Antonio Carlos Fernandes da Silva

Compilação para arquitetura reconfigurável

São José do Rio Preto - SP, Brasil

Junho de 2009

Antonio Carlos Fernandes da Silva

Compilação para arquitetura reconfigurável

Dissertação apresentada para obtenção do título de Mestre em Ciência da Computação, área de Sistemas de Computação junto ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

Orientadora:
Profa. Dra. Renata Spolon Lobato

Programa de Pós-Graduação em Ciência da Computação
Departamento de Ciências de Computação e Estatística
Universidade Estadual Paulista "Júlio de Mesquita Filho"

São José do Rio Preto - SP, Brasil

Junho de 2009

Antonio Carlos Fernandes da Silva

Compilação para arquitetura reconfigurável

Dissertação apresentada para obtenção do título de Mestre em Ciência da Computação, área de Sistemas de Computação junto ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

BANCA EXAMINADORA

Profa. Dra.
Renata Spolon Lobato
UNESP - São José do Rio Preto
Orientadora

Prof. Dr.
Aleardo Manacero Junior
UNESP - São José do Rio Preto

Prof. Dr.
Jorge Luiz e Silva
USP - São Carlos

São José do Rio Preto, 18 de Junho de 2009

Nem tudo que se enfrenta pode ser modificado mas nada pode ser modificado até que seja enfrentado.

Albert Einstein

Dedico aos meus pais.

Agradecimentos

Aos meus pais e minha namorada, pelo incentivo e apoio.

À Profa. Dra. Renata Spolon Lobato, pela orientação, incentivo, paciência e disponibilidade na realização deste trabalho. Ao Prof. Dr. Aleardo Manacero Junior pela participação em vários momentos do trabalho.

Aos amigos Adriana, André, Evandro, Geraldo, Jorge, Leandro, Pedro, Otávio, Rodrigo, Tiago, Vanessa e Willian pelo apoio, amizade e momentos de trabalho e de descontração passados juntos.

Agradeço a Universidade Tecnológica Federal do Paraná, pelo auxílio durante o desenvolvimento deste trabalho. Aos professores da UTFPR que me auxiliaram sempre que necessário.

Resumo

A computação reconfigurável aparece como uma alternativa viável para a crescente demanda por desempenho em sistemas computacionais. Devido ao grande desenvolvimento de pesquisas nesta área, tornam-se cada vez mais necessárias ferramentas para auxílio ao desenvolvimento ou migrações de aplicativos para as arquiteturas que dão suporte a este novo paradigma. Dentro deste contexto, neste trabalho é apresentado o desenvolvimento de um compilador para arquitetura reconfigurável, desenvolvido com base no *framework* Phoenix, que tem como objetivo gerar código para o Nios II. Nios II é um processador RISC virtual que pode ser executado sobre um FPGA. Os resultados obtidos durante o desenvolvimento do trabalho demonstram sua viabilidade e sua utilidade na geração de aplicativos para plataformas reconfiguráveis.

Abstract

The reconfigurable computing appears as an possible alternative for the growing demand for performance in computing systems. Due to the large research's development in this area, it becomes increasingly necessary tools for development aiding or migration of applications for architectures that supports this new paradigm. In this context, this work presents the development of a compiler for reconfigurable architecture. It was based on Phoenix framework, that aims to generate code for Nios II. Nios II is a virtual RISC processor that can be implemented on a FPGA. The results that was obtained while the work development evidences its practicability and utility to generate applications for reconfigware.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

1.1 Motivação

As arquiteturas reconfiguráveis têm como grande característica a possibilidade de modificação do *hardware* durante o ciclo de vida do dispositivo. Segundo Cardoso (CARDOSO, 2000) existem áreas de aplicação em que a utilização destes sistemas fornece implementações com desempenhos inalcançáveis quando comparados com sistemas computacionais que utilizam processadores de uso geral, visto que estas arquiteturas têm como principal objetivo executar o processamento mais intenso em *hardware*, acelerando assim sua execução.

A programação para este tipo de arquitetura depende do conhecimento em projeto de *hardware*, por isso um grande desafio é criar ferramentas de apoio que permitam a qualquer programador de linguagem de alto nível gerar circuitos de *hardware* com qualidade.

Dentro deste contexto, apresenta-se neste trabalho o desenvolvimento de um compilador que permita a programadores de linguagem de alto nível desenvolver aplicativos para arquiteturas reconfiguráveis que utilizem o processador de núcleo virtual Nios II (ALTERA, 2007a).

Este compilador apresenta alguns diferenciais quando comparado a outros compiladores, como por exemplo, a geração de aplicativos para o processador virtual Nios II e também o desenvolvimento de um módulo externo para manipulação de tipos de dados de ponto flutuante.

1.2 Objetivos

O presente trabalho tem como objetivo acrescentar novas funcionalidades ao *framework* Phoenix (DUARTE, 2006). Com a inclusão destas novas funcionalidades busca-se desenvolver um compilador para arquitetura reconfigurável, que tem como alvo o processador virtual Nios II da Altera, que visa ser parte do projeto *Architect+* (DUARTE, 2006), do Laboratório de Computação Reconfigurável do Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo.

O Phoenix é um *framework* para síntese de alto nível de circuitos digitais, e foi utilizado como ponto de partida do desenvolvimento.

1.3 Organização do texto

No capítulo 2 são apresentados os conceitos fundamentais da computação reconfigurável, descrevendo os tipos de computação existentes, a estrutura de um dos dispositivos reconfiguráveis existentes no mercado e um resumo dos processos para geração de *software* para sistemas reconfiguráveis. Além disso, este capítulo também aborda o processo de compilação para arquiteturas reconfiguráveis, a geração de representação intermediária, as alterações aplicadas buscando a otimização do código gerado e trabalhos relevantes na área.

No capítulo 3 são descritos o *framework* Phoenix e o processador virtual Nios II. Busca-se neste capítulo demonstrar as características do *framework*, que servirá como base para o trabalho e também as características do processador para o qual o código será gerado.

No capítulo 4 são apresentados o desenvolvimento do trabalho e os módulos gerados para a obtenção do compilador.

No capítulo 5 são apresentados os testes realizados para verificar o funcionamento e também corretude do compilador, além de justificar algumas abordagens adotadas durante

o desenvolvimento.

No capítulo 6 são apresentadas as conclusões e propostas de trabalhos futuros.

2 Computação Reconfigurável

2.1 Introdução

Neste capítulo é descrito o paradigma de computação reconfigurável, que teve seu conceito básico apresentado em 1963 por Estrin (ESTRIN et al., 1963). Este modelo de computação busca unir as facilidades de um processador de uso comum com a velocidade de execução de um processador para uso específico.

Este capítulo também aborda o processo de compilação para arquiteturas reconfiguráveis, a geração de representação intermediária e as alterações que podem ser aplicadas buscando a otimização do código gerado.

Inicialmente são apresentados os modelos de computação clássicos existentes e os conceitos ligados a computação reconfigurável. É apresentado também um dispositivo de lógica reconfigurável, além dos processos de criação de *software* para computação reconfigurável.

Na compilação para arquiteturas reconfiguráveis o processo de compilação tem preocupações um pouco diferentes do processo tradicional, principalmente com o que diz respeito ao particionamento da aplicação e a definição do que será desenvolvido em *software* e o que será mapeado em *hardware* (PANAINTE; BERTELS; VASSILIADIS, 2007).

Mesmo no caso de compilação para uma arquitetura já definida e na qual o aplicativo não irá gerar circuito de *hardware*, deve existir uma maior preocupação com a identificação de pontos de paralelismo a nível de instruções que podem ser explorados. Esta identificação pode ser feita através da representação intermediária, que é formada por uma série

de estruturas com o objetivo de permitir a exploração de paralelismo. Esta representação pode ser formada por vários grafos que armazenam o fluxo e dependência de dados, fluxo e dependência de controle, informações de ordem de execução entre os blocos básicos e estruturas de controle.

2.2 Modelos de computação

A execução de algoritmos pode ser realizada tradicionalmente através de ASICs (*Applicattion Specific Integrated Circuit*) ou microprocessadores de uso geral (HAUCK; DEHON, 2007).

Os ASICs são desenvolvidos para aplicações específicas, tendo assim um ótimo desempenho na execução destas aplicações, mas sofrem com a falta de flexibilidade, ou seja, caso necessitem ser alterados devem ser totalmente reprojetados, o que é uma tarefa que consome tempo e tem um custo elevado. Por outro lado, os processadores de uso geral são dispositivos que permitem sua programação para a execução de qualquer tipo de operação digital, alterando-se o seu conjunto de instruções, sem a necessidade de alteração de *hardware*. Estes processadores têm um desempenho menor quando comparado com ASICs, mas apresentam flexibilidade maior (HAUCK; DEHON, 2007).

Para tentar unir as melhores características de cada um deste modelos de computação surgiu a computação reconfigurável que, segundo Panainte (PANAINTE; BERTELS; VASSILIADIS, 2007), vem demonstrando um grande crescimento devido à sua promessa de aliar desempenho do *hardware* e a flexibilidade do *software*. O conceito básico deste modelo surgiu nos anos 60, quando Estrin (ESTRIN et al., 1963) desenvolveu uma máquina que poderia ser reconfigurada através da alteração da interconexão de seus componentes, que era feita através de um painel de fios e conectores. Este é o mesmo conceito utilizado atualmente, com a utilização de componentes de *hardware* que podem ter sua lógica alterada através de código gerado em alguma linguagem de descrição de *hardware*, em substituição aos fios e conectores da máquina de Estrin.

Embora sistemas reconfiguráveis possam ser vistos como uma alternativa para a crescente demanda por desempenho, a geração de sua arquitetura é complexa e exige conhecimento de *hardware* e de *software* para que todos os seus benefícios sejam explorados (PANAINTE; BERTELS; VASSILIADIS, 2007).

Para tornar este cenário mais atrativo para programadores em geral, ferramentas que permitem o desenvolvimento de sistemas de *hardware* vêm sendo desenvolvidas com o intuito de permitir que um programador, usando linguagem de alto nível, gere um circuito tão eficiente quanto o gerado de forma manual por um especialista da área. Com o auxílio destas ferramentas torna-se possível a reutilização de código, tornando o processo de desenvolvimento de sistemas reconfiguráveis mais rápido e fácil.

Segundo Hauck (HAUCK; DEHON, 2007) a geração de sistemas reconfiguráveis pode ser realizada das seguintes formas:

- Automática: meio simples e rápido para a descrição que pode ser feita em linguagem de alto nível ou em linguagens de descrição de *hardware* (HDLs), como por exemplo VHDL (*Very High Speed Integrated Circuits Hardware Description Language*) e Verilog. Esta descrição é substituída por elementos de *hardware*, que são depois mapeados para os componentes específicos da arquitetura do dispositivo reconfigurável (mapeamento tecnológico). O compilador pode ser responsável por detectar automaticamente as partes da descrição que deverão ser mapeadas para *hardware* e *software*;

- Automática/Manual: neste tipo de processo o projetista decide quais as partes do sistema vão ser mapeadas para *hardware* e *software*. Depois de feita a descrição estrutural do sistema, a mesma pode ser mapeada para o dispositivo reconfigurável por ferramentas de fabricantes;

- Manual: neste tipo de processo o projetista é responsável por toda a definição do sistema, tornando o processo lento e mais suscetível a erros.

Para o desenvolvimento de arquiteturas reconfiguráveis são utilizados dispositivos lógicos conhecidos como FPGAs (*Field-programmable gate array*), apresentados na seção 2.3.

2.3 FPGA (*Field-programmable gate array*)

Os FPGAs são constituídos pelo arranjo de células lógicas em um circuito integrado, e oferecem suporte para a implementação de um grande número de circuitos lógicos. A estrutura geral de um FPGA pode ser observada na Figura 1. Verifica-se que um FPGA é formado basicamente por blocos lógicos, blocos de interconexão e chaves de interconexão. As chaves de interconexão são as responsáveis por fazer o roteamento, para conectar de forma conveniente os blocos lógicos, atendendo assim a necessidade de cada projeto.

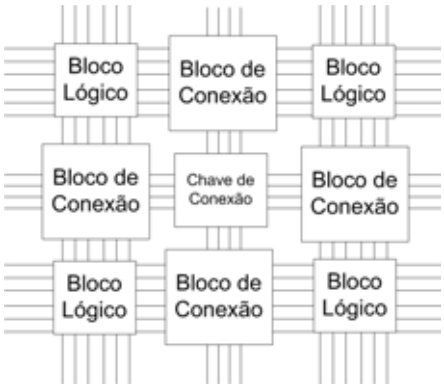

Figura 1: Estrutura de um FPGA (HAUCK; DEHON, 2007).

A forma de interconexão dos blocos lógicos de um FGPA pode mudar, definindo assim a estratégia de interconexão. Na Figura 1 é mostrada uma estrutura totalmente segmentada, na qual cada parte do circuito é separada por uma chave de interconexão. Outras estruturas possíveis são roteamento hierárquico ou roteamento parcialmente segmentado na qual chaves de conexão são utilizadas para interligar blocos contendo vários FPGAs, como pode ser visto na Figura 2.

As funções lógicas dentro de um FPGA podem ser implementadas na forma de um bloco de memória LUT (*Look-up table*), na qual as células de armazenamento são utiliza-

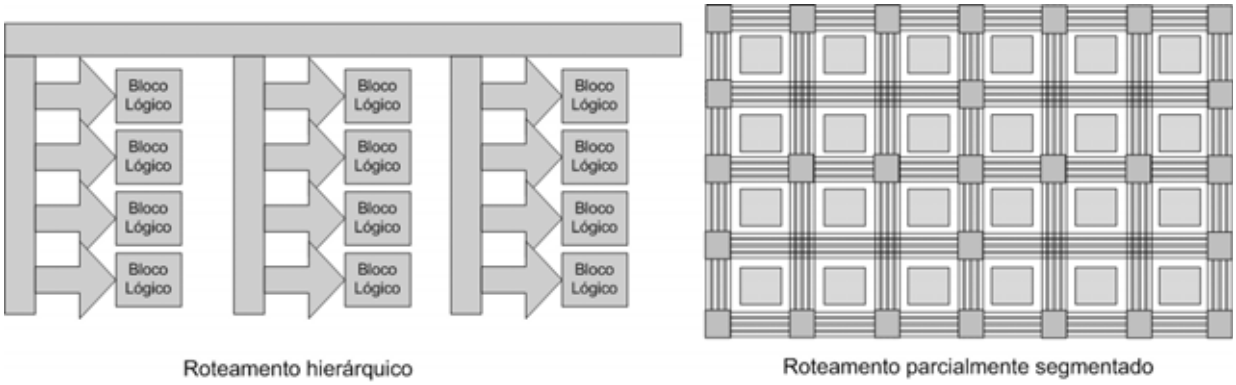

Figura 2: Estratégias de interconexão dos blocos lógicos de um FPGA (PONTARELLI et al., 2008).

das para implementar pequenas funções lógicas. Estas células são voláteis, ou seja, caso sejam desenergizadas perdem seu conteúdo, necessitando ser reprogramadas (MORENO; PENTEADO; SILVA, 2005).

As células de um FPGA podem variar de tamanho, o que é denominado como granulação da célula e indica a sua complexidade. Segundo Hauck (HAUCK; DEHON, 2007), as arquiteturas podem ser classificadas em três sub-conjuntos de acordo com sua granulação:

- **Granulação fina**: as células implementam portas lógicas *AND*, *OR*, *XOR* e *NOR* que podem ser combinadas para formarem circuitos mais complexos;
- **Granulação média**: as células implementam funções lógico-aritméticas e multiplicadores de poucos bits;
- **Granulação grossa**: as células implementam somadores e multiplicadores de maiores quantidades de bits e em nível de bytes. Pode-se encontrar arquiteturas como pequenos processadores e unidades lógica e aritmética.

2.4 Arquitetura FPGA/UCP

Os FPGAs são ineficientes para a realização de algumas operações, como por exemplo laços de tamanho variável e controle de saltos, além de necessitar de uma fonte de controle

para a sua reconfiguração. Por estes motivos e também para obter maior desempenho e suporte a um número maior de aplicações, os FPGAs são normalmente conectados a processadores de uso geral, criando assim uma arquitetura híbrida FPGA/UCP (MITTAL et al., 2007).

Segundo Mittal (MITTAL et al., 2007), este novo modelo de arquitetura tende a criar uma grande necessidade de migração de *softwares* projetados para uso em processador de uso geral, para o modelo híbrido FPGA/UCP, pelo fato deste modelo alcançar maior desempenho a um baixo custo. Este tipo de migração é um desafio, pois nem sempre os usuários possuem o código fonte de suas aplicações e nem sempre estão disponíveis ferramentas para auxiliar neste processo.

Na arquitetura híbrida o FPGA pode ser utilizado de várias formas, como por exemplo:

- Unidade funcional dentro de um processador hospedeiro (HAUCK et al., 2004) (RAZDAN; SMITH, 1994): esta unidade funcional executa nas vias de dados do processador, o que permite a adição de instruções especiais, sem alterar o modo de programação;
- Co-processador (WITTIG; CHOW, 1996) (HPCWIRE, 2009): é iniciado pelo processador do qual recebe os dados para a realização de processamento de forma independente, podendo acessar memórias internas ou externas ao sistema;
- Sistema multi-processador (VUILLEMIN et al., 1996) (LAUFER; TAYLOR; SCHMIT, 1999): o FPGA é interligado a um processador de uso geral. A comunicação entre eles ocorre através de primitivas especializadas, assim como nos sistemas multiprocessadores tradicionais;
- Unidade de processamento externa (QUICKTURN, 1999a) (QUICKTURN, 1999b): Semelhante ao uso do FPGA como co-processador, mas neste caso existe pouca comunicação entre o processador e o dispositivo reconfigurável.

2.5 Compilação para Arquiteturas Reconfiguráveis

2.5.1 Representação Intermediária

A representação intermediária é uma tradução do programa-fonte, através da qual o módulo de *back-end* gera o código-alvo, realizada após o programa passar pelas análises léxica, sintática e semântica, e estar correto do ponto de vista das três análises. Embora seja possível a geração do código-alvo diretamente através do programa-fonte, a representação intermediária apresenta alguns benefícios, como por exemplo a facilidade de criação de um compilador para uma máquina-alvo diferente, trocando-se apenas o módulo de *back-end*, e a utilização de um otimizador independente da máquina (AHO et al., 2007).

A representação intermediária criada por um compilador para arquiteturas reconfiguráveis pode ser formada por uma série de grafos, descritos a seguir:

- Grafo de fluxo de controle (GFC): O grafo de fluxo de controle é um grafo dirigido, onde cada nó representa um segmento do código e cada arco representa um caminho possível para a seqüência de execução. Uma seqüência é um conjunto de instruções que devem ser executadas em ordem, a última instrução de uma seqüência é uma instrução de salto e a primeira é o destino de determinado salto;

  Este grafo constitui a base para a geração dos grafos de dependências para exploração do paralelismo máximo do programa (CARDOSO, 2000), e é criado durante a fase de tradução do código-fonte, sendo gerado através de atributos específicos;

- Grafo de dependência de controle (GDC): O grafo de dependência de controle é um grafo dirigido, que engloba o conjunto de nós do grafo de fluxo de controle, onde as ligações entre os blocos representam a dependência de controle e restringem a ordem de execução dos blocos, de forma a preservar a funcionalidade do programa. Para cada nó deste grafo existe um nó correspondente no grafo de fluxo de controle;

- Grafo de dependência de dados (GDD): Para a geração do grafo de dependência de dados, cada nó do grafo de fluxo de controle é correspondente a um nó no grafo de

dependência de dados, as ligações entre os nós representam dependência de ordem de execução em termos de dependência de dados;

Através deste grafo é possível explorar o paralelismo a nível de blocos básicos do programa. Sempre que dois blocos forem independentes em termos de fluxo de dados, os mesmos podem ser executados em paralelo;

- Grafo de fluxo de dados (GFD): O grafo de fluxo de dados demonstra o fluxo de dados entre as operações do programa, seus nós representam as operações e a ligação entre dois nós representa a existência de dependência de fluxo entre eles e todas as condições segundo as quais determinada operação é realizada, ou seja as construções condicionais.

Na Figura 3 é possível verificar os grafos GFC, GDC, GDD e GFD gerados para um pequeno programa.

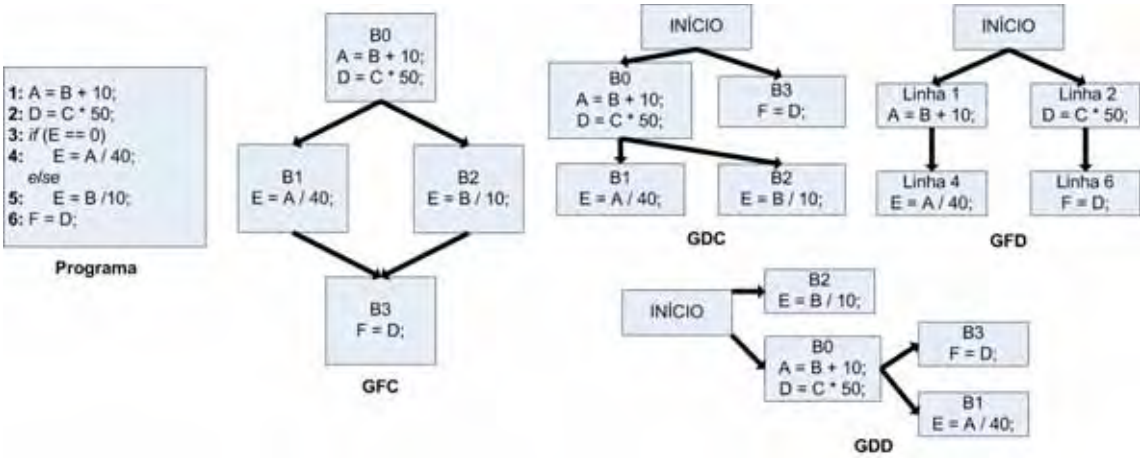

Figura 3: Exemplo de GFC, GDC, GFD, GDD (DUARTE, 2006).

- Grafo de dependência de fusão (GDF) : O grafo de dependências de fusão, apresentado na Figura 4, proposto por Cardoso (CARDOSO, 2000), é um grafo dirigido, cujos nós são os blocos básicos gerados pelo GFC, onde os nós fontes indicam os controladores e os nós destinos indicam os nós que necessitam ser controlados. Para cada nó do GDD que represente uma dependência de dados rotulada com a mesma variável

local, é inserido um ponto de seleção, que irá indicar qual definição de determinada variável atingirá o próximo ponto;

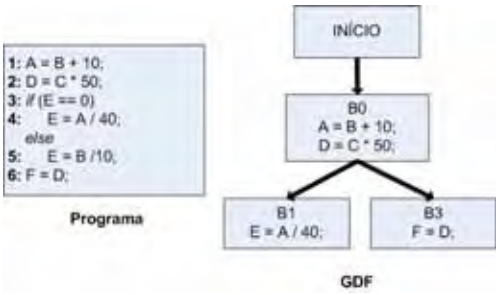

Figura 4: Exemplo de GDF.

- Grafo hierárquico de tarefas (GHT): O grafo hierárquico de tarefas, apresentado na Figura 5, proposto por Girkar (GIRKAR; POLYCHRONOPOULOS, 1992), construído através do GDD e GDF, é um grafo que lida de forma eficiente com o paralelismo funcional, pois permite a representação explícita de todos os fluxos de controle e dependências de dados, fluxos de controle múltiplos e os ciclos existentes no programa fonte. Este grafo é formado por três tipos de nós:
  - Nós simples: representam instruções ou blocos básicos do programa;
  - Nós compostos: representam blocos de decisão *if-then-else*;
  - Nós ciclos: representam blocos que podem ser executados várias vezes consecutivas dentro do GHT.

- Grafo de dominância: Este grafo auxilia na construção do GDC, indicando os caminhos possíveis para se alcançar determinado nó. Considerado-se os vértices $v1$ e $v2$, pode-se dizer que $v1$ é dominante sobre $v2$ se todos os caminhos partindo do nó START que chegam a $v2$, passam por $v1$. Todo nó é dominante sobre si mesmo. Este grafo é apresentado na Figura 6(a);

- Grafo de pós-dominância: Grafo semelhante ao grafo de dominância, mas demonstra a dominância ao se percorrer o grafo partindo do nó *End* em direção ao *Start*. Este grafo é apresentado na Figura 6(b).

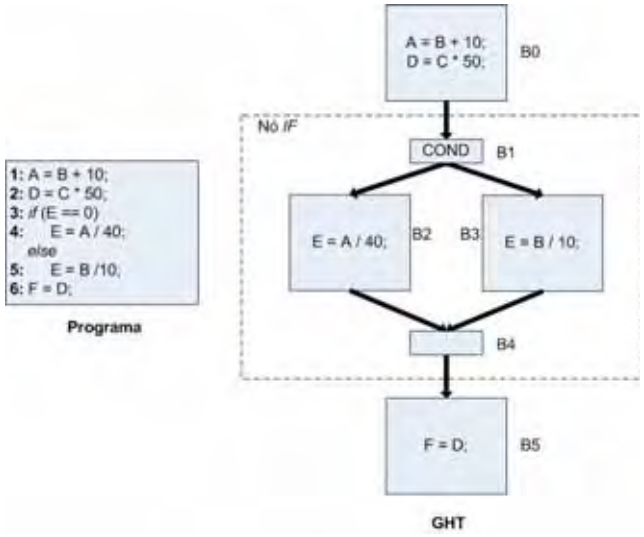

Figura 5: Exemplo de GHT.

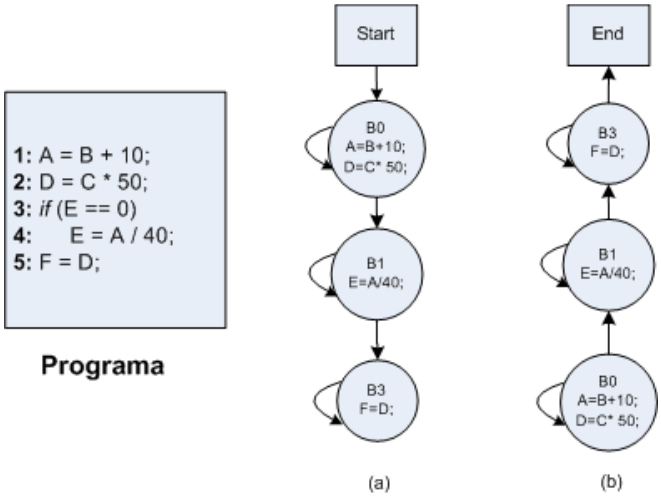

Figura 6: Exemplo dos grafos de dominância (a) e pós-dominância (b).

2.5.2 Otimizações

Nesta seção serão apresentadas técnicas que podem ser utilizadas para melhoria no código gerado pelo compilador. Estas melhorias também são chamadas de otimizações, embora não seja possível garantir que estas alterações gerem código ótimo. Apesar disso, algumas alterações simples podem melhorar significativamente o tempo de execução ou as exigências de espaço do programa-alvo (AHO et al., 2007).

Eliminação de sub-expressões comuns

Uma sub-expressão é considerada comum caso seu valor seja previamente computado e não mude até atingir sua próxima utilização dentro do bloco, como exemplificado na Figura 7. Um exemplo de sub-expressão comum pode ser visto nas instruções $T_8 := 4 * J$ e $T_{10} := 4 * J$, em que o valor de J não é alterado entre a execução da primeira instrução e da ultima, o que faz com que as duas resultem no mesmo valor.

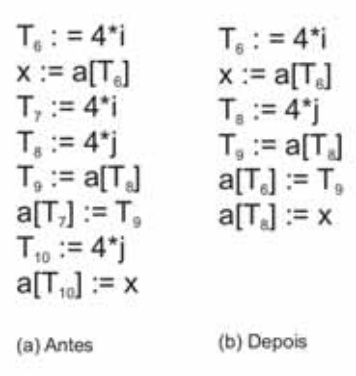

Figura 7: Exemplo da utilização da eliminação de sub-expressões comuns (AHO et al., 2007).

Propagação de cópias

A propagação de cópias consiste em substituir f por g, após o enunciado de cópia f := g. Este tipo de substituição visa eliminar pelo menos uma atribuição de valor, que se tornará um código morto, como mostrado na Figura 8.

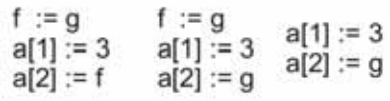

Figura 8: Exemplo da utilização da propagação de cópias.

Transposição para constantes

A transposição para constante consiste em observar, em tempo de execução, que o valor de uma determinada variável não sofre alteração, tornando possível a utilização de

uma constante. Na Figura 9(a) tem-se um exemplo da utilização desta otimização, na qual pode-se verificar que a variável `c` terá seu valor constante e pode ser substituída pelo valor 12 (Figura 9(b)). Após esta substituição, verifica-se que a expressão *if (12 > 10)* será sempre verdadeira e pode ser eliminada, através de uma otimização conhecida como eliminação de código morto. Com isso, conclui-se que o retorno de um valor fixo pode substituir todo o bloco de código (Figura 9(c)).

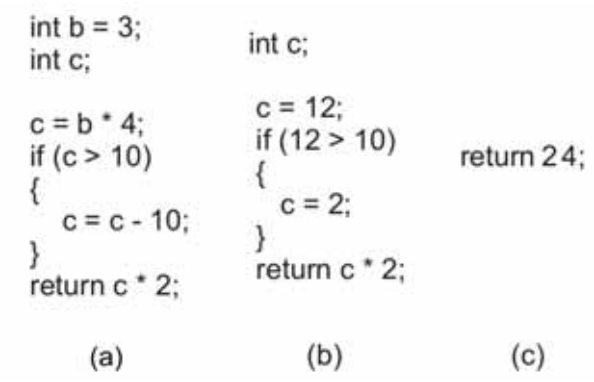

Figura 9: Exemplo da utilização da transposição para constantes.

Movimentação de código ciclo-invariante

Esta é uma modificação que busca diminuir a quantidade de código em um laço, tomando uma instrução que produz sempre o mesmo resultado e colocando-a fora do laço. Na Figura 10(a) a instrução $\texttt{limite} - 2$ é um código ciclo-invariante, ou seja, não se altera durante a execução do laço, portanto não precisa ser calculado a cada iteração. Desta forma, esta instrução pode ser executada uma única vez (Figura 10(b)).

Figura 10: Exemplo da utilização da movimentação de código ciclo-invariante (AHO et al., 2007).

Otimização *Peephole*

Um *peephole* é uma pequena janela móvel no programa-alvo. Nesta janela busca-se fazer melhorias no código e, com estas melhorias, criar oportunidades para melhorias adicionais. Isto é feito através da avaliação de pequenas seqüências de instruções, buscando-se a substituição desta seqüência de instruções por outra equivalente, que seja mais rápida (BANSAL; AIKEN, 2006).

Segundo Aho (AHO et al., 2007) as seguintes transformações são características da otimização *peephole*:

- Eliminação de instruções redundantes: consiste em eliminar instruções que resultam no mesmo resultado, como por exemplo: em `mov R0, R1` e `mov R1, R0`, a segunda instrução irá assegurar que o valor de `R1` já está em `R0`;
- Otimização de fluxo de controle: consiste em eliminar desvios que levem a outros desvios, como por exemplo: em `if a < b goto L1`; `L1 goto L2`, um desvio pode ser eliminado gerando o seguinte código: `if a < b goto L2`;
- Simplificações algébricas: consiste em substituir uma expressão de maior custo computacional por uma mais simples. Como por exemplo, substituir x^2 por $x * x$.

2.5.3 Alocação de registradores

A alocação de registradores consiste em determinar os registradores a serem utilizados para armazenar os dados durante a execução das instruções. A determinação da alocação de registradores deve ser feita durante o processo de geração de código para a arquitetura alvo, através da representação intermediária.

A alocação de registradores busca reduzir o número de acessos à memória e apresenta uma grande importância pelo fato de os processadores não possuírem registradores suficientes para armazenar todas as variáveis de um programa, sendo necessário assim acessos a memória para a buscar as variáveis que não puderam ser alocadas. Vários algoritmos foram propostos para alocação de registradores, as principais técnicas utilizadas nestes

algoritmos são:

- Alocação de registradores por coloração de grafos (COOPER; HARVEY; PEIXOTTO, 2006);
- Alocação de registradores por coloração baseada em prioridade (CHOW; HENNESSY, 1990);
- Alocação de registradores por abreviatura para programas *straight-line*(LEE; PALSBERG; PEREIRA, 2008).

A alocação de registradores pode ser vista como uma forma de otimização de código (COOPER; DASGUPTA; ECKHARDT, 2008) por influenciar diretamente no tempo de execução de uma aplicação.

2.6 Compiladores para arquiteturas reconfiguráveis

Nesta seção serão apresentados trabalhos relevantes na área de compilação para arquiteturas reconfiguráveis, estes compiladores tem como objetivo tornar o uso da computação reconfigurável uma realidade para programadores de linguagens de alto nível.

De acordo com Hall (HALL; PADUA; PINGALI, 2009) as linguagens de alto nível e os compiladores podem ser vistos como a base central para o desenvolvimento deste tipo de tecnologia.

2.6.1 Compiladores não comerciais

2.6.1.1 Nenya / Galadriel

Este projeto foi desenvolvido como uma proposta inovadora para a compilação de algoritmos descritos em Java, através de seus *bytecodes*, para *hardware* reconfigurável (CARDOSO, 2000).

No processo de compilação existem dois compiladores que atuam em série para a geração de *hardware* especializado de forma eficiente. A compilação é realizada a partir dos *bytecodes* da linguagem Java, objetivando gerar uma representação intermediária que explore altos graus de paralelismo.

A forma de atuação dos compiladores e algumas técnicas utilizadas pelos mesmos podem ser observadas na Figura 11 e são descritas a seguir.

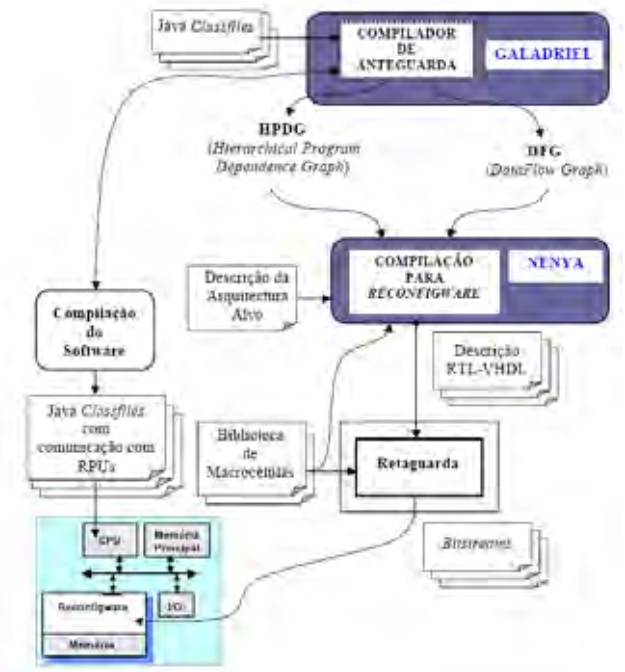

Figura 11: Fluxo dos compiladores Galadriel e Nenya (CARDOSO, 2000).

O Galadriel é o compilador de *front-end* responsável por retirar as informações necessárias do *classfile* fonte e através delas gerar a representação intermediária, que consiste em:

- Grafo de fluxo de controle;
- Grafo de dependência de controle;
- Grafo de dependência de dados;
- Grafo de dependências de fusão;

- Grafo hierárquico de dependências de programa;
- Grafo de fluxo de dados.

O Galadriel não suporta todo o conjunto Java, não permitindo, por exemplo, que o código a ser analisado possua invocação de métodos.

Após a geração da representação intermediária, o Nenya é responsável por executar uma série de transformações nos grafos gerados para otimização do circuito a ser gerado. Estas transformações envolvem:

- Re-associação das operações;
- Redução do custo de operações;
- Aferição do número de *bits* de representação;
- Propagação de padrões de constantes ao nível de *bits*;
- Aferição do número de *bits* em regiões cíclicas.

O código gerado pelo *Nenya* em VHDL é otimizado para uma arquitetura constituída por um FPGA e uma ou mais memórias RAM *(Random Access Memory)* (CARDOSO; NETO, 2001). O *hardware* reconfigurável gerado é formado por duas unidades:

- Unidade de dados: responsável por enviar os dados sobre seu estado à unidade de controle;
- Unidade de controle: responsável pelo controle dos acessos às memórias, pelo controle de escritas em registradores e pela execução correta dos ciclos.

Para a geração da unidade de dados o compilador utiliza uma biblioteca de macrocélulas parametrizáveis. Cada uma destas macrocélulas é responsável pela geração de circuito especializado para a realização de determinada operação (CARDOSO, 2000).

A descrição da unidade de controle é gerada pelo compilador em VHDL-RTL (*Very High Speed Integrated Circuit Hardware Description Language - Register Transfer Level*) comportamental e por isso é necessário utilizar uma ferramenta de síntese lógica para se obter o circuito final.

O Nenya implementa partição temporal com base no algoritmo *simulated annealing* (CARDOSO, 2000). A partição temporal consiste em permitir que um programa que exceda a capacidade do FPGA seja executado. Para esta execução o circuito gerado é divido em partes chamadas de frações temporais, que são executadas individualmente.

2.6.1.2 *Spark*

Spark (GUPTA et al., 2003) é um projeto de um compilador de síntese de alto nível, que tem como objetivo alcançar resultados semelhantes aos obtidos em projetos desenvolvidos manualmente, por especialistas da área. A sua linguagem de entrada é o *ANSI-C*, com algumas restrições como a utilização de ponteiros e funções recursivas.

O *Spark* foi desenvolvido para facilitar o desenvolvimento de aplicações de granulação grossa e fina, aplicando a estas técnicas de otimização de código que podem ter seus efeitos avaliados no código VHDL gerado.

Uma das partes principais deste compilador é a chamada caixa de ferramentas de transformações, que permite ao programador definir os parâmetros para definir as transformações e otimizações a serem aplicadas no código e assim testar diferentes parâmetros para a geração do código VHDL. Dentre as funções da caixa de ferramentas pode-se citar:

- Extrair a dependência de dados;
- Aplicar técnicas para paralelismo de código;
- Aplicar técnicas para renomear variáveis;
- Aplicar otimizações, como por exemplo, propagação de cópias, propagação de constantes e eliminação de código morto.

Para aplicar as transformações no código e armazenar toda a descrição comportamental o *Spark* utiliza como representação intermediária um GHT e um GFCD (grafo de fluxo de controle-dados), que são acessados durante todo o processo de compilação, como pode ser visto na Figura 12.

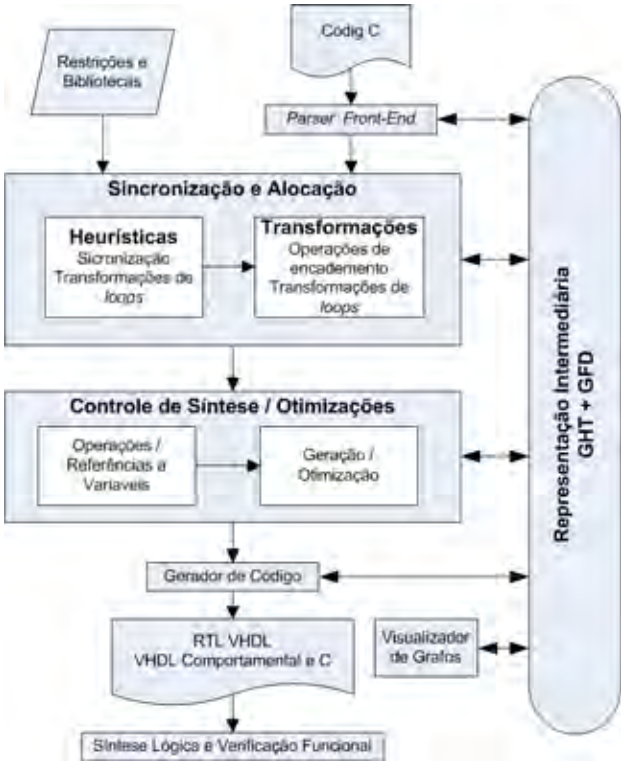

Figura 12: Fluxo de funcionamento do *Spark* (GUPTA et al., 2003).

2.6.1.3 ROCCC (*Riverside Optimizing Configurable Computing Compiler*)

O ROCCC (BUYUKKURT; GUO; NAJJAR, 2006) (GUO et al., 2005) é um compilador para sistemas reconfiguráveis, construído a partir do SUIF (Stanford SUIF Compiler Group, 1994), um *framework* de compilação desenvolvida na universidade de Stanford para a pesquisa colaborativa em técnicas de compilação (HALL et al., 1998) e do Machine SUIF (SMITH; HOLLOWAY, 2008), que é um *framework* para construção de *back-ends* de compiladores, desenvolvido pela Universidade de Harvard, com suporte a várias linguagens de alto nível como: C/C++, Fortran, Java.

Este compilador foi desenvolvido com o objetivo de otimizar aplicações de fluxo in-

tenso de dados, portanto opera melhor com aplicações sem muitos desvios no fluxo de controle. O ROCCC está dividido em dois componentes principais: o *front-end*, responsável por transformações de laços (como por exemplo movimentação de código ciclo-invariante, abertura total ou parcial de laços e fusão de laços), e o *back-end* responsável por otimizações em nível de procedimentos (como por exemplo propagação de constantes, propagação de cópias e eliminação de código morto).

Estas otimizações são aplicadas à representação intermediária gerada, que é denominada CIRRF (*Compiler Intermediate Representation for Reconfigurable Fabrics*), que tem como objetivos:

- Explorar ao máximo o paralelismo existente nos laços;
- Minimizar o acesso à memória, através da reutilização de dados;
- Gerar um *pipeline* eficiente para minimizar a necessidade de ciclos de processamento.

Através da CIRRF o compilador gera o componente VHDL para cada nó do CFG que corresponda a partes da aplicação que serão executadas com maior freqüência. Para as partes de código executadas com menor freqüência ou para as quais o FPGA não é eficiente, são geradas as funções em linguagem C.

O fluxo de funcionamento do ROCCC pode ser visto na Figura 13. O ROCCC apresenta algumas restrições no código de entrada, associadas à restrições ao mapeamento do código para o FPGA, como por exemplo, o uso de funções recursivas e uso de ponteiros.

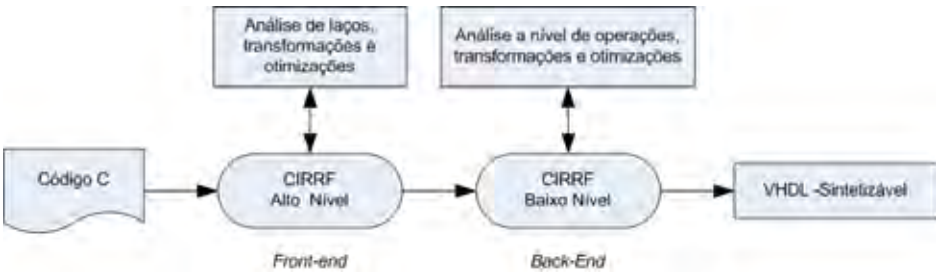

Figura 13: Fluxo de funcionamento do ROCCC (BUYUKKURT; GUO; NAJJAR, 2006).

2.6.1.4 *Trident*

O *Trident* (TRIPP; GOKHALE; PETERSON, 2007) é um *framework* aberto para compilação, que visa a geração de VHDL através de algoritmos descritos em linguagem C. O *Trident* utiliza técnicas de otimização de código tradicional e técnicas voltadas a computação de alto desempenho, como por exemplo verificação de operações que podem ser executadas concorrentemente, geração de circuitos para controle de fluxo de dados entre a memória, registradores e unidades de operação.

Para uso do *Trident*, o programador deve particionar manualmente o programa e escrever o código C responsável pela comunicação entre o *software* e o *hardware*. O trecho de código a ser mapeado para *hardware* contém restrições quanto ao código de entrada, não sendo permitidos comandos de impressão, código recursivo, uso de ponteiros, funções com argumentos ou passagem de parâmetros ou vetores sem tamanho declarado. Durante sua execução, os vetores e variáveis são alocadas de forma estática e todas as operações de ponto flutuante são mapeadas para unidades de *hardware*.

Os passos utilizados na compilação são apresentados na Figura 14 e descritos a seguir:

- Criação da representação intermediária: Esta representação ocorre através do LLVM *(Low Level Virtual Machine)* (LATTNER; ADVE, 2004), que é um *framework* que utiliza o gcc como *front-end*, para a geração de um código-objeto independente de plataforma, conhecido como *LLVM Bytecode*;

- Transformações na representação intermediária: Para cada instrução `if` é criado um hyperbloco[1], para explorar o paralelismo a nível de instruções. Nesta fase também é gerado o GFC, que é utilizado para as otimizações de código e para mapeamento de todas as operações de hardware em módulos selecionados pelo usuário;

- Sincronização: Neste passo é feita a alocação de vetores, esta alocação é feita em tempo de compilação, de forma a se obter um tempo de acesso determinístico à me-

[1] Denominação dada aos blocos de código.

mória e baixa latência no acesso aos dados. Após isso é realizada a sincronização, ou determinação de ordem de execução dos hyperblocos. Para isso, o *Trident* seleciona a melhor opção entre os seguintes algoritmos: *as soon as possible*,*as late as possible*, *force directed* ou *iterative modulo*.

- Síntese: Neste passo é feita a tradução para VHDL-RTL através do GFC.

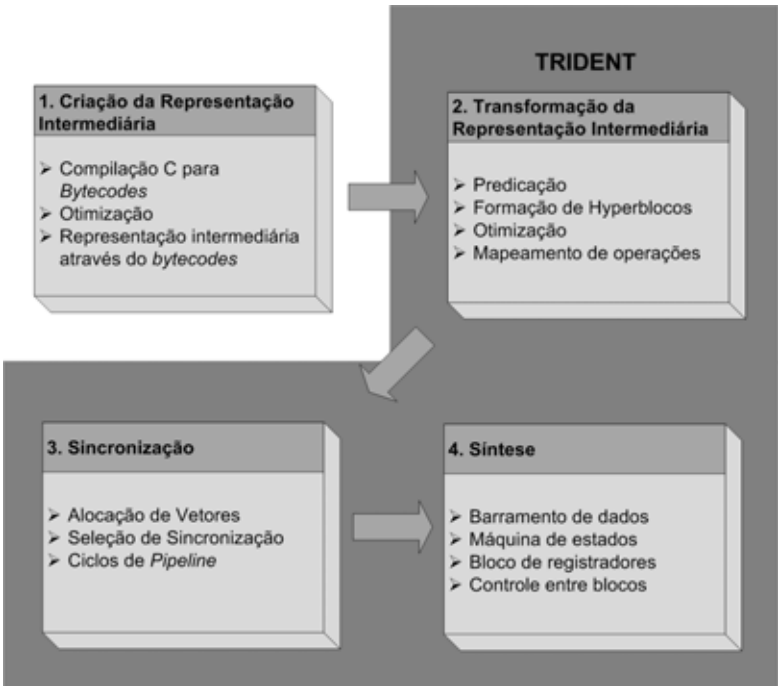

Figura 14: Fluxo de Compilação *Trident* (TRIPP; GOKHALE; PETERSON, 2007).

O *Trident* compartilha seu código e serve de extensão para o *SeaCucumber* (TRIPP; JACKSON; HUTCHINGS, 2002), que é um compilador desenvolvido pela *Brigham Young University*, com objetivo de gerar VHDL através da linguagem Java. O *Trident* fornece a este compilador a capacidade de aceitar código de entrada em C, suportar operações de ponto-flutuante, além de ser o responsável pela geração do código VHDL.

2.6.1.5 Molen

O Molen (PANAINTE; BERTELS; VASSILIADIS, 2007) é um compilador que tem como objetivo gerar código para uma arquitetura específica conhecida como máquina de Molen

(VASSILIADIS et al., 2004), constituída por um processador de uso geral e um FPGA. Este compilador foi desenvolvido para gerar código para uma máquina de Molen formado por um processador IBM *PowerPC* 405 e um FPGA Virtex II Pro.

O Molen foi construído a partir de dois *frameworks*, o SUIF (HALL et al., 1998), utilizado como *front-end*, e o *Machine* SUIF (SMITH; HOLLOWAY, 2008), utilizado como *back-end* e ferramenta para aplicar as otimizações para a geração do código para o FPGA Virtex II. Este código é gerado através de uma representação intermediária própria do *Machine* SUIF.

Para a geração de código para o processador *PowerPC*, foi desenvolvido um *back-end* específico de acordo com o *PowerPC* EABI (SOBEK; BURKE, 1995) (*Embedded Application Binary Interface*), no qual estão definidos os padrões de instruções, alocação de registradores, emulação de operações de ponto-flutuante e outras informações relevantes para o desenvolvimento do código alvo.

O Molen propõe uma nova abordagem para sincronização da configurações de *hardware* necessárias para executar trechos de uma aplicação. Devido à complexidade de algumas aplicações, os circuitos em *hardware* necessários podem ocupar uma área maior que a disponível no FPGA, o que pode gerar conflito ou áreas de *hardware* sobrescritas.

Para evitar este problema foi proposto, com base na eliminação parcial de expressões redundantes (CAI; XUE, 2003), um algoritmo que busca reduzir a área final necessária no FPGA. No caso de não haver espaço suficiente para a implementação dos circuitos de *hardware*, parte da aplicação, que inicialmente seria implementada em *hardware*, pode ser executada em *software*, paralelamente com a execução de outros trechos em *hardware*. Desta forma pode-se assim ganhar desempenho, devido aos altos tempos para a reconfiguração de um FPGA (SIMA et al., 2002).

2.6.1.6 *HThreads*: Modelo de programação *multithreads*

O *HTreads* (ANDREWS et al., 2008) é um compilador para a linguagem C, que utiliza como *front-end* o GCC (*Gnu C Compiler*), que é responsável pela produção de uma forma intermediária de *hardware*.

Através desta representação intermediária é gerado o código VHDL para as *threads* que serão executadas em *hardware*, o código para o compilador alvo e as bibliotecas necessárias para a sincronização e comunicação entre as *threads*.

Para representação intermediária o *HTreads* utiliza o GIMPLE (MERRILL, 2003), que é uma forma de árvore para representação intermediária que visa facilitar a otimização de código.

O fluxo de compilação do *HTreads* é apresentado na Figura 15.

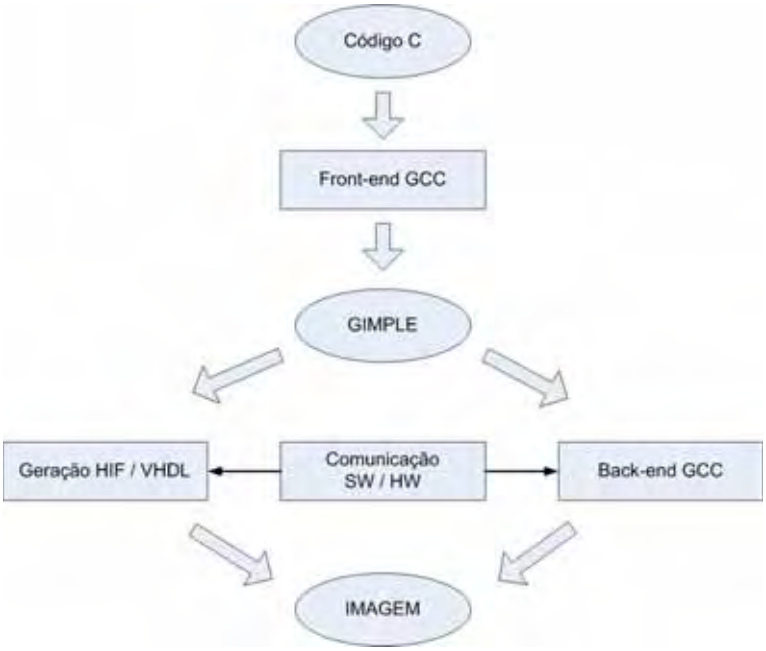

Figura 15: Fluxo de Compilação *HThreads* (ANDREWS et al., 2008).

O *HThreads* utiliza a mesma sintaxe e é totalmente compatível com a biblioteca *pthreads*, permitindo inclusive o teste do código em linguagem de alto nível em um computador executando qualquer distribuição do sistema operacional Linux.

Para o funcionamento correto das *threads*, as funções que seriam desempenhadas pelo sistema operacional foram implementadas via *hardware*, como por exemplo o gerenciamento de *threads*, gerenciamento de semáforos e interrupções da UCP.

2.6.2 Compiladores comerciais

2.6.2.1 C2H - Nios II *C-to-Hardware Acceleration*

O C2H (ALTERA, 2007b) é um compilador para geração de aceleradores em forma de *hardware*, não tendo como objetivo criar código VHDL para novos circuitos, e sim alcançar maior desempenho em aplicações desenvolvidas para executar no processador Nios II.

Os aceleradores são blocos lógicos que têm o papel de executar funções da linguagem C que necessitem de muito processamento. Este blocos são acoplados a um sistema já existente e que possua o processador Nios II. Por estes motivos o C2H assume que: o código C a ser compilado possui instruções que podem ser executadas no processador Nios II e que o resultado da compilação será necessariamente executado em um sistema com processador Nios II.

A estrutura típica de um acelerador ser vista na Figura 16.

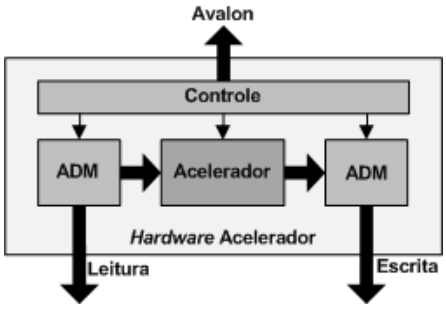

Figura 16: Acelerador de *hardware* típico (ALTERA,).

Os aceleradores gerados possuem as seguintes características:

- Exploração de paralelismo: reconhece pontos que podem ser executados em paralelo;
- Acesso direto a memória: os aceleradores em hardware acessam a mesma memória

que o processador Nios II;

- *Loop pipelining*: *pipeline* em *loop* baseado na latência de acesso a memória e na quantidade de instruções que podem ser executadas em paralelo;

- *Pipeline* de acesso a memória: acesso a memória em forma de *pipeline*, diminuindo assim o efeito da latência de acesso.

O C2H oferece suporte a códigos de entrada em ANSI C, e suporta várias construções da linguagem C, como ponteiros, vetores, *structs*, variáveis globais, chamada a funções e laços. Para acoplar os aceleradores de *hardware* aos sistemas com o processador Nios II, são utilizados o Quartus II e o SOPC Builder, que geram automaticamente a conexão entre o processador e os aceleradores.

Após o processo de compilação o C2H gera um relatório completo descrevendo toda a estrutura de *hardware* gerada, recursos utilizados e o ganho de desempenho apresentado.

2.6.2.2 *EDGE Compiler*

O *EDGE Compiler* (MENTOR, 2009) é uma ferramenta proprietária produzida pela *Mentor Graphics* para desenvolvimento de aplicativos para várias plataformas, entre ela o Nios II.

Este compilador é parte de uma ferramenta de desenvolvimento integrada, *EDGE Developer Suite*, baseada no *Eclipse Framework* (ECLIPSE, 2009). Esta ferramenta utiliza a camada de abstração de *hardware* Altera (HAL - *Hardware Abstraction Layer*) para a configuração do Nucleus RTOS, que é um sistema operacional de tempo real formado por componentes modulares. Esta característica permite ao RTOS se adaptar a qualquer configuração de *hardware* definida no Altera *SOPC Builder*.

O *EDGE Compiler* apresenta as seguinte características:

- suporte as linguagens C, C++ e *Assembler*;

- otimização de código;
- suporte as bibliotecas de entrada e saída da linguagem C;
- extensões para compilação.

2.7 Conclusão

A computação reconfigurável busca uma maior flexibilidade para os modelos de computação existentes, preenchendo uma lacuna entre eles. Também permite explorar de forma eficiente o paralelismo existente em uma aplicação, conseguindo assim alcançar bom desempenho na execução, superando o modelo de execução (busca-decodificação-execução) dos microprocessadores de uso comum, e com um custo menor do que o de desenvolvimento de um processador para uso específico (HAUCK; DEHON, 2007).Estas características tornam a computação reconfigurável uma alternativa para a crescente demanda de processamento em todas as áreas da computação.

Embora os compiladores para sistemas reconfiguráveis não apresentem suporte para todos os recursos das linguagens de alto nível, normalmente por limitações na arquitetura alvo, eles representam uma evolução para síntese de alto nível de circuitos digitais.

As técnicas de compilação para arquiteturas reconfiguráveis embora tenham algumas preocupações diferentes, assemelham-se com as técnicas de compilação para arquiteturas convencionais. Estas técnicas podem ser utilizadas em conjunto no desenvolvimento de um compilador.

3 Phoenix *e Nios II*

3.1 Introdução

Neste capítulo são apresentados o *Framework Phoenix* (DUARTE, 2006) e o processador virtual Nios II (ALTERA, 2006b). O Phoenix é um *framework* de arquitetura aberta para compilador de sistemas reconfiguráveis que objetiva a tradução de código em linguagem C, permitindo a geração de código nativo para o processador Nios II da *Altera Inc.*.

3.2 *Framework Phoenix*

Phoenix é um *framework* para síntese de alto nível de circuitos digitais, voltado para o projeto do *Architect+*, do Laboratório de Computação Reconfigurável do Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (DUARTE, 2006). Este *framework* foi desenvolvido em linguagem C, seguindo o padrão *ANSI*. O fluxograma do *Phoenix* pode ser visto na Figura 17.

Figura 17: Fluxograma do Phoenix (DUARTE, 2006).

Os objetivos deste *framework* são (DUARTE, 2006):

- Permitir compilação eficiente em tempo e espaço de código em linguagem C;
- Permitir geração de uma representação intermediária de uso tanto para geração de código para processadores quanto para o processo de síntese de alto nível de circuitos digitais;
- Gerar código para o processador virtual *Nios* II (ALTERA, 2006b);
- Permitir expansão de recursos.

O *framework* Phoenix segue uma série de fases para a geração do código para o alvo específico, descritas a seguir.

Fase 1:

- Análise sintática e semântica do código fonte, gerando a relação de erros sintáticos ou semânticos para o usuário;
- Geração da árvore sintática das expressões e sua representação em forma instruções de três endereços;
- Geração dos grafos de fluxo de controle e hierarquia de tarefas para cada função.

Fase 2:

- Análise da dependência dos dados das operações;
- Geração dos grafos auxiliares para o processo de geração de código e síntese de alto nível;
  - Grafo de hierarquia de tarefas;
  - Grafo de fluxo de controle;

- Grafo de dependência de controle;
- Grafo de fluxo de dados;
- Grafo de dependência de dados;
- Grafos de dominância e pós-dominância.

Durante esta fase são implementadas as instruções de desvios comuns *continue*, *break*, *goto* e *return*, utilizando-se estruturas de dados auxiliares como por exemplo filas e pilhas, que armazenam informações para que as ligações entre os nós envolvidos possam ser feitas mais tardiamente durante o processo de tradução (DUARTE, 2006);

Fase 3:

- Esta fase, não implementada, é destinada para as otimizações, como por exemplo, eliminação de sub-expressões comuns, eliminação de código morto, propagação de cópias, transposição para constantes, movimentação de código ciclo-invariante e otimização *peephole*(AHO et al., 2007);

Fase 4:

- Fase de geração do código para o alvo específico. O *framework* Phoenix propõe-se a gerar código para o processador virtual Nios II e os testes realizados em (DUARTE, 2006) utilizaram um simulador da arquitetura do processador. Um dos objetivos do *framework* é permitir a geração de código para múltiplos alvos, portanto, esta fase pode ser adaptada de acordo com a utilização do *framework*;
- Permite também a visualização gráfica de todos os grafos gerados utilizando-se o *software Graphviz* (AT&T, 2007).

3.3 Geração da representação intermediária

Esta é uma fase muito importante para o *framework* Phoenix, pois a partir dela deve ser possível a geração de código nativo para qualquer processador ou para o processo de síntese de alto nível. Nesta representação os fluxos de controle do programa original são encapsulados juntamente com suas instruções e dependências em uma série de grafos.

A escolha dos grafos utilizados, segundo Duarte (DUARTE, 2006) deu-se através da análise de vários trabalhos anteriores, relativos a síntese de alto nível e que segundo Aho (AHO et al., 2007), em sua grande maioria são também utilizadas nos algoritmos tradicionais de otimização para geração de código nativo. Os seguintes grafos são gerados como representação intermediária no *Phoenix*.

- Grafo de fluxo de controle;
- Grafo de dependência de controle;
- Grafo de fluxo de dados;
- Grafo de dependência de dados;
- Grafo de hierarquia de tarefas.

3.4 Geração de código no *Phoenix*

O gerador de código consiste em um objeto acoplado à interface de *front-end* do *framework Phoenix*, que recebe a representação intermediária de uma função e gera seu código. Toda a geração do código consiste na tradução das instruções de 3 endereços para as instruções de máquina da arquitetura alvo.

O *framework* foi desenvolvido de forma que o gerador de código possa ser reestruturado para gerar código para vários alvos, a partir da mesma interface de *front-end*, como representado na Figura 18.

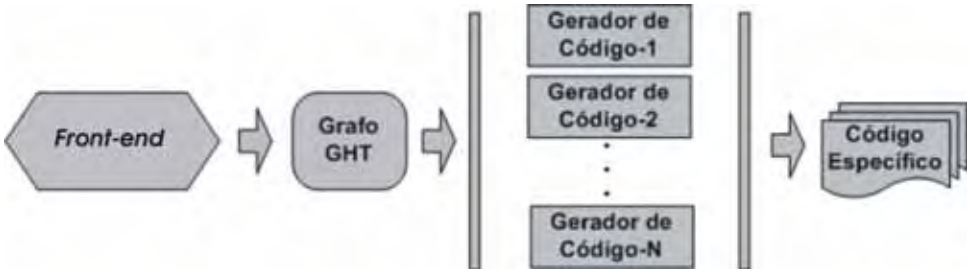

Figura 18: Modelo do Phoenix para a geração de código para múltiplos alvos (DUARTE, 2006).

Para a geração do código faz-se necessária apenas a passagem do grafo de hierarquia de tarefas da função, assim o gerador é invocado para compilar uma função de cada vez.

3.5 Nios II

O Nios II da *Altera Inc.*, é um processador RISC virtual, ou seja, ele é baseado em *software*, fornecido em HDL (*Hardware Description Language)* e não como uma placa de silício, e que pode ser incorporado a qualquer FPGA da familia Altera.

O Nios II, por ser incorporado a um FPGA que é um dispositivo reconfigurável, permite expansão de suas funcionalidades, adição de novos periféricos ou até mesmo criação de periféricos específicos de acordo com a necessidade de cada aplicação o que torna altamente flexível. Alguns exemplos de uso do Nios II são:

- Criptografia para sistemas de tempo real (NAMBIAR; KHALIL-HANI; ZABIDI, 2008);
- Controle de tráfego (ZHAO; ZHENG; LIU, 2009);
- Controladores de potência para sistemas eletrônicos. (ALCALDE; ORTMANN; MUSSA, 2008)

Um modelo de sistema utilizando o Nios II pode ser visto na Figura 19.

O Nios II é um processador RISC de propósito geral, com as seguintes características (ALTERA, 2006b):

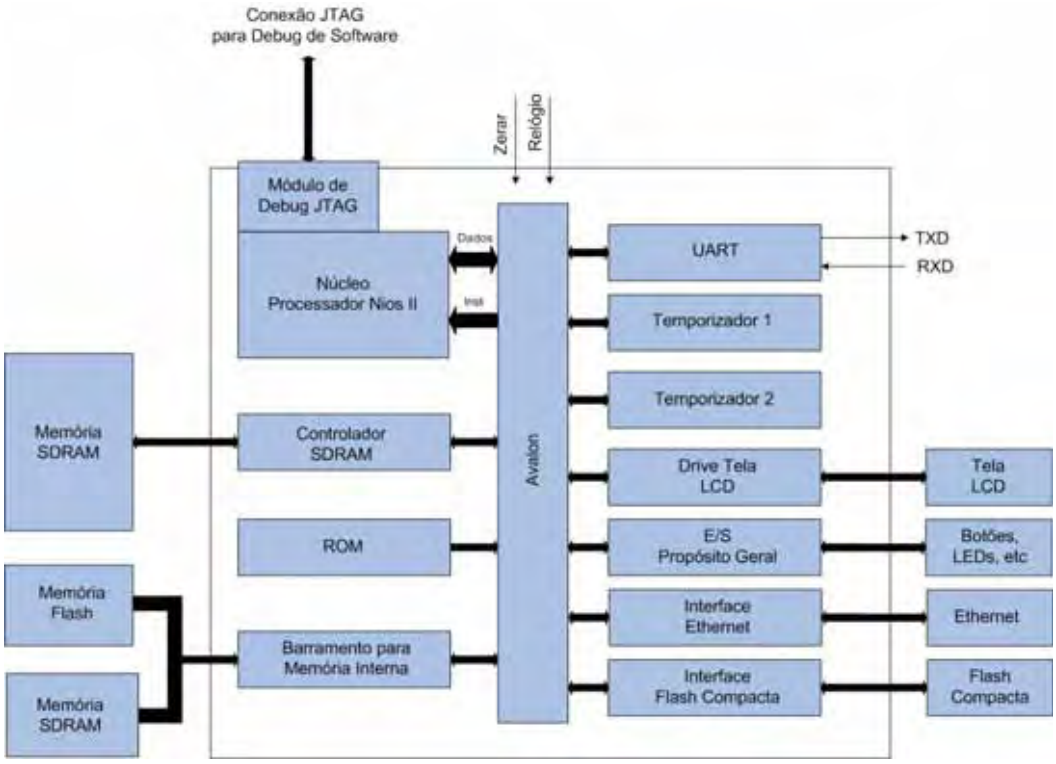

Figura 19: Modelo de sistema com o processador Nios II (ALTERA, 2006b).

- Conjunto de instruções, *datapath* e espaço de endereçamento de 32 bits;
- 32 registradores de propósito geral;
- 32 fontes de interrupção externa;
- Instruções simples para multiplicação e divisão de 32 bits produzindo resultado de 32 bits;
- Instruções dedicadas para multiplicação de 64 e 128 bits;
- Acesso a vários periféricos em *chip* e interfaces para acesso a memórias e interfaces fora do chip;
- Módulo de depuração assistido por *hardware* que permite depuração do processador por uma aplicação de desenvolvimento integrada;
- Arquitetura do conjunto de instruções compatível com todas as versões disponíveis do Nios II;

- Alto desempenho executando em um FPGA.

As 32 instruções do processador Nios II são divididas em 4 categorias, todas elas representadas por palavras de 32 bits, mas que possuem formatos diferentes (ALTERA, 2006b). Estas instruções são apresentadas a seguir.

- Instruções *I-Type*: Este tipo de instrução apresenta em sua palavra 4 campos sendo eles: um campo para OP (Opcode), ou seja o código da instrução a ser executada, com 6 bits, o endereço de 2 registradores (A e B) de 5 bits cada, sendo que A é normalmente um operando e B o registrador de destino, e um campo IMM16 de 16 bits, que é um valor imediato com sinal, exceto quando utilizado para operações lógicas ou comparações sem sinal. Este formato de instrução é apresentado na Figura 20. Neste formato encontram-se instruções de comparação lógica, soma, subtração e similares (ALTERA, 2007a).

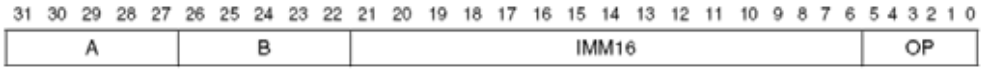

Figura 20: Formato das instruções do tipo *I-Type* (ALTERA, 2007a).

- Instruções *R-Type*: Este tipo de instrução apresenta em sua palavra 5 campos sendo eles: OP com 6 bits, que neste tipo de instruções sempre terá o valor `0X3A`, OPX (*Opcode-Extension*), que definirá a instrução a ser executada, e o endereço de 3 registradores (A, B e C) com 5 bits cada, sendo A e B os operandos e C o registrador de destino. Este formato de instrução é apresentado na Figura 21. Neste formato encontram-se instruções aritméticas e comparação entre registradores (ALTERA, 2007a).

  Este também é o formato apresentado pelas instruções implementadas em *hardware*, que têm sua lógica definidas pelo projetista do circuito, e são adicionadas ao processador na forma de blocos lógicos. Estas instruções são normalmente utilizadas para

acelerar a execução de operações complexas. É possível a criação de até 256 instruções deste tipo. O formato desta instrução pode ser visto na Figura 22, sendo que o campo destinado ao OPX é divido em 4 outros campos, sendo N um número de 8 bits que irá representar a nova instrução, e 3 campos de 1 bit cada (*readra, readrb* e *readrc*) que definirá se os registradores a serem utilizados serão os de propósito geral do processador ou registradores externos (ALTERA, 2006a).

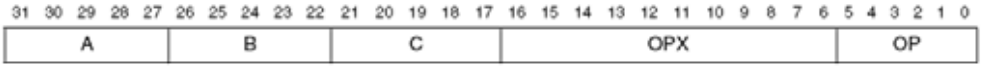

Figura 21: Formato das instruções do tipo *R-Type* (ALTERA, 2007a).

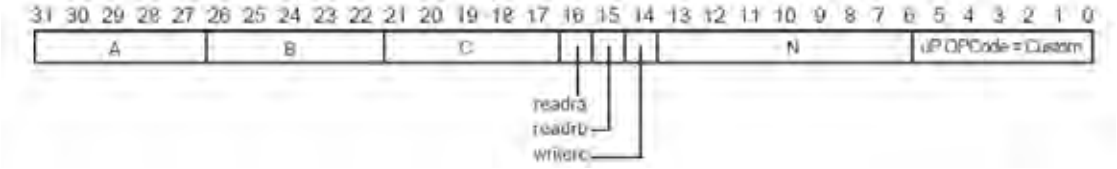

Figura 22: Formato das instruções implementadas em *hardware*. (ALTERA, 2006a).

- Instruções *J-Type*: Este tipo de instrução possui apenas 2 campos, sendo eles: OP com 6 bits, que neste tipo de instrução sempre terá o valor `0X00`, e um campo IMM26, que é o endereço de memória para o qual a execução deve ser desviada, com 26 bits. Este formato de instrução é apresentado na Figura 23. Neste formato tem-se a instrução `call`, que serve para desvio da execução de um programa para determinado endereço de memória (ALTERA, 2007a).

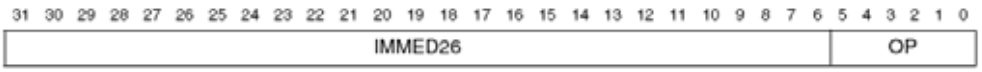

Figura 23: Formato da instrução do tipo *J-Type* (ALTERA, 2007a).

3.6 Conclusão

O Phoenix foi desenvolvido em linguagem C++, utilizando o ambiente de programação Visual C++ da *Microsoft*, e o paradigma de orientação a objetos, visando ser de fácil

extensão e incorporação ao projeto *Architect+* do Laboratório de Computação Reconfigurável do Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo.

O Phoenix tem com um de seus objetivos a geração de código para o Nios II, assim como a geração de circuitos para síntese de alto nível.

O Nios II tem com uma das suas principais características o fato de poder ser configurado de acordo com a aplicação para a qual será utilizado. Para o trabalho descrito nesta dissertação isso será de grande valia por permitir a adequação do processador ao suporte aos tipo de dados de ponto flutuante *Float* e *Double*.

4 *N-Compiler*

4.1 Introdução

Este capítulo apresenta o desenvolvimento de um compilador, denominado *N-compiler* (SILVA; LOBATO; ULSON, 2008), que utiliza como base para o *front-end* o *framework* Phoenix, e tem como objetivo a geração de código para o processador virtual Nios II (ALTERA, 2006b), apresentado no capítulo 3. Durante este capítulo serão apresentadas as características do compilador e do seu desenvolvimento, além de uma comparação com outros compiladores estudados.

4.2 Desenvolvimento do *N-Compiler*

O desenvolvimento baseou-se no uso do *framework* Phoenix, pois o mesmo tem como objetivo ser parte do projeto do *Architect+* (DUARTE, 2006), projeto ao qual pretende-se integrar o *N-Compiler*. Com a opção pelo uso do *framework* não houve a necessidade de desenvolver um *front-end* completo, fez-se necessário apenas adaptá-lo de acordo com os objetivos do trabalho, com isso foi possível dar maior ênfase ao *back-end*.

Durante o trabalho foram desenvolvidos os seguintes módulos:

- Desenvolvimento de um módulo responsável pela otimização do código;
- Desenvolvimento de um gerador de código para o processador virtual Nios II;
- Desenvolvimento de um módulo em *hardware* para manipulação de tipos de ponto-flutuante.

Além das necessidades de implementação relacionadas ao compilador, surgiu também a necessidade de desenvolvimento de um simulador que permitisse o teste do código gerado e a validação do compilador.

Com a implementação deste módulos o fluxo de compilação do *N-Compiler* ficou como apresentado na Figura 24, este fluxo de compilação foi adaptado do proposto no *framework* Phoenix (DUARTE, 2006) que foi apresentado na Figura 17. As partes preenchidas em cinza e circuladas pela linha tracejada na Figura 24 foram desenvolvidas durante este trabalho. O fluxo de compilação é semelhante ao fluxo encontrado nos compiladores estudados durante o desenvolvimento do trabalho. Porém, o aspecto que diferencia o fluxograma apresentado é o desenvolvimento de um módulo externo para manipulação de dados de ponto-flutuante. O desenvolvimento deste módulo será apresentado posteriormente.

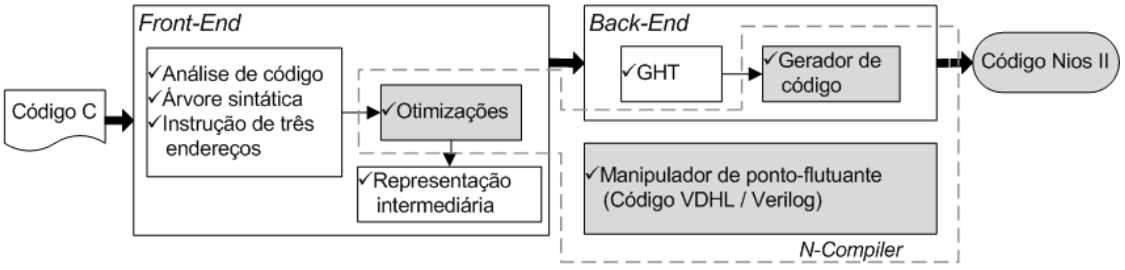

Figura 24: Fluxo de compilação do *N-Compiler*, adaptado de (DUARTE, 2006).

O *N-Compiler* foi desenvolvido em linguagem C++, pois foi a linguagem utilizada para o desenvolvimento do *framework* Phoenix. O projeto foi todo desenvolvido utilizando o ambiente de programação Visual Studio da Microsoft. Nas próximas seções deste capítulo serão descritos os passos seguidos durante o desenvolvimento do compilador e de um simulador para possibilitar seus testes.

4.2.1 Especificação do otimizador e gerador de código implementado

A fase de análise do sistema ocorreu através da definição do diagrama de classes, do diagrama de caso de uso e diagrama de componentes, visando documentar o que foi

desenvolvido no escopo deste trabalho.

Esta documentação visa facilitar futuras alterações do compilador desenvolvido.

Diagrama de caso de uso

Os diagramas de caso de uso buscam representar a iteração de um ator com uma atividade realizada pelo sistema. Um ator pode ser uma pessoa, um sistema, ou qualquer outra entidade que possa interagir com o sistema que está sendo modelado (MEDEIROS, 2004).

Os diagramas de caso de uso apresentados abordam o desenvolvimento descrito neste trabalho, não apresentando os casos de uso do *framework* Phoenix. A seguir são mostrados os diagramas de caso de uso e suas descrições.

Na Figura 25 pode ser visto o diagrama de caso de uso **Otimizar instruções de três endereços**. A descrição deste diagrama pode ser vista na Tabela 1 na qual é apresentada uma explicação sobre as funcionalidades do diagrama.

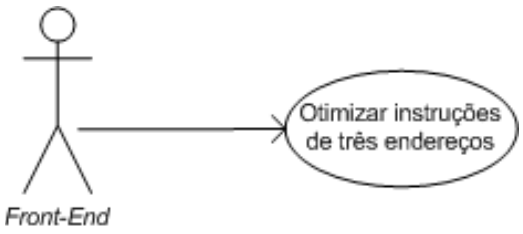

Figura 25: Diagrama de caso de uso Otimizar instruções de três endereços.

Na Figura 26 pode ser visto o diagrama de caso de uso **Gerar código**. A descrição deste diagrama pode ser vista na Tabela 2, na qual é apresentada uma explicação sobre as funcionalidades do diagrama.

Tabela 1: Descrição do caso de uso **Otimizar instruções de três endereços**.

UC001: Otimizar instruções de três endereços
Breve descritivo: Recebe o conjunto de instruções de três endereços gerado pelo *front-end* e submete a execução dos algoritmos de otimização.
Pré-Condições: *Front-end* gerar as instruções de três endereços
Ator: *Front-end*
Cenário Principal:
1 - Recebe instruções de três endereços geradas;
2 - Executa algoritmos de otimização;
3 - Gera conjunto de instruções de três endereços otimizadas.

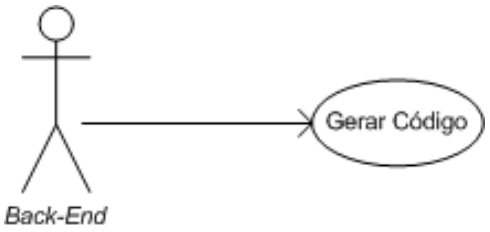

Figura 26: Diagrama de caso de uso Gerar código.

Tabela 2: Descrição do caso de uso **Gerar código**.

UC002: Gerar código
Breve descritivo: Recebe o GHT gerado pelo *front-end* e gera o código para o processador Nios II.
Pré-Condições: *Front-end* ter gerado corretamente o GHT
Ator: *Back-end*
Cenário Principal:
1 - Recebe o GHT;
2 - Lê instruções de três endereços contidas no GHT;
3 - Verifica tipo de instrução de três endereços lida;
3 - Gera o código de maquina referente à instrução de três endereços lida

Diagrama de classes

Este diagrama visa mostrar as classes necessárias para a implementação de um sistema (MEDEIROS, 2004).

Devido às alterações realizadas e à inclusão dos novos módulos ao *framework* Phoenix, foi necessário alterar o diagrama de classes original, encontrado em (DUARTE, 2006), para representar o estado atual do projeto desenvolvido. Este novo diagrama pode ser visto na Figura 27. As classes preenchidas em cinza foram desenvolvidas no decorrer deste

trabalho e terão seu desenvolvimento explicado a seguir. As demais classes encontram-se implementadas no *framework* Phoenix.

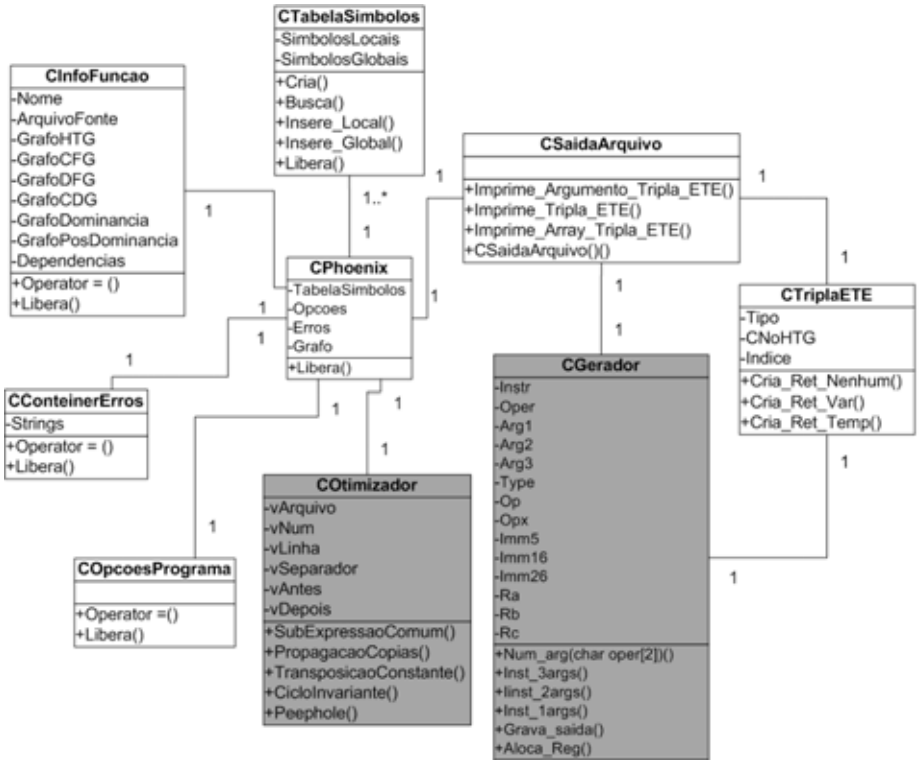

Figura 27: Diagrama de classes do N-Compiler, adaptado de (DUARTE, 2006).

Diagrama de componentes

Na Figura 28 é apresentado o diagrama de componentes referente ao *N-Compiler*, que mostra as partes necessárias para o desenvolvimento de um *software* (MEDEIROS, 2004).

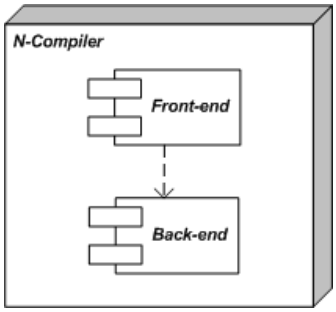

Figura 28: Diagrama de componentes do N-Compiler.

4.2.2 Adaptações implementadas no *Framework* Phoenix

Durante o desenvolvimento fez-se necessário implementar adaptações no *framework* de modo a possibilitar o desenvolvimento do trabalho, além de ajustes que visaram permitir um melhor funcionamento do compilador desenvolvido.

- Criação de um novo projeto no Visual Studio e importação de código fonte do *framework*: Para dar continuidade ao desenvolvimento do *framework* um novo projeto foi criado, mantendo assim uma cópia do código original;

- Alteração na forma de execução: O Phoenix exigia a passagem do nome do arquivo com o código fonte e também alguns outros parâmetro que definiam se seriam geradas as imagens dos grafos, arquivos textos com as instruções de três endereços. Como o objetivo do N-Compiler é gerar sempre o código para o Nios II, esta obrigatoriedade de passagem de parâmetros de compilação foi retirada, alterando-se as classes COpcoesPrograma e CPhoenix. Desta forma, para a compilação basta executar o aplicativo passando como único parâmetro o arquivo com o código fonte;

- Redefinição de métodos para representação gráfica do grafos: A geração da representação gráfica foi alterada devido a uma redefinição feita nos métodos de acesso aos valores armazenados na classe CTriplaETE. Esta redefinição teve como objetivo facilitar o acesso aos dados;

- Redefinição de métodos para geração das instruções de três endereços: A geração da representação intermediária foi alterada devido a uma redefinição feita nos métodos de acesso aos valores armazenados na classe CSaidaArquivo. Esta redefinição teve como objetivo facilitar o acesso aos dados;

- Redefinição do diagrama de classes do *framework* com a inclusão de novas classes: Algumas classes encontradas no *framework* não foram utilizadas, sendo necessário redefinir o diagrama de classes original, e também incluir as novas classes desenvolvidas;

- Desenvolvimento dos módulos necessários ao compilador:
  - Otimizador de Código: responsável por receber as instruções de três endereços e executar a otimização;
  - Gerador de Código: responsável por receber o GHT e gerar o código para a arquitetura alvo.

4.2.3 Implementação do tipos *Float* e *Double*

O processador virtual Nios II não possui, em sua configuração original, nenhuma forma para a manipulação de tipos de dados de ponto flutuante. Diante deste cenário fez-se necessário definir uma forma de implementar estes tipos, para permitir que uma gama maior de aplicações pudessem ser desenvolvidas para este processador.

Durante o desenvolvimento do trabalho foi feito um estudo para se definir a melhor forma para a implementação deste tipo de instruções. Neste estudo foram testadas duas formas para a realização destas implementações, através do uso de mantissa e expoente e a implementação em *hardware* permitida no Nios II.

No formato mantissa e expoente o campo de expoente corresponde à soma de 128 com o expoente de base 2 do número representado e o campo de mantissa corresponde à parte fracionária da mantissa do número representado. Embora este tipo de implementação seja muito utilizado (como por exemplo em ferramentas como o Matlab), nos testes realizados durante a fase de desenvolvimento esta forma de implementação apresentou baixo desempenho por exigir grande número de acessos à memória. Esta implementação também aumentou significativamente o número de registradores necessários, por necessitar de 2 registradores para armazenar um único número de ponto flutuante. Isso mostrou-se um grande obstáculo para o seu uso, devido ao número reduzido de registradores existentes no Nios II.

A implementação em forma de *hardware* mostrou-se uma alternativa mais viável, por apresentar melhor desempenho em relação à implementação usando mantissa e expoente,

além da facilidade para implementação. Como possíveis obstáculos para este tipo de implementação pode-se citar o fato de ser necessário conhecer a ferramenta Quartus II e estar familiarizado com o processo de síntese do código VHDL ou Verilog para um FPGA. A inclusão do suporte a ponto flutuante em forma de *hardware* no Nios II deve ser prevista no projeto do *hardware*, para que ofereça suporte para todas as instruções utilizadas no compilador.

Como a implementação se deu na forma de *hardware*, o módulo para manipulação de ponto flutuante foi criado fora do compilador, portanto este módulo deve ser acoplado à configuração em VHDL ou Verilog, necessária para a instalação do processador Nios II no FPGA.

A unidade de ponto flutuante para o processador Nios II foi implementada na forma de blocos lógicos que funcionam em conjunto com a ULA (Unidade de Lógica e Aritmética) no processador. O esquema de funcionamento desta unidade lógica pode ser visto na Figura 29.

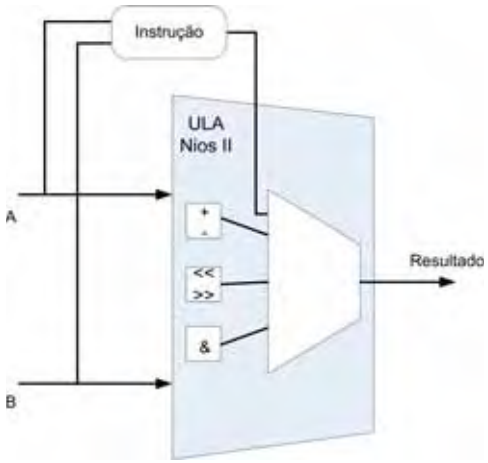

Figura 29: Instrução em forma de *hardware* conectada a ULA do processador Nios II (ALTERA, 2006a).

Esta implementação foi mais rápida quando comparado com a implementação dos tipos de dados de tipo flutuante via *software*. A Tabela 3 apresenta o fator de aceleração apresentado pelo fabricante, na implementação em *hardware* quando comparado com a implementação em *software* em dois modelos de FPGAs diferentes.

Tabela 3: Fator de aceleração com o uso de instruções para ponto flutuante implementadas em *hardware* (ALTERA, 2006a).

FPGA	**Soma**	**Subtração**	**Multiplicação**	**Divisão**
EP2S60	14x	15x	12x	14x
EP1S40	20x	20x	19x	18x

O desempenho deste tipo de implementação pode variar de acordo com o FPGA utilizado, como pode ser visto na Tabela 3. Isso ocorre pois o desempenho está relacionado ao número de LUTs (*Look-Up Table*) ou elementos lógicos que cada FPGA possui, sendo maior em FGPAs com número maior de LUTs ou elementos lógicos (ALTERA, 2006a).

Assim como as instruções definidas ou implementadas no processador Nios II, as instruções implementadas em *hardware* possuem um `opcode` para identificá-las, estes valores podem ser encontrados na documentação do fabricante. Na Tabela 4 são apresentados os valores de `opcode` para cada uma das operações básicas.

Tabela 4: Opcodes para instruções implementadas em *hardware* (ALTERA, 2008a).

	Multiplicação	**Soma**	**Subtração**	**Divisão**
Opcode	252	253	254	255

Toda a implementação deste módulo de *hardware* foi realizada através da ferramenta Quartus II da Altera. Esta ferramenta permite criar de forma gráfica todos a arquitetura de um sistema e também faz a síntese do circuito digital no FPGA. Dentro da ferramenta já existe um módulo para manipulação para ponto flutuante desenvolvido, o mesmo deve ser incluído a um projeto em VHDL ou Verilog.

A implementação destes tipos de dados no processador Nios II segue o padrão IEEE 754 (ANSI/IEEE, 1985), que é uma recomendação dos institutos ANSI (*American National Standard Institute*) e IEEE (*Institute of Electrical and Eletronic Engineers*), e faz referência às normas que devem ser seguidas para a utilização da aritmética binária para números de ponto flutuante.

4.2.4 Algoritmos de otimização

Este módulo tem como objetivo permitir ao compilador gerar código otimizado. Para isso, são aplicados algoritmos de otimização que têm como alvo as instruções de três endereços geradas. Uma descrição do funcionamento dos algoritmos implementados pode ser encontrada no capítulo 2 deste trabalho.

A implementação dos algoritmos de otimização foi realizada com base em (AHO et al., 2007). Foram realizados testes somente para verificação de funcionamento dos algoritmos e não para verificação da eficiência das otimizações, visto que como citado em Aho (AHO et al., 2007), não se pode garantir que estas otimizações tragam melhorias ao código, por terem grande dependência do código de entrada. Por este motivo, em alguns padrões de código de entrada pode-se ter um grande ganho e para outros padrões pode-se não ter ganho significativo, ou até nenhum ganho. Na Figura 30 pode ser visto um exemplo de código (a) que apresenta ganho quando submetido aos algoritmos de otimização e também um código (b) que não apresentaria ganho por ter sido desenvolvido de forma otimizada.

```
1   int main( )
2   {
3     int A, B, C;
4     B = 10;
5     A = 15;
6     C=0;
7     while (C < B*2)
8     {
9           C++;
10          if (C > A)
11                A = pow(C,2);
12    }
13    return 0;
14  }
```

(a)

```
1   int main( )
2   {
3     int A, C;
4     A = 15;
5     C=0;
6     while (C < 20)
7     {
8           C++;
9           if (C > A)
10                A = C*C;
11    }
12    return 0;
13  }
```

(b)

Figura 30: Código que necessita de otimização (a) e código desenvolvido de forma otimizada (b).

Para a implementação foi criada uma classe chamada COtimizador, que pode ser vista na Figura 31. Esta classe realiza a leitura das instruções de três endereços armazenadas em um vetor, e aplica os algoritmos de otimização de forma seqüencial e armazena as instruções em um vetor temporário até que sejam aplicadas as otimizações.

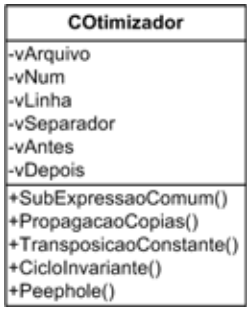

Figura 31: Classe Otimizador.

Para efeito de implementação e acompanhamento da execução das otimizações, os algoritmos foram desenvolvidos na forma de uma ferramenta gráfica, que pode ser vista na Figura 32, na qual pode-se acompanhar passo a passo toda a execução dos algoritmos, visando verificar o correto funcionamento e também o ganho alcançado. Para o teste de funcionamento dos algoritmos foi passado para a ferramenta um arquivo texto contendo um conjunto de instruções de três endereços. Através dos testes realizados pode-se verificar que o resultado é quase sempre satisfatório no que diz respeito à otimização de código.

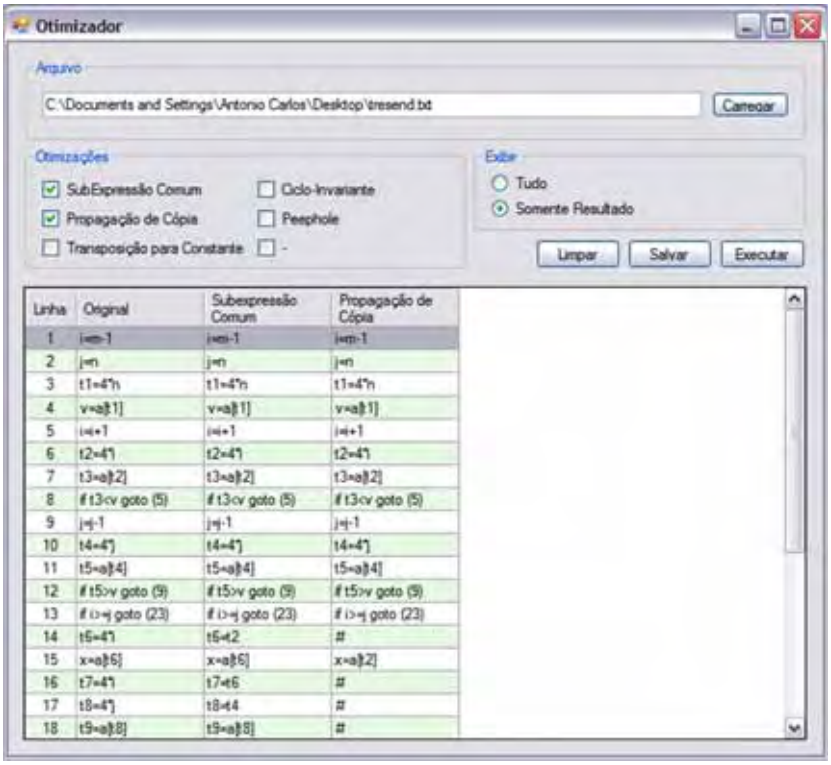

Figura 32: Interface gráfica da ferramenta de otimização de código.

Os algoritmos implementados nesta ferramenta foram posteriormente migrados para o *framework* Phoenix, agora sem o uso da interface gráfica. Durante esta migração foram feitas as adaptações necessárias para a leitura e armazenamento dos dados de acordo com o padrão utilizado no *framework* Phoenix.

Durante a compilação, logo após a geração das instruções de três endereços são executados os algoritmos de otimização na seguinte ordem:

- Eliminação de sub-expressões comuns;
- Propagação de cópias;
- Tranposição para constantes;
- Movimentação de código ciclo-invariante;
- Otimização *Peephole*:
  - Eliminação de instruções redundantes;
  - Otimização do fluxo de controle;
  - Simplificações algébricas.

Estes algoritmos são aplicados a todos os programas submetidos ao compilador, buscando-se sempre alcançar o melhor resultado possível.

4.2.5 Gerador de Instruções para o Nios II

O gerador existente na versão do *framework* Phoenix utilizada foi descartado pois não se adequava às necessidades do *N-Compiler*. Por este motivo foi desenvolvido este módulo que tem como objetivo gerar código para a máquina alvo.

Este módulo foi desenvolvido na forma de uma classe chamada CGerador, que pode ser vista na Figura 33. Esta classe foi adicionada ao código fonte do *framework*.

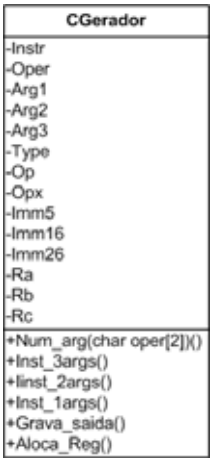

Figura 33: Classe Gerador.

Para a geração do código esta classe recebe as informações representadas no GHT, gerado pelo *front-end*. Através do GHT é possível ter acesso as instruções de três endereços necessárias para a tradução do código. Estas instruções encontram-se armazenadas na classe CTriplaETE do *framework*.

Dentro do *framework* existem métodos, que foram alterados durante o desenvolvimento do trabalho e que servem para a listagem destas instruções. Estes métodos são apresentados na Tabela 5.

Tabela 5: Forma de acesso a instruções de três endereços.

Linha	**Forma de acesso**	**Elemento acessado**
1	TriplaETE->Op->Token.CString()	Operador
2	TriplaETE->No->Token.CString()	Resultado ou argumentos não numéricos
3	TriplaETE->Args[x].No->Token.CString()	Argumentos numéricos
4	TriplaETE->ID	Identifica variável virtual

Foram feitas algumas alterações na forma de chamada de alguns métodos, dentro do *framework*, visando facilitar o acesso aos dados contidos no vetor TriplaETE.

Dentro do *framework* as instruções foram identificadas como Unárias ou Binárias. Na Tabela 6 é exemplificado cada um dos tipos de instruções definidos no *framework*. Esta definição permite saber o que deve ser acessado para a geração do código para cada tipo de instrução, isso é importante pois permite economia de tempo, no que diz respeito ao

número de operandos de cada uma das operações.

Tabela 6: Exemplo de instruções Unárias e Binárias.

Tipo de operação	**Exemplo**
Operação unária	A ++
Operação binária	A = B + C

Cada uma das operações encontradas no campo operador deve ser codificada de acordo com o tipo de instrução a qual se enquadra, com base nisso foi necessário listar um grupo de instruções, como pode ser visto nas Tabelas 7, 8 e 9, que foram implementadas no *N-Compiler*, com as particularidades de cada uma delas.

Na Tabela 7 são apresentadas as instruções, implementadas no *N-Compiler*, que utilizam um valor imediato na formação de sua palavra, seus *opcodes*, descrição de sua função e a sintaxe da instrução em *assembler*.

Tabela 7: Instruções *I-Type* implementadas no N-Compiler.

Instruções *I-Type* implementadas			
Instruções	**OP**	**Descrição**	**Sintaxe (Assembler)**
addi	0x04	Adição com uso de valor imediato	addi rB, rA, IMM16
cmpgei	0x08	Comparação maior igual com uso de valor imediato	cmpgei rB, rA, IMM16
cmplti	0x10	Comparação menor igual com uso de valor imediato	cmplti rB, rA, IMM16
andi	0x0c	Função E com uso de valor imediato	andi rB, rA, IMM16
bge	0x0E	Salto se maior ou igual	bge rA, rB, label
blt	0x16	Salto se menor ou igual	blt rA, rB, label
cmpnei	0x18	Comparação não igual com uso de valor imediato	cmpnei rB, rA, IMM16
bne	0x1E	Salto se não igual	bne rA, rB, label
muli	0x24	Multiplicação com uso de valor imediato	muli rB, rA, IMM16
beq	0x26	Salto se igual	beq rA, rB, label
bgeu	0x2E	Salto se maior ou igual sem sinal	bgeu rA, rB, label

Na tabela 8 são apresentadas as instruções, implementadas no compilador, que utilizam somente registradores na montagem de sua palavra, seus opcodes, descrição de sua função e sua sintaxe em *assembler*.

Para permitir a definição de um número menor de instruções, o Nios II implementa um conjunto de pseudo-instruções. Estas instruções são implementadas através do uso de algumas das instruções citadas anteriormente, ao invés de terem uma implementação es-

Tabela 8: Instruções *R-Type* implementadas no N-Compiler.

Instruções *R-Type* implementadas			
Instruções	**OPX**	**Descrição**	**Sintaxe (Assembler)**
0x08	cmpge	Comparação maior ou igual	cmpeg rC, rA, rB
0x0D	jmp	Salto	jmp rA
0x0E	and	Operação E	and rC, rA, rB
0x10	cmplt	Comparação menor que	cmplt rC, rA, rB
0x16	or	Operaçao OU	or rC, rA, rB
0x18	cmpne	Comparação não igual	cmpne rC, rA, rB
0x24	divu	Divisão sem sinal	divu rC, rA, rB
0x25	div	Divisão	div rC, rA, rB
0x27	mul	Multiplicação	mul rC, rA, rB
0x28	cmpgeu	Comparação maior ou igual sem sinal	cmpgeu rC, rA, rB
0x20	cmpeq	Comparação igual	cmpeq rC, rA, rB
0x31	add	Soma	add rC, rA, rB
0x39	sub	Subtração	sub rC, rA, rB
0x30	cmpltu	Comparação menor que sem sinal	cmpltu rC, rA, rB

pecífica. A Tabela 9 apresenta o conjunto destas instruções implementadas no compilador, suas descrições e suas sintaxes em *assembler*.

Tabela 9: Pseudoinstruções implementadas no N-Compiler.

Pseudo-instruções implementadas			
Instruções	**Descrição**	**Sintaxe (Assembler)**	**Sintaxe Implementação**
mov	Movimentação	mov rC, rA	add rB, rA, IMM16
movi	Movimentação com uso de valor imediato	moi rC, rA	addi rB, rA, IMM16
cmpgt	Comparação maior que	cmpgt rC, rA, rB	cmplt rC, rA, rB
cmpgti	Comparação maior que com uso de valor imediato	cmpgti rB, rA, IMMED	cmpgei rB, rA, IMM16+1
cmple	Comparação menor ou igual	cmple rC, rA, rB	cmpeg rC, rA, rB
cmplei	Comparação menor ou igual com uso de valor imediato	cmplei rB, rA, IMMED	cmplti rB, rA, IMM16+1

Durante o desenvolvimento do trabalho pode-se verificar que o processador Nios II utiliza palavras no formato hexadecimal e que são escritas no formato *little-endian* (HENNESSY; PATTERSON, 2003). Neste formato o byte mais significativo é o primeiro a ser armazenado, começando a ser armazenado no bit 0, o segundo mais significativo começará a ser armazenado no bit 8 e assim sucessivamente até o armazenamento de toda a palavra. Um exemplo de uma palavra armazenada neste formato pode ser visto na Figura 34.

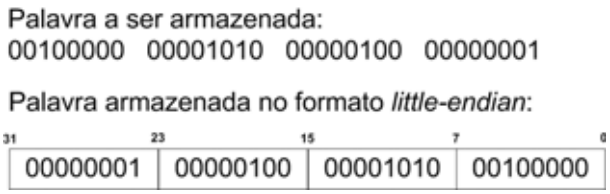

Figura 34: Palavra armazenada no formato *little-endian*.

A implementação da palavra neste formato e também outras particularidades como o preenchimento de campos em alguns tipos de instruções que reservam 11 bits para armazenar o opcode de 6 bits, acabaram exigindo um grande tempo de estudo sobre as instruções e sobre a forma correta de escreve-las, já que não foram encontradas estas informações na documentação do fabricante.

Após as definições de como acessar cada um dos dados necessários e do tipo de palavra a ser escrita para a geração do código também faz-se necessária a definição dos registradores a serem utilizados na geração das instruções. Isso ocorre pois não está previsto o uso de memória devido a independência da arquitetura, que se espera obter no N-Compiler.

Para resolver o problema de alocação de registradores foi desenvolvida uma descrição da arquitetura do processador Nios II que pode ser vista na Tabela 10. Foram considerados na implementação os registradores contidos entre `r2` e `r23`. Além destes, o registrador `r0` é utilizado para representar o valor zero.

Segundo Aho (AHO et al., 2007), embora a alocação de registradores seja muito importante para a geração de um bom código, ela é um problema difícil de ser resolvido. Para resolução deste problema alguns algoritmos foram propostos, como por exemplo, alocação de registradores através de coloração de grafos (BRIGGS et al., 2004) ou baseada em prioridade (CHOW; HENNESSY, 1990).

Para a implementação do gerador de código optou-se por uma abordagem simples para a alocação de registradores, baseada no algoritmo *Least Recently Used* (NOH et al., 1997). Nesta abordagem os registradores serão alocados de forma seqüencial até que estejam esgotados. A partir deste momento começa a ser verificado qual o registrador está a mais

Tabela 10: Conjunto de registradores do Nios II (ALTERA, 2007a).

Reg.	**Nome**	**Função**		**Reg.**	**Nome**	**Função**
r0	Zero	`0x0000000`		r16		Propósito Geral
r1	at	Assembler temporário		r17		Propósito Geral
r2		Valor de retorno		r18		Propósito Geral
r3		Valor de retorno		r19		Propósito Geral
r4		Argumentos		r20		Propósito Geral
r5		Argumentos		r21		Propósito Geral
r6		Argumentos		r22		Propósito Geral
r7		Argumentos		r23		Propósito Geral
r8		Propósito Geral		r24	et	Exceção temporária
r9		Propósito Geral		r25	bt	*Breakpoint* temporário
r10		Propósito Geral		r26	gp	Ponteiro global
r11		Propósito Geral		r27	sp	Apontador de pilha
r12		Propósito Geral		r28	fp	Ponteiro de quadro [1]
r13		Propósito Geral		r29	ea	End. de retorno de exceções
r14		Propósito Geral		r30	ba	End. de retorno de *breakpoints*
r15		Propósito Geral		r31	ra	End. de retorno

[1]Este registrador contém a cópia do valor de `sp` e pode ser utilizado como um registrador temporário.

tempo sem ser acessado, para que o mesmo possa ser desalocado e um novo valor seja armazenado. O valor desalocado é armazenado em uma estrutura para ser reutilizado caso seja necessário.

Este controle é feito através de uma estrutura de dados que armazena os registradores em uso e um valor que define o número de vezes em que valores foram alocados em algum registrador. Sempre que um registrador for utilizado, seja somente para leitura ou para alteração de seu valor, o campo destinado ao controle do número de acessos é zerado, indicando que ele foi utilizado recentemente.

Para evitar a necessidade de realizar a configuração da placa de *hardware*, através de código VHDL ou Verilog, para realizar o teste de aplicativos escritos para o processador Nios II, construiu-se um simulador do conjunto de instruções, capaz de ler instruções no formato exigido pelo processador e emular os resultados.O simulador desenvolvido tem como objetivo permitir a rotina de testes que necessitavam ser realizados para validação do gerador de código desenvolvido no compilador, e é descrito na próxima seção.

4.3 Simulador de execução de instruções no Nios II

Para permitir o teste somente do código gerado foi desenvolvido um simulador do conjunto de instruções do Nios II (LIMA et al., 2008), que tem como objetivo ler um conjunto de instruções no formato definido na documentação do processador (ALTERA, 2007a).

Na Figura 35 pode ser visto um trecho do código do simulador desenvolvido em ANSI C (SCHILDT, 1997). Este trecho de código é o responsável por verificar se o PC aponta para uma instrução ou para o final do programa (linhas 4, 5 e 6). Caso o PC aponte para uma instrução, faz-se a leitura do *opcode* contido na palavra (linhas 9 e 10), após isso é feita a verificação dos parâmetros (linhas 12, 13 ou 14), caso estejam corretos é feita a validação dos registradores acessados pela instrução, de forma a garantir que ela não acesse nenhum dos registradores reservados, além disso verifica-se se a instrução é válida, ou seja pode ser decodificada pelo processador Nios II.

```
1  printf("Starting simulation...\nState of registers:\n");
2    /* leitura das palavras contendo as instruções*/
3    pc = 1;
4    while ( pc <= sizef ){
5       if ( pc + 3 > sizef ) return 4;   /* estourou a RAM*/
6       word = get_word( pc );   /* obtém a palavra (PC - PC+3)*/
7
8       /* determinar o tipo de instrução: obtendo apenas os 6 primeiros bits*/
9       op = word << 26;
10      op = op >> 26;
11
12      if ( op == 0x3A ) err = int_rtype( word, &pc );
13      else if ( op == 0x00 ) err = int_jtype( word, &pc );
14           else err = int_itype( word, &pc );
15
16      if ( err == 2 ){
17         printf("ERROR: access to private register.\n");
18         return 2;
19      }else if ( err == 3 ){
20               printf("ERROR: couldn't decode instruction.\n");
21               return 3;
22            }
```

Figura 35: Trecho de código do Simulador.

O simulador recebe como entrada um arquivo com um conjunto de palavras que representam instruções do Nios II, este arquivo de entrada é passado como parâmetro no

momento da execução do simulador. Após a leitura do arquivo, o simulador inicia a execução das instruções. Ao final da simulação é apresentado o estado final dos registradores, ou então o estado dos registradores após a execução de cada uma das instruções além do tempo gasto na execução, os valores do PC (*Program Counter*), o número e o tipo das instruções executadas, além do estado da memória simulada que é exibida através da passagem do parâmetro opcional `-m`. O formato de saída completa do simulador pode ser visto na Figura 36.

```
Starting simulation (file size 16) ...

Program Counter:
pc = 1.
pc = 5.
pc = 9.
pc = 13.

Memory Simulation:
-------------------------------------------------------------------
 ram[16] = -2080374784. | ram[20] = 0. | ram[24] = 0.  | ram[28] = 0. | ram[32] = 0. |
-------------------------------------------------------------------
 ram[36] = 0. | ram[40] = 0. | ram[44] = 0.  | ram[48] = 0. | ram[52] = 0. |
-------------------------------------------------------------------
 ram[56] = 0. | ram[60] = 0. | ram[64] = 0.  | ram[68] = 0. | ram[72] = 0. |
-------------------------------------------------------------------
 ram[76] = 0. | ram[80] = 0. | ram[84] = 0.  | ram[88] = 0. | ram[92] = 0. |
-------------------------------------------------------------------
 ram[96] = 0. | ram[100] = 0. | ram[104] = 0.  | ram[108] = 0. | ram[112] = 0. |
-------------------------------------------------------------------
 ram[116] = 0. | ram[120] = 0. | ram[124] = 0.  | ram[128] = 0. | ram[132] = 0. |
-------------------------------------------------------------------
 ram[136] = 0. | ram[140] = 0. | ram[144] = 0.  | ram[148] = 0. | ram[152] = 0. |
-------------------------------------------------------------------
 ram[156] = 0. | ram[160] = 0. | ram[164] = 0.  | ram[168] = 0. | ram[172] = 0. |
-------------------------------------------------------------------

Success in running program.

State of registers:
reg[02] = 9          reg[03] = 1          reg[04] = 10         reg[05] = 0
reg[06] = 0          reg[07] = 0          reg[08] = 0          reg[09] = 0
reg[10] = 0          reg[11] = 0          reg[12] = 0          reg[13] = 0
reg[14] = 0          reg[15] = 0          reg[16] = 0          reg[17] = 0
reg[18] = 0          reg[19] = 0          reg[20] = 0          reg[21] = 0
reg[22] = 0          reg[23] = 0
============================================================
Intructions executed by type:
Add    : 4    | Sub    : 0    | Mul    : 0    |
Div    : 0    | Logic. : 0    | Relat. : 0    |
Rotat. : 0    | Load   : 0    | Store  : 0    |
Branch : 0    | Store  : 0    |
============================================================
Elapsed time: 0.000ms.
```

Figura 36: Exemplo de saída completa do simulador.

Devido ao fato de não ser efetuada configuração de *hardware* (e por não ser o objetivo do compilador desenvolvido), durante a simulação não é permitido o uso de periféricos para entrada ou saída, pois estes dispositivos devem ser configurados em uma placa de FPGA através de código VHDL/Verilog, não gerados pelo compilador.

As instruções contidas no arquivo são lidas uma a uma e interpretadas de forma a identificar o tipo de instrução e os parâmetros necessários para sua execução. Neste ponto é verificada a validade da instrução, se os parâmetros obtidos estão de acordo com o tipo de instrução identificada, e se a instrução não tenta utilizar algum dos registradores reservados.

Para simular o uso da memória foi definida uma estrutura para armazenamento das instruções a serem executadas. O PC aponta inicialmente para o primeiro elemento desta estrutura. Após a execução de uma instrução, o PC é incrementado de forma a apontar a próxima instrução, até que ele aponte para uma posição que não possui instrução do programa, neste caso a execução é interrompida. Caso uma instrução inválida seja encontrada um erro é gerado e a simulação é encerrada.

4.4 Conclusão

Este capitulo apresentou o passos seguidos para o desenvolvimento do *N-Compiler*, que utilizou como base para o *front-end* o *framework* Phoenix.

A utilização deste *framework* foi de grande importância por disponibilizar toda a parte inicial de compilação, permitindo assim que durante o desenvolvimento deste trabalho fosse dada maior atenção à fase de geração de código e ao desenvolvimento de uma solução para a falta de suporte a tipos de ponto-flutuante.

Para alcançar o objetivo deste trabalho fez-se necessário a implementação de três módulos: **otimizador**, **gerador de código** e o **suporte a ponto-flutuante**. Os módulos otimizador e gerador de código estão incorporados ao compilador e buscam tornar o compilador funcional.

O módulo de suporte a ponto-flutuante é um módulo externo, que deve ser adicionado à configuração da placa de FPGA, e tem como objetivo permitir que o programador utilize tipos de dados de ponto-flutuante, possibilitando assim que o compilador atenda a um

domínio maior de aplicações.

A Tabela 11 apresenta um resumo comparativo entre o *N-Compiler* e os compiladores não-comerciais apresentados no capítulo 2. Nesta tabela pode-se verificar que grande parte dos compiladores estudados tem como código de entrada a linguagem C, por ser uma linguagem de propósito geral, com um ótimo conjunto de operadores e estruturas de dados (KERNIGHAN; RITCHIE, 1988). Pode-se também perceber que os recursos básicos das linguagens, como comando de decisão `IF` e laços de repetição, estão presentes em todos os compiladores, exceto no caso do *N-Compiler* em que somente o laço de repetição `FOR` está implementado.

Um aspecto a ser destacado na comparação é o suporte ao uso de ponteiros, que não está presente em grande parte dos compiladores estudados. A justificativa para a não implementação deste recurso no *N-Compiler* está no fato de o processador virtual Nios II permitir a configuração de memória de acordo com a necessidade ou característica da arquitetura na qual será utilizado, podendo assim ser acoplado a bancos de memória externos, a processadores de uso geral que farão o controle de acesso a memória, além de outras alternativas possíveis de projeto de *hardware* que inclua o processador virtual Nios II, como as apresentadas na seção 2.4.

Um aspecto importante a ser destacado sobre o *N-Compiler* é sua independência de plataforma e também o fato de gerar código de aplicativos para um processador. Os compiladores estudados, com exceção do Molen, têm como objetivo a geração de código VHDL para a síntese de circuitos digitais. O Molen, embora tenha como objetivo o desenvolvimento de aplicativos, necessita de uma arquitetura específica conhecida como máquina de Molen.

Todos estes compiladores demonstram uma tendência no uso de algoritmos e técnicas de compilação no desenvolvimento de *hardware*. Segundo Hall (HALL; PADUA; PINGALI, 2009) esta é uma tendência positiva e que tem grande potencial de desenvolvimento.

Além do desenvolvimento do compilador, fez-se necessário o desenvolvimento de um

Tabela 11: Comparação entre os compiladores não-comerciais estudados e o *N-Compiler*.

Compilador	**Nenya/ Galadriel**	**Spark**	**ROCCC**	**Trident**	**Molen**	**HThreads**	***N-Compiler***
Código Fonte	Java	C	C/C++, Fortran, Java	C	C	C	C
If	Sim	Sim	Sim	Sim	Sim	Sim	Sim
Laços	Sim	Sim	Sim	Sim	Sim	Sim	Sim (*For*)
Estruturas	Não	Não	Sim	Sim	Sim	Sim	Sim
Ponteiros	Não	Não	Não	Não	Sim	Sim	Não
Funções	Não	Não	Não	Não	Sim	Sim	Não
Repres. Intermediária	GHDP	GHT+ GFCD	CIRRF	LLVM	Machine SUIF IR	Gimple	GHT
Código Gerado	VHDL	VHDL	VHDL+C	VHDL	Virtex II Pro + PowerPc	VHDL	Nios II

simulador do processador virtual Nios II, para realização dos testes do gerador de código. Esta necessidade surgiu devido a característica do processador de necessitar do envio do programa compilado juntamente com o código de configuração da placa.

O desenvolvimento deste simulador foi de grande importância, pois além de facilitar a rotina de testes, agregou grande conhecimento sobre a arquitetura do processador, como por exemplo, formato da palavra e organização dos registradores de uso específico e de uso geral.

5 Testes

5.1 Introdução

Este capítulo tem como objetivo apresentar os testes realizados para verificar o funcionamento do simulador e o ganho de desempenho da implementação dos tipos de dados de ponto flutuante em *hardware*, mostrar exemplo de código otimizado e validar o gerador de código do compilador. A seguir serão descritos os testes e os resultados obtidos.

5.2 Teste do simulador

Para realizar testes no simulador foi necessário primeiro gerar um conjunto de instruções do Nios II válidas, estas instruções foram geradas de acordo com a documentação do fabricante (ALTERA, 2006b). Para facilitar a geração destas instruções foi criado um aplicativo montador (*Assembler*).

Após geradas as instruções foi realizado o teste, executando no simulador este conjunto de instruções. Cada palavra contida no arquivo é lida para a identificação do tipo da instrução e para verificação de seus parâmetros. Caso uma instrução não seja identificada pelo simulador, ela não é processada.

Com os resultados dos teste realizados desta forma, foi possível refazer os testes manualmente e comparar os resultados obtidos com os resultados esperados na simulação.

Além da comparação manual de resultados, foi realizada também uma comparação com um simulador, de propósito educacional, chamado Nios2Sim (LAUSANNE, 2008), vi-

sando assim garantir a validação do funcionamento do simulador.

5.3 Teste dos tipos de dados de ponto flutuante

Para o teste dos tipos de pontos flutuante fez-se necessário a configuração de um FPGA de modo a instalar o processador virtual Nios II e também a configuração de *hardware* para oferecer suporte aos tipos de ponto flutuante. Devido a esta necessidade foi utilizado um *kit* de desenvolvimento *DE2 - Altera Development and Education Board*, equipado com o FPGA EP2C35. Este *kit* conta com um conjunto completo de periféricos que visam atender diversas necessidades de configuração e uso do FPGA.

O *kit* precisa ser configurado de acordo com a característica ou necessidade da aplicação a ser executada. Esta configuração é realizada através de códigos em VHDL ou Verilog, que são enviados para a placa através de qualquer computador que possua uma conexão USB e *drivers* e aplicativos instalados.

Como o foco deste trabalho não foi o estudo das configurações completas do *kit* de desenvolvimento, optou-se pelo uso de uma configuração padrão, fornecida pelo fabricante, alterada de modo a fornecer suporte ao *hardware* responsável pela execução das operações de ponto flutuante. Para gerar o código VHDL e Verilog do *hardware* foram utilizados aplicativos gráficos (Quartus II e SOPC Builder), disponibilizados pelo fabricante do *kit*.

Após a geração do *hardware* para manipulação de ponto flutuante, foi gerada, no processo de compilação da linguagem de descrição de *hardware*, uma biblioteca da linguagem C (`system.h`), na qual está contida as chamadas das funções descritas em *harware*. Esta biblioteca pode ser utilizada em qualquer ambiente de programação da linguagem C/C++, através da diretiva `#include`.

A chamada às funções para a manipulação destes tipos de dados seguem a descrição encontrada em (ALTERA, 2006a), um exemplo destas chamadas pode ser visto na Figura 37.

```
int __builtin_custom_in (int n);
int __builtin_custom_ini (int n, int dataa);
```

Figura 37: Exemplo de chamada a funções para manipulação de tipos de ponto flutuante.

Para a realização do teste foi utilizado um programa desenvolvido com a Nios II IDE (ALTERA, 2008b), *suite* de programação para o Nios II. Neste programa, um vetor de 1000 posições foi iniciado com valores gerados de forma aleatória, os quais foram submetidos às quatro operações básicas (adição, subtração, multiplicação e divisão). Um trecho do programa de teste pode ser visto na Figura 38.

```
89  /*********************************************
90   * ADD
91   *********************************************/
92  PERF_RESET (PERFORMANCE_COUNTER_BASE);
93  PERF_START_MEASURING (PERFORMANCE_COUNTER_BASE);
94
95  for (i = 0; i < NUM_ITERATIONS; i++)
96  {
97      context = alt_irq_disable_all();
98
99    result_CI = fp_add_CI(random_a[i], random_b[i]);
100   result_SW = fp_add_SW(random_a[i], random_b[i]);
101
102   alt_irq_enable_all(context);
103
104   if (result_CI != result_SW)
105   {
106     printf("\nCI: %f + %f = %f", random_a[i], random_b[i], result_CI);
107     printf("\nSW: %f + %f = %f", random_a[i], random_b[i], result_SW);
108   }
109 }
```

Figura 38: Trecho do programa utilizado para o teste de desempenho da implementação dos tipos de ponto flutuante em *hardware*.

A execução deste teste teve como objetivo a verificação de ganho de desempenho da implementação da manipulação dos tipos de dados de ponto flutuante em *hardware* quando comprada com a implementação via *software*.

O execução do programa de teste apresentou como saída o tempo gasto na execução de cada uma das operações definidas em seu código, em *hardware* e do seu equivalente quando executado em *software*, tanto em segundos quanto em ciclos de relógio do processador. A Tabela 12 e o Gráfico 39 apresentam o percentual de tempo gasto na execução das instruções em forma de *hardware* e mantissa e expoente em relação ao tempo total de execução da aplicação.

Tabela 12: Percentagem de tempo para execução das operações em forma de *hardware* e em forma de mantissa e expoente, em relação ao tempo total de execução do programa.

	Mantissa Exponente	**Hardware**
Soma	48,30%	2,20%
Subtração	49,10%	2,34%
Multiplicação	56,80%	3%
Divisão	60,40%	3,02%

No Gráfico 39 pode-se perceber a diferença de desempenho apresentada, assim como a variação de tempo entre uma operação e outra.

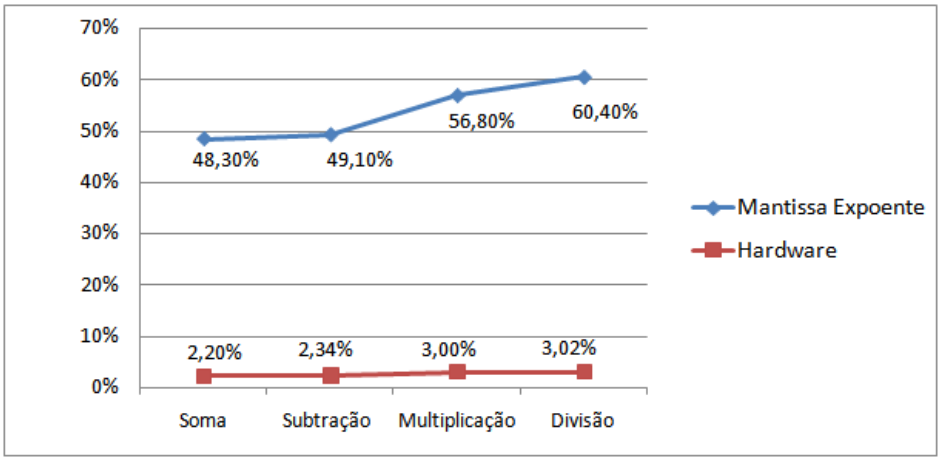

Figura 39: Gráfico apresentando a percentagem de tempo para execução das operações em forma de *hardware* e em forma de mantissa e expoente, em relação ao tempo total de execução do programa.

Com estes dados foi possível fazer uma análise do ganho de desempenho obtido na execução de uma instrução e obter o fator de aceleração, ou seja, quantas vezes uma implementação é mais rápida que a outra. A Tabela 13 apresenta a relação de ganho de desempenho da implementação em *hardware* em relação a implementação na forma de mantissa e expoente, para o programa de teste executado.

Tabela 13: Ganho de desempenho alcançado com o uso de instruções para ponto flutuante implementadas em *hardware*

FPGA	**Soma**	**Subtração**	**Multiplicação**	**Divisão**
EP2C35	22x	21x	19x	20x

Devido ao fato do tempo gasto com cada operação básica tender a ser constante quando executado nas mesmas condições repetidas vezes, a realização de um teste único mostrou-se satisfatória para a obtenção dos resultados.

Os resultados obtidos demonstram uma grande diferença de desempenho entre as duas implementações testadas. A implementação em forma de *hardware* se mostrou mais rápida além de ser implementada de forma simples através das ferramentas disponibilizadas pelo fabricante do FPGA, quando comparada a implementação em forma de mantissa e expoente.

A implementação baseada em mantissa e expoente apresentou, no teste realizado, um desempenho baixo, por exigir grande número de acessos a memória. Este grande número de acessos a memória é justificado pelo uso excessivo de registradores exigido, um para a parte decimal e um para a parte fracionária. Em um processador com número reduzido de registradores, como é o caso do Nios II, isso representa uma grande limitação.

A implementação baseada em *hardware* apresentou um desempenho muito bom. Além disso, este tipo de implementação permite a definição de qual banco de registradores utilizar, caso exista mais de um no sistema, o que aumenta a flexibilidade do sistema a ser desenvolvido.

5.4 Teste do otimizador de código

Embora não se possa garantir ganho em um código submetido aos algoritmos de otimização de código, foi realizado um teste para verificar o funcionamento dos algoritmos de otimização. Na Figura 40 (a) pode ser visto o exemplo de um programa convertido para instruções de três endereços antes de ser submetido ao otimizador de código. Na Figura 40 (b) pode ser visto o resultado, ou seja, o código após a execuções dos algoritmos:

- Eliminação de sub-expressões comuns;
- Propagação de cópias;

- Tranposição para constantes;
- Movimentação de código ciclo-invariante;
- Otimização Peephole.

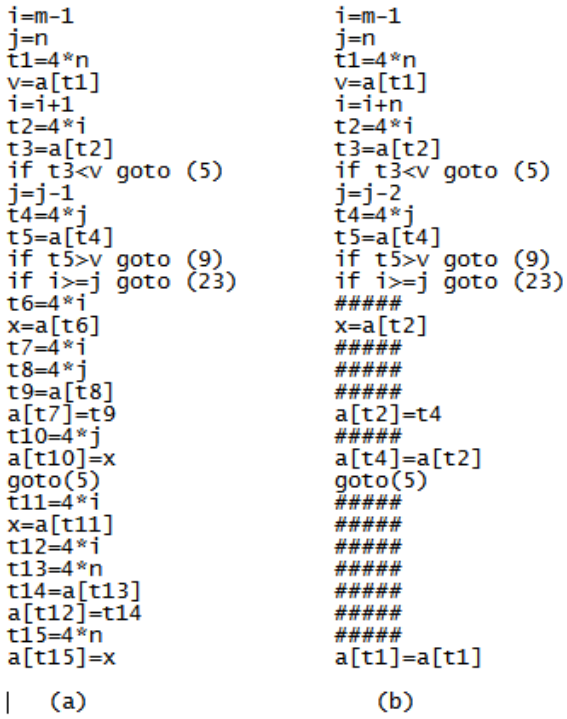

Figura 40: Conjunto de instruções de três endereços (a) (AHO et al., 2007). Instruções após otimização (b).

O código não otimizado foi retirado de (AHO et al., 2007) e submetido ao simulador de forma a testar o correto funcionamento do mesmo, o resultado obtido com este teste foi o esperado, mostrando que os algoritmos funcionam corretamente.

5.5 Testes do gerador de código para o processador virtual Nios II

Para teste do gerador de código foi necessária a utilização de um simulador de instruções do Nios II (LIMA et al., 2008), este simulador foi descrito na seção 4.3 e foi baseado

na documentação do fabricante (ALTERA, 2007a). Para a realização dos testes foram gerados códigos de alguns programas simples, visando verificar se as instruções geradas pelo *N-Compiler* são corretas.

No primeiro teste compilou-se o código apresentado na Figura 41. Neste trecho de código são criadas algumas variáveis e são realizados três cálculos, com o objetivo de testar o código gerado e também a alocação dos registradores. Por ser um teste pequeno, foi possível realiza-lo manualmente. Além disso, este teste tem como objetivo verificar como o compilador decompõe instruções com mais de três argumentos, como pode ser visto na linha 7, para a geração do GHT e também como o gerador de código se comporta ao receber este grafo.

```
1   void funcao()
2   {
3       int Cont;
4       int D = 9;
5       int A = 1;
6       int B = D;
7       Cont = A + B * D;
8       B = 10 + 2;
9       A = A * 15;
10  }
```

Figura 41: Código fonte usado no primeiro teste do compilador.

Durante o processo de compilação do código fonte é gerado o GHT da função definida no programa de teste, este grafo é formado por instruções de três endereços, como pode ser visto na Figura 42. Verifica-se que foi criada uma variável virtual (`T0`) para auxiliar no processo de tradução da instrução encontrada na linha 7.

O grafo apresentado na Figura 42 foi passado para o gerador de código. O gerador lê as instruções de três endereços contidas no grafo da função, identifica o tipo da instrução (unária ou binária), verifica seus parâmetros e executa a geração do código para a máquina alvo.

Após gerado o código para a arquitetura alvo o mesmo foi testado no simulador, para

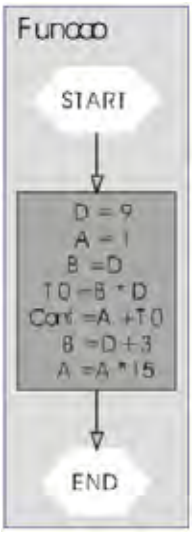

Figura 42: GHT gerado pelo compilador.

verificar se estava de acordo com a especificação do processador. O resultado da execução no simulador pode ser visto na Figura 43, na qual apresenta-se o estado dos registradores do processador virtual após a execução do programa de teste.

No resultado pode-se ver que a estratégia de alocar os registradores seqüencialmente está corretamente implementada, embora não tenha sido necessário reutilizar nenhum registrador por se tratar de um trecho de programa bastante simples.

Na Figura 44 é apresentado o resultado da execução do mesmo programa de teste, mas agora mostrando passo a passo o estado dos registradores, para uma avaliação mais detalhada da execução. O simulador permite selecionar qual o tipo de saída será gerada completa ou estado final dos registradores.

A execução do programa de teste pelo simulador mostrou que o código gerado pelo compilador está de acordo com as especificações das instruções do processador, o que garante que a linguagem para a máquina alvo foi corretamente gerada pelo gerador de código.

Após a realização do primeiro teste, que teve como objetivo garantir que as instruções simples estavam sendo geradas de forma correta, foi realizado um teste visando verificar a geração de código para o comando de decisão *IF*. Para esta verificação foi utilizado o

```
Starting simulation (file size 16) ...

Program Counter:
pc = 1.
pc = 5.
pc = 9.
pc = 13.

Memory Simulation:
-------------------------------------------------------------
 ram[16] = -2080374784. | ram[20] = 0. | ram[24] = 0.  | ram[28] = 0. | ram[32] = 0. |
-------------------------------------------------------------
 ram[36] = 0. | ram[40] = 0. | ram[44] = 0.  | ram[48] = 0. | ram[52] = 0. |
-------------------------------------------------------------
 ram[56] = 0. | ram[60] = 0. | ram[64] = 0.  | ram[68] = 0. | ram[72] = 0. |
-------------------------------------------------------------
 ram[76] = 0. | ram[80] = 0. | ram[84] = 0.  | ram[88] = 0. | ram[92] = 0. |
-------------------------------------------------------------
 ram[96] = 0. | ram[100] = 0. | ram[104] = 0.  | ram[108] = 0. | ram[112] = 0. |
-------------------------------------------------------------
 ram[116] = 0. | ram[120] = 0. | ram[124] = 0.  | ram[128] = 0. | ram[132] = 0. |
-------------------------------------------------------------
 ram[136] = 0. | ram[140] = 0. | ram[144] = 0.  | ram[148] = 0. | ram[152] = 0. |
-------------------------------------------------------------
 ram[156] = 0. | ram[160] = 0. | ram[164] = 0.  | ram[168] = 0. | ram[172] = 0. |
-------------------------------------------------------------

Success in running program.

State of registers:
reg[02] = 9          reg[03] = 1          reg[04] = 10         reg[05] = 0
reg[06] = 0          reg[07] = 0          reg[08] = 0          reg[09] = 0
reg[10] = 0          reg[11] = 0          reg[12] = 0          reg[13] = 0
reg[14] = 0          reg[15] = 0          reg[16] = 0          reg[17] = 0
reg[18] = 0          reg[19] = 0          reg[20] = 0          reg[21] = 0
reg[22] = 0          reg[23] = 0
================================================================
Intructions executed by type:
Add    : 4    | Sub    : 0    | Mul    : 0    |
Div    : 0    | Logic. : 0    | Relat. : 0    |
Rotat. : 0    | Load   : 0    | Store  : 0    |
Branch : 0    | Store  : 0    |
================================================================
Elapsed time: 0.000ms.
```

Figura 43: Resultado da execução do código de teste no simulador.

mesmo método do teste anterior, ou seja a definição de um trecho de código simples, a geração do GHT correspondente a função, a geração do código da máquina alvo e logo após a análise dos resultados obtidos pelo simulador.

Na Figura 45 apresenta-se o trecho de programa utilizado durante a execução do teste. Este código possui uma condição simples de verificação de uma variável.

Com a execução deste código pelo compilador foi gerado o grafo que pode ser visto na Figura 46. Durante a geração deste grafo foi feita a separação dos blocos básicos do programa, que são representados pelos retângulos brancos. O código para a arquitetura alvo é gerado através da tradução de cada um destes blocos básicos. Cada bloco básico é identificado através do nome da função que ele representa e um identificador (`NO->ID`), que é usado de forma crescente e identifica a ordem na qual devem ser traduzidos.

```
Starting simulation (file size 16) ...
Program Counter:

pc = 1.
------------------------------------------------------------
reg[02] = 0     reg[03] = 0     reg[04] = 0     reg[05] = 0
reg[06] = 0     reg[07] = 0     reg[08] = 0     reg[09] = 0
reg[10] = 0     reg[11] = 0     reg[12] = 0     reg[13] = 0
reg[14] = 0     reg[15] = 0     reg[16] = 0     reg[17] = 0
reg[18] = 0     reg[19] = 0     reg[20] = 0     reg[21] = 0
reg[22] = 0     reg[23] = 0
------------------------------------------------------------
pc = 5.
------------------------------------------------------------
reg[02] = 9     reg[03] = 0     reg[04] = 0     reg[05] = 0
reg[06] = 0     reg[07] = 0     reg[08] = 0     reg[09] = 0
reg[10] = 0     reg[11] = 0     reg[12] = 0     reg[13] = 0
reg[14] = 0     reg[15] = 0     reg[16] = 0     reg[17] = 0
reg[18] = 0     reg[19] = 0     reg[20] = 0     reg[21] = 0
reg[22] = 0     reg[23] = 0
------------------------------------------------------------
pc = 9.
------------------------------------------------------------
reg[02] = 9     reg[03] = 1     reg[04] = 0     reg[05] = 0
reg[06] = 0     reg[07] = 0     reg[08] = 0     reg[09] = 0
reg[10] = 0     reg[11] = 0     reg[12] = 0     reg[13] = 0
reg[14] = 0     reg[15] = 0     reg[16] = 0     reg[17] = 0
reg[18] = 0     reg[19] = 0     reg[20] = 0     reg[21] = 0
reg[22] = 0     reg[23] = 0
------------------------------------------------------------
pc = 13.
------------------------------------------------------------
reg[02] = 9     reg[03] = 1     reg[04] = 10    reg[05] = 0
reg[06] = 0     reg[07] = 0     reg[08] = 0     reg[09] = 0
reg[10] = 0     reg[11] = 0     reg[12] = 0     reg[13] = 0
reg[14] = 0     reg[15] = 0     reg[16] = 0     reg[17] = 0
reg[18] = 0     reg[19] = 0     reg[20] = 0     reg[21] = 0
reg[22] = 0     reg[23] = 0
------------------------------------------------------------
Success in running program.

State of registers:
====================================================================
Intructions executed by type:

Add   : 4   | Sub   : 0   | Mul   : 0   |
Div   : 0   | Logic. : 0   | Relat. : 0   |
Rotat. : 0   | Load   : 0   | Store  : 0   |
Branch : 0   | Store  : 0   |
====================================================================
Elapsed time: 0.000ms.
```

Figura 44: Resultado completo da execução do código de teste no simulador.

```
void funcao()
{

   int D = 9;
   int A = 1;
   int B = D;
   int Cont;
   Cont = A + B * D;
   if (Cont > 10)
     B = D + 3;
   else
      A = A * 15;
}
```

Figura 45: Código fonte utilizado para o teste de uso do comando *IF*.

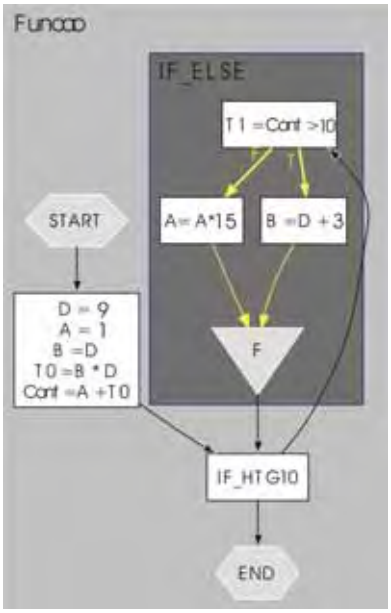

Figura 46: Grafo gerado durante o teste de uso do comando *IF*.

Após a realização da geração do grafo pode-se ver que todos os elementos necessários para a geração do código para a máquina alvo estavam presentes no grafo. Com isso foi realizado o teste do código com o uso do simulador. Neste teste pode ser verificado o correto funcionamento da geração do código para o comando de decisão *IF*. O resultado deste teste é apresentado na Figura 47.

State of registers:

reg[02] = 9	reg[03] = 15	reg[04] = 12	reg[05] = 81
reg[06] = 82	reg[07] = 0	reg[08] = 0	reg[09] = 0
reg[10] = 0	reg[11] = 0	reg[12] = 0	reg[13] = 0
reg[14] = 0	reg[15] = 0	reg[16] = 0	reg[17] = 0
reg[18] = 0	reg[19] = 0	reg[20] = 0	reg[21] = 0
reg[22] = 0	reg[23] = 0		

==========

Figura 47: Estado final dos registradores após teste de uso do comando *IF* no simulador.

Como último teste, foi verificada a geração de código para o comando de repetição *For*, para isso foi gerado um pequeno trecho de programa que pode ser visto na Figura 48. Neste código também é utilizado um comando *IF*.

Com a execução deste código pelo compilador foi gerado o grafo que pode ser visto na Figura 49. A montagem deste grafo foi feita seguindo o mesmo método descrito na

```
void funcaoFor()
{
   int A = 2;
   int B = 10;
   int Cont;
   for (Cont=0; Cont < 5; Cont++)
   {
      A = A + Cont;
      if (A > B)
        B = A;
   }
}
```

Figura 48: Código fonte utilizado para o teste de uso do comando *FOR*.

geração do grafo apresentado na Figura 46, ou seja separação dos blocos básicos da função e a identificação de cada um deles.

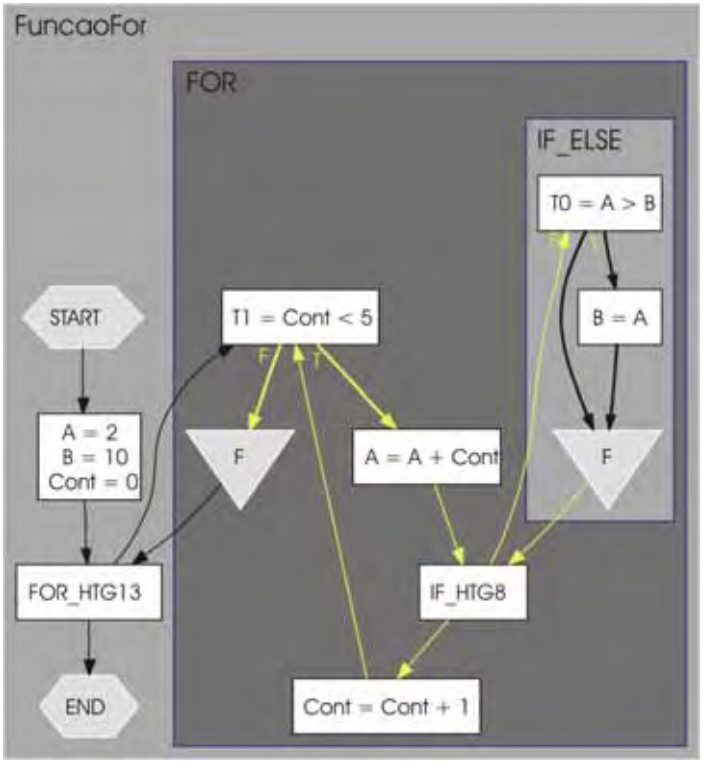

Figura 49: Grafo gerado durante o teste de uso do comando *FOR*.

Com a correta geração do grafo foi realizado o teste do código gerado para a máquina alvo no simulador. O resultado encontrado foi o esperado, ou seja, o obtido no teste realizado manualmente a fim de validar a geração do código. O resultado deste teste é apresentado na Figura 50.

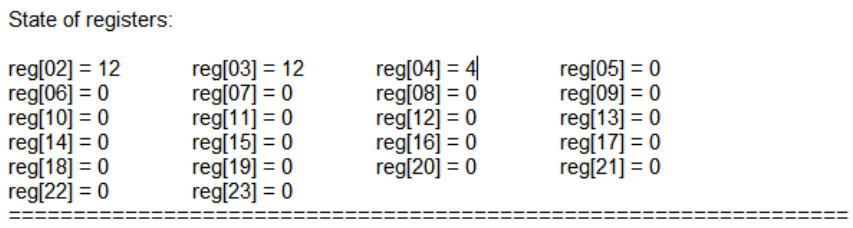

Figura 50: Estado final dos registradores após teste de uso do comando *For* no simulador.

Como citado anteriormente, o uso do simulador para os testes foi necessário pois, para o teste em um *kit* de desenvolvimento o compilador precisa enviar, juntamente com o código gerado, o código VHDL ou Verilog necessário para a configuração do FPGA.

5.6 Conclusão

Os testes realizados com a implementação dos tipos de dados de ponto flutuante em *hardware* mostraram-se eficientes e com bom desempenho. Devido a estes fatores esta abordagem foi a escolhida, mesmo sendo necessário acoplar o código necessário ao projeto do *hardware* descrito em VHDL ou Verilog.

Durante os testes do gerador de código foi possível verificar que as instruções geradas estão de acordo com a documentação disponibilizada pelo fabricante e portanto prontas para serem executadas em um FPGA Altera que possua o processador Nios II instalado.

Quanto aos algoritmos de otimização, foi realizado um teste para verificação do seu funcionamento, mesmo sabendo-se que código otimizado depende do código de entrada, podendo apresentar ganho em termos de tempo de execução ou diminuição do número de instruções a serem executadas, bem como podem não apresentar nenhum ganho.

6 Conclusões e Trabalhos Futuros

6.1 Conclusão

Este trabalho teve como objetivo apresentar o desenvolvimento de um compilador que tem o objetivo de gerar código para o processador de núcleo virtual Nios II. Este compilador visa permitir o desenvolvimento e migração de aplicativos para uma arquitetura reconfigurável, arquitetura esta que apresenta um grande ganho de desempenho quando comparada com as arquiteturas tradicionais formadas por processadores de uso geral (HAUCK; DEHON, 2007).

O código gerado pode ser executado em qualquer arquitetura que possua o processador Nios II, que por ser virtual deve ser configurado através de uma linguagem de descrição de *hardware*, como por exemplo, Verilog ou VHDL independente de periféricos ou outros processadores que podem estar conectados a arquitetura.

A utilização do *framework*, como base para o desenvolvimento, foi de grande importância para a conclusão deste trabalho, por oferecer toda a parte inicial do processo de compilação, como análise léxica, análise sintática, análise semântica e toda a geração da representação intermediária. Tendo o *framework* como base foi possível dar maior atenção para as fases de otimização, implementação dos tipos de ponto flutuante e geração de código. O gerador é o principal módulo desenvolvido contando com a geração do conjunto básico de instruções e também com instruções desenvolvidas para a manipulação dos tipos de ponto flutuante.

Durante o desenvolvimento também foram encontradas dificuldades como:

- Definição do formato correto da palavra do processador;
- Desenvolvimento de um simulador para possibilitar os testes do gerador de código;
- Uso de uma placa com FPGA para testes da implementação dos tipos de dados desenvolvidos em *hardware*;
- Necessidade de aprendizagem sobre síntese de alto nível de circuitos digitais para uso de placas com FPGA.

Estas dificuldades foram superadas através do estudo da documentação do fabricante do processador, permitindo assim a definição do formato correto da palavra. O aprendizado sobre síntese de alto nível foi adquirido com o uso do *Kit* de desenvolvimento do fabricante.

6.2 Trabalhos futuros

Como trabalhos futuros pode-se propor algumas melhorias, que tornem o compilador uma ferramenta mais completa, com objetivo de permitir a migração de qualquer tipo de aplicativos para uma arquitetura reconfigurável. Algumas melhorias que podem ser implementadas são:

- Desenvolver um módulo para a otimização da alocação de registradores durante a geração de código;
- Incorporar ao compilador outros recursos da linguagem C, visando maior compatibilidade com o código fonte de aplicativos já existentes. Dentre estes recursos pode-se citar:
  - Chamada a funções;
  - Recursividade;
  - Comando de decisão *case*;

- laços de repetição *while* e *do-while* e
- Suporte a bibliotecas externas existentes para a linguagem C.
- Realizar também um estudo sobre a possibilidade de implementação de ponteiros, tendo em vista que a memória a ser utilizada pode ser configurada de acordo com a arquitetura do sistema a ser utilizado.

Como citado anteriormente estas alterações visam viabilizar o uso do compilador para o desenvolvimento e migração dos mais diversos tipos de aplicativos, além de tornar o compilador mais atraente para os desenvolvedores habituados com a linguagem de programação C.

Além das implementações que podem ser adicionadas ao compilador, podem ser realizadas alterações ou desenvolvimento de um novo *back-end*, visando permitir seu uso com FPGAs de outros fabricantes que apresentam um menor custo. Com esta alteração será possível a utilização deste compilador em outros projetos, como por exemplo, o projeto Ninho dos Pardais da Universidade Tecnológica do Paraná (NIPAR, 2009), que tem como objetivo desenvolver um dispositivo de robótica móvel para uso do ensino de robótica para alunos do ensino médio.

Referências

AHO, A. V. et al. *Compilers*: Principles, techniques, and tools. Reading, Massachusetts: Addison-Wesley, 2007.

ALCALDE, A. L. P.; ORTMANN, M. S.; MUSSA, S. A. Nios II processor implemented in fpga: An application on control of a pfc converter. *Power Electronics Specialists Conference, 2008. PESC 2008. IEEE*, Rhodes, Grécia, p. 4446–4451, 2008. ISSN 0275-9306.

ALTERA, I. *Hardware Acceleration*. Acessado em abril de 2008. Disponível em: <http://www.altera.com/products/ip/processors/nios2/benefits/performance/ni2-acceleration.html.>.

ALTERA, I. *Nios* II *Custom Instruction User Guide.* 2006.

ALTERA, I. *Nios* II *Processor Reference Handbook.* 2006.

ALTERA, I. *Instruction Set Reference.* 2007.

ALTERA, I. *Nios II C2H Compiler: User Guide.* 2007.

ALTERA, I. *Nios* II *Floating Point Custom Instructions*. 2008.

ALTERA, I. *Nios* II *Integrated Development Environment.* 2008.

ANDREWS, D. L. et al. Achieving programming model abstractions for reconfigurable computing. *IEEE Trans. VLSI Syst*, v. 16, n. 1, p. 34–44, 2008.

ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. New York: ANSI/IEEE Std 754-1985, 1985.

AT&T. *Graphviz: Open source graph drawing software*. Acessado em dezembro de 2007., 2007. Disponível em: <http://www.research.att.com/sw/tools/graphviz.>.

BANSAL, S.; AIKEN, A. Automatic generation of peephole superoptimizers. In: SHEN, J. P.; MARTONOSI, M. (Ed.). *ASPLOS*. [S.l.]: ACM, 2006. p. 394–403. ISBN 1-59593-451-0.

BRIGGS, P. et al. Coloring heuristics for register allocation. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 39, n. 4, p. 283–294, 2004. ISSN 0362-1340.

BUYUKKURT, B.; GUO, Z.; NAJJAR, W. A. Impact of loop unrolling on area, throughput and clock frequency in ROCCC: C to VHDL compiler for FPGAs. In: BERTELS, K.; CARDOSO, J. M. P.; VASSILIADIS, S. (Ed.). *ARC*. [S.l.]: Springer, 2006. (Lecture Notes in Computer Science, v. 3985), p. 401–412.

CAI, Q.; XUE, J. Optimal and efficient speculation-based partial redundancy elimination. In: *CGO*. [S.l.]: IEEE Computer Society, 2003. p. 91–104. ISBN 0-7695-1913-X.

CARDOSO, J. M. P. Tese, *Compilação de algoritmos em java para sistemas computacionais reconfiguráveis com exploração de paralelismo ao nível das operações*. [S.l.]: Escola Técnica de Lisboa, Portugal, 2000.

CARDOSO, J. M. P.; NETO, H. C. Compilation increasing the scheduling scope for multi-memory-FPGA-based custom computing machines. *Lecture Notes in Computer Science*, v. 2147, 2001.

CHOW, F. C.; HENNESSY, J. L. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 12, n. 4, p. 501–536, 1990. ISSN 0164-0925.

COOPER, K. D.; DASGUPTA, A.; ECKHARDT, J. Revisiting graph coloring register allocation: A study of the chaitin-briggs and callahan-koblenz algorithms. *Lecture notes in computer science*, v. 4339, p. 1–16, 2008. ISSN 0302-9743.

COOPER, K. D.; HARVEY, T. J.; PEIXOTTO, D. M. Chow and hennessy vs. chaitin-briggs register allocation: Using adaptive compilation to fairly compare algorithms. Springer Berlin / Heidelberg, v. 4339/2006, p. 1 – 16, 2006. ISSN 0302-9743 (Print) 1611-3349 (Online).

DUARTE, F. L. Dissertação, *Phoenix – um framework para trabalhos em síntese de alto nível de circuitos digitais*. [S.l.]: Universidade Federal de Uberlândia, fev. 17 2006.

ECLIPSE, F. *Eclipse Org*. Acessado em abril de 2009., 2009. Disponível em: <http://www.eclipse.org/>.

ESTRIN, G. et al. Parallel processing on a restructurable computer system. *IEEE Trans. Electronic Computers*, EC-12, p. 747–754, 1963.

GIRKAR, M.; POLYCHRONOPOULOS, C. D. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 3, n. 2, p. 166–178, 1992. ISSN 1045-9219.

GUO, Z. et al. *Optimized Generation of Data-Path from C Codes for FPGAs*. Washington, DC, USA: IEEE Computer Society, 2005. 112–117 p.

GUPTA, S. et al. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In: *International Conference on VLSI Design*. [S.l.: s.n.], 2003. p. 461 – 466.

HALL, M.; PADUA, D.; PINGALI, K. Compiler research: the next 50 years. *Commun. ACM*, ACM, New York, NY, USA, v. 52, n. 2, p. 60–67, 2009. ISSN 0001-0782.

HALL, M. W. et al. Maximizing multiprocessor performance with the SUIF compiler. *Digital Technical Journal of Digital Equipment Corporation*, v. 10, n. 1, 1998.

HAUCK, S.; DEHON, A. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon)*. [S.l.]: Morgan Kaufmann, 2007. Hardcover. ISBN 0123705223.

HAUCK, S. et al. The chimaera reconfigurable functional unit. *IEEE Trans. VLSI Syst*, v. 12, n. 2, p. 206–217, 2004.

HENNESSY, J. L.; PATTERSON, D. A. *Arquitetura de Computadores*: Uma abordagem quantitativa. São Paulo: Editora Campus, 2003.

HPCWIRE. *Cray Selects DRC FPGA Coprocessors for Supercomputers.* Acessado em junho de 2009, 2009. Disponível em: <http://www.hpcwire.com/offthewire/17884069.html>.

KERNIGHAN, B. W.; RITCHIE, D. M. *The C Programming Language*. Second. [S.l.]: Prentice Hall, 1988.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California: [s.n.], 2004. p. 75–86. ISBN 0-7695-2102-9.

LAUFER, R.; TAYLOR, R. R.; SCHMIT, H. PCI-PipeRench and the SwordAPI: A system for stream-based reconfigurable computing. In: POCEK, K. L.; ARNOLD, J. (Ed.). *IEEE Symposium on FPGAs for Custom Computing Machines*. Los Alamitos, CA: IEEE Computer Society Press, 1999. p. 200–208.

LAUSANNE, E. P. F. de. *Nios* II *Simulator.* Acessado em fevereiro de 2009., 2008. Disponível em: <http://lapwww.epfl.ch/courses/archord1/labs/niosIIsimulator.pdf>.

LEE, J. K.; PALSBERG, J.; PEREIRA, F. M. Q. Aliased register allocation for straight-line programs is np-complete. *Theor. Comput. Sci.*, Elsevier Science Publishers Ltd., Essex, UK, v. 407, n. 1-3, p. 258–273, 2008. ISSN 0304-3975.

LIMA, W. S. et al. Simulação de execução de instruções do processador de núcleo virtual Nios II. *XXXIV Conferência Latinoamericana de Informática.*, p. 649–658, 2008.

MEDEIROS, E. *Desenvolvendo Software com UML 2.0*: Definitivo. São Paulo: Pearson Makron Books, 2004.

MENTOR, G. *Datasheet: EDGE Developer Suite.* 2009.

MERRILL, J. GENERIC and GIMPLE: A new tree representation for entire functions. In: HUTTON, A. J.; DONOVAN, S.; ROSS, C. C. (Ed.). *Proceedings of the GCC Developers Summit May 25–27, 2003, Ottawa, Ontario Canada.* [S.l.: s.n.], 2003. p. 171–193.

MITTAL, G. et al. An overview of a compiler for mapping software binaries to hardware. *IEEE Trans. VLSI Syst*, v. 15, n. 11, p. 1177–1190, 2007.

MORENO, E. D.; PENTEADO, C. G.; SILVA, A. C. R. d. *Microcontroladores e FPGAs*: aplicações em automação. [S.l.]: Novatec Editora, 2005.

NAMBIAR, V. P.; KHALIL-HANI, M.; ZABIDI, M. M. A. Accelerating the aes encryption function in openssl for embedded systems. *International Conference on Electronic Design, ICED 2008*, Penang, Malaysia, p. 1 – 5, 2008.

NIPAR. *Ninho dos Pardais.* Acessado em maio de 2009., 2009. Disponível em: <http://www.cp.utfpr.edu.br/nipar/>.

NOH, S. H. et al. Implementation and performance evaluation of the lrfu replacement policy. In: *in Proceeding of the 23th Euromicro Conference*. [S.l.: s.n.], 1997. p. 106–111.

PANAINTE, E. M.; BERTELS, K.; VASSILIADIS, S. The molen compiler for reconfigurable processors. *ACM Trans. Embedded Comput. Systems*, v. 6, n. 1, p. 6, 2007.

PONTARELLI, S. et al. Analysis and evaluations of reliability of reconfigurable fpgas. In: . [S.l.]: Journal of Electronic Testing, 2008. v. 24, n. 1-3.

QUICKTURN, A. A. C. proceeding with peer review, *MercuryTM design verification system technology backgrounder*. Acessado em dezembro de 2006., 1999. Disponível em: <http://www.quickturn.com/products/mercurybackgrounder.htm.>.

QUICKTURN, A. A. C. proceeding with peer review, *System realizerTM*. Acessado em dezembro de 2006., 1999. Disponível em: <http://www.quickturn.com/products/systemrealizer.htm.>.

RAZDAN, R.; SMITH, M. D. A high-performance microarchitecture with hardware-programmable functional units. In: IEEE/ACM. *Proceedings of the 27th Annual International Symposium on Microarchitecture*. [S.l.], 1994. p. 172–80.

SCHILDT, H. *C*: Completo e total. São Paulo: Editora Pearson, 1997.

SILVA, A. C. F. d.; LOBATO, R. S.; ULSON, R. S. Compilação para o processador de núcleo virtual Nios II. *XXXIV Conferência Latinoamericana de Informática.*, p. 639–648, 2008.

SIMA, M. et al. Field-programmable custom computing machines — A taxonomy. *Lecture Notes in Computer Science*, v. 2438, p. 79–88, 2002. ISSN 0302-9743.

SMITH, M. D.; HOLLOWAY, G. *An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization*. Acessado em abril de 2008., 2008. Disponível em: <http://www.eecs.harvard.edu/hube/software/nci/overview.>.

SOBEK, S.; BURKE, K. *PowerPC Embedded Application Binary Interface*. Acessado em abril de 2008., 1995. Disponível em: <http://ftp.twaren.net/Unix/Sourceware/binutils/ppc-docs/ppc-eabi-1995-01.pdf.>.

Stanford SUIF Compiler Group. *SUIF: A Parallelizing & Optimizing Research Compiler*. [S.l.], maio 1994.

TRIPP, J. L.; GOKHALE, M. B.; PETERSON, K. D. *Trident: From High-Level Language to Hardware Circuitry*. Los Alamitos, CA, USA: IEEE Computer Society, 2007. 28-37 p.

TRIPP, J. L.; JACKSON, P. A.; HUTCHINGS, B. Sea cucumber: A synthesizing compiler for fpgas. In: *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*. London, UK: Springer-Verlag, 2002. p. 875–885. ISBN 3-540-44108-5.

VASSILIADIS, S. et al. The molen programming paradigm. In: *SAMOS*. [S.l.]: Springer, 2004. (Lecture Notes in Computer Science, v. 3133). ISBN 3-540-22377-0.

VUILLEMIN, J. et al. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, v. 4, n. 1, p. 56–69, 1996.

WITTIG, R.; CHOW, P. OneChip: An FPGA processor with reconfigurable logic. In: POCEK, K. L.; ARNOLD, J. (Ed.). *IEEE Symposium on FPGAs for Custom Computing Machines*. Napa Valley, CA, USA: IEEE Computer Society Press, 1996. p. 126–135. ISBN 0-8186-7548-9.

ZHAO, H.; ZHENG, X.; LIU, W. Intelligent traffic control system based on dsp and nios II. *International Asia Conference on Informatics in Control, Automation and Robotics*, Bangkok, Thailandia, p. 90 – 94, 2009.