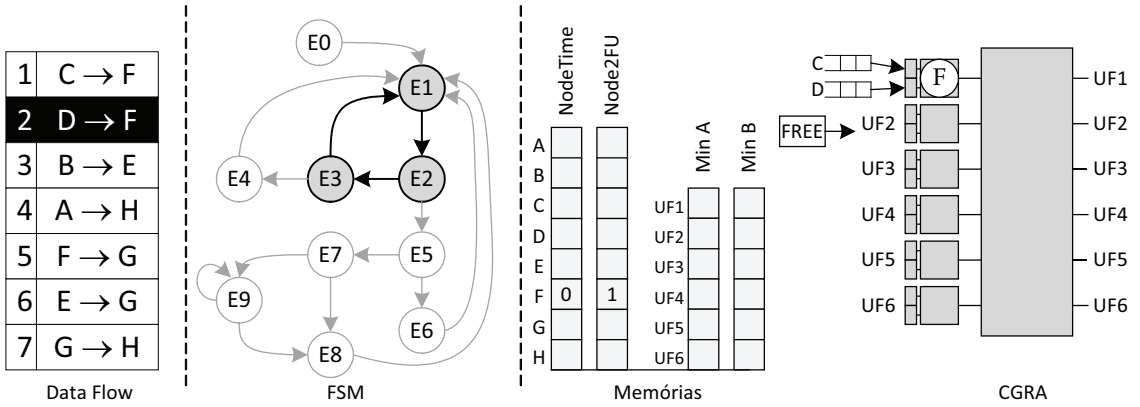


Figura 4.17. Exemplo de mapeamento. Passo 2

nó B e A serão processados como sendo entradas de dados e os nós E e H serão escalonados no tempo 0 e posicionados na UF_2 e UF_3 respectivamente.

Na Figura 4.19 vemos o processamento da aresta $F \rightarrow G$, que é um caso onde os dois nós são internos. Neste exemplo, é verificado no estado E_2 que F já está posicionado e escalonado e por isso ele não pode ser uma entrada. Por isso, a máquina começa um caminho diferente, passando agora para o estado E_5, onde é verificado que o nó G ainda não foi posicionado. Então no estado E_6 é feito o posicionamento e o escalonamento de G e também será feito o roteamento da UF_1 para UF_4, onde estão posicionados os nós F e G respectivamente. No estado E_6 o roteamento é feito utilizando a memória **MinA**, por isso, o roteamento entre a UF_1 para a UF_4 é feito a partir da saída da primeira UF para o primeiro registrador de entrada da segunda UF.

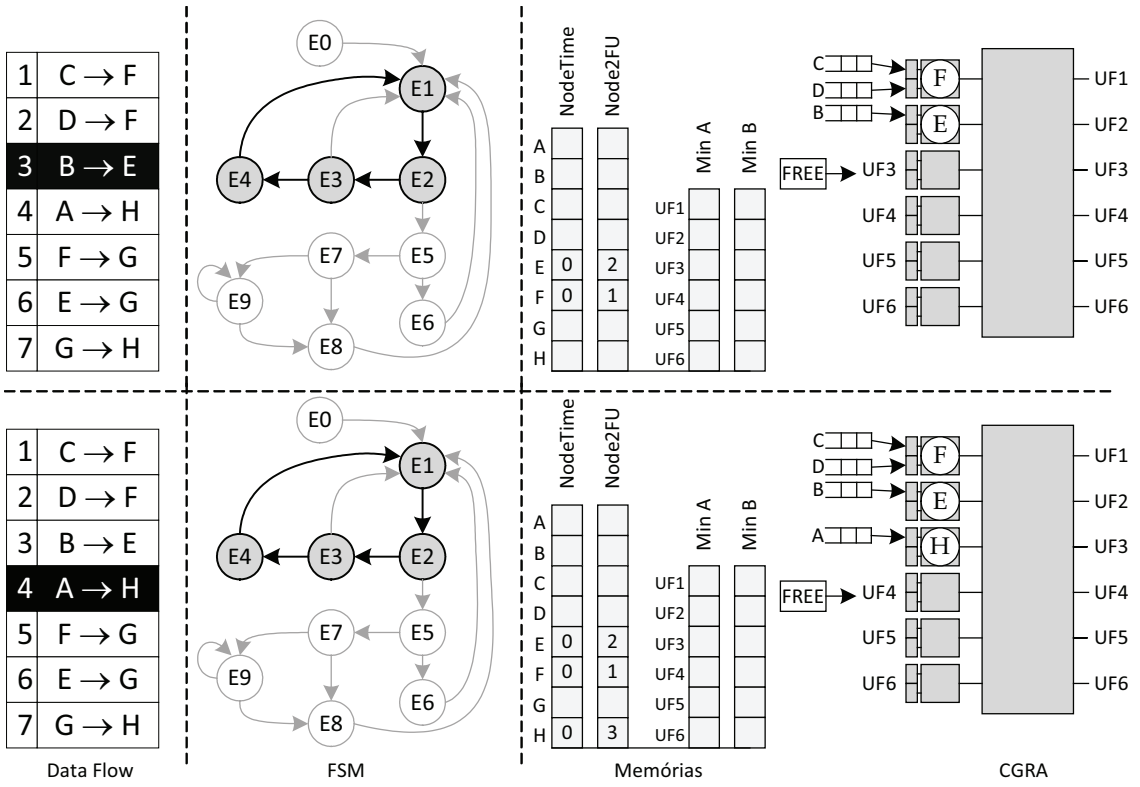


Figura 4.18. Exemplo de mapeamento. Passo 3 e passo 4

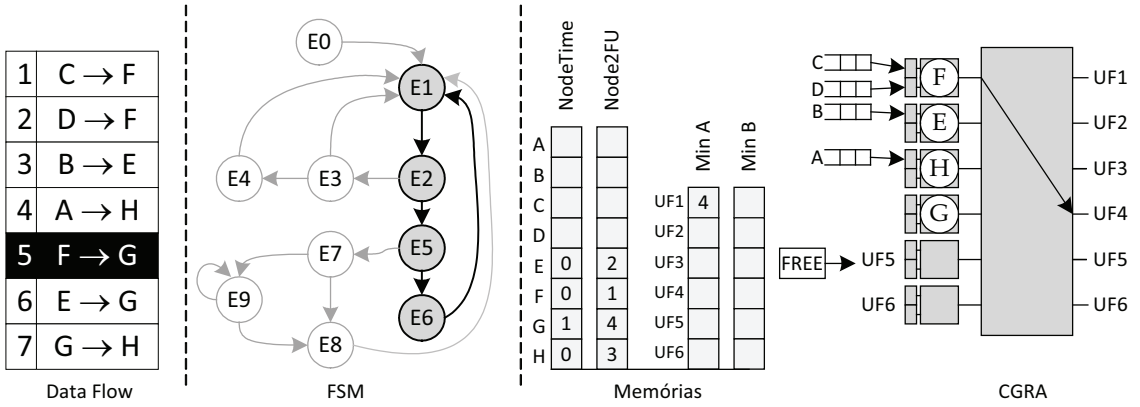


Figura 4.19. Exemplo de mapeamento. Passo 5

A Figura 4.20 ilustra o resultado do algoritmo no processamento do arco $F \rightarrow G$. A FSM verifica no estado E_2 que E já está posicionado e no estado E_5 que G também foi posicionado. Por isso, agora a máquina segue para o estado E_7 que verifica o tempo de escalonamento entre os dois nós para verificar se o novo roteamento para o nó G poderá ser feito diretamente. G foi escalonado no tempo

1 e E escalonado com 0 e por isso, a restrição de tempo é satisfeita, ou seja, G é uma unidade de tempo maior que F. Por isso, será feito o roteamento direto da UF_2 para a UF_4, onde estão posicionados E e G respectivamente. No passo anterior, mostrado na Figura 4.19, o roteamento para G foi feito usando a rede **MinA**, porém agora, no estado E_8, segundo roteamento para o Nó G é feito pela rede **MinB**.

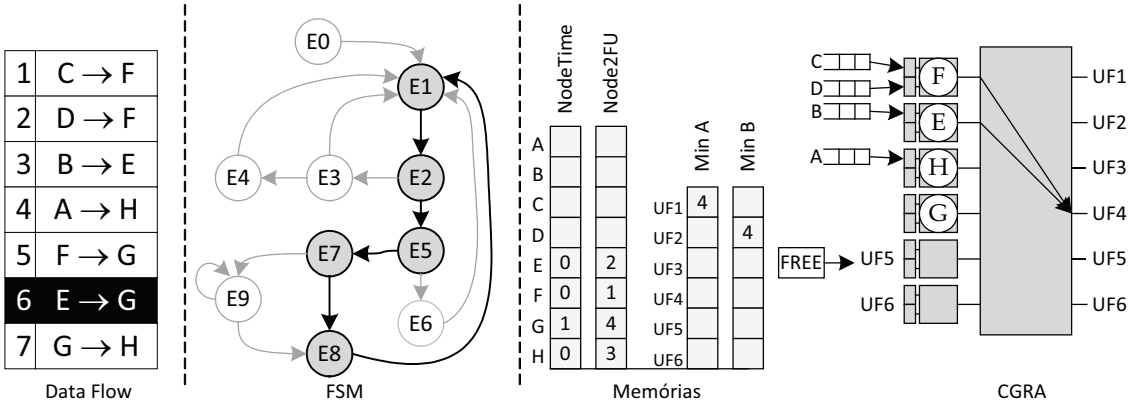


Figura 4.20. Exemplo de mapeamento. Passo 6

No processamento da aresta $G \rightarrow H$, ilustrado na Figura 4.21, que também são dois nós internos, o processamento do algoritmo acontece da mesma forma que no passo anterior até o estado E_7. Porém, diferente da aresta $E \rightarrow G$, onde a restrição de tempo foi satisfeita, na aresta $G \rightarrow H$, o nó G está posicionado no tempo 1, enquanto que H foi posicionado do tempo 0. Por isso a FSM segue agora para o estado E_9 para a inserção de registro.

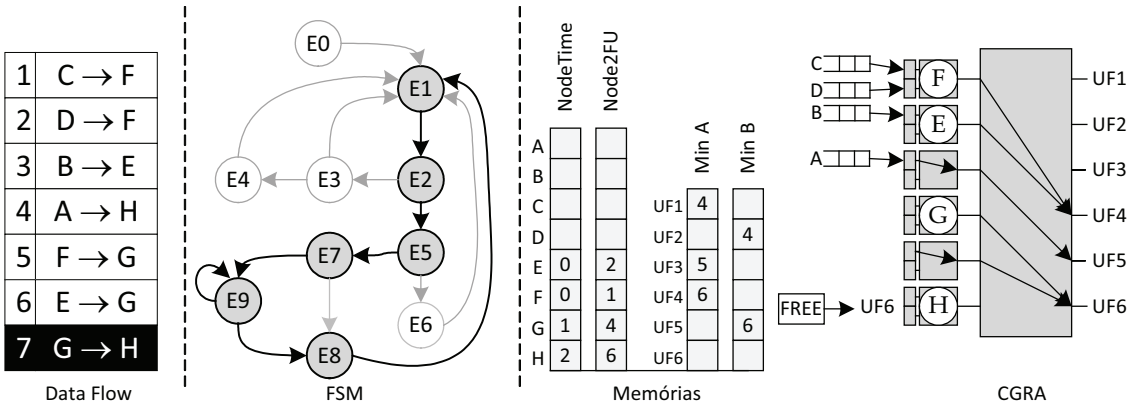


Figura 4.21. Exemplo de mapeamento. Passo 7

Na primeira execução do estado E_9, a UF_3 passa ser usada como registro e o

nó H, que estava posicionado nesta UF, será movido para a UF_5 e marcado com tempo de escalonamento igual a 1. Um roteamento usando a memória **MinA** é feito da UF_3 para a UF_5. Um novo teste verificará se a restrição de tempo entre G e H é satisfeita. Como o tempo de H e G são iguais a 1, a restrição de tempo não foi satisfeita e por isso o estado E_9 será repetido.

Na segunda repetição do estado E_9, a UF_5 passa ser usada como registro e o nó H é escalonado com o tempo 2 e posicionado na UF_6. Agora, com G escalonado no tempo 1 e H com tempo 2, a restrição de tempo está correta e no estado E_8 será feito o roteamento entre a UF_5 para a UF_6 usando a memória **MinB**.

5. Resultados

Neste capítulo avaliamos as arquiteturas propostas e a implementação do *Modulo Scheduling*, usando como métricas o custo em área e o atraso. São apresentados os resultados de escalabilidade do CGRA, com redes *crossbar* e redes multiestágio de tamanhos diferentes. É feito um comparativo entre os CGRAs com rede global e uma arquitetura em malha simplificada da arquitetura ADRES apresentada na Seção 3.4.5, que é a arquitetura padrão das implementações com *Modulo Scheduling*. Porém a maioria dos trabalhos não implementam a arquitetura para validar o ciclo de relógio em função das decisões de projeto. Outro ponto importante, é que a maioria dos CGRAs que são sintetizados em VLSI, nunca foram implementados fisicamente. A virtualização permite a validação e implementação física em um FPGA.

Em seguida são apresentados os resultados de escalabilidade, relacionados aos CGRAs com rede *crossbar* de 16, 20 e 24 FUs e algumas variações na arquitetura. Por fim, na Seção 5.4, são apresentados os resultados para o algoritmo de *Modulo Schedulinge* uma comparação com a solução *softcore* para o mesmo algoritmo.

5.1 Escalabilidade do CGRA

Nesta seção é feita uma análise inicial a respeito da escalabilidade do CGRA. Foram avaliadas arquiteturas de diferentes tamanhos, variando a quantidades de UPs com o objetivo de identificar a utilização de recursos do FPGA e o ciclo de relógio em função do tamanho da arquitetura. O tipo de arquitetura avaliada neste experimento foi apresentado na seção 3.4.1, onde cada UP possui apenas uma UF e um registrador conectada à saída da UF. Os testes foram realizados com arquiteturas de 32 bits com 16, 20, 32 e 64 UPs. As UFs são simples, com 3 bits de configuração para 8 operações, conforme foi apresentada na Figura 3.1 da seção 3.1.1. Portanto o custo do CGRA é dominado pela rede de interconexão.

A Tabela 5.1 apresenta os resultados após a etapa de posicionamento e roteamento usando a ferramenta **Xilinx Webpack ISE 14.2** para o FPGA **Xilinx Virtex 6 XC6vlx240t** para as arquiteturas avaliadas. A coluna *Bits Config* apresenta o número de bits necessários para a configuração de cada arquitetura. Todas

as arquiteturas usam memórias de configuração com 6 bits de endereçamento, que permitem 64 configurações diferentes do CGRA. As Colunas *LUT*, *FF* apresentam o número de *slice LUT* e *slice register* e na coluna *Mem* o número de módulos de memórias RAM embarcadas utilizadas após o passo de P&R. A coluna *Frequência (Mhz)* é a máxima frequência de operação alcançada para cada arquitetura. Os resultados apresentados não consideram os recursos necessários para a unidade do algoritmo de *Modulo Scheduling*.

Arquitetura	Bits Config	FF	LUT	Mem	Frequência (Mhz)
Crossbar 16 UPs	204	1612	8355	6	116
Crossbar 20 UPs	344	2137	18197	10	98
MIN 32 UPs	396	2828	12144	11	108
MIN 32 UPs + 1 extra	524	2956	14121	15	80
MIN 64 UPs	780	5260	23466	22	96
MIN 64 UPs + 1 extra	1036	5516	27334	29	49

Tabela 5.1. Resultados de sínteses dos CGRAs com 16, 2, 32 e 64 UPs, para avaliar a escalabilidade da Arquitetura Mono1R.

Devido ao custo quadrático das redes *crossbar*, avaliamos seu uso apenas nas arquiteturas com 16 e 20 UPs, e para as arquiteturas com 32 e 64 UPs usamos redes multiestágio (MIN). Como nas redes multiestágio podem ocorrer conflito de roteamento, também foram testadas arquiteturas com 1 estágio extra, que duplica a capacidade de roteamento da rede.

Unidades de processamento com DSPs permitem aumentar a velocidade de execução além de economizar recursos. Além disso, ao substituir as UF comuns pelas as UF construídas a partir de DSPs, é possível avaliar o impacto da rede de interconexão, já que desta forma, quase todos os resultados de área em termos de LUTs são usados para a comunicação. A Figura 5.1 apresenta a área ocupada pelas arquiteturas a partir dos *slices luts* e dos *slices register*. Nos resultados apresentados para as arquiteturas DSPs, podemos considerar que os recursos são em sua maioria dedicados a implementação das redes de interconexão.

Os resultados do gráfico da Figura 5.1 indicam que o tamanho da rede de interconexão nas arquiteturas multiestágio é de 39% do tamanho total da arquitetura (Rede e UFs) e de 47% para as redes com 1 estágio extra. Nas arquiteturas com redes *crossbar* a proporção entre o tamanho da rede e o tamanho total da arquitetura aumenta quando aumenta o número de UPs. A rede do CGRA com 16 UPs tem um custo de 54% enquanto que para o CGRA com 20 UPs o custo da rede aumenta para 70%.

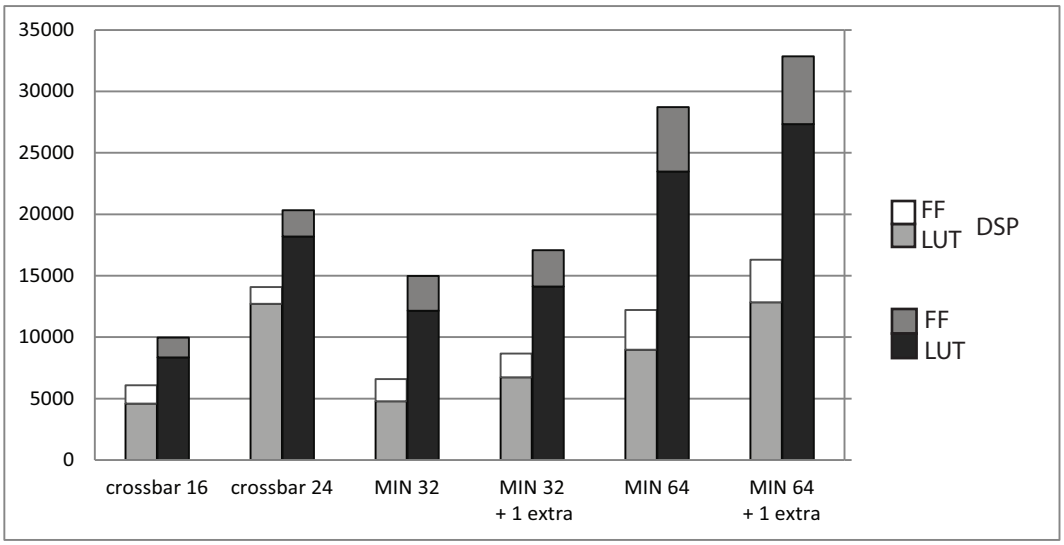


Figura 5.1. Área ocupado pelos CGRAs em termos de *slices luts* e *slices register*.

A Figura 5.2 apresenta a máxima frequência de relógio para os CGRA avaliados. O gráfico indica que a frequência de relógio do CGRA reduz na media que se aumenta a quantidade de UPs. Pode ser observado também que o estágio extra nas redes multiestágio reduz consideravelmente a frequência. As arquiteturas com UPs DSPs são mais rápidas mas também são mais sensíveis ao aumento de tamanho da arquitetura. Provavelmente esta variação na frequência do CGRA acontece devido o posicionamento físico de um módulo DSP no FPGA. Então, como um bloco DSP não pode ser movido, as conexões entre a rede e as UPs podem se tornar muito longas, prejudicando a frequência de operação. Ademais, o DSP tem latência de 2 a 3 ciclos em função da operação a ser desempenhada.

5.1.1 ADRES × CGRAs com rede global

Nesta seção apresentamos a comparação entre os CGRAs com rede global e uma arquitetura em malha baseada na arquitetura ADRES mostrado na Seção 3.4.5. Todas as arquiteturas trabalham com palavras de 32 bits e foram sintetizadas para o mesmo FPGA, Xilinx Virtex 6 XC6vlx240t. A Tabela 5.2 apresenta o resultado da área e a frequência de operação (MHz) para os CGRAs sintetizados, incluindo a a quantidade de bits necessária para configurar cada partição temporal no CGRA.

Na primeira comparação, assumimos os CGRAs com 16 UFs em ambas as arquiteturas. A primeira linha mostra os resultados após o P&R para a arquitetura ADRES padrão, com 16 UPs. A arquitetura é uma malha de 4x4 com 8 registros

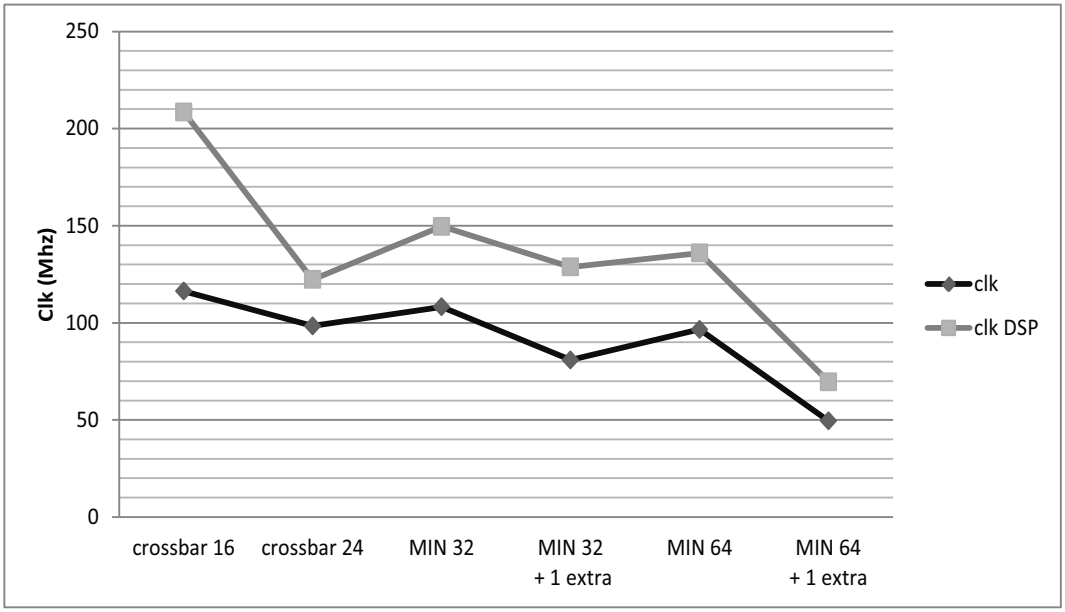


Figura 5.2. Frequência de relógio dos CGRAs, com e sem unidades DSPs

Arquitetura	Bits Config	FF	LUT	Mem	Frequência (Mhz)
ADRES 4x4 UPs	316	1692	6901	9	113
ADRES 4x4 DSPs	316	1180	2724	9	199
Crossbar 16 UPs	204	1612	8355	6	116
Crossbar 16 DSPs	204	1496	4584	6	208
MIN 64 UPs	780	5260	23466	22	96
MIN 64 DSPs	780	3212	8988	22	136
MIN 64 UPs + 1 extra	1036	5516	27334	29	49
MIN 64 DSPs + 1 extra	1036	3468	12832	29	69
ADRES 8x8 UPs	1400	6456	29144	39	112
ADRES 8x8 DSPs	1400	4408	12248	39	162

Tabela 5.2. Resultados das síntese dos CGRAs comparadas com a arquitetura ADRES compatível

locais, como especificado em [Bouwens et al., 2007]. Cada UF possui a capacidade de realizar 8 operações. A segunda linha apresenta a área e a frequência de operação quando todas UFs são substituídas por módulos DSPs. A terceira e quarta linhas nos mostram os resultados para as arquiteturas *crossbar* com 16 UFs. Comparando os resultados entre as duas arquiteturas, é possível observar que a solução *crossbar* proposta é por volta de 16% maior que a arquitetura ADRES. Porém, a arquitetura *crossbar* economiza 33% das memórias embarcadas necessária na síntese das arquiteturas. Além disso, ambas as arquiteturas apresentam uma frequência de

operação compatível. Vale ressaltar que as arquiteturas com rede global simplificam o *Modulo Scheduling*, reduzindo o tempo de geração da configuração de 3 a 6 ordens de grandeza quando comparadas as abordagens que usam arquiteturas do tipo ADRES [Mei et al., 2005a].

Os resultados obtidos quando a unidade funcional é substituída por um módulo DSP demonstram que, na arquitetura ADRES, a interconexão entre as unidades é responsável por pelo menos 40% do total da área. No caso da solução proposta, esse resultado é maior, por volta de 54%, consequência do uso de uma rede *crossbar*. Em relação ao número de memórias RAM, usada para armazenar os bits de configuração para todas as arquiteturas e o registro global do ADRES, a arquitetura de 16 UPs usa poucos recursos, apenas 6 RAMs comparado ao ADRES, que usa 9 RAMs para armazenar os bits de configuração, somado a 16 RAMs para o banco de registros global.

Agora iremos avaliar os resultados para os CGRA com 64 unidades funcionais. O número de UPs aumentou 4×, no entanto, a área utilizada do FPGA teve um aumento de $2,9\times$ para o primeiro e $3,3\times$ para o CGRA com estágio extra. Os resultados também mostram uma diminuição de $1,2 - 1,5\times$ na frequência de operação e uma queda ainda maior, $2,36 - 3\times$ para o CGRA com estágio extra. Porém, isso é compensado por um ILP de $30,3$ que permite aceleração mesmo considerando um aumento de mais de 60% da área. A diferença em área é explicada pelo ganho ao se trocar a rede crossbar pela multiestágio. O aumento do ILP é explicado pelo fato do CGRA dispor de mais unidades e a perda na frequência de relógio devido número de níveis e conexões longas ao adicionar um estagio extra.

Um CGRA ADRES com 64 UPs também foi utilizado na comparação. Devido à interconexão em malha constante, os resultados apresentados são os esperados. O número de UPs aumentou em 4× e sua área também teve um aumento aproximado de 4×. A frequência de operação se manteve praticamente a mesma, já que as distâncias das conexões entre as UPs não são alteradas, independente do aumento da arquitetura. Para a arquitetura ADRES com UPs DSP, observamos uma queda na frequência de operação, que pode ser explicada devido ao posicionamento físico dos módulos DSP embarcados no FPGA, não permitindo um posicionamento das UPs que interferiu nas distâncias entre as UPs.

Como consideração final para as arquiteturas com rede global, vale ressaltar que o código para descrever as arquiteturas foi escrito em VHDL e sintetizado. Não foram realizadas otimizações no projeto visando redução de atraso a partir de um estudo detalhado do circuito gerado pela síntese. Como a arquitetura é gerada apenas uma vez, trabalhos futuros podem ser feitos para modificar a descrição VHDL

e buscar uma síntese mais eficiente em termos de frequência de operação.

5.2 CGRA crossbar 16, 20, 24

Um CGRA *crossbar* com 16 UPs possui um custo em área baixo e frequência de operação compatível com uma arquitetura em malha, porém para arquiteturas de 32 UPs ou superiores, a rede crossbar é inviável. Nesta seção apresentamos também os resultados de síntese para CGRAs intermediários, com 20 e 24 UPs.

A Tabela 5.3 apresenta os resultados a partir da síntese da arquitetura *crossbar* tipo 1, apresentada na Seção 3.4.1 e da arquitetura tipo 2 apresentada na Seção 3.4.2. A diferença entres as 2 arquiteturas é a posição dos registradores de sincronização nas UPs. Também foram feitas duas sínteses de cada arquitetura, uma usando FUs comuns e outra utilizando FUs a partir de DSPs.

Arquitetura	Bits Config.	FF	LUT	Mem	Frequência (Mhz)
Arq1 crossbar 16	200	1481	8153	6	106
Arq1 crossbar 20	288	1824	13948	8	104
Arq1 crossbar 24	344	2137	18197	10	98
Arq1 crossbar 16 DSP	200	969	4613	6	226
Arq1 crossbar 20 DSP	288	1184	9348	8	126
Arq1 crossbar 24 DSP	344	1369	12709	10	122
Arq2 crossbar 16	216	2008	8472	6	126
Arq2 crossbar 20	308	2484	14348	9	104
Arq2 crossbar 24	368	2928	18380	10	90
Arq2 crossbar 16 DSP	216	1496	4584	6	209
Arq2 crossbar 20 DSP	308	1844	9476	9	158
Arq2 crossbar 24 DSP	368	2160	12836	10	173

Tabela 5.3. Resultados das síntese dos CGRAs com 16, 20 e 24 UPs da arquitetura Mono1Re arquiteturas 2.

A Figura 5.3 apresenta o número de *slices LUTs* e *slice Registers* (FF) consumido pela rede *crossbar* e pelas unidades de processamento da arquitetura Mono1Rcom 16, 20 e 24 UPs. O gráfico indica um crescimento de aproximadamente 244 LUTs e de exatamente de 32 *slice Register* para cada UP adicionada, justificado pelas UFs de 32 bits utilizadas.

O crescimento de uma rede *crossbar* é quadrático, porém o tamanho da rede após o P&R vai depender da tecnologia empregada no FPGA alvo. Para redes com 16 entradas/saídas, a árvore de multiplexadores é completa, já que 16 é potência de

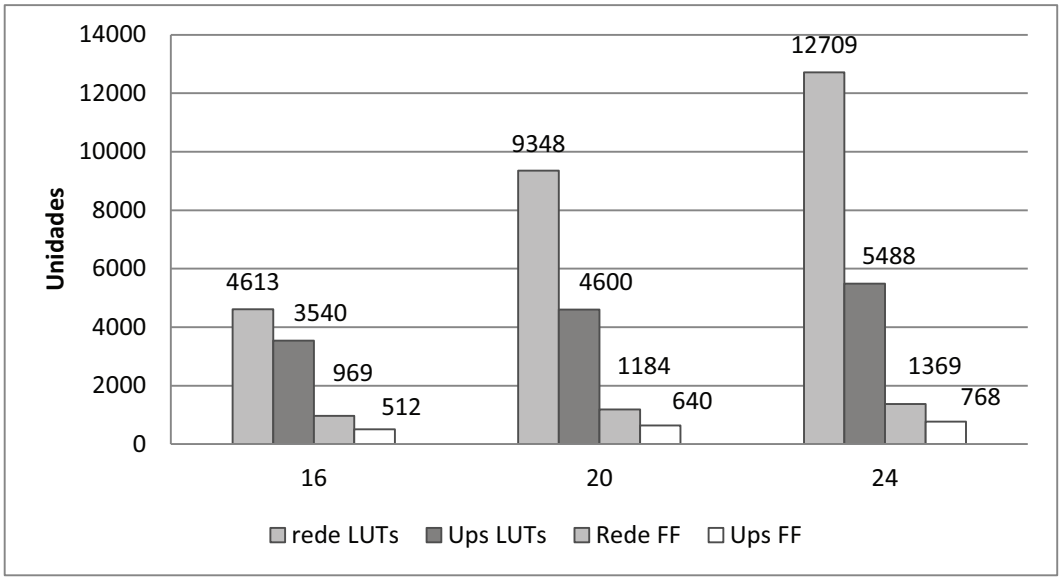


Figura 5.3. Comparação entre o crescimento de área da rede e das UPs na arquiteturas 1 crossbar para 16, 20 e 24 UPs.

2, mas para redes com 20 e 24 ela é desbalanceada fazendo o *delay* aumentar, como mostra a Tabela 5.3.

A variação da frequência de operação para as 4 variações do CGRA é apresentada no gráfico da Figura 5.4. A partir do gráfico podemos observar que o CGRA 16 tem o melhor desempenho em termos de frequência de operação. A arquitetura Mono2Ré 19% mais rápida para o CGRA com 16 UPs, mas apresenta desempenho igual à arquitetura 1 para o CGRA com 20 UPs e 8% mais lento para o CGRA com 24 UPs.

A Tabela 5.4 apresenta os resultados do CGRAs com UPs que também podem fazer operação de multiplicação de dois números, conforme apresentada na Seção 3.1.1. Comparando estes resultados com os resultados da Tabela 5.3, podemos observar que mesmo com unidades mais complexas, o gasto em *slices LUTs* e *slice Registers* diminuiu, porém neste caso, cada UP utiliza também 3 DSPs, reduzindo a utilização dos outros recurso. A frequência de operação no CGRA com multiplicação reduziu em cerca de 21% no pior caso.

5.2.1 Arquitetura Mult2R3M

O desempenho de uma arquitetura não pode ser medido apenas por sua frequência de operação. O intervalo de inicialização que é equivalente ao numero de ciclos para executar uma iteração do laço deve ser considerado. Por exemplo, se a arquitetura

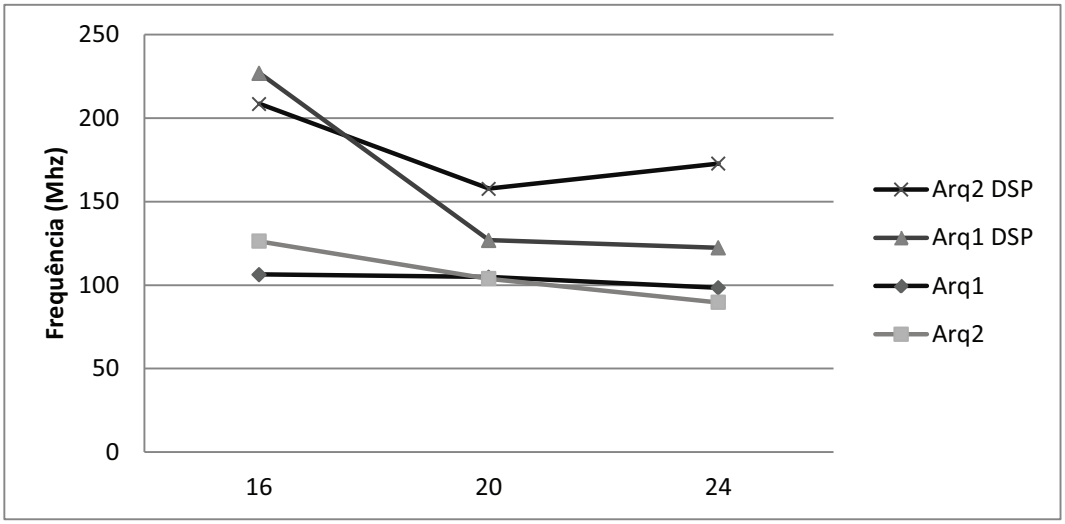


Figura 5.4. Comparação entre as frequências de operação das arquiteturas crossbar 1 e 2 para 16, 20 e 24 UPs

Arquitetura	Bits Config.	FF	LUT	Mem	DSP	Frequência (Mhz)
Arq2 crossbar 16	216	2520	6904	6	48	109
Arq2 crossbar 20	308	3125	12369	8	60	91
Arq2 crossbar 24	368	3697	16324	10	72	74

Tabela 5.4. Resultados das síntese dos CGRAs com 16, 20 e 24 UPs, arquiteturas 2 e UPs com operação de multiplicação.

X tem frequência de 100 Mhz, mas o escalonamento de uma aplicação demanda um II de 3 ciclos, cada iteração será executada em uma frequência de $100/3 = 33.3$. Se a arquitetura Y tem frequência de 80 Mhz, mas possibilita um escalonamento de 2 ciclos, a frequência por iteração do laço da aplicação será de 40 Mhz que é melhor, mesmo que a frequência seja pior.

A arquitetura Mult2R3M(Seção 3.4.3) é uma alternativa proposta por [Ferreira et al., 2013] e [Lopes, 2013] para melhorar a utilização do CGRA, possibilitando a utilização dos dois registros de uma UP, quando necessário o balanceamento do grafo. A Tabela 5.5 apresenta os resultados da Arquitetura Mult2R3Mde 16, 20 e 24 UPs com multiplicação e rede crossbar.

Comparando os resultados da Tabela 5.5 com os resultados da arquitetura Mono2Rcom multiplicação, apresentados na Tabela 5.4, podemos notar que a esta nova arquitetura possui em média 87 LUTs a mais por unidade de processamento e é em média 12% maior que a arquitetura Mono2Rcom multiplicação.

No trabalho de [Lopes, 2013] foi demonstrado que esta estratégia teve um

Arquitetura	Bits Config.	FF	LUT	Mem	DSP	Frequência (Mhz)
Arq3 crossbar 16	265	2569	8195	8	48	89
Arq3 crossbar 20	369	3186	14044	11	60	72
Arq3 crossbar 24	441	3769	18400	13	72	57

Tabela 5.5. Resultados das síntese dos CGRAs com 16, 20 e 24 UPs, arquiteturas 3 e UPs com operação de multiplicação.

grande impacto no resultado do escalonamento, além de possibilitar o escalonamento de outros *benchmarks* que falharam para as arquiteturas base, devido ao grande número de registros de balanceamento necessários.

Com o objetivo de mostrar a evolução do escalonamento quando se aumenta o número de UPs disponíveis, o trabalho de [Lopes, 2013] apresentou o resultado da comparação do intervalo de iniciação (II) para as arquiteturas com 16, 20 e 24 UPs. O resultado desta comparação é apresentado na Figura 5.5. O gráfico mostra o resultado do II em termos de ciclos de relógio para cada *benchmark*, em cada um dos três tamanhos da arquitetura.

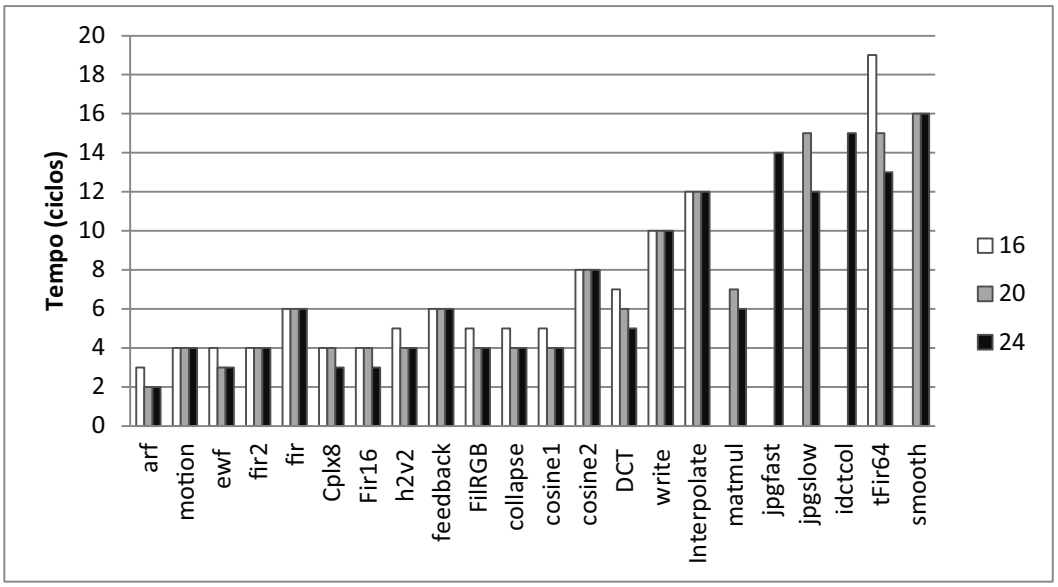


Figura 5.5. Intervalo de Iniciação em ciclos de relógio para três arquiteturas: 16, 20 e 24 PEs. Gráfico retirado do trabalho de [Lopes, 2013]

A partir dos resultados encontrados, ele concluiu que para a maioria dos casos o II diminui com o aumento do número de UPs, melhorando a qualidade do escalonamento. Porém, devemos considerar a frequência de operação, que é umas das contribuições desta dissertação. Os resultados conjuntos dos dois trabalhos, esta

dissertação e o trabalho de [Lopes, 2013] foram publicados [Ferreira et al., 2013]

A Figura 5.6 apresenta o tempo do II em *ns* considerando a implementação no CGRA, resultante do II, para os mesmos *benchmarks*. Estes resultados indicam que aumentando a quantidade de UPs aumenta também o tempo gasto para uma nova leitura.

Isto significa que nas arquiteturas maiores, com 20 e 24 UPs, apesar de uma melhor qualidade de escalonamento, na prática a vazão de uma arquitetura com 16 UPs é em média maior que a vazão das arquiteturas com 20 e 24 UPs, devido a diminuição da frequência de operação em função da complexidade da rede de conexão.

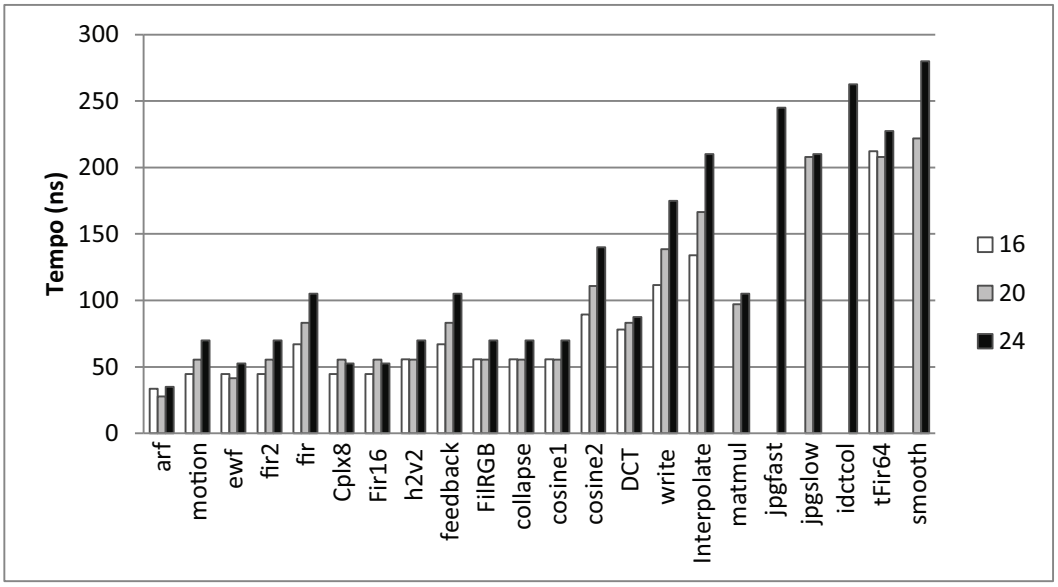


Figura 5.6. Intervalo de Iniciação em nano segundos para três arquiteturas: 16, 20 e 24 PEs.

5.2.2 Arquitetura Mult2R

Assim como na arquitetura Mult2R3M, a arquitetura Mult2R, apresentada na Seção 3.4.4 é uma tentativa de utilizar melhor os registradores disponíveis em cada UP, quando necessário fazer o balanceamento do grafo. A arquitetura utiliza uma única rede multiestágio para criar uma rede de interconexão global com capacidade de roteamento completo para as duas entradas da FU. Na Tabela 5.6 apresentamos os resultados de síntese da arquitetura Mult2Rcom 32 UPs.

Arquitetura	Bits Config.	FF	LUT	Mem	Frequência (Mhz)
Arq4 MIN 32 UPs + 1 extra	554	4905	11432	15	78

Tabela 5.6. Resultados de síntese da Arquitetura Mult2Rcom 32 UPs com multiplicação e rede multiestágio com um estágio extra

5.3 Multitarefas

Na arquitetura proposta foi avaliada também a possibilidade da execução de multitarefas, alternado o uso das UPs entre as tarefas. A Figura 5.7 mostra como duas tarefas podem ser alternadas na arquitetura. A cada ciclo, a informação retida nos registradores de entrada é calculada pela UF e armazenada nos registradores de saída, enquanto que os dados armazenados nos registradores de saída serão roteados e entregues aos registradores de entrada da próxima UP. Desta forma os dados das tarefas 1 e 2 vão sendo alternados juntamente com a configuração da arquitetura.

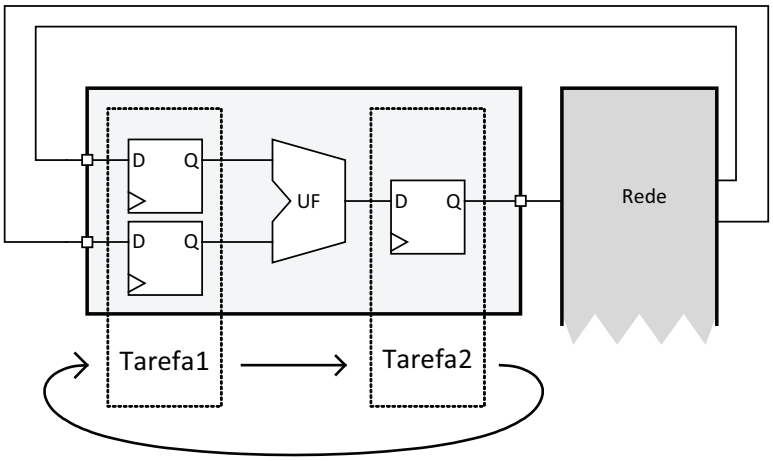


Figura 5.7. Execução de duas tarefas simultâneas

Esta abordagem aumenta a latência, pois são necessários 2 ciclos para executar cada passo, mas também possibilita um ganho na frequência de execução do CGRA, já que um registrador extra é adicionado entre os caminhos de dado encurtando as distâncias. Também foi avaliada na arquitetura em malha ADRES a possibilidade a execução de multitarefas com a inserção de registradores extras nas entradas da UF. A Tabela 5.7 apresenta o resultado desta abordagem e uma comparação com a arquitetura sem a possibilidade de multitarefas.

Os resultados na Tabela 5.7 indicam que esta estratégia tem em média um custo de 12% na área necessária nos CGRAs com rede global e 7% para os CGRAs

	Arch.	Bits Config	LUT	slice register	BRAM	Freq Mhz
simples	16 ADRES	316	6901	1692	9	113
	16 CrossBar	204	8355	1612	6	178
	32 multiestágio	524	14121	2956	15	82
	64 multiestágio	1036	27334	5516	29	49
	64 ADRES	1400	29144	6456	39	112
multi tarefa	16 ADRES	348	6480	2748	10	187
	16 CrossBar	236	8664	2668	7	209
	32 multiestágio	588	14076	5068	16	137
	64 multiestágio	1164	27260	9740	32	76
	64 ADRES	1528	27288	10680	42	137

Tabela 5.7. CGRA multitarefa

ADRES. O ganho na frequência de operação varia de 17% à 67% nas arquiteturas apresentadas. A memória de configuração aumenta em 2 bits para cada UP da arquitetura que resulta em um aumento de 8% no custo final do CGRA.

5.4 Modulo scheduling em hardware

Ao se propor uma solução de arquitetura reconfigurável, deve se considerar também a complexidade do algoritmo de mapeamento em função dos recursos de interconexão para prover reconfiguração em tempo de execução. Por isso, além dos resultados das arquiteturas apresentados nas seções anteriores, nesta seção apresentamos os resultados dos algoritmos de *Modulo Scheduling*apresentados no capítulo 4.

Os resultados estão divididos para cada uma das duas abordagens: a primeira implementação que não trabalha com partições temporais e a segunda implementação que é capaz de mapear um grafo maior que o CGRA criando partições temporais.

5.4.1 *Modulo Scheduling*sem partições temporais

O *Modulo Scheduling*(MSPR) apresentado na Seção 4.2.4.1 foi implementado em um FPGA Virtex6 e o resultado de síntese da arquitetura pode ser visto na Tabela 5.8. A Coluna N mostra o número máximo de Nós suportados pela arquitetura. A arquitetura é descrita usando a linguagem VHDL e o código é parametrizado em função de N.

As colunas LUT, FF retratam o número de *slice* LUT e *slice register* após o passo de P&R feito pela ferramenta Xilinx Webpack ISE 14.2. Para valores pequenos de N, a ferramenta Xilinx automaticamente implementa as memórias usando LUTs.

Para grandes valores de N, podemos ver na coluna Mem o número de módulos de memórias RAM embarcadas que foram usadas no MSPR. Na tabela é apresentado também á máxima frequência de relógio suportada por cada uma das arquiteturas. O MSPR usa pouco recurso como mostra a tabela, mesmo para grandes valores de N. Por exemplo, são usados somente 119 LUTs para um algoritmo que pode configurar uma arquitetura alvo com 64 UFs, que poderá computar até 64 operações em paralelo.

N	LUT	FF	Mem	Clk (Mhz)
16	82	59	0	302
64	199	123	0	275
256	519	315	4	262
1024	1973	1097	4	162

Tabela 5.8. Recursos de FPGA para *Modulo Scheduling*que trabalha com uma única configuração

De acordo com os resultados, o MSPR aumenta a área total do CGRA (incluindo a memória de configuração) em no máximo, 15% comparado com um CGRA de 16 UFs com DSP e em 7% sem DSP. Para CGRAs com 64 UFs, o aumento de área é menor do que 7,3%. Estes resultados demonstram que a unidade de *Modulo Scheduling*é de baixo custo com elevada escalabilidade. Sua área é inferior a 1% da área total do FPGA utilizado.

5.4.2 *Modulo Scheduling*com partições temporais

A tabela 5.9 apresenta os recursos para o MSPR para várias configurações. A coluna FU mostra o número máximo de unidades funcionais do CGRA. Como esta versão do algoritmo pode mapear um grafo com mais operações do que o número de unidades funcionais do CGRA, a coluna *Cfg* apresenta o número máximo de configurações ou partições temporais suportadas para o processo de mapeamento. Assim como na tabela 5.8, as colunas LUT e FF retratam o número de *slice* LUT e *slice register* após o passo de P&R e na coluna Mem o número de módulos de memórias RAM embarcadas utilizadas. O custo de implementação do algoritmo dependerá também do número máximo de configurações ou partições temporais suportadas. Cada configuração adicional vai requerer dois módulos de memória para armazenar a tabela de roteamento. Porém, mesmo para várias configurações, o MSPR usa poucos recursos do FPGA.

N	FU	Cfg	LUT	FF	Mem
32	16	2	137	85	0
128	64	2	284	197	2
512	256	2	812	581	6
64	16	4	239	128	0
256	64	4	626	336	2
1024	256	4	2004	1096	11
128	16	8	368	203	1
512	64	8	949	594	3
256	16	16	723	334	3

Tabela 5.9. *Modulo Scheduling*com 2, 4, 8 e 16 configurações

5.4.3 MSPR vs Microblaze

O algoritmo MSPR foi testado sobre um conjunto de grafos de fluxos de dados disponíveis em [Group, 2013] como mostra a tabela 5.10. As colunas $Name$ e N mostram o nome dos *Benchmarks* e a quantidade de nós que cada um destes possuem. A coluna Reg apresenta o número de registros adicionados para balancear as arestas, necessários para a execução em *pipeline* do algoritmo. A coluna FU lista o número de UFs disponíveis no CGRA alvo. A coluna MII mostra o mínimo intervalo de inicialização de *pipeline*. O mínimo II é computado por $(N + Reg)/FU$. A coluna II mostra o II alcançado pelo algoritmo MSPR. Se $MinII = 1$, o MSPR que não faz uso de partições temporais será usado, caso contrário, mais que uma partição temporal será usada.

name	I/O	N	Reg	FU	MinII	II	FSM	MicroBlaze
arf	8	36	10	64	1	1	172	25398
collapse	6	65	38	64	2	2	374	22877
cosine2	31	112	62	64	3	3	591	46661
interpolate	48	156	40	64	3	4	1327	31376
smooth	64	260	128	256	2	2	1232	54963

Tabela 5.10. performance do MSPR para alguns *benchmarks*

Na coluna FSM da tabela é apresentado o total de ciclos do relógio necessários para mapear cada *benchmark* pelo algoritmo em hardware na arquitetura com 16 UPs. Na coluna *MBlaze*, está listado o número de ciclos gastos no mapeamento do mesmo *benchmark* na mesma arquitetura, porém por um algoritmo equivalente escrito em C executando no MicroBlaze. O MicroBlaze é um *soft processor core* designado para os FPGAs da Xilinx. A versão do MicroBlaze utilizada nesta dissertação é a MB 8.00.B e ele foi configurado com um bloco de memória de 128kb, sem

cache e sem unidades de ponto flutuantes. O algoritmo em C foi restrito de modo conseguir uma versão mais rápida e eficiente para ser executado em um processador de uso geral. Este algoritmo foi compilado usando Cygwin make 3.79.1 com a opção de otimização -O3.

A versão do MSPR em *hardware* é no mínimo 23× mais rápida que a implementação em C executada pelo MicroBlaze. Para o melhor caso, o *arf benchmark*, o MSPR em *hardware* é até 150× mais rápido que a implementação *soft-core* baseada em C. Neste caso, a FSM requer somente 172 ciclos de relógio em comparação como os 25398 ciclos usados pelo MicroBlaze. Além disso, o MSPR usa somente 200 LUTs, enquanto que o MicroBlaze usa por volta de 2000 LUTs.

6. Conclusões

O objetivo deste trabalho foi validar com implementações em FPGA um algoritmo de *Modulo Scheduling*para uma arquitetura CGRA virtual [Mucida, 2013; Lopes, 2013] que realiza o mapeamento de laços paralelizáveis em um CGRA derivado de [Ferreira et al., 2011a]. Nesta dissertação foram avaliadas algumas variações das arquiteturas [Mucida, 2013; Lopes, 2013], apresentando suas implementações em FPGA. O algoritmo de *Modulo Scheduling*descrito em C proposto em [Mucida, 2013] foi detalhado e implementado em hardware com uma descrição em VHDL parametrizada.

Este trabalho faz parte de uma linha de pesquisa onde duas abordagens diferentes foram desenvolvidas com um CGRA virtualizado sobre um FPGA. Além disso, a implementação diretamente em hardware do algoritmo de *Modulo Scheduling*permitiu reduzir o tempo de geração das configurações para uso do *Modulo Scheduling*em tempo de execução.

O requisito fundamental da solução foi o tempo de execução da heurística, que deveria ser compatível com a compilação *Just-In-Time*. No trabalho de [Ferreira et al., 2011a] o escalonamento era feito em software, utilizando as técnicas ASAP e ALAP [1] para balancear o grafo. Porém, no trabalho apresentado nesta dissertação, buscou-se reduzir a complexidade da implementação, em comparação com o trabalho anterior, ao implementar a derivação do algoritmo apresentada em [Mucida, 2013].

Os trabalhos anteriores [Mucida, 2013] e [Lopes, 2013], modificaram a arquitetura de [Ferreira et al., 2011a], substituindo a rede de interconexão multiestágio por uma rede *crossbar*. O objetivo desta troca foi simplificar o roteamento, visto que na rede multiestágio é necessário encontrar um caminho para conectar uma unidade à outra, podendo haver bloqueios, enquanto que a rede *crossbar* simplifica os passos de posicionamento e roteamento, já que a ligação é direta e não acontecem conflitos. Embora as redes *crossbar* não sejam escaláveis, já que elas possuem complexidade de $O(N^2)$, esta dissertação mostrou que em FPGA, a implementação da arquitetura *crossbar* para $N = 16$, o tamanho do CGRA é semelhante ao CGRA baseado em malha. Mesmo com o aumento da área, o número de LUTs das arquiteturas com e sem DSP é inferior a $10-20\%$ do total disponíveis no FPGA utilizado (XC6vlx240t).

[1] Ordenação topológica das operações de acordo com o fluxo de dados.

Essa avaliação mostra que uma arquitetura com rede *crossbar* pode ser implementada em FPGA, com custo viável, o que possibilita a exploração de heurísticas para a compilação JIT, pois simplifica ao máximo o posicionamento e o roteamento.

Também foi avaliado nas arquiteturas a possibilidade de executar 2 tarefas simultaneamente no CGRA, sem que fosse necessário aumentar o número de unidades de processamento. Para isso, enquanto uma tarefa era executada a outra estava sendo roteada. Apesar do aumento da latência a estratégia permite a execução de duas tarefas simultâneas e um ganho na frequência de operação de até 67% com um custo médio de 12% na área.

Em seguida o algoritmo de *Modulo Scheduling*proposto em [Mucida, 2013] no nível de simulação foi validado em hardware com sua implementação em FPGA. Então, diferente do trabalho de [Ferreira et al., 2011a], que apresenta um algoritmo de *Modulo Scheduling*para ser usado em compiladores, o algoritmo de *Modulo Scheduling*apresentado neste trabalho possibilita programar o CGRA em tempo de execução.

As principais características do algoritmo proposto são: configuração em tempo de execução, capacidade de mapear um grafo maior que a arquitetura e capacidade de executar em diferentes tipos de FPGAs e de diferentes fabricantes.

Os resultados experimentais demonstram ganhos de cerca de 4 a 7 vezes quando comparado aos tempos de compilação das soluções existentes. Além disso, os resultados também mostram um alto nível de extração de paralelismo e um intervalo inicial otimizado em mais de 70% dos *benchmarks* testados para a maioria dos gráficos de fluxo de dados multimídia, com um tamanho médio de 70 operações pipeline por laço. A ocupação média da arquitetura é em torno de 76%.

Portanto, através da combinação de uma arquitetura CGRA simples que possibilita posicionamento e roteamento diretos, com uma heurística baseada em decisões gulosas, atribuições simples e poucos vetores para armazenar os dados de escalonamento, posicionamento e roteamento, foi possível alcançar um alto desempenho, a fim de permitir uma configuração de tempo de execução tão eficiente quanto os algoritmos em tempo de compilação.

6.1 Trabalhos Futuros

Trabalhos futuros podem possibilitar a execução de multitarefas visando mais aceleração. Outras variações a partir da arquitetura base também poderão ser experimentadas. Nesta seção propomos algumas sugestões para a continuação deste

trabalho baseadas nos resultados encontrados.

6.1.1 CGRA com unidades de processamento heterogêneas

Todas as arquiteturas apresentadas neste trabalho fazem uso de unidades funcionais homogêneas, ou seja, todas as unidades de processamento são iguais e cada uma destas unidades pode realizar o mesmo conjunto de operações. Porém, utilizar unidades funcionais heterogêneas pode reduzir tanto a área da arquitetura quanto a potência necessária [Park et al., 2008]. Os CGRAs apresentados neste trabalho podem facilmente ser adaptados, sem muito custo, para que possam funcionar com unidades heterogêneas. No entanto o algoritmo de *Modulo Scheduling*apresentado nesta dissertação deverá sofrer algumas modificações. Contudo, para que isto seja possível, basta modificar o vetor livre, adicionando uma nova dimensão. Desta forma o vetor passará ter a informação da próxima unidade livre para cada tipo de unidade diferente. Então caso as unidades específicas para uma operação estejam ocupadas na configuração atual, o algoritmo irá buscar uma unidade na próxima configuração da mesma forma que já vem sendo feito. O resto do algoritmo permanecerá sem modificações.

6.1.2 Tratamento de *Loop* com dependências recorrentes

Em algumas das aplicações, pode existir uma operação com dependência de dados com a iteração anterior. A Figura 6.1(a) ilustra um caso onde existe uma dependência entre iterações. Neste caso existe um caminho fechado no fluxo de dados a partir da conexão do nó E ao nó C. Na Figura 6.1(b) podemos ver que um registro deverá ser adicionado para fazer o balanceamento do grafo. Neste caso, o II mínimo seria 3, em função da dependência.

Para mapear um grafo que possui ciclos de dependência, não é necessária nenhuma alteração no CGRA, conforme podemos ver na Figura 6.1(c), que mostra como ficaria o mapeamento de uma grafo deste tipo em uma arquitetura com 5 unidades de processamento. Porém, o algoritmo de *Modulo Scheduling*proposto neste trabalho não trata esse tipo de situação, mas pode ser alterado como trabalho futuro. Para que seja possível, a adaptação do algoritmo deverá incluir mais um caso além das 4 possibilidades, para tratar as arestas que possuem realimentação entre os vértices.

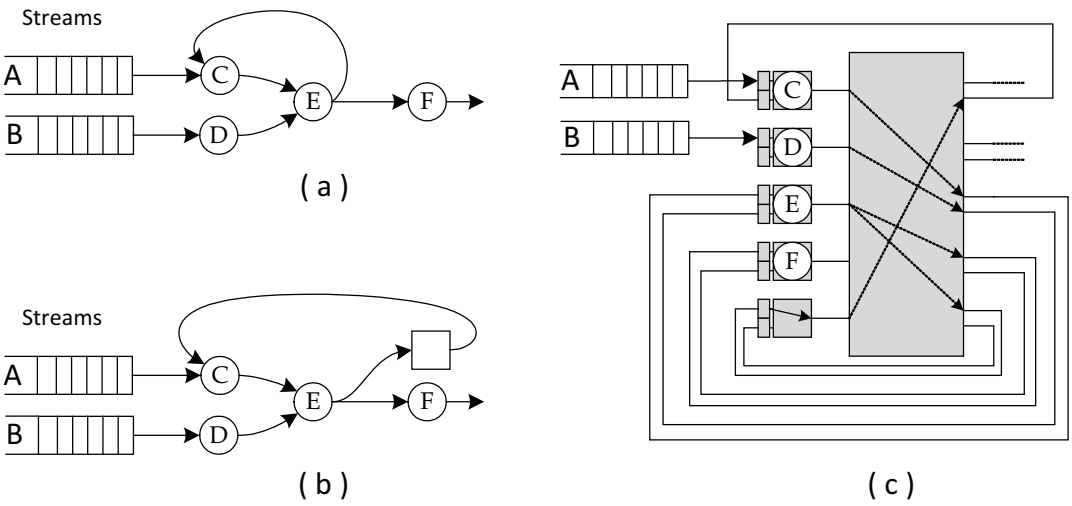


Figura 6.1. (a) *Loop* interno com o Nó E ligando ao Nó C, (b) Balanceamento do grafo com *loop* interno. (c) Mapeamento do grafo com *Loop* interno no CGRA.

6.1.3 Unidades de processamento com banco de registradores

Nos CGRAs propostos neste trabalho, algumas UPs são utilizadas apenas como registradores quando se faz necessário o balanceamento do grafo, diferente da arquitetura ADRES que possui um banco de registros em cada unidade de processamento para resolver a questão do balanceamento do grafo. No caso da arquitetura ADRES, os resultados de uma UF podem ser escrito no banco de registro local ou poderão ser encaminhadas para outras UFs.

Como mostrado nos resultados apresentados por [Lopes, 2013], grande parte das unidades de processamento são utilizadas como registro de balanceamento. Por isso, baseado no trabalho de [Hamzeh et al., 2013], na tentativa de liberar a unidade de processamento quando necessário a inclusão de uma registros de balanceamento, propõe-se como trabalho futuro implementar as unidades de processamento dotadas de um banco de registradores locais. Dessa forma, podemos utilizar uma arquitetura que possibilite que os registros de balanceamento não bloquem uma UP.

Um exemplo de arquitetura que utiliza banco de registradores, é apresentada na Figura 6.2. As UPs dessa arquitetura são semelhante às UPs da arquitetura ADRES. Dessa forma, o resultado da operação de uma UF pode ser escrito no banco de registro local ou poderá ser encaminhado para a rede. Assim como a arquitetura ADRES, neste caso, o banco de registro também deverá possuir a capacidade de leitura em duas posições de memória simultaneamente.

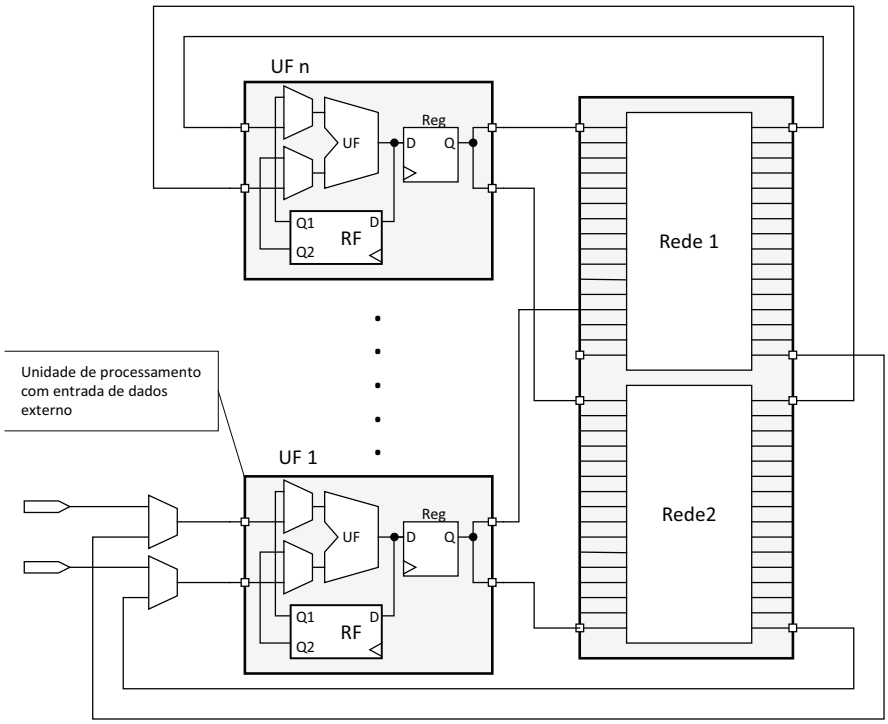


Figura 6.2. CGRA com unidades de processamento que utilizam banco de registradores locais.

Trabalhos futuros que adotem a utilização de registros locais nas arquiteturas de rede global deverão fazer uma maior investigação de modo definir a melhor maneira desta abordagem para permitir uma execução mais eficiente dos algoritmos nestas arquiteturas. Uma grande alteração no algoritmo de *Modulo Scheduling*também será necessária, já que dessa forma o processo de escalonamento, mapeamento e roteamento nestas arquiteturas será em partes bem diferente do processo mostrado aqui.

6.1.4 Algoritmo MSG

O algoritmo de *Modulo Scheduling*implementado nesta dissertação é baseado no algoritmo EPR, como já foi dito no Capítulo 4. No entanto, o trabalho MSG [Lopes, 2013] apresenta uma proposta de implementação do algoritmo na forma de máquina de estados finitos que pode ser vista na Figura 6.3.

A abordagem MSG realiza uma redução no número de registradores com *multicasting* utilizados no mapeamento. Para isso, essa heurística faz uso da arquitetura Mult2R3Mapresentado na Seção 3.4.3, que possibilita o melhor uso dos registros de cada elemento de processamento.

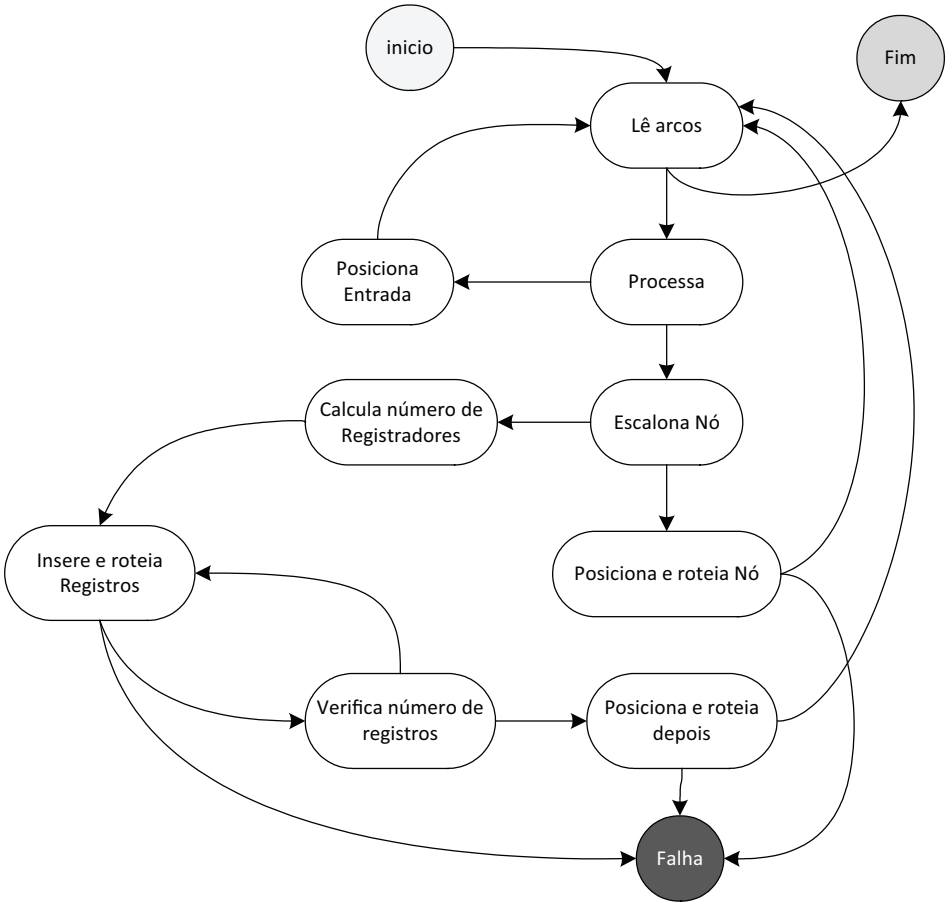


Figura 6.3. Máquina de estados finitos da heurística MSG.

O algoritmo usa apenas alguns registros para guardar valores locais e 7 memórias distribuídas. A estrutura interna do módulo de escalonamento, posicionamento e roteamento necessária para executar a heurística MSG pode ser vista na Figura 6.4. Diferente da heurística EPR, que é baseada no processamento de aresta, a heurística MSG trabalha com processamento de vértices.

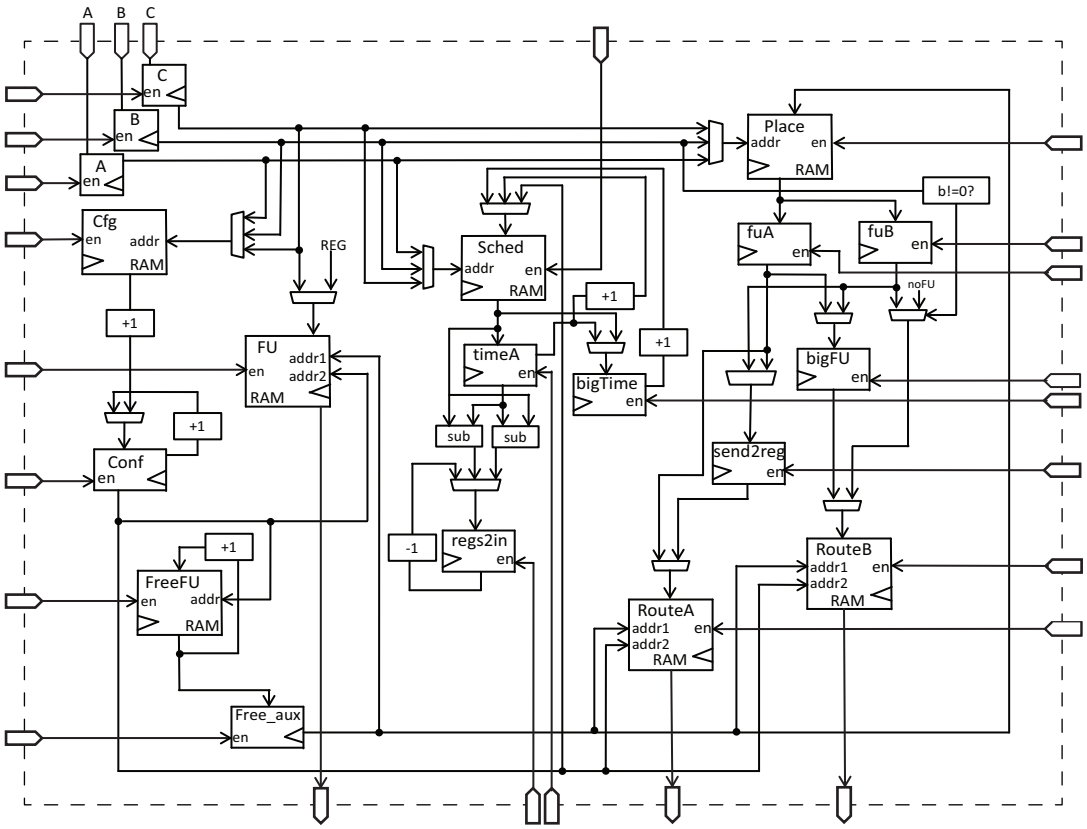


Figura 6.4. Memórias de Escalonamento, Posicionamento e Roteamento para a heurística MSG.

Referências Bibliográficas

Baumgarte, V.; Ehlers, G.; May, F.; Nückel, A.; Vorbach, M. & Weinhardt, M. (2003). Pact xpp-a self-reconfigurable data processing architecture. *the Journal of Supercomputing*, 26(2):167--184.

Bouwens, F.; Berekovic, M.; Kanstein, A. & Gaydadjiev, G. (2007). Architectural exploration of the adres coarse-grained reconfigurable array. In *Proc. ARC*, pp. 1–13.

Brant, A. & Lemieux, G. (2012). Zuma: An open fpga overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pp. 93 –96.

Chen, L. & Mitra, T. (2012). Graph minor approach for application mapping on cgras. In *FPT*, pp. 285–292.

Colwell, R. P.; Nix, R. P.; O'Donnell, J. J.; Papworth, D. B. & Rodman, P. K. (1988). A vliw architecture for a trace scheduling compiler. *Computers, IEEE Transactions on*, 37(8):967--979.

Coole, J. & Stitt, G. (2010). Intermediate fabrics: virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pp. 13--22, New York, NY, USA. ACM.

DeHon, A. (2000). The density advantage of configurable computing. *Computer*, 33(4):41--49.

Eguro, K. & Hauck, S. (2006). Armada: Timing-driven pipeline-aware routing for fpgas. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pp. 169--178, New York, NY, USA. ACM.

Ferreira, R.; Duarte, V.; Meireles, W.; Pereira, M.; Carro, L. & Wong, S. (2013). A just-in-time modulo scheduling for virtual coarse-grained reconfigurables architectures. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, Samos, Greece. IEEE Computer Society.

Ferreira, R.; Vendramini, J.; Mucida, L.; Pereira, M. & Carro, L. (2011a). An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2011 Proceedings of the 14th International Conference on*, pp. 195 –204.

Ferreira, R.; Vendramini, J. & Nacif, M. (2011b). Dynamic reconfigurable multicast interconnections by using radix-4 multistage networks in fpga. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pp. 810–815.

Filho, W. D. M. & Ferreira, R. S. (2012). A hardware-assisted modulo scheduling, placement and routing algorithm for stream computing in coarse-grained reconfigurable architectures. In *25th Symposium on Integraded Circuits and Systems Design - XI Microelectronics Students Forum (SForum)*.

Fung, R.; Betz, V. & Chow, W. (2004). Simultaneous short-path and long-path timing optimization for fpgas. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pp. 838--845. IEEE Computer Society.

Group, E. (2013). Extensible, programmable and reconfigurable embedded systems group. http://express.ece.ucsb.edu/benchmark/.

Guo, M. (2005). Energy-aware compiler scheduling for vliw embedded software. In *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pp. 197–.

Hamzeh, M.; Shrivastava, A. & Vrudhula, S. (2012). Epimap: using epimorphism to map applications on cgras. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 1284--1291, New York, NY, USA. ACM.

Hamzeh, M.; Shrivastava, A. & Vrudhula, S. (2013). Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, New York, NY, USA. ACM.

Hartenstein, R. (2001a). Coarse grain reconfigurable architectures. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pp. 564 –569.

Hartenstein, R. (2001b). A decade of reconfigurable computing: a visionary retrospective. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pp. 642 –649.

Hatanaka, A. & Bagherzadeh, N. (2007). A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1 –8.

Hauck, S. & DeHon, A. (2007). *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Lopes, V. D. (2013). Uma heurística polinomial para escalonamento de loops em arquiteturas reconfiguráveis de grão grosso. Master's thesis, "Universidade Federal de Viçosa, UFV", Viçosa, Minas Gerais.

Lu, G.; Singh, H.; Lee, M.-H.; Bagherzadeh, N.; Kurdahi, F. & Eliseu Filho, M. (1999). The morphosys parallel reconfigurable system. In *Euro-Par 99 Parallel Processing*, pp. 727--734. Springer.

Mei, B.; Lambrechts, A.; Mignolet, J.-Y.; Verkest, D. & Lauwereins, R. (2005a). Architecture exploration for a reconfigurable architecture template. *Design Test of Computers, IEEE*, 22(2):90 – 101.

Mei, B.; Veredas, F.-J. & Masschelein, B. (2005b). Mapping an h. 264/avc decoder onto the adres reconfigurable architecture. In *Field Programmable Logic and Applications, 2005. International Conference on*, pp. 622--625. IEEE.

Mei, B.; Vernalde, S.; Verkest, D.; De Man, H. & Lauwereins, R. (2002). Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. In *Proc. FPT*, pp. 166 – 173.

Mei, B.; Vernalde, S.; Verkest, D.; Man, H. D. & Lauwereins, R. (2003). Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. DATE*.

Mucida, L. (2013). Uma heurística gulosa para modulo scheduling em arquiteturas reconfiguráveis em tempo de execução. Master's thesis, "Universidade Federal de Viçosa, UFV", Viçosa, Minas Gerais.

Mucida, L.; Lopes, V.; Meireles, W. & Ferreira, R. (2012). Problem oriented approach to hardware-assisted algorithm design in c: A case study for scheduling, placement and routing. In *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*, pp. 1–8.

Oh, T.; Egger, B.; Park, H. & Mahlke, S. (2009). Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. *SIGPLAN Not.*, 44(7):21--30.

Park, H.; Fan, K.; Kudlur, M. & Mahlke, S. (2006). Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *Proc. CASES*, pp. 136--146.

Park, H.; Fan, K.; Mahlke, S. A.; Oh, T.; Kim, H. & Kim, H.-s. (2008). Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 166--176, New York, NY, USA. ACM.

Park, H.; Park, Y. & Mahlke, S. (2009). Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 370--380, New York, NY, USA. ACM.

Quax, M.; Huisken, J. & van Meerbergen, J. (2004). A scalable implementation of a reconfigurable wcdma rake receiver. In *Proceedings of the conference on Design, automation and test in Europe-Volume 3*, p. 30230. IEEE Computer Society.

Rau, B. R. (1994). Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pp. 63--74, New York, NY, USA. ACM.

Sanchez, E.; Sipper, M.; Haenni, J.-O.; Beuchat, J.-L.; Stauffer, A. & Perez-Uribe, A. (1999). Static and dynamic configurable systems. *Computers, IEEE Transactions on*, 48(6):556–564.

Sánchez, J. & González, A. (2000). Modulo scheduling for a fully-distributed clustered vliw architecture. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pp. 124--133, New York, NY, USA. ACM.

Shrivastava, A.; Pager, J.; Jeyapaul, R.; Hamzeh, M. & Vrudhula, S. (2011). Enabling multithreading on cgras. In *Parallel Processing (ICPP), 2011 International Conference on*, pp. 255–264.

Singh, H.; Lee, M.-H.; Lu, G.; Kurdahi, F.; Bagherzadeh, N. & Chaves Filho, E. (2000). Morphosys: an integrated reconfigurable system for data-parallel and

computation-intensive applications. *Computers, IEEE Transactions on*, 49(5):465 –481.

Vendramini, J. & Ferreira, R. (2010). Parallel routing algorithm for extra level omega networks on reconfigurable systems. In *Computing Systems (WSCAD-SCC), 2010 11th Symposium on*, pp. 1–8.

Vendramini, J. C. G. (2012). Rede omega virtual em fpga com reconfiguração em tempo de execução. estudo de caso: cálculo de atratores em redes regulares de genes. Master's thesis, "Universidade Federal de Viçosa, UFV", Viçosa, Minas Gerais.

Xilinx (2010). Virtex 6 fpga gtx transceivers user guide, may 24, 2010 revision history.

Yoon, J.; Shrivastava, A.; Park, S.; Ahn, M.; Jeyapaul, R. & Paek, Y. (2008). Spkm : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pp. 776–782.

Yun, H.-S. & Kim, J. (2001). Power-aware modulo scheduling for high-performance vliw processors. In *Low Power Electronics and Design, International Symposium on, 2001.*, pp. 40–45.

Zalamea, J.; Llosa, J.; Ayguade, E. & Valero, M. (2001). Modulo scheduling with integrated register spilling for clustered vliw architectures. In *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pp. 160–169.