UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"

Pós-Graduação em Ciência da Computação

Willian dos Santos Lima

Compilação de *bytecodes* Java para um ambiente de arquitetura reconfigurável

UNESP

2009

Willian dos Santos Lima

Compilação de *bytecodes* Java para um ambiente de arquitetura reconfigurável

Orientadora: Prof[a]. Dr[a]. Renata Spolon Lobato

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.

UNESP

2009

Willian dos Santos Lima

Compilação de *bytecodes* Java para um ambiente de 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. Eduardo Marques
USP – São Carlos

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

São José do Rio Preto, 27 de fevereiro de 2009.

Agradecimentos

A Deus por me dar forças e sabedoria, especialmente nos momentos mais adversos. À minha família por me dar suporte para que eu pudesse chegar até esta etapa da minha vida. À minha namorada por toda a compreensão.

À minha orientadora, Prof^{a}. Dr^{a}. Renata Spolon Lobato, por toda a dedicação não só durante o período de mestrado, mas desde a graduação.

A todos os professores e colegas que estiveram perto de mim e me ajudaram a chegar até aqui.

À banca examinadora e ao CNPq.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ASIC: *Application Specific Integrated Circuit*

DAG: *Directed Acyclic Graph*

DU: Definição-Uso

FPGA: *Field Programmable Gate Array*

GCD: *Greatest Common Divisor*

GDC: Grafo de Dependência de Controle

GDD: Grafo de Dependência de Dados

GFC: Grafo de Fluxo de Controle

GFD: Grafo de Fluxo de Dados

GDP Grafo de Dependência de Programa

HDL: Hardware *Description Language*

JVM: *Java Virtual Machine*

JaNi: *Java to Nios II compiler*

PC: *Program Counter*

UCP: Unidade Central de Processamento

UD: Uso-Definição

ULA: Unidade Lógico-Aritmética

RISC: *Reduced Instruction Set Computer*

SOC: *System On a Chip*

SOPC: *System On Programmable Chip*

Resumo

Durante esta pesquisa, foram investigados conceitos relacionados à computação reconfigurável, processo de compilação e funcionamento da compilação Java, especialmente no que se refere à manipulação de *bytecodes*. O principal objetivo é a elaboração conceitual de um compilador capaz de traduzir *bytecodes* de um aplicativo Java para código binário obediente ao conjunto de instruções do processador Nios II da Altera®, acompanhada de algumas implementações. Com este compilador, será possível a construção de programas para dispositivos que utilizem o processador Nios II, como dispositivos de sistemas embarcados, a partir da linguagem Java. Isto proporciona uma forma ágil para a elaboração de aplicativos para sistemas desse tipo.

Implementações relevantes foram desempenhadas de forma a comprovar o funcionamento do referido compilador. Tais implementações abrangeram suporte de *bytecodes* referentes a um subconjunto da linguagem Java, bem como a criação de estruturas de representação intermediária e mecanismos para geração de código final. Além disso, foram desenvolvidos um simulador e um montador de instruções do Nios II de forma a auxiliar no andamento do trabalho.

Como resultado, pôde-se verificar que é possível o desenvolvimento de tal ferramenta e que sua existência contribui para o desenvolvimento de outras ferramentas ou aplicativos inseridos no contexto de computação reconfigurável.

Abstract

During this research, concepts concerning about reconfigurable computing, compiling process and Java compilation were investigated, especially relating to *bytecode* generation. The goal of this research is twofold: first, the conceptual development of a compiler that is able to translate *bytecodes* from a Java application to binary code compliant with Altera™ Nios II processor instruction set, and second, provide a set of appropriate implementations for the proposed compiler. With this compiler, it will be possible to construct programs for devices that use Nios II processor, as embedded systems devices, from Java language, providing an agile manner for applications' building for this kind of systems.

Relevant implementations were performed to prove compiler's functionalities. Such implementations covered *bytecodes*' support to a Java language subset, as well as intermediate structures and final code generation mechanisms. Besides, a simulator and an assembler for Nios II instructions were developed to help this work's evolution.

As result, it was possible to verify that the development of such tool is feasible and contributes to the generation of other tools and applications inserted in the context of reconfigurable computing.

Capítulo 1 – Introdução

Neste capítulo são apresentados os principais aspectos que motivaram a implementação de um compilador a partir dos *bytecodes* da linguagem Java, bem como o porquê da escolha de tal linguagem. A escolha do processador-alvo também é justificada, mostrando como o trabalho poderá contribuir para os avanços das pesquisas referentes a sistemas embarcados, que são os maiores consumidores de processadores para arquitetura reconfigurável.

Na seção 1.1 é descrita a motivação do trabalho e na seção 1.2 o seu escopo. Na seção 1.3 são mostrados os objetivos do projeto e na seção 1.4 é apresentada uma breve descrição sobre como esta dissertação está organizada.

1.1 Motivação

Devido à necessidade crescente em se produzir sistemas para arquiteturas reconfiguráveis (CHEUNG *et. al.*, 2005; COMPTON; HAUCK, 2002; GONÇALVES, 2002), é relevante que se tenha um compilador para a linguagem Java, cuja demanda também é crescente, que tenha como linguagem-destino um conjunto de instruções de um dispositivo reconfigurável – *reconfigware*.

O Nios II é um processador de núcleo virtual, desenvolvido pela Altera® (ALTERA, 2007), que possui a propriedade de poder ser configurado para determinadas aplicações. É possível retirar alguns componentes do mesmo e incluir outros, bem como excluir ou incluir instruções em seu conjunto de instruções. Por

este motivo, o Nios II torna-se um dispositivo interessante no contexto da computação reconfigurável.

Considerando as funcionalidades e a importância do Nios II – e considerando que até o momento não foram encontradas implementações de compiladores Java para o mesmo – um trabalho que possa efetuar tal tarefa torna-se relevante para o desenvolvimento dos sistemas de computação reconfigurável.

Sendo a linguagem Java uma das mais usadas atualmente (TPCI, 2009), é importante suprir as necessidades de criação de sistemas reconfiguráveis sobre a mesma. Como Cardoso (2000) implementou um compilador cujo objetivo é a geração de circuito para um determinado FPGA, é importante que se gere código para o Nios II, como realizado por Duarte (2006) e, desta forma, o presente trabalho surge para preencher esta lacuna.

A opção pela geração de código para um processador de núcleo virtual é motivada pela relevância que tal tipo de dispositivo tem tido atualmente. Além disso, todas as estruturas de controle, como registradores e contadores, já estão implementadas, sendo necessário, apenas, suas configurações. Para o caso em que o compilador realiza síntese de circuitos digitais, todos os mecanismos de controle também devem ser gerados através do compilador.

Outros trabalhos têm sido realizados com o intuito de compilar um programa para arquitetura reconfigurável (CHATTOPADHYAY *et. al.*, 2008; LEE; MILNE, 2005; BECK; CARRO, 2005; CARDOSO; NETO, 1999, 2003), com destaque para o trabalho de Choi *et. al.* (2007), referente à construção de um AOTC (*Ahead-Of-Time Compiler*) para Java. A existência de tais trabalhos mostra a importância deste ramo de pesquisa para o cenário atual da computação.

1.2 Escopo

Segundo Bracha *et al.* (2005), Java é uma linguagem de propósito geral, relacionada com C e C++, porém, organizada de maneira diferente, sendo uma linguagem mais voltada para produção. Trata-se de uma linguagem fortemente tipada, cuja compilação consiste em traduzir programas para uma representação de códigos de *byte* (*bytecodes*) (BURDY; PAVLOVA, 2006; CIERNIAK; LI, 1997;

LAMBRIGHT, 1997; GOSLING, 1995) independentes da arquitetura da máquina. Assim, um programa escrito em linguagem Java é compilado para os conjuntos de instruções em forma de *bytecodes*, cuja definição pode ser encontrada no documento de Lindholm e Yellin (1999). A partir desses *bytecodes*, uma máquina virtual Java (*Java Virtual Machine* – JVM) executa o programa da forma descrita por Bracha *et al*. (2005) e Lindholm e Yellin (1999). Arquiteturas diferentes possuem implementações diferentes de máquinas virtuais. Isso permite compilar o programa apenas uma vez e executá-lo diversas vezes em arquiteturas distintas.

O processador Nios II é um processador RISC (*Reduced Instruction Set Computer*) de propósito geral, cujo sistema é equivalente a um micro-controlador com uma UCP (Unidade Central de Processamento) e uma combinação de periféricos e memória em um único *chip* (ALTERA, 2007). Com este processador, é possível a adição de instruções personalizadas ao seu conjunto de instruções quando se deseja buscar uma aceleração para determinado trecho de um algoritmo, o que permite maior flexibilidade e adição de novas funcionalidades à ULA (Unidade Lógico Aritmética) do Nios II (ALTERA, 2005).

Como o Nios II é um dispositivo configurável, efetuar a compilação de um programa tendo linguagem-alvo como sendo o conjunto de instruções do mesmo pode auxiliar na tarefa de construção de sistemas para arquitetura reconfigurável.

Sistemas de computação reconfigurável são sistemas que, ao mesmo tempo em que permitem a execução de instruções em plataforma de hardware, são capazes de ser modificados via software em tempo de execução de modo a se adequar à aplicação executada, garantindo flexibilidade (COMPTON; HAUCK, 2002; GONÇALVES, 2002).

De acordo com Ferrari e Skliarova (2004), apesar dos conceitos de arquitetura reconfigurável terem sido propostos na década de 60, as tecnologias que permitem sua execução na prática começaram a se tornar disponíveis apenas a partir dos anos 90 e, desde então, a computação reconfigurável (CHEUNG *et al.*, 2005) tem sido alvo de diversas pesquisas.

Um ambiente que explora o uso de dispositivos reconfiguráveis é o ARCHITECT+, derivado do ARCHITECT-R (GONÇALVES, 2002). O ARCHITECT+ é um ambiente para co-projeto de hardware/software em plataformas de FPGAs com aplicação em robótica móvel, sendo desenvolvido junto ao

Laboratório de Computação Reconfigurável do Instituto de Ciências Matemáticas e de Computação da USP, campus de São Carlos. A idéia do ARCHITECT+ é que um programa escrito em linguagem C ou Java seja compilado e que um SOC (*System-On-a-Chip*), implementado através de FPGAs, seja gerado para aceleração da aplicação.

Nos trabalhos de Duarte (2006) e Cardoso (2000) são implementados compiladores para *reconfigwares*. No primeiro, foi desenvolvido um *framework* que permite que um programa escrito em linguagem ANSI C possa ser compilado para o processador Nios II. Duarte (2006) descreve as implementações das etapas de compilação desde a análise léxica até a geração de código para o Nios II – como o trabalho foi direcionado à compilação de programa em C, não foi possível utilizar uma representação intermediária como a dos *bytecodes* de Java. Além disso, Duarte (2006) aborda assuntos como os grafos de representação intermediária utilizados, geração de código para algumas instruções e simulação para o Nios II.

Cardoso (2000) implementou um compilador de linguagem Java, a partir dos *bytecodes*, para um hardware reconfigurável constituído por um FPGA, como reportado parcialmente por Cardoso e Neto (1998). Cardoso (2000) aborda detalhadamente tópicos como a construção de modelos de representação intermediária, sendo discutidos os tipos de grafo utilizados, bem como suas otimizações, e dependências de dados e instruções. Entre as contribuições de Cardoso (2000), pode-se citar a possibilidade de compilação rápida para FPGAs com memórias acopladas, exposição dos vários fluxos de controle de um programa e escalonamento de operações. Tais técnicas desempenhadas são significativas para o trabalho com hardware reconfigurável, de maneira a utilizá-lo eficientemente.

1.3 Objetivos e metodologias

O objetivo deste trabalho é a implementação de um compilador, chamado JaNi (*Java to Nios II compiler*), para a arquitetura do processador Nios II da Altera® partindo-se dos *bytecodes* gerados da compilação de um programa em um subconjunto da linguagem Java, mostrado com fundo cinza na Figura 1.1. São

aproveitadas as fases de análise e geração de código intermediário (*bytecodes*), desempenhadas por um compilador *front-end* (BAL *et al.*, 2001; AHO *et. al.*, 2008).

A implementação, realizada em linguagem C (SHILDT, 1997), tem foco na geração dos grafos de representação intermediária das instruções (MUCHNICK, 1997) e tradução dos mesmos para instruções do Nios II.

O compilador proposto tem a tarefa de tomar como entrada os *opcodes* – identificadores de *bytecodes* – e montar, a partir dos mesmos, os grafos de representação intermediária. O projeto deverá fornecer ao ARCHITECT+ a opção de compilação Java para a síntese de alto nível de circuitos digitais.

Tipos de dados	Operações	Controle e outros mecanismos
boolean	/ % *	Tratamento de exceções
byte	++ --	Invocação de métodos
short	+ -	Criação de objetos
int	<< >>	Criação de arrays
char	|| &&	
long	| &	while, for, do while
float	^ ~	break, continue
double	> <	if
Referências a arrays	<= >=	
Referências a objetos	== !=	Conversões (*cast*)

Figura 1.1 Subconjunto de Java a ser suportado (*fonte*: CARDOSO, 2000).

1.4 Organização da dissertação

Os demais capítulos desta dissertação estão organizados como segue:

- Capítulo 2 – Análise dos *bytecodes* Java: são apresentadas no capítulo 2 as análises existentes para obtenção de informações do código sendo compilado. Com essas análises, é possível construir os grafos de representação intermediária com informações sobre o programa. Os grafos, a relevância de cada uma das análises e o modo com que o arquivo com os *bytecodes* é interpretado também são abordados;
- Capítulo 3 – Geração de código: no capítulo 3 são discutidas as operações realizadas quando já se tem os grafos de representação intermediária. Portanto, são levadas em conta funcionalidades dependentes da arquitetura, geração de código e propriedades do processador Nios II;

- Capítulo 4 – Trabalho realizado: nesse capítulo são apresentadas e descritas as implementações realizadas, com apresentação dos algoritmos utilizados para a construção dos grafos de representação intermediária. Também no capítulo 4 é apresentado o simulador construído para os testes sobre o JaNi;
- Capítulo 5 – Testes: o objetivo do capítulo 5 é apresentar exemplos de compilação sobre o JaNi, mostrando como a compilação funciona, bem como a efetividade das implementações realizadas e descritas no capítulo 4. Este capítulo proporciona uma visão mais detalhada sobre o compilador, com alguns detalhamentos mais aprofundados dos que os expostos no capítulo 4, devido ao seu cunho prático;
- Capítulo 6 – Conclusões: as principais conclusões, contribuições e considerações finais sobre todo o trabalho realizado.

Capítulo 2 – Análise dos *bytecodes* Java

Este capítulo tem como principal objetivo contextualizar o leitor sobre as operações a serem realizadas sobre os *bytecodes* de um aplicativo Java. Na seção 2.1 é descrito o processo de compilação em Java e a organização da representação de uma classe. Nas seções de 2.2 a 2.5 são mostradas as análises de fluxo e dependência sobre o código intermediário, a geração dos respectivos grafos e suas utilidades. Na seção 2.6 são apresentadas as considerações finais do capítulo.

2.1 Compilação Java

O processo de compilação de um programa escrito em Java pode ser dividido em duas fases, sendo a primeira de validação do programa e compilação do código fonte para o código de representação intermediária do Java, os *bytecodes*. A segunda refere-se à interpretação (através de uma JVM) ou compilação para uma arquitetura alvo. Tal processo é ilustrado na Figura 2.1.

Na subseção 2.1.1 é descrito brevemente como é organizado o arquivo de representação intermediária contendo os *bytecodes* Java.

2.1.1 Representação intermediária de Java

Em um projeto Java com diversas classes são criados vários arquivos de *bytecodes*, sendo um para cada classe do projeto, de maneira que todas as

informações referentes a cada classe sejam armazenadas em um mesmo local, tais como constantes, atributos e métodos, bem como dados sobre herança e classes abstratas.

Cada arquivo de representação de classe possui a extensão `.class` e contém as seguintes informações (LINDHOLM; YELLIN, 1999):

- Versão do arquivo;
- Tabela das estruturas referenciadas dentro da classe;
- *Flags* que identificam o tipo da classe;
- Referência para uma superclasse, se existir;
- Tabela de interfaces da classe;
- Tabela com a descrição dos campos da classe;
- Tabela de métodos e tabela de atributos da classe.

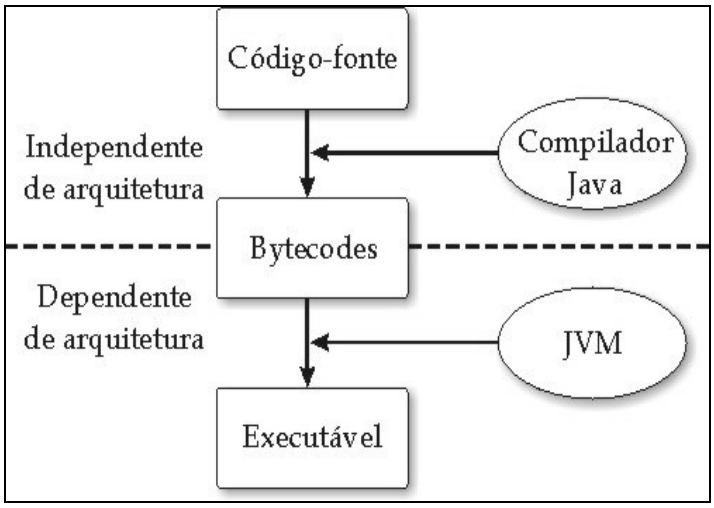

Figura 2.1 Ilustração do processo de compilação Java.

Dentro da tabela de métodos também consta a implementação do método, caso este não seja abstrato. É na implementação dos métodos que se localizam os *bytecodes*, já que é ali onde reside o código efetivamente executado no programa. O conteúdo de cada método é uma seqüência de números inteiros, *opcodes*, representando os *bytecodes*, que são estruturados para representar uma organização de instruções com existência de uma pilha de operandos (HANNA; HASKELL, 2006; HENESSY; PATTERSON, 2003).

Usualmente, toda a informação necessária para a execução de uma instrução de *bytecode* encontra-se em 1 byte, por isso o seu nome. Entretanto, há *bytecodes* que necessitam de parâmetros extras para sua execução, os quais são obtidos através da

leitura de bytes adjacentes. Por este motivo, pode-se dizer que existem *bytecodes* de comprimentos diferentes, sendo que esses comprimentos podem variar de 1 até 5, dependendo da instrução. Holst e Stephenson (2006; 2004) classificam os *bytecodes* em especializados e gerais, sendo que os especializados podem ser replicados através de outros *bytecodes* e os gerais possuem função unicamente desempenhada por eles.

2.2 Análise de fluxo de controle

Como grande parte das otimizações realizadas em um compilador depende do grau de conhecimento que se tem do programa que está sendo compilado, é necessário que haja técnicas e métodos que possam proporcionar conhecimento suficiente para a realização das otimizações. Boa parte deste conhecimento é obtida através da análise de fluxo de controle, que é capaz de resgatar o fluxo de controle hierárquico do programa (MUCHNICK, 1997) e fornecer informações preciosas para as análises subseqüentes, especialmente a análise de fluxo de dados.

A análise de fluxo de um programa (tanto controle quanto dados) pode ser realizada em três níveis, como é mostrado na Figura 2.2, considerando que bloco básico consiste em um trecho do código que não possui desvios nem destinos de desvios de outros trechos, ou seja, todas as instruções desde a primeira até a última são executadas. Caso a análise de fluxo seja dada no escopo de programa, será necessária a construção do grafo de chamadas de procedimentos, que mapeia as chamadas entre as rotinas do programa.

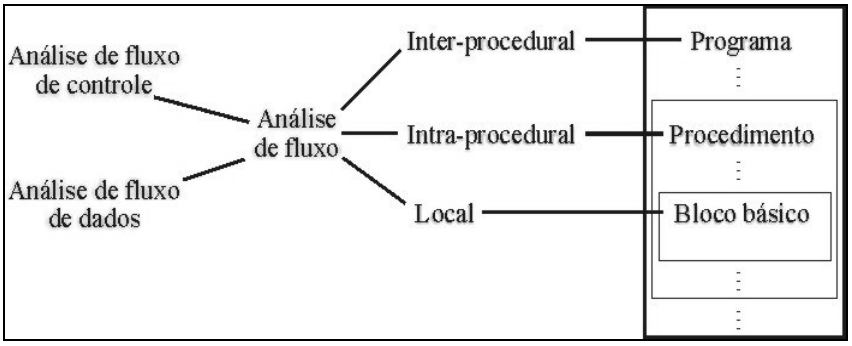

Figura 2.2 Classes de análise de fluxo (*fonte*: CHAPMAN; ZIMA, 1990).

De uma análise de fluxo de controle, deve surgir o grafo de fluxo de controle (GFC), que modela a estrutura de controle do programa, sendo que seus vértices

podem representar instruções do programa ou blocos básicos, e suas arestas a transferência de controle entre os vértices.

Exames sobre o grafo de fluxo de controle podem ser úteis para algumas otimizações, como, por exemplo, a eliminação de código-morto, em que basta um percurso sobre este grafo para determinar se algum de seus vértices não poderá ser atingido pelo fluxo do programa. Contudo, este processo não elimina todo código-morto, já que muitos trechos que nunca serão executados só podem ser encontrados analisando-se o fluxo de dados do programa.

Uma operação importante na análise de fluxo de controle é a identificação de ciclos dentro do grafo de fluxo de controle. Para tanto, fazem-se necessárias duas definições (CHAPMAN; ZIMA, 1990), a de dominância e a de laço, realizada com base na dominância:

- ✓ Definição 2.1 - <u>dominância</u>: Considerando N o conjunto dos vértices de um grafo, E o conjunto das arestas e s um vértice do grafo, seja G = (N, E, s) um grafo de fluxo e $n, n' \in$ N:
  - n **domina** n' ($n \leq n'$) $\Leftrightarrow$ cada caminho de s a n' contém n.
  - n **domina propriamente** n' ($n < n'$) $\Leftrightarrow$ $n \leq n'$ e $n \neq n'$.
  - n **domina diretamente** n', ($n <_d n'$) $\Leftrightarrow$ $n < n'$ e não existe um $n'' \in$ N tal que $n < n'' < n'$.
  - DOM(n) := $\{n'' : n'' \leq n\}$ é o conjunto de dominadores de n.
- ✓ Definição 2.2 - <u>laço</u>: Seja G = (N, E, s) um grafo de fluxo. Um subgrafo de fluxo G' = (N', E', s') de G é um laço com ponto de entrada s' se, e somente se:
  - Para todo $(n, n') \in$ E: $n' \in$ N' $\Rightarrow$ $n' = s'$ ou $n \in$ N', e
  - Para todo par de nós $n, n' \in$ N' existem caminhos não-triviais de n a n' e vice-versa.

A partir das definições 2.1 e 2.2, diz-se que uma aresta do grafo cujo destino domina sua origem é chamada de *back edge*. Os tipos de arestas em um grafo de fluxo podem ser identificados a partir da execução do algoritmo de busca em profundidade em grafos (CORMEN *et. al.*, 2001; MUCHNICK, 1997).

Um grafo de fluxo de controle também pode ser reduzido, ou seja, sofrer transformações que façam com que subgrafos se tornem nós, reduzindo, assim, o grafo de fluxo para grafos mais simples, o que pode facilitar a análise de fluxo.

De acordo com Muchnick (1997), há duas formas principais de se realizar a análise de fluxo sobre o grafo: análise de intervalos e análise estrutural.

Na análise de fluxo de controle, a análise de intervalos consiste na divisão do grafo de fluxo em regiões de diferentes tipos, formando os chamados nós abstratos. Um grafo de fluxo com nós abstratos é chamado de grafo de fluxo abstrato. As transformações realizadas na análise de intervalos formam a árvore de controle, em que a raiz representa todo o grafo, os nós intermediários (abstratos) representam partes do grafo e as folhas blocos básicos. A análise de intervalos trabalha basicamente com a redução de laços.

A análise estrutural é superior à análise de intervalos, principalmente porque identifica mais tipos de estruturas de controles além dos laços, e, conseqüentemente, sua árvore de controle é maior. A análise estrutural é realizada a partir da construção de uma árvore de espalhamento através da busca em profundidade no grafo de fluxo seguida de um percurso em pós-ordem visando à formação dos nós abstratos e, assim, construindo a árvore de controle.

2.3 Análise de fluxo de dados

O objetivo da análise de fluxo de dados é obter informações sobre como o (trecho de) programa analisado manipula os dados. Ela é efetuada a partir de inspeções sobre o grafo de fluxo de controle, examinando o fluxo de valores escalares através do programa.

Como para várias otimizações e para a geração de código final a análise de fluxo de dados é um fator limitante na transformação de código, a mesma deve ser conservativa no sentido de proibir alterações que possam deixar incorreto um código inicialmente correto.

Uma das aplicações mais importantes desta análise é na identificação do alcance de definições. Uma definição é uma atribuição de valor a alguma variável, e um uso é o acesso a um valor de variável definido anteriormente. O problema, então, é encontrar até que ponto uma definição de valor será usada e, dado um uso, quais definições de valor podem alcançá-lo. Desta maneira, torna-se possível determinar

quais objetos do programa[1] são válidos em um determinado ponto do programa ou rotina.

As informações de fluxo de dados referidas no parágrafo anterior podem ser representadas através das cadeias de definição-uso (DU) e uso-definição (UD) (CHAPMAN; ZIMA, 1990). As cadeias DU mapeiam uma definição a todos os usos alcançados por ela, e as cadeias UD mapeiam um uso a todas as definições que possam alcançá-lo. Uma *web* é a união máxima entre as cadeias DU, sendo muito importante para a fase de alocação de registradores do compilador, a qual é discutida na seção 3.4.

Na Tabela 2.1 são mostradas algumas outras aplicações da análise de fluxo de dados, também apresentadas por Muchnick (1997).

Tabela 2.1 Algumas otimizações relacionadas ao fluxo de dados.

Aplicação	**Descrição**
Expressões disponíveis	Determinando, para um dado ponto do programa, quando uma expressão já tiver sido calculada, ajudando a evitar recálculos.
Vida de variáveis	Fortemente associada à identificação das cadeias DU e UD, a fim de determinar quando um valor de variável é válido, sendo também essencial na fase de alocação de registradores.
Propagação de cópias	Consiste na eliminação de cópias desnecessárias entre variáveis.
Propagação de constantes	Eliminação de cálculos que sempre resultam em um valor constante.

Pode-se dizer que os problemas discutidos até aqui são problemas de fluxo de dados. Para a solução destes problemas, existem vários métodos e abordagens. As mais comuns são: abordagem iterativa, método baseado em árvore de controle utilizando análise estrutural e método baseado em árvore de controle utilizando análise de intervalos.

[1] Entenda-se constantes, variáveis, expressões, entre outros.

O método mais simples de implementar é a análise de fluxo de dados iterativa, sendo a mais utilizada segundo Muchnick (1997). O que é feito nesta abordagem é a definição de equações para modelar o fluxo de dados e realizar iterações sobre elas até que não haja alterações nos dados resultantes.

Tanto a análise estrutural quanto a análise de intervalos para fluxo de dados são mais difíceis de implementar do que a análise iterativa. Todavia, mudanças no código são mais bem recebidas, e a adaptação ao modelo gerado é mais fácil de ser feita, ao passo que na análise iterativa normalmente todo o algoritmo deve ser executado novamente. Muchnick (1997) e Chapman e Zima (1990) discutem com mais detalhes a análise de fluxo de dados, mostrando como as equações são modeladas.

A análise de fluxo de dados também poderá produzir um grafo como saída, o grafo de fluxo de dados (GFD), no qual as arestas representam a transferência de dados de um bloco básico (ou instrução, o que é mais comum para fluxo de dados) a outro. Além disso, pode-se construir um grafo para cada variável escalar, caso haja a necessidade de examinar alguma variável em específico.

2.4 Análise de dependência de controle

As análises de dependência, seja de controle ou de dados, servem principalmente para o escalonamento de instruções do programa e para otimizações em relação à utilização da memória *cache*.

Para discutir a análise de dependência de controle, duas definições formais são necessárias (CHAPMAN, ZIMA, 1990):

- ✓ Definição 2.3 - <u>pós-dominância</u>: É uma relação reflexiva à dominância para o grafo invertido. Seja G = (N, E, *s*, *t*) um grafo de fluxo de **única saída**, em *t*, e *n*, *n'* $\in$ N. Diz-se que *n* é **pós-dominado** por *n'* se, e somente se:
  - $n \neq n'$, e
  - Todo caminho de *n* a *t* contém *n'*.
- ✓ Definição 2.4 - <u>dependência de controle</u>: Seja G = (N, E, *s*, *t*) um grafo de fluxo de única saída, em *t*, e *n*, *n'* $\in$ N. Diz-se que *n'* é **dependente de controle** de *n* se, e somente se:

- Existe um caminho não-trivial de *n* a *n'* tal que todo nó *n''* no caminho ($n'' \neq n, n'$) é pós-dominado por *n'* e
- *n* não é pós-dominado por *n'*.

A dependência de controle (definição 2.4) ocorre quando uma instrução é capaz de determinar se outra será executada ou não. Ou seja, é uma restrição quanto à ordem de execução das instruções do programa, o que a caracteriza como sendo um forte limitante na fase de reestruturação do programa.

O grafo de dependência de controle (GDC) deve ser gerado nesta fase de análise do programa-fonte compilado. Os vértices do GDC geralmente representam instruções da linguagem intermediária e as arestas a existência de dependência de controle entre duas instruções. Este grafo pode ser gerado tanto para o programa ou rotina quanto para cada bloco básico[2], o que permite que a análise de dependência possa ser aplicada a vários níveis do código.

2.5 Análise de dependência de dados

A redação de programas seqüenciais impõe uma ordem linear de execução das instruções. Muitas vezes, esta ordem imposta é mais proibitiva do que o necessário para que o programa funcione corretamente. Há expressões e valores que podem ser calculados em uma ordem diferente da que foi especificada, sem que o resultado final seja comprometido.

É nesse contexto que as relações de dependência de dados (KYRIAKOPOULOS; PSARRIS, 2005) aparecem no sentido de relaxar a ordem linear de execução das instruções, de modo a possibilitar uma reestruturação no programa que viabilize sua execução concorrente. Existem 3 relações de dependência de dados básicas dentro de um programa, a saber (HWANG, 1993; CHAPMAN; ZIMA, 1990):

- <u>Entrada ou fluxo</u>: quando a saída de um bloco é necessária para a entrada de outro;

---

[2] No caso do grafo ser gerado para blocos básicos, ele terá a propriedade de ser direcionado e acíclico, um DAG (*Directed Acyclic Graph*), o que facilita a aplicação de alguns algoritmos sobre dígrafos.

- Anti-dependência: quando a saída de um bloco for entrada para outro que venha antes na ordem linear de execução;
- Saída: quando há intersecção das saídas de dois blocos de instruções.

As duas últimas relações de dependência estão mais relacionadas a linguagens imperativas, o que é o caso de Fortran e C, que permitem o reuso de memória através do acesso de variáveis. Assim, dependendo da semântica e do paradigma da linguagem de programação, tais relações de dependência podem deixar de existir e, por este motivo, são chamadas de dependências artificiais.

O grafo de dependência de dados (GDD) é construído a partir da verificação de dependência entre cada par de instruções do programa, em que cada vértice representa uma instrução e as arestas representam a existência de dependência entre as instruções, sendo direcionadas no sentido da ordem linear de escrita dessas instruções. As arestas podem ser rotuladas de modo a fornecer informações sobre qual o tipo de dependência existente.

Outro tipo de análise de dependência de dados é a verificação de dependência entre laços, mais precisamente, entre os dados manipulados entre as iterações dos laços. Para este tipo de análise, considera-se principalmente as variáveis e expressões subscritas de índices de vetores.

Com relação às dependências entre iterações de laços (BIRCH *et. al.*, 2006; PADUA; WOLFE, 1986), existem vários métodos para teste, um deles é o teste das equações restritas de Diophanto (MUCHNICK, 1997; CHAPMAN; ZIMA, 1990). Neste teste, modela-se o laço como uma equação em relação aos índices subscritos dos vetores que estão sendo utilizados, depois resolve-se a equação dentro do intervalo delimitado pelos limites do laço. Se houver uma ou mais soluções, significa que existe dependência entre iterações do laço.

Outro teste de dependência bastante utilizado é o GCD (*Greatest Common Divisor*), desenvolvido em 1976, que utiliza conceitos de separabilidade aplicados a uma operação de máximo divisor comum.

A análise de dependência de dados é a mais importante tanto para as operações de paralelização quanto de vetorização de um programa, pois é capaz de fornecer restrições que impõem execução linear estritamente onde é necessário para que os resultados do programa sejam sempre válidos.

2.6 Considerações finais

O propósito deste capítulo foi apresentar os mecanismos necessários para o conhecimento do programa que deverá ser compilado. Inicialmente, foi mostrado como um executável pode ser gerado a partir da representação intermediária utilizada na linguagem Java, obtendo-se as informações necessárias e implementações a partir dos *bytecodes*. Neste contexto, Cherem e Rugina (2005) construíram um verificador de *bytecodes* com o objetivo de validar a alocação de objetos de acordo com a região do código em que são utilizados.

Como as otimizações de código exigem conhecimento do programa a ser compilado, são necessárias análises neste sentido. Foram apresentados os quatro tipos principais de análise de código para obtenção de informações do programa: fluxo de controle, fluxo de dados, dependência de controle e dependência de dados. Os dois últimos tipos estão mais relacionados a escalonamento de instruções – especialmente no caso de arquiteturas vetoriais e paralelas (HWANG, 1993) – e otimizações da utilização da memória *cache* do processador alvo.

Os grafos de dependência de controle e de dependência de dados podem ser representados através de um único grafo, o grafo de dependência de programa (GDP), projetado para uso em otimizações (MUCHNICK, 1997) e também utilizado por Duarte (2006), Grimmer, Hammer e Krinke (2006) e Edwards e Zeng (2007).

Portanto, diferente das análises léxica, sintática e semântica (AHO *et. al.*, 2008), utilizadas para validação do código, as análises de fluxo e dependência servem para um conhecimento mais profundo das operações realizadas por um programa, buscando o fornecimento de informações úteis na fase de otimização.

Capítulo 3 – Geração de código

Este capítulo apresenta os aspectos relacionados à geração de código para a arquitetura alvo. Na seção 3.1 são mostrados alguns aspectos e aplicações de sistemas com arquitetura reconfigurável. Na seção 3.2 é dada uma breve revisão sobre o processador Nios II, da Altera®, cujo conjunto de instruções é o destino da compilação, sendo que na seção 3.3 são apresentados mais detalhes sobre tal conjunto de instruções. Na seção 3.4 o problema da alocação de registradores na fase de geração de código é discutido e na seção 3.5 as conclusões do capítulo são mostradas.

3.1 Computação reconfigurável

Compton e Hauck (2000) mencionam duas formas de se executar um programa. A primeira delas é através dos processadores ASIC (*Application Specific Integrated Circuit*), os quais são especificamente elaborados para uma dada aplicação. A outra é utilizar um microprocessador genérico e especificar a aplicação através de software, ao invés de hardware.

A principal diferença entre as maneiras colocadas no parágrafo anterior é que, utilizando ASICs, a execução se dá de forma mais rápida, ao passo que a execução através de microprocessadores é mais flexível, permitindo que programas distintos sejam executados, o que acarreta a perda de desempenho em relação aos ASICs (MARQUES, 2004).

A principal desvantagem dos ASICs é o fato de não permitirem modificações no aplicativo que executam, já que este está gravado em hardware e só poderá ser trocado, não alterado. A computação reconfigurável surgiu para combinar o desempenho dos ASICs e a flexibilidade dos microprocessadores (NATARAJAN; PERINBAM; RAMADASS, 2007; COMPTON; HAUCK, 2002; 2000), sendo acompanhada pelas arquiteturas reconfiguráveis, ou *reconfigware*.

A base para a existência da computação reconfigurável atualmente são os FPGAs (AHMADINIA *et. al.*, 2007; ROSE; STEFFAN; YIANNACOURAS, 2007; DUARTE, 2006; MARQUES, 2004; COMPTON; HAUCK, 2002; 2000), que são circuitos reprogramáveis, possibilitando a flexibilidade e o desempenho propostos pela computação reconfigurável.

3.1.1 Considerações sobre hardware

Compton e Hauck (2000) também discutem alguns aspectos relacionados a hardware para computação reconfigurável, como nível de acoplamento, granulação e roteamento entre blocos lógicos do circuito.

O nível de acoplamento refere-se ao grau de ligação entre os componentes do hardware. Há várias maneiras de realizar o acoplamento de componentes do circuito, dependendo do que se deseja atingir. Em um acoplamento mais forte, tem-se uma sobrecarga de comunicação menor, ao passo que em um acoplamento mais fraco o paralelismo é facilitado.

A granulação refere-se à complexidade lógica embutida nas células do FPGA (DUARTE, 2006), podendo ser fina, média ou grossa, sendo que blocos finos são úteis para aplicações que trabalham com manipulação de bit (criptografia e processamento de imagens) e os grossos – e também médios – para aplicações que trabalham com caminhos de dados, como operações aritméticas sobre valores de 16 ou 32 bits.

Com relação ao roteamento de informações entre os blocos lógicos, tem-se a importância da área do hardware, uma vez que a complexidade das ligações afeta o tamanho do *chip* utilizado, o que pode gerar outros problemas como aumento no consumo de energia. Esse pode ser um problema em situações nas quais a arquitetura

é projetada para servir de base a um sistema embarcado (COSTA *et. al.*, 2007; NATARAJAN; PERINBAM; RAMADASS, 2007; MASKELL; OLIVER; SHIAN, 2005).

3.1.2 Considerações sobre software

Os benefícios da computação reconfigurável podem até mesmo ser ignorados pelos desenvolvedores de software, de maneira que possa ficar transparente para os mesmos a arquitetura que está sendo utilizada (COMPTON; HAUCK, 2000).

Por este motivo, as fábricas que produzem dispositivos reconfiguráveis também, geralmente, disponibilizam aplicativos que auxiliam na criação do circuito a ser gerado, o que é o caso da Altera® (ALTERA, 2007; 2005).

Em outros casos, o próprio compilador do programa em questão poderá identificar as implementações em hardware que mais se adequam ao software e se encarregar de gerar o circuito para o mesmo. Para as circunstâncias em que o dispositivo reconfigurável possui um conjunto de instruções, a compilação passa a ser semelhante ao processo de compilação padrão.

3.1.3 Compilação para arquiteturas reconfiguráveis

A compilação para *reconfigware* segue o mesmo esquema das compilações comuns (AHO *et. al.*, 2008), possuindo as fases de análise léxica, sintática e semântica, para verificar se o programa obedece às regras da linguagem. A partir daí tem-se a geração da representação intermediária, otimizações de código e, por fim, criação do código executável.

Os grafos de representação intermediária discutidos no capítulo 2 são de grande utilidade na compilação para *reconfigware*, uma vez que existe sempre o interesse em construir códigos pequenos e eficientes, pois sistemas que utilizam *reconfigware* geralmente necessitam do uso de pouco espaço, tal como os sistemas embarcados (MARQUES, 2004).

Especialmente, o grafo de dependência de dados é muito útil para especificação de um circuito digital, em que os pontos de seleção identificados

podem ser transformados em multiplexadores (CARDOSO, 2000). Esta propriedade é indispensável quando o compilador deve gerar a especificação de um circuito para a execução do programa. No caso de processadores virtuais, o processo de geração de executável é semelhante quando se está utilizando processadores reais.

Alguns compiladores para *reconfigware* foram identificados até o momento, os quais são listados a seguir:

- Garp: arquitetura proposta em 2000 com o objetivo de acelerar a execução dos ciclos de programas, permitindo a configuração do *reconfigware* em tempo de execução, buscando a otimização do hardware. Ele combina a arquitetura MIPS com dispositivo reconfigurável e possui um compilador para linguagem C, o Garpcc (CALLAHAN; HAUSER; WAWRZYNEK, 2000);
- Galadriel e Nenya: compiladores com intuito de efetuar a síntese de alto nível de circuitos digitais a partir de um programa escrito em linguagem Java. O compilador Galadriel transforma os *bytecodes* na representação intermediária e o Nenya efetua a síntese. Ambos foram propostos por Cardoso (2000);
- Spark: realiza a síntese de alto nível a partir de um programa escrito em linguagem C. Seu modo de funcionamento assemelha-se ao dos compiladores Galadriel e Nenya;
- Mentor Graphics: ferramenta proprietária que auxilia no desenvolvimento de aplicativos para algumas plataformas, sendo o Nios II uma delas. Trata-se de uma ferramenta robusta, porém com alto custo (MENTOR, 2009);
- DRESC: seu nome é uma sigla para *Dynamically Reconfigurable Embedded System Compiler*. Trata-se de um compilador para uma nova arquitetura reconfigurável, denominada ADRES (*Architecture for Dynamically Reconfigurable Embedded Systems*), permitindo a compilação de um programa escrito em linguagem C para esta plataforma (SOUDRIS; VASSILIADIS, 2007);
- Molen: compilador que tem como alvo a arquitetura reconfigurável Molen, capaz de traduzir programas escritos em linguagem C, possuindo um fluxo integrado para geração de *bitstream* para o processador reconfigurável (BERTELS; PANAINTE; VASSILIADIS, 2007);

- Compilador PARS: PARS é uma arquitetura reconfigurável de propósito geral capaz de executar vários programas evitando problemas com tamanho dos programas através de reconfiguração. Juntamente com esta arquitetura, foi desenvolvido um compilador que se propõe a gerar código de alta qualidade em tempo hábil (HADA *et. al.*, 2007);
- CHiMPS: tem o intuito de permitir compilação tendo como alvo plataformas híbridas UCP-FPGA, de maneira a abstrair do programador detalhes referentes à movimentação de dados e à separação de memórias. Inicialmente suporta a linguagem C como fonte e também modelos com *cache* e memórias compartilhadas (BENNETT *et. al.*, 2008);
- TASKING VX-*toolset*: ferramenta proprietária produzida pela Altium® com o objetivo de proporcionar a compilação de aplicativos escritos em linguagens C ou C++ para o Nios II. Nesta ferramenta, estão inclusos *plugin* para Eclipse (ECLIPSE, 2009), montador com pré-processador de macros, bibliotecas – linguagem C, tempo de execução e ponto flutuante – e também um *linker* (TASKING, 2009).

Além desses, pode-se citar também os trabalhos de Budiu *et. al.* (2000) – um compilador voltado a *pipelines* reconfiguráveis – e de Bachmann *et. al.* (2000), que desenvolveu um compilador de MATLAB para arquitetura reconfigurável.

3.2 O processador Nios II

Como já mencionado na seção 1.2, o processador Nios II possui arquitetura RISC, é para propósito geral e "customizável", de maneira a possibilitar a inserção e/ou remoção de seus componentes e de instruções do seu conjunto de instruções. Na Figura 3.1 é mostrado um exemplo de como o núcleo do Nios II pode ser aplicado em um sistema computacional, relacionando-se com os demais componentes.

Na Figura 3.2 é apresentado um diagrama de blocos para o núcleo do processador, o qual é um projeto de hardware que implementa instruções do Nios II.

Uma das principais particularidades do Nios II é a sua capacidade de proporcionar flexibilidade para os projetistas de hardware, sendo que os projetistas

de software não são afetados pelos detalhes de implementação do dispositivo (ALTERA, 2007).

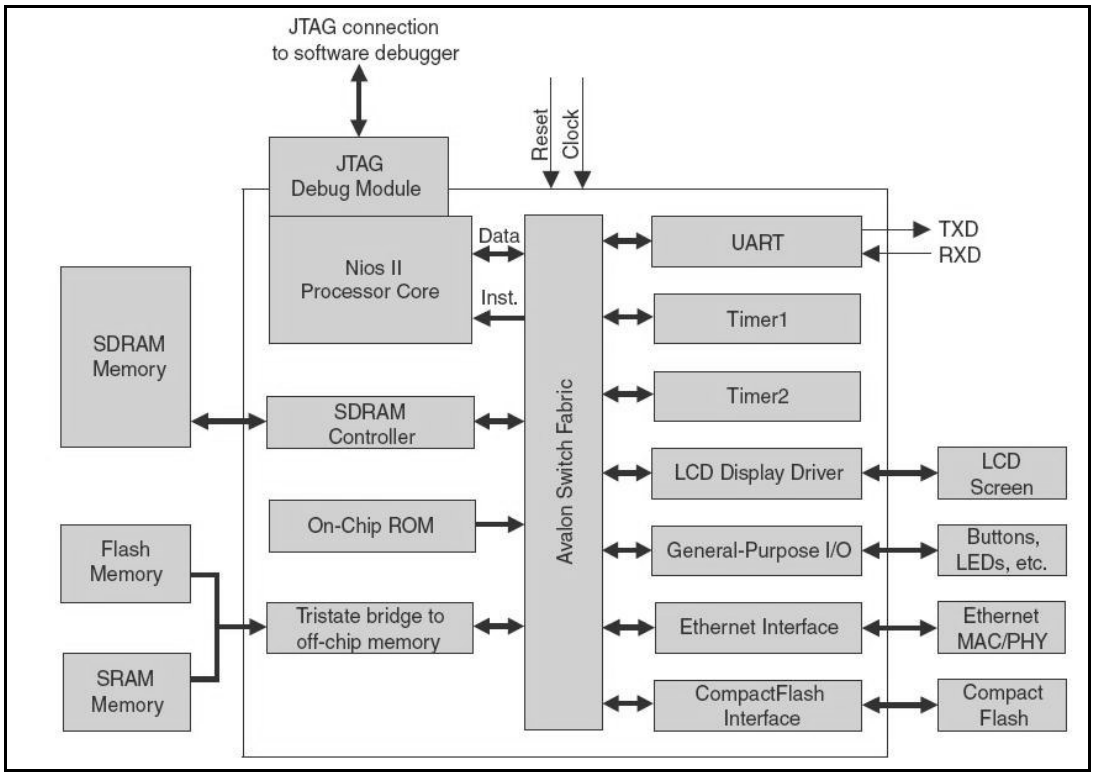

Figura 3.1 Exemplo de sistema com processador Nios II (*fonte*: ALTERA, 2007).

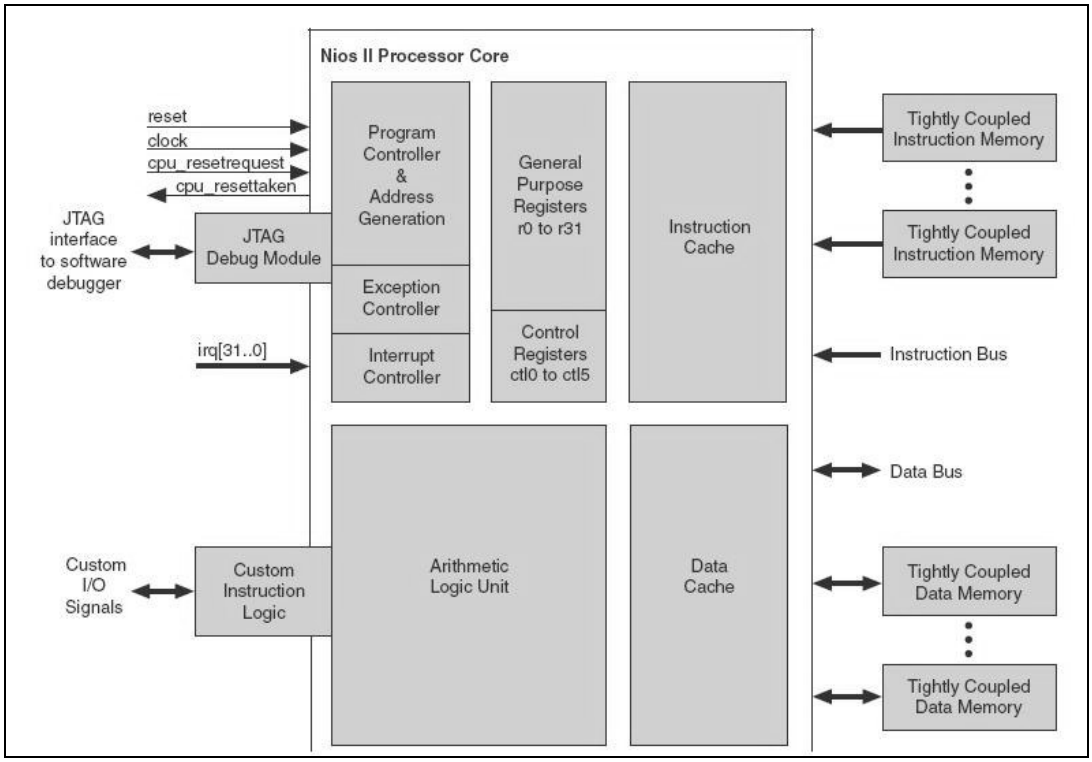

Figura 3.2 Diagrama de blocos para o núcleo do Nios II (*fonte*: ALTERA, 2007).

A classificação do Nios II como um dispositivo configurável significa que novos atributos (como, por exemplo, instruções personalizadas) podem ser adicionados ou retirados do mesmo, caso haja a necessidade de que isso seja feito para aumento do desempenho de um sistema ou por algum outro motivo conveniente ao projetista. O termo "*soft-core*" é utilizado para descrever o núcleo que o Nios II possui – projetado em forma de software, e não hardware – permitindo que o mesmo possa ser mapeado para qualquer família de FPGA da Altera®. Por este motivo, os projetistas de software podem utilizar um simulador do conjunto de instruções do Nios II antes que a configuração final da arquitetura seja estabelecida. As instruções personalizadas podem ser definidas em funções de uma linguagem de programação, retirando do desenvolvedor de software a necessidade de conhecimento de uma linguagem de baixo nível (ALTERA, 2007).

A geração automatizada do sistema pode ser propiciada pela ferramenta *SOPC Builder* da Altera®. Esta ferramenta é capaz de automatizar todo o processo de configuração das características do processador, gerando o projeto de hardware a ser programado em um FPGA. Desta maneira, sistemas inteiros de processador podem ser criados sem a necessidade de qualquer descrição em HDL (*Hardware Description Language*). A partir daí, o projeto pode ser programado em um dispositivo físico, de onde não poderá mais ser modificado (ALTERA, 2007).

3.3 Conjunto de instruções do processador Nios II

O conjunto de instruções do processador Nios II possui três categorias, nas quais cada instrução se encaixa em apenas uma delas (ALTERA, 2007), porém todas são representadas através de palavras de 32 bits. Nas subseções de 3.3.1 a 3.3.3 cada uma delas é detalhada.

Na seção 3.3.4 são apresentados os conceitos de pseudo-instruções existentes para o Nios II, os quais devem ser considerados tanto na elaboração de compiladores para este dispositivo quanto na confecção de simuladores.

3.3.1 Instruções *I-Type*

As instruções *I-Type* possuem formato de palavra com 6 bits para identificar o *opcode* da instrução, 16 bits para um valor imediato (o que caracteriza este tipo), e mais dois campos de 5 bits cada, contendo identificação para dois registradores.

Geralmente, neste tipo de instrução, os valores do primeiro registrador (registrador A) e o valor imediato formam operandos, e o outro registrador o destino do resultado de tal operação.

Dentre as instruções que se encaixam neste tipo tem-se as de comparação lógica, em que o conteúdo de um registrador é comparado com um valor imediato e o resultado é armazenado em outro registrador, bem como instruções de soma, subtração e similares, em que um operando é imediato.

3.3.2 Instruções *R-Type*

A diferença entre o tipo de palavra que caracteriza instruções *R-Type* do que caracteriza as *I-Type* está no fato de que o campo de 16 bits para valor imediato é dividido em dois, um contendo 5 bits, referente ao endereço de um terceiro registrador, e outro com 11 bits, relativo à operação que será realizada sobre os registradores.

O campo de 11 bits existe com a finalidade de estender o campo de *opcode*. Através do campo de *opcode* comum, podemos distinguir qual instrução *I-Type* ou *J-Type* é representada ou então simplesmente informar que a instrução é *R-Type* e, neste caso, o campo de extensão de *opcode* é necessário.

Entre as instruções *R-Type* existentes, podemos destacar as de comparação entre valores de registradores, além das aritméticas.

3.3.3 Instruções *J-Type*

Uma instrução *J-Type* é representada através de uma palavra com apenas 2 campos, sendo um de 6 bits para o *opcode* e o outro com 26 bits para uma posição de dados.

A única instrução deste tipo existente no Nios II é a instrução `call`, responsável por transferir a execução do programa para o endereço especificado, salvando o contador de programa em um registrador específico.

3.3.4 Pseudo-instruções

As pseudo-instruções são instruções existentes no conjunto de instruções do Nios II que não possuem implementação própria. Elas não têm *opcode* definido, pois são efetivamente implementadas por meio de instruções comuns.

Como exemplo, temos a pseudo-instrução `nop`. A implementação desta pseudo-instrução é conseguida a partir da instrução `add r0, r0, r0`, a qual possui mesmo efeito prático de `nop`. O mesmo ocorre nas instruções de comparação, sendo que apenas parte delas possuem implementação específica e as outras são pseudo-instruções, visto que é possível conseguir o mesmo efeito esperado manipulando-se as demais instruções dessa categoria.

3.4 Alocação de registradores

No momento da tradução do código de representação intermediária otimizado para o conjunto de instruções do processador-alvo, é necessário que se realize a alocação de registradores.

Esta fase da compilação mapeia as variáveis escalares que existem no programa para os registradores do processador, de forma que, quando for feita a referência a uma determinada variável, seja necessário apenas acessar o seu valor no registrador. Sua importância reside principalmente no fato de os processadores não possuírem registradores suficientes para armazenar todas as variáveis do programa e, sendo assim, aquelas que não foram mapeadas a registradores devem ser buscadas na memória.

Por este motivo, deve-se ter uma política que gerencie a alocação de um número **n** de variáveis do programa (ou rotina) aos **k** registradores do processador-alvo. Também por isso, diz-se que a alocação de registradores está incluída na etapa

de otimização de código, pois influencia diretamente o tempo de execução do programa gerado.

A principal maneira de efetuar alocação de registradores é através da coloração dos chamados grafos de interferência. Como a coloração de grafos é um problema NP-completo, são necessárias heurísticas para a solução de tal problema (MUCHNICK, 1997).

O algoritmo mais comum para alocação de registradores foi proposto por Chaitin e Briggs, no ano de 1981, sendo descrito por vários autores, como Muchnick (1997), Goldstein e Koes (2006), e Cooper, Dasgupta e Eckhardt (2005).

O algoritmo de Chaitin-Briggs começa com a construção do grafo de interferência, geralmente para uma rotina. Este grafo é construído com informações da análise de fluxo de dados referentes a uso e definição de variáveis e, desta forma, consegue determinar quais variáveis estão mutuamente vivas. Esta informação é armazenada no grafo, em que os vértices representam as variáveis e as arestas a existência de interferência entre as mesmas, ou seja, o fato de elas estarem simultaneamente vivas em um dado intervalo do programa.

A partir daí, são seguidos os algoritmos mostrados na Figura 3.3 e na Figura 3.4, em que *spill cost* refere-se ao custo de se deixar uma variável sem registrador alocado, ou seja, qual o custo associado de se trazer esta variável da memória quando for necessária sua utilização. Este valor varia, pois, por exemplo, variáveis de controle em laços têm custo maior já que são acessadas várias vezes, a cada iteração. Para variáveis comuns este custo é menor, visto que são utilizadas poucas vezes e demandariam um número menor de acessos à memória.

O termo *spill code* diz respeito ao código extra necessário para buscar e armazenar um valor na memória.

```
1: Repetir:
2:   Remover e empilhar os nós com grau menor que k;
3:   Se não existirem mais vértices desse tipo:
4:     Escolher nó com menor spill_cost e colocá-lo
       na pilha;
5: Até que todos os nós estejam na pilha e o grafo
   vazio;
6: Para cada nó desempilhado:
7:   Recolocá-lo no grafo e tentar atribuir uma cor;
8:   Se não conseguir, deixar o nó sem cor;
```

Figura 3.3 Algoritmo do procedimento colorir do algoritmo de Chaitin-Briggs.

```
1: Construir o grafo de interferência;
2: Remover cópias entre registradores caso eles não
   interfiram e retornar a 1 se ocorreram remoções;
3: Calcular o spill_cost para cada nó do grafo;
4: Ordenar os nós segundo o spill_cost de cada um;
5: Acionar procedimento colorir();
6: Inserir código extra (spill_code) para os nós que
   não foram coloridos e retornar a 1. Se todos os
   nós foram coloridos, terminar.
```

Figura 3.4 Algoritmo clássico de Chaitin-Briggs para alocação de registradores.

O algoritmo proposto por Chaitin-Briggs apresenta complexidade quadrática, sendo que a fase de construção do grafo de interferência é a que demanda mais tempo de execução para ser desempenhada. Pode-se considerar este algoritmo como sendo clássico, visto que, além de ser amplamente utilizado, serve de base para a criação de outros algoritmos, como, por exemplo, os propostos por Cooper e Dasgupta (2006) para compilação em tempo de execução e Holloway, Ramsey e Smith (2004).

3.5 Considerações finais

Foram apresentados, neste capítulo, os aspectos referentes à compilação para arquitetura reconfigurável, com ênfase no processador-alvo e na geração de código.

Primeiramente, foi realizado um breve resumo sobre computação e arquiteturas reconfiguráveis, com conceitos chave que servem de base para contextualizar o restante do capítulo. As duas seções seguintes tiveram a finalidade de descrever a arquitetura do processador Nios II da Altera®.

Por fim, foi feita uma explanação sobre a etapa de alocação de registradores de um compilador, mostrando como a mesma é relevante para a geração de código eficiente.

Capítulo 4 – Trabalho desenvolvido

Neste capítulo são apresentadas descrições sobre o que foi desenvolvido ao longo do presente trabalho. Na seção 4.1 é mostrado o modelo conceitual do compilador desenvolvido. Nas seções de 4.2 a 4.6 são apresentados detalhes de implementações dos módulos introduzidos por meio da seção 4.1. Na seção 4.7 é descrita a implementação de um simulador para realização dos testes. A implementação de um montador é delineada na seção 4.8 e na seção 4.9 são apresentadas as considerações finais sobre o capítulo.

4.1 Arquitetura do compilador

O compilador construído tem como entrada um arquivo de representação intermediária de instruções, gerado por qualquer compilador *front-end* que segue os padrões Java (LINDHOLM; YELLIN, 1999). Neste arquivo, deverá haver a descrição da classe principal e do método principal desta classe. O compilador é responsável por identificar o código deste método e por processá-lo nas etapas subseqüentes.

É importante ressaltar que o objetivo do compilador JaNi é ser capaz de compilar um projeto codificado em linguagem Java, no entanto, o escopo do presente trabalho atém-se na construção dos módulos principais do compilador, permitindo que os mesmos possam ser expandidos em trabalhos futuros.

O código identificado do método sendo compilado encontra-se sempre no formato de código intermediário de um compilador Java, ou seja, *bytecodes*.

Portanto, é necessária a existência de uma etapa que processe os *bytecodes* tendo em vista a geração de código intermediário do JaNi. Após a fase de processamento dos *bytecodes*, é gerado um código de representação intermediária baseado no conjunto de instruções do processador Nios II. Desta forma, a única tarefa da etapa de geração de código final consiste em construir um arquivo com a representação binária das instruções de código intermediário, sobre o qual são construídos os grafos de representação intermediária.

O esquema conceitual do compilador é apresentado na Figura 4.1. Nele podem ser visualizadas as camadas do JaNi, bem como seus dados de entrada e de saída.

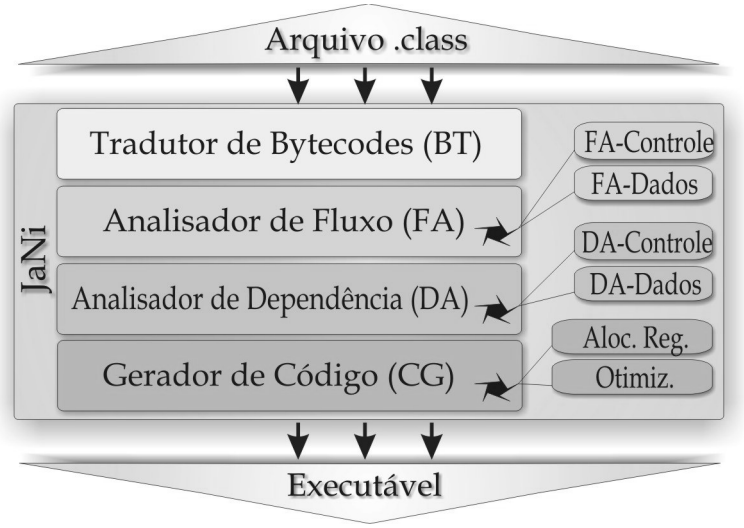

Figura 4.1 Esquema conceitual do compilador.

A partir do sentido das setas na Figura 4.1, pode-se observar como o fluxo de informações do programa sendo compilado obedece à disposição das camadas, na ordem apresentada. Esse é o esquema conceitual do compilador, sendo que o mesmo foi codificado em linguagem C-ANSI, com artifícios de programação para simular a implementação orientada a objetos.

A modelagem do compilador pode ser visualizada no Apêndice A, no qual é apresentado o modelo conceitual, contemplando os módulos em nível de desenvolvimento e suas interações. Também podem ser vistos os diagramas de casos de uso e de seqüência, que possuem o objetivo de esclarecer a maneira como o JaNi deve ser enxergado de maneira externa e as seqüências de operações realizadas para que o executável seja gerado.

4.2 Módulo reconhecedor de entrada

Conforme já mencionado, a entrada para o JaNi é o arquivo intermediário de compilação Java referente a uma classe, o qual possui a extensão `.class`[3]. Este arquivo passa por uma etapa de pré-processamento, para que os *bytecodes* referentes ao método principal possam ser reconhecidos. Essa etapa de pré-processamento é realizada dentro do módulo reconhecedor de entrada, conceitualmente localizado entre o arquivo `.class` e a camada BT (*Bytecodes Translator*), na Figura 4.1.

Além de reconhecer os *bytecodes* a serem processados pela etapa seguinte, o reconhecedor de entrada é capaz de interpretar outros dados contidos no arquivo de entrada, como o cabeçalho e as informações gerais sobre as estruturas da classe. A tradução do cabeçalho da classe é imprescindível para a identificação dos *bytecodes* pertencentes ao método que será compilado.

Este arquivo é lido byte a byte, sendo que a semântica de cada conjunto de bytes lidos é determinada na especificação do arquivo `.class` de Java (LINDHOLM; YELLIN, 1999). Os dados obtidos desse arquivo são apresentados por meio de um arquivo de saída do compilador, e os *bytecodes* encontrados são enviados para o módulo tradutor.

Na Figura 4.2 é apresentado um esquema para exibir conceitualmente as tarefas realizadas pelo reconhecedor de entrada.

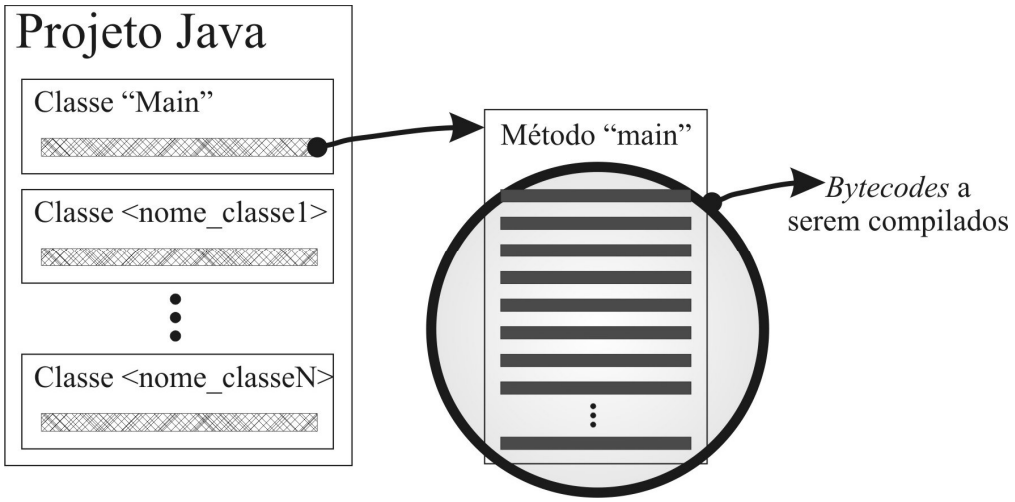

Figura 4.2 Funcionamento do módulo reconhecedor de entrada.

[3] Como o JaNi atualmente apenas trata uma classe, somente este arquivo é usado. No caso do compilador passar a tratar projetos inteiros, a entrada deverá ser um conjunto de arquivos `.class`.

4.3 Módulo tradutor de *bytecodes*

O módulo tradutor de *bytecodes* (*Bytecodes Translator* – BT) tem a tarefa de receber uma seqüência de *bytecodes* do módulo reconhecedor e gerar o código intermediário para o compilador.

O tradutor de *bytecodes* passa por cada um dos *bytecodes* existentes na seqüência recebida do módulo anterior, simulando sua execução. No entanto, ao invés de produzir os resultados, ele gera as instruções que pertencerão ao código intermediário. Desta maneira, os operandos da pilha de operandos utilizada na representação intermediária de Java são transformados em variáveis. Depois disso, é buscada a instrução, ou seqüência de instruções, do código intermediário que mais se adequa à operação desempenhada pela instrução de *bytecode* sendo processada.

Como exemplo, pode-se citar a operação de adição. Na representação de *bytecodes*, os dois operandos devem ser desempilhados da pilha de operandos e, após a operação aritmética ter sido realizada, o resultado da soma deve ser empilhado na mesma estrutura. Esses dois valores somados podem ter sido originados por uma operação de carregamento da memória, ou podem ser constantes. Em uma situação como esta, o BT identifica a origem de cada operando e monta a instrução intermediária apropriada, levando em conta esta informação.

A única informação que é preservada referente à simulação dos *bytecodes* é a localização do topo da pilha de operandos, porque as variáveis auxiliares utilizadas nas operações são delimitadas levando-se em conta esta informação. Logo, o topo da pilha de operandos é atualizado no momento da tradução de cada um dos *bytecodes* que fazem uso dessa estrutura de dados.

Para as instruções que manipulam endereços de memória, como, por exemplo, as de *arrays*, o cálculo do endereço é realizado quando uma passagem completa por todos os *bytecodes* já foi realizada. Os locais no código nos quais há a necessidade de indicação de uma posição de memória são marcados durante a primeira passagem e processados no momento de seu término.

A etapa de interpretação de *bytecodes* termina no momento em que foram geradas instruções intermediárias para todos os *bytecodes* e quando todos os endereços de memória já tiverem sido calculados.

4.4 Módulo de criação de grafos intermediários

O módulo de criação de grafos intermediários tem a finalidade de gerar tais grafos, que darão suporte para demais análises e otimizações realizadas no compilador.

Para realizar a tarefa supracitada, este módulo foi divido em outros quatro módulos, sendo cada um deles responsável pela criação de cada um dos grafos intermediários. As funcionalidades de cada um desses sub-módulos são detalhadas nas subseções seguintes.

4.4.1 Grafo de fluxo de controle

O grafo de fluxo de controle é responsável por armazenar informações sobre o fluxo de controle do programa, o que significa que esta estrutura é capaz de rastrear possíveis caminhos de execução. O GFC é usado como a estrutura básica no processo de compilação, especialmente porque fornece informações importantes para as fases seguintes de compilação. No JaNi, ele é gerado pelo sub-módulo FA-Controle do módulo Analisador de Fluxo (*Flow Analyser* – FA).

A implementação de um GFC varia de acordo com o nível de informação que um nó contém, sendo que o mesmo pode representar tanto uma instrução intermediária quanto um bloco básico. Já em relação às arestas, estas são dirigidas e representam um fluxo de controle entre dois nós do GFC, o que significa que o fluxo do programa pode partir de um nó para o outro.

Dentro do compilador desenvolvido, o GFC foi construído baseado no algoritmo apresentado na Figura 4.3, utilizando os blocos básicos como vértices. Todos os blocos básicos foram encontrados, analisando instrução a instrução e verificando quando cada uma delas se tratava de um salto ou um endereço para salto e, com isso, as seqüências de instruções foram separadas.

Tanto o GFC quanto os outros três grafos abordados neste trabalho são representados internamente através de listas de adjacência, sendo que existe uma estrutura de armazenamento de dados em cada vértice, dependendo das restrições e características de cada grafo.

1.	Encontrar todos os blocos básicos no código intermediário, considerando instruções que realizam saltos e instruções que são alvos para saltos.
2.	Para cada bloco básico:
3.	Inspecionar a última instrução para verificar para onde o controle pode fluir a partir de sua execução.
4.	Criar uma aresta partindo do bloco básico analisado para o(s) bloco(s) básico(s) que pode(m) ser atingido(s) pelo controle do programa a partir do bloco em questão.

Figura 4.3 Algoritmo para construção do grafo de fluxo de controle.

4.4.2 Grafo de fluxo de dados

O grafo de fluxo de dados auxilia na tarefa de representação de certas relações referentes à manipulação de dados entre instruções. Tal grafo possui a característica de representar informações de quando os dados de uma instrução fluem para outra instrução, de acordo com o fluxo de controle do programa (CHAPMAN; ZIMA, 1990). O GFD é construído pelo sub-módulo FA-Dados do Analisador de Fluxo.

Em um grafo de fluxo de dados, os nós representam tipicamente instruções do código intermediário, contudo, variações podem ocorrer quanto ao conteúdo de cada nó, sendo que eles podem, inclusive, representar blocos básicos. As arestas são direcionadas e representam fluxo de dados do nó de origem para o nó destino, o que significa que a instrução correspondente ao destino utiliza algum dado que flui através da instrução correspondente à origem.

Neste trabalho, o grafo de fluxo de dados é construído para cada programa compilado, de maneira a fornecer apoio para as fases posteriores de análise de fluxo de dados. A abordagem utilizada para a construção deste grafo é descrita por meio do algoritmo mostrado na Figura 4.4.

O GFD é imprescindível para que seja dado um suporte adequado para o módulo de alocação de registradores, já que é esta análise de fluxo de dados que mapeia as relações de uso e definição de variáveis, necessárias para que as políticas de alocação de registradores possam produzir códigos mais eficientes. Por este motivo, a semântica inclusa nos vértices do grafo é voltada para a representação de instruções de código intermediário. Além disso, no GFD do JaNi, não há arestas

entre instruções que apenas usam um mesmo valor, já que este tipo de informação não é necessária para as outras fases de compilação, ao menos no que está inserido o escopo deste trabalho.

1.	Encontrar todos os blocos básicos no código intermediário.
2.	Construir o grafo de fluxo de controle a partir dos blocos básicos encontrados.
3.	Percorrer o código intermediário linearmente. Para cada instrução encontrada, fazer:
4.	Verificar se a instrução define algum valor. Em caso negativo, ir para passo 7. Em caso afirmativo, fazer:
5.	Caminhar através do grafo de fluxo de controle para determinar o alcance da definição encontrada no passo 4, utilizando o algoritmo de busca em profundidade (DFS – *Depth-First Search*) (CORMEN *et. al.*, 2001). Para cada uso no alcance da definição, fazer:
6.	Criar uma aresta que conecta a instrução sendo analisada no passo 4 à instrução que faz uso da definição em questão.
7.	Buscar próxima instrução e voltar ao passo 4. Caso não haja mais instruções, terminar.

Figura 4.4 Algoritmo para construção do grafo de fluxo de dados.

4.4.3 Grafo de dependência de controle

A existência do GFC torna possível a construção do GDC, uma vez que as relações de dependência de controle estão fortemente associadas com o fluxo de controle (AMTOFT *et. al.*, 2007; MUCHNICK, 1997). O grafo de dependência de controle serve como base para a análise de dependência de controle, a qual tem a propriedade de permitir um estudo mais aprofundado sobre o escalonamento de instruções em um programa, além de servir como suporte para algumas otimizações. O Analisador de Dependência (*Dependence Analyser* – DA) é o módulo responsável pela construção do GDC, realizada efetivamente pelo sub-módulo DA-Controle.

Para usar as definições de dependência de controle apresentadas na seção 2.4, é necessário que haja apenas um nó final[4] no GFC de modo que se tenha apenas um ponto de saída do programa, propriedade esta que pode ser conseguida através da

[4] Último nó executado antes do programa terminar.

adição de uma instrução sorvedoura no programa (e conseqüentemente no GFC). Esta instrução deverá ser o endereço alvo de qualquer outra instrução que venha a causar o fim do programa, de forma que o programa não possa parar sua execução sem que esta instrução seja executada.

Considerando o fato de que quando um bloco básico é alcançado durante a execução de um programa todas as instruções pertencentes ao mesmo são executadas, então o GDC foi construído tendo seus nós como blocos básicos ao invés de instruções. Como conseqüência, a estrutura para o grafo será menor e não haverá prejuízo para a análise de dependência de controle, bastando para tal identificar o bloco básico que abrange duas determinadas instruções. Caso o bloco básico seja o mesmo, haverá a dependência de controle entre as instruções, pois tanto a instrução que está depois na ordem linear de execução, como as demais nesta mesma situação dentro do bloco básico precisam ser executadas para que a primeira instrução na ordem linear também o seja. Caso as instruções estejam em blocos distintos, a análise deve prosseguir para o GDC em si.

Na Figura 4.5 é apresentado o algoritmo baseado em busca em profundidade para computar o GDC no JaNi.

1.	Obter o GFC gerado anteriormente e copiar seus nós para o GDC.
2.	Para cada dois nós no GFC:
3.	Se eles forem dependentes de controle:
4.	Criar uma aresta conectando os dois nós no GDC.

Figura 4.5 Algoritmo para construção do grafo de dependência de controle.

4.4.4 Grafo de dependência de dados

O grafo de dependência de dados é construído a partir da verificação de dependência entre cada par de instruções do programa, em que cada vértice representa uma instrução e as arestas representam a existência de dependência entre as instruções. As arestas podem ser rotuladas de modo a fornecer informações sobre qual o tipo de dependência existente. O sub-módulo DA-Dados do Analisador de Dados é o que possui a tarefa de gerar o GDD no JaNi.

Na Figura 4.6 pode-se verificar o algoritmo utilizado para a construção do grafo de dependência de dados que, da mesma forma que o algoritmo para o GFD, tomou como base instruções do código intermediário. Para a verificação de saída e entrada das instruções, foram utilizados os conceitos de definição (saída) e uso (entrada).

Como esta informação sobre qual dependência (ou conjunto de dependências) foi identificada em um par de instruções não é relevante para o atual estágio de implementação, ela foi omitida para que a estrutura do grafo fosse menor. Entretanto, caso seja necessário especificar todos os tipos de dependência de uma relação de dependência para atender a um requisito de futuras implementações, basta efetuar esta alteração, de pequeno esforço, na representação de vértice. Tal informação poderá ser útil em um algoritmo mais sofisticado de construção do GDD, visando uma redução ainda maior no número de arestas.

1.	Para cada par de instruções de código intermediário, fazer:
2.	Verificar a existência de dependência entre as instruções: saída, entrada e anti-dependência, nesta ordem. Se não houver qualquer uma, ir para passo 4. Caso haja alguma, fazer:
3.	Criar aresta no GDD ligando as instruções.
4.	Buscar próximo par de instruções e voltar ao passo 2. Caso não haja mais pares para analisar, terminar.

Figura 4.6 Algoritmo para a construção do grafo de dependência de dados.

4.5 Módulo de alocação de registradores

Depois da geração dos quatro grafos de representação intermediária, tem-se início a etapa de alocação de registradores, a qual é realizada mediante a execução das funcionalidades do módulo de alocação de registradores.

A descrição deste módulo é dividida em duas partes, sendo que a seção 4.5.1 descreve as transformações realizadas antes da aplicação da principal estratégia de alocação de registradores, a qual é descrita na seção 4.5.2.

4.5.1 Transformações preliminares

Antes de aplicar a principal política de alocação de registradores, a qual é baseada no princípio de coloração de grafos (seção 4.5.2), é realizada uma verificação da real necessidade de aplicação desta estratégia. O número máximo de registradores disponíveis para alocar as variáveis contidas no programa compilado é de 18 no Nios II, que possui um total de 32 registradores. Desse total, 10 registradores não podem ser manipulados no modo de execução do aplicativo (ALTERA, 2007), e os outros 4 são reservados pelo JaNi para armazenamento de informações referentes a deslocamento de endereço de memória e movimentação de dados.

No caso do número máximo de registradores que um programa pode utilizar ser maior do que o número de registradores que o programa utiliza efetivamente, a política de alocação de registradores não será acionada. Desta forma, apenas haverá a formação das palavras binárias na relação de 1 para 1 com as instruções intermediárias.

Uma estratégia para eliminação de algumas redundâncias no código é aplicada com o objetivo de reduzir o tamanho da estrutura de grafo a ser construída, mesmo quando a política de alocação de registradores não é acionada, pois a estratégia contribui para uma diminuição no número de instruções geradas. Entretanto, esta estratégia não é robusta o suficiente para ser considerada uma otimização de código, visto que utiliza heurísticas para a eliminação de instruções ao invés de análises profundas no fluxo de dados do programa.

Além de reduzir o número de instruções de baixo nível do programa sendo compilado, a eliminação das operações desnecessárias de movimento de valor (finalidade da estratégia aplicada) entre registradores pode diminuir a quantidade de registradores utilizados para a execução do programa. Este fato contribui para a execução mais eficiente da política de alocação de registradores, para quando a mesma tenha que ser ativada.

A necessidade da existência da etapa de transformações preliminares no compilador é estimulada porque o JaNi não contempla, em seu atual estado de implementação, otimizações nas camadas de análise de fluxo e dependência. No

momento em que estes mecanismos forem inseridos nas fases de análise, pode-se estudar a remoção desta etapa no gerador de código final.

4.5.2 Estratégia de coloração de grafos

A implementação da estratégia de alocação de registradores utilizada pelo JaNi baseou-se nos algoritmos apresentados na seção 3.4.

O grafo de interferência, utilizado por esta política, é representado através da estrutura de listas de adjacências, em que cada nó representa uma variável do programa e contém informações como, por exemplo, cor (registrador ao qual a variável será atribuída), custo de deixar a variável na memória (ou seja, não alocada a um registrador específico), entre outras para controle da própria estrutura do grafo.

Os passos do algoritmo de alocação de registradores através de coloração de grafos são seguidos até o momento em que seja possível colorir todos os nós do grafo de interferência restante, já que a cada passo o grafo é reconstruído.

A reconstrução do grafo de interferência é o ponto mais custoso da camada de alocação de registradores, pois, como a cada iteração do algoritmo é possível que ocorra a inserção de código em alguns trechos do programa-fonte, todas as relações de independência necessitam ser restabelecidas. Isto implica na reconstrução do Grafo de Fluxo de Dados e, conseqüentemente, na criação do Grafo de Fluxo de Controle, pois este consiste em um pré-requisito para que o GFD seja estabelecido. A construção do grafo de interferência é totalmente dependente das relações de fluxo de dados do GFD.

A eliminação de redundâncias também é realizada a cada iteração do algoritmo de alocação de registradores, já que a inserção de código pode resultar em um código com instruções neste perfil. Este passo do algoritmo também é conhecido como *register coalescing*. Caso haja alguma alteração de código neste sentido, os demais passos do algoritmo não são realizados e o grafo de interferência é construído em seguida.

Quando não houver cópias a serem eliminadas, é realizado o corte no grafo de interferência, a coloração de algumas variáveis – ou seja, a alocação efetiva de uma

variável a um registrador – e a inserção do *spill code* ao redor das instruções que fazem uso de variáveis que não puderam ser alocadas a registradores.

Ao término da execução das rotinas de alocação de registradores, há a existência de outro código intermediário, em que todas as variáveis já foram "coloridas". A partir deste momento, o código intermediário está pronto para se tornar o código final.

4.6 Módulo de geração do código final

Ao ser finalizada a etapa de alocação de registradores, a etapa de geração de código final, desempenhada pelo seu respectivo módulo, é iniciada. Esta é a última fase de compilação realizada pelo JaNi.

A única tarefa atribuída a este módulo é a de traduzir cada instrução no formato intermediário para o formato de palavra com 32 bits que atende às especificações do processador Nios II (ALTERA, 2007).

As instruções do código intermediário (saída do módulo de alocação de registradores) são processadas uma a uma, linearmente. Para cada instrução, a palavra apropriada é montada e gravada no arquivo, que corresponde ao arquivo binário gerado pelo compilador e apto para ser executado sobre um dispositivo configurado com o processador Nios II.

O código gerado pelo JaNi é constituído de uma seqüência de instruções para o Nios II que pode ser interpretada ou alterada por qualquer sistema que utilize este processador.

O gerenciamento de memória dos programas gerados pelo JaNi é desempenhado relativamente à posição ocupada pelo código, considerando o endereço zero como sendo o endereço da primeira instrução. O armazenamento das demais instruções segue disposição seqüencial e as demais estruturas inerentes à memória são armazenadas a partir da primeira posição livre após a última instrução do programa. É importante ressaltar que memória *cache* não é considerada.

4.7 Simulador de código final

Devido às propriedades do processador Nios II, para que seja possível a execução de um programa, é necessário que haja a configuração da placa de FPGA na qual o hardware referente ao processador será gerado, de modo a permitir que um código escrito para o Nios II possa ser executado.

Para evitar a necessidade de realizar a configuração da placa de hardware para simplesmente testar um aplicativo escrito para o processador Nios II, construiu-se um simulador (LIMA *et. al.*, 2008) do conjunto de instruções do mesmo, capaz de ler instruções no formato exigido pelo processador e emular os resultados. Este simulador também atende às necessidades de teste do JaNi, já que, diferente de outros simuladores para Nios II, como SimNios (LAUTERBACH, 2008) e ModelSim (MODELSIM, 2008), não é necessário que uma configuração de placa seja passada como parâmetro para que a simulação possa ser realizada.

É importante observar que instruções e estruturas referentes à utilização de memória *cache* do Nios II não fazem parte das implementações. Desta forma, considera-se como memória apenas os registradores do processador e a memória RAM.

A entrada do simulador construído deve conter uma seqüência de palavras representando instruções do Nios II. O simulador executa as instruções exibindo, ao final ou durante (dependendo da configuração dos parâmetros internos), o estado do conjunto de registradores junto com o tempo gasto na execução. Os valores armazenados na memória pelo programa, bem como suas posições relativas ao endereço da última instrução também são mostrados.

Duas tabelas referentes a estatísticas de instruções no código simulado também são mostradas mediante a execução do simulador. Ambas possuem agrupamento por categoria de instrução. A primeira apresenta dados de forma estática, sendo originada de um pré-contagem no número das instruções do programa, sem que o mesmo tenha sido simulado. Na segunda tabela, os dados são originados contando-se as operações efetivamente realizadas até o término da simulação do programa e, neste caso, apresentam-se de maneira quantitativa as instruções executadas pelo programa, também agrupadas por categoria.

A exibição das estatísticas de instruções existentes e operações realizadas por meio do simulador auxilia na análise de complexidade do código, visto que fornece parâmetros para avaliação de desempenho que não são influenciados pelas características da máquina em que a simulação é submetida.

O simulador deve ser acionado tendo como parâmetro o nome do arquivo binário que contém a seqüência das instruções do Nios II a serem processadas. No caso do JaNi, trata-se do arquivo `.bin` produzido pelo compilador. Este arquivo deve conter uma seqüência de palavras de 4 bytes cada uma, de acordo com o especificado na referência do processador Nios II (ALTERA, 2007) e mencionado na seção 3.3. As palavras devem estar no formato *little-endian* (HENNESSY; PATTERSON, 2003).

Nenhuma informação sobre configuração de placas de FPGA é necessária para o simulador. O intuito é simular o programa em si de maneira a verificar se as instruções do Nios II estão corretamente elaboradas e se não há erros na formação das palavras. Após esta comprovação, sabe-se que o programa obedece às restrições do código-alvo e, então, deve-se verificar os resultados produzidos comparando-os com os esperados, independente de uma plataforma de hardware específica.

Como o simulador foi desenvolvido para não depender da configuração de hardware, comandos Java de entrada e saída de dados através de periféricos não são simulados, implicando no fato de que os dados só podem ser observados mediante análise das estruturas do simulador.

As palavras contidas no arquivo de entrada são lidas uma a uma e interpretadas de forma a identificar primeiramente a que tipo de instrução pertence. Feito isto, os bits da palavra são interpretados adequadamente obtendo-se, assim, os parâmetros para a instrução identificada.

Com a instrução e seus parâmetros identificados, há uma checagem no que se refere à validade daquela instrução, ou seja, se aquela instrução não tenta escrever em registrador somente de leitura ou se os parâmetros não condizem com a operação a ser realizada.

No caso da instrução estar corretamente especificada na palavra, simula-se o efeito de sua execução. Cada instrução poderá alterar algum dos elementos constituintes do simulador, os quais são listados a seguir:

- Lista de registradores de propósito geral: arranjo de 32 *containers* para números inteiros de 32 bits que simula o conjunto de registradores de propósito geral do Nios II;
- Lista de registradores de controle: arranjo de 6 elementos inteiros de 32 bits que simula o conjunto de registradores de controle (ALTERA, 2007) do Nios II;
- Memória interna: estrutura de dados do tipo vetor que simula a porção de memória utilizada pelo programa em execução e por dados manipulados pelo programa;
- Contador de programa (PC – *Program Counter*): estrutura cujo valor significa a posição do controle do programa dentro da memória interna.

A cada instrução executada, exceto as de desvio, o contador de programa é incrementado de 4 unidades, para que seja posicionado no início da próxima palavra a ser interpretada.

Quando o contador de programa aponta para uma posição de memória que não pertence ao código o programa termina. Neste caso, o simulador pára a interpretação e exibe os resultados.

Terminada a simulação, cada um dos 32 registradores de propósito geral tem seu valor mostrado, sendo possível a análise de quais registradores efetivamente foram modificados e manipulados pelo programa simulado. Neste momento, as estatísticas e os valores das posições de memória também são apresentados, bem como o tempo total gasto na simulação. A medida de tempo poderá ser utilizada na análise de complexidade de um programa em relação a outro.

Além disso, se um erro de interpretação de instrução é encontrado ou se uma instrução manipula indevidamente seus parâmetros, uma mensagem de erro de execução será exibida e a simulação não prossegue.

4.8 Montador de código final (*assembler*)

Após a construção do simulador descrito na seção 4.7, houve a necessidade de testá-lo. Para facilitar a etapa de testes sobre o simulador foi construído um montador (*assembler*) para as instruções do Nios II. Desta forma, a criação de

seqüências de instruções submetidas ao simulador para teste pôde ser desempenhada de maneira mais eficiente.

O montador foi implementado de maneira simples, sem possuir verificações sintáticas fortes para os códigos em mnemônicos. A entrada para o *assembler* é um arquivo-texto contendo, em cada uma de suas linhas, instruções do processador Nios II escritas de acordo com os padrões (ALTERA, 2007).

A cada linha processada, a instrução é identificada e seus parâmetros são buscados de acordo com o padrão esperado. Como não há análises léxica e sintática sobre este código, caso a linha esteja fora do padrão esperado, a mesma deixa de ser processada. No entanto, a montagem das instruções não pára, sendo que as linhas obedientes ao referido padrão são processadas; o montador exibe mensagens de erro para cada linha não processada.

O processamento do arquivo de entrada é realizado em duas etapas:

1. Para cada linha do arquivo-texto de entrada identificada corretamente, é criada uma estrutura de armazenamento de instrução com as mesmas características da utilizada no JaNi;
2. Com as instruções no formato de código intermediário do JaNi, foi possível o acoplamento do módulo de geração de código final do compilador ao montador para a geração do arquivo binário.

4.9 Considerações finais

Neste capítulo, o trabalho realizado foi descrito conceitualmente, sendo possível ter uma visão do que o compilador JaNi faz e qual o intuito de sua construção, bem como quais das suas funcionalidades pertencem ao escopo do presente trabalho.

As análises mais profundas sobre cada um dos módulos são realizadas no capítulo 5, onde são apresentados os testes realizados sobre o JaNi e os seus respectivos resultados, permitindo que seja possível obter mais detalhes sobre o que foi implementado.

Capítulo 5 – Testes

Neste capítulo são apresentados e descritos os testes realizados sobre o JaNi, de maneira que se possa verificar a efetividade das implementações realizadas. Na seção 5.1 são descritos os módulos do JaNi e como cada um deles pode ser avaliado. A descrição de como o código gerado pelo compilador pode ser testado é feita na seção 5.2. Nas seções 5.3 e 5.4 os principais testes realizados são apresentados, juntamente com suas análises. Por fim, na seção 5.5 são feitas algumas considerações sobre o processo de testes sobre o JaNi.

5.1 Considerações sobre os módulos testados

A essência dos testes realizados no JaNi é apresentar evidências de que o compilador é capaz de transformar o código de entrada em *bytecodes* Java para o conjunto de instruções do Nios II. Por conseguinte, os testes são formulados elaborando-se aplicativos em Java com as estruturas suportadas pelo Nios II e compilando-o com um compilador *front-end* Java para que os *bytecodes* possam ser gerados. Assim, os *bytecodes* gerados servem como entrada para o JaNi. Após a execução do compilador, o arquivo binário gerado é passado ao simulador para que se possa visualizar os resultados da execução do conjunto de instruções criado.

Além disso, são gerados arquivos referentes aos dados produzidos por alguns módulos do JaNi, os quais são descritos a seguir:

- Tradutor de *bytecodes*: durante a fase de identificação dos *bytecodes* a serem compilados é gerado um arquivo apresentando toda a estrutura do arquivo

`.class` que foi reconhecida pelo compilador, detalhando, ainda, os *bytecodes* que serão posteriormente processados. Após o término da etapa de tradução dos *bytecodes* para o código intermediário do JaNi é criado um arquivo contendo a seqüência de instruções intermediárias criada no decorrer da execução do módulo BT;

- Analisadores de fluxo e dependência: após a execução de cada um dos sub-módulos responsáveis por criar os grafos intermediários, um arquivo correspondente é gerado para que se possa visualizar de maneira textual cada um dos quatro grafos gerados;
- Gerador de código: após a geração de código, além do arquivo binário que corresponde à saída do compilador, é gerado um arquivo-texto delineando os mnemônicos das instruções após a etapa de alocação de registradores e otimizações, o qual corresponde exatamente ao conjunto de instruções que serão transformadas para código binário.

Todos esses arquivos gerados durante a compilação de um programa têm como objetivo proporcionar o acompanhamento do processo de compilação no JaNi, e são gravados em um diretório apropriado.

5.2 Validação do código gerado

A validação do código final produzido pelo JaNi é feita junto ao simulador para conjunto de instruções do Nios II construído como parte do trabalho. Desta maneira, dado um arquivo com *bytecodes* de entrada para o JaNi, verifica-se que o mesmo foi corretamente compilado quando:

- O arquivo `.class` foi corretamente interpretado pelo módulo reconhecedor, gerando o arquivo de informações;
- Todos os grafos de representação intermediária foram gerados corretamente correspondendo com a estrutura do programa compilado;
- O arquivo binário foi gerado conforme o esperado e o compilador não produziu erros durante sua execução;
- Ao se executar o arquivo binário gerado através do JaNi sobre o respectivo simulador, pôde-se observar que são produzidos os mesmos resultados que o

aplicativo compilado produziria se executado sobre qualquer JVM em outro ambiente. Neste ponto, deve-se levar em conta a restrição imposta pelo fato de que operações de entrada e saída[5] não são suportadas pelo JaNi no seu atual estado de implementação.

Logo, quando o arquivo executável produzido pelo JaNi se comporta da mesma forma que o aplicativo em Java sobre uma JVM, significa que o compilador foi capaz de gerar código final fiel ao programa compilado.

5.3 Análise de testes sobre os principais módulos

Nas subseções seguintes, são apresentados os dados produzidos através das principais etapas da compilação no JaNi. São realizadas análises sobre esses dados levando em conta o aplicativo Java mostrado na Figura 5.1, utilizado como exemplificação da maneira com que os referidos dados são gerados. Este programa-exemplo mostra um algoritmo para cálculo do valor Fib(10).

```
// declaração de variáveis
int f1 = 1, f2 = 1;
short c;

// inicialização da variável de controle do loop
c = 10;

// computação da seqüência de Fibonacci para o número
// de termos escolhidos na variável 'c'
int aux;
for ( int i = 2; i < c; i++ ){
    aux = f1;
    f1 = f2;
    f2 = aux + f1;
}
```

Figura 5.1 Programa-exemplo Java.

O código mostrado na Figura 5.1 está compreendido em um método com nome “main” da classe “Main” de um projeto Java.

[5] Sem fornecer suporte para os *bytecodes* dos comandos de entrada e saída, não é possível exibir resultados da forma esperada pelo programador. A dificuldade em se implementar tais comandos está no fato dos mesmos estarem em bibliotecas Java, as quais não são suportadas no momento.

5.3.1 Informações adquiridas do arquivo de entrada

Considerando o programa Java da Figura 5.1, é apresentado na Tabela 5.1 o resumo dos principais dados (LINDHOLM; YELLIN, 1999) do arquivo `.class` reconhecidos pelo JaNi.

Tabela 5.1 Resumo das informações reconhecidas do arquivo `.class` referente ao programa Java apresentado na Figura 5.1.

Tipo de campo	Descrição	Valor encontrado
Número mágico	Sempre deve possuir o valor CAFEBABE em hexadecimal para um arquivo `.class` válido.	`0xCAFEBABE`
Versão	Versão em que se encontra o arquivo com os *bytecodes*.	`49.0`
Constant pool	Estrutura que armazena todo e qualquer tipo de constante utilizada no aplicativo.	–
Contador de *constant pool*	Número de registros na estrutura anterior.	`27`
Flag de acesso	Cada bit deste número corresponde a um tipo de acesso, para a classe.	`0x0021`
Classe	Número identificador da classe sendo compilada.	`2`
Super-classe	Número identificador da super-classe da classe sendo compilada.	`3`
Interfaces	Estrutura que armazena todas as interfaces da classe.	–
Contador de interfaces	Número de registros na estrutura anterior.	`0`
Métodos	Estrutura que armazena vários registros que especificam métodos da classe.	–
Contador de métodos	Número de registros na estrutura anterior.	`2`
Atributos	Estrutura que armazena todos os atributos pertencentes à classe.	–
Contador de atributos	Número de registros na estrutura anterior.	`1`

As principais estruturas de um arquivo .class no que diz respeito ao JaNi são a *constant pool* e a estrutura de armazenamento dos métodos. Mesmo apesar de os métodos serem reconhecidos antes dos atributos, o JaNi também reconhece os

atributos, com o objetivo de tornar o reconhecimento completo para futuras implementações.

As informações contidas na *constant pool* são essenciais para a identificação do método que será compilado, pois é nessa estrutura que são armazenados todos os nomes de atributos, variáveis e métodos, bem como especificação de tipos de dados utilizados, como nome e tamanho. Por este motivo, esta estrutura é a primeira especificada no arquivo após os dados de identificação do mesmo.

O reconhecimento de um arquivo `.class` deve ser linear, pois, conforme exemplificado no parágrafo anterior com a *constant pool*, vários dados dependem de outros também contidos no arquivo para que possam ser absorvidos.

Após o processamento total dos dados contidos no arquivo de entrada, o JaNi inicia a etapa de seleção da seqüência de *bytecodes* que será compilada. Isto é feito através de uma varredura sobre a estrutura de métodos, buscando-se qual deles possui o atributo nome com valor "main", sendo que este valor é referenciado no atributo do método como um índice para um registro da *constant pool*. Quando este método é encontrado, seu atributo código, que corresponde a um arranjo de bytes, é copiado para a próxima fase de compilação, a tradução de *bytecodes*, desempenhada pelo módulo BT.

5.3.2 Tradução de *bytecodes*

Os *bytecodes* são identificados através do atributo código do registro que representa o método, e são representados através de um vetor de bytes. Cada um desses bytes pode conter uma operação (neste caso, o número armazenado representará um *opcode*) ou um operando para um *bytecode*, cujo *opcode* tenha sido previamente identificado. A seqüência de *bytecodes* identificada para o programa da Figura 5.1 é apresentada na Figura 5.2, sendo que o número entre parênteses é o comprimento do *bytecode*, ou seja, a quantidade de bytes ocupada pelo *opcode* e seus parâmetros.

É possível notar a inicialização das variáveis inteiras nos *bytecodes* de 1 até 4 e da variável do tipo **short** nos *bytecodes* 5 e 6. Pode-se também perceber a inicialização da variável "c" nos *bytecodes* 7 e 8 e o início da estrutura de repetição a

partir do *bytecode* 9, sendo finalizada no *bytecode* 21, onde o fluxo de controle pode ir para a instrução de retorno do método ou retornar à análise da condição de continuidade do laço.

```
 1: ICONST_1  (1)          12: ILOAD_1  (1)
 2: ISTORE_1  (1)          13: ISTORE   (2)
 3: ICONST_1  (1)          14: ILOAD_2  (1)
 4: ISTORE_2  (1)          15: ISTORE_1 (1)
 5: BIPUSH    (2)          16: ILOAD    (2)
 6: ISTORE_3  (1)          17: ILOAD_1  (1)
 7: ICONST_2  (1)          18: IADD     (1)
 8: ISTORE    (2)          19: ISTORE_2 (1)
 9: ILOAD     (2)          20: IINC     (3)
10: ILOAD_3   (1)          21: GOTO     (3)
11: IF_ICMPGE (3)          22: RETURN   (1)
```

Figura 5.2 *Bytecodes* e seus comprimentos para programa da Figura 5.1.

É importante observar que, através da Figura 5.2, não é possível saber exatamente para qual endereço os saltos estão apontando e qual o valor exato armazenado em algumas variáveis, visto que os parâmetros são omitidos. Todavia, pode-se inferir o armazenamento do valor 1 na primeira e na segunda variável do método, pois tais atribuições são desempenhadas por meio de *bytecodes* com comprimento 1, ou seja, que possuem apenas o *opcode*, não necessitando de parâmetros.

Esta seqüência de *bytecodes* é processada linearmente de maneira que o código de representação intermediária primário[6] seja gerado. Para o programa-exemplo sendo compilado no JaNi, foi produzido o código representado por meio da Tabela 5.2.

Neste código intermediário, o conjunto de instruções é igual ao do Nios II, sendo que a única diferença é que os números de identificação de registradores não são válidos no contexto de Nios II, pois se referem às variáveis utilizadas no programa Java compilado. Para diferenciar variáveis do programa de registradores em instruções que devam manipular registradores específicos, são atribuídos números maiores que 32 para referência às variáveis. Quando o número especificado no campo de registrador for menor que 32, significa uma referência direta a um registrador do processador.

---

[6] Denomina-se, no contexto do JaNi, código intermediário primário o código criado a partir do processamento dos *bytecodes* e sem qualquer otimização ou renomeação de registradores.

A Tabela 5.2 está estruturada de forma similar às instruções do Nios II, sendo que a operação a ser desempenhada é identificada através do mnemônico. Não há, neste código intermediário, a distinção de tipo de instrução, sendo que em todas elas há campos para os 3 registradores e o número imediato de 16 bits. A utilização de cada um dos campos é determinada pela operação a ser desempenhada, em que aqueles que não são utilizados pela instrução em questão necessariamente possuem zero como valor.

Tabela 5.2 Representação do código intermediário não processado após tradução dos *bytecodes*.

Seq.	Mnemônico	Reg. A	Reg. B	Reg. C	IMM16
00	ADDI	0	23	0	4
01	ADDI	0	38	0	1
02	ADD	0	38	33	0
03	ADDI	0	38	0	1
04	ADD	0	38	34	0
05	ADDI	0	38	0	10
06	ADD	0	38	35	0
07	ADDI	0	38	0	2
08	ADD	0	38	37	0
09	ADD	0	37	38	0
10	ADD	0	35	39	0
11	CMPGE	38	39	38	0
12	ADDI	0	39	0	1
13	BEQ	38	39	0	40
14	ADD	0	33	38	0
15	ADD	0	38	36	0
16	ADD	0	34	38	0
17	ADD	0	38	33	0
18	ADD	0	36	38	0
19	ADD	0	33	39	0
20	ADD	39	38	38	0
21	ADD	0	38	34	0
22	ADDI	37	37	0	1
23	BR	0	0	0	-60
24	BR	0	0	0	4

A primeira instrução trata da inicialização do registrador de deslocamento (r23). Este registrador é utilizado durante o código para determinar o início da área de dados na memória utilizada pelo programa. Inicialmente, lhe é atribuído o valor 4 (bytes), entretanto, na etapa de geração de código final o valor atribuído ao registrador será alterado para corresponder à realidade do código gerado.

Caso o código compilado possua estrutura de *array*, duas outras instruções serão incluídas no código intermediário após a inicialização do r23. A primeira é a instrução de inicialização do registrador r22, utilizado para armazenar o início da área de memória ocupado pelos arranjos. A segunda é a inicialização do registrador r21, utilizado para guardar o tamanho da última estrutura de *array* criada, tornando-se importante para a definição do endereço de memória dos outros arranjos que eventualmente existirem no mesmo código.

A última instrução é a sorvedoura, mencionada durante o capítulo 3. Sempre que o programa for terminar esta instrução será atingida, pois isto contribui na geração dos grafos intermediários. Após a geração das outras instruções, é incluída a instrução de salto para 4 bytes à frente e, como trata-se da última instrução, isto causa o término do programa.

Durante o processo de tradução, cada operação é transformada em uma ou mais instruções. Como o código de *bytecodes* é estruturado em forma de pilha de operandos, faz-se uso de variáveis auxiliares para armazenamento de valores ou resultados de expressões a serem atribuídos. Muitas das instruções que envolvem variável auxiliar podem ser eliminadas devido à redundância, de acordo com o que foi abordado na seção 4.5.

Para as estruturas não seqüenciais, sejam elas de decisão ou repetição, os endereços destino das instruções de salto são calculados em uma etapa pós-processamento dos *bytecodes*. Tais instruções são sinalizadas para que, após o término da primeira passada sobre a seqüência de *bytecodes*, seja possível preencher estas instruções com os endereços corretos, de acordo com o tamanho do código gerado.

No código intermediário representado na Tabela 5.2 pode-se identificar uma atribuição do valor 1 à variável com número interno 33 nas instruções 1 e 2, sendo que na instrução 1 o valor é primeiro atribuído a uma variável auxiliar. Este é um exemplo de redundância, já que é possível eliminar a variável auxiliar cujo número interno é 38, realizando toda operação por meio de uma única instrução.

De maneira análoga à análise de percepção de *loop* na Figura 5.2, é possível notar a existência de uma estrutura de repetição cujo corpo está contido entre as instruções 14 e 21, inclusive. Pode-se perceber o incremento da variável de controle na instrução 22, bem como o cabeçalho do loop nas instruções de 09 a 13, com a

inicialização da variável de controle e análise das condições de continuidade. Na instrução 13, tem-se um salto condicional, que alterará o contador de programa para a instrução seguinte ou para a instrução 24, a qual é a imediatamente seguinte ao término do laço. Conforme já mencionado, os *offsets* dessas instruções são calculados em uma segunda fase de processamento de *bytecodes*.

5.3.3 Geração do GFC

Além de construir o grafo de fluxo de controle, a etapa de geração do GFC também possui a responsabilidade de identificar os blocos básicos no código intermediário. Como o GFC possui blocos básicos como vértices, este é um pré-requisito para que o grafo seja gerado. Todavia, uma vez que os blocos básicos foram identificados, não será necessário realizar esta etapa novamente no momento da criação de outras estruturas que dependam desses blocos básicos.

Para encontrar os blocos básicos, efetua-se uma passagem por todas as instruções do código intermediário, adicionando cada uma a um bloco básico, caso elas não impliquem em desvios nem sejam alvos de desvios. Esta verificação é feita para todas as instruções, respeitando as definições de bloco básico discutidas durante o capítulo 2. Para o programa da Figura 5.1, foram identificados os blocos básicos descritos na Tabela 5.3.

Tabela 5.3 Blocos básicos identificados na compilação do programa da Figura 5.1.

Bloco básico	Instruções compreendidas
B0	00 a 08
B1	09 a 13
B2	14 a 23
B3	24

Na Figura 5.3, é apresentado o GFC construído a partir dos blocos básicos da Tabela 5.3. Nela, pode-se observar a razão de cada instrução estar em seus respectivos blocos básicos.

O bloco básico B1 possui as instruções de verificação da condição da estrutura de repetição existente no programa compilado. A partir desta análise, decide-se se o controle do programa deve passar para a primeira instrução do corpo do laço e, portanto, uma iteração será realizada, ou se a próxima instrução deve ser a

imediatamente seguinte ao laço, neste caso, o fim do programa. Conforme mencionado, uma instrução sorvedoura é sempre inserida, e esta instrução constitui o bloco B3, com fundo diferenciado por este motivo.

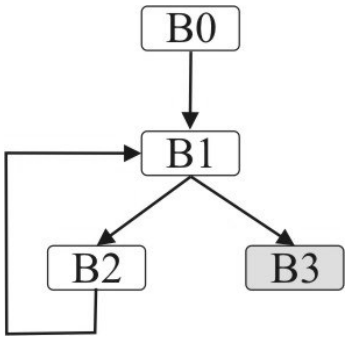

Figura 5.3 Grafo de Fluxo de Controle para programa da Figura 5.1.

A associação entre cada bloco básico e as suas instruções permanece durante todo o processo de compilação, podendo ser alterado apenas durante a etapa de alocação de registradores, no caso da necessidade de se executar a política de alocação de registradores por coloração de grafo. Isto porque, como o código intermediário deve ser modificado, tais alterações devem ser levadas em conta para que os grafos fiquem consistentes, já que o GFC e o GFD são utilizados a cada iteração deste algoritmo (seção 3.4).

5.3.4 Geração do GFD

O GFD é uma estrutura fundamental para a etapa de alocação de registradores por armazenar informações relacionadas ao fluxo das variáveis utilizadas dentro do programa. Sua construção é realizada a partir da análise do grafo de fluxo de controle e das variáveis existentes em cada uma das instruções. Na Tabela 5.4 é apresentado o GFD referente ao programa da Figura 5.1, lembrando que, neste caso, cada vértice do grafo corresponde a uma instrução de código intermediário.

Observando a Tabela 5.4, percebe-se que as instruções temporárias (atribuição de valor a variável temporária) têm fluxo de dados para a instrução da atribuição deste valor temporário, o que é o caso dos pares de instruções (1;2), (3;4), (5;6) e (7;8).

Há uma aresta no GFD sempre que um valor definido para uma variável em uma instrução possa ser utilizado por outra variável de outra instrução. Além disso, deve-se garantir que a segunda instrução possa ser atingida pelo controle do programa na mesma execução em que a primeira instrução é processada. Essa restrição é garantida, pois a análise é realizada com base em percursos pelo GFC.

Tabela 5.4 Representação tabular do GFD.

Instrução	Instrução para a qual os dados fluem
00	
01	02
02	14
03	04
04	16
05	06
06	10
07	08
08	09
09	11
10	11
11	13
12	13
13	
14	15
15	18
16	17
17	14 e 19
18	20
19	20
20	21
21	16
22	09
23	
24	

5.3.5 Geração do GDC

Da mesma forma que o GFC, o GDC também tem como vértices os blocos básicos do programa. Como os blocos básicos já foram encontrados na etapa de construção do GFC, a etapa de geração do GDC apenas verifica as relações entre cada par de blocos básicos tomando como base o GFC.

O GDC referente ao programa da Figura 5.1 é mostrado na Figura 5.4.

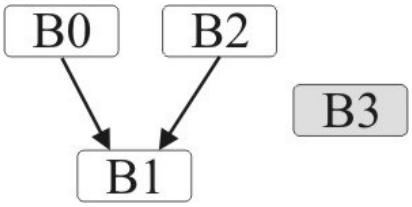

Figura 5.4 Grafo de Dependência de Dados para o programa da Figura 5.1.

Analisando-se o GFC na Figura 5.3, pode-se notar que não é possível atingir o fim do programa caso o controle esteja em alguma instrução dos blocos B0 ou B2, sem que o controle passe por todas as instruções do bloco B1. Neste caso, tem-se a dependência de controle de B0 e B2 em relação a B1, conforme mostrado no GDC criado. Como o bloco B1 tem ligação direta com o bloco B3 (final) no GFC, então ele não depende de qualquer outro bloco básico para que o fim do programa possa ser atingido. Da mesma forma, o bloco B3 não possui qualquer aresta ligada a ele.

5.3.6 Geração do GDD

O GDD é o grafo intermediário mais complexo existente no JaNi. Nesta estrutura são mapeadas as dependências de dados entre cada uma das instruções, sejam essas dependências de entrada, saída ou anti-dependência. Na Tabela 5.5 são apresentadas as arestas do GDD para 5 instruções intermediárias do programa da Figura 5.1, sendo que o GDD também é gerado com base em cada uma das instruções, e não em relação aos blocos básicos.

O GDD gerado possui 310 arestas e, por razões de simplificação, apenas arestas referentes às instruções de 09 a 13 são apresentadas. Este conjunto de instruções foi escolhido porque abrange instruções aritméticas, relacionais e de salto, de maneira a proporcionar uma visão mais ampla de como o GDD é gerado no JaNi.

Tabela 5.5 Parte do GDD para o programa da Figura 5.1.

Instrução	Instruções com dependência de dados
09	01, 02, 03, 04, 05, 06, 07, 08, 09, 11, 13, 14, 15, 16, 17, 18, 20, 21 e 22.
10	06, 10, 11, 12, 13, 19 e 20.
11	01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 e 21.
12	10, 11, 12, 13, 19 e 20.
13	01, 03, 05, 07, 09, 10, 11, 12, 14, 16, 18, 19 e 20.

No caso do par (11;1), por exemplo, tem-se as relações de dependência de entrada e saída entre essas duas instruções, pois na instrução 1 temos a atribuição de um valor imediato à variável de identificador 38, sendo que o valor desta variável é utilizado (caracterização da dependência de entrada) pela instrução 11 e, ainda, outro valor é atribuído (caracterização da dependência de saída) a essa mesma variável nesta instrução.

Um exemplo de relação de anti-dependência identificado para a compilação em questão é a aresta do GDD (13;20). Na instrução 20, tem-se a definição de valor para a variável 38, sendo que esta mesma variável é utilizada como uso na instrução 13, e, portanto, caracteriza-se a anti-dependência.

O tamanho do grafo de dependência de dados do JaNi pode ser reduzido, pois grande parte das arestas existentes são geradas por causa de instruções temporárias, sendo que caso elas sejam retiradas antes da criação deste grafo, isto pode implicar na geração de um GDD com número menor de arestas. Para isso, deve-se adiantar algumas otimizações do compilador, criando-se uma nova etapa antes da geração dos grafos de representação intermediária. Essa mesma concepção é válida no estudo do tamanho dos outros grafos, especialmente os que possuem instruções como sendo os nós, desde que as otimizações não dependam de grafos ainda não gerados.

Outra estratégia a ser estudada para a redução no tamanho do GDD é a identificação de dependências transitivas. Como exemplo, pode-se observar as dependências existentes entre as instruções 9, 11 e 13: sendo 11 dependente de 9 e 13 dependente de 11, logo 13 também é dependente de 9, não havendo necessidade da aresta (9;13) no grafo. Todavia, a identificação deste tipo de transitividade inclui um custo no algoritmo de geração do GDD, o qual deve ser avaliado.

5.3.7 Alocação de registradores e geração de código

Há dois casos de alocação de registradores: quando o número de variáveis no código intermediário excede o número de registradores disponíveis para uso no Nios II ou quando não. No processo de compilação do programa da Figura 5.1, a alocação

é realizada de maneira mais simples, já que o número de variáveis é relativamente pequeno.

Desta forma, nenhum grafo de interferência foi gerado e as variáveis foram atribuídas aos registradores de maneira direta após uma transformação de código para retirada de algumas cópias propagadas. Na Tabela 5.6 é apresentado o código intermediário após ser realizada a etapa de alocação de registradores para o programa da Figura 5.1.

Tabela 5.6 Código intermediário para o programa da Figura 5.1 após etapa de alocação de registradores.

Seq.	Mnemônico	Reg. A	Reg. B	Reg. C	IMM16
00	ADDI	0	23	0	84
01	ADDI	0	2	0	1
02	ADDI	0	3	0	1
03	ADDI	0	4	0	10
04	ADDI	0	6	0	2
05	ADD	0	6	7	0
06	ADD	0	4	8	0
07	CMPGE	7	8	7	0
08	ADDI	0	8	0	1
09	BEQ	7	8	0	40
10	ADD	0	3	7	0
11	ADD	0	7	5	0
12	ADD	0	3	7	0
13	ADD	0	7	2	0
14	ADD	0	5	7	0
15	ADD	0	2	8	0
16	ADD	8	7	7	0
17	ADD	0	7	3	0
18	ADDI	6	6	0	1
19	BR	0	0	0	-60
20	BR	0	0	0	4

Pode-se notar que o número de instruções no código foi reduzido devido ao fato de que algumas instruções de atribuição de valores por meio de variáveis temporárias terem sido retiradas. Cada uma das variáveis foi associada a um registrador diferente e o valor imediato da primeira instrução do código foi alterado com base no tamanho final do código intermediário.

É importante observar também que os endereços relativos nas instruções de salto (BR e BEQ) não precisaram ser recalculados, pois as instruções retiradas não influenciaram nesses endereços. Entretanto, há casos em que isto pode acontecer e, desta maneira, todos os endereços de salto são verificados quando o código

intermediário sofre alteração no número de instruções. Caso a instrução retirada (no caso da otimização) ou inserida (no caso de *spill code*) esteja entre uma instrução de salto e seu respectivo endereço, este endereço deverá ser ajustado.

Como para o programa da Figura 5.1 não houve a necessidade de se gerar o grafo de interferência, devido ao uso de um número pequeno de registradores[7], uma alteração nos parâmetros do compilador foi realizada. Assim, o grafo de interferência pôde ser gerado para este código de modo apenas didático, para que a visualização da estrutura gerada através do JaNi pudesse ser compreendida. Esta alternativa foi escolhida ao invés de se criar um programa maior para teste, já que isto poderia dificultar o entendimento do leitor devido ao tamanho da estrutura resultante.

O parâmetro referente à quantidade de registradores disponíveis para alocação de variáveis foi alterado para 5 e, com isso, o grafo intermediário representado por meio da Figura 5.5 foi criado. Com isto, a política de alocação de registradores via coloração de grafos foi acionada, permitindo também a observação de seu funcionamento.

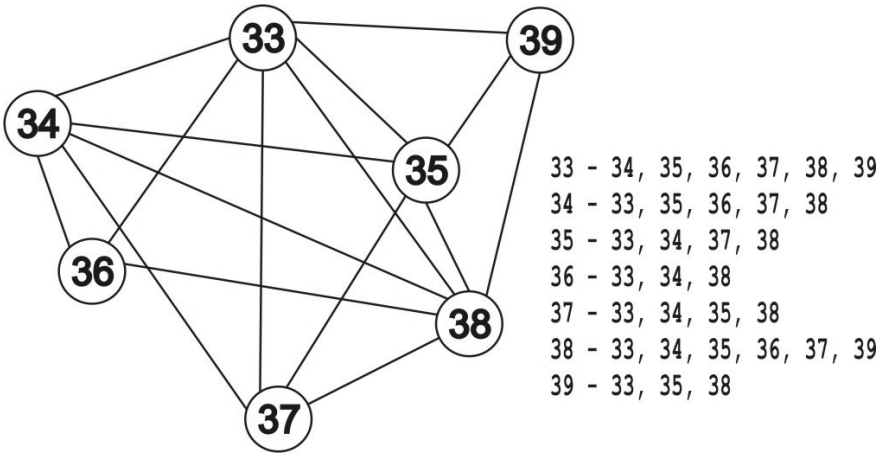

Figura 5.5 Grafo de interferência para o programa da Figura 5.1.

Como se tem um total de 7 variáveis no programa e apenas 5 registradores para que elas possam ser alocadas, então há a necessidade de que 2 dessas variáveis sejam armazenadas em memória, e não no processador. A política de alocação de registradores efetuou a escolha dessas variáveis e relacionou as demais a cada um dos registradores disponíveis. Às variáveis que necessitam armazenamento na

[7] A elaboração de programas maiores, com utilização de várias estruturas de dados, para testes sobre o JaNi é limitada ao subconjunto da linguagem Java suportado até o presente momento. Este fato não possibilita a geração de exemplos mais robustos para os testes aqui desempenhados.

memória foi associado um endereço relativo ao registrador de deslocamento, que é o r23 e cujo valor é atribuído na primeira instrução do código, conforme mencionado em seções anteriores.

O código intermediário depois de terminada a alocação de registradores nesta situação é apresentado por meio da Tabela 5.7.

Tabela 5.7 Código intermediário com *spill code* para programa da Figura 5.1.

Seq.	Mnemônico	Reg. A	Reg. B	Reg. C	IMM16
00	ADDI	0	23	0	128
01	ADDI	0	3	0	1
02	ADD	0	3	20	0
03	STW	23	20	0	4
04	ADDI	0	3	0	1
05	ADD	0	3	20	0
06	STW	23	20	0	8
07	ADDI	0	3	20	0
08	ADD	0	3	4	0
09	ADDI	0	3	0	2
10	ADD	0	3	2	0
11	ADD	0	2	3	0
12	ADD	0	2	0	1
13	CMPGE	3	2	3	0
14	ADDI	0	2	0	1
15	BEQ	3	2	0	60
16	LDW	23	20	0	4
17	ADD	0	20	3	0
18	ADD	0	3	2	0
19	LDW	23	20	0	8
20	ADD	0	20	3	0
21	ADD	0	3	20	0
22	STW	23	20	0	4
23	ADD	0	2	3	0
24	LDW	23	20	0	4
25	ADD	0	20	2	0
26	ADD	2	3	3	0
27	ADD	0	3	20	0
28	STW	23	20	0	8
29	ADDI	2	2	0	1
30	BR	0	0	0	-80
31	BR	0	0	0	4

Para que essas variáveis possam ter seus valores usados e atribuídos, faz-se necessária a utilização das instruções *load*/*store*, que carregam/armazenam valores dessas variáveis. Para isso, essas instruções são inseridas no código intermediário de acordo com a operação realizada sobre a variável. Caso seja uma operação de escrita, então o valor a ser escrito, ao invés de ser gravado na variável, é gravado em um

registrador temporário (sempre o r20) seguido de uma instrução *store* do valor de r20 para o endereço correspondente à variável em questão. Para operações de leitura, é realizado o processo inverso, sendo o valor carregado no r20 para posterior utilização. Logo, quando o *spill code* é gerado, é o registrador r20 que substitui a variável armazenada em memória dentro da operação em si.

Um resumo com a relação dos registradores especiais utilizados pelo JaNi é apresentado no Apêndice B.

5.4 Análise de testes sobre o compilador

Os testes apresentados nesta seção são direcionados à compilação de um programa-teste no JaNi e sua simulação, para que os resultados possam ser comparados aos esperados, no que se refere às variáveis de armazenamento de resultados finais.

O código-fonte do programa testado é apresentado na Figura 5.6. Neste código, buscou-se a inserção das estruturas de programação suportadas pelo JaNi, com resultados finais que dependam da execução correta dessas estruturas. Da mesma forma que o programa da Figura 5.1, este novo programa-teste também é abrangido pelo método principal da classe principal do projeto a que pertence.

Analogamente à compilação descrita na seção 5.3, nesta o código da Figura 5.6 foi compilado em um *front-end* de Java – NetBeans® (NETBEANS, 2009) – para a geração dos *bytecodes* a serem processados pelo JaNi. Neste programa, encontra-se variáveis escalares de diversos tipos de dados suportados, bem como uma estrutura de arranjo simples. Como os métodos de captação de dados não são ainda suportados pelo JaNi, todas essas estruturas declaradas foram iniciadas com valores constantes no próprio código. A finalidade deste programa é fazer com que os *bytecodes* suportados sejam utilizados e não representa um algoritmo que produz dados significantes, como o programa da Figura 5.1.

O JaNi identificou, para este programa compilado, 32 estruturas na *constant pool* do arquivo `.class`. Também foram identificados 2 métodos, sendo um deles o método “main” e o outro o método “<init>”, o qual é interno ao aplicativo e utilizado

para inicialização de componentes internos. Um atributo foi identificado, sendo este padrão nos aplicativos Java e denominado “SourceFile”, usado internamente.

```
// declaração de algumas variáveis
short entr01, entr02;   // tipo short
int a, b, c;    // tipo int
char ch1, ch2;  // tipo char
boolean flag;   // tipo boolean

// declaração do array
int [] array;

// inicializações
entr01 = -34;
entr02 = (short) ( entr01 + 75 );
a = 0;
b = 236;
flag = false;
array = new int[13];

// estruturas de decisão
if ( b >= 230 && a == 0 )
    ch1 = 'X';
else ch1 = 'Y';

ch2 = 'L';
if ( entr01 > 0 || entr02 < 0 ){
    entr01 *= -1;
    ch2 = 'W';
}else{
    if ( ch1 != 'X' )
        ch2 = 'I';
    entr01 /= 2;
}

// estruturas de repetição
c = 13;
while ( --c > -1 )
    array[c] = -c;

do{
    b++;
    for ( int i = 0; i < 13; i++ ){
        if ( flag )
            array[i] += b / 5;
        flag = !flag;
    }
    if ( entr01 > 0 ) entr01--;
    else entr01++;
}while ( entr01 != 0 );
```

Figura 5.6 Programa Java para teste no JaNi.

A seqüência de *bytecodes* gerada possui tamanho 182 (este tamanho não representa o número de *bytecodes*, mas, na verdade, a soma dos tamanhos de cada

um dos *bytecodes*). O número máximo de variáveis locais utilizadas ao mesmo tempo é de 11, e a profundidade máxima da pilha é de 5 operandos[8].

O compilador identificou 26 blocos básicos para este programa, gerando um GFC com 36 arestas. O maior bloco básico é o que contém a seqüência com as primeiras instruções do código, ou seja, as instruções de inicialização das variáveis, as quais são feitas seqüencialmente sem que haja saltos a partir delas ou para elas. O menor deles é o que contém a instrução sorvedoura do código intermediário.

No GFD foram construídas 129 arestas, um número relativamente baixo se comparado com o GFD do teste da seção 5.3, cujo programa-teste relacionado possuía tamanho e complexidade menores. Entretanto, como o módulo construtor do GFD leva em conta a passagem dos valores das variáveis de uma instrução a outra, e como este último programa-teste tem maior número de variáveis sendo utilizadas, então o fluxo de valores entre as instruções tende a ser menor.

O GDC gerado pelo JaNi possui 113 arestas conectando pares de blocos básicos encontrados no momento da construção do GFC. Conforme esperado, a maior parte das ligações entre blocos básicos neste grafo ocorrem para os blocos que abrangem instruções mais próximas do começo do programa, visto que dificilmente será possível atingir o final do programa a partir desses blocos sem que seja necessário que o controle passe por instruções de outros blocos.

Ainda em relação ao GDC, é possível perceber que o bloco básico que abrange as instruções de código intermediário referentes à condição do laço `do-while` (o qual é o último antes do que contém o nó sorvedouro) provoca dependência de controle a todos os outros blocos básicos, pois sempre será necessário que esta condição seja analisada ao menos uma vez antes que o programa termine.

Como já mencionado na seção anterior, a maior estrutura de dados construída pelo JaNi é o GDD. Para o programa da Figura 5.6, o mesmo possui 6.946 arestas, sendo que algumas instruções possuem mais de 70 vértices como vizinhos. Todavia, é importante ressaltar novamente que a redução deste número pode ser conseguida incluindo-se etapas de otimização para a construção do grafo.

---

[8] Esses dados são extraídos também do arquivo `.class`, contidos no atributo código.

Na fase de alocação de registradores, nenhuma ação especial é tomada, pois a soma das variáveis escalares utilizadas no programa não excede o número de registradores do processador. Portanto, cada variável é alocada a um registrador sem que o grafo de interferência seja gerado. A etapa de geração de código final foi iniciada sobre este código intermediário alterado e, assim, o arquivo binário com as palavras do Nios II foi gerado.

Para que se possa verificar o funcionamento do programa gerado pelo JaNi, submeteu-se, então, este arquivo binário ao simulador de instruções do Nios II construído como parte deste trabalho. A associação entre cada registrador mostrado pelo simulador e as variáveis do programa Java, bem como os respectivos valores finais, são apresentados na Tabela 5.8.

Tabela 5.8 Associação entre variáveis do programa Java e registradores do Nios II.

Variável	Registrador	Valor final
entr01	r02	0
entr02	r03	41
a	r04	0
b	r05	253
c	r06	-1
ch1	r07	88
ch2	r08	76
flag	r09	1
i	r10	13
-	r11	1
-	r13	1
-	r22	548
-	r23	544

Os valores finais mostrados na Tabela 5.8 foram produzidos tanto através da execução do programa Java sobre uma JVM quanto a partir da execução do programa gerado pelo JaNi sobre o simulador. Pode-se observar que, além dos registradores associados a variáveis do programa, outros 4 registradores possuem valor diferente de zero armazenado após o término do programa. Eles são registradores associados a variáveis temporárias e/ou internas do código intermediário no caso dos registradores r11 e r13. O registrador r23, como já explicado, armazena o primeiro endereço de memória que não corresponde a uma instrução do programa, relativo ao endereço da primeira instrução deste programa. Já o r22, contém o primeiro endereço disponível para armazenamento de estruturas dinâmicas do programa compilado.

É a partir do endereço do r23 que as variáveis escalares são armazenadas em memória (quando há mais variáveis que registradores). Já os arranjos são armazenados a partir do endereço contido em r22. Portanto para que se possa consultar os valores em cada posição do vetor de inteiros criado no programa Java da Figura 5.6, deve-se examinar, mediante os dados produzidos pelo simulador, os valores dessas posições de memória. Esses valores são mostrados através da Tabela 5.9.

Como não houve a necessidade de nenhuma variável ser armazenada em memória para este programa-teste, então o vetor de inteiros é a única estrutura do programa a utilizar este espaço.

Tanto a compilação quanto a simulação foram executadas em tempo desprezível, em um computador com 1.92GHz de *clock* e 2GB de memória RAM.

Tabela 5.9 Associação entre vetor e memória após compilação.

Elemento	**End. relativo a r22**	**End. relativo à primeira instrução**	**Valor**
`array[0]`	4	560	389
`array[1]`	8	564	436
`array[2]`	12	568	387
`array[3]`	16	572	434
`array[4]`	20	576	385
`array[5]`	24	580	432
`array[6]`	28	584	383
`array[7]`	32	588	430
`array[8]`	36	596	381
`array[9]`	40	600	428
`array[10]`	44	604	379
`array[11]`	48	608	426
`array[12]`	52	612	377

Para que o módulo de alocação de registradores pudesse ser acionado também para o teste com o programa da Figura 5.6, o parâmetro referente à quantidade de registradores disponíveis foi reduzido para 10, de maneira que se torne necessária a execução da política de alocação de registradores do JaNi.

A redução da quantidade de registradores disponíveis para 10 provocou a criação do grafo de interferência, com 116 arestas. O grafo foi construído apenas uma vez, pois dadas as restrições para este caso, foi possível encontrar uma forma de colorir todos os vértices desse grafo sem a necessidade de escolher uma variável para

ser armazenada em memória. Assim, constata-se que é possível que todas as variáveis sejam alocadas em registradores mesmo quando o número destes é inferior ao número daquelas.

Mais um teste foi realizado no que se refere à alocação de registradores. O número de registradores disponíveis foi agora reduzido a 5. Nesta situação, o grafo necessitou ser construído também apenas uma única vez, porém, seis variáveis foram movidas para a memória a partir da análise da estrutura; as outras foram coloridas com sucesso e associadas a registradores disponíveis.

Nenhuma das modificações teve impacto significativo no tempo de compilação do programa testado. As estruturas produzidas na compilação do programa da Figura 5.6 podem ser vistas no Apêndice C. No Apêndice D há uma relação de todos os *bytecodes* atualmente suportados pelo JaNi. Já no Apêndice E são elencadas todas as instruções do Nios II que podem ser geradas pelo JaNi de acordo com o subconjunto da linguagem suportado.

5.5 Considerações finais

Esse capítulo apresentou, mediante exemplos, o funcionamento do JaNi, objetivando mostrar como o compilador recebe e processa os dados.

Buscou-se um teste de "caixa branca", ou seja, a exposição de como cada módulo do compilador se comporta, bem como quais os dados necessários para sua execução com sucesso e quais os dados produzidos para o módulo subseqüente do esquema conceitual do JaNi.

Foi realizado um teste de "caixa preta", em que o JaNi foi submetido a um programa mais complexo e, por este motivo, o detalhamento do funcionamento de cada módulo poderia se tornar confuso devido à quantidade de informação gerada. Desta forma, o foco principal foi dirigido para os dados essenciais produzidos, apresentados majoritariamente de maneira quantitativa.

Portanto, através deste capítulo, foi possível averiguar o funcionamento das implementações realizadas, com análises referentes aos pontos fortes e fracos.

Capítulo 6 – Conclusões

Com base no que foi discutido no decorrer deste trabalho, o intuito do capítulo 6 é realizar uma apanhado geral sobre o que foi desenvolvido. O objetivo da seção 6.1 é realizar uma separação do escopo do presente trabalho e as implementações agregadas ao mesmo, ou seja, para que a construção do compilador proposto pudesse ser viabilizada. As principais contribuições são analisadas durante a seção 6.2, em que também são sugeridos os trabalhos futuros sobre o JaNi. Na seção 6.3 são apresentadas algumas considerações finais.

6.1 Trabalho agregado

Para que os objetivos do trabalho pudessem ser alcançados foi necessária a construção de um simulador de modo que a saída do JaNi fosse testada. De acordo com o que foi mencionado na seção 4.7, não foi possível a utilização de um simulador já pronto. Então, um novo simulador foi construído, sendo que mais detalhes sobre o mesmo podem ser encontrados em Lima *et. al.* (2008).

Conforme explicado na seção 4.8, também era necessário que o simulador fosse testado, de maneira a garantir a efetividade dos testes sobre o JaNi. Com a finalidade de facilitar os testes sobre o simulador, foi implementado o montador de código para o Nios II. No entanto, é necessário que se implemente analisadores (léxico e sintático) sobre o arquivo-texto com as instruções a serem montadas, tornando-o robusto.

6.2 Principais contribuições e trabalhos futuros

O levantamento de trabalhos correlatos mostra que até a presente data não existe um compilador que traduz um conjunto de *bytecodes* Java para as instruções do Nios II, apesar da existência de geradores de circuito para FPGA. Desta forma, a principal contribuição do JaNi é proporcionar esta compilação, visto que tanto a demanda por Java quanto por Nios II têm ganho importância atualmente.

É importante observar a existência da ferramenta proprietária *Mentor Graphics*® (MENTOR, 2009), a qual auxilia na geração de aplicativos para vários processadores, dentre eles o Nios II. Contudo, o custo desta ferramenta é alto, dificultando sua utilização em trabalhos acadêmicos e que visam construção de softwares livres e com código aberto. Isto cria uma motivação extra para a continuidade dos trabalhos com o JaNi.

A finalidade deste trabalho foi especificar as propriedades do compilador, bem como sua arquitetura e propósitos. Foi constatada a viabilidade de se efetuar os aperfeiçoamentos listados a seguir sobre o compilador.

Como o JaNi está em estado inicial de desenvolvimento, são muitas as sugestões para trabalhos futuros, com destaque para as que seguem. Para cada uma delas, devem ser realizados testes que comprovem o funcionamento, ampliando a possibilidade de verificações a serem realizadas:

- Implementação dos tipos de dados e estruturas de controle de um programa Java: trata-se de um dos objetivos do JaNi, ser capaz de suportar todos os *bytecodes* que podem ser gerados a partir de um programa Java;
- Implementação do suporte a métodos: fazendo com que o JaNi suporte criação de métodos pode-se habilitar vários outros testes sobre o compilador, como por exemplo chamadas recursivas. Além disso, o suporte aos métodos é um grande passo para que o compilador possa suportar projetos mais robustos. Para isto, a etapa de identificação dos *bytecodes* deve ser alterada para que outros métodos possam ser também reconhecidos; a construção dos grafos intermediários também deverá levar em conta a existência desses métodos;
- Implementação de suporte a classes: esta é a implementação que deve trazer mais impactos na estrutura do JaNi, pois haverá a necessidade de reorganizar

a entrada de dados do compilador, visto que um projeto com várias classes possuirá vários arquivos `.class` que devem ser processados de maneira conjunta, já que as classes podem reconhecer métodos umas das outras.

- Implementação do suporte às bibliotecas Java: quando o JaNi for capaz de reconhecer métodos e classes, será possível fazer com que os métodos referentes a entrada e saída de dados também sejam suportados, bem como toda a gama de operações contidas nas bibliotecas padrão;
- Otimizações: inclusão de todas as otimizações de código possíveis de serem aplicadas em um compilador deste tipo. Isto reduzirá tanto o tamanho do programa produzido quanto o tamanho das estruturas internas utilizadas pelo JaNi. Outra alteração neste sentido refere-se à instrução utilizada como sorvedoura no código intermediário; deve-se verificar a possibilidade de desempenhar HALT ao invés do salto para 4 bytes à frente, pois com a primeira o processador cessará imediatamente o processamento sem buscar pela próxima instrução;
- Paralelismo: com as estruturas intermediárias geradas pelo JaNi, é possível incluir técnicas para paralelização do código compilado, fazendo com o que JaNi se transforme em um super-compilador. Neste caso, a geração de código não mais deverá ser para um processador, e sim para um conjunto deles;
- Testes com simulador da Altera®: a construção do simulador neste trabalho foi motivada pelo fato de não ter sido encontrada ferramenta capaz de simular instruções do Nios II sem a necessidade de especificação da configuração de placa. Todavia, é possível a realização de um trabalho de adaptação do simulador da Altera® (*Nios II Instruction Set Simulator* – ISS) de maneira a acoplar o código binário gerado pelo JaNi, tornando possível sua simulação nesta ferramenta (NIOS, 2009);
- Elaboração de tutorial: com o levantamento bibliográfico realizado é possível a criação de um tutorial sobre compilação a partir de *bytecodes* Java, sendo que o exemplo de compilação na seção 5.3 pode ser utilizado para uma explicação do funcionamento das estruturas de dados utilizadas neste tipo de compilação.

6.3 Considerações finais

Buscou-se, com este trabalho, descrever toda estrutura do novo compilador, sua finalidade e as etapas já implementadas. Os objetivos foram superados e o JaNi pode ser utilizado para a compilação dos *bytecodes* gerados por uma programa Java para o conjunto de instruções do Nios II, dadas suas limitações e propriedades delineadas ao longo do texto.

No que diz respeito às sugestões para trabalhos futuros, todas referem-se a implementações que visam a inclusão de módulos adicionais ao compilador, permitindo uma ampliação nas situações em que o mesmo pode ser utilizado, bem como seus produtos agregados: simulador e montador.

Referências

AHMADINIA, A.; BOBDA, C.; MAJER, M.; TEICH, J. The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer. **Journal of VLSI Signal Processing**, v. 47, n. 01, p. 15-31, Abr. 2007.

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. **Compiladores**: Princípios, Técnicas, e Ferramentas. 2. ed. Addison-Wesley, 2008.

ALTERA **Nios II Custom Instruction - User Guide**. 2005. Disponível em: <http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf>. Acesso em: 25 jul. 2007.

ALTERA Literature: **Nios II Processor**. 2007. Disponível em: <http://www.altera.com/literature/lit-nio2.jsp>. Acesso em: 23 ago. 2007.

AMTOFT, T.; BANERJEE, A.; HATCLIFF, J.; RANGANATH, V. P. A New Foundation for Control Dependence and Slicing for Modern Program Structures. **ACM Transactions on Programming Languages and Systems**, v. 29, n. 5, p. 1-43, 2007.

BAL, H. E.; GRUNE, D.; JACOBS, C. J. H.; LANGENDOEN, K. G. **Projeto Moderno de Compiladores**: Implementações e Aplicações. 1. ed. Rio de Janeiro: Editora Campus, 2001.

BACHMANN, C.; BANERJEE, P. CHOUDHARY, A.; HALDAR, M.; HAUCK, S.; JOISHA, P.; JONES, A.; KANHARE, A.; NAYAK, A.; PERIYACHERI, S.; SHENOY, N.; WALKDEN, M.; ZARETSKY, D. A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems. In: FCCM, 2000, **Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines**, Washington: 2000. p. 39.

BECK, A. C. S.; CARRO, L. Application of binary translation to Java reconfigurable architectures. In: Parallel and Distributed Processing Symposium, 2005, **Proceedings - 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2005**, 2005. 1 CD-ROM.

BENNETT, D.; DELLINGER, E.; MASON, J.; PUTNAN, A. R.; SUNDARARAJAN, P. CHiMPS: a high-level compilation flow for hybrid CPU-FPGA architectures. In: International Symposium on Field Programmable Gate Arrays, 2008, Monterey, EUA, **Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays**, New York: 2008, p. 261-261.

BERTELS, K.; PANAINTE, E. M.; VASSILIADIS, S. The Molen compiler for reconfigurable processors. **ACM Transactions on Embedded Computing Systems (TECS)**, v. 06, n. 01, Fev. 2007.

BIRCH, K. A.; GALIVAN, K. A.; SHOU, R. A.; VAN ENGELEN, R. A. Toward Efficient Flow-Sensitive Induction Variable Analysis and Dependence Testing for Loop Optimization. In: ACM Southeast Regional Conference, 2006, Melbourne, EUA, **Proceedings of the 44th annual Southeast regional conference**, New York: ACM Press, 2006. p. 1-6.

BRACHA, G.; GOSLING, J.; JOY, B.; STEELE, G. **The Java™ Language Specification - Third Edition**. Addison-Wesley, 2005. Disponível em: <http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf>. Acesso em: 25 jul. 2007.

BUDIU, M.; CADAMBI, S.; GOLDSTEIN, S. C.; MOE, M.; SCHMIT, S.; TAYLOR, R. R. PipeRench: a reconfigurable architecture and compiler. **Computer**, v. 33, n. 04, p. 70-77, Abr. 2000.

BURDY, L.; PAVLOVA, M. Java Bytecode Specification and Verification. In: SYMPOSIUM ON APPLIED COMPUTING, 2006, Dijon, França, **Proceedings of the 2006 ACM symposium on Applied Computing**, New York: ACM Press, 2006. p. 1835-1839.

CALLAHAN, T.J.; HAUSER, J.R.; WAWRZYNEK, J. The Garp architecture and C compiler. **Computer**, v. 33, n. 04, p. 62-69, Abr. 2000.

CARDOSO, J.M.P.; NETO, H.C. Compilation for FPGA-based reconfigurable hardware. **IEEE Design and Test of Computers**, v. 20, n. 02, p. 65-75, 2003.

CARDOSO, J. M. P. **Compilação de Algoritmos em Java™ para Sistemas Computacionais Reconfiguráveis com Exploração do Paralelismo ao Nível das Operações**. 2000. 312 f. Tese (Doutorado em Engenharia Electrotécnica e de Computadores) – Instituto Superior Técnico, Universidade Técnica de Lisboa, Lisboa, 2000.

CARDOSO, J.M.P.; NETO, H.C. Macro-based hardware compilation of Java™ bytecodes into a dynamic reconfigurable computing system. In: IEEE Symposium on FPGAs for Custom Computing Machines, 1999, Napa Valley, EUA, **Field-Programmable Custom Computing Machines, 1999. FCCM '99. Proceedings. Seventh Annual IEEE Symposium on**: 1999, p. 2-11.

CARDOSO, J. M. P.; NETO, C. H. Towards an Automatic Path from Java™ Bytecodes to Hardware Through High-Level Synthesis. In: 5TH IEEE INTERNATIONAL CONFERENCE ON ELECTRONICS, CIRCUITS AND SYSTEMS (ICECS-98), Set. 1998, Lisboa, Portugal, **Proceedings of the5th IEEE International Conference on Electronics, Circuits and Systems (ICECS-98)**, Lisboa: 1998. v. 01, p. 85-88.

CHAPMAN, B.; ZIMA, H. . **Supercompilers for Parallel and Vector Computers**. 1. ed. Cornwall: ACM Press, 1990.

CHATTOPADHYAY, A.; CHEN, X.; HAO, L.; KAMMLER, D.; KARURI, K.; LEUPERS, R. A Design Flow for Architecture Exploration and Implementation of Partially Reconfigurable Processors. **IEEE Transactions On Very Large Scale Integration (Vlsi) Systems**, v. 16, n. 10, p. 1281-1294, Out. 2008.

CHEREM, S.; RUGINA, R. A verifier for region-annotated java bytecodes. **Electronic Notes in Theoretical Computer Science**, v. 141, n. 01, p. 183-201, Dez. 2005.

CHEUNG, P. Y. K.; CONSTANTINIDES, G. A.; LUK W.; MENCER, O.; TODMAN, T. J.; WILTON, S. J. E. Reconfigurable Computing: architectures and design methods. **IEE Proceedings - Computers And Digital Techniques**, v. 152 n. 02, p. 193-207, Mar. 2005.

CHOI, H.; HONG, S.; KIM, J.; LEE, J.; MOON, S.; OH, H.; SHIN, J. L. Java Cliente Ahead-of-Time Compiler for Embedded Systems. **Proceedings of the 2007 LCTES conference**, v. 42, n.07, p. 63-72, Jul. 2007.

CIERNIAK, M.; LI, W. Optimizing Java Bytecodes. **Concurrency-Practice and Experience**, v. 09, n. 06, p. 427-444, Jun. 1997.

COMPTON, K.; HAUCK, S. Reconfigurable Computing: A Survey of Systems and Software. **ACM Computing Surveys**, v. 34, n. 02, p.171-210, Jun. 2002.

COMPTON, K.; HAUCK, S. An Introduction to Reconfigurable Computing. **IEEE Computer**, Invited Paper, Abr. 2000.

COOPER, K. D.; DASGUPTA, A. Tailoring Graph-coloring Register Allocation For Runtime Compilation. In: IEEE International Symposium on Code Generation and

Optimization (CGO'06), Mar. 2006, Houston, EUA, **Proceedings of the IEEE International Symposium on Code Generation and Optimization (CGO'06)**, 2006.

COOPER, K. D.; DASGUPTA, A.; ECKHARDT, J. Revisiting Graph Coloring Register Allocation: A Study of Chaitin-Briggs and Callahan-Koblenz Algorithms. **Proc. of the Workshop on Languages and Compilers for Parallel Computing**, Out. 2005.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. 2. ed. London: The MIT Press & McGraw-Hill, 2001.

COSTA, P.; COULSON, G.; MASCOLO, C.; MOTOLLA, L; PICCO, G. P.; ZACHARIADIS, S. Reconfigurable Component-based Middleware for Network Embedded Systems. **International Journal of Wireless Information Networks**, v. 14, n. 02, p. 149-162, Jun. 2007.

DUARTE, F. L. **PHOENIX – Um *Framework* para Trabalhos de Síntese de Alto Nível de Circuitos Digitais**. 2006. 154 f. Dissertação (Mestrado em Ciência da Computação) – Faculdade de Computação, Universidade Federal de Uberlândia, Uberlândia, 2006.

ECLIPSE.org home. Disponível em: < http://www.eclipse.org/>. Acesso em: 07 mar. 2009.

EDWARDS, S. A.; ZENG, J. Code Generation in the Columbia Esterel Compiler. **EURASIP Journal on Embedded Systems**, v. 2007, n. 01, p. 1-31, 2007.

FERRARI, A. B.; SKLIAROVA, I. Reconfigurable Hardware SAT Solvers: A Survey of Systems. **IEEE Transactions on Computers**, v. 53, n. 11, p. 1449-1461, Nov. 2004.

GRIMMER, M.; HAMMER, C.; KRINKE, G. Dynamic Path Conditions in Dependence Graphs. In: ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation, Jan. 2006, Charleston, EUA, **Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation**, New York: 2006. p. 58-67.

GOLDSTEIN, S. C.; KOES, D. An Analysis of Graph Coloring Register Allocation. **Carnegie Mellon University Technical Report No. CMU-CS-06-111**, p. 10, Mar. 2006.

GONÇALVES, R. A. **ARCHITECT-R: Uma ferramenta para o Desenvolvimento de Robôs Reconfiguráveis**. 2002. 167 f. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2002.

GOSLING, J. Java Intermediate Bytecodes. In: SIGPLAN, 1995, San Francisco, EUA, **Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations**, New York: 1995. p. 111-118.

HADA, R.; HIRONAKA, T.; KOJIMA, A.; TANIGAWA, K. A Reconfigurable Processor PARS and its Compiler. In: International workshop on innovative architecture for future generation high-performance processors and systems, 2007, Maui, EUA, **International workshop on innovative architecture for future generation high-performance processors and systems**, p. 91-100.

HANNA, D. M.; HASKELL, R. E. Flowpaths: Compiling stack-based IR to hardware. **Microprocessors and Microsystems**, v. 30, n. 03, p. 125-136, Mai. 2006.

HENNESSY, J. L.; PATTERSON, D. A. **Arquitetura de Computadores**: Uma Abordagem Quantitativa. 3. ed. São Paulo: Editora Campus, 2003.

HOLLOWAY, G.; RAMSEY, N.; SMITH, M. D. A Generalized Algorithm for Graph-Coloring Register Allocation. **ACM SIGPLAN Notices**, v. 39, n. 06, p. 277-288, Jun. 2004.

HOLST, W.; STEPHENSON, B. An Evaluation of Specialized Java Bytecodes. In: Conference on Object Oriented Programming Systems Languages and Applications, 2006, Portland, EUA, **Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications**, New York: 2006. p. 671-672.

HOLST, W.; STEPHENSON, B. A Quantitative Analysis of the Performance Impact of Specialized Bytecodes in Java. In: IBM Centre for Advanced Studies Conference, 2004, Markham, Canadá, **Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research**, New York: 2004. p. 267-281.

HWANG, K. **Advanced Computer Architecture**: Parallelism, Scalability, Programmability. 1. ed. Singapore: McGraw-Hill, 1993.

KYRIAKOPOULOS, K.; PSARRIS, K. Efficient Techniques for Advanced Data Dependence Analysis. In: 14th IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'05), Set. 2005, San Antonio, EUA, **Proceedings of the 14th IEEE International Conference on Parallel Architectures and Compilation Techniques**, 2005. p. 143-153.

LAMBRIGHT, H. D. Java Bytecode Optimizations. **COMPCON**, v. 00, n. 00, p. 206-210, Fev. 1997.

LAUTERBACH **GmbH – Trace32 Microprocessor Development Tools, Emulators, Debuggers, Simulators**. Disponível em: <http://www.lauterbach.co.uk/frames.html?simnios.html>. Acesso em: 26 out. 2008.

LEE, G.; MILNE, G. Programming paradigms for reconfigurable computing. **Microprocessors and Microsystems**, v. 29, n. 10, p. 435-450, Dez. 2005.

LIMA, W. S.; LOBATO, SILVA, A. C. F.; R. S.; ULSON, R. S. Simulação de Execução de Instruções do Processador de Núcleo Virtual Nios II. In: XXXIV Conferência Latino Americana de Informática - CLEI 2008, 2008, Santa Fé, Argentina. **Anais da XXXIV Conferência Latinoamericana de Informática**, 2008. v. 1. p. 649-658.

LINDHOLM, T.; YELLIN, F. **The Java™ Virtual Machine Specification - Second Edition**. Addison-Wesley, 1999. Disponível em: <http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>. Acesso em: 25 jul. 2007.

MARQUES, E. **Ensino de hardware baseado em computação reconfigurável**. 2004. 160 f. Tese (Obtenção do Título de Livre-Docente) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2004.

MASKELL, D. L.; OLIVER, T. F.; SHIAN, L. Y. Reconfigurable Computing: Peripheral Power and Area Optimization Techniques. **TENCON 2005 IEEE Region 10**, v. 2007, n. 01, p. 1-4, Nov. 2005.

MENTOR Graphics – Additional Nucleus Supported CPUs. Embedded Software. Disponível em: <http://www.mentor.com/products/embedded_software/nucleus_cpu_support>. Acesso em: 11 mar. 2009.

MODELSIM Downloads. Disponível em: <http://www.model.com/downloads/default.asp>. Acesso em: 26 out. 2008.

MUCHNICK, S. S. **Advanced Compiler Design and Implementation**. 1. ed. San Francisco: Morgan Kaufmann Publishers, 1997.

NATARAJAN, S.; PERINBAN, R. P.; RAMADASS, N. Dynamically Reconfigurable (Self-modifiable) Architecture for Embedded System-on-Chip Applications**. Information Technology Journal**, v. 06, n. 01, p. 66-73, 2007.

NETBEANS – Welcome to NetBeans. Disponível em: < http://www.netbeans.org/>. Acesso em: 09 mar. 2009.

NIOS II Development Kit, Cyclone II Edition. Nios II Development Kit, Cyclone II Edition. Disponível em: <http://www.altera.com/products/devkits/altera/kit-nios-2c35.html>. Acesso em: 10 mar. 2009.

PADUA, D. A.; WOLFE M. J. Advanced Compiler Optimizations for Super-computers. **Communications of the ACM**, v. 29, n.02, p. 1184-1201, Dez. 1986.

ROSE, J.; STEFFAN, J. G.; YIANNACOURAS, P. Exploration and Customization of FPGA-Based Soft Processors. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 26, n. 02, p. 266-277, Fev. 2007.

SOUDRIS, D.; VASSILIADIS, S. ADRES & DRESC: Architecture and Compiler for Coarse-Grain Reconfigurable Processors. **Fine- and Coarse-Grain Reconfigurable Computing,** v. 01, n. 01, p. 255-297, Set. 2007.

SCHILDT, H. **C Completo e Total**. 3. ed. São Paulo: Makron Books, 1997.

TASKING – Nios II Software Development Tools. C/C++ compiler for Nios® II. Disponível em: < http://www.tasking.com/NiosII/>. Acesso em: 07 mar. 2009.

TPCI – TIOBE Programming Community Index. Estatísticas de uso de linguagens de programação para janeiro de 2009. Disponível em: <http://www.tiobe.com/tpci. htm>. Acesso em: 06 jan. 2009.

Apêndice A – Modelagem do compilador

Neste apêndice são apresentados os diagramas construídos para modelagem do JaNi. No modelo conceitual da seção A.1 pode-se observar os módulos existentes no nível de codificação do compilador e como os mesmos se relacionam. A maneira como esses módulos interagem é apresentada por meio do diagrama de seqüências apresentado na seção A.2. Por fim, a visão externa de todo o processo é simplificada no digrama de casos de uso da seção A.3.

A.1 Modelo conceitual

Na Figura A.1 é apresentado o modelo conceitual dos módulos do JaNi em nível de desenvolvimento. Cada um desses módulos é, fisicamente, um ou dois arquivos de código-fonte contendo definições ou implementações.

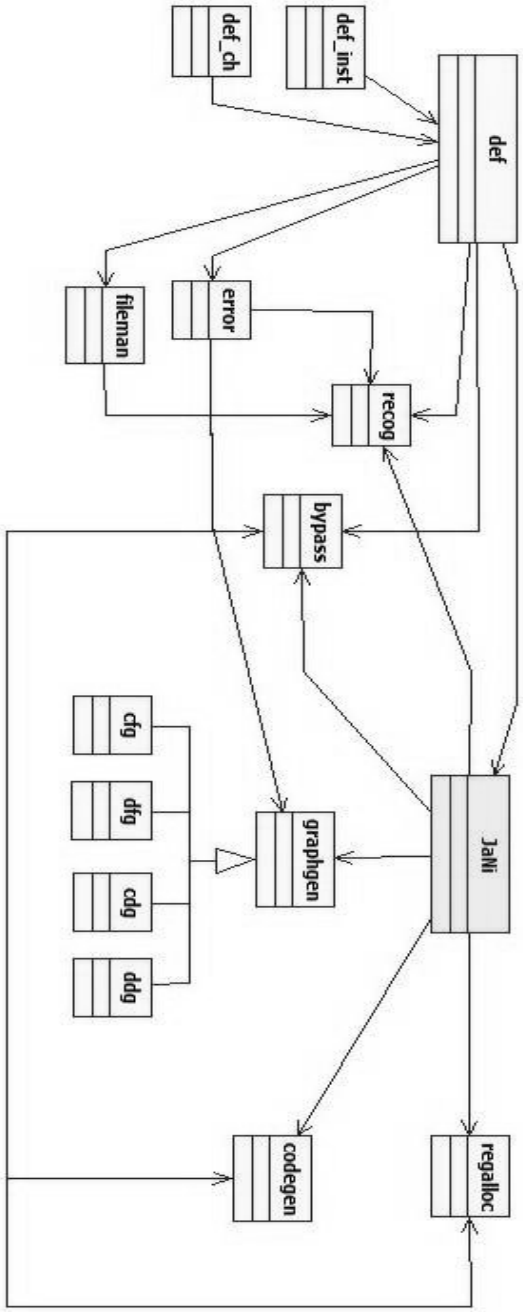

Figura A.1 Modelo conceitual do JaNi.

A.2 Digrama de seqüência

O diagrama de seqüência para as principais ações executadas entre os módulos de desenvolvimento do JaNi é mostrado na Figura A.2.

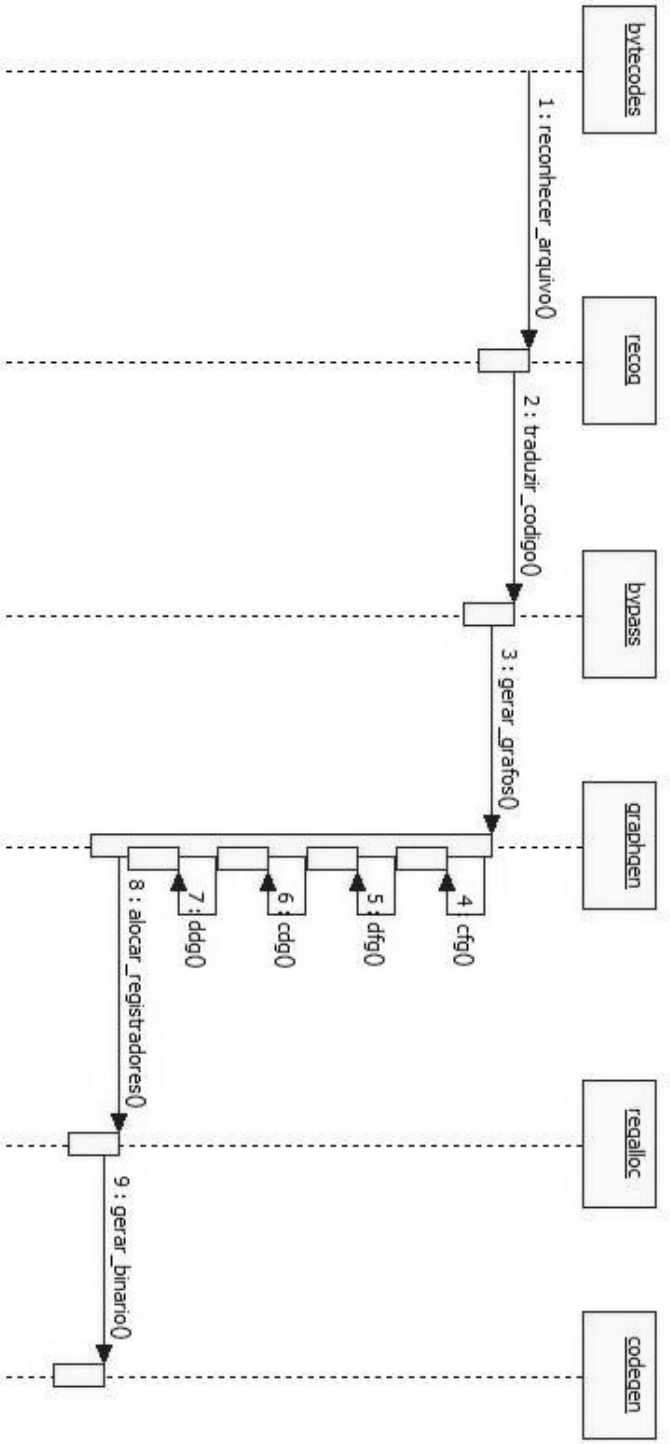

Figura A.2 Diagrama de seqüência da execução do JaNi.

A.3 Diagrama de casos de uso

Na figura A.3 é apresentado, de maneira simplificada, o diagrama de casos de uso para o processo que envolve a utilização do JaNi. Pode-se perceber que o programador deve realizar a compilação de um código Java em um compilador *front-end* para a geração dos *bytecodes*, possuindo também o papel de fornecer este arquivo intermediário gerado ao JaNi para seu processamento. A simulação do arquivo binário originado pelo compilador também é desempenhada pelo programador.

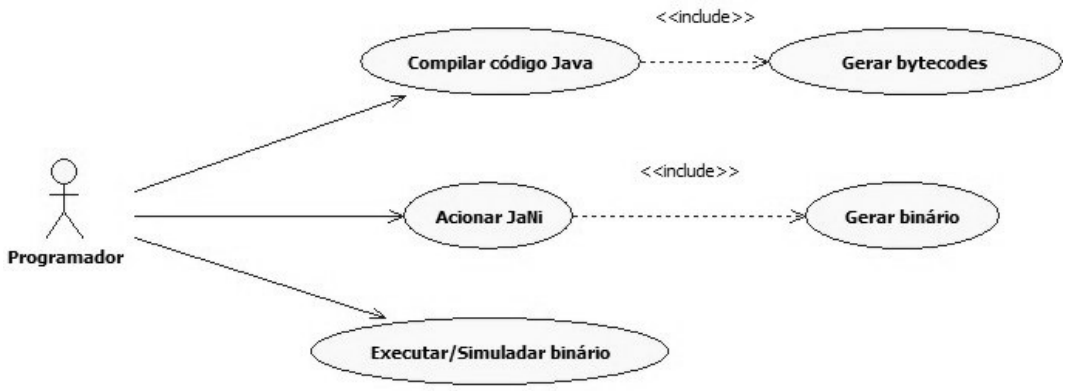

Figura A.3 Diagrama de casos uso para o JaNi.

Apêndice B – Registradores especiais do JaNi

Neste apêndice são apresentados os registradores do Nios II reservados pelo JaNi para controle de estruturas do aplicativo sendo compilado. Esses registradores estão reservados para tais finalidades e não são liberados para uso das variáveis.

Tabela B.1 Registradores especiais do JaNi.

Registrador	Descrição
`r20`	Utilizado quando há a necessidade de manipulação de *spill code*. Os dados lidos da memória são inicialmente armazenados neste registrador. Da mesma forma, os dados a serem gravados em uma posição de memória são lidos deste repositório.
`r21`	Armazena o tamanho do último arranjo criado, servindo de parâmetro para a definição do endereço a ser ocupado pelos arranjos declarados posteriormente no código.
`r22`	Armazena o endereço de início do espaço de memória manipulado pelo programa sendo compilado onde os arranjos serão armazenados.
`r23`	Armazena o endereço de início do espaço de memória manipulado pelo programa sendo compilado onde as variáveis que não puderem ser guardadas em registradores devem ser armazenadas.

Apêndice C – Estruturas produzidas na compilação

O objetivo deste apêndice é apresentar com todos os detalhes os dados produzidos pelo JaNi para o programa da Figura 5.6 utilizado nos testes da seção 5.4. Na seção C.1 são apresentados os dados do arquivo intermediário reconhecido, sendo que os *bytecodes* a serem processados são mostrados por meio da seção C.2. O código intermediário primário gerado pela camada BT é exibido na seção C.3. Os grafos de representação intermediária são detalhados nas seções de C.4 a C.7, e o código após a etapa de alocação de registradores na seção C.8, sendo que este corresponde ao código final obtido.

Na seção C.7, são exibidas as dependências das 30 primeiras instruções do programa, devido ao seu tamanho.

C.1 Dados do arquivo-parâmetro

```
Class file properties:
'magic = 0xCAFEBABE
'version = 49.0
'constant_pool_count = 35
'constant_pool:
'  1: cf.constant_pool[1].tag = 10
      ''this entry is a method info with properties:
      ''> class index: 3
      ''> name and type index: 32
'  2: cf.constant_pool[2].tag = 7
      ''this entry is a class info with properties:
      ''> name index: 33
'  3: cf.constant_pool[3].tag = 7
```

```
      ''this entry is a class info with properties:
      ''> name index: 34
'  4: cf.constant_pool[4].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 6
      ''> string: <init>
'  5: cf.constant_pool[5].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 3
      ''> string: ()V
'  6: cf.constant_pool[6].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 4
      ''> string: Code
'  7: cf.constant_pool[7].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 15
      ''> string: LineNumberTable
'  8: cf.constant_pool[8].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 18
      ''> string: LocalVariableTable
'  9: cf.constant_pool[9].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 4
      ''> string: this
' 10: cf.constant_pool[10].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 22
      ''> string: Ljanitestcrucial/Main;
' 11: cf.constant_pool[11].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 4
      ''> string: main
' 12: cf.constant_pool[12].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 22
      ''> string: ([Ljava/lang/String;)V
' 13: cf.constant_pool[13].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 3
      ''> string: ch1
' 14: cf.constant_pool[14].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 1
      ''> string: C
' 15: cf.constant_pool[15].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 1
      ''> string: i
' 16: cf.constant_pool[16].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 1
      ''> string: I
' 17: cf.constant_pool[17].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 4
      ''> string: args
' 18: cf.constant_pool[18].tag = 1
      ''this entry is a string Ut8 with properties:
```

```
      ''> length: 19
      ''> string: [Ljava/lang/String;
' 19: cf.constant_pool[19].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 6
      ''> string: entr01
' 20: cf.constant_pool[20].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 1
      ''> string: S
' 21: cf.constant_pool[21].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 6
      ''> string: entr02
' 22: cf.constant_pool[22].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 1
      ''> string: a
' 23: cf.constant_pool[23].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 1
      ''> string: b
' 24: cf.constant_pool[24].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 1
      ''> string: c
' 25: cf.constant_pool[25].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 3
      ''> string: ch2
' 26: cf.constant_pool[26].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 4
      ''> string: flag
' 27: cf.constant_pool[27].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 1
      ''> string: Z
' 28: cf.constant_pool[28].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 5
      ''> string: array
' 29: cf.constant_pool[29].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 2
      ''> string: [I
' 30: cf.constant_pool[30].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 10
      ''> string: SourceFile
' 31: cf.constant_pool[31].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 9
      ''> string: Main.java
' 32: cf.constant_pool[32].tag = 12
      ''this entry is a name and type info with properties:
      ''> name index: 4
      ''> descriptor index: 5
' 33: cf.constant_pool[33].tag = 1
      ''this entry is a string Ut8 with properties:
```

```
      ''> length: 20
      ''> string: janitestcrucial/Main
' 34: cf.constant_pool[34].tag = 1
      ''this entry is a string Ut8 with properties:
      ''> length: 16
      ''> string: java/lang/Object
'access_flags = 0x0021
'this_class = 2
'super_class = 3
'interfaces_count = 0
'interfaces:
'fields_count = 0
'methods_count = 2
'  0: cf.methods[0].access_flags = 1
'  0: cf.methods[0].name_index = 4
'  0: cf.methods[0].descriptor_index = 5
'  0: cf.methods[0].attributes_count = 1
'     0: cf.methods[0].attributes[0].attr_name_index = 6
'     0: cf.methods[0].attributes[0].attr_length = 47
'  1: cf.methods[1].access_flags = 9
'  1: cf.methods[1].name_index = 11
'  1: cf.methods[1].descriptor_index = 12
'  1: cf.methods[1].attributes_count = 1
'     1: cf.methods[1].attributes[0].attr_name_index = 6
'     1: cf.methods[1].attributes[0].attr_length = 446
'attributes_count = 1
'  0: cf.attributes[0].attr_name_index = 30
'  0: cf.attributes[0].attr_length = 2
```

C.2 *Bytecodes* encontrados

```
max_stack = 5, max_locals = 11, code_length = 182
BIPUSH (2)
ISTORE_1 (1)
ILOAD_1 (1)
BIPUSH (2)
IADD (1)
INT2SHORT (1)
ISTORE_2 (1)
ICONST_0 (1)
ISTORE_3 (1)
SIPUSH (3)
ISTORE (2)
ICONST_0 (1)
ISTORE (2)
BIPUSH (2)
NEWARRAY (2)
ASTORE (2)
ILOAD (2)
SIPUSH (3)
IF_ICMPLT (3)
ILOAD_3 (1)
IFNE (3)
BIPUSH (2)
ISTORE (2)
GOTO (3)
```

```
BIPUSH (2)
ISTORE (2)
BIPUSH (2)
ISTORE (2)
ILOAD_1 (1)
IFGT (3)
ILOAD_2 (1)
IFGE (3)
ILOAD_1 (1)
ICONST_M1 (1)
IMUL (1)
INT2SHORT (1)
ISTORE_1 (1)
BIPUSH (2)
ISTORE (2)
GOTO (3)
ILOAD (2)
BIPUSH (2)
IF_ICMPEQ (3)
BIPUSH (2)
ISTORE (2)
ILOAD_1 (1)
ICONST_2 (1)
IDIV (1)
INT2SHORT (1)
ISTORE_1 (1)
BIPUSH (2)
ISTORE (2)
IINC (3)
ILOAD (2)
ICONST_M1 (1)
IF_ICMPLE (3)
ALOAD (2)
ILOAD (2)
ILOAD (2)
INEG (1)
IASTORE (1)
GOTO (3)
IINC (3)
ICONST_0 (1)
ISTORE (2)
ILOAD (2)
BIPUSH (2)
IF_ICMPGE (3)
ILOAD (2)
IFEQ (3)
ALOAD (2)
ILOAD (2)
DUP2 (1)
IALOAD (1)
ILOAD (2)
ICONST_5 (1)
IDIV (1)
IADD (1)
IASTORE (1)
ILOAD (2)
IFNE (3)
ICONST_1 (1)
GOTO (3)
ICONST_0 (1)
```

```
ISTORE (2)
IINC (3)
GOTO (3)
ILOAD_1 (1)
IFLE (3)
ILOAD_1 (1)
ICONST_1 (1)
ISUB (1)
INT2SHORT (1)
ISTORE_1 (1)
GOTO (3)
ILOAD_1 (1)
ICONST_1 (1)
IADD (1)
INT2SHORT (1)
ISTORE_1 (1)
ILOAD_1 (1)
IFNE (3)
RETURN (1)
```

C.3 Código intermediário primário

```
000 - ADDI: Ra = 0, Rb = 23, Rc = 0 e imm16 = 4
001 - ADDI: Ra = 0, Rb = 22, Rc = 0 e imm16 = 8
002 - ADDI: Ra = 0, Rb = 21, Rc = 0 e imm16 = 0
003 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = -34
004 - ADD: Ra = 0, Rb = 43, Rc = 33 e imm16 = 0
005 - ADD: Ra = 0, Rb = 33, Rc = 43 e imm16 = 0
006 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 75
007 - ADD: Ra = 44, Rb = 43, Rc = 43 e imm16 = 0
008 - ADD: Ra = 0, Rb = 43, Rc = 34 e imm16 = 0
009 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 0
010 - ADD: Ra = 0, Rb = 43, Rc = 35 e imm16 = 0
011 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 236
012 - ADD: Ra = 0, Rb = 43, Rc = 36 e imm16 = 0
013 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 0
014 - ADD: Ra = 0, Rb = 43, Rc = 40 e imm16 = 0
015 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 13
016 - MULI: Ra = 43, Rb = 43, Rc = 0 e imm16 = 4
017 - ADD: Ra = 43, Rb = 21, Rc = 44 e imm16 = 0
018 - ADDI: Ra = 43, Rb = 21, Rc = 0 e imm16 = 0
019 - ADDI: Ra = 43, Rb = 44, Rc = 0 e imm16 = 0
020 - ADD: Ra = 0, Rb = 43, Rc = 41 e imm16 = 0
021 - ADD: Ra = 0, Rb = 36, Rc = 43 e imm16 = 0
022 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 230
023 - CMPLT: Ra = 43, Rb = 44, Rc = 43 e imm16 = 0
024 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 1
025 - BEQ: Ra = 43, Rb = 44, Rc = 0 e imm16 = 28
026 - ADD: Ra = 0, Rb = 35, Rc = 43 e imm16 = 0
027 - CMPNE: Ra = 43, Rb = 0, Rc = 43 e imm16 = 0
028 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 1
029 - BEQ: Ra = 43, Rb = 44, Rc = 0 e imm16 = 12
030 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 88
031 - ADD: Ra = 0, Rb = 43, Rc = 38 e imm16 = 0
032 - BR: Ra = 0, Rb = 0, Rc = 0 e imm16 = 8
033 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 89
```

```
034 - ADD: Ra = 0, Rb = 43, Rc = 38 e imm16 = 0
035 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 76
036 - ADD: Ra = 0, Rb = 43, Rc = 39 e imm16 = 0
037 - ADD: Ra = 0, Rb = 33, Rc = 43 e imm16 = 0
038 - CMPLT: Ra = 0, Rb = 43, Rc = 43 e imm16 = 0
039 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 1
040 - BEQ: Ra = 43, Rb = 44, Rc = 0 e imm16 = 16
041 - ADD: Ra = 0, Rb = 34, Rc = 43 e imm16 = 0
042 - CMPGE: Ra = 43, Rb = 0, Rc = 43 e imm16 = 0
043 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 1
044 - BEQ: Ra = 43, Rb = 44, Rc = 0 e imm16 = 28
045 - ADD: Ra = 0, Rb = 33, Rc = 43 e imm16 = 0
046 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = -1
047 - MUL: Ra = 44, Rb = 43, Rc = 43 e imm16 = 0
048 - ADD: Ra = 0, Rb = 43, Rc = 33 e imm16 = 0
049 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 87
050 - ADD: Ra = 0, Rb = 43, Rc = 39 e imm16 = 0
051 - BR: Ra = 0, Rb = 0, Rc = 0 e imm16 = 44
052 - ADD: Ra = 0, Rb = 38, Rc = 43 e imm16 = 0
053 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 88
054 - CMPEQ: Ra = 43, Rb = 44, Rc = 43 e imm16 = 0
055 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 1
056 - BEQ: Ra = 43, Rb = 44, Rc = 0 e imm16 = 8
057 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 73
058 - ADD: Ra = 0, Rb = 43, Rc = 39 e imm16 = 0
059 - ADD: Ra = 0, Rb = 33, Rc = 43 e imm16 = 0
060 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 2
061 - DIV: Ra = 43, Rb = 44, Rc = 43 e imm16 = 0
062 - ADD: Ra = 0, Rb = 43, Rc = 33 e imm16 = 0
063 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 13
064 - ADD: Ra = 0, Rb = 43, Rc = 37 e imm16 = 0
065 - ADDI: Ra = 37, Rb = 37, Rc = 0 e imm16 = -1
066 - ADD: Ra = 0, Rb = 37, Rc = 43 e imm16 = 0
067 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = -1
068 - CMPGE: Ra = 44, Rb = 43, Rc = 43 e imm16 = 0
069 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 1
070 - BEQ: Ra = 43, Rb = 44, Rc = 0 e imm16 = 36
071 - ADD: Ra = 0, Rb = 41, Rc = 43 e imm16 = 0
072 - ADD: Ra = 0, Rb = 37, Rc = 44 e imm16 = 0
073 - ADD: Ra = 0, Rb = 37, Rc = 45 e imm16 = 0
074 - MULI: Ra = 45, Rb = 45, Rc = 0 e imm16 = -1
075 - MULI: Ra = 44, Rb = 44, Rc = 0 e imm16 = 4
076 - ADD: Ra = 44, Rb = 43, Rc = 46 e imm16 = 0
077 - ADD: Ra = 46, Rb = 22, Rc = 46 e imm16 = 0
078 - STW: Ra = 46, Rb = 45, Rc = 0 e imm16 = 0
079 - BR: Ra = 0, Rb = 0, Rc = 0 e imm16 = -60
080 - ADDI: Ra = 36, Rb = 36, Rc = 0 e imm16 = 1
081 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 0
082 - ADD: Ra = 0, Rb = 43, Rc = 42 e imm16 = 0
083 - ADD: Ra = 0, Rb = 42, Rc = 43 e imm16 = 0
084 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 13
085 - CMPGE: Ra = 43, Rb = 44, Rc = 43 e imm16 = 0
086 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 1
087 - BEQ: Ra = 43, Rb = 44, Rc = 0 e imm16 = 120
088 - ADD: Ra = 0, Rb = 40, Rc = 43 e imm16 = 0
089 - CMPEQ: Ra = 43, Rb = 0, Rc = 43 e imm16 = 0
090 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 1
091 - BEQ: Ra = 43, Rb = 44, Rc = 0 e imm16 = 64
092 - ADD: Ra = 0, Rb = 41, Rc = 43 e imm16 = 0
093 - ADD: Ra = 0, Rb = 42, Rc = 44 e imm16 = 0
```

```
094 - ADD: Ra = 44, Rb = 0, Rc = 46 e imm16 = 0
095 - ADD: Ra = 43, Rb = 0, Rc = 45 e imm16 = 0
096 - MULI: Ra = 46, Rb = 46, Rc = 0 e imm16 = 4
097 - ADD: Ra = 45, Rb = 46, Rc = 46 e imm16 = 0
098 - ADD: Ra = 46, Rb = 22, Rc = 46 e imm16 = 0
099 - LDW: Ra = 46, Rb = 45, Rc = 0 e imm16 = 0
100 - ADD: Ra = 0, Rb = 36, Rc = 46 e imm16 = 0
101 - ADDI: Ra = 0, Rb = 47, Rc = 0 e imm16 = 5
102 - DIV: Ra = 46, Rb = 47, Rc = 46 e imm16 = 0
103 - ADD: Ra = 46, Rb = 45, Rc = 45 e imm16 = 0
104 - MULI: Ra = 44, Rb = 44, Rc = 0 e imm16 = 4
105 - ADD: Ra = 44, Rb = 43, Rc = 46 e imm16 = 0
106 - ADD: Ra = 46, Rb = 22, Rc = 46 e imm16 = 0
107 - STW: Ra = 46, Rb = 45, Rc = 0 e imm16 = 0
108 - ADD: Ra = 0, Rb = 40, Rc = 43 e imm16 = 0
109 - CMPNE: Ra = 43, Rb = 0, Rc = 43 e imm16 = 0
110 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 1
111 - BEQ: Ra = 43, Rb = 44, Rc = 0 e imm16 = 8
112 - ADDI: Ra = 0, Rb = 43, Rc = 0 e imm16 = 1
113 - BR: Ra = 0, Rb = 0, Rc = 0 e imm16 = 4
114 - ADDI: Ra = 0, Rb = 44, Rc = 0 e imm16 = 0
115 - ADD: Ra = 0, Rb = 44, Rc = 40 e imm16 = 0
116 - ADDI: Ra = 42, Rb = 42, Rc = 0 e imm16 = 1
117 - BR: Ra = 0, Rb = 0, Rc = 0 e imm16 = -140
118 - ADD: Ra = 0, Rb = 33, Rc = 44 e imm16 = 0
119 - CMPGE: Ra = 0, Rb = 44, Rc = 44 e imm16 = 0
120 - ADDI: Ra = 0, Rb = 45, Rc = 0 e imm16 = 1
121 - BEQ: Ra = 44, Rb = 45, Rc = 0 e imm16 = 20
122 - ADD: Ra = 0, Rb = 33, Rc = 44 e imm16 = 0
123 - ADDI: Ra = 0, Rb = 45, Rc = 0 e imm16 = 1
124 - SUB: Ra = 44, Rb = 45, Rc = 44 e imm16 = 0
125 - ADD: Ra = 0, Rb = 44, Rc = 33 e imm16 = 0
126 - BR: Ra = 0, Rb = 0, Rc = 0 e imm16 = 16
127 - ADD: Ra = 0, Rb = 33, Rc = 44 e imm16 = 0
128 - ADDI: Ra = 0, Rb = 45, Rc = 0 e imm16 = 1
129 - ADD: Ra = 45, Rb = 44, Rc = 44 e imm16 = 0
130 - ADD: Ra = 0, Rb = 44, Rc = 33 e imm16 = 0
131 - ADD: Ra = 0, Rb = 33, Rc = 44 e imm16 = 0
132 - CMPNE: Ra = 44, Rb = 0, Rc = 44 e imm16 = 0
133 - ADDI: Ra = 0, Rb = 45, Rc = 0 e imm16 = 1
134 - BEQ: Ra = 44, Rb = 45, Rc = 0 e imm16 = -220
135 - BR: Ra = 0, Rb = 0, Rc = 0 e imm16 = 4
```

C.4 Grafo de fluxo de controle

```
INFO: 26 basic blocks were found!
B0 (0 - 25)
B1 (26 - 29)
B2 (30 - 32)
B3 (33 - 34)
B4 (35 - 40)
B5 (41 - 44)
B6 (45 - 51)
B7 (52 - 56)
B8 (57 - 58)
B9 (59 - 62)
```

```
B10 (63 - 64)
B11 (65 - 70)
B12 (71 - 79)
B13 (80 - 82)
B14 (83 - 87)
B15 (88 - 91)
B16 (92 - 107)
B17 (108 - 111)
B18 (112 - 113)
B19 (114 - 114)
B20 (115 - 117)
B21 (118 - 121)
B22 (122 - 126)
B23 (127 - 130)
B24 (131 - 134)
B25 (135 - 135)

B0 -> B1
B0 -> B3
B1 -> B2
B1 -> B3
B2 -> B4
B3 -> B4
B4 -> B5
B4 -> B6
B5 -> B6
B5 -> B7
B6 -> B10
B7 -> B8
B7 -> B9
B8 -> B9
B9 -> B10
B10 -> B11
B11 -> B12
B11 -> B13
B12 -> B11
B13 -> B14
B14 -> B15
B14 -> B21
B15 -> B16
B15 -> B17
B16 -> B17
B17 -> B18
B17 -> B19
B18 -> B20
B19 -> B20
B20 -> B14
B21 -> B22
B21 -> B23
B22 -> B24
B23 -> B24
B24 -> B25
B24 -> B13
```

C.5 Grafo de fluxo de dados

```
Instr. 000:
Instr. 001:  77  98  106
Instr. 002:  17
Instr. 003:  4
Instr. 004:  5  37  45  59
Instr. 005:  7
Instr. 006:  7
Instr. 007:  8
Instr. 008:  41
Instr. 009:  10
Instr. 010:  26
Instr. 011:  12
Instr. 012:  21  80
Instr. 013:  14
Instr. 014:  88  108
Instr. 015:  16
Instr. 016:  17  18  19  20
Instr. 017:
Instr. 018:
Instr. 019:
Instr. 020:  71  92
Instr. 021:  23
Instr. 022:  23
Instr. 023:  25
Instr. 024:  25
Instr. 025:
Instr. 026:  27
Instr. 027:  29
Instr. 028:  29
Instr. 029:
Instr. 030:  31
Instr. 031:  52
Instr. 032:
Instr. 033:  34
Instr. 034:
Instr. 035:  36
Instr. 036:
Instr. 037:  38
Instr. 038:  40
Instr. 039:  40
Instr. 040:
Instr. 041:  42
Instr. 042:  44
Instr. 043:  44
Instr. 044:
Instr. 045:  47
Instr. 046:  47
Instr. 047:  48
Instr. 048:  118  122  127
Instr. 049:  50
Instr. 050:
Instr. 051:
Instr. 052:  54
Instr. 053:  54
Instr. 054:  56
Instr. 055:  56
```

```
Instr. 056:
Instr. 057:  58
Instr. 058:
Instr. 059:  61
Instr. 060:  61
Instr. 061:  62
Instr. 062:
Instr. 063:  64
Instr. 064:
Instr. 065:  66  72  73  65
Instr. 066:  68
Instr. 067:  68
Instr. 068:  70
Instr. 069:  70
Instr. 070:
Instr. 071:  76
Instr. 072:  75
Instr. 073:  74
Instr. 074:  78
Instr. 075:  76
Instr. 076:  77
Instr. 077:  78
Instr. 078:
Instr. 079:
Instr. 080:  100  80
Instr. 081:  82
Instr. 082:
Instr. 083:  85
Instr. 084:  85
Instr. 085:  87
Instr. 086:  87
Instr. 087:
Instr. 088:  89
Instr. 089:  91
Instr. 090:  91
Instr. 091:
Instr. 092:  95  105
Instr. 093:  94  104
Instr. 094:  96
Instr. 095:  97
Instr. 096:  97
Instr. 097:  98
Instr. 098:  99
Instr. 099:  103
Instr. 100:  102
Instr. 101:  102
Instr. 102:  103
Instr. 103:  107
Instr. 104:  105
Instr. 105:  106
Instr. 106:  107
Instr. 107:
Instr. 108:  109
Instr. 109:  111
Instr. 110:  111  115
Instr. 111:
Instr. 112:
Instr. 113:
Instr. 114:  115
Instr. 115:  88  108
```

```
Instr. 116:  83  93  116
Instr. 117:
Instr. 118:  119
Instr. 119:  121
Instr. 120:  121
Instr. 121:
Instr. 122:  124
Instr. 123:  124
Instr. 124:  125
Instr. 125:  131  118  122  127
Instr. 126:
Instr. 127:  129
Instr. 128:  129
Instr. 129:  130
Instr. 130:
Instr. 131:  132
Instr. 132:  134
Instr. 133:  134
Instr. 134:
Instr. 135:
# of edges: 129
```

C.6 Grafo de dependência de controle

```
B0 -> B4 B10 B11 B13 B14 B21 B24
B1 -> B4 B10 B11 B13 B14 B21 B24
B2 -> B4 B10 B11 B13 B14 B21 B24
B3 -> B4 B10 B11 B13 B14 B21 B24
B4 -> B10 B11 B13 B14 B21 B24
B5 -> B10 B11 B13 B14 B21 B24
B6 -> B10 B11 B13 B14 B21 B24
B7 -> B9 B10 B11 B13 B14 B21 B24
B8 -> B9 B10 B11 B13 B14 B21 B24
B9 -> B10 B11 B13 B14 B21 B24
B10 -> B11 B13 B14 B21 B24
B11 -> B13 B14 B21 B24
B12 -> B11 B13 B14 B21 B24
B13 -> B14 B21 B24
B14 -> B21 B24
B15 -> B14 B17 B20 B21 B24
B16 -> B14 B17 B20 B21 B24
B17 -> B14 B20 B21 B24
B18 -> B14 B20 B21 B24
B19 -> B14 B20 B21 B24
B20 -> B14 B21 B24
B21 -> B24
B22 -> B24
B23 -> B24
B24 ->
B25 ->

# of edges: 113
```

C.7 Grafo de dependência de dados

```
Instr. 000:  0(1)
Instr. 001:  1(1)  77(0)  98(0)  106(0)
Instr. 002:  2(1)  17(0)  18(1)
Instr. 003:  3(1)  4(0)  5(1)  7(1)  8(0)  9(1)  10(0)  11(1)  12(0)
13(1)  14(0)  15(1)  16(1)  17(0)  18(0)  19(0)  20(0)  21(1)  23(1)
25(0)  26(1)  27(1)  29(0)  30(1)  31(0)  33(1)  34(0)  35(1)  36(0)
37(1)  38(1)  40(0)  41(1)  42(1)  44(0)  45(1)  47(1)  48(0)  49(1)
50(0)  52(1)  54(1)  56(0)  57(1)  58(0)  59(1)  61(1)  62(0)  63(1)
64(0)  66(1)  68(1)  70(0)  71(1)  76(0)  81(1)  82(0)  83(1)  85(1)
87(0)  88(1)  89(1)  91(0)  92(1)  95(0)  105(0)  108(1)  109(1)
111(0)  112(1)
Instr. 004:  3(2)  4(1)  5(0)  7(2)  9(2)  11(2)  13(2)  15(2)
16(2)  21(2)  23(2)  26(2)  27(2)  30(2)  33(2)  35(2)  37(0)  38(2)
41(2)  42(2)  45(0)  47(2)  48(1)  49(2)  52(2)  54(2)  57(2)  59(0)
61(2)  62(1)  63(2)  66(2)  68(2)  71(2)  81(2)  83(2)  85(2)  88(2)
89(2)  92(2)  108(2)  109(2)  112(2)  118(0)  122(0)  125(1)  127(0)
130(1)  131(0)
Instr. 005:  3(1)  4(0)  5(1)  7(1)  8(0)  9(1)  10(0)  11(1)  12(0)
13(1)  14(0)  15(1)  16(1)  17(0)  18(0)  19(0)  20(0)  21(1)  23(1)
25(0)  26(1)  27(1)  29(0)  30(1)  31(0)  33(1)  34(0)  35(1)  36(0)
37(1)  38(1)  40(0)  41(1)  42(1)  44(0)  45(1)  47(1)  48(0)  49(1)
50(0)  52(1)  54(1)  56(0)  57(1)  58(0)  59(1)  61(1)  62(0)  63(1)
64(0)  66(1)  68(1)  70(0)  71(1)  76(0)  81(1)  82(0)  83(1)  85(1)
87(0)  88(1)  89(1)  91(0)  92(1)  95(0)  105(0)  108(1)  109(1)
111(0)  112(1)  125(2)  130(2)
Instr. 006:  6(1)  7(0)  17(1)  19(1)  22(1)  23(0)  24(1)  25(0)
28(1)  29(0)  39(1)  40(0)  43(1)  44(0)  46(1)  47(0)  53(1)  54(0)
55(1)  56(0)  60(1)  61(0)  67(1)  68(0)  69(1)  70(0)  72(1)  75(1)
76(0)  84(1)  85(0)  86(1)  87(0)  90(1)  91(0)  93(1)  94(0)
104(1)  105(0)  110(1)  111(0)  114(1)  115(0)  118(1)  119(1)
121(0)  122(1)  124(1)  125(0)  127(1)  129(1)  130(0)  131(1)
132(1)  134(0)
Instr. 007:  3(1)  4(0)  5(1)  6(2)  7(1)  8(0)  9(1)  10(0)  11(1)
12(0)  13(1)  14(0)  15(1)  16(1)  17(0)  18(0)  19(0)  20(0)  21(1)
22(2)  23(1)  24(2)  25(0)  26(1)  27(1)  28(2)  29(0)  30(1)  31(0)
33(1)  34(0)  35(1)  36(0)  37(1)  38(1)  39(2)  40(0)  41(1)  42(1)
43(2)  44(0)  45(1)  46(2)  47(1)  48(0)  49(1)  50(0)  52(1)  53(2)
54(1)  55(2)  56(0)  57(1)  58(0)  59(1)  60(2)  61(1)  62(0)  63(1)
64(0)  66(1)  67(2)  68(1)  69(2)  70(0)  71(1)  72(2)  75(2)  76(0)
81(1)  82(0)  83(1)  84(2)  85(1)  86(2)  87(0)  88(1)  89(1)  90(2)
91(0)  92(1)  93(2)  95(0)  104(2)  105(0)  108(1)  109(1)  110(2)
111(0)  112(1)  114(2)  118(2)  119(2)  122(2)  124(2)  127(2)
129(2)  131(2)  132(2)
Instr. 008:  3(2)  5(2)  7(2)  8(1)  9(2)  11(2)  13(2)  15(2)
16(2)  21(2)  23(2)  26(2)  27(2)  30(2)  33(2)  35(2)  37(2)  38(2)
41(0)  42(2)  45(2)  47(2)  49(2)  52(2)  54(2)  57(2)  59(2)  61(2)
63(2)  66(2)  68(2)  71(2)  81(2)  83(2)  85(2)  88(2)  89(2)  92(2)
108(2)  109(2)  112(2)
Instr. 009:  3(1)  4(0)  5(1)  7(1)  8(0)  9(1)  10(0)  11(1)  12(0)
13(1)  14(0)  15(1)  16(1)  17(0)  18(0)  19(0)  20(0)  21(1)  23(1)
25(0)  26(1)  27(1)  29(0)  30(1)  31(0)  33(1)  34(0)  35(1)  36(0)
37(1)  38(1)  40(0)  41(1)  42(1)  44(0)  45(1)  47(1)  48(0)  49(1)
50(0)  52(1)  54(1)  56(0)  57(1)  58(0)  59(1)  61(1)  62(0)  63(1)
64(0)  66(1)  68(1)  70(0)  71(1)  76(0)  81(1)  82(0)  83(1)  85(1)
87(0)  88(1)  89(1)  91(0)  92(1)  95(0)  105(0)  108(1)  109(1)
111(0)  112(1)
```

Instr. 010: 3(2) 5(2) 7(2) 9(2) 10(1) 11(2) 13(2) 15(2)
16(2) 21(2) 23(2) 26(0) 27(2) 30(2) 33(2) 35(2) 37(2) 38(2)
41(2) 42(2) 45(2) 47(2) 49(2) 52(2) 54(2) 57(2) 59(2) 61(2)
63(2) 66(2) 68(2) 71(2) 81(2) 83(2) 85(2) 88(2) 89(2) 92(2)
108(2) 109(2) 112(2)
Instr. 011: 3(1) 4(0) 5(1) 7(1) 8(0) 9(1) 10(0) 11(1) 12(0)
13(1) 14(0) 15(1) 16(1) 17(0) 18(0) 19(0) 20(0) 21(1) 23(1)
25(0) 26(1) 27(1) 29(0) 30(1) 31(0) 33(1) 34(0) 35(1) 36(0)
37(1) 38(1) 40(0) 41(1) 42(1) 44(0) 45(1) 47(1) 48(0) 49(1)
50(0) 52(1) 54(1) 56(0) 57(1) 58(0) 59(1) 61(1) 62(0) 63(1)
64(0) 66(1) 68(1) 70(0) 71(1) 76(0) 81(1) 82(0) 83(1) 85(1)
87(0) 88(1) 89(1) 91(0) 92(1) 95(0) 105(0) 108(1) 109(1)
111(0) 112(1)
Instr. 012: 3(2) 5(2) 7(2) 9(2) 11(2) 12(1) 13(2) 15(2)
16(2) 21(0) 23(2) 26(2) 27(2) 30(2) 33(2) 35(2) 37(2) 38(2)
41(2) 42(2) 45(2) 47(2) 49(2) 52(2) 54(2) 57(2) 59(2) 61(2)
63(2) 66(2) 68(2) 71(2) 80(1) 81(2) 83(2) 85(2) 88(2) 89(2)
92(2) 100(0) 108(2) 109(2) 112(2)
Instr. 013: 3(1) 4(0) 5(1) 7(1) 8(0) 9(1) 10(0) 11(1) 12(0)
13(1) 14(0) 15(1) 16(1) 17(0) 18(0) 19(0) 20(0) 21(1) 23(1)
25(0) 26(1) 27(1) 29(0) 30(1) 31(0) 33(1) 34(0) 35(1) 36(0)
37(1) 38(1) 40(0) 41(1) 42(1) 44(0) 45(1) 47(1) 48(0) 49(1)
50(0) 52(1) 54(1) 56(0) 57(1) 58(0) 59(1) 61(1) 62(0) 63(1)
64(0) 66(1) 68(1) 70(0) 71(1) 76(0) 81(1) 82(0) 83(1) 85(1)
87(0) 88(1) 89(1) 91(0) 92(1) 95(0) 105(0) 108(1) 109(1)
111(0) 112(1)
Instr. 014: 3(2) 5(2) 7(2) 9(2) 11(2) 13(2) 14(1) 15(2)
16(2) 21(2) 23(2) 26(2) 27(2) 30(2) 33(2) 35(2) 37(2) 38(2)
41(2) 42(2) 45(2) 47(2) 49(2) 52(2) 54(2) 57(2) 59(2) 61(2)
63(2) 66(2) 68(2) 71(2) 81(2) 83(2) 85(2) 88(0) 89(2) 92(2)
108(0) 109(2) 112(2) 115(1)
Instr. 015: 3(1) 4(0) 5(1) 7(1) 8(0) 9(1) 10(0) 11(1) 12(0)
13(1) 14(0) 15(1) 16(1) 17(0) 18(0) 19(0) 20(0) 21(1) 23(1)
25(0) 26(1) 27(1) 29(0) 30(1) 31(0) 33(1) 34(0) 35(1) 36(0)
37(1) 38(1) 40(0) 41(1) 42(1) 44(0) 45(1) 47(1) 48(0) 49(1)
50(0) 52(1) 54(1) 56(0) 57(1) 58(0) 59(1) 61(1) 62(0) 63(1)
64(0) 66(1) 68(1) 70(0) 71(1) 76(0) 81(1) 82(0) 83(1) 85(1)
87(0) 88(1) 89(1) 91(0) 92(1) 95(0) 105(0) 108(1) 109(1)
111(0) 112(1)
Instr. 016: 3(1) 4(0) 5(1) 7(1) 8(0) 9(1) 10(0) 11(1) 12(0)
13(1) 14(0) 15(1) 16(1) 17(0) 18(0) 19(0) 20(0) 21(1) 23(1)
25(0) 26(1) 27(1) 29(0) 30(1) 31(0) 33(1) 34(0) 35(1) 36(0)
37(1) 38(1) 40(0) 41(1) 42(1) 44(0) 45(1) 47(1) 48(0) 49(1)
50(0) 52(1) 54(1) 56(0) 57(1) 58(0) 59(1) 61(1) 62(0) 63(1)
64(0) 66(1) 68(1) 70(0) 71(1) 76(0) 81(1) 82(0) 83(1) 85(1)
87(0) 88(1) 89(1) 91(0) 92(1) 95(0) 105(0) 108(1) 109(1)
111(0) 112(1)
Instr. 017: 2(2) 3(2) 5(2) 6(1) 7(0) 9(2) 11(2) 13(2) 15(2)
16(2) 17(1) 18(2) 19(1) 21(2) 22(1) 23(0) 24(1) 25(0) 26(2)
27(2) 28(1) 29(0) 30(2) 33(2) 35(2) 37(2) 38(2) 39(1) 40(0)
41(2) 42(2) 43(1) 44(0) 45(2) 46(1) 47(0) 49(2) 52(2) 53(1)
54(0) 55(1) 56(0) 57(2) 59(2) 60(1) 61(0) 63(2) 66(2) 67(1)
68(0) 69(1) 70(0) 71(2) 72(1) 75(1) 76(0) 81(2) 83(2) 84(1)
85(0) 86(1) 87(0) 88(2) 89(2) 90(1) 91(0) 92(2) 93(1) 94(0)
104(1) 105(0) 108(2) 109(2) 110(1) 111(0) 112(2) 114(1)
115(0) 118(1) 119(1) 121(0) 122(1) 124(1) 125(0) 127(1)
129(1) 130(0) 131(1) 132(1) 134(0)
Instr. 018: 2(1) 3(2) 5(2) 7(2) 9(2) 11(2) 13(2) 15(2)
16(2) 17(0) 18(1) 21(2) 23(2) 26(2) 27(2) 30(2) 33(2) 35(2)
37(2) 38(2) 41(2) 42(2) 45(2) 47(2) 49(2) 52(2) 54(2) 57(2)

59(2) 61(2) 63(2) 66(2) 68(2) 71(2) 81(2) 83(2) 85(2) 88(2)
89(2) 92(2) 108(2) 109(2) 112(2)
Instr. 019: 3(2) 5(2) 6(1) 7(0) 9(2) 11(2) 13(2) 15(2)
16(2) 17(1) 19(1) 21(2) 22(1) 23(0) 24(1) 25(0) 26(2) 27(2)
28(1) 29(0) 30(2) 33(2) 35(2) 37(2) 38(2) 39(1) 40(0) 41(2)
42(2) 43(1) 44(0) 45(2) 46(1) 47(0) 49(2) 52(2) 53(1) 54(0)
55(1) 56(0) 57(2) 59(2) 60(1) 61(0) 63(2) 66(2) 67(1) 68(0)
69(1) 70(0) 71(2) 72(1) 75(1) 76(0) 81(2) 83(2) 84(1) 85(0)
86(1) 87(0) 88(2) 89(2) 90(1) 91(0) 92(2) 93(1) 94(0)
104(1) 105(0) 108(2) 109(2) 110(1) 111(0) 112(2) 114(1)
115(0) 118(1) 119(1) 121(0) 122(1) 124(1) 125(0) 127(1)
129(1) 130(0) 131(1) 132(1) 134(0)
Instr. 020: 3(2) 5(2) 7(2) 9(2) 11(2) 13(2) 15(2) 16(2)
20(1) 21(2) 23(2) 26(2) 27(2) 30(2) 33(2) 35(2) 37(2) 38(2)
41(2) 42(2) 45(2) 47(2) 49(2) 52(2) 54(2) 57(2) 59(2) 61(2)
63(2) 66(2) 68(2) 71(0) 81(2) 83(2) 85(2) 88(2) 89(2) 92(0)
108(2) 109(2) 112(2)
Instr. 021: 3(1) 4(0) 5(1) 7(1) 8(0) 9(1) 10(0) 11(1) 12(0)
13(1) 14(0) 15(1) 16(1) 17(0) 18(0) 19(0) 20(0) 21(1) 23(1)
25(0) 26(1) 27(1) 29(0) 30(1) 31(0) 33(1) 34(0) 35(1) 36(0)
37(1) 38(1) 40(0) 41(1) 42(1) 44(0) 45(1) 47(1) 48(0) 49(1)
50(0) 52(1) 54(1) 56(0) 57(1) 58(0) 59(1) 61(1) 62(0) 63(1)
64(0) 66(1) 68(1) 70(0) 71(1) 76(0) 80(2) 81(1) 82(0) 83(1)
85(1) 87(0) 88(1) 89(1) 91(0) 92(1) 95(0) 105(0) 108(1)
109(1) 111(0) 112(1)
Instr. 022: 6(1) 7(0) 17(1) 19(1) 22(1) 23(0) 24(1) 25(0)
28(1) 29(0) 39(1) 40(0) 43(1) 44(0) 46(1) 47(0) 53(1) 54(0)
55(1) 56(0) 60(1) 61(0) 67(1) 68(0) 69(1) 70(0) 72(1) 75(1)
76(0) 84(1) 85(0) 86(1) 87(0) 90(1) 91(0) 93(1) 94(0)
104(1) 105(0) 110(1) 111(0) 114(1) 115(0) 118(1) 119(1)
121(0) 122(1) 124(1) 125(0) 127(1) 129(1) 130(0) 131(1)
132(1) 134(0)
Instr. 023: 3(1) 4(0) 5(1) 6(2) 7(1) 8(0) 9(1) 10(0) 11(1)
12(0) 13(1) 14(0) 15(1) 16(1) 17(0) 18(0) 19(0) 20(0) 21(1)
22(2) 23(1) 24(2) 25(0) 26(1) 27(1) 28(2) 29(0) 30(1) 31(0)
33(1) 34(0) 35(1) 36(0) 37(1) 38(1) 39(2) 40(0) 41(1) 42(1)
43(2) 44(0) 45(1) 46(2) 47(1) 48(0) 49(1) 50(0) 52(1) 53(2)
54(1) 55(2) 56(0) 57(1) 58(0) 59(1) 60(2) 61(1) 62(0) 63(1)
64(0) 66(1) 67(2) 68(1) 69(2) 70(0) 71(1) 72(2) 75(2) 76(0)
81(1) 82(0) 83(1) 84(2) 85(1) 86(2) 87(0) 88(1) 89(1) 90(2)
91(0) 92(1) 93(2) 95(0) 104(2) 105(0) 108(1) 109(1) 110(2)
111(0) 112(1) 114(2) 118(2) 119(2) 122(2) 124(2) 127(2)
129(2) 131(2) 132(2)
Instr. 024: 6(1) 7(0) 17(1) 19(1) 22(1) 23(0) 24(1) 25(0)
28(1) 29(0) 39(1) 40(0) 43(1) 44(0) 46(1) 47(0) 53(1) 54(0)
55(1) 56(0) 60(1) 61(0) 67(1) 68(0) 69(1) 70(0) 72(1) 75(1)
76(0) 84(1) 85(0) 86(1) 87(0) 90(1) 91(0) 93(1) 94(0)
104(1) 105(0) 110(1) 111(0) 114(1) 115(0) 118(1) 119(1)
121(0) 122(1) 124(1) 125(0) 127(1) 129(1) 130(0) 131(1)
132(1) 134(0)
Instr. 025: 3(2) 5(2) 6(2) 7(2) 9(2) 11(2) 13(2) 15(2)
16(2) 17(2) 19(2) 21(2) 22(2) 23(2) 24(2) 26(2) 27(2) 28(2)
30(2) 33(2) 35(2) 37(2) 38(2) 39(2) 41(2) 42(2) 43(2) 45(2)
46(2) 47(2) 49(2) 52(2) 53(2) 54(2) 55(2) 57(2) 59(2) 60(2)
61(2) 63(2) 66(2) 67(2) 68(2) 69(2) 71(2) 72(2) 75(2) 81(2)
83(2) 84(2) 85(2) 86(2) 88(2) 89(2) 90(2) 92(2) 93(2)
104(2) 108(2) 109(2) 110(2) 112(2) 114(2) 118(2) 119(2)
122(2) 124(2) 127(2) 129(2) 131(2) 132(2)
Instr. 026: 3(1) 4(0) 5(1) 7(1) 8(0) 9(1) 10(0) 11(1) 12(0)
13(1) 14(0) 15(1) 16(1) 17(0) 18(0) 19(0) 20(0) 21(1) 23(1)

```
25(0)  26(1)  27(1)  29(0)  30(1)  31(0)  33(1)  34(0)  35(1)  36(0)
37(1)  38(1)  40(0)  41(1)  42(1)  44(0)  45(1)  47(1)  48(0)  49(1)
50(0)  52(1)  54(1)  56(0)  57(1)  58(0)  59(1)  61(1)  62(0)  63(1)
64(0)  66(1)  68(1)  70(0)  71(1)  76(0)  81(1)  82(0)  83(1)  85(1)
87(0)  88(1)  89(1)  91(0)  92(1)  95(0)  105(0)  108(1)  109(1)
111(0)  112(1)
Instr. 027:  3(1)  4(0)  5(1)  7(1)  8(0)  9(1)  10(0)  11(1)  12(0)
13(1)  14(0)  15(1)  16(1)  17(0)  18(0)  19(0)  20(0)  21(1)  23(1)
25(0)  26(1)  27(1)  29(0)  30(1)  31(0)  33(1)  34(0)  35(1)  36(0)
37(1)  38(1)  40(0)  41(1)  42(1)  44(0)  45(1)  47(1)  48(0)  49(1)
50(0)  52(1)  54(1)  56(0)  57(1)  58(0)  59(1)  61(1)  62(0)  63(1)
64(0)  66(1)  68(1)  70(0)  71(1)  76(0)  81(1)  82(0)  83(1)  85(1)
87(0)  88(1)  89(1)  91(0)  92(1)  95(0)  105(0)  108(1)  109(1)
111(0)  112(1)
Instr. 028:  6(1)  7(0)  17(1)  19(1)  22(1)  23(0)  24(1)  25(0)
28(1)  29(0)  39(1)  40(0)  43(1)  44(0)  46(1)  47(0)  53(1)  54(0)
55(1)  56(0)  60(1)  61(0)  67(1)  68(0)  69(1)  70(0)  72(1)  75(1)
76(0)  84(1)  85(0)  86(1)  87(0)  90(1)  91(0)  93(1)  94(0)
104(1)  105(0)  110(1)  111(0)  114(1)  115(0)  118(1)  119(1)
121(0)  122(1)  124(1)  125(0)  127(1)  129(1)  130(0)  131(1)
132(1)  134(0)
Instr. 029:  3(2)  5(2)  6(2)  7(2)  9(2)  11(2)  13(2)  15(2)
16(2)  17(2)  19(2)  21(2)  22(2)  23(2)  24(2)  26(2)  27(2)  28(2)
30(2)  33(2)  35(2)  37(2)  38(2)  39(2)  41(2)  42(2)  43(2)  45(2)
46(2)  47(2)  49(2)  52(2)  53(2)  54(2)  55(2)  57(2)  59(2)  60(2)
61(2)  63(2)  66(2)  67(2)  68(2)  69(2)  71(2)  72(2)  75(2)  81(2)
83(2)  84(2)  85(2)  86(2)  88(2)  89(2)  90(2)  92(2)  93(2)
104(2)  108(2)  109(2)  110(2)  112(2)  114(2)  118(2)  119(2)
122(2)  124(2)  127(2)  129(2)  131(2)  132(2)
(...)
```

C.8 Código final

```
  0 - ADDI: A = 0, B = 23, C = 0 e IMM16 = 544
  1 - ADDI: A = 0, B = 22, C = 0 e IMM16 = 548
  2 - ADDI: A = 0, B = 21, C = 0 e IMM16 = 0
  3 - ADDI: A = 0, B = 12, C = 0 e IMM16 = -34
  4 - ADD: A = 0, B = 12, C = 2 e IMM16 = 0
  5 - ADD: A = 0, B = 2, C = 12 e IMM16 = 0
  6 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 75
  7 - ADD: A = 13, B = 12, C = 12 e IMM16 = 0
  8 - ADD: A = 0, B = 12, C = 3 e IMM16 = 0
  9 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 0
 10 - ADD: A = 0, B = 12, C = 4 e IMM16 = 0
 11 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 236
 12 - ADD: A = 0, B = 12, C = 5 e IMM16 = 0
 13 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 0
 14 - ADD: A = 0, B = 12, C = 9 e IMM16 = 0
 15 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 13
 16 - MULI: A = 12, B = 12, C = 0 e IMM16 = 4
 17 - ADD: A = 12, B = 21, C = 13 e IMM16 = 0
 18 - ADDI: A = 12, B = 21, C = 0 e IMM16 = 0
 19 - ADDI: A = 12, B = 13, C = 0 e IMM16 = 0
 20 - ADD: A = 0, B = 12, C = 10 e IMM16 = 0
 21 - ADD: A = 0, B = 5, C = 12 e IMM16 = 0
 22 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 230
```

```
23 - CMPLT: A = 12, B = 13, C = 12 e IMM16 = 0
24 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 1
25 - BEQ: A = 12, B = 13, C = 0 e IMM16 = 28
26 - ADD: A = 0, B = 4, C = 12 e IMM16 = 0
27 - CMPNE: A = 12, B = 0, C = 12 e IMM16 = 0
28 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 1
29 - BEQ: A = 12, B = 13, C = 0 e IMM16 = 12
30 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 88
31 - ADD: A = 0, B = 12, C = 7 e IMM16 = 0
32 - BR: A = 0, B = 0, C = 0 e IMM16 = 8
33 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 89
34 - ADD: A = 0, B = 12, C = 7 e IMM16 = 0
35 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 76
36 - ADD: A = 0, B = 12, C = 8 e IMM16 = 0
37 - ADD: A = 0, B = 2, C = 12 e IMM16 = 0
38 - CMPLT: A = 0, B = 12, C = 12 e IMM16 = 0
39 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 1
40 - BEQ: A = 12, B = 13, C = 0 e IMM16 = 16
41 - ADD: A = 0, B = 3, C = 12 e IMM16 = 0
42 - CMPGE: A = 12, B = 0, C = 12 e IMM16 = 0
43 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 1
44 - BEQ: A = 12, B = 13, C = 0 e IMM16 = 28
45 - ADD: A = 0, B = 2, C = 12 e IMM16 = 0
46 - ADDI: A = 0, B = 13, C = 0 e IMM16 = -1
47 - MUL: A = 13, B = 12, C = 12 e IMM16 = 0
48 - ADD: A = 0, B = 12, C = 2 e IMM16 = 0
49 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 87
50 - ADD: A = 0, B = 12, C = 8 e IMM16 = 0
51 - BR: A = 0, B = 0, C = 0 e IMM16 = 44
52 - ADD: A = 0, B = 7, C = 12 e IMM16 = 0
53 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 88
54 - CMPEQ: A = 12, B = 13, C = 12 e IMM16 = 0
55 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 1
56 - BEQ: A = 12, B = 13, C = 0 e IMM16 = 8
57 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 73
58 - ADD: A = 0, B = 12, C = 8 e IMM16 = 0
59 - ADD: A = 0, B = 2, C = 12 e IMM16 = 0
60 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 2
61 - DIV: A = 12, B = 13, C = 12 e IMM16 = 0
62 - ADD: A = 0, B = 12, C = 2 e IMM16 = 0
63 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 13
64 - ADD: A = 0, B = 12, C = 6 e IMM16 = 0
65 - ADDI: A = 6, B = 6, C = 0 e IMM16 = -1
66 - ADD: A = 0, B = 6, C = 12 e IMM16 = 0
67 - ADDI: A = 0, B = 13, C = 0 e IMM16 = -1
68 - CMPGE: A = 13, B = 12, C = 12 e IMM16 = 0
69 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 1
70 - BEQ: A = 12, B = 13, C = 0 e IMM16 = 36
71 - ADD: A = 0, B = 10, C = 12 e IMM16 = 0
72 - ADD: A = 0, B = 6, C = 13 e IMM16 = 0
73 - ADD: A = 0, B = 6, C = 14 e IMM16 = 0
74 - MULI: A = 14, B = 14, C = 0 e IMM16 = -1
75 - MULI: A = 13, B = 13, C = 0 e IMM16 = 4
76 - ADD: A = 13, B = 12, C = 15 e IMM16 = 0
77 - ADD: A = 15, B = 22, C = 15 e IMM16 = 0
78 - STW: A = 15, B = 14, C = 0 e IMM16 = 0
79 - BR: A = 0, B = 0, C = 0 e IMM16 = -60
80 - ADDI: A = 5, B = 5, C = 0 e IMM16 = 1
81 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 0
82 - ADD: A = 0, B = 12, C = 11 e IMM16 = 0
```

```
 83 - ADD: A = 0, B = 11, C = 12 e IMM16 = 0
 84 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 13
 85 - CMPGE: A = 12, B = 13, C = 12 e IMM16 = 0
 86 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 1
 87 - BEQ: A = 12, B = 13, C = 0 e IMM16 = 120
 88 - ADD: A = 0, B = 9, C = 12 e IMM16 = 0
 89 - CMPEQ: A = 12, B = 0, C = 12 e IMM16 = 0
 90 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 1
 91 - BEQ: A = 12, B = 13, C = 0 e IMM16 = 64
 92 - ADD: A = 0, B = 10, C = 12 e IMM16 = 0
 93 - ADD: A = 0, B = 11, C = 13 e IMM16 = 0
 94 - ADD: A = 13, B = 0, C = 15 e IMM16 = 0
 95 - ADD: A = 12, B = 0, C = 14 e IMM16 = 0
 96 - MULI: A = 15, B = 15, C = 0 e IMM16 = 4
 97 - ADD: A = 14, B = 15, C = 15 e IMM16 = 0
 98 - ADD: A = 15, B = 22, C = 15 e IMM16 = 0
 99 - LDW: A = 15, B = 14, C = 0 e IMM16 = 0
100 - ADD: A = 0, B = 5, C = 15 e IMM16 = 0
101 - ADDI: A = 0, B = 16, C = 0 e IMM16 = 5
102 - DIV: A = 15, B = 16, C = 15 e IMM16 = 0
103 - ADD: A = 15, B = 14, C = 14 e IMM16 = 0
104 - MULI: A = 13, B = 13, C = 0 e IMM16 = 4
105 - ADD: A = 13, B = 12, C = 15 e IMM16 = 0
106 - ADD: A = 15, B = 22, C = 15 e IMM16 = 0
107 - STW: A = 15, B = 14, C = 0 e IMM16 = 0
108 - ADD: A = 0, B = 9, C = 12 e IMM16 = 0
109 - CMPNE: A = 12, B = 0, C = 12 e IMM16 = 0
110 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 1
111 - BEQ: A = 12, B = 13, C = 0 e IMM16 = 8
112 - ADDI: A = 0, B = 12, C = 0 e IMM16 = 1
113 - BR: A = 0, B = 0, C = 0 e IMM16 = 4
114 - ADDI: A = 0, B = 13, C = 0 e IMM16 = 0
115 - ADD: A = 0, B = 13, C = 9 e IMM16 = 0
116 - ADDI: A = 11, B = 11, C = 0 e IMM16 = 1
117 - BR: A = 0, B = 0, C = 0 e IMM16 = -140
118 - ADD: A = 0, B = 2, C = 13 e IMM16 = 0
119 - CMPGE: A = 0, B = 13, C = 13 e IMM16 = 0
120 - ADDI: A = 0, B = 14, C = 0 e IMM16 = 1
121 - BEQ: A = 13, B = 14, C = 0 e IMM16 = 20
122 - ADD: A = 0, B = 2, C = 13 e IMM16 = 0
123 - ADDI: A = 0, B = 14, C = 0 e IMM16 = 1
124 - SUB: A = 13, B = 14, C = 13 e IMM16 = 0
125 - ADD: A = 0, B = 13, C = 2 e IMM16 = 0
126 - BR: A = 0, B = 0, C = 0 e IMM16 = 16
127 - ADD: A = 0, B = 2, C = 13 e IMM16 = 0
128 - ADDI: A = 0, B = 14, C = 0 e IMM16 = 1
129 - ADD: A = 14, B = 13, C = 13 e IMM16 = 0
130 - ADD: A = 0, B = 13, C = 2 e IMM16 = 0
131 - ADD: A = 0, B = 2, C = 13 e IMM16 = 0
132 - CMPNE: A = 13, B = 0, C = 13 e IMM16 = 0
133 - ADDI: A = 0, B = 14, C = 0 e IMM16 = 1
134 - BEQ: A = 13, B = 14, C = 0 e IMM16 = -220
135 - BR: A = 0, B = 0, C = 0 e IMM16 = 4
```

Apêndice D – *Bytecodes* suportados

Os *bytecodes* atualmente suportados pelo compilador JaNi estão relacionados na Tabela D.1.

Tabela D.1 Relação dos *bytecodes* atualmente suportados pelo JaNi.

NOP	ACONST_NULL	ICONST_M1	ICONST_0
ICONST_1	ICONST_2	ICONST_3	ICONST_4
ICONST_5	BIPUSH	SIPUSH	ILOAD
ILOAD_0	ILOAD_1	ILOAD_2	ILOAD_3
ALOAD	IALOAD	ISTORE	ISTORE_0
ISTORE_1	ISTORE_2	ISTORE_3	ASTORE
IASTORE	POP	POP2	DUP
DUP2	SWAP	IADD	ISUB
IMUL	IDIV	IREM	INEG
ISHL	ISHR	IAND	IOR
IXOR	IINC	INT2SHORT	IFEQ
IFNE	IFLT	IFGE	IFGT
IFLE	IF_ICMPEQ	IF_ICMPNE	IF_ICMPLT
IF_ICMPGE	IF_ICMPGT	IF_ICMPLE	GOTO
RETURN	NEWARRAY		

Apêndice E – Instruções do Nios II geradas

São relacionadas na Tabela E.1 todas as instruções do processador Nios II que o JaNi pode gerar em seu atual estado de implementação. Para essa listagem, não são consideradas as pseudo-instruções.

Tabela E.1 Instruções do Nios II geradas pelo JaNi.

add	addi	sub	mul
muli	div	ldw	stw
rol	ror	and	or
xor	cmpeq	cmpne	cmplt
cmpge	br		

Livros Grátis

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

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas