UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO
MESTRADO EM SISTEMAS E COMPUTAÇÃO

PROPOSTA E IMPLEMENTAÇÃO DE UMA ARQUITETURA RECONFIGURÁVEL HÍBRIDA PARA APLICAÇÕES BASEADAS EM FLUXO DE DADOS

Monica Magalhães Pereira

Fevereiro, 2008
Natal/RN

MONICA MAGALHÃES PEREIRA

PROPOSTA E IMPLEMENTAÇÃO DE UMA ARQUITETURA RECONFIGURÁVEL HÍBRIDA PARA APLICAÇÕES BASEADAS EM FLUXO DE DADOS

Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos para obtenção do título de Mestre em Sistemas e Computação (MSc.).

Orientador: Prof. Dr. Ivan Saraiva Silva

Fevereiro, 2008
Natal/RN

Agradecimentos

Dedico este trabalho a André, e agradeço enormemente o apoio durante todos esses anos de intenso trabalho. À compreensão e às palavras de incentivo sempre me encorajaram a manter a dedicação e o foco.

Agradeço também a minha família, parte essencial na minha vida e que sempre esteve presente em todos os momentos, me dando todo apoio e amor que precisei.

Ao meu orientador Ivan Saraiva, que foi o principal responsável pelo meu ingresso na pesquisa acadêmica. Sua dedicação à pesquisa e seu comprometimento com a orientação dos seus alunos são motivos de inspiração e exemplo a ser seguido.

Aos amigos Sílvio Fernandes, Gustavo Girão e Bruno Cruz que tanto torceram por meu sucesso. As palavras de encorajamento, o apoio e até mesmo os momentos de descontração foram de grande contribuição para a conclusão desta etapa.

Agradeço também à Alba Sandyra, pela contribuição no desenvolvimento deste trabalho. Mesmo tendo ingressado há pouco tempo na pesquisa acadêmica, já demonstra um grande talento. Tenho certeza que será muito bem sucedida nesta área.

Finalmente, agradeço a todos que direta ou indiretamente contribuíram para o sucesso deste trabalho e a conclusão de mais uma etapa da minha carreira profissional.

Resumo

O aumento na complexidade das aplicações vem exigindo dispositivos cada vez mais flexíveis e capazes de alcançar alto desempenho. As soluções de hardware tradicionais são ineficientes para atender as exigências dessas aplicações. Processadores de propósito geral, embora possuam flexibilidade inerente devido à capacidade de executar diversos tipos de tarefas, não alcançam alto desempenho quando comparados às arquiteturas de aplicação específica. Este último, por ser especializado em uma pequena quantidade de tarefas, alcança alto desempenho, porém não possui flexibilidade. Arquiteturas reconfiguráveis surgiram como uma alternativa às abordagens convencionais e vem ganhado espaço nas últimas décadas. A proposta desse paradigma é alterar o comportamento do hardware de acordo com a aplicação a ser executada. Dessa forma, é possível equilibrar flexibilidade e desempenho e atender a demanda das aplicações atuais.

Esse trabalho propõe o projeto e a implementação de uma arquitetura reconfigurável híbrida de granularidade grossa, voltada a aplicações baseadas em fluxo de dados. A arquitetura, denominada RoSA, consiste de um bloco reconfigurável anexado a um processador. Seu objetivo é explorar paralelismo no nível de instrução de aplicações com intenso fluxo de dados e com isso acelerar a execução dessas aplicações no bloco reconfigurável. A exploração de paralelismo no nível de instrução é feita em tempo de compilação e para tal, esse trabalho também propõe uma fase de otimização para a arquitetura RoSA a ser incluída no compilador GCC. Para o projeto da arquitetura esse trabalho também apresenta uma metodologia baseada no reuso de hardware em caminho de dados, denominada RoSE. Sua proposta é visualizar as unidades reconfiguráveis através de níveis de reusabilidade, que permitem a economia de área e a simplificação do projeto do caminho de dados da arquitetura.

A arquitetura proposta foi implementada em linguagem de descrição de hardware (VHDL). Sua validação deu-se através de simulações e da prototipação em FPGA. Para análise de desempenho foram utilizados alguns estudos de caso que demonstraram uma aceleração de até 11 vezes na execução de algumas aplicações.

Palavras-chave: Arquitetura Reconfigurável, Paralelismo, Flexibilidade e Desempenho.

Abstract

The increase of applications complexity has demanded hardware even more flexible and able to achieve higher performance. Traditional hardware solutions have not been successful in providing these applications constraints. General purpose processors have inherent flexibility, since they perform several tasks, however, they can not reach high performance when compared to application-specific devices. Moreover, since application-specific devices perform only few tasks, they achieve high performance, although they have less flexibility. Reconfigurable architectures emerged as an alternative to traditional approaches and have become an area of rising interest over the last decades. The purpose of this new paradigm is to modify the device's behavior according to the application. Thus, it is possible to balance flexibility and performance and also to attend the applications constraints.

This work presents the design and implementation of a coarse grained hybrid reconfigurable architecture to stream-based applications. The architecture, named RoSA, consists of a reconfigurable logic attached to a processor. Its goal is to exploit the instruction level parallelism from intensive data-flow applications to accelerate the application's execution on the reconfigurable logic. The instruction level parallelism extraction is done at compile time, thus, this work also presents an optimization phase to the RoSA architecture to be included in the GCC compiler. To design the architecture, this work also presents a methodology based on hardware reuse of datapaths, named RoSE. RoSE aims to visualize the reconfigurable units through reusability levels, which provides area saving and datapath simplification.

The architecture presented was implemented in hardware description language (VHDL). It was validated through simulations and prototyping. To characterize performance analysis some benchmarks were used and they demonstrated a speedup of 11x on the execution of some applications.

Key Words: *Reconfigurable Architecture, Parallelism, Flexibility and Performance.*

Lista de Figuras

Sumário

Capítulo 1
Introdução

O crescente aumento da complexidade das aplicações computacionalmente intensivas vem exigindo cada vez mais dispositivos de hardware capazes de executar aplicações com alto desempenho. Além disso, a rápida evolução das tecnologias utilizadas para o desenvolvimento de componentes de hardware tem gerado novas opções para os desenvolvedores, aumentando a demanda por dispositivos que atendam determinados requisitos como: flexibilidade, área, energia consumida, dentre outros.

Durante muito tempo, os esforços na busca por maior desempenho e flexibilidade foram concentrados em dois paradigmas de arquitetura de computadores: os processadores de propósito geral e as arquiteturas de aplicação específica. Os processadores de propósito geral, também conhecidos como arquiteturas de Von Neumann, são descritos pela sua flexibilidade intrínseca, resultado da sua capacidade de realizar uma grande quantidade de tarefas diferentes (PLATZNER, 1998).

A grande flexibilidade dos processadores possibilita a execução de diversas aplicações através de modificações feitas em software. Essa característica permite que os processadores possuam ampla funcionalidade, mesmo que isso cause um impacto no desempenho ou energia consumida do hardware (GONZALEZ, 2006).

Para determinadas aplicações onde o desempenho do hardware é requisito essencial para uma execução bem sucedida, é recomendada a utilização dos circuitos de aplicação específica, ou hardware dedicados, também chamados de ASICs (*Application Specific Integrated Circuits*). Tais arquiteturas são projetados de forma direcionada a um número pequeno de aplicações, partes de uma aplicação ou até mesmo a apenas uma tarefa. Por esse motivo, alcançam alto desempenho quando executam as tarefas para as quais foram projetados (TODMAN et al, 2005).

Apesar de sua flexibilidade inerente, os processadores de propósito geral, por não serem especializados em nenhuma tarefa, não atingem o desempenho comparável ao dos ASICs. Por outro lado, a principal desvantagem dos ASICs está na ausência de flexibilidade. Uma vez que não podem ser modificados após a fabricação, para qualquer alteração a ser

realizada em alguma parte do hardware é necessário um novo projeto e uma nova fabricação desses circuitos (COMPTON e HAUCK, 2002).

Como uma possível alternativa às soluções tradicionais, na tentativa de equilibrar flexibilidade e desempenho, surgiram as arquiteturas reconfiguráveis. Esta é uma abordagem recente que vem sendo bastante explorada nos últimos anos e tem alcançado resultados significativos em atender os requisitos das aplicações atuais (HARTENSTEIN, 2001c).

Arquiteturas reconfiguráveis são capazes de adaptar seu hardware de acordo com a aplicação a ser executada. Essa capacidade torna possível alcançar tanto a flexibilidade dos processadores quanto o desempenho dos ASICs (BHATIA, 1997). Já que a estratégia implementada por elas é de modificar o hardware para atender ao software ao invés de modificar o software para se ajustar ao hardware como tradicionalmente é feito (GONZALEZ, 2006).

A demanda cada vez maior por arquiteturas flexíveis, capazes de executar aplicações com alto desempenho, faz das arquiteturas reconfiguráveis uma solução bastante promissora e que tem demonstrado resultados significativos na computação dessas aplicações nos últimos anos. A quantidade de arquiteturas reconfiguráveis encontradas na literatura comprova sua evolução nos últimos anos (HARTENSTEIN, 2001b).

Para contribuir com a busca por uma arquitetura capaz de unir flexibilidade e alto desempenho em um mesmo sistema é que esse trabalho propõe o projeto e a implementação de uma arquitetura reconfigurável híbrida denominada RoSA (*Reconfigurable Stream-based Architecture*). RoSA consiste de um bloco IP (*Intelectual Property*) reconfigurável, anexado a um processador de propósito geral. O bloco reconfigurável executa trechos de código computacionalmente intensivos e, portanto, computacionalmente custosos para o processador. O processador, por sua vez, é responsável por controlar a execução do bloco e executar os trechos de código menos custosos.

A arquitetura visa explorar o paralelismo no nível de instrução (ILP – *instruction-level parallelism*) de aplicações baseadas em fluxo de dados (CALLAHAN e WAWRZYNEK, 1998). As técnicas adotadas para a extração de ILP em RoSA realizam análise de grafos de fluxo de dados em tempo de compilação. Por esse motivo, esse trabalho também propõe uma fase de otimização para RoSA em um compilador GCC.

Esse trabalho também apresenta uma nova metodologia para o reuso de hardware em caminho de dados, para ser utilizada no projeto de caminhos de dados

complexos, tais como as arquiteturas reconfiguráveis. A metodologia proposta, denominada RoSE (*Reuse-based Standard Datapath Architecture*), organiza as unidades operacionais pertencentes às unidades reconfiguráveis em um caminho de dados baseado em níveis de reusabilidade. Com isso, as unidades reconfiguráveis podem ser visualizadas como uma arquitetura VLIW (*Very Long Instruction Word*), que provê compartilhamento de área por todas as unidades operacionais.

Este documento está estruturado da seguinte forma:

- **Fundamentação teórica:** no segundo capítulo são introduzidos alguns conceitos que incluem as técnicas para exploração de paralelismo no nível de instruções e as principais técnicas de compilação utilizadas na exploração desse paralelismo. Também são apresentadas as principais características das arquiteturas reconfiguráveis, sua classificação e uma breve descrição de algumas das principais arquiteturas encontradas na literatura.
- **Técnicas de compilação:** no terceiro capítulo são descritas com detalhes as técnicas de compilação adotadas no projeto da arquitetura RoSA.
- **Apresentação da arquitetura:** o quarto capítulo apresenta a arquitetura RoSA com detalhes de seus componentes e seu fluxo de execução.
- **Análise dos Resultados:** os resultados de área, freqüência e desempenho alcançados são analisados no quinto capítulo. Para a análise dos resultados foram comparadas diversas implementações da arquitetura, com alterações em alguns componentes e para análise de desempenho algumas aplicações foram utilizadas.
- **Conclusão:** finalmente, no sexto capítulo são apresentadas as conclusões e os trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Antes de aprofundar o conhecimento em arquiteturas reconfiguráveis é necessário introduzir alguns conceitos que estão relacionados com a capacidade de reconfiguração dessas arquiteturas.

O conceito, apresentado na seção 2.1, corresponde ao paradigma arquitetural de suporte a paralelismo no nível de instrução. Esse conceito está relacionado com a propriedade das arquiteturas reconfiguráveis de executar tarefas em paralelo.

Uma breve descrição das técnicas de compilação existentes para extrair ILP é apresentada na seção 2.2. Finalmente, na seção 2.3 é realizada uma descrição mais aprofundada sobre arquiteturas reconfiguráveis e suas características. Também é feita uma breve exposição de algumas das principais arquiteturas existentes na literatura.

2.1 Paralelismo no Nível de Instrução

De acordo com (RAU e FISHER, 1992) a partir dos anos 1980 a adoção do ILP (*instruction-level parallelism*) como um novo paradigma de exploração de paralelismo se tornou a principal estratégia de acelerar a execução das aplicações. Essa abordagem tornou-se popular por ser uma forma natural de extrair o paralelismo de aplicações seqüenciais sem a necessidade de reescrevê-las (SCHLANSKER et al, 1997).

No projeto de arquiteturas reconfiguráveis, a exploração de ILP pode ser utilizada como uma estratégia para atender a necessidade de particionar a aplicação entre a parte do processador e a parte do bloco reconfigurável (que executa a aplicação em paralelo). Uma vez que as aplicações são naturalmente descritas de forma seqüencial, devido tanto aos programadores quanto a própria semântica das linguagens, é mais simples aplicar técnicas para a exploração de paralelismo das aplicações seqüenciais do que modificar toda cultura de desenvolvimento de software.

Apesar disso, para alcançar alto grau de paralelismo através de ILP é necessário tanto um hardware equipado com caminhos de dados paralelos para a execução simultânea, quanto um compilador que exponha o paralelismo já existente nas aplicações. Por esse motivo, alguns autores consideram a exploração de ILP uma técnica limitada por

depender da quantidade de paralelismo existente nas aplicações. Em (BECK e CARRO, 2005) é proposta a utilização de tradução binária como uma alternativa à exploração de ILP. Essa técnica pode ser aplicada a qualquer trecho da aplicação e não depende do paralelismo disponível na aplicação.

Para explorar ILP é necessário comparar as instruções de uma aplicação seqüencial e analisar as dependências de controle e de dados existentes entre elas. O objetivo da análise é encontrar instruções independentes que possam ser executadas em paralelo sem alterar o propósito da aplicação (POZZI, 2000).

Existem duas abordagens para explorar o paralelismo no nível de instruções: estática e dinâmica. A primeira abordagem baseia-se no software para explorar ILP e é adotada pelas arquiteturas VLIW (*Very Long Instruction Word*). A segunda abordagem depende do hardware para explorar o paralelismo e é encontrada nas máquinas super-escalares.

Arquiteturas VLIW exploram paralelismo no nível de instrução através do agrupamento de várias operações em uma única instrução maior (por esse motivo a denominação *Very Long Instruction Word*). A execução dessa instrução "longa" corresponde à execução paralela das operações nela agrupadas (FISHER et al, 2004).

Para executar as instruções longas, as arquiteturas VLIW utilizam várias unidades funcionais independentes. As unidades funcionais trabalham paralelamente na execução de cada operação presente na instrução. Porém, o número de unidades funcionais e o grau de paralelismo explorado dependem da quantidade de ILP disponível na aplicação. Portanto, o projeto de arquiteturas VLIW deve levar em conta o ILP disponível de modo a maximizar a utilização do hardware, inclusive fazendo uso de técnicas de compilação que exponham o paralelismo disponível na aplicação.

Nas arquiteturas super-escalares o paralelismo está em executar várias instruções ao mesmo tempo (POZZI, 1999). Isto é feito através de múltiplos pipelines que executam instruções independentes simultaneamente. Para selecionar as instruções que serão executadas em paralelo é necessário verificar a dependência entre elas, em tempo de execução. Assim, uma parte do hardware é utilizada para a análise das dependências entre as instruções e outra parte para realizar a execução das múltiplas instruções paralelamente.

A principal diferença entre arquiteturas VLIW e super-escalares encontra-se na forma como é feito o escalonamento das instruções. Em máquinas VLIW o escalonamento é feito estaticamente, dessa forma, o compilador é responsável por expor o paralelismo que será

explorado pela arquitetura (SCARAFICCI, 2006). Por outro lado, as máquinas super-escalares realizam o escalonamento dinâmico. Neste caso, a busca por candidatas a serem executadas em paralelo é feita em tempo de execução, através do hardware.

A maior vantagem das arquiteturas VLIW sobre as super-escalares está no fato das primeiras possuírem o hardware bem mais simples e barato do que as segundas, uma vez que aumentar a complexidade no software (compilador), para a exploração de paralelismo é menos custoso do que aumentar a complexidade no hardware. Além disso, o compilador possui escalabilidade a um custo mais baixo. Isto significa que alterações no compilador são também são mais simples e menos custosas do que alterações no hardware.

A principal desvantagem das arquiteturas VLIW está relacionada com a compatibilidade binária. O código executável gerado pelo compilador pode não ser compatível com diferentes versões do próprio compilador. Além disso, o compilador gera o código para a arquitetura atual, isso significa que modificações na arquitetura implicam em atualização do compilador. (FISHER et al, 2004) apresenta mais detalhes sobre máquinas VLIW e super-escalares.

As arquiteturas reconfiguráveis são uma evolução desses dispositivos. Muitas técnicas aplicadas em máquinas VLIW e super-escalares são hoje adotadas no projeto de arquiteturas reconfiguráveis. A exploração de paralelismo pelo compilador é aplicada em arquiteturas reconfiguráveis estáticas. Nessas arquiteturas, o compilador além de ser utilizado para encontrar paralelismo nas aplicações, também é responsável por gerar a configuração que será utilizada pela arquitetura, de acordo com as instruções selecionadas.

Por outro lado, a busca por paralelismo em tempo de execução, utilizada em máquinas super-escalares, é aplicada em arquiteturas que são configuradas durante a execução da aplicação, denominadas arquiteturas reconfiguráveis dinâmicas (SANCHEZ et al, 1999).

A exploração de paralelismo pela arquitetura reconfigurável apresentada nesse trabalho é realizada estaticamente. Portanto, as técnicas de exploração de ILP utilizadas por arquiteturas VLIW também são adotadas em RoSA. Detalhes sobre essas técnicas são apresentados a seguir.

2.2 Técnicas de Compilação para a Extração de ILP

Para explorar paralelismo no nível de instrução através do compilador, o primeiro passo é analisar as dependências de controle e de dados da aplicação, com o objetivo

de encontrar instruções independentes que possam ser executadas paralelamente (POZZI, 2000).

As dependências de controle ocorrem quando o código não segue sua ordem seqüencial. O salto (do inglês *jump*) é um exemplo de instrução que quando encontrada altera o fluxo de execução da aplicação. Por outro lado, as dependências de dados obrigam a execução do código a seguir o caminho seqüencial de trechos da aplicação. Existem dois tipos de dependências de dados: as verdadeiras e as falsas.

Dependências verdadeiras ocorrem quando dados de saída de uma instrução são entrada de outra instrução. Dessa forma, a execução da segunda instrução depende do resultado gerado pela primeira.

As falsas dependências, também chamadas de antidependências, ocorrem quando duas instruções diferentes utilizam o mesmo local de armazenamento físico. Em outras palavras, as falsas dependências ocorrem quando duas instruções diferentes utilizam o mesmo registrador para alocar um dado. A Figura 1 ilustra os tipos de dependências existentes.

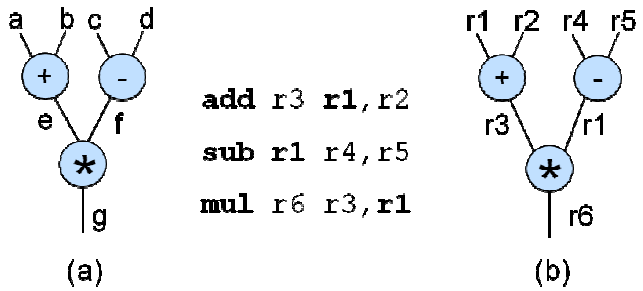

Figura 1. Dependências de Dados a) Verdadeira b) Falsa

De acordo com a Figura 1a, o resultado do operando *g* depende dos resultados de *e* e *f*, caracterizando uma dependência de dados verdadeira. Na Figura 1b os dados são representados pelos registradores alocados na execução das operações. Na figura, é possível observar que o registrador r1 é utilizado para leitura pela operação de adição e escrita pela subtração, por esse motivo, as duas instruções devem ser executadas seqüencialmente. Se o registrador para armazenar a saída da operação de subtração fosse diferente daquele utilizado na entrada da adição essas instruções seriam completamente independentes e poderiam ser executadas em paralelo. Por esse motivo dependências desse tipo são consideradas falsas.

A primeira vista a execução de instruções na ordem seqüencial parece o comportamento óbvio da aplicação, uma vez que os programas são escritos para serem seguidos dessa forma. Porém, a exploração do paralelismo no nível de instrução está em

procurar instruções na aplicação onde não exista nenhum tipo de dependência e executá-las em paralelo. Quando as dependências são encontradas, é possível aplicar técnicas que eliminam ou pelo menos diminuem a quantidade de ocorrências dessas dependências.

Quando ocorrem dependências de controle não é possível saber qual instrução será executada até que a condição que leva ao salto seja resolvida. Para eliminar essas dependências existem técnicas que utilizam predição e especulação que tentam estimar por algum critério qual será o caminho seguido. Geralmente a decisão tomada sobre o caminho é baseada no histórico daquela instrução ou em estatísticas resultantes de estudos sobre o comportamento das aplicações. Exemplos dessas técnicas são: predição de salto e execução especulativa (LAM e WILSON, 1992).

O ganho de desempenho alcançado por essas técnicas ocorre quando a decisão do caminho a ser seguido é acertada. Dessa forma, como não é mais necessário esperar pela resolução da condição, é possível paralelizar as instruções acelerando a execução da aplicação. Porém, quando a decisão tomada não é a correta, é necessário interromper a execução e iniciá-la no trecho que deveria ser executado desde o princípio.

Dessa forma, a utilização dessas técnicas só é interessante quando a sua eficácia é comprovada. Por exemplo, é possível perceber que para laços do tipo FOR, a possibilidade de acertar o caminho a ser tomado é bem maior do que para laços do tipo IF-THEN-ELSE, em que as duas técnicas muitas vezes não apresentam bons resultados.

Uma terceira técnica aplicada nas dependências de controle é a execução predicada (do inglês *predicated execution*) (PARK e SCHLANSKER, 1991). Essa técnica consiste em executar os dois caminhos que podem ser tomados após o salto antes de a condição ser resolvida. Dessa forma, após resolver a condição basta apenas validar o resultado correto e descartar o outro. A desvantagem dessa técnica é a demanda por hardware, uma vez que é preciso executar dois blocos de instruções ao mesmo tempo, referentes aos dois possíveis caminhos.

As técnicas para eliminação de dependências verdadeiras também são baseadas em previsões. A diferença é que nesse caso a previsão é feita através do histórico do operando. A geração desse histórico baseia-se no princípio da localidade de valor (do inglês *value locality*), que corresponde à probabilidade de um mesmo valor ser encontrado em sucessivos acessos ao conteúdo de um registrador ou endereço de memória (LIPASTI, 1997).

A exploração desse tipo de dependência é bastante recente. Até alguns anos atrás se considerava que dependências verdadeiras não poderiam ser eliminadas ou nem

sequer atenuadas. Esse problema era chamado de limite de fluxo de dado. Por esse motivo as técnicas atuais são aplicáveis apenas a processadores super-escalares, pois para implementá-las é necessário além do compilador, o suporte em tempo de execução. Caso a previsão do valor esteja errada, a instrução é recalculada em tempo de execução (RYCHLIK et al, 1998).

Por fim, a eliminação das falsas dependências de dados utiliza uma técnica denominada renomeação de registradores. Essa técnica realiza a troca dos registradores que causaram conflito por outros que não estejam sendo usados. A desvantagem dessa técnica está na possibilidade de esgotarem os registradores disponíveis e ser necessário armazenar os dados em memória.

A eliminação das dependências é apenas uma etapa realizada para prover uma maior quantidade de instruções independentes que possam ser executadas em paralelo. Além dessa abordagem, existem técnicas para a otimização da aplicação que expõem o paralelismo no nível de instrução através de alterações no código. Essas técnicas trabalham em dois escopos diferentes: o local, que trabalha dentro dos blocos básicos; e o global, que inclui as técnicas que trabalham entre blocos básicos.

2.2.1 Escopo Local

Muitas técnicas em escopo local são aplicadas nos laços, já que, na maioria das vezes, essas estruturas são as maiores responsáveis pelo tempo de execução da aplicação. Considerando que as instruções internas de um laço serão executadas em todas as suas iterações, reorganizá-las pode representar um ganho de desempenho significativo (JÚNIOR, 2006).

As técnicas aplicadas nos laços são chamadas de transformações de laço (do inglês *loop transformations*), e como exemplos têm-se: descascamento de laço (do inglês *loop peeling)*, fusão de laço (do inglês *loop fusion)*, desenrolamento de laço (do inglês *loop unrolling)*, dentre outras.

- Descascamento de laço: simplifica o laço removendo um pequeno número de iterações do início ou do final do laço para executá-las separadamente. Essa técnica pode ser aplicada com dois objetivos diferentes: eliminar as dependências criadas pelas iterações removidas permitindo o paralelismo, ou preparar o laço para aplicar uma segunda técnica denominada fusão de laço. A Figura 2 apresenta um exemplo de aplicação dessa técnica.

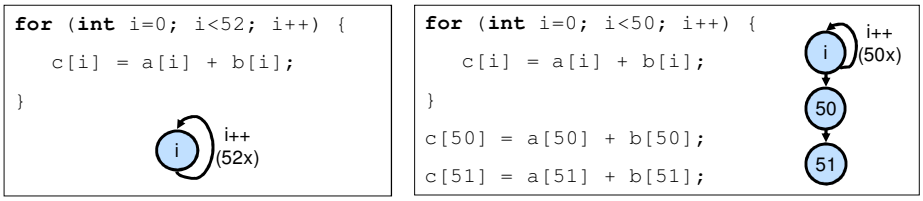

Figura 2. Descascamento de Laço

- Fusão de laço: corresponde a união de dois laços de mesma iteração em apenas um. Geralmente essa técnica é utilizada com o descascamento de laço, que primeiramente, retira do laço as instruções que estão fora do limite da iteração, para em seguida realizar a fusão dos dois laços com mesma iteração. A fusão de laço é ilustrada na Figura 3.

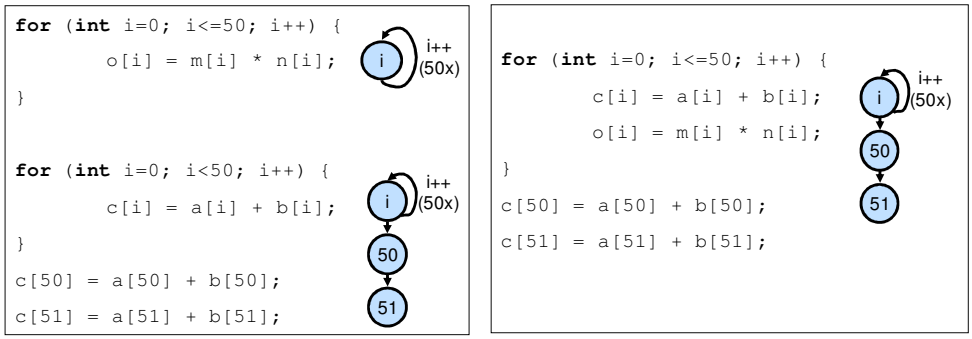

Figura 3. Fusão de Laço

- Desenrolamento de laço: diminui a quantidade de comparações realizadas no laço através da replicação das instruções internas. Para realizar o desenrolamento de laço é necessário replicar as instruções um número *n* de vezes e reiterar o laço em um fator *n*, como pode ser visto na Figura 4. As instruções replicadas podem ser executadas paralelamente.

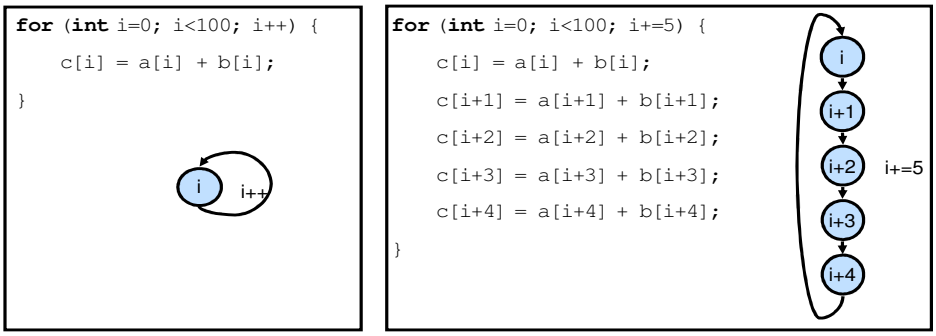

Figura 4. Desenrolamento de Laço

Mais detalhes sobre as técnicas descritas e outras técnicas de transformações de laço podem ser encontrados em (BACON, GRAHAM e SHARP, 1994).

2.2.2 Escopo Global

As técnicas utilizadas em escopo global atuam entre blocos básicos, no nível de grafo de controle da aplicação. Exemplos dessas técnicas são: o escalonamento do caminho crítico (do inglês *trace scheduling*) e o escalonamento de superbloco (do inglês *superblock scheduling*), que consistem em gerar blocos básicos maiores a partir da união de blocos básicos menores.

A principal desvantagem dessas técnicas está no fato de ambos considerarem um único caminho. Portanto, se o caminho escolhido for o errado, será necessário reiniciar a execução a partir do início do caminho correto.

As técnicas em escopo global estão fora do escopo desse trabalho. Maiores detalhes sobre essas técnicas podem ser encontrados em (POZZI, 1999).

2.3 Arquiteturas Reconfiguráveis

A primeira proposta de arquitetura reconfigurável surgiu em 1960 na Universidade da Califórnia (UCLA), Los Angeles. Gerald Estrin apresentou uma proposta de estrutura de computador fixa e variável (ESTRIN, 2002). A Figura 5 apresenta a proposta de Estrin que consistia em um processador padrão (F de fixo) aumentado por um *array* de hardware reconfigurável (V de variável), cujo comportamento era controlado pelo processador principal.

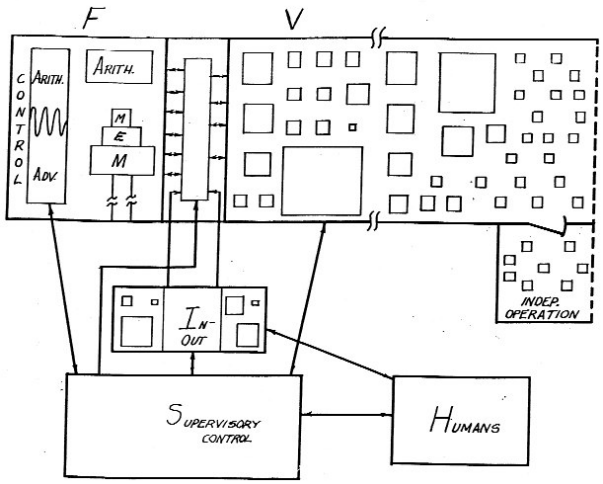

Figura 5. Primeira Proposta de Arquitetura Reconfigurável por Gerald Estrin, 1960. Retirado de (ESTRIN, 2002)

Apesar do impacto causado na época pela nova abordagem, foi apenas no final dos anos 1980 e início da década de 1990 que surgiram propostas de arquiteturas reconfiguráveis consolidadas e com resultados eficientes baseados em testes realizados com

conjuntos de aplicações reais. Essa época foi marcada pela evolução dos FPGAs (*Field Programmable Gate Arrays*) e pela demanda de arquiteturas customizadas para atender os requisitos de desempenho das aplicações cada vez mais complexas (DEHON, 1996).

De acordo com a definição apresentada no capítulo 1, as arquiteturas reconfiguráveis são capazes de mudar seu comportamento de acordo com as restrições da aplicação. Essa propriedade permite que arquiteturas reconfiguráveis alcancem altas taxas de desempenho, que podem ser comparadas a arquiteturas de aplicação específica (ASICs), além de exibirem flexibilidade que até algum tempo atrás era encontrado apenas nos processadores de propósito geral.

O projeto de arquiteturas reconfiguráveis envolve diversos fatores que determinam as características do hardware a ser desenvolvido. Geralmente, esses fatores estão relacionados ao tipo de arquitetura reconfigurável de acordo com alguns critérios de classificação descritos a seguir.

2.3.1 Classificação

As principais classificações são quanto: à granularidade, ao grau de acoplamento da lógica reconfigurável e à metodologia de reconfiguração (BONDALAPATI e PRASANNA, 2002).

Granularidade

A granularidade está relacionada ao nível de abstração de uso de hardware, podendo ser fina, grossa ou mista.

Arquiteturas reconfiguráveis de granularidade fina baseiam-se em dispositivos lógicos programáveis como os FPGAs (HAUCK, 1998) e incluem unidades reconfiguráveis (CLBs – *Configurable Logic Blocks*) que executam funções de largura de um bit ou de uma pequena quantidade de bits. Por outro lado, granularidade grossa refere-se a arquiteturas que trabalham com unidades reconfiguráveis com largura de palavras ou até mesmo pequenos microprocessadores distribuídos em um *array* de unidades de processamento (TODMAN, 2005). Por fim, arquiteturas de granularidade mista, como o próprio nome indica, utilizam as duas abordagens, incluindo em um mesmo hardware tanto dispositivos de granularidade grossa quanto de granularidade fina.

As vantagens das arquiteturas de granularidade grossa sobre as de granularidade fina estão relacionadas principalmente com a redução do tempo de

configuração, redução da área utilizada para rotear as funções e a diminuição da complexidade da arquitetura em geral (HARTENSTEIN, 2001a).

Grau de Acoplamento da Lógica Reconfigurável

A maioria das arquiteturas reconfiguráveis encontradas na literatura são compostas por um processador e uma ou mais lógicas reconfiguráveis. Esse tipo de estrutura é chamada de arquitetura reconfigurável híbrida (LEVINE e SCHMIT, 2003).

O processador anexado à lógica reconfigurável é geralmente chamado de processador principal ou hospedeiro (do inglês *host*). Suas tarefas são: executar as funções de controle para configurar à lógica reconfigurável; escalonar os dados de entrada e saída; realizar a interface externa; indicar a lógica reconfigurável o momento de executar; dentre outras.

O grau de acoplamento do processador hospedeiro à lógica reconfigurável influencia diretamente nos custos de reconfiguração e acesso aos dados (BONDALAPATI e PRASANNA, 2002). Basicamente, existem dois tipos de grau de acoplamento entre processador e bloco reconfigurável: fracamente acoplado e fortemente acoplado.

Arquiteturas reconfiguráveis fracamente acopladas usam o bloco reconfigurável como co-processador ou acelerador. Podem estar localizados dentro do chip do processador ou fora. Nesse tipo de arquitetura as tarefas de alto custo computacional são executadas no bloco reconfigurável sem a intervenção do processador.

Embora essa abordagem geralmente proporcione melhoras significativas no desempenho do hardware, uma de suas desvantagens é a demanda por recursos de hardware para executar as tarefas complexas, além do fato de que apenas algumas tarefas podem ser executadas na arquitetura (CHEN e MASKELL, 2007).

O segundo grau de acoplamento corresponde às arquiteturas reconfiguráveis fortemente acopladas. Nessa abordagem, ao contrário do co-processador que só executa quando o processador hospedeiro pára a execução, tanto o bloco reconfigurável quanto o processador executam ao mesmo tempo. A lógica reconfigurável é integrada no caminho de dados do processador hospedeiro, servindo como unidades funcionais adicionais.

Essa abordagem reduz o custo de comunicação entre processador e lógica reconfigurável. Porém, sua capacidade de paralelização é limitada pela quantidade de lógica reconfigurável disponível (COMPTON e HAUCK, 2002).

Metodologia de Reconfiguração

Em arquiteturas reconfiguráveis, uma das tarefas mais críticas consiste em configurar o hardware. Essa tarefa deve ser realizada toda vez que uma aplicação for executada na lógica reconfigurável. Por esse motivo, existe um limite de tempo que pode ser gasto na geração e carregamento da configuração para que esta não seja um gargalo para o projeto.

Existem duas metodologias para configurar o hardware: a configuração estática e a dinâmica. A primeira consiste em configurar o hardware apenas uma vez e não mudá-la durante toda a execução da aplicação. Essa abordagem utiliza o compilador, e por esse motivo também é chamada de reconfiguração em tempo de compilação.

A reconfiguração estática é utilizada para acelerar trechos da aplicação muito custosos para serem executados no processador. Dessa forma, a reconfiguração em tempo de compilação é utilizada em co-processadores ou aceleradores.

Na reconfiguração dinâmica ou reconfiguração em tempo de execução, o hardware pode ser configurado mais de uma vez durante a execução da aplicação. Nessa abordagem a aplicação é divida em segmentos que são executados sucessivamente utilizando a mesma lógica reconfigurável.

Enquanto a reconfiguração estática é vantajosa por introduzir complexidade apenas uma vez durante a compilação, a reconfiguração dinâmica não é limitada a trechos de código, podendo executar qualquer parte da aplicação.

Algumas arquiteturas realizam configuração em tempo de execução em apenas parte do bloco reconfigurável. Essa abordagem evita consumo desnecessário de energia e de recursos de hardware e é chamada de reconfiguração dinâmica parcial.

2.3.2 Trabalhos Relacionados

Antes de apresentar alguns trabalhos relacionados, é preciso ressaltar que a classificação das arquiteturas reconfiguráveis pode variar entre diferentes autores. Por exemplo, é possível encontrar na literatura autores que denominam arquiteturas de granularidade mista de arquiteturas híbridas e alguns consideram que a lógica reconfigurável é fortemente acoplada por está localizada no mesmo chip do processador, embora a função dela seja de co-processador. Para esse trabalho a classificação utilizada foi a que a autora considerou mais adequada dentre as encontradas na literatura. Essa classificação foi baseada de (POZZI, 2000).

A Tabela 1 apresenta algumas das principais arquiteturas reconfiguráveis encontradas na literatura com suas respectivas classificações. Uma breve descrição de algumas das arquiteturas citadas é apresentada a seguir. Mais detalhes sobre essas arquiteturas podem ser encontrados nas referências indicadas no final desse documento.

Além das arquiteturas apresentadas na Tabela 1, é possível citar diversas outras, como: MATRIX (MIRSKY e DEHON, 1996); DReAM (ALSOLAIM et al, 2000); REMARC (MIYAMORI e OLUKOTUM, 1998); CHESS (MARSHAL et al, 1999); QUKU (SHUKLA, BERGMANN, e BECKER, 2006); dentre outras.

Tabela 1. Principais Arquiteturas Reconfiguráveis encontradas na Literatura

	Referência	Granularidade	Grau de Acoplamento	Metodologia de Reconfiguração	Domínio de Aplicações
Chimaera	(HAUCK, 2004)	Fina	Forte	Dinâmica	Aplicações de propósito geral
GARP	(HAUSER e WAWRZYNEK, 1997)	Fina	Fraco	Estática	Processamento de imagem, criptografia
HASTE	(LEVINE e SCHMIT, 2003)	Grossa	Fraco	Dinâmica	Processamento de imagem, sinais e vídeo, Criptografia, Codificação de Canal
MorphoSys	(SINGH, 1998)	Grossa	Fraco	Dinâmica	Processamento de imagem
OneChip	(JACOB e CHOW, 1999)	Fina	Forte	Dinâmica	Controles embarcados, aceleradores de aplicação
PipeRench	(GOLDSTEIN, 1999)	Grossa	Fraco	Dinâmica	Aplicações baseadas em fluxo uniforme
RaPid	(EBELING, CRONQUIST e FRANKLIN, 1996)	Grossa	Fraco	Estática	Arrays sistólicos, computação intensiva
XPP-III	(XPP TECHNOLOGIES, 2006)	Grossa	Processador*	Dinâmica	Multimídia, simulação, processamento digital de sinais, criptografia
X4VLIW	(AZEVEDO, 2005)	Grossa	Processador*	Estática e Dinâmica	Aplicações de propósito geral
---	(BECK e CARRO, 2005)	Grossa	Forte	Dinâmica	Aplicações de propósito geral

**Essas arquiteturas não possuem processador de propósito geral. Ao invés disso, as arquiteturas podem ser programadas para agirem como um processador.*

Chimaera

Ao contrário de muitas arquiteturas cujo bloco reconfigurável é um recurso fixo, Chimaera considera a lógica reconfigurável como uma cache para instruções do tipo reconfigurável, chamadas de instruções RFU (*Reconfigurable Functional Unit*). Dessa forma, as instruções que foram executadas recentemente ou que serão executadas logo (de acordo

com a previsão), são mantidas na lógica reconfigurável. Caso outra instrução seja requisitada ela substitui uma ou mais instruções atuais que estão na lógica reconfigurável.

Para gerenciar a lógica reconfigurável de forma a permitir a implementação da estratégia de cache de instruções, a arquitetura Chimaera utiliza técnicas de reconfiguração dinâmica parcial. O uso das instruções RFU é feito através de chamadas à RFU embutidas no código da aplicação e o mapeamento da respectiva RFU está contido no segmento de instrução da aplicação.

A Figura 6 apresenta uma visão geral da arquitetura Chimaera retirada de (HAUCK, 2004). O principal componente da arquitetura é o *array* reconfigurável, que consiste de um FPGA projetado para suportar computações de alto desempenho. O banco de registradores do bloco reconfigurável mantém uma cópia do conteúdo do banco de registradores do processador (por esse motivo é chamado *shadow* – sombra em inglês). O componente CAM (*Content-addressable Memory*) indica quais das instruções presentes na lógica reconfigurável foram completadas. O controle de *caching/prefetching* interrompe o processador e carrega a instrução RFU da memória para o *array* reconfigurável.

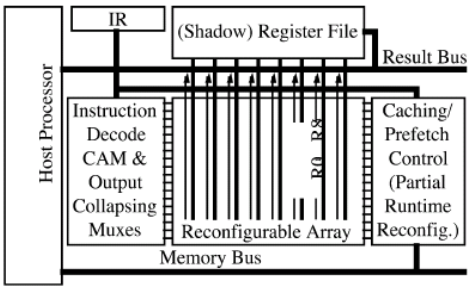

Figura 6. Chimaera. Retirado de (Hauck, 2004)

A principal vantagem dessa arquitetura está no fato do *array* reconfigurável executar constantemente, baseado nos valores de entrada armazenados no banco de registradores do próprio *array*. Portanto, uma chamada ao bloco reconfigurável significa apenas buscar o valor do resultado. Vale salientar que isso ocorre apenas quando a configuração presente no *array* for utilizada. Para casos em que ainda é necessário buscar a configuração na memória, o tempo gasto pode introduzir atraso na execução de toda a aplicação.

Para mapear algumas operações em operações para RFU (chamadas de RFUOPs) de forma automática, foi desenvolvido um compilador C para Chimaera. O compilador foi implementado a partir do *framework* GCC e possui uma fase de otimização para Chimaera. O objetivo do compilador é identificar seqüências de operações com múltiplas

entradas de dados e uma única saída e transformá-las em operações para a RFU. Os detalhes sobre o compilador e a fase de otimização para Chimaera podem ser encontrados em (YE, SHENOY e BANERJEE, 2000).

X4VLIW

Para apresentar a arquitetura X4VLIW é necessário primeiramente fazer uma introdução sobre a arquitetura X4CP32 que deu origem ao projeto.

X4CP32 foi desenvolvida pelo Grupo de Concepção de Sistemas Integrados, do Departamento de Informática e Matemática Aplicada da UFRN e consiste de uma arquitetura que combina os paradigmas de reconfiguração e multiprocessamento para obter programabilidade (AZEVEDO et al, 2005).

A arquitetura consiste de uma grade (do inglês *grid*) de unidades reconfiguráveis e programáveis (URP). Cada URP possui quatro células distribuídas em linhas e colunas e é conectada a todas as outras da mesma linha e a todas as outras da mesma coluna através de barramentos, como ilustrado na Figura 7a.

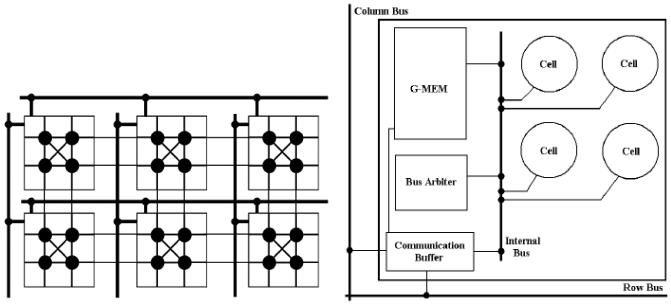

Figura 7. X4CP32 a) Conexões URP b) URP. Retirado de (AZEVEDO et al, 2005)

A Figura 7b ilustra uma URP composta por quatro células, uma memória interna, um buffer de comunicação, um barramento interno, um árbitro de barramento e uma lógica de controle.

A arquitetura X4CP32 possui um modo de execução onde estão implementados os dois paradigmas citados anteriormente. No modo de execução de programação, a URP age como um processador paralelo. A célula no canto superior esquerdo busca e envia instruções para as outras células, que executam essas instruções. No modo de execução Reconfigurável, a URP configura as entradas, operações e saídas, além do roteamento do caminho de dados.

As principais alterações da arquitetura X4CP32 para a X4VLIW foram: a implementação da URP de acordo com a metodologia VLIW. Assim, cada instrução

manipulada pela arquitetura passa a ser composta de quatro operações, distribuídas entre as células da URP. E o projeto das células em *pipeline*, que dividiu a arquitetura das células em três estágios independentes: decodificação de instrução e busca de operandos, execução, e escrita.

Essas alterações proporcionaram um aumento na quantidade de instruções por célula executada pela URP em oito vezes. Porém, como conseqüência, aumentou a área em aproximadamente 26% (AZEVEDO et al, 2005).

XPP-III

A arquitetura XPP-III (*eXtreme Processing Platform III*) é uma proposta da PACT XPP Technologies[1] que combina núcleos de processadores seqüenciais com um *array* reconfigurável de granularidade grossa. XPP-III é voltada tanto para aplicações regulares baseadas em fluxo de dados quanto para aplicações irregulares de fluxo de controle. Por esse motivo, suporta diferentes tipos de paralelismo, como: *pipelining*, nível de instrução, fluxo de dados e paralelismo no nível de tarefas. A arquitetura é apropriada para aplicações multimídia, de telecomunicações, simulação, processamento digital de sinais, criptografia e domínios de aplicações similares.

XPP-III é uma arquitetura de processamento de dados baseada em um *array* hierárquico de granularidade grossa, elementos de computação denominados *Processing Array Elements* (PAEs) e uma rede de comunicação orientada a pacote. A Figura 8 ilustra a estrutura do núcleo do XPP-III.

O núcleo do XPP-III é composto de três tipos de PAEs: os ALU-PAEs localizado no centro do *array*. À esquerda e a direita dos ALU-PAEs estão os RAM-PAEs com I/O (*Input/Output*). E a coluna no lado direito do *array* encontram-se os FNC-PAEs.

Os ALU-PAEs são formados por três ALUs (*Arithmetic-Logic Units*). Os RAM-PAEs contêm duas ALUs, uma pequena memória RAM, e um objeto de I/O. Finalmente, cada FNC-PAE contém um núcleo de processador seqüencial VLIW. Mais detalhes sobre a arquitetura do XPP-III podem ser encontrados em (XPP TECHNOLOGIES, 2006).

[1] http://www.pactxpp.com/main/index.php

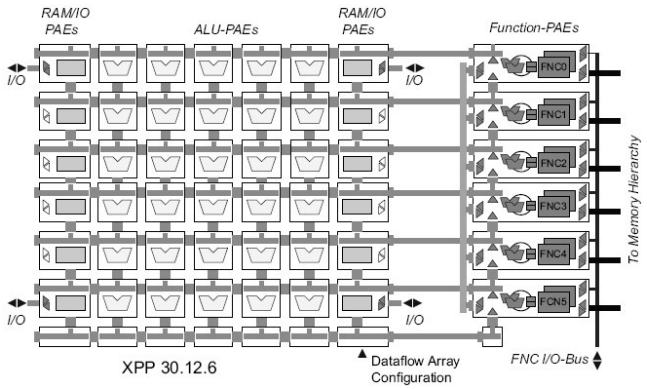

Figura 8. Estrutura do núcleo do XPP-III. Retirado de (XPP TECHNOLOGIES, 2006)

O núcleo apresentado na Figura 8 ilustra apenas uma estrutura do XPP-III. A quantidade de elementos de processamento varia de acordo com as necessidades de cada projeto. (BECKER et al, 2003) propõem uma arquitetura que integra um processador LEON (GAISLER, 2001) com um XPP composto de 16 ALU-PAEs e 8 RAM-PAEs.

Arquitetura Reconfigurável Proposta em (BECK e CARRO, 2005)

(BECK e CARRO, 2005) empregam o mecanismo de tradução binária, que traduz dinamicamente qualquer conjunto de instruções seqüenciais de uma aplicação Java em lógica combinacional. A lógica combinacional é então executada em uma arquitetura reconfigurável de granularidade grossa e fortemente acoplada ao processador.

De acordo com os autores, as vantagens dessa abordagem são: a possibilidade de aplicar a tradução binária em qualquer trecho da aplicação, inclusive em trechos onde existe pouco paralelismo a ser explorado; a tradução binária dinâmica em aplicações Java assegura a compatibilidade do software, eliminando a necessidade de ferramentas para o particionamento hardware/software e compiladores especiais; e a complexidade de configuração é menor por utilizar uma arquitetura reconfigurável de granularidade grossa.

A seleção dos trechos da aplicação que serão executados na arquitetura reconfigurável é feita através de uma unidade de detecção dinâmica, que analisa as instruções em tempo de execução.

A arquitetura reconfigurável é formada por blocos, denominados células e para a execução de um trecho da aplicação selecionado pela unidade de detecção pode ser utilizada uma ou mais células. A Figura 9 apresenta a arquitetura do *array* reconfigurável. Mais detalhes sobre as técnicas aplicadas e a arquitetura reconfigurável podem ser encontrados em (BECK e CARRO, 2005).

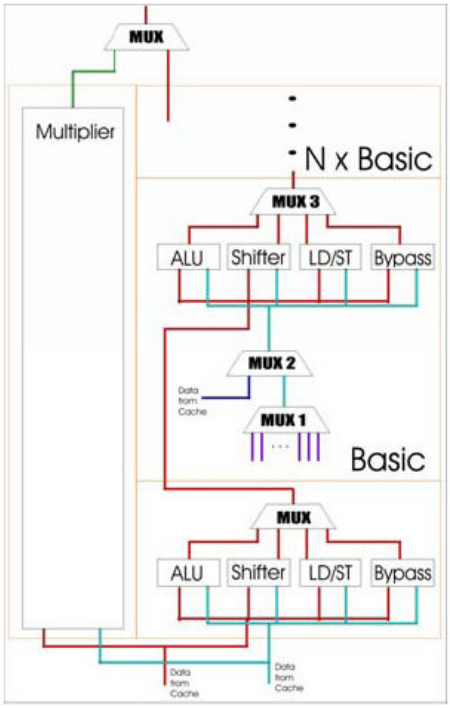

Figura 9. Arquitetura do *Array* Reconfigurável. Retirado de (BECK e CARRO, 2005)

A principal diferença entre RoSA e as arquiteturas reconfiguráveis mencionadas acima está no grau de acoplamento. Enquanto Chimaera e a arquitetura de (BECK e CARRO, 2005) são fortemente acopladas aos seus respectivos processadores hospedeiros e as arquiteturas XPP e X4VLIW podem ser programadas para trabalharem como um processador de propósito geral, RoSA é anexada ao processador hospedeiro através do barramento.

A utilização de barramento como modelo de comunicação possui como principal vantagem a facilidade de anexar o bloco reconfigurável a um processador possibilitando o acoplamento do bloco reconfigurável a um vasto conjunto de processadores existentes.

Dessa forma, apesar de diversos autores defenderem que arquiteturas reconfiguráveis fortemente acopladas são mais vantajosas por possuírem o tempo gasto com comunicação bastante reduzido, isto pode ser atenuado pela complexidade de incluir um bloco lógico adicional ao caminho de dados do processador. Já que isto introduz algumas implicações que devem ser consideradas, como a interferência no caminho crítico do processador e a ausência de flexibilidade em adaptar o bloco reconfigurável a outro processador. No caso de arquiteturas como XPP e X4VLIW o tempo gasto para a

programação dessas arquiteturas também pode aumentar o tempo de execução total da aplicação.

Capítulo 3

Técnicas de Compilação

Esse capítulo apresenta as técnicas de compilação aplicadas para a seleção dos trechos de código que serão executados no bloco reconfigurável. Uma análise do grafo de fluxo de dados e dos blocos básicos de diversas aplicações auxiliou na seleção das técnicas a serem aplicadas e no projeto da arquitetura.

É importante ressaltar que o foco desse trabalho é a elaboração e implementação da arquitetura reconfigurável, portanto, o compilador proposto nesse capítulo não foi desenvolvido. Porém, sua implementação faz parte dos trabalhos futuros, como descrito no capítulo 6.

3.1. Seleção de Trechos de Código

Para seleção dos trechos de código a serem executados no bloco reconfigurável foram escolhidas técnicas de compilação em escopo local.

Uma das formas mais utilizadas de exploração de paralelismo em escopo local é através da análise do grafo de fluxo de dados da aplicação. A análise objetiva identificar, em cada bloco básico, subgrafos independentes onde as saídas intermediárias e finais não interfiram em outro subgrafo.

Esses subgrafos são caracterizados por possuírem várias entradas e apenas uma saída. Por esse motivo, eles são chamados de MISOs (*Multiple Input Single Output*). A definição formal de MISO, de acordo com Alippi (ALIPPI et al, 1999) é apresentada a seguir:

Denote por $G^i = <V^i, E^i>$ um subgrafo onde V^i é o conjunto de nós em G^i e E^i é o conjunto de todas as arestas partindo dos nós. Uma aresta $e^i \in E^i$ é identificada pelo seu nó fonte ($v_k^i \in V^i$) e seu nó de destino v_l^i e é denotada por $e^i(k,l)$. Se $\forall v_k^i \in V^i$, com exceção do nó v_o^i é verdadeiro que:

$$\forall e^i(k,l) \in E^i,\ v_l \in V^i$$

Então G é um MISO. Um MISO é indicado como M^i, e v_o^i é seu nó de saída.

Figura 10. Definição Formal de MISO

No âmbito arquitetural, a seleção de MISOs significa restringir o registrador que armazena a saída das unidades reconfiguráveis com apenas uma porta de escrita. Isto simplifica o projeto da arquitetura, evitando conflitos de escrita que podem ocorrer com subgrafos que possuem mais de uma saída (POZZI, 2000). Outra vantagem está no fato de o

algoritmo de busca por determinados MISOs ser mais simples do que para outros tipos de subgrafos. Dessa forma, implementar uma ferramenta que automatize a extração desses MISOs também é mais simples.

Os MISOs independentes encontrados na aplicação são candidatos à execução paralela no bloco reconfigurável da arquitetura RoSA. Porém, para garantir que a execução dos MISOs na lógica reconfigurável seja vantajosa, a escolha dos subgrafos deve ser feita baseada em algumas restrições que consideram tanto o tamanho quanto a quantidade de MISOs. São elas:

1. *Se existir um único MISO, este deve ter um custo computacional alto o suficiente que justifique sua execução na arquitetura reconfigurável;*
2. *Um grande número de MISOs justifica a execução na arquitetura;*
3. *A combinação de custo computacional e uma quantidade intermediária de MISOs paralelos também pode justificar uma execução na arquitetura reconfigurável.*

Para exemplificar como é feita a análise de grafos de fluxo de dados de uma aplicação, a Figura 11 apresenta os subgrafos de um pequeno trecho de uma aplicação. A execução dos subgrafos segue a seqüência ilustrada na figura.

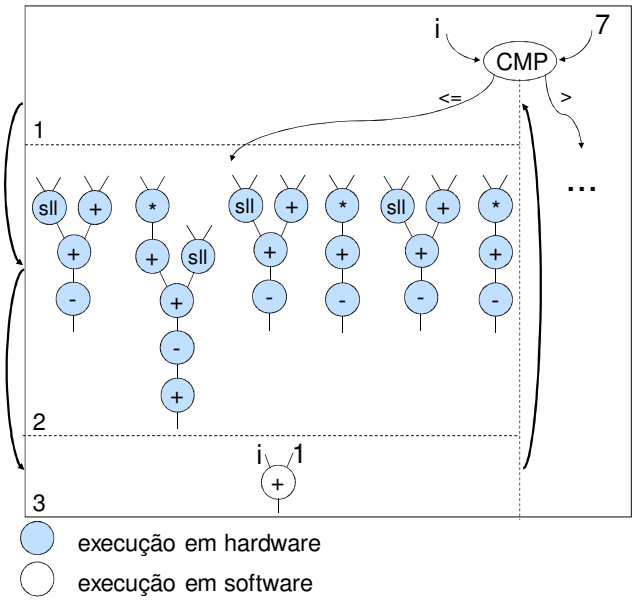

Figura 11. Exemplo de Subgrafos de Fluxo de Dados

De acordo com a figura, o primeiro subgrafo corresponde a uma comparação, que no exemplo apresentado, compara a variável *i* com o valor *7*. De acordo com a análise realizada, os resultados mostraram que não existe ganho de desempenho se a comparação for

executada no bloco reconfigurável. Portanto, todas as comparações são executadas no processador hospedeiro.

Após a comparação, o controle da execução pode apontar para o conjunto de subgrafos (2), ou para outros subgrafos fora do escopo do exemplo. Nesse caso, é necessário analisar se os subgrafos atendem as restrições citadas anteriormente. Se for o caso, uma configuração deve ser gerada considerando a execução na arquitetura RoSA. Para o trecho da aplicação que corresponde ao conjunto de subgrafos (2) é possível observar que existe uma quantidade significativa de MISOs independentes. Assim, esses MISOs são fortes candidatos a serem executados paralelamente na lógica reconfigurável.

Seguindo a seqüência da execução, o próximo subgrafo a ser executado corresponde ao número (3) da Figura 11. Esse trecho de código apenas atualiza o valor da variável *i*, após essa atualização a execução volta para a comparação. Portanto, essa operação também é executada no processador hospedeiro.

Na arquitetura RoSA a análise de fluxo de dados, verificação das restrições e o particionamento hardware/software são feitos pelo compilador. Por essa razão, assim como a arquitetura Chimaera (YE, SHENOY e BANERJEE, 2000), esse trabalho propõe a inclusão de uma fase de otimização entre as fases de um compilador GCC. A Figura 12 apresenta as fases do compilador, incluindo a fase de otimização RoSA, descritas a seguir.

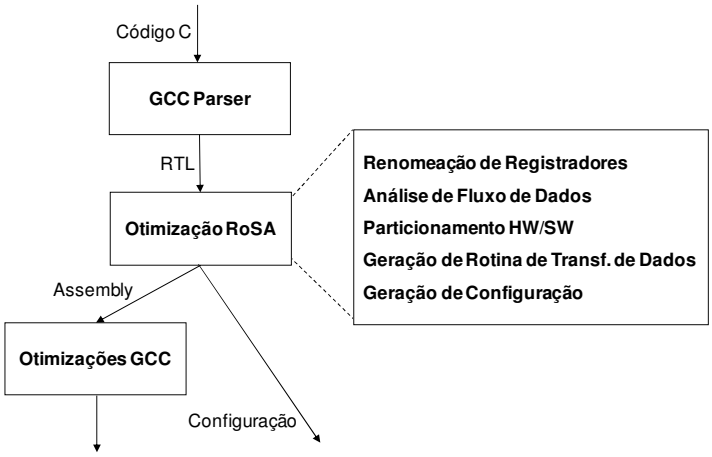

Figura 12. Fases de Compilação para a Arquitetura RoSA

A compilação do código para RoSA inicia como uma compilação regular: o código C é analisado e transformado em uma linguagem intermediária RTL (do inglês *Register Transfer Language*), em seguida o RTL entra na fase de otimização RoSA.

A otimização RoSA é composta por cinco etapas: renomeação de registradores, análise de fluxo de dados, particionamento hardware/software, geração de rotina de transferência de dados e geração de configuração.

A primeira etapa realiza a renomeação de registradores para resolver as falsas dependências, como explicado na seção 2.2 (página 6). A etapa seguinte corresponde à análise do fluxo de dados da aplicação. Nessa etapa um algoritmo gera o fluxo de dados da aplicação e procura por candidatos para a execução paralela. Os melhores candidatos são os MISOs que satisfazem as restrições mencionadas anteriormente.

Após a análise de fluxo de dados é feito o particionamento hardware/software. Essa etapa divide a aplicação em duas partes: a parte do processador (software), que corresponde ao código *assembly* e será enviada à fase de otimização do GCC. E a parte da lógica reconfigurável (hardware), que será enviada a última etapa da otimização RoSA. Nessa fase instruções especiais de reconfiguração são incluídas no código para delimitar o início e o fim de um bloco de código que será executado na arquitetura (bloco reconfigurável). Essas novas instruções devem ser incluídas no conjunto de instruções do processador hospedeiro.

A etapa seguinte consiste na geração da rotina de transferência dos dados do processador para a arquitetura. Todos os dados que serão utilizados por RoSA devem ser enviados ao banco de registradores da arquitetura. Os detalhes sobre a geração e execução da rotina são apresentados na seção seguinte.

A última etapa gera a configuração a partir dos MISOs selecionados. A configuração então é armazenada em memória e será acessada pelo bloco reconfigurável em um momento futuro.

3.2. Acesso aos Dados

A transferência de dados do processador para o bloco reconfigurável é realizada em dois momentos. Primeiro é feito o mapeamento dos dados armazenados nos registradores do processador e na memória externa para os registradores da arquitetura. O compilador é responsável por fazer esse mapeamento, através da geração de uma rotina em *assembly*, que envia os dados para o bloco reconfigurável. Também é nessa etapa que o compilador gera a instrução de envio do endereço de configuração.

Uma vez que o compilador conhece a localização de todos os dados da aplicação, cabe a ele indicar onde os dados que serão usados pela arquitetura se encontram imediatamente antes de um bloco reconfigurável ser iniciado. Assim, a rotina gerada por ele

deve conter instruções em *assembly* para copiar os dados para o banco de registradores do bloco reconfigurável. A Figura 13 ilustra um exemplo de rotina de transferência de dados. Na figura são apresentados os dados em linguagem C e as instruções em *assembly* que correspondem à transferência desses dados.

A segunda parte de transferência de dados ocorre durante a execução da aplicação. No momento que uma instrução de início de bloco reconfigurável é detectada, o processador salta para a rotina de transferência de dados e a executa, em seguida envia o endereço da configuração para o bloco reconfigurável e este inicia sua execução.

```
int a = 3   movhi  r3, 1028     // endereço base de RoSA
            movi   r2,3         // a
            stwio  r2,0(r3)     // enviar dado para a posição 0 de RoSA

int b = 4   addi   r3,r3,4      // deslocamento de 32 bits - 1 palavra
            movi   r2,4         // b
            stwio  r2,0(r3)     // enviar dado para a posição 0+4 de RoSA

int c = 1   addi   r3,r3,8      // deslocamento de 64 bits - 2 palavras
            movi   r2,1         // c
            stwio  r2,0(r3)     // enviar dado para a posição 0+8 de RoSA

int d = 9   addi   r3,r3,12     // deslocamento de 96 bits - 3 palavras
            movi   r2,9         // d
            stwio  r2,0(r3)     // enviar dado para a posição 0+12 de RoSA

int e = 3   addi   r3,r3,16     // deslocamento de 128 bits - 4 palavras
            movi   r2,3         // e
            stwio  r2,0(r3)     // enviar dado para a posição 0+16 de RoSA

int f = 5   addi   r3,r3,20     // deslocamento de 160 bits - 5 palavras
            movi   r2,5         // f
            stwio  r2,0(r3)     // enviar dado para a posição 0+20 de RoSA

int g = 2   addi   r3,r3,24     // deslocamento de 192 bits - 6 palavras
            movi   r2,2         // g
            stwio  r2,0(r3)     // enviar dado para a posição 0+24 de RoSA

int h = 4   addi   r3,r3,28     // deslocamento de 224 bits - 7 palavras
            movi   r2,4         // h
            stwio  r2,0(r3)     // enviar dado para a posição 0+28 de RoSA

            movhi  r3,16        // início da rotina de envio de configuração
            addi   r3,r3, 5116  // deslocamento de 255
            movhi  r2,9
            addi   r2,r2,-13352 // endereço da configuração
            stwio  r2,0(r3)     // enviar endereço da configuração para RoSA
```

Figura 13. Rotina de Transferência de Dados

A figura acima ilustra uma rotina de transferência de oito dados de entrada e o endereço de uma configuração. Os dados são enviados para os primeiros oito registradores do banco de registradores de RoSA.

O envio dos dados e da configuração utiliza como referência o endereço base de RoSA no sistema. Assim, cada dado é enviado para o endereço base da arquitetura RoSA somado ao deslocamento do tamanho de cada palavra (32 bits). O endereço de configuração é sempre enviado ao registrador de número 255 de RoSA, que corresponde ao registrador

utilizado pelo gerenciador de configuração para consultar o endereço da configuração e buscá-la na cache ou na memória.

Capítulo 4

Arquitetura RoSA

RoSA é uma arquitetura reconfigurável híbrida voltada para a aceleração de aplicações baseadas em fluxo de dados. Seu nome foi inspirado na combinação das letras que indicam o significado da arquitetura em inglês: *RecOnfigurable Stream-based Architecture* (arquitetura reconfigurável baseada em fluxo de dados).

A Figura 14 apresenta a organização da arquitetura. RoSA consiste de um bloco reconfigurável fracamente acoplado a um processador hospedeiro e uma memória externa, que se comunicam através de um barramento.

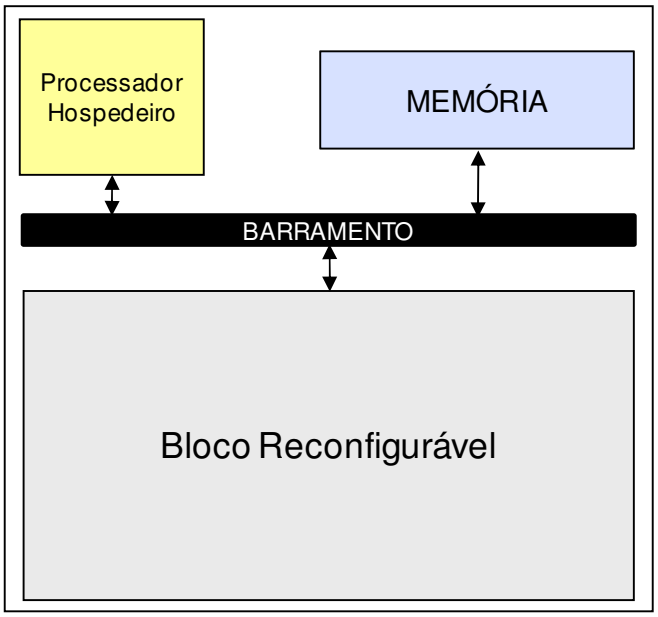

Figura 14. Organização da Arquitetura RoSA

Os principais componentes do bloco reconfigurável são: unidades reconfiguráveis (células), bancos de registradores e gerenciador de configuração. A Figura 15 apresenta um diagrama de blocos da arquitetura. A seção seguinte apresenta detalhes dos componentes da arquitetura, e o fluxo de execução é descrito na seção 4.2.

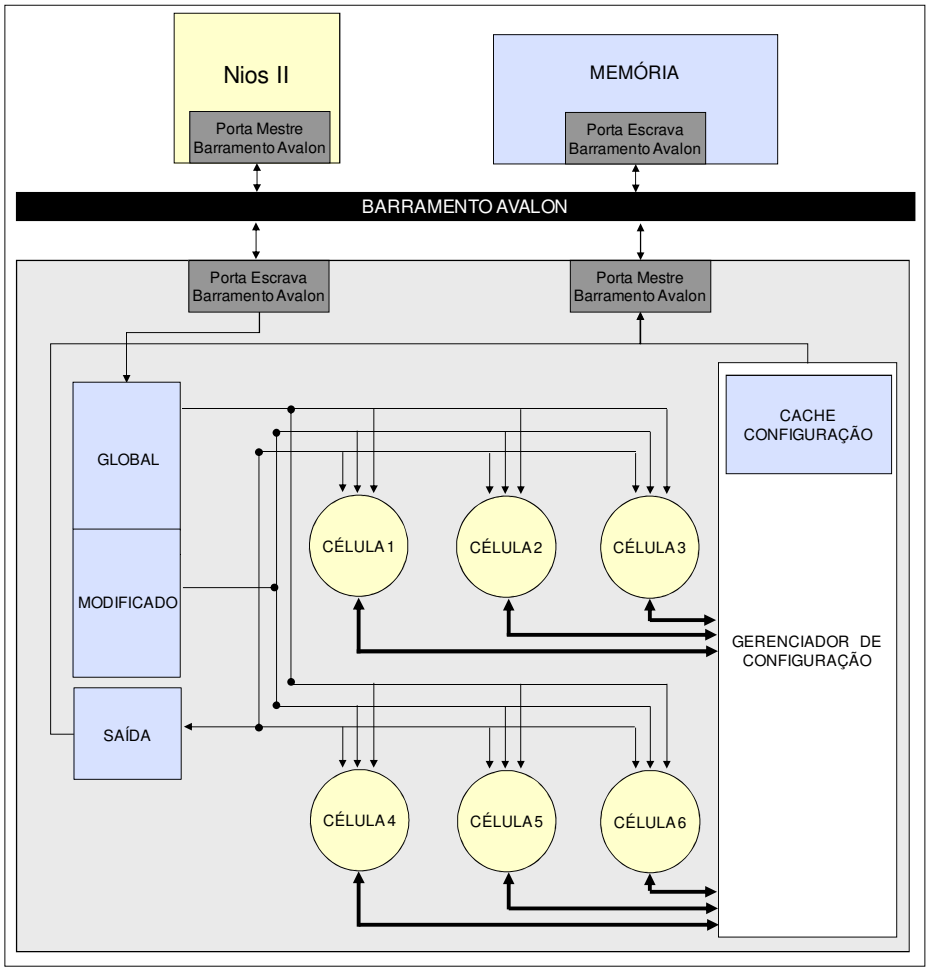

Figura 15. Diagrama de Blocos da Arquitetura RoSA

4.1. Componentes da Arquitetura

4.1.1 Processador hospedeiro

Os projetos de arquiteturas reconfiguráveis encontrados na literatura em sua maioria envolvem algum processador para realizar o controle do bloco reconfigurável e o escalonamento das instruções. Alguns exemplos são a arquitetura Garp que utiliza um processador MIPS (HAUSER e WAWRZYNEK, 1997) e a MorphoSyS, que possui um processador RISC (*Reduced Instruction Set Computer*) de 32 bits, chamado de Tiny RISC (SINGH et al, 1998).

Nesse projeto foi utilizado o processador Nios II, desenvolvido pela Altera (ALTERA CORPORATION, 2008b). O Nios II é um processador RISC, *soft-core*[2] e de propósito geral, projetado para atender uma grande variedade de aplicações. Visando o oferecimento de maior flexibilidade, a Altera desenvolveu um processador configurável, ao qual é possível adicionar novas instruções e periféricos de forma simplificada. A Figura 16 ilustra um exemplo de um sistema com um processador Nios II, disponível em um kit de desenvolvimento Altera Nios II (ALTERA CORPORATION, 2008b).

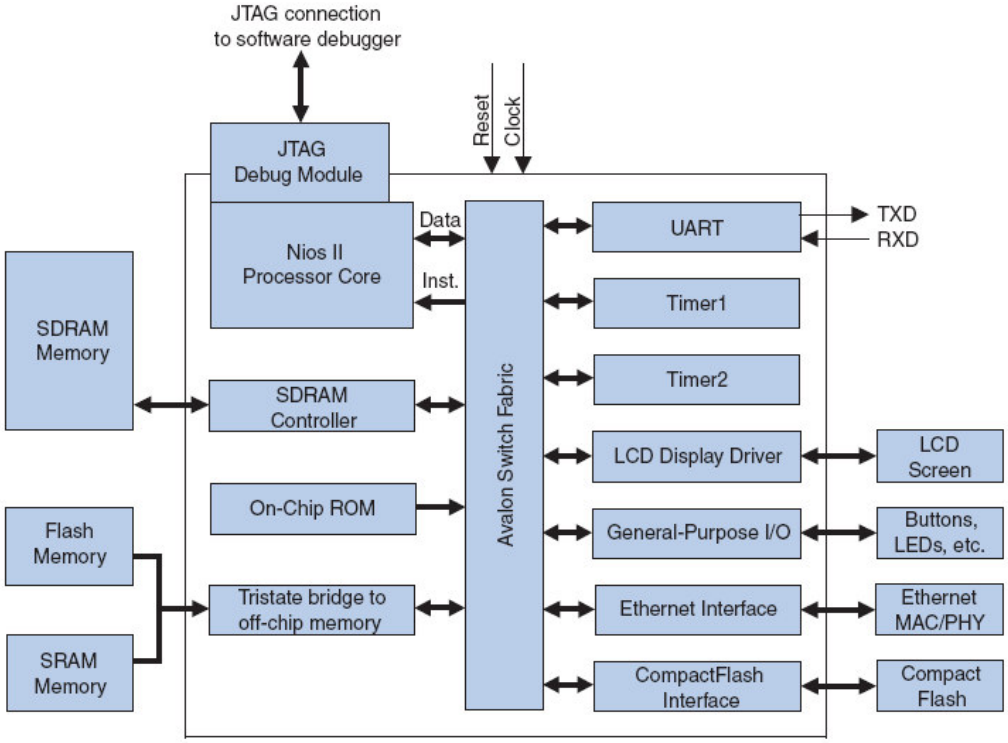

Figura 16. Exemplo de Sistema com Processador Nios II. Retirado de (ALTERA CORPORATION, 2008b)

A família de processadores Nios II disponibiliza três opções de núcleo. O Nios II/f (*fast*) é maior e mais rápido, com a maior quantidade de opções de configuração dentre os três núcleos. O núcleo padrão corresponde ao Nios II/s (*standard*), foi projetado menor do que o Nios II/f, porém mantendo o alto desempenho. Por fim, o núcleo mais simples é o Nios II/e (*economy*), projetado para ser o menor núcleo possível e como resultado, seu conjunto de características é limitado e, por esse motivo, muitas configurações não se encontram disponíveis.

[2] *Soft-core* significa que o processador foi descrito em software, geralmente em linguagem de descrição de hardware, e pode ser sintetizado em hardware programável, tal como FPGA. Com relação ao Nios II, este pode ser sintetizado para qualquer dispositivo da Altera.

No projeto apresentado neste documento foram utilizados os três núcleos da família Nios II. O bloco reconfigurável foi anexado a cada um dos núcleos disponíveis e para cada implementação foi avaliado o impacto do núcleo na área, na freqüência de execução e no desempenho da arquitetura.

Antes de adotar o Nios II como processador hospedeiro, estudos foram realizados utilizando o processador SPARC V8 (SUN MICROSYSTEMS, 2007). O SPARC V8 também é baseado na arquitetura RISC (do inglês *Reduced Instruction Set Computer*), e foi escolhido no início do projeto por já existir uma implementação em SystemC[3] com precisão de ciclos, desenvolvida pelo Grupo de Concepção de Sistemas Integrados do DIMAp.

No projeto da arquitetura RoSA esse processador foi utilizado para avaliar desempenho das técnicas aplicadas na arquitetura através da comparação entre a execução seqüencial no SPARC V8 e a simulação do comportamento paralelo da arquitetura. Esses resultados de desempenho são apresentados em (PEREIRA, OLIVEIRA e SILVA, 2007).

Embora o SPARC V8 tenha auxiliado no desenvolvimento do projeto, por estar implementado em SystemC, não foi possível utilizá-lo na arquitetura, que foi implementada em VHDL (NAVABI, 1998). Para selecionar outro processador foram realizados estudos comparativos entre o desempenho do SPARC V8 e o Nios II da Altera.

Os resultados da comparação entre os dois processadores, encontrados em (LOPES et al, 2007), demonstraram que o Nios II, com configuração padrão, apresentou melhor desempenho do que o SPARC V8. Em vista disso, o Nios II foi selecionado como processador hospedeiro da arquitetura RoSA.

A comparação foi realizada com o Nios II/s por ser a implementação com características equivalentes ao SPARC V8 (estágios de *pipeline*, uso de caches, dentre outras). Porém, para os resultados apresentados neste documento foram utilizados os três núcleos da família Nios II.

4.1.2. Célula

Como pode ser observado na Figura 15, o bloco reconfigurável possui seis unidades reconfiguráveis, denominadas células. O número de células foi definido baseado em uma análise de grafos de fluxo de dados de um conjunto de aplicações.

[3] http://www.systemc.org

As aplicações analisadas foram: a Transformada Rápida de Fourier (do inglês *Fast Fourier Transform* – FFT) e a sua inversa (IFFT) (RAMIREZ, 1985), a Transformada Discreta do Cosseno (do inglês *Discrete Cosine Transform* – DCT) (RAO e YIP, 1990), a JPEG (WALLACE, 1991) e a Estimação de Movimento H.264 (ITU, 2005).

A análise consistiu da geração do grafo de fluxo de dados das aplicações mencionadas acima. A partir do grafo de fluxo os MISOs independentes candidatos a execução paralela foram selecionados para execução na arquitetura e os MISOs dependentes entre si foram selecionados para a execução no processador hospedeiro.

A análise mostrou que em 80% de todos os grafos de fluxos das aplicações analisadas foram encontrados no máximo seis MISOs independentes que podem ser executados em paralelo. Para os outros 20% dos grafos de fluxo das aplicações é possível gerar duas configurações sem perda significante de desempenho.

Cada célula consiste de uma unidade reconfigurável projetada para executar um MISO por vez. As células se comunicam umas com as outras através de um banco de registradores, e o gerenciador de configuração indica quais registradores as células devem acessar.

A célula é composta de quatro unidades funcionais (UFs) que executam operações lógicas e aritméticas, um banco de registradores e uma unidade de controle, como ilustrado no diagrama de blocos da Figura 17.

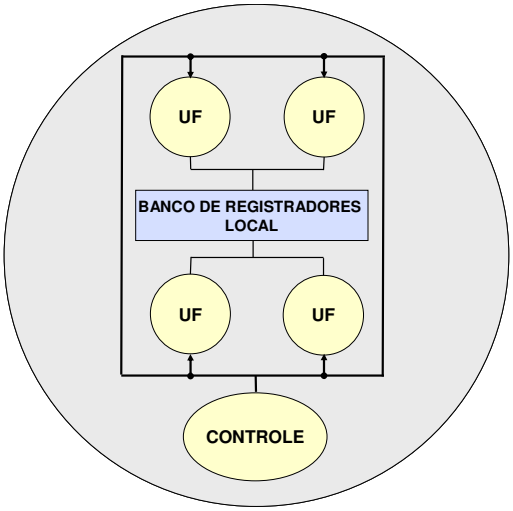

Figura 17. Diagrama de Blocos da Célula

Para definir a quantidade de unidades funcionais também foi utilizada a estratégia de análise de grafo de fluxo de dados aplicada às células. A análise mostrou que o maior subgrafo encontrado possui quatro nós em um mesmo nível. Isto significa que quatro

operações podem ser executadas em paralelo. A Figura 18a ilustra a estrutura do subgrafo mais largo encontrado (com a maior quantidade de operações em um mesmo nível) e na Figura 18b é ilustrado o subgrafo de maior profundidade (com a maior quantidade de níveis).

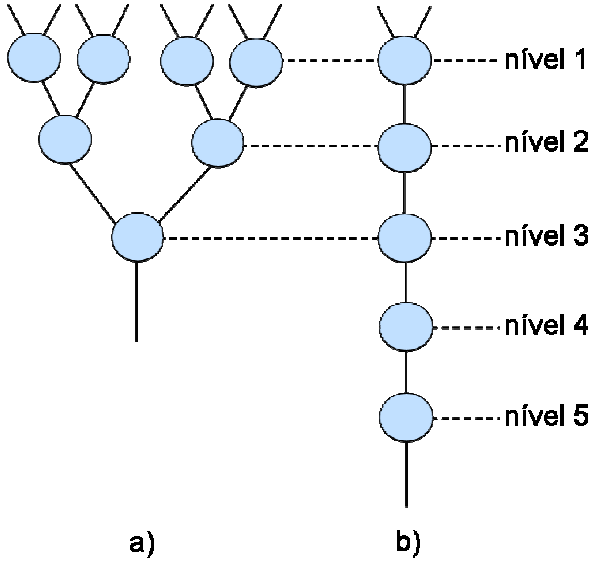

Figura 18. Grafo de Fluxo de Dados a) Mais largo b) Mais profundo

De acordo com a Figura 18, o MISO mais largo encontrado possui quatro operações em um nível e três níveis de operações. Enquanto o MISO de maior profundidade possui cinco níveis e uma operação em cada nível.

Da mesma forma que as células, as unidades funcionais se comunicam através de um banco de registradores. Esse banco armazena os cálculos intermediários de cada UF. A unidade de controle da célula é responsável por gerenciar as UFs e o banco de registradores de acordo com a configuração enviada pelo gerenciador de configuração.

As unidades funcionais foram projetadas de acordo com o conjunto de instruções do Nios II. Entretanto, algumas instruções não estão incluídas no conjunto de operações da arquitetura RoSA. Esse é o caso das instruções de transferência de dados; instruções de mover; de comparação; de controle de programa e outras instruções de controle, que são executadas apenas pelo processador. O Anexo I apresenta o conjunto de instruções do Nios II.

O resultado da análise dos grafos de fluxo de dados das aplicações, além de auxiliar na definição da quantidade de células e de unidades funcionais da arquitetura, também permitiu avaliar os padrões de operações encontrados nos grafos. Dessa forma, foi possível obter precisamente a quantidade e o tipo de operações diferentes existentes em cada nível dos subgrafos. A partir dessa análise, percebeu-se que apenas alguns conjuntos

específicos de operações estão presentes em um mesmo nível. Por exemplo, operações de multiplicação e divisão só aparecem em um nó em cada nível. Assim, a análise permitiu elaborar a arquitetura da célula visando à economia de área e conseqüentemente economia do custo total do projeto.

Baseado nessa análise, o caminho de dados da arquitetura RoSA foi projetado a partir de uma metodologia de reuso de hardware em caminho de dados. A metodologia RoSE visualiza os blocos operacionais das unidades funcionais através de níveis de reusabilidade, como ilustrado na Figura 19.

O nome da metodologia também foi inspirado em seu significado em inglês (*Reuse-based Standard Datapath Architecture* - padrão de reuso para caminho de dados) e na sua representação gráfica que assemelha-se a uma rosa, como pode ser visto na figura.

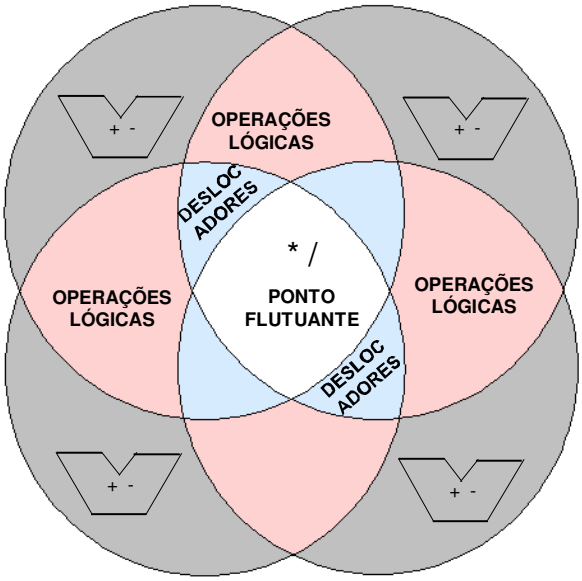

Figura 19. Metodologia RoSE

O primeiro nível de reusabilidade corresponde às operações mais custosas em hardware e/ou as menos usadas. Fazem parte desse nível as operações de multiplicação, divisão e operações com ponto flutuante. Nesse nível existe apenas um bloco operacional compartilhado entre as quatro UFs. Assim, operações desse tipo são sempre executadas seqüencialmente.

As áreas compartilhadas por três unidades funcionais representam o segundo nível de reusabilidade e incluem operações não usadas tão freqüentemente e menos custosas que as do primeiro nível. Pertencem a esse nível os deslocadores lógicos e aritméticos. Como

pode ser observado na figura, existem dois blocos operacionais que realizam essas operações, portanto é possível executar até duas operações de deslocamento em paralelo.

O terceiro nível consiste de operações pouco custosas em hardware e usadas com freqüência. As operações lógicas pertencem a esse nível, e como existem três blocos para realizar operações lógicas compartilhados entre as quatro unidades funcionais, é possível executar até três operações desse tipo em paralelo.

Finalmente, as operações utilizadas mais freqüentemente e as de menor custo em hardware estão no quarto nível de reusabilidade. Nesse nível todas as unidades funcionais possuem blocos operacionais. Incluem-se nesse nível as operações de soma e subtração, que são as mais freqüentes nos subgrafos.

Para avaliar a eficiência da metodologia três tipos de célula foram implementadas. O primeiro tipo aplica a metodologia RoSE, porém permite que as operações nos três primeiros níveis de reusabilidade possam ser executadas até quatros vezes em um mesmo nível, mesmo que seqüencialmente. Esse tipo foi implementado para poder executar qualquer aplicação, inclusive aquelas que possuam mais subgrafos independentes e mais operações nos subgrafos do que o encontrado nas aplicações analisadas até o presente momento. O segundo tipo também aplica a metodologia e restringe a quantidade de operações de acordo com a quantidade de blocos operacionais indicados na metodologia. Dessa forma, não é possível executar operações seqüenciais nessa implementação, reduzindo a quantidade de subgrafos que podem ser executados na célula. Por fim, o terceiro tipo não aplica a metodologia RoSE e possui quatro blocos operacionais para cada tipo de operação. Portanto, nesta implementação qualquer operação do conjunto de operações da arquitetura pode ser executada em paralelo. Os detalhes sobre cada implementação serão descritos a seguir.

Os três tipos de implementações são denominados: implementação com RoSE e sem limite de subgrafos; implementação com RoSE e com limite de subgrafos e implementação sem RoSE. As futuras referências a essas implementações serão feitas a partir desses nomes.

Vale ressaltar que, embora façam parte do conjunto de operações da célula, as operações de ponto flutuante não foram consideradas nesse trabalho e serão incluídas posteriormente, como indicado nos trabalhos futuros descritos no capítulo 6.

Célula com RoSE e sem limite de subgrafos

Nessa implementação a célula executa qualquer subgrafo que possua no máximo quatro operações no primeiro nível e cinco níveis de operações (como indicado na Figura 18, página 33).

A Figura 20 ilustra a disposição dos componentes na célula. A unidade de controle é responsável por ler a configuração, e selecionar os registradores e os blocos operacionais que serão utilizados. O caminho de dados é configurado de acordo com as informações sobre o formato do grafo e o endereço dos registradores que serão utilizados na execução.

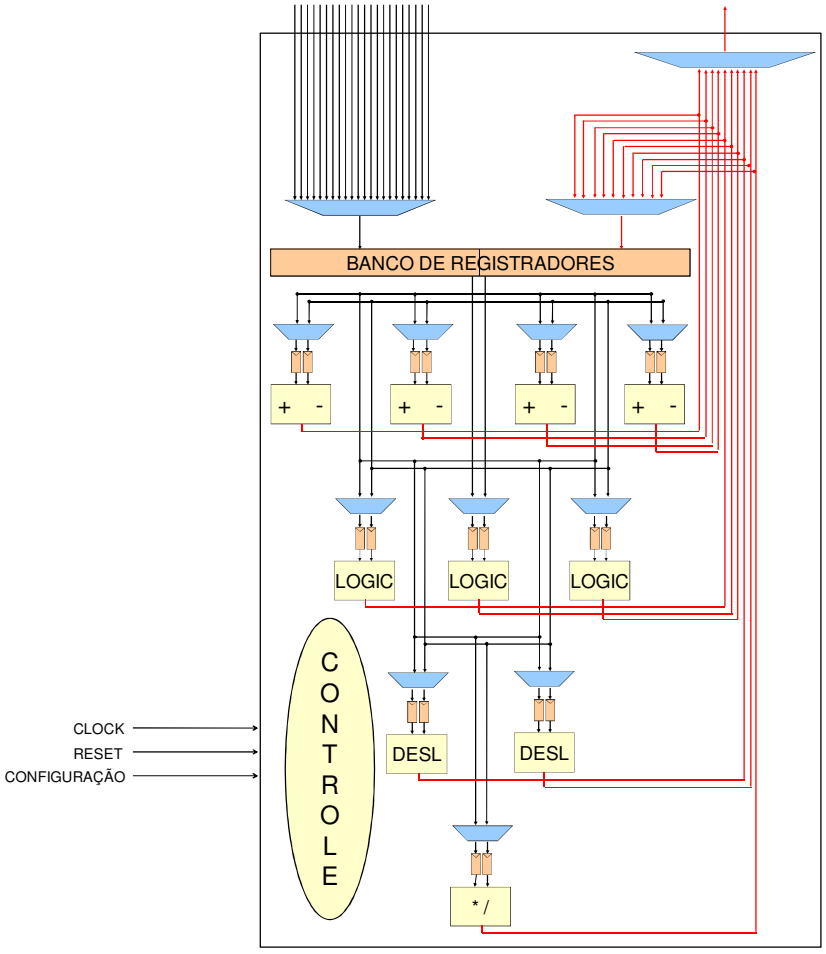

Figura 20. Diagrama de Blocos da Célula com RoSE

Para configurar a célula, a unidade de controle possui quatro máquinas de estados (MEs) que executam em paralelo as operações de cada um dos quatro níveis de reusabilidade. Porém, como a quantidade de blocos operacionais é limitada de acordo com a metodologia, algumas operações são executadas seqüencialmente.

A Figura 21 ilustra a máquina de estados mais complexa pertencente ao primeiro nível de reusabilidade. Os estados que aplicam a metodologia RoSE selecionam os blocos operacionais que serão utilizados e configuram o caminho de dados de cada nível. Após a execução das operações, os estados seguintes armazenam os resultados intermediários e o final nos bancos de registradores.

Figura 21. Máquina de Estados da Célula com Metodologia sem Limite de Subgrafos - 1º Nível de Reusabilidade

Os estados *aplicar_metodologia* habilitam a execução dos blocos operacionais e guardam as informações de que operações serão executadas em seguida, caso haja execução seqüencial. Os estados *armazer_saida*, como o próprio nome diz, indicam em quais registradores os resultados intermediários serão armazenados, além de indicar qual o resultado final da célula.

Uma vez que existe apenas um bloco operacional para executar operações do quarto nível de reusabilidade, a execução de mais de uma operação em um mesmo nível de

subgrafo é feita seqüencialmente. Assim, caso exista mais de uma operação, a primeira é executada e enquanto seu resultado é armazenado, a outra operação é executada.

A única diferença entre as MEs do segundo e do terceiro nível de reusabilidade e a apresentada na Figura 21 está na quantidade de operações executadas. Enquanto no primeiro nível de reusabilidade as operações são executadas seqüencialmente, no segundo nível podem ser executadas até duas operações em paralelo e no terceiro nível podem ser executadas até três operações.

A máquina de estados do quarto nível de reusabilidade é ilustrada na Figura 22. Por possuir os quatro blocos operacionais, esse nível possui a máquina de estados mais simples, uma vez que todas as operações podem ser executadas em paralelo.

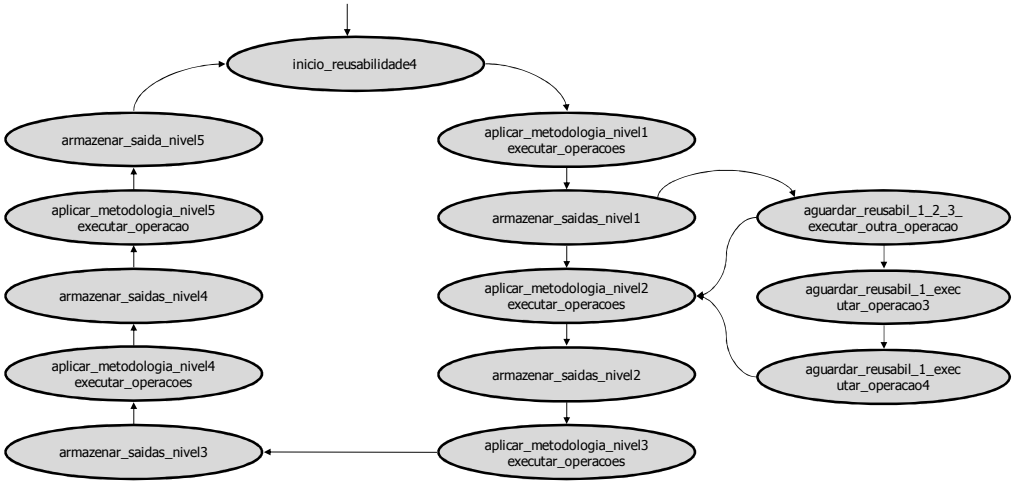

Figura 22. Máquina de Estados da Célula com Metodologia sem Limite de Subgrafos - 4º Nível de Reusabilidade

É importante ressaltar que quando um subgrafo possui operações que devem ser executadas seqüencialmente, enquanto essas operações estão sendo executadas, as máquinas de estados dos outros níveis de reusabilidade aguardam antes de iniciar a execução do próximo nível de subgrafo (estados *aguardar* nas máquinas de estados das figuras 21 e 22).

Além das máquinas de estados, a célula também possui um processo que recebe os dados do banco de registradores e outro que analisa a configuração para indicar quantas e quais são as operações em cada nível do subgrafo, indicando para as MEs o que executar e o momento de iniciar. Um mecanismo de sincronização, baseado em sinais, indica o início da execução das máquinas de estados e estas indicam o término de sua execução.

Célula com RoSE e com limite de subgrafos

Essa implementação da célula executa um conjunto restrito de subgrafos, cuja quantidade de operações não ultrapassa a quantidade de blocos operacionais disponíveis. Dessa forma, subgrafos que possuam mais de uma operação de multiplicação, divisão ou ponto flutuante (primeiro nível de reusabilidade); com mais de duas operações de deslocamento (segundo nível de reusabilidade) ou então com mais de três operações lógicas (terceiro nível) não podem ser executadas nessa célula.

Embora essa célula execute uma quantidade restrita de subgrafos, essa limitação é resultado dos padrões de grafos encontrados nas aplicações analisadas. Portanto, os subgrafos fora do escopo dessa implementação não interferem na execução das aplicações avaliadas uma vez que eles não foram detectados na análise.

A diferença entre essa implementação e a célula sem limitações de subgrafos é refletida apenas nas máquinas de estados da unidade de controle. A arquitetura apresentada na Figura 20 (página 36), e o restante da unidade de controle é igual para as duas implementações.

A Figura 23 apresenta a máquina de estados do primeiro nível de reusabilidade. Como pode ser observado, só é possível executar uma operação em cada nível de subgrafo (na ME da Figura 21 é possível executar até quatro operações em paralelo no primeiro nível de subgrafo, duas do terceiro ao quarto nível de subgrafo e uma no quinto nível).

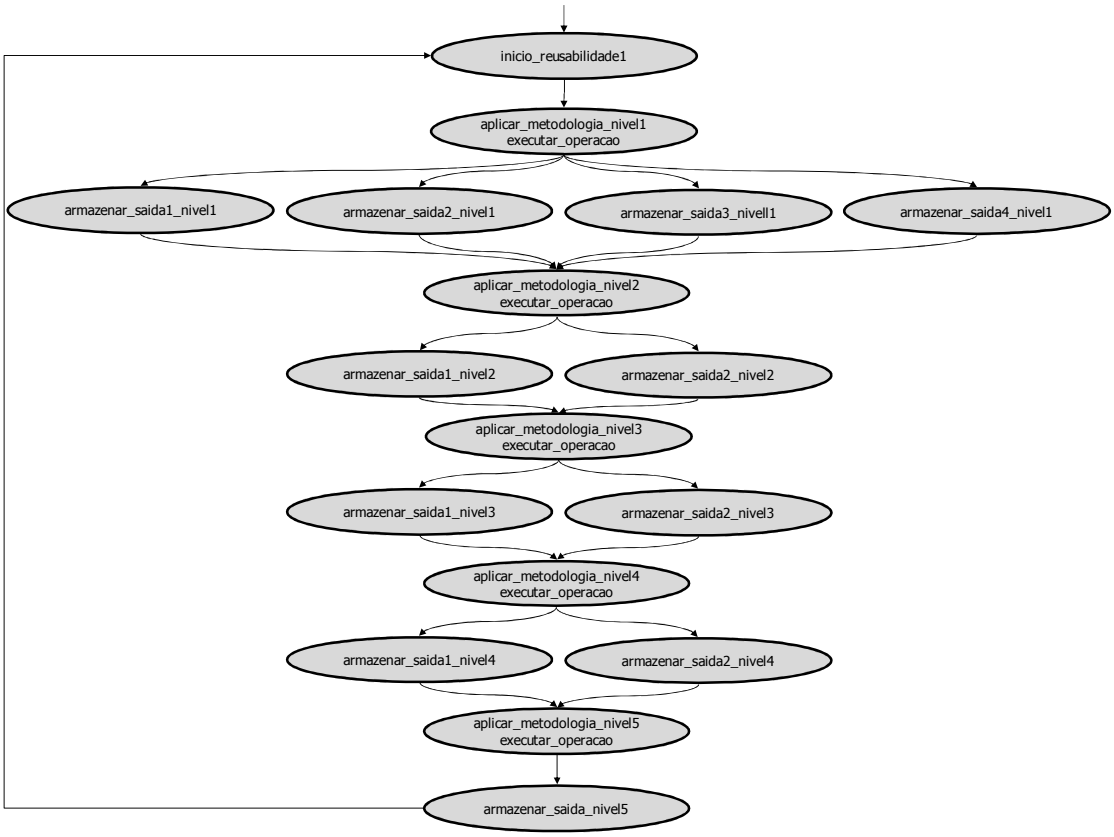

Figura 23. Máquina de Estados da Célula com Metodologia e com Limite de Subgrafos - 1º Nível de Reusabilidade

A máquina de estados do quarto nível de reusabilidade é ilustrada na Figura 24. Como nessa implementação os níveis de reusabilidade com quantidade restrita de blocos operacionais (primeiro, segundo e terceiro níveis) executam apenas operações em paralelo, não existem estados *aguardar* nas máquinas de estados dessa implementação, uma vez que não existe execução seqüencial.

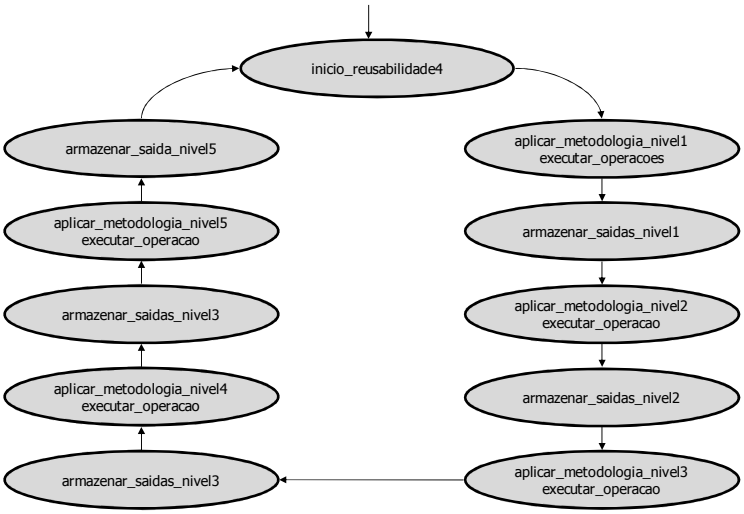

Figura 24. Máquina de Estados da Célula com Metodologia e com Limite de Subgrafos - 4º Nível de Reusabilidade

Célula sem RoSE

A implementação da célula sem a metodologia consiste de quatro blocos operacionais para cada tipo de operação. A Figura 25 apresenta a arquitetura dessa célula.

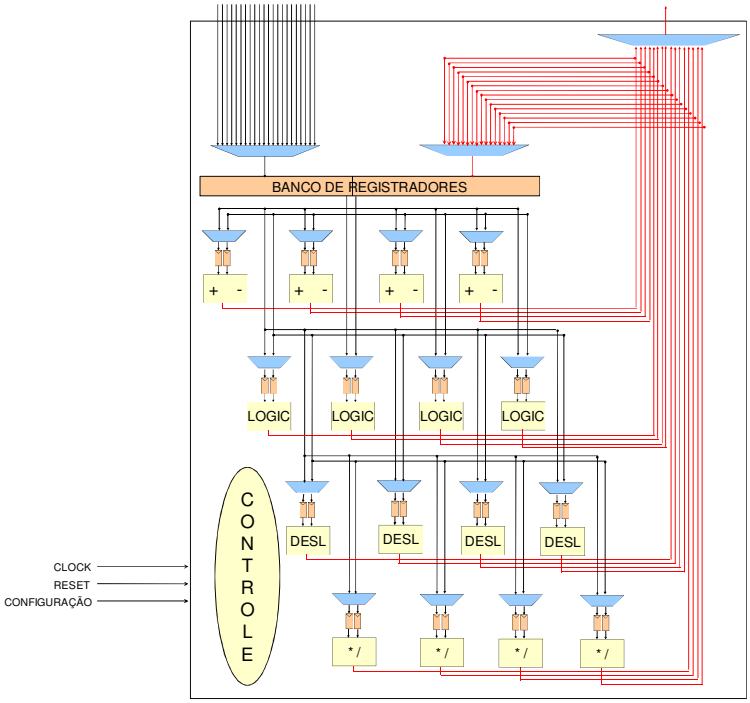

Figura 25. Diagrama de Blocos da Célula sem RoSE

Nessa implementação, a configuração do caminho de dados e a seleção dos blocos operacionais são mais simples do que nas implementações com metodologia, visto que não é necessário executar as operações seqüencialmente. Assim, a unidade de controle possui quatro máquinas de estados (uma para cada tipo de operação) que trabalham em paralelo. Portanto, a execução de todas as operações e o armazenamento dos dados é feito em paralelo.

Embora não existam níveis de reusabilidade, nessa implementação foram mantidas as quatro máquinas de estados para poder habilitar a execução em paralelo de operações de diferente tipos. Com essa estratégia, evitou-se a utilização de testes exaustivos que poderiam causar atrasos na execução da arquitetura. A Figura 26 ilustra uma das máquinas de estados dessa implementação. As outras MEs são idênticas a apresentada na figura, com diferença apenas no tipo de operação cuja execução será habilitada.

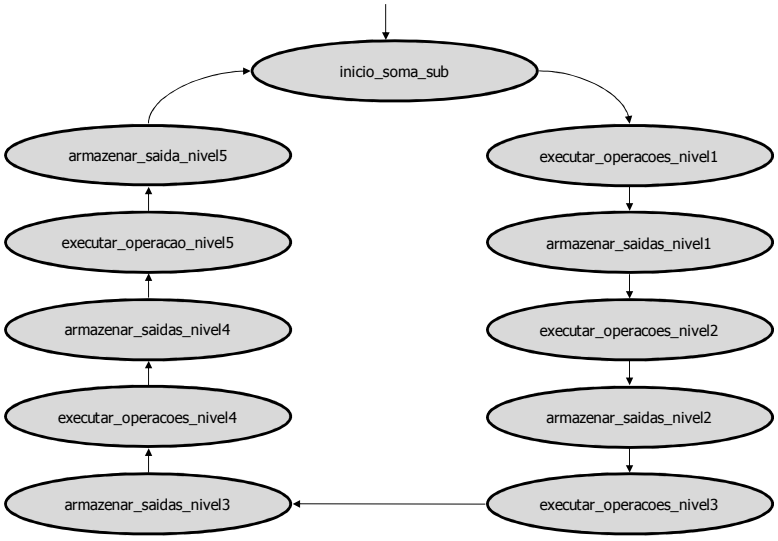

Figura 26. Máquina de Estados da Célula sem Metodologia

4.1.3. Bancos de Registradores

O bloco reconfigurável da arquitetura RoSA possui quatro tipos de banco de registradores: o global, o modificado, o de saída e o local.

O banco de registradores global armazena os dados que são acessados, porém não são modificados pelas células. Os parâmetros das funções e as variáveis globais fazem parte desse conjunto. Ambos são modificados apenas no início da execução pelo processador hospedeiro.

No banco de registradores modificado são armazenadas as informações que são lidas pelas células e modificadas pelo processador em tempo de execução. Esse é o caso dos dados que são atualizados pelo processador. Um exemplo desse tipo de dado é a variável *i* do trecho de código apresentado na seção 3.1 (página 22).

O banco de registradores de saída, como o próprio nome indica, armazena a saída das células

O último tipo de banco de registradores é o local. Nesse banco são armazenados todos os cálculos intermediários das unidades funcionais de uma célula. Cada célula possui um banco de registradores local. A Figura 15 (página 29) ilustra o banco de registradores global e o modificado e a Figura 17 (página 32) ilustra o banco de registradores local.

Inicialmente, o tamanho dos bancos de registradores foi definido a partir de uma análise dos grafos de fluxo de dados das aplicações (PEREIRA, OLIVEIRA e SILVA,

2007). Porém, durante a implementação da arquitetura foi constatada a necessidade de modificar essa quantidade para atender as restrições das células, de acordo com a metodologia.

Assim, o banco de registradores global e o modificado são os bancos de maior capacidade e possuem o mesmo tamanho, pois os bancos devem ter espaço suficiente para armazenar todos os parâmetros das funções e todas as variáveis globais e modificadas. O banco de registradores local armazena os cálculos intermediários para cada nível de subgrafo. Finalmente, como o banco de registradores de saída armazena a saída de cada célula, este deve ter apenas um registrador por célula. A quantidade de registradores e de bancos de registradores é apresentada na Tabela 2.

Tabela 2. Banco de Registradores na Arquitetura RoSA

	#Registradores	#Banco de Registradores
Global	48	1
Modificado	48	1
Local	30	6
Saída	6	1

A segunda coluna da tabela mostra a quantidade de registradores no banco de registradores indicado na primeira coluna. A última coluna determina a quantidade de banco de registradores da arquitetura. De acordo com a tabela, a versão atual da arquitetura RoSA contém 282 registradores.

O processo de escrita nos bancos de registradores global e modificado é feito através da rotina de transferência de dados descrita na seção 3.2 (página 25). A escrita nos bancos local e de saída é feita pelas células, através de suas máquinas de estados, como pode ser visto na seção 4.1.2 (página 31).

4.1.4. Gerenciador de Configuração

Esse componente é responsável por buscar a configuração na cache de configurações (ou na memória) e distribuí-la entre as células, bem como gerenciar os registradores da arquitetura e as saídas das células.

A configuração do bloco reconfigurável corresponde a uma instrução VLIW de 720 bits que inclui a configuração das seis células. Cada célula recebe uma palavra de configuração de 120 bits.

A configuração da célula consiste de: operações das unidades funcionais (todas as operações do subgrafo que devem ser executadas pela célula); os endereços dos dados de

entrada das operações e a estrutura do subgrafo. A estrutura indica como as operações são combinadas para formar o grafo, tanto em largura quanto em profundidade. A Figura 27 ilustra o formato de uma instrução de configuração de todo o bloco reconfigurável e a sintaxe da configuração da célula. A descrição dos campos da configuração é apresentada a seguir.

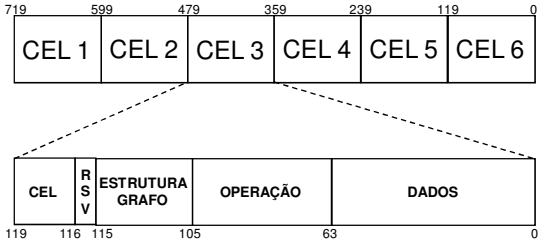

Figura 27. Instrução de Configuração

- DADOS: indica o endereço dos registradores da lógica reconfigurável que armazenam os dados de entrada. Baseado no MISO mais largo encontrado (ver Figura 18, página 33), existem até 8 dados de entrada e são necessários 8 bits de endereçamento para cada dado. Portanto esse campo possui 64 bits.

- OPERAÇÃO: indica quais operações serão executadas na célula. Esse campo determina todas as operações de um MISO. Baseado no MISO mais largo, onde são executadas sete operações, e com 6 bits para indicar cada operação, a largura total desse campo é de 42 bits.

- CEL: possui 3 bits que indicam a qual das seis células a configuração pertence.

- ESTRUTURA DO GRAFO: possui 10 bits que indicam quantas operações são executadas em paralelo (limite máximo de 4 operações paralelas).

Para esse campo também foi considerado o grafo de maior profundidade encontrado (ver Figura 18, página 33), que possui 5 níveis. Dessa forma, cada dois bits indicam quantas operações existem em cada nível. A combinação desse campo com o campo OPERAÇÃO determina o formato do grafo em relação à largura e à altura. A Figura 28 exemplifica como os dois campos são combinados para obter o MISO a ser executado na célula.

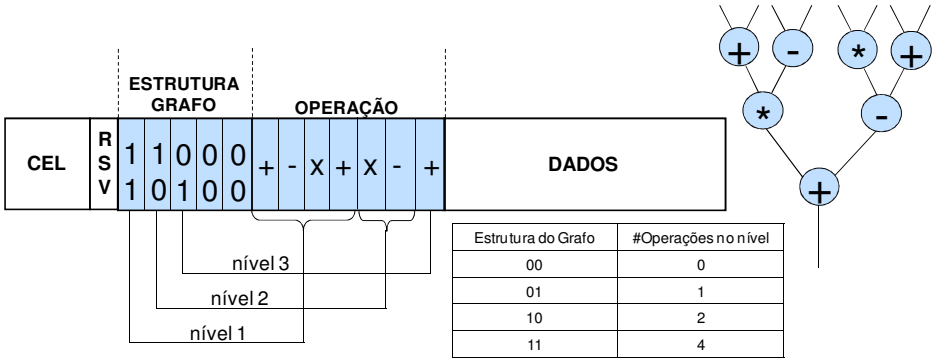

Figura 28. Exemplo de Estrutura de Grafo

Inicialmente, a unidade de controle da célula lê o campo ESTRUTURA DO GRAFO e após conhecer o número de operações existentes no primeiro nível, essas operações são buscadas no campo correspondente, na ordem indicada na figura. Esse processo é feito repetidamente até que não existam mais níveis (indicado com 00). Quando o campo OPERAÇÃO não possui todas as sete operações, isso é indicado com um valor referente a NOP (*no operation*).

A unidade de controle da célula ao receber a configuração, combina essa informação com a arquitetura da célula (isto é feito através da busca de operações e dados nas UFs correspondentes) e associa os registradores de saída das UFs com os registradores de entrada.

Quando uma chamada ao bloco reconfigurável é encontrada durante a execução da aplicação, o processador hospedeiro carrega os dados nos bancos de registradores e chama o gerenciador de configuração para buscar e carregar a configuração em RoSA. A cache de configurações é verificada antes de buscar uma configuração na memória e atualizada quando esta não é encontrada (ocorre *miss*).

A cache de configurações da arquitetura RoSA possui mapeamento associativo e sua política de substituição é a FIFO (*First In First Out*), onde o primeiro dado a entrar é o primeiro a sair.

O tamanho da cache foi definido baseado na análise das aplicações utilizadas como estudo de casos. A partir da análise, constatou-se que uma cache composta por 8 blocos de 23 palavras de largura de 32 bits cada, seria suficiente para apresentar uma boa taxa de acertos (taxa de *hit*) em requisições de configuração pelo gerente de configuração. Cada bloco da cache com 23 palavras corresponde a uma configuração de 720 bits (22,5 palavras são suficientes, porém a interface de leitura da cache possui tamanho fixo de 32 bits).

4.2. Fluxo de Execução

Esta seção descreve como ocorre a execução de aplicações na arquitetura e qual a função de cada componente nesse processamento.

A execução da aplicação em RoSA inicia no processador hospedeiro, após a aplicação ter sido compilada para a arquitetura como descrito no capítulo 3. Ao detectar início de bloco de instruções reconfiguráveis, o processador pára a execução da aplicação e inicia a execução da rotina de transferência de dados incluída no código *assembly* da aplicação pelo compilador. Essa rotina envia todos os dados armazenados na memória ou em registradores do processador para o banco de registradores do bloco reconfigurável. A Figura 13 (página 26) apresenta um exemplo de rotina de transferência de dados. Após transferir todos os dados, o processador envia para o gerenciador de configuração o endereço da configuração que será utilizada pelas células, e fica aguardando o bloco reconfigurável finalizar sua execução.

Enquanto isso, o gerenciador de configuração busca a configuração na cache de configurações ou na memória e a distribui entre as células, sinalizando para as mesmas para iniciarem a execução das operações. As células, então, realizam a execução das operações indicadas pela configuração, utilizando os dados armazenados nos bancos de registradores. Ao final da execução, as células armazenam os resultados no banco de registradores de saída, e sinalizam ao gerenciador de configuração que os cálculos foram finalizados.

O gerenciador aguarda todas as células indicarem que os resultados estão prontos e então interrompe o processador, indicando que o bloco reconfigurável terminou sua execução. O processador, que estava aguardando sinalização do bloco, após ser interrompido, retoma a execução logo após o bloco de instruções reconfiguráveis.

A execução em RoSA descrita acima pode ser dividida nas seguintes tarefas:

T01. Detecção de início de bloco de instruções reconfiguráveis

T02. Execução da rotina de transferência de dados

T03. Envio do endereço da configuração

T04. Busca da configuração na cache ou na memória

T05. Distribuição da configuração entre as células

T06. Busca dos dados no banco de registradores

T07. Análise da configuração

T08. Execução das operações e armazenamento de resultados intermediários

T09. Armazenamento do resultado final nos registradores de saída

T10. Sinalização de término de execução das células

T11. Sinalização de término de execução do bloco reconfigurável

T12. Retomada de execução após o bloco de instruções reconfiguráveis

A Figura 29 apresenta a seqüência de execução das tarefas descritas anteriormente separada pelos componentes da arquitetura que as executam.

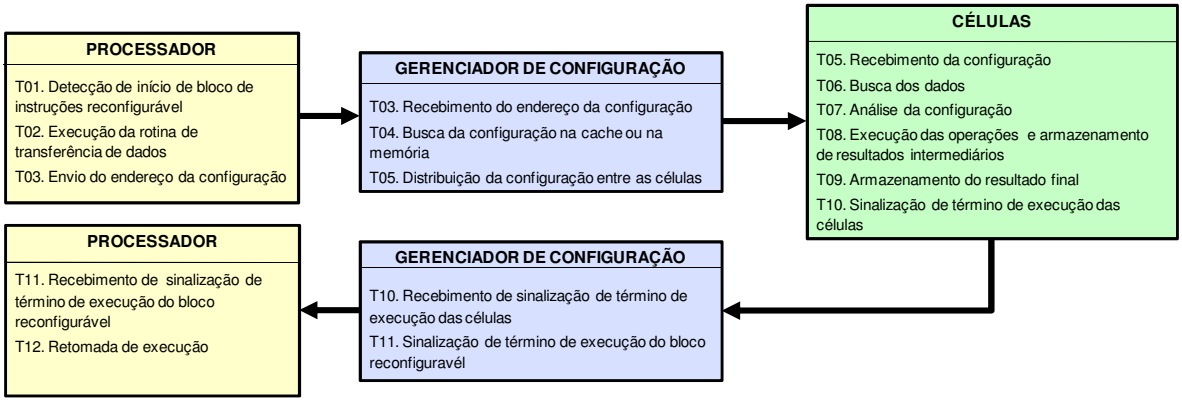

Figura 29. Execução nos Componentes da Arquitetura RoSA

As tarefas com numeração repetida na figura representam que um componente executa aquela tarefa, enquanto o outro recebe o resultado da execução. Esse é o caso da tarefa 3, que corresponde ao envio do endereço da memória onde a configuração que será executada está armazenada. De acordo com a figura, o componente responsável por enviar esse endereço é o processador. Porém, a tarefa 3 também está presente no gerenciador de configuração, indicando que ele recebe o endereço enviado pelo processador. O mesmo acontece com a tarefa 5, que é executada pelo gerenciador de configuração e as células recebem a configuração que foi enviada nessa tarefa. A tarefa 10 é mais uma desse tipo. De acordo com a figura, as células enviam um sinal para o gerenciador de configuração indicando que terminaram a execução e este por sua vez recebe esse sinal. E por fim, a tarefa 11 corresponde ao envio de um sinal pelo gerenciador indicando término de execução do bloco reconfigurável, e a mesma tarefa no processador indica o recebimento desse sinal por esse componente.

Capítulo 5

Resultados

A arquitetura RoSA descrita no capítulo anterior foi implementada em linguagem de descrição de hardware VHDL e validada através de simulações e da prototipagem em FPGA, utilizando o kit de prototipagem *Altera Development and Education Board*[4] (DE2).

A seguir será apresentada uma análise comparativa dos resultados de área ocupada, freqüência e desempenho para três implementações diferentes da célula. Esse capítulo também apresenta a área e a freqüência da arquitetura completa, tal como apresentada na Figura 15 (página 29). Além disso, também serão apresentados os resultados de desempenho da arquitetura comparados aos resultados de desempenho do processador Nios II.

Para a descrição em VHDL e as simulações dos componentes foi utilizada a ferramenta Quartus II[5], versão 7.2, da Altera. Também foi utilizada a ferramenta SOPC Builder[6], versão 7.2, para criação do sistema composto por arquitetura RoSA, memória e processador Nios II. Por fim, para a implementação das aplicações a serem executadas no protótipo do sistema foi utilizada a ferramenta Nios II IDE[7], versão 7.2 (*Integrated Development Environment*).

A arquitetura RoSA foi sintetizada para dois dispositivos programáveis. O primeiro, EP2C35F672C6, pertence à família Cyclone II e corresponde ao dispositivo presente no kit de prototipagem *Altera Development and Education Board* (DE2), utilizado para a prototipagem de teste e validação da arquitetura. Este kit foi desenvolvido pela Altera para uso em universidades, com o objetivo de auxiliar na aprendizagem de lógica digital, organização de computadores e FPGAs. Por esse motivo, o kit apresenta uma interface simples e didática e um dispositivo com pequena capacidade e de desempenho intermediário. Assim, para avaliar o desempenho da arquitetura em um dispositivo de grande porte, o projeto também foi sintetizado para o dispositivo EP2S60F1020C3 da família Stratix II. Porém, neste

[4] http://www.altera.com/education/univ/materials/boards/unv-de2-board.html
[5] http://www.altera.com/products/software/products/quartus2
[6] http://www.altera.com/products/software/products/sopc
[7] http://www.altera.com/products/ip/processors/nios2/tools/ide/ni2-ide.html

caso não foi realizada prototipagem, apenas resultados de área ocupada e freqüência de operação foram obtidos com o auxílio da ferramenta Quartus II.

Além do sistema completo, resultados de área e freqüência de partes da arquitetura também foram avaliados. As partes analisadas correspondem à implementação de uma célula e a implementação da arquitetura somente com as seis células, sem qualquer outro componente. A seção seguinte apresenta os resultados de área e freqüência da arquitetura e na seção 5.2 são apresentados os resultados de desempenho.

Também foram realizados experimentos com os três processadores da família Nios II. Esses resultados incluem área e freqüência da arquitetura RoSA anexada a cada um dos processadores. A análise de desempenho de RoSA inclui comparação entre a implementação de RoSA anexada a cada um dos processadores e o desempenho alcançado por cada processador separadamente.

5.1. Área e Freqüência

A implementação da arquitetura possui aproximadamente 13000 linhas de código, e foi realizada em três etapas principais:

- Implementação da célula: consistiu na realização de três implementações distintas para a célula. Na primeira implementação a metodologia RoSE foi usada e não existe limite de subgrafos que podem ser executados. A segunda implementação também utiliza a metodologia RoSE, porém com limitação dos subgrafos que podem ser executados na célula. Por fim, na terceira implementação a metodologia RoSE não foi utilizada. Detalhes sobre essas implementações podem ser encontrados na seção 4.1.2 (página 31).
- Implementação do gerenciador de configuração e dos bancos de registradores: nessa etapa além da arquitetura dos componentes, também foram desenvolvidas as rotinas de envio de dados e de endereço de configuração pelo processador, ao banco de registradores e gerenciador de configuração, respectivamente. Também foi implementada a hierarquia de memória para o armazenamento da configuração.
- Integração dos componentes: nessa etapa todos os componentes desenvolvidos nas etapas anteriores foram integrados, formando o bloco reconfigurável. Durante a prototipagem, este bloco foi integrado à memória e ao processador, para formar o sistema. O protótipo obtido foi usado para teste e validado em FPGA.

Os resultados de área e freqüência apresentados a seguir foram divididos em três conjuntos de implementações. O primeiro conjunto corresponde as três implementação

diferentes para uma única célula. O segundo conjunto é formado pelas três implementações da arquitetura com seis células, porém sem nenhum outro componente integrado. Por fim, o terceiro conjunto contém as implementações do projeto completo, também com os três diferentes tipos de células.

A Tabela 3 apresenta os resultados da célula, da instanciação das seis células e do projeto completo com seus três diferentes tipos de célula. O primeiro tipo corresponde às implementações com metodologia RoSE e sem limitação de subgrafos, (representado pela sigla RsL – RoSE sem Limitação); o segundo tipo indica as implementações com metodologia RoSE e com limitação de subgrafos (RcL – RoSE com Limitação) e o último tipo corresponde às implementações sem RoSE (sR). Os resultados apresentados na tabela não incluem o bloco operacional de divisão. O motivo para a não inclusão dessa operação será descrito posteriormente.

As implementações foram sintetizadas para o dispositivo EP2S60F1020C3 da família Stratix II, que possui 48.352 ALUTs (*adaptive look-up tables*) e as freqüências foram medidas em MHz (*MegaHertz*).

Tabela 3. Resultados de Área e Freqüência - Stratix II

	Uma Célula			Seis Células			Projeto Completo		
	RsL	RcL	sR	RsL	RcL	sR	RsL	RcL	sR
Freq. (MHz)	130,91	129,77	148,79	128,98	144,49	125,42	85,16	97,03	92,71
Área (ALUTs)	2.466	2.400	2.260	18.705	18.137	16.032	25.462	24.941	23.458

Os resultados apresentados na tabela acima indicam que para a implementação de apenas uma célula, a metodologia RoSE não traz vantagens ao projeto. Embora o aumento de área introduzido pela metodologia seja de apenas 206 ALUTs para a implementação sem limitação de subgrafo, e 140 ALUTs para a implementação com limitação de subgrafo, a freqüência da célula sem metodologia supera as duas outras implementações.

Apesar dos resultados desfavoráveis à metodologia exibidos na implementação de uma única célula, no projeto das seis células é possível observar que os resultados se invertem. Isto significa que, a uma escala maior, as implementações com a metodologia RoSE superam a implementação sem a metodologia. A melhor freqüência alcançada corresponde à implementação com a metodologia e com limitação nos tipos de subgrafos a serem executados, seguida da implementação sem limitações. Embora a implementação sem metodologia tenha apresentado a menor área ocupada, as diferenças são de apenas 2.105 ALUTs entre a implementação sR e a RcL e 2.673 ALUTs entre sR e RsL.

A implementação de seis células com metodologia e sem limites de subgrafos também se mostrou mais vantajosa do que a implementação sem metodologia. A área da primeira – RsL – apresenta, aproximadamente, 1.200 ALUTs a mais do que a área da segunda – sR, entretanto, com relação à freqüência, a implementação sem metodologia apresenta resultado inferior.

Os resultados de freqüência e área do projeto completo exibidos na Tabela 3, indicam que com relação à freqüência, a melhor implementação é a que aplica a metodologia RoSE com limites de tipos de subgrafos executados. Porém, o melhor resultado de área foi alcançado com a implementação sem metodologia, com uma diferença de apenas 973 ALUTs. Uma vez que a diferença de área entre as duas implementações de maior freqüência é pouco significante, a melhor opção é a implementação com metodologia e com limites de subgrafos, devido a sua maior freqüência. Os resultados de área indicam que o sistema completo, com bloco reconfigurável, processador e memória, ocupam aproximadamente 41% da área (em ALUTs) de todo o dispositivo.

A Figura 30 apresenta a ocupação, em porcentagem, dos componentes da arquitetura. Os resultados correspondem à implementação do projeto com metodologia e com limitação de subgrafos, uma vez que esta é a implementação de maior freqüência.

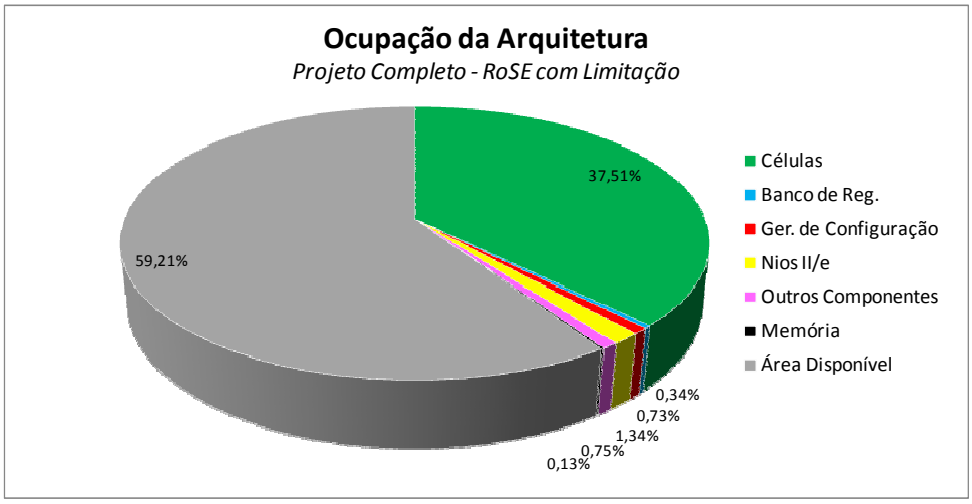

Figura 30. Área ocupada pelo Projeto Completo - Implementação RcL – Stratix II EP2S60F1020C3

As células são responsáveis pela maior parte da área ocupada pela arquitetura. Os outros componentes do bloco reconfigurável, o Nios II e a memória ocupam menos de 3% da área do dispositivo. Os outros componentes não detalhados no gráfico correspondem a

componentes instanciados na criação do sistema como o módulo de acesso à memória e o módulo JTAG[8], que permite realizar testes com a placa de prototipagem.

Como mencionado no início do capítulo, a arquitetura também foi sintetizada no kit de prototipagem Altera DE2. Com o kit foi possível anexar o bloco reconfigurável ao processador Nios II e executar aplicações utilizando a ferramenta Nios II 7.2 IDE.

O FPGA disponível no kit DE2 corresponde a um Cyclone II EP2C35F672C6. A unidade de área utilizada por essa família é o elemento lógico (do inglês *Logic Element* - LE), e o dispositivo possui 33.216 LEs. Os resultados de freqüência e área, sem o bloco operacional de divisão são apresentados na Tabela 4.

Tabela 4. Resultados de Área e Freqüência - Cyclone II

	Uma Célula			Seis Células			Projeto Completo		
	RsL	RcL	Sr	RsL	RcL	sR	RsL	RcL	sR
Freq. (MHz)	61,19	63,98	64,88	62,79	64,09	62,71	62,88	64,06	63,08
Área (LEs)	4.358	4.254	3.937	18.420	17.952	15.926	22.951	22.488	20.468

Assim como nos resultados de síntese para o dispositivo da família Stratix II, nos resultados apresentados na Tabela 4, a implementação sem metodologia é mais vantajosa quando existe apenas uma célula. No projeto com seis células, os resultados para o dispositivo da família Cyclone II também seguiram o mesmo padrão dos resultados do dispositivo da família Stratix II. Com relação à freqüência, a melhor implementação foi a RcL e a menor área ocupada corresponde à implementação sR.

No projeto completo, os resultados de freqüência também foram favoráveis à implementação com metodologia e com limites de subgrafo e a menor área foi alcançada pela implementação sem metodologia. Porém, a diferença nos resultados de freqüência das implementações é bem menor em valores absolutos na Cyclone II do que na Stratix II. A diferença entre os três resultados de freqüência do projeto completo apresentados na Tabela 4 é menor do que 1 MHz.

O impacto na área da arquitetura se manteve semelhante nos dois dispositivos, considerando a diferença entre os resultados das três implementações. Por exemplo, a diferença entre a área ocupada pelo projeto completo na implementação RsL e a área ocupada na implementação RcL é de 521 ALUTs no dispositivo da família Stratix II, e de 463 LEs no dispositivo da família Cyclone II.

[8] O *Joint Test Action Group* (JTAG) é o nome utilizado para o padrão IEEE 1149.1, desenvolvido para testes em circuitos após fabricação. O Nios II utiliza esse módulo para controlar o processador do sistema através do computador remoto, permitindo os testes no sistema. Detalhes podem ser encontrados em: http://www.jtag.com

A área ocupada pelo projeto completo no dispositivo da família Cyclone II representa 68% do dispositivo. Embora a freqüência da arquitetura seja superior a 60 MHz, a placa DE2 trabalha a uma freqüência de 50 MHz, limitando a freqüência do sistema.

A Figura 31 apresenta a área ocupada (em porcentagem) pelos componentes do projeto completo. Os resultados apresentados correspondem à implementação com metodologia RoSE e com limitação de subgrafos.

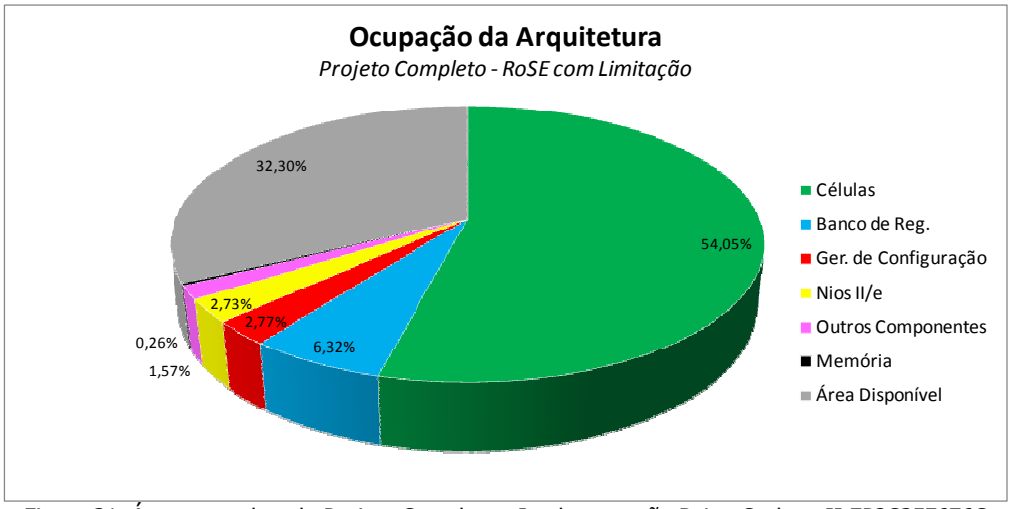

Figura 31. Área ocupada pelo Projeto Completo - Implementação RcL – Cyclone II EP2C35F676C

As células são responsáveis por mais da metade dos elementos lógicos do dispositivo. O banco de registradores é o segundo componente com maior área da arquitetura, porém sua ocupação não ultrapassa 7% de toda a área do dispositivo. Os outros componentes somados ocupam menos de 8% da área do dispositivo.

No Anexo II são apresentados os resultados de área ocupada de todas as implementações, com detalhes de ocupação por componente. Os resultados são apresentados para os dois dispositivos utilizados nesse trabalho. Além dos resultados de área, o anexo também apresenta a freqüência alcançada pelas implementações.

Para simplificar o desenvolvimento do projeto, a implementação de alguns blocos operacionais das células (deslocadores e multiplicador) utilizou componentes da biblioteca da Altera, gerados pela ferramenta *Mega-Wizard Plug-In Manager*, disponível no Quartus II.

Apesar do alto desempenho alcançado pelos componentes utilizados, o componente de divisão não seguiu o mesmo padrão. Como pode ser observado na Tabela 5, a freqüência alcançada pela célula com componente de divisão é muito inferior a freqüência apresentada nas tabelas 3 e 4, que não possuem esse bloco operacional.

Na tentativa de melhorar essa freqüência, outras versões do componente de divisão da Altera foram instanciadas, fazendo uso de *pipeline* de execução, e aumentando a quantidade de ciclos para realizar a divisão. Como indicado na Tabela 5, mesmo aumentando a quantidade de ciclos de divisão para 4 ciclos, a freqüência ainda ficou bem abaixo da freqüência obtida sem o componente de divisão. Portanto, concluiu-se que a baixa freqüência alcançada é resultado da estrutura do componente divisão da Altera. Assim, optou-se por retirar esse bloco operacional do projeto e implementar outro bloco de divisão. Porém, por limites de tempo de desenvolvimento do projeto, até o presente momento este bloco ainda não foi desenvolvido, e os resultados do projeto apresentados nesse trabalho não incluem os blocos operacionais de divisão.

Tabela 5. Resultados de Freqüência da Célula com Divisão

Freqüência (MHz)		
	Cyclone II EP2C35F672C6	**Stratix II EP2S60F1020C3**
Divisão em 1 ciclo	13,07	19,95
Divisão em 2 ciclos	15,68	22,98
Divisão em 3 ciclos	19,93	29,30
Divisão em 4 ciclos	20,92	37,35

5.2. Desempenho

Esta seção apresenta uma análise dos resultados de desempenho da arquitetura. Inicialmente, foi realizada uma análise do custo, em ciclos de relógio, de todo processo de execução das aplicações na arquitetura, que vai desde a detecção de início de bloco de instruções reconfiguráveis pelo processador, até o momento em que o processador retoma a execução após o bloco reconfigurável ter finalizado, como descrito na seção 4.2 (página 46).

Em seguida, foi feita uma análise comparativa entre o desempenho da arquitetura RoSA e o desempenho do processador Nios II, ao executar algumas aplicações.

Uma vez que o compilador para a arquitetura RoSA está fora do escopo desse trabalho, não foi possível gerar configurações e rotinas de transferência de dados de acordo com o descrito no capítulo 3 (página 22). Assim, para avaliar o desempenho da arquitetura, trechos de aplicações foram selecionados e a partir dos grafos de fluxo de dados desses trechos, configurações e rotinas de transferências de dados foram criadas manualmente e utilizadas para a execução sobre o protótipo obtido com a placa DE2. Nestas execuções, configurações são enviadas do processador para a arquitetura e os resultados finais calculados pelas células foram enviados para a memória.

A Tabela 6 apresenta o custo, em ciclos de relógio, de cada uma das tarefas descritas na seção 4.2. A coluna Componente indica o componente responsável por executar a tarefa. Os detalhes sobre como esses custos foram calculados são apresentados a seguir.

Os custos são referentes à execução no protótipo da arquitetura RoSA (obtido com o uso da placa DE2) com metodologia RoSE e com limite de subgrafos, utilizando todas as versões do processador Nios II/e.

Tabela 6. Custo de processamento da arquitetura RoSA com metodologia RcL

Custo em Ciclos				
Tarefas		**Nios II/e**	**Nios II/s**	**Nios II/f**
T02	Execução da rotina de transferência de dados	208	144	96
T03	Envio do endereço da configuração	26	18	12
T04	Busca da configuração na cache ou na memória	97	26	26
T05	Distribuição da configuração entre as células	1	1	1
T06	Busca dos dados no banco de registradores	1	1	1
T07	Análise da configuração	1	1	1
T08	Execução das operações e armazenamento de resultados intermediários	11	11	11
T09	Armazenamento do resultado final nos registradores de saída	1	1	1
T10	Sinalização de término de execução das células	1	1	1
T11	Sinalização de término de execução do bloco reconfigurável	92	58	81
Total		**439**	**262**	**231**

As tarefas T01 - **Detecção de início de bloco de instruções reconfiguráveis** e T12 - **Retomada de execução a partir do final do bloco de instruções reconfiguráveis** descritas na seção 4.2, representam apenas o início e o fim da execução da aplicação na arquitetura RoSA, portanto não estão incluídas no cálculo de custos de processamento da arquitetura.

O cálculo de custo das tarefas indicadas na tabela foi realizado através da inclusão de contadores de ciclos nos componentes da arquitetura. Em todas as tarefas, foi considerado o pior caso de execução.

Para o cálculo de custo da tarefa T02, um contador de ciclos foi incluído no banco de registradores. Este contador é atualizado todas as vezes que um dado chega ao banco. O pior caso para a transferência de dados corresponde ao subgrafo com o maior número de dados de entrada (8 entradas), devido ao custo de comunicação entre as unidades funcionais, uma vez que quanto maior a quantidade operações maior a comunicação entre as células para a execução dessas operações. Portanto, o custo apresentado na Tabela 6 corresponde à transferência de 8 dados para a arquitetura. O processo de envio de um endereço de configuração, do processador para o gerenciador de configuração (T03) é semelhante ao envio de um dado para o banco de registradores, portanto, o custo de execução

dessa tarefa é o mesmo da tarefa T02. A única diferença está na quantidade de informação transmitida, já que o custo de envio do endereço de configuração é o mesmo da tarefa T02 ao transferir apenas um dado.

O maior custo na execução da tarefa T04 ocorre quando a configuração não é encontrada na cache (ocorre *miss*) e é necessário buscá-la na memória. Esse custo é calculado através do contador de ciclos incluído na máquina de estados do gerenciador de configuração. O custo para executar essa tarefa também inclui a busca pela configuração na cache, através do mapeamento associativo, o qual requer comparação de *tags* de endereçamento.

Com relação ao custo da tarefa T08, foram considerados os três tipos de células existentes. O pior caso para os três tipos é o subgrafo com a maior quantidade de operações encontrado, que contém 7 operações. Porém, para a implementação com metodologia e sem limites de subgrafos, o pior caso ocorre quando todas as operações pertencem ao primeiro nível de reusabilidade, pois como só existe um bloco operacional para esse tipo, é necessário executá-las de forma seqüencial. Assim, o mesmo subgrafo foi considerado para as outras implementações e o custo para executá-las foi calculado a partir da máquina de estados de cada implementação.

O custo da tarefa T11 corresponde ao custo de enviar um sinal de interrupção do gerenciador de configuração para o processador, este tratar a interrupção e retornar um sinal ao gerenciador indicando que a interrupção foi tratada. Todo esse processo de envio de sinais é feito através do barramento Avalon.

As outras tarefas (T05, T06, T07, T09 e T10) são executadas em apenas um ciclo. De acordo com a Tabela 6, para a execução de uma configuração na arquitetura, que possua oito dados de entrada, sete operações e todas pertencentes ao primeiro nível de reusabilidade, são necessários 439 ciclos para o Nios II/e, 262 para o Nios II/s e 231 para o Nios II/f. Esse custo tem como base a implementação das células com limitação nos tipos de subgrafos.

A utilização das outras implementações da célula altera apenas o custo da tarefa T08, e somente na implementação com metodologia e sem limite de subgrafos. Para essa implementação são necessários 17 ciclos para a execução dessa tarefa considerando o pior caso (apenas operações do primeiro nível de reusabilidade), aumentando o custo de execução total de uma configuração em apenas 6 ciclos. A tarefa T08 na implementação sem metodologia tem o mesmo custo da execução na implementação com metodologia e com limite de subgrafos.

Para avaliar o desempenho da arquitetura, configurações foram geradas manualmente a partir de aplicações, e o desempenho na execução dessas configurações foi comparado com o desempenho do processador ao executar os trechos de código equivalentes às configurações.

As aplicações foram implementadas em linguagem C e para obter o desempenho na execução dessas aplicações no processador foi utilizado o componente *Performance Counter* da Altera que é anexado ao processador e realiza o cálculo de desempenho na execução de trechos da aplicação.

O *Performance Counter* pertence a biblioteca de componentes da Altera e pode ser incluído no sistema através do SOPCBuilder. Esse componente utiliza a interface Avalon para realizar análises de desempenho de aplicações, em tempo real.

A principal vantagem da utilização desse componente é a precisão nos resultados de desempenho obtidos. Uma vez que o *Performance Counter* requer apenas instruções que indicam o início e o fim do trecho da aplicação que se deseja obter o desempenho, não é necessária a execução de rotinas de cálculo de desempenho pelo processador ou acesso à memória. Em (ALTERA CORPORATION, 2008c) são encontradas mais informações sobre este componente.

É importante salientar que embora o *Performance Counter* possa ser incluído em qualquer sistema que utilize o Nios II, neste trabalho ele foi utilizado apenas para obter resultados de desempenho do processador. Assim, o sistema em que este componente foi instanciado contém apenas Nios II, *Performance Counter* e o módulo JTAG, para interação do sistema com o Nios II IDE. Os resultados de desempenho da arquitetura RoSA foram obtidos através da utilização dos contadores de ciclos incluídos nos componentes, como indicado anteriormente. Essa estratégia foi utilizada por prover resultados de desempenho ainda mais precisos, e uma vez que a implementação em VHDL do Nios II não é disponibilizada pela Altera, não foi possível utilizar a mesma estratégia de contadores no processador.

Devido a ausência de compilador para a geração das configurações das rotinas de transferência de dados, não é possível executar uma aplicação completa em RoSA. Dessa forma, para demonstrar o ganho de desempenho alcançado pela arquitetura, trechos de aplicações foram selecionados e as configurações equivalentes a essas instruções, implementadas manualmente, foram executadas em RoSA. Os resultados são apresentados na Tabela 7.

A primeira aplicação utilizada foi a DCT. Para estimar o custo da DCT, foram geradas configurações para cada um dos passos de execução que a DCT possui e cada passo foi executado individualmente em RoSA. Em seguida, o desempenho na execução de cada passo foi acumulado, e somou-se a esse resultado o custo de executar as outras instruções no processador (no caso da DCT estas instruções correspondem apenas ao laço iterativo FOR). O desempenho da execução da DCT no processador foi medido através do *Performance Counter*.

A mesma estratégia de estimação de desempenho foi feita para a aplicação FFT. A Tabela 7 apresenta o desempenho, em ciclos, na execução dessas aplicações e a Tabela 8 apresenta o ganho de desempenho (em porcentagem) e a aceleração alcançados por RoSA. Os resultados são exibidos de acordo com o processador utilizado (Nios II econômico, Nios II padrão e Nios II rápido) e na arquitetura RoSA anexada à esses três processadores.

Os grafos de fluxo das aplicações satisfazem os limites da metodologia RoSE, portanto, para qualquer uma das implementações das células o custo de execução é o mesmo.

A diferença entre os resultados de desempenho da arquitetura RoSA anexada a diferentes processadores se deve à utilização da cache de configurações e à execução das tarefas que dependem do processador. A cache de configurações só é utilizada para as implementações de RoSA com o Nios II/s e o Nios II/f. Uma vez que o Nios II/e não possui cache de instruções, a arquitetura RoSA anexada ao Nios II/e também não utilizou cache de configuração, isto permitiu uma análise com a utilização de processador e arquitetura compatíveis.

O desempenho na execução das tarefas que dependem do processador é influenciado pela versão do processador utilizado. Este é o caso da tarefa T11, uma vez que a sinalização de término de execução do bloco reconfigurável corresponde a interromper o processador, o custo do tratamento da rotina de interrupção pelo processador depende da versão que está sendo utilizada (Nios II/e, Nios II/s ou Nios II/f). Portanto, para permitir uma análise de desempenho entre arquiteturas compatíveis, a comparação deve ser realizada entre o processador e arquitetura RoSA que possua o mesmo processador.

Tabela 7. Resultados de Desempenho (em #ciclos) de Instruções da DCT

Desempenho - #ciclos						
	Econômico		Padrão		Rápido	
	Nios II	RoSA	Nios II	RoSA	Nios II	RoSA
DCT	106.662	32.168	22.104	18.056	16.309	17.284
FFT	355.116	30.036	63.856	13.295	32.818	9.142

Tabela 8. Ganho de Desempenho e Aceleração

	Ganho de Desempenho (%)			Aceleração		
	RoSA com Nios II/e	RoSA com Nios II/s	RoSA com Nios II/f	RoSA com Nios II/e	RoSA com Nios II/s	RoSA com Nios II/f
DCT	69,84	18,31	<0	3,32	1,22	0,94
FFT	91,54	79,18	72,14	11,82	4,80	3,59

Como pode ser observado na Tabela 7, o Nios II econômico obteve o pior desempenho dentre os apresentados. Esse resultado está relacionado com o fato de o Nios II/e executar qualquer instrução em, pelo menos, 6 ciclos não possuir caches de instrução e de dados e não possuir suporte a *pipeline*.

Na Tabela 8 são apresentados o ganho de desempenho alcançado por RoSA e a aceleração na execução das aplicações por RoSA em comparação com a execução no Nios II. O desempenho de RoSA utilizando o Nios II/e é, aproximadamente, 70% superior ao desempenho do Nios II/e, isto implica em uma aceleração de mais de 3 vezes.

Ao se comparar o desempenho do Nios II/s e da implementação de RoSA com este mesmo processador deve ser levado em conta que o Nios II padrão possui cache de instruções. O equivalente a cache de instruções do processador corresponde a cache de configurações na arquitetura RoSA. As configurações executadas repetidamente em RoSA são mantidas na cache, permitindo a redução no custo da tarefa T04, como indicado na Tabela 6.

O ganho de desempenho na execução de parte da DCT em RoSA com Nios II/s é de 18,31%, comparada à execução no Nios II/s, que corresponde a uma aceleração de 1,2 vezes.

O desempenho na execução da DCT no processador Nios II/f é superior a execução das configurações em RoSA com este mesmo processador. Esse resultado se deve ao fato de o Nios II/f possuir cache de dados, enquanto que todos os dados de entrada utilizados por RoSA são enviados através da rotina de transferência de dados.

Uma estratégia para acelerar o envio de dados para RoSA é acrescentar uma cache de dados e manter a coerência entre as caches do processador e de RoSA através do

protocolo *snoopy*[9]. Assim, qualquer dado requisitado por RoSA ao ser atualizado pelo processador, também seria atualizado na cache de RoSA, através do barramento. O problema dessa estratégia está no fato de o Nios II não possuir coerência de cache. Então seria necessário implementar esse mecanismo para incluir a cache de RoSA e o protocolo *snoopy*.

Os resultados apresentados na Tabela 8 também indicam que o ganho de desempenho e a aceleração alcançada por RoSA na execução da FFT se mostraram superiores ao alcançado na execução da DCT. Isto ocorreu devido à quantidade de iterações que a FFT possui. Enquanto a DCT possui 28 configurações diferentes e cada uma é executada 16 vezes, a FFT possui 11 configurações e algumas são executadas 5120 vezes. Assim, embora a DCT apresente uma maior quantidade de instruções que podem ser executadas em paralelo, as repetições na execução das configurações da FFT permitem um maior ganho de desempenho devido ao rápido acesso à cache de configurações, superando, inclusive o Nios II rápido que possui cache de dados.

Vale salientar que para a DCT analisada nesta seção foi utilizado um bloco de tamanho 8x8 como entrada. Uma DCT com tamanho de bloco maior implica em mais iterações, permitindo um maior ganho de desempenho de RoSA.

[9] Protocolo *Snoopy*: Protocolo de coerência de cache que utiliza controladores de caches locais que, através de *broadcast,* comunicam às outras caches do sistema as ações executadas sobre seus dados compartilhados.

Capítulo 6

Conclusão

Esse trabalho apresentou uma arquitetura reconfigurável híbrida, denominada RoSA. No desenvolvimento da arquitetura foram realizadas as seguintes etapas: proposta, implementação, simulação, prototipação, validação e análise dos resultados de desempenho.

RoSA (*Reconfigurable Stream-based Architecture*) consiste de um bloco reconfigurável anexado a um processador hospedeiro. Seu objetivo é acelerar aplicações com grandes fluxos de dados através da execução paralela no bloco reconfigurável de determinados trechos da aplicação.

Para acelerar os trechos de código algumas técnicas de extração de paralelismo no nível de instrução foram aplicadas na fase de compilação. A exploração de paralelismo baseia-se na análise do fluxo de dados das aplicações. Essa análise permitiu a seleção de trechos da aplicação para execução paralela.

A exploração de paralelismo pela arquitetura RoSA gerou a necessidade de propor uma fase de otimização específica para a arquitetura que deve ser introduzida em um compilador regular GCC. Essa fase, denominada fase de otimização RoSA, é composta por etapas que realizam a análise dos grafos de fluxo de dados da aplicação, verificação de restrições para seleção dos candidatos e particionamento hardware/software.

Visando a economia de área em chip, esse trabalho também apresentou e utilizou uma nova metodologia de reuso de hardware em caminhos de dados, baseada em níveis de reusabilidade. A metodologia RoSE (*Reuse-based Standard Datapath Architecture*) visualiza uma unidade reconfigurável da arquitetura RoSA (denominada célula) como um único caminho de dados. Essa estratégia mostrou-se vantajosa para a implementação do conjunto de células da arquitetura.

Para avaliar a área e o desempenho da arquitetura, a mesma arquitetura foi implementada com três tipos de células diferentes. O primeiro tipo aplica a metodologia RoSE e não possui qualquer tipo de limitação com relação aos subgrafos que podem ser executados. Isto significa que determinadas operações de um subgrafo podem ser executadas seqüencialmente caso não existam blocos operacionais suficientes para a execução paralela. O segundo tipo de célula executa apenas os grafos que atendem as restrições da metodologia.

Assim, não existe execução seqüencial neste tipo, porém a execução de operações em paralelo é limitada. Por fim, o terceiro tipo de célula não aplica a metodologia RoSE, portanto, existe a mesma quantidade de blocos operacionais para todas as operações executadas na célula. Nesse tipo de célula qualquer subgrafo é permitido e todos os tipos de operações podem ser executados em paralelo.

Além dos três tipos de célula, a análise de área e desempenho também incluiu a utilização dos três tipos de processadores disponíveis. O Nios II corresponde a uma família de processadores com três versões diferentes. O Nios II econômico que é o mais simples e com menor desempenho; o Nios II padrão, que possui cache de instruções e pipeline e o Nios II rápido, que possui o melhor desempenho dentre os processadores da família, devido ao pipeline de seis estágios, cache de instruções e de dados e opções para melhorar o desempenho de instruções de multiplicação, divisão e deslocamento, dentre outras características. A conseqüência desse alto desempenho alcançado pelo Nios II rápido é o aumento da área (o Nios II econômico é aproximadamente 50% menor do que o Nios II padrão e este utiliza aproximadamente 20% menos lógica do que o Nios II rápido).

Para validar as técnicas escolhidas e avaliar o desempenho da arquitetura RoSA alguns estudos de caso foram realizados. Nos estudos foram comparadas a execução seqüencial e a paralela de um conjunto de aplicações. Os resultados alcançaram até 91% de ganho de desempenho com a execução de aplicações em RoSA e aceleração de 11 vezes, comparando-se a execução no processador Nios II da Altera.

Dentre as contribuições futuras para esta arquitetura estão algumas tarefas em curto prazo como incluir o módulo de divisão nas unidades reconfiguráveis e acrescentar operações de ponto flutuantes, ainda não suportadas pela arquitetura. Dentre as contribuições em longo prazo é necessário desenvolver o compilador com a fase de otimização para RoSA, que irá viabilizar a execução de toda a aplicação na arquitetura RoSA.

Outra contribuição corresponde a acrescentar cache de dados no bloco reconfigurável de RoSA para melhorar o desempenho na busca dos dados. Os resultados de desempenho analisados indicam que para algumas aplicações o processador Nios II na versão otimizada possui melhor desempenho do que RoSA. Isto se deve a cache de dados introduzida nesta versão do Nios II. Uma vez que todos os dados utilizados pelo bloco reconfigurável de RoSA devem ser enviados através de rotinas de transferência de dados, é possível perceber que este atraso prejudica o desempenho de RoSA. Assim, incluir uma cache de dados em

RoSA e implementar um protocolo de coerência de caches permite que RoSA alcance um alto desempenho, compatível com o desempenho alcançado pelo Nios II rápido.

Os trabalhos futuros também incluem a análise da potência consumida pela arquitetura, considerando tanto a metodologia RoSE quanto cada um dos processadores da família Nios II.

Referências

ALIPPI, Cesare, et al. **A DAG-Based Design Approach for Reconfigurable VLIW Processors**. In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'99), Munique, Alemanha, IEEE Computer Society, pp. 778-780, 09-12 de Março de 1999.

ALSOLAIM, Ahmad, BECKER, Jurgen, GLESNER, Manfred e STARZYK, Janusz. **Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communications.** In IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society, pp. 205-214, 17-19 de Abril de 2000.

ALTERA CORPORATION. **Nios II Custom Instruction:** User Guide. Disponível em: <http://www.altera.com/literature/lit-nio2.jsp>. Acesso em: Jan. 2008, 2008a.

_____. **Nios II Processor Reference Handbook**. Disponível em: <http://www.altera.com/literature/lit-nio2.jsp>. Acesso em: Jan. 2008, 2008b.

_____. **Quartus II Version 7.2 Handbook Volume 5**: Embedded Peripherals. Disponível em: <http://www.altera.com/literature/quartus2/lit-qts-peripherals.jsp>. Acesso em: Jan. 2008, 2008c.

AZEVEDO, Arnaldo, et al. **Accelerating a Multiprocessor Reconfigurable Architecture with Pipelined VLIW Units**. In Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05), Montreal, Canadá, IEEE Computer Society, pp. 255-257, 8-10 de Junho de 2005.

BACON, David F., GRAHAM, Susan L. e SHARP, Oliver J. **Compiler Transformations for High Performance Computing**. ACM Computing Surveys (CSUR), Volume: 26, *Issue*: 4, IEEE Computer Society, pp. 345-420, Dezembro de 1994.

BECK, Antônio Carlos S. e CARRO, Luigi. **Dynamic Reconfiguration with Binary Translation: Breaking the ILP Barrier with Software Compatibility.** In Design Automation Conference (DAC'2005), Califórnia, EUA, ACM Press, 13-17 de Junho de 2005.

BECKER, Jurgen, et al. **An Industrial/Academic Configurable System-on-Chip Project (CSoC):** Coarse-grain XPP-/Leon-based Architecture Integration. In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'03), Munique, Alemanha, IEEE Computer Society, pp. 1120-1121, 03-07 de Março de 2003.

BHATIA, Dinesh. **Reconfigurable computing**. In Proceedings of the 10th International Conference on VLSI Design, Índia, IEEE Computer Society, pp. 356-359, 04-07 de Janeiro de 1997.

BONDALAPATI, Kiran e PRASANNA, Viktor K. **Reconfigurable Computing Systems**. Proceedings of the IEEE, Volume: 90, *Issue*: 7, IEEE Computer Society, 1201-1217, Julho de 2002.

CALLAHAN, Timothy J. e WAWRZYNEK, John. **Instruction-Level Parallelism for Reconfigurable Computing**. In Proceedings of the 8th International Conference on Field Programmable Logic and Applications (FPL'98), Volume: 1482, Estônia, Springer-Verlag, pp.248-257, 31 de Agosto - 03 de Setembro 1998.

CHEN, Xiaoyong e MASKELL, Douglas L. **Supporting multiple-input, multiple-output custom functions in configurable processors**. Journal of Systems Architecture: the EUROMICRO Journal, Volume: 53, *Issue*: Elsevier North-Holland, pp. 263-271, 5-6 de Maio de 2007.

COMPTON, Katherine e HAUCK, Scott. **Reconfigurable Computing:** A Survey of Systems and Software. ACM Computing Surveys (CSUR), Volume: 34, *Issue*: 2, ACM Press, pp. 171-210, Junho de 2002.

DEHON, André. **Reconfigurable Architectures for General-Purpose Computing**. Relatório Técnico Nº 1586, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Outubro de 1996.

EBELING, Carl, CRONQUIST, Darren C. e FRANKLIN, Paul. **RaPiD –** Reconfigurable Pipelined Datapath. In Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL'96), Alemanha, Springer-Verlag, pp. 126-135, 23-25 de Septembro de 1996.

ESTRIN, Gerald. **Reconfigurable computer origins:** the UCLA fixed-plus-variable (F+V) structure computer. IEEE Annals of the History of Computing, Volume: 24, N^{o} 4, IEEE Computer Society, pp. 3-9, Outubro de 2002.

FISHER, Joseph, FARABOSCHI, Paolo e YOUNG, Cliff. **Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools.** Morgan Kaufmann, 2004.

FLYNN, Michael J. **Computer Architecture:** Pipelined and Parallel Processor Design. Jones and Bartlett Publishers, 1995.

GAISLER, Jiri. **The LEON Processor User's Manual**. Version 2.3.7. Gaisler Research, Agosto de 2001.

GOLDSTEIN, Seth Copen, et al. **PipeRench:** A Coprocessor for Streaming Multimedia Acceleration. In Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99), Atlanta, Georgia, EUA, 2-4 de Maio de 1999.

GONZALEZ, José Artur Quilici. **Uma Metodologia de Projetos para Circuitos com Reconfiguração Dinâmica de Hardware Aplicada a Support Vector Machines**. Tese de Doutorado, Departamento de Sistemas Eletrônicos, USP, São Paulo, 2006.

HARTENSTEIN, Reiner. **Coarse Grain Reconfigurable Architectures**. In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'01), Yokohama, Japão, IEEE Computer Society, pp. 564-569, 30 de Janeiro - 2 de Fevereiro 2001a.

_____. **A Decade of Reconfigurable Computing: a Visionary Retrospective.** In Proceedings of Design, Automation and Test in Europe (DATE'01), Munique, Alemanha, IEEE Computer Society, pp. 642-649, 13-16 de Março de 2001b.

_____. **Reconfigurable Computing: A New Business Model – and its Impact on SoC Design**. In Proceedings of Euromicro Symposium on Digital System Design, IEEE Computer Society, pp. 103-110, 4-6 de Setembro de 2001c.

HAUCK, Scott. **The Roles of FPGAs in Reprogrammable Systems**. In Proceedings of the IEEE, Volume: 86, *Issue*: 4, IEEE Computer Society, pp. 615-638, Abril de 1998.

HAUCK, Scott, et al. **The Chimaera Reconfigurable Functional Unit**. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume: 12, *Issue*: 2, IEEE Computer Society, pp. 206-217, Fevereiro de 2004.

HAUSER, John R. e WAWRZYNEK, John. **Garp:** A MIPS Processor with a Reconfigurable Coprocessor. Proceedings of the 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society, pp. 12-21, 16-18 de April de 1997.

ITU – International Telecommunication Union. **Advanced video coding for generic audiovisual services**, ITU-T H.264 Recommendation, 2005.

JACOB, Jeffrey A. e CHOW, Paul. **Memory Interface and Instruction Specification for Reconfigurable Processors.** In Proceedings of ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays, Califórnia, EUA, pp. 145-154, 1999.

JÚNIOR, Francisco Pereira. **Seleção de Variáveis e Características como Aplicação Paralela para Cluster MPI**. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, Universidade Estadual de Maringá, PR, 2006.

LAM, Monica S. e WILSON, Robert P. **Limits of Control Flow on Parallelism**. In Proceedings of the 19th Annual International Symposium on Computer Architecture, Queensland, Austrália, IEEE Computer Society, pp. 46-57, 19-21 de Maio de 1992.

LEVINE, Benjamin A. e SCHMIT, Herman H. **Efficient Application Representation for HASTE:** Hybrid Architectures with a Single, Transformable Executable. 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03), Napa, CA, EUA, IEEE Computer Society, pp. 101-110, 9-11 de Abril de 2003.

LIPASTI, Mikko H. **Value Locality and Speculative Execution**. Tese de Doutorado, Carnegie Mellon University, Pittsburgh, PA, EUA, 1997.

LOPES, Alba S. Bezerra, COSTA, Miklécio Bezerra da, PEREIRA, Monica Magalhães e SILVA, Ivan Saraiva. **Estudo sobre o impacto do processador hospedeiro no desempenho das arquiteturas reconfiguráveis híbridas.** In I Concurso de Trabalhos de Iniciação Científica em Arquitetura de Computadores e Computação de Alto Desempenho (WSCAD-CTIC), Rio Grande do Sul, Brasil, Outubro de 2007.

MARSHAL, Alan, et al. **A Reconfigurable Arithmetic Array for Multimedia Applications.** In Proceedings of the ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays, Califórnia, EUA, ACM Press, pp. 135-143, 1999.

MIRSKY, Ethan e DEHON, André. **MATRIX:** A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, Napa, CA, EUA, IEEE Computer Society, pp. 157-166, 17-19 de April de 1996.

MIYAMORI, Takashi e OLUKOTUM, Kunle. **REMARC: Reconfigurable Multimedia Array Coprocessor.** In Proceedings of ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays, Califórnia, EUA, ACM Press, 1998.

NAVABI, Zainalabedin. **VHDL:** Analysis and Modeling of Digital Systems. McGraw-Hill, 1998.

NELSON, Katherine e HAUCK, Scott. **Mapping Methods for the Chimaera Reconfigurable Functional Unit**. Relatório Técnico, Dept ECE, Northwestern University, EUA, 1997.

PARK , Joseph C. H. e SCHLANSKER, Mike. **On Predicated Execution**. Relatório Técnico HPL-91-58, Software and Systems Laboratory, Hewlett Packard, Maio de 1991.

PEREIRA, Monica Magalhães, OLIVEIRA, Bruno Cruz de, SILVA, Ivan Saraiva. **RoSA: a Reconfigurable Stream-based Architecture.** In Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design, Rio de Janeiro, Brasil, ACM Press, pp. 159-164, 3-6 de Setembro de 2007.

PHILIPS SEMICONDUCTORS. **An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture**. White Paper. http://whitepapers.silicon.com, 2007.

PLATZNER, Marco. **Reconfigurable Computer Architectures**. e&i Elektrotechnik und Informationstechnik, Volume: 155, *Issue*: 3, Springer, pp. 143-148, 1998.

POZZI, Laura. **Compilation Techniques for Exploiting Instruction Level Parallelism, a Survey**. Relatório Técnico TR-99.2, Politecnico di Milano, Milão, Itália, 1999, Disponível em: <http://icwww.epfl.ch/~lpozzi/pub.html>. Acesso em: Jun. 2007.

POZZI, Laura. **Methodologies for the Design of Application-Specific Reconfigurable VLIW Processors**. Tese de Doutorado, Politecnico di Milano, Milão, Itália, 2000.

RAO, K. Ramamohan. e YIP, P. **Discrete Cosine Transform**: algorithms, advantages, applications. Academic Press Professional, Inc. 1990.

RAU, B. Ramakrishna e FISHER, Joseph A. **Instruction-Level Parallel Processing:** History, Overview and Perspective. HP Labs Relatório Técnico, HP Laboratories, 26 de Outubro de 1992.

RAMIREZ, Robert W. **The FFT** Fundamentals and Concepts. Prentice-Hall, Inc., 1985.

RYCHLIK, Bohuslav, et al. **Efficacy and performance impact of value prediction**. In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, Paris, França, IEEE Computer Society, pp.148-154, 12-18 de Outubro de 1998.

SANCHEZ, Eduardo, et al. **Static and Dynamic Configurable Systems.** IEEE Transactions on Computers, Volume; 48, *Issue:* 6, IEEE Computer Society, pp. 556-564, Junho de 1999.

SCARAFICCI, Rafael Augusto. **Arquiteturas VLIW**. Arquiteturas de Computadores I, Instituto de Computação, UNICAMP, 2006.

SCHLANSKER, Michael, et al. **Compilers for Instruction-Level Parallelism**. Computer, Volume: 30, *Issue*: 12, IEEE Computer Society, pp. 63-69, Dezembro de 1997.

SHUKLA, Sunil, BERGMANN, Neil W. e BECKER, Jurgen. **QUKU:** A Fast Run Time Reconfigurable Platform for Image Edge Detection. International Workshop on Applied Reconfigurable Computing (ARC'06), Delft, Holanda, 1-3 de Março de 2006, Springer.

SINGH, Hartej, et al. **MorphoSys:** An Integrated Re-configurable Architecture. In Proceedings of NATO Symposium System Concepts and Integration, Califórnia, USA, Abril de 1998.

SUN MICROSYSTEMS. **The SPARC Architecture Manual:** Version 8. Disponível em: <http://www.sun.com>. Acesso em: Jun. 2007.

TODMAN, T. J. et al. **Reconfigurable Computing:** architectures and design methods. IEE Proceedings-Computers and Digital Techniques, IEEE Computer Society, pp. 193-207, Março de 2005.

WALLACE, Gregory K. **The JPEG still picture compression standard**. Communications of the ACM, Volume: 34, *Issue*: 4, ACM Press, pp.30-44, Abril de 1991.

WITTING, Ralph D. e CHOW, Paul. **OneChip:** An FPGA Processor with Reconfigurable Logic. Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, EUA, IEEE Computer Society, pp. 126-135, 17-19 de Abril de 1996.

XPP TECHNOLOGIES. **XPP-III Processor Overview** - White Paper. XPP Technologies, 6 Julho de 2006.

YE, Zhi Alex, SHENOY, Nagaraj e BANERJEE, Prithviraj. A C Compiler for a Processor with a Reconfigurable Functional Unit. In Proceedings of the 18th International Symposium on Field Programmable Gate Arrays, Califórnia, EUA, ACM Press, pp. 95-100, 2000.

ANEXO I – Conjunto de Instruções do Nios II

Instruções de Transferência de Dados					
Instruções de Transferência de Dados Largos					
ldw	stw	ldwio	stwio		
Instruções de Transferência de Dados Curtos					
ldb	ldbu	stb	ldh	ldhu	sth
ldbio	ldbuio	stbio	ldhio	ldhuio	sthio

Instruções Lógicas e Aritméticas					
and	or	xor	nor		
andi	ori	xori	andhi	orhi	xorhi
add	sub	mul	div	divu	
addi	subi	muli	mulxss	mulxuu	mulxsu

Instruções de Mover					
mov	movhi	movi	movui	movia	

Instruções de Comparação					
cmpeq	cmpne	cmpge	cmpgeu	cmpgt	cmpgtu
cmple	cmpleu	cmplt	cmpltu		
cmpeqi	cmpnei	cmpgei	cmpgeui	cmpgti	cmpgtui
cmplei	cmpleui	cmplti	cmpltui		

Instruções de Deslocamento e Rotação					
rol	ror	roli			
sll	slli	sra	srl	srai	srli

Instruções de Controle de Programa					
Instruções de Salto e Chamadas Incondicionais					
call	callr	ret	jmp	jmpi	br
Instruções de Salto Condicionais					
bge	bgeu	bgt	bgtu	ble	bleu
blt	bltu	beq	bne		

Outras Instruções de Controle					
trap	eret	break	bret	rdctl	wrctl
flushd	flushi	initd	initi	flushp	sync

ANEXO II – Resultados de Freqüência e Área de Todas as Implementações

Frequência									
	RoSE sem Limitação			RoSE com Limitação			sem RoSE		
	Nios II/e	Nios II/s	Nios II/f	Nios II/e	Nios II/s	Nios II/f	Nios II/e	Nios II/s	Nios II/f
Cyclone II - EP2C35F672C6	62,88	65,01	63,61	64,06	63,42	65,12	63,01	63,37	63,00
Stratix II - EP2S60F1020C3	85,16	96,10	95,33	97,03	91,12	100,12	92,71	92,32	91,02

Cyclone II - EP2C35F672C6									
	RoSE sem Limitação			RoSE com Limitação			sem RoSE		
	Nios II/e	Nios II/s	Nios II/f	Nios II/e	Nios II/s	Nios II/f	Nios II/e	Nios II/s	Nios II/f
Total de Elementos Lógicos	22.951	23.849	24.574	22.488	23.183	24.090	20.466	21.275	22.025
Total de Funções Combinacionais	20.420	21.160	21.784	19.845	20.574	21.196	17.476	18.239	18.774
Registradores Lógicos Dedicados	10.520	11.107	11.521	10.388	10.975	11.400	10.949	11.536	11.961
Total de Pinos	82	82	82	82	82	82	82	82	82
Total de Bits de Memória	16.128	52.096	69.824	16.128	52.096	70.016	16.128	52.096	70.016
Multiplicadores Embutidos de elementos de 9-bits	16	20	20	16	20	20	64	68	68

Cyclone II - EP2C35F672C6									
	RoSE sem Limitação			RoSE com Limitação			sem RoSE		
	Nios II/e	Nios II/s	Nios II/f	Nios II/e	Nios II/s	Nios II/f	Nios II/e	Nios II/s	Nios II/f
Processador	909	1.748	2.436	908	1.749	2.446	961	1.746	2.446
Memória	85	87	90	86	87	86	86	85	87
Outros Componentes	520	503	525	521	500	531	472	519	523
RoSA	21.437	21.511	21.523	20.973	20.847	21.027	18.947	18.925	18.969
Células	*18.420*	*18.494*	*18.491*	*17.952*	*17.831*	*18.002*	*15.926*	*15.903*	*15.947*
Gerenciador de Configuração	*917*	*918*	*921*	*921*	*917*	*923*	*922*	*922*	*918*
Banco de Registradores	*2.100*	*2.099*	*2.111*	*2.100*	*2.099*	*2.102*	*2.099*	*2.100*	*2.104*

Stratix II - EP2S60F1020C3									
	RoSE sem Limitação			RoSE com Limitação			sem RoSE		
	Nios II/e	Nios II/s	Nios II/f	Nios II/e	Nios II/s	Nios II/f	Nios II/e	Nios II/s	Nios II/f
Utilização Lógica	25.462	26.042	26.468	24.941	25.515	25.924	23.458	23.994	24.651
ALUTs Combinacionais	20.284	20.766	20.988	19.722	20.178	20.403	17.633	18.065	18.254
Registradores Lógicos Dedicados	15.072	15.413	15.729	14.932	15.274	15.598	15.564	15.908	16.219
Total de Pinos	83	83	83	83	83	83	83	83	83
Total de Blocos de Bits de Memória	16.128	52.096	70.016	16.128	52.096	70.016	16.128	52.096	70.016
Bloco DSP de elementos de 9-bits	48	56	56	48	56	56	192	200	200

Stratix II - EP2S60F1020C3									
	RoSE sem Limitação			RoSE com Limitação			sem RoSE		
	Nios II/e	Nios II/s	Nios II/f	Nios II/e	Nios II/s	Nios II/f	Nios II/e	Nios II/s	Nios II/f
Processador	653	1.006	1.282	647	1.002	1.283	653	1.017	1.299
Memória	62	64	64	61	60	62	65	63	61
Outros Componentes	360	441	435	362	444	436	354	447	434
RoSA	19.209	19.255	19.207	18.652	18.672	18.622	16.561	16.538	16.460
Células	*18.705*	*18.736*	*18.695*	*18.137*	*18.148*	*18.112*	*16.032*	*16.015*	*15.943*
Gerenciador de Configuração	*341*	*354*	*346*	*352*	*357*	*347*	*351*	*356*	*352*
Banco de Registradores	*163*	*165*	*166*	*163*	*167*	*163*	*178*	*167*	*165*

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo