

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



PHOENIX – Um *Framework* para
Trabalhos em Síntese de Alto Nível de Circuitos Digitais

Flávio Luis Duarte

Fevereiro
2006

Flávio Luis Duarte

PHOENIX – Um *Framework* para
Trabalhos em Síntese de Alto Nível de Circuitos Digitais

Faculdade de Computação
Universidade Federal de Uberlândia
2006

Flávio Luis Duarte

PHOENIX – Um *Framework* para
Trabalhos em Síntese de Alto Nível de Circuitos Digitais

Dissertação submetida à Universidade Federal de Uberlândia - Minas Gerais, em atendimento parcial dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA

Elaborada pelo Sistema de Bibliotecas da UFU / Setor de Catalogação e Classificação

D812p	Duarte, Flávio Luis, 1978 – <i>Phoenix – Um Framework para Trabalhos em Síntese de Alto Nível de Circuitos Digitais</i> / Flávio Luis Duarte. – Uberlândia, 2006. 136f. : il. Orientador: Sérgio de Mello Schneider. Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação. Inclui bibliografia. 1. Engenharia de Software – Teses. 2. Compiladores (Programas de Computador) – Teses. I. Schneider, Sérgio de Mello. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título. CDU: 681.3.06
-------	--

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada “**PHOENIX – Um Framework para Trabalhos em Síntese de Alto Nível de Circuitos Digitais**” por **Flávio Luis Duarte** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 17 de Fevereiro de 2006.

Orientador:

Prof. Dr. Sérgio de Mello Schneider
Universidade Federal de Uberlândia UFU/MG

Banca Examinadora:

Prof. Dr. Eduardo Marques
Universidade de São Paulo USP/SC

Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia UFU/MG

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Data: Fevereiro de 2006

Autor: **Flávio Luis Duarte**
Título: **PHOENIX – Um *Framework* para Trabalhos em Síntese de Alto Nível de Circuitos Digitais**
Faculdade: **Faculdade de Computação**
Grau: **Mestre** Convocação: **Fevereiro** Ano: **2006**

A Universidade Federal de Uberlândia possui permissão para distribuir e ter cópias deste documento para propósitos exclusivamente acadêmicos, desde que a autoria seja devidamente divulgada.

Autor

O AUTOR RESERVA OS DIREITOS DE PUBLICAÇÃO DESTE DOCUMENTO. NÃO PODE SER IMPRESSO OU REPRODUZIDO DE OUTRA FORMA, SEJA NA TOTALIDADE OU EM PARTES SEM A PERMISSÃO ESCRITA DO AUTOR.

Aos meus pais Sebastião e Maria pela eterna
dedicação que tiveram a mim para que tivesse
um futuro melhor.

Agradecimentos

Aos meus pais Sebastião e Maria pelo eterno apoio e estímulo ao meu desenvolvimento.

Ao meu orientador Sérgio de Mello Schneider pelo apoio, atenção, responsabilidade e paciência gastos no desenvolvimento deste trabalho.

Aos meus colegas da pós-graduação pelo coleguismo e pelo apoio em todas as etapas do curso.

A todos que direta ou indiretamente contribuíram com este trabalho.

A Deus, por me permitir saúde e vontade para conseguir mais esta realização.

Sumário

1	Introdução.....	1
1.1	Organização desta dissertação.....	5
2	Computação Reconfigurável	6
2.1	Introdução	6
2.2	Modelos de Computação	6
2.3	Arquitetura de um FPGA.....	9
2.4	Suporte aos Sistemas Reconfiguráveis	14
2.5	<i>Software</i> para Sistemas Reconfiguráveis.....	15
2.5.1	Modelo de Co-Projeto <i>Hardware/Software</i>	17
2.6	Conclusão	20
3	Compilação para <i>Reconfigware</i>	21
3.1	Introdução	21
3.2	Processo de Compilação para <i>Reconfigware</i>	21
3.2.1	Geração da Representação Intermediária.....	23
3.2.2	Otimização dos Grafos.....	25
3.2.3	Mapeamento de Componentes <i>Hardware</i>	25
3.2.4	Escalonamento.....	26
3.2.5	Geração do Circuito	26
3.3	Compiladores de <i>Reconfigware</i>	28
3.3.1	<i>Garp</i>	26
3.3.2	<i>Galadriel</i> e <i>Nenia</i>	30
3.3.3	<i>Spark</i>	41
3.3.4	<i>Impulse C</i>	43
3.4	Conclusão	45
4	O <i>Framework Phoenix</i>	47
4.1	Introdução	47
4.2	ARCHITECT+	48
4.3	Nios II	50
4.3.1	Conjunto de Instruções.....	52
4.4	<i>Phoenix</i>	54
4.5	Componentes	57

4.5.1	Tabela de Símbolos.....	57
4.5.2	Árvores Sintáticas	60
4.5.3	Instruções de Três Endereços	61
4.5.4	Representação de Tipos da Linguagem C	62
4.6	Processos do Compilador <i>Phoenix</i>	65
4.6.1	Análise Léxica e Sintática	65
4.6.2	Análise Semântica.....	67
4.6.3	Geração da Representação Intermediária	72
4.6.3.1	Grafo de Fluxo de Controle.....	71
4.6.3.2	Grafo de Hierarquia de Tarefas	73
4.6.3.3	Grafo de Fluxo de Dados	75
4.6.3.4	Grafo de Dependência de Dados	79
4.6.3.5	Grafo de Dependência de Controle.....	81
4.6.4	Decomposição de <i>Structs</i> e <i>Unions</i> ao Nível Primitivo ou <i>Array</i>	81
4.6.5	Interface de Retaguarda.....	87
4.6.5.1	Compilação para Múltiplos Alvos	87
4.6.5.2	Geração de Código no <i>Phoenix</i>	90
4.6.5.3	Simulador para o Nios II.....	103
4.7	Conclusões	104
5	Resultados.....	106
5.1	Exemplo 1	108
5.2	Exemplo 2	112
6	Conclusão	115
6.1	Trabalhos Futuros	116
	Referências Bibliográficas	118
	Bibliografia.....	122
A1	- Diagrama de Classes da Tabela de Símbolos do <i>Phoenix</i>	125
A2	- Diagrama de Classes das Árvores Sintáticas do <i>Phoenix</i>	126
A3	- Diagrama de Classes da Representação Intermediária do <i>Phoenix</i>	127
A4	- Diagrama de Classes da Representação Intermediária do <i>Phoenix</i> (continuação)	128
A5	- Diagrama de Classes do <i>Phoenix</i>	129
A6	- Descrição da Arquitetura do Nios II	130
A7	- Especificação ANSI C.....	136
A8	- Gramática ANSI C para o YACC.....	136

A9 - Analisador Léxico ANSI C para o Lex	136
--	-----

Lista de Figuras

Figura 2.1 – Célula genérica da maioria dos FPGAs.....	10
Figura 2.2 – Localização das entradas e saída de uma célula.	11
Figura 2.3 – Ligação do pino de saída aos segmentos do canal de roteamento.	11
Figura 2.4 – Estrutura genérica de um FPGA.	12
Figura 2.5 – Exemplo de arquitetura multi-FPGA	13
Figura 2.6 – Fases dos diversos meios de criação de sistemas reconfiguráveis.....	16
Figura 3.1 – Processo de compilação tradicional.	22
Figura 3.2 – Grafos de fluxo de controle, dependência de controle, fluxo de dados e dependência de dados para um programa exemplo.	24
Figura 3.3 – Fluxograma genérico da síntese de alto nível de circuitos digitais.....	27
Figura 3.4 – Exemplo de grafo de pós-dominâncias (b) e CDG (c) para um programa (a).....	32
Figura 3.5 – Exemplo de pontos de seleção em um programa.	34
Figura 3.6 – Exemplo de pontos de seleção com multiplexadores.....	35
Figura 3.7 – Exemplo de um DFG global	37
Figura 3.8 – Fluxograma do <i>Spark</i>	41
Figura 3.9 – Modelo de programação do <i>Impulse C</i>	45
Figura 4.1 – Diagrama de blocos do ARCHITECT+.	49
Figura 4.2 – Organização do SOPC gerado pelo ARCHITECT+ no FPGA.....	50
Figura 4.3 – Exemplo de um sistema do processador Nios II.....	51
Figura 4.4 – Formatos de instruções do Nios II.	53
Figura 4.5 – Formato da instrução customizada do Nios II.	54
Figura 4.6 – Fluxograma do <i>Phoenix</i>	55
Figura 4.7 – Função de <i>hash</i> utilizada no <i>Phoenix</i>	59
Figura 4.8 – Exemplo de construção de instruções de três endereços a partir da árvore de derivação sintática da expressão $A = B * C + 10 / D + -E$	62
Figura 4.9 – Exemplo de uma declaração de uma variável em linguagem C.	62
Figura 4.10 – Exemplo da lista que representa a declaração da figura 4.9.	64
Figura 4.11 – Função de verificação de tipos.....	69
Figura 4.12 – Variáveis e tipos associados à expressão $A = B * C + D.F + -E$	70
Figura 4.13 – Árvore sintática da expressão $A = B * C + D.F + -E$	70
Figura 4.14 – Equações de fluxo de dados utilizadas no cômputo da cadeias <i>ud</i> e <i>du</i>	76

Figura 4.15 – Equação utilizada no cômputo do conjunto <i>Mortas</i>	77
Figura 4.16 – Equação da figura 4.15 definida em termos de operações <i>bitwise</i>	77
Figura 4.17 – Exemplo de cadeias <i>ud</i> e <i>du</i> de um programa.....	79
Figura 4.18 – Árvore sintática para a expressão $S1.S2.A = 0$	85
Figura 4.19 – Árvore sintática para a expressão $S3.S2 \rightarrow S1.A = 0$	86
Figura 4.20 – Modelo do <i>Phoenix</i> para a geração de código para múltiplos alvos.	91
Figura 4.21 – Formato geral para descrição das instruções.	95
Figura 4.22 – Exemplo de parte da especificação do Pentium TM	98
Figura 4.23 – Declarações das variáveis da expressão $S.A[B+C].D = S.E$	99
Figura 4.24 – Instruções de três geradas para a expressão $S.A[B+C].D = S.E$	99
Figura 4.25 – Instruções de três geradas para a expressão $S.A[B+C].D = S.E$ com o processo de decomposição de <i>structs</i> ativado.....	102
Figura 5.1 – Programa exemplo para a geração do CFG e HTG.....	108
Figura 5.2 – Grafo CFG do programa da figura 5.1.	109
Figura 5.3 – Grafo HTG do programa da figura 5.1.....	110
Figura 5.4 – Código gerado para o programa da figura 5.1	111
Figura 5.5 – Programa exemplo para a geração do DDG.	112
Figura 5.6 – Grafo DDG do programa da figura 5.4.	112
Figura 5.7 – Grafo CFG com as definições incidentes do programa da figura 5.4.	113
Figura 5.8 – Código gerado para o programa da figura 5.5.	114

Lista de Tabelas

Tabela 1.1 – Contribuição do <i>Phoenix</i> para a síntese de alto nível de circuitos digitais	3
Tabela 4.1 – Alguns compiladores de síntese de alto nível de circuitos digitais e seus suportes à linguagem C	82

Lista de Acrônimos

Acrônimo em Inglês	Tradução Utilizada Nesta Dissertação
ALAP – <i>As Late As Possible</i>	-
ALU – <i>Arithmetic Logic Unit</i>	Unidade Lógico Aritmética
ANSI – <i>American National Standard Institute</i>	-
ASAP – <i>As Soon As Possible</i>	-
ASIC – <i>Application Specific Integrated Circuit</i>	Circuito Integrado de Aplicação Específica
CDFG – <i>Control-Data Flow Graph</i>	Grafo de Fluxo de Controle-Dados
CDG – <i>Control Dependence Graph</i>	Grafo de Dependência de Controle
CFG – <i>Control Flow Graph</i>	Grafo de Fluxo de Controle
CISC – <i>Complex Instruction Set Computer</i>	Computador de Conjunto de Instruções Complexo
CPU – <i>Central Process Unit</i>	Unidade de Processamento Central
DDG – <i>Data Dependence Graph</i>	Grafo de Dependência de Dados
DFG – <i>Data Flow Graph</i>	Grafo de Fluxo de Dados
DSP – <i>Digital Signal Processor</i>	Processador de Sinais Digitais
FCCM– <i>Field-Custom Computing Machine</i>	Máquina de Computação Personalizada no Campo
FPGA – <i>Field Programmable Gate Array</i>	Arranjo de portas lógicas programável
GCC – <i>Gnu C Compiler</i>	-
HDL – <i>Hardware Description Language</i>	Linguagem de Descrição de <i>Hardware</i>
HPDG – <i>Hierarchical Program Dependence Graph</i>	Grafo Hierárquico de Dependências do Programa
HTG – <i>Hierarchical Task Graph</i>	Grafo Hierárquico de Tarefas

JVM – <i>Java Virtual Machine</i>	Máquina Virtual Java.
LALR – <i>Lookahead LR</i>	-
LUT – <i>Look Up Table</i>	Tabela de Consulta
MDG – <i>Merge Dependence Graph</i>	Grafo de Dependências de Fusão
PCC2 – <i>Portable C Compiler 2</i>	-
PAL – <i>Programmable Array Logic</i>	Lógica de Cadeia Programável
PLA – <i>Programmable Logic Array</i>	Cadeia de Lógica Programável
RAM – <i>Random Access Memory</i>	Memória de Acesso Aleatório
RISC – <i>Reduced Instruction Set Computers</i>	Computadores de Conjunto de Instruções Reduzido
RPU – <i>Reconfigurable Processing Unit</i>	-
RTL – <i>Register Transfer Level</i>	Nível de Transferência de Registros
SSA– <i>Static Single Assignment</i>	-
SOPC – <i>System On a Programable Chip</i>	Sistema em um <i>Chip</i> Programável
VHDL – <i>VHSIC (Very High Speed Integrated Circuits) Hardware Description Language</i>	-
VLIW – <i>Very Long Instruction Word</i>	-
YACC – <i>Yet Another Compiler Compiler</i>	-

Resumo

Este trabalho descreve o desenvolvimento de um *framework* de código aberto para síntese de circuitos digitais, para uso em projetos de *hardware/software* co-design. O *framework* consiste de um compilador que aceita ANSI C como código fonte e que permite a construção de um dado sistema e sua execução em *hardware*. Este compilador, intitulado *Phoenix*, permite grande flexibilidade de uso e fácil expansão de suas funcionalidades.

O compilador, além de seus módulos usuais (analisadores léxico e sintático, gerador de código) constrói grafos que descrevem dependências de dados, controle de dados, fluxo e dependências de hierarquia, informações essenciais para a geração e exploração de execução de código em paralelo. O compilador gera código para o processador virtual Nios IITM da Altera Corporation através de geração de código para múltiplos alvos. Estruturas de dados construídas com o construtor *struct* da linguagem C são automaticamente suportadas na síntese de alto nível de circuitos digitais.

Palavras chave: Engenharia de Software, compiladores, FPGAs, síntese de alto nível de circuitos digitais.

Abstract

This work describes the development of an open framework for the synthesis of digital circuits, for use in hardware/software co-design projects. The framework consists of a compiler which accepts ANSI C as source code allowing the construction of a given system and its execution in hardware. This compiler, named Phoenix, allows great flexibility of use and easy expansion of its functionality.

The compiler, besides its usual modules (scanning, parsing, code generation) builds graphs that describe data dependency, data control, flow and hierarchical dependency, essential information in order to generate and to explore parallel execution of code. The compiler generates code for execution in Nios IITM Altera Corporation “virtual” processor through retargetable code generation. Data structures built with the C language *struct* constructor are automatic supported for high-level synthesis of digital circuits.

Key-words: Software Engineering, compilers, FPGAs, high level synthesis of digital circuits.

Capítulo 1

Introdução

Desde o surgimento dos computadores, o código executável resultante da compilação de aplicações é dependente da arquitetura do modelo de computação de propósito geral alvo. Neste caso, o código executável constitui-se de uma sequência de instruções no formato exigido pela arquitetura. Esta dependência em relação ao conjunto de instruções acarreta uma falta de flexibilidade quando a migração da aplicação para outras arquiteturas se faz necessária. Tal processo de migração envolve em muitos casos, modificação do código e recompilação. Por sua vez, uma mudança em plataformas de *hardware* mostra-se ainda mais complexa, pois envolve um tempo de projeto bastante elevado bem como investimentos consideráveis para a modificação do projeto industrial do novo circuito.

Este panorama vem mudando graças à criação dos primeiros dispositivos reconfiguráveis. Estes dispositivos permitem uma flexibilidade maior, pois permitem alteração do modelo de *hardware*. A comercialização de tais dispositivos contribuiu ainda mais para o desenvolvimento deste novo paradigma. A partir desta renovação ficou mais fácil alterar os modelos de sistemas.

Aplicações mistas de *hardware* e *software* podem trabalhar em conjunto em um único dispositivo. Nestas aplicações, parte do código é executada sob a forma de *software* em processadores tradicionais e parte é executada sob a forma de *hardware* em dispositivos reconfiguráveis.

Compiladores foram construídos com o objetivo de melhorar o suporte e acelerar o processo de desenvolvimento de tais aplicações, permitindo gerar tanto o *software* como o *hardware* do sistema. Estes compiladores são ferramentas poderosas para o desenvolvimento

deste tipo de sistemas, pois fazem praticamente todo o trabalho de mapeamento da descrição do circuito para o dispositivo, abstraindo todo o conhecimento sobre o mesmo que seria exigido para a criação do sistema. Esta atividade é conhecida na comunidade científica e industrial como *síntese de alto nível de circuitos digitais*.

Dentro deste contexto, o Laboratório de Computação Reconfigurável do Instituto de Ciências Matemáticas e de Computação da USP - Campus São Carlos, vem desenvolvendo uma ferramenta para a geração de sistemas mistos *hardware/software* para dispositivos reconfiguráveis com vistas em geração de sistemas para robótica móvel. Esta ferramenta é denominada ARCHITECT+.

O objetivo do trabalho aqui apresentado é fornecer para o projeto ARCHITECT+ um *framework* de arquitetura aberta (com acesso ao código fonte e documentação) que permita o suporte à criação destes sistemas a partir de descrições feitas em linguagem ANSI C. Este *framework*, denominado *Phoenix*, proverá ao ARCHITECT+ suporte para o processo de síntese de alto nível de circuitos digitais, provendo várias das estruturas comumente utilizadas em tal processo. Adicionalmente, o *framework* terá também implementada a geração de código nativo para uma arquitetura RISC. O desenvolvimento do *framework* visa também contribuir positivamente com um problema comum entre compiladores de síntese de alto nível de circuitos digitais já existentes. Tal problema é a falta de suporte total aos recursos da linguagem para o processo de síntese de alto nível. Para este processo, alguns recursos da linguagem precisam de tratamento especial como, por exemplo, o uso de ponteiros, de recursividade e de estruturas de dados. O *framework* proverá um suporte automático para as estruturas de dados criadas com o construtor *struct* da linguagem C de modo que na futura implementação do processo de síntese de alto nível estes tipos de dados não necessitarão de nenhum tratamento especial para a geração do circuito.

A tabela 1.1 ilustra a contribuição do *Phoenix* quanto aos outros compiladores de

síntese de alto nível existentes. A figura mostra alguns dos compiladores mais relevantes existentes para síntese de alto nível e seus respectivos suportes quanto aos recursos da linguagem C. Os itens assinalados com asterisco (*) na coluna do *Phoenix* indicam que a implementação do suporte dos correspondentes recursos da linguagem C para a síntese de alto nível de circuitos digitais está reservada para trabalhos futuros.

Compilador	PRISM II [Athanas 1992]	Transmo- grifierC [Galloway 1995]	Chichkov [Chichkov 1998]	Garpcc [Callahan <i>et al</i> 2000]	SPARK [Gupta <i>et al</i> 2003]	<i>Phoenix</i>
1ª Publicação	1993	1995	1998	2000	2003	2006
<i>if-then-else</i>	V	V	V	V	V	*
Ciclos	V	<i>while</i>	V	V	V	*
<i>Arrays</i>	-	-	-	V	V	*
Estruturas	-	-	-	V	-	V
Ponteiros	-	-	-	V	-	*
Chamada de Funções	-	-	-	V	V	*
Recursividade	-	-	-	-	-	*
Representação Intermediária	DFG+CFG	AST	CDG+ DDG+ AST	DFG+ Hiperbloco	HTG+ DDG+ CDFG	HTG+ CFG+CDG+ DFG+DDG
Saída	VHDL	<i>Bitstreams</i>	VHDL	<i>Bitstreams</i>	VHDL	*

Tabela 1.1: Contribuição do *Phoenix* para a síntese de alto nível de circuitos digitais

Embora o compilador *Garpcc* suporte tais tipos de estruturas, até onde sabemos, o compilador não implementa o processo de suporte implementado no *Phoenix*.

As motivações para a construção de um *framework* de compilação a partir de seu início, não se utilizando de código de terceiros, são:

1. No que tange compiladores de síntese de alto nível de circuitos digitais comerciais, estes compiladores possuem arquitetura fechada, não permitindo que as estruturas internas criadas pelos mesmos sejam acessadas para utilização dentro do ARCHITECT+;
2. No que tange ferramentas de compilação tradicionais *open-source*, estes possuem propósitos específicos, possuindo representação intermediária voltada especificamente para o seu objetivo. Além disto, não podem ser facilmente adaptados e, visto que são de acesso e utilização gratuitos, não existe um compromisso de suporte. O suporte muitas vezes se dá através de listas de discussão. Um exemplo deste caso é o GCC (*GNU Compiler Collection*) [Stallman 2001]. O GCC hoje é constituído de aproximadamente 8000 arquivos e de 43 mega *bytes* de código fonte. Sua representação intermediária consiste em uma representação ao nível de transferência de registros (RTL) criada a partir da especificação da arquitetura da máquina alvo em uma linguagem no mesmo nível. Desta forma, a representação intermediária é específica ao alvo e não pode ser utilizada para outra máquina;
3. No que tange as ferramentas de compilação tradicionais comerciais, estes possuem preços elevados.

Tendo em vista estas dificuldades, surgiu a necessidade de se construir um *framework* de compilação a partir de seu início, para sua utilização no projeto ARCHITECT+.

Este projeto mostra-se relevante de um ponto de vista acadêmico porque facilita a criação de complexos sistemas de robótica. A construção de tais sistemas envolve a implementação de algoritmos de Inteligência Artificial, manipulação de vários tipos de dispositivos e geração de *hardware* específico. A contribuição deste trabalho está em permitir

a automatização da geração de sistemas mistos *hardware/software* a fim de facilitar o processo de desenvolvimento de sistemas robóticos. A relevância, do ponto de vista econômico, vem da busca de sistemas de *hardware* que possam ter desempenho superior ao dos dispositivos comercializados atualmente pela exploração do paralelismo em circuitos como os FPGAs.

1.1 Organização desta dissertação

O capítulo 2 apresenta uma revisão de literatura sobre a computação reconfigurável. São descritos os modelos de computação existentes, a arquitetura do dispositivo reconfigurável mais utilizado atualmente e os tipos de suporte para os sistemas reconfiguráveis. É apresentado um resumo sobre o processo de compilação para sistemas reconfiguráveis. É apresentado também o modelo de Co-Projeto *hardware/software*.

O capítulo 3 apresenta a compilação para sistemas reconfiguráveis. São descritos o processo geral de compilação e suas fases, e são apresentados quatro dos trabalhos mais recentes nesta área.

O capítulo 4 faz uma breve descrição do projeto ARCHITEC+ e do processador Nios II. Apresenta a arquitetura do *framework Phoenix*, sendo descritos seus componentes e processos mais importantes, dando-se ênfase aos aspectos dos mesmos considerados mais relevantes.

No capítulo 5 é feita a conclusão geral do trabalho sendo apresentadas as contribuições e possíveis trabalhos futuros.

Capítulo 2

Computação Reconfigurável

2.1 Introdução

Este capítulo apresenta a computação reconfigurável, um paradigma de computação que surgiu a partir da década de 80, com o objetivo de revolucionar o modo como são feitos sistemas computacionais. Inicialmente são apresentados os modelos de computação de algoritmo e um resumo da história do desenvolvimento da computação reconfigurável. É apresentado ainda o dispositivo de lógica reconfigurável mais utilizado no paradigma e sua arquitetura: o FPGA. Uma apresentação sobre arquiteturas de suporte para computação reconfigurável e *softwares* para sistemas reconfiguráveis é feita com base nos trabalhos de Compton e Hauck [Compton e Hauck 2000] e Cardoso [Cardoso 2000]. É descrito ainda o modelo de criação de sistemas mistos *hardware/software*, utilizado por Cardoso e também no projeto *Spark* [Gupta et al. 2003].

2.2 Modelos de Computação

Os sistemas computacionais mais utilizados na atualidade consistem em microprocessadores tradicionais de execução de *software* acoplados a uma ou mais unidades de memória e dispositivos de entrada e saída. Tal modelo de execução de algoritmos, embora satisfaça os quesitos de desempenho necessários às aplicações (*software*) atuais e embora seja mais flexível que os circuitos integrados de aplicação específica (ASICs), sofre de inflexibilidade quanto à arquitetura do microprocessador, pois as aplicações compiladas são dependentes do conjunto de instruções por ela provido. Para sua execução em outra

arquitetura, uma aplicação poderá necessitar de várias adaptações, além do processo de compilação específico, para que a mesma possa ser executada¹. Além disto, os passos executados pelo microprocessador para a execução das instruções do *software* como, leitura das instruções em memória e decodificação da instrução, representam uma sobrecarga considerável no tempo de computação.

O modelo de execução de algoritmos através de ASICs, apesar de ser mais veloz e eficiente em relação ao tempo de computação que o modelo baseado em microprocessadores de propósito geral, sofre da falta de flexibilidade na modificação do circuito após a fabricação, sendo para isto necessário um novo projeto do circuito e a fabricação do novo *chip*.

A computação reconfigurável permite maior flexibilidade que os dois modelos anteriores. Provê velocidade de computação potencialmente maior que os processadores de propósito geral. Permite ainda a alteração do modelo de *hardware* programado no dispositivo. Tal característica permite maior adaptabilidade ao processo de computação, pois permite a adaptação do *hardware* à aplicação, bem como a execução de partes específicas da aplicação como *hardware* e não como *software*. A computação reconfigurável surgiu com o objetivo de preencher esta lacuna entre *hardware* e *software*, permitindo maior performance enquanto oferece flexibilidade maior da configuração programada no *hardware*.

O conceito de adaptabilidade de uma arquitetura à aplicação vem do início da década de 60, decorrente de pesquisas feitas por Estrin [Estrin *et al.* 1963]. No final da década de 50, Estrin desenvolveu uma máquina computacional que podia ser reconfigurada. Especificamente, esta máquina era composta de três partes:

¹ Recentemente surgiram várias alternativas, baseadas em *software*, para tentar resolver tal problema. Uma delas é a execução de aplicações em cima de Máquinas Virtuais (VMs), como por exemplo o Java, que permitem uma independência relevante da linguagem e da aplicação em relação à arquitetura.

- Um computador de propósito geral, originalmente um IBM 7090, que era a parte fixa do modelo.
- Uma parte variável que consistia de várias subestruturas digitais que podiam ser reorganizadas em uma configuração de acordo com o propósito específico.
- Um controle supervisor que coordenava o chaveamento entre o módulo fixo e o módulo variável.

A parte variável da máquina de Estrin era formada por blocos que podiam ser inseridos em qualquer uma das 36 posições da placa mãe do sistema. A conexão entre estes blocos era feita através de fios, que conectavam os blocos por ação manual de operadores do sistema. A reconfiguração da função do sistema consistia em mudar alguns blocos e a reconfiguração do roteamento dos blocos consistia em mudar alguns fios. Estrin gastou a maior parte de seu esforço no processo de reconfiguração manual do sistema.

Pesquisas feitas por Franz J. Ramming com o propósito de reconfiguração sem interferência mecânica ou manual culminaram na construção da Máquina de Ramming em 1977. A máquina consistia em um *array* de seletores que permitiam a programação do roteamento através de informações guardadas em registradores que eram acessíveis a partir de um computador acoplado. Desta forma, a reconfiguração podia ser feita via *software*, sem intervenção manual.

Em meados dos anos 80, após a criação dos primeiros componentes de lógica programável (PALs e PLAs), a XilinxTM [Xilinx 2006] e AlteraTM [Altera 2006] fabricaram os primeiros dispositivos de lógica programável comerciais [Brown 1996]. Somente a partir daí é que este novo conceito começou a ser investigado como um novo paradigma, mais tarde denominado por computação reconfigurável. De lá para cá, a computação reconfigurável vem atraindo cada vez mais pesquisadores, visto suas possibilidades e áreas de aplicação. As

possibilidades de aplicação da computação reconfigurável são enormes devido à sua adaptabilidade e versatilidade, e vão desde sistemas embutidos, muito comuns nos eletrodomésticos atuais, até sistemas de comunicação e exploração espacial.

Os FPGAs (*Field Programmable Gate Arrays*) são os dispositivos de lógica programável mais utilizados pela comunidade de computação reconfigurável [Hauck 1998]. Os mais recentes FPGAs integram facilidades voltadas para este paradigma como bancos de memórias internos ao dispositivo, capacidade de reconfiguração parcial do dispositivo e tempos de reconfiguração mais reduzidos.

Tendo os processadores de sinais digitais (DSPs) atingido o limite de potência dissipada facilmente resfriada à temperatura ambiente, torna-se cada vez mais complexo produzir *chips* com velocidade de processamento maiores, abrindo oportunidades para *chips* onde o processamento eficiente resulte mais do paralelismo entre operações do que da velocidade intrínseca da velocidade de execução de cada operação elementar. Tudo isto pode permitir que os dispositivos reconfiguráveis tenham um papel revolucionário nos sistemas computacionais do futuro.

2.3 Arquitetura de um FPGA

Um FPGA é um dispositivo semicondutor que contém componentes lógicos programáveis e interconexões programáveis. Estes componentes lógicos podem ser programados para expandir a funcionalidade de portas lógicas básicas (tais como portas AND, OR, XOR, etc.) ou funções combinatórias mais complexas como decodificadores ou funções matemáticas simples. Em muitos FPGAs, estes componentes lógicos programáveis (ou blocos lógicos, ou células) também incluem elementos de memória, que podem ser simples *flip-flops* ou módulos de memória completos. Uma hierarquia de interconexões programável permite aos blocos lógicos de um FPGA serem interconectados pelo projetista do sistema como

necessário. Estes blocos lógicos e interconexões podem ser programados de maneira tal que o FPGA pode executar qualquer função lógica.

Os FPGAs são geralmente mais lentos que os circuitos integrados de aplicação específica e não podem suportar projetos tão complexos quanto os ASICs em geral. Entretanto possuem diversas vantagens como resposta de mercado rápida, habilidade de reprogramação para corrigir erros e baixos custos de engenharia. Os FPGAs podem ainda serem utilizados como base de teste para projetos de microprocessadores. Neste caso, o desenvolvimento e testes destes projetos são feitos em FPGAs normais sendo então migrados para a versão definitiva no ASIC.

Genericamente, uma célula de FPGA típica consiste de uma LUT de 4-entradas e um *flip-flop* como mostrado na figura 2.1.

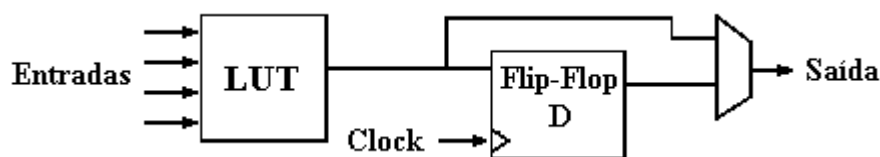


Figura 2.1 – Célula genérica da maioria dos FPGAs.

Em tal célula existe apenas uma saída, que pode ser a saída direta da LUT ou a saída vinda do registrador. A célula possui 4-entradas para a LUT e uma entrada de *clock*. A disposição das entradas e da saída de tal célula é mostrada na figura 2.2.

Cada entrada é acessível de um lado da célula, enquanto que o pino de saída pode se conectar aos fios de roteamento em ambos os canais à direita e abaixo da célula. Cada pino de saída da célula por sua vez pode se conectar a qualquer um dos segmentos dos canais adjacentes a ele. A figura 2.3 ilustra esta situação.

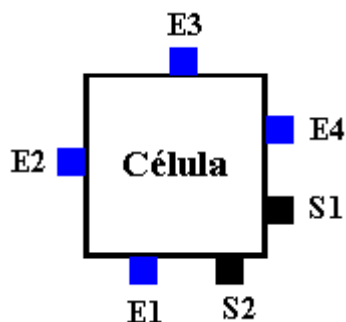


Figura 2.2 – Localização das entradas e saída de uma célula.

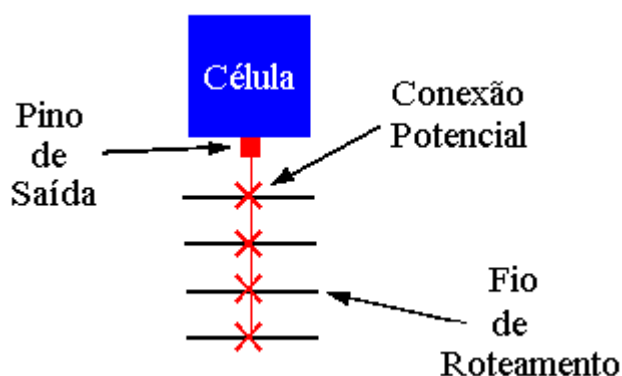


Figura 2.3 – Ligação do pino de saída aos segmentos do canal de roteamento.

A arquitetura básica típica de um FPGA consiste em uma cadeia de células e canais de roteamento como mostrado na figura 2.4. Geralmente, todos os canais de roteamento possuem o mesmo comprimento (número de fios).

Algumas arquiteturas possuem, além da rede de roteamento, linhas globais que provêm conexões de alta velocidade, conseqüentemente com poucos desvios, à todas as células da matriz do FPGA, suprindo sinais como *clock*, *resets* e outros tipos de sinais globais.

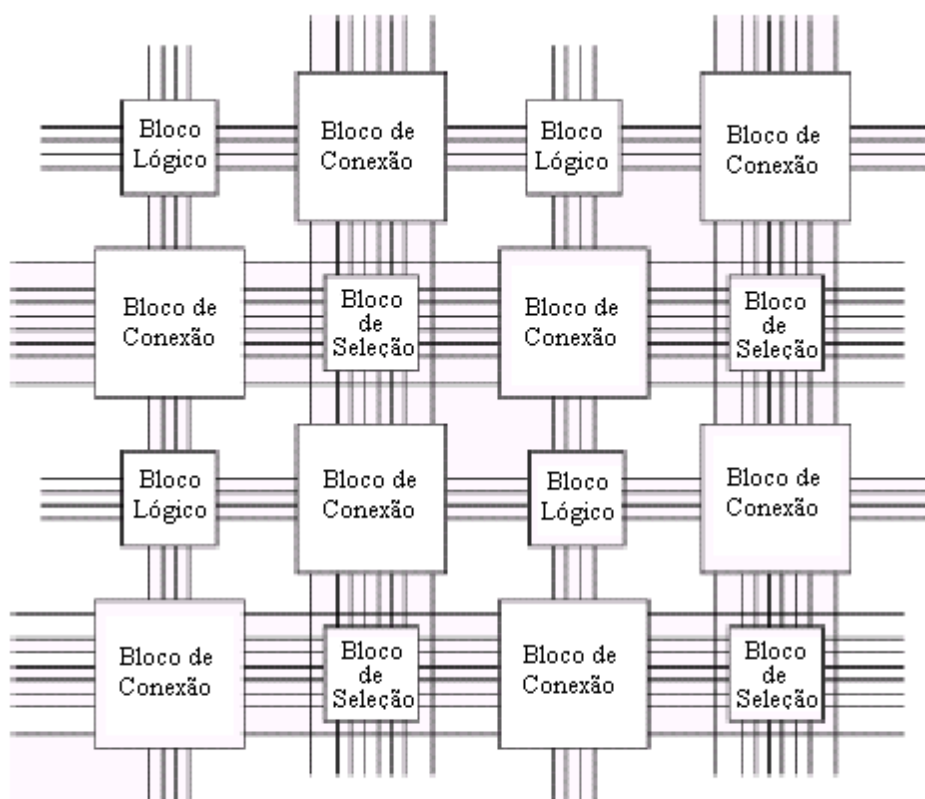


Figura 2.4 – Estrutura genérica de um FPGA.

A implementação da célula como na figura 2.4 não é única. Existem várias outras propostas de arquitetura, onde as células implementam lógicas de maior complexidade, tais como unidades lógico aritméticas (ALUs), ou até mesmo núcleos de processamento completos como microprocessadores com módulos de memórias. Esta diferença no tamanho da célula é usualmente referida como *granularidade*. Compton e Hauck [Compton e Hauck 2000] classificam as arquiteturas quanto a granularidade em três subconjuntos:

- **Granularidade fina:** as células implementam funções simples como portas lógicas *AND*, *OR*, *XOR*, *NOR*, sendo a base para implementação de circuitos lógicos mais complexos como somadores e multiplicadores. Os FPGAs de caráter comercial pertencem a este subconjunto.
- **Granularidade média:** as células implementam lógicas mais complexas que as

de granularidade fina tais como funções lógico-aritméticas e multiplicadores de poucos *bits*.

- **Granularidade grossa:** as células implementam lógicas completas como, por exemplo, somadores e multiplicadores de maiores quantidades de *bits* e até em nível de *bytes*. Além disto, pode-se encontrar arquiteturas com ALUs completas e até mesmo pequenos processadores.

É difícil estabelecer um limiar entre as diversas granularidades. A bibliografia da área contém vários exemplos que nos levam a ver que o limite entre elas é de certa forma indefinível. Compton e Hauck mencionam que a granularidade fina é útil na manipulação ao nível de *bit* sendo possível a construção de estruturas de computação com largura de *bit* arbitrárias, a granularidade média é útil na implementação de circuitos de caminho de dados de larguras de bit variadas, enquanto que células com granularidade grossa são mais otimizadas para aplicações de caminho de dados de comprimento de palavra (múltiplos de *byte*).

Existem ainda propostas de arquiteturas baseadas em múltiplas unidades de FPGA em uma única placa de processamento. Neste tipo de arquitetura, existe também uma rede de roteamento, para comunicação inter-FPGAs, como ocorre entre as células de cada FPGA. Tal sistema pode ser visto na figura 2.5.

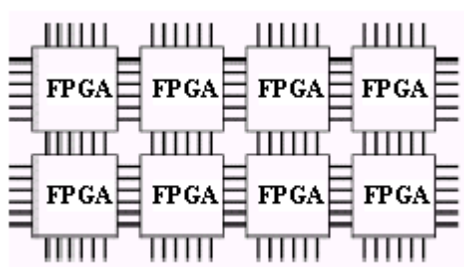


Figura 2.5 – Exemplo de arquitetura multi-FPGA

2.4 Suporte aos Sistemas Reconfiguráveis

Freqüentemente, dispositivos de lógica reconfigurável são acoplados a microprocessadores tradicionais, pois os dispositivos de lógica programável tendem a ser ineficientes na implementação de operações como ciclos e desvio de execução condicional comuns em programas criados em linguagens de alto nível [Compton e Hauck 2000]. Desta forma, as partes da aplicação que possuem tais tipos de estruturas são executadas em *software* em um microprocessador tradicional, enquanto que as partes da aplicação de fluxo de execução mais intensas são mapeadas para o dispositivo reconfigurável. Neste tipo de acoplamento, o *hardware* reconfigurável pode ser utilizado de várias formas como se segue:

- Como um meio de prover unidades funcionais dentro de um processador hospedeiro [Razdan e Smith 1994; Hauck *et al.* 1997]. Neste caso, as unidades reconfiguráveis executam dentro do caminho de dados do processador sendo utilizados registradores para guardar os operandos de entrada e saída.
- Como um co-processador [Wittig e Chow 1996; Chamaeleon 2000]. Neste caso, o processador pode inicializar o componente reconfigurável e enviar dados para que este faça a respectiva computação de forma independente do processador central. Ao término da computação, os resultados são repassados ao processador central. Esta forma de ligação permite ao componente reconfigurável e o processador central executarem simultaneamente diminuindo a sobrecarga de computação.
- Como uma unidade de processamento reconfigurável ligada a um processador como se fosse um sistema multi-processador [Vuillemin *et al.* 1996; Annapolis 1998; Laufer *et al.* 1999]. A comunicação entre os dispositivos é feita através de primitivas especializadas como nos sistemas multi-processadores tradicionais.

Este tipo de sistema permite um alto grau de independência de computação, pois permite a troca de grandes quantidades de dados com o dispositivo reconfigurável.

- Como uma unidade de processamento *standalone* externa [Quickturn 1999a, 1999b]. Neste caso, o dispositivo reconfigurável se comunica poucas vezes com o processador central.

De forma geral, quanto mais fortemente acoplado o componente reconfigurável, mais ele pode ser utilizado dentro da aplicação, pois acarreta uma pequena sobrecarga de comunicação. Mas desta forma, o componente não pode operar muito tempo sem a intervenção do processador central. Um acoplamento mais fraco entre o componente reconfigurável e o microprocessador permite um maior paralelismo na execução do programa. Entretanto, neste caso, perde-se com a sobrecarga de comunicação entre os dispositivos. Desta forma, o tipo da arquitetura de suporte ao sistema reconfigurável deve ser escolhida cuidadosamente de acordo com o tipo da aplicação.

2.5 *Software* para Sistemas Reconfiguráveis

Embora a computação reconfigurável tenha se mostrado uma excelente alternativa para aumento de performance de sistemas, ela necessita de ferramentas que permitam que seja facilmente incorporada aos sistemas pelos programadores. Tais ferramentas podem ser desde assistentes de criação manual de circuitos até ferramentas completas de projeto automático de circuitos. A construção manual de circuitos requer um alto nível de conhecimento relativo à plataforma reconfigurável a se utilizar, além de um grande tempo de projeto. Desta forma, ferramentas de geração automática de sistemas acabam se tornando mais atrativas para programadores de aplicações gerais, visto que permitem uma forma fácil e rápida de se criar

sistemas reconfiguráveis. Tanto o processo manual quanto o automático destes sistemas possuem um número distinto de fases. A figura 2.6 ilustra estas fases para os diversos processos.

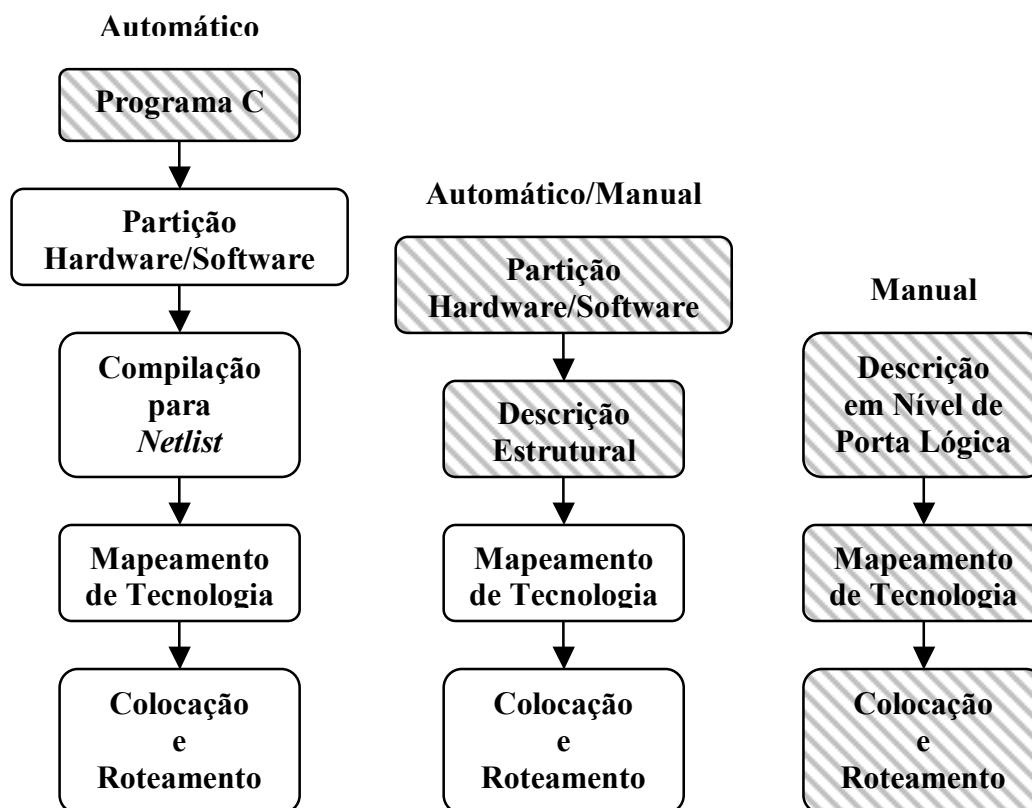


Figura 2.6 – Fases dos diversos meios de criação de sistemas reconfiguráveis.

Na figura 2.6, as fases em que o trabalho é feito manualmente estão hachuradas.

A descrição do circuito pode ser feita tanto em nível de manipulação de portas lógicas quanto através de uma linguagem de alto nível como C. Na primeira situação a descrição torna-se extremamente complexa, visto que é necessário lidar com entradas e saídas de todas as operações que envolvem o sistema. Já a segunda fornece um meio simples e rápido para a descrição. A descrição através de linguagens de alto nível pode ser feita também por linguagens de descrição de hardware (HDLs) como por exemplo VHDL e Verilog. Dada uma descrição estrutural, criada em uma linguagem de alto nível ou especificada manualmente

pelo usuário, a mesma pode ser substituída por elementos de *hardware* representados em alto nível que abstraem a função original. Depois, estes elementos podem ser mapeados para os componentes específicos da arquitetura do dispositivo reconfigurável (mapeamento tecnológico).

O processo completamente manual é extremamente dispendioso e suscetível a erros. Nele o projetista executa todas as fases manualmente até a obtenção do sistema final. No processo misto o projetista decide quais as partes do sistema vão ser mapeadas para *hardware* e *software*. Depois de feita a descrição estrutural do sistema, a mesma pode ser mapeada para o dispositivo reconfigurável por ferramentas de fabricantes. No processo automático, o projetista tem apenas que descrever o sistema em uma linguagem de alto nível para a posterior compilação do sistema. O compilador pode ser responsável por detectar automaticamente as partes da descrição que deverão ser mapeadas para *hardware* e *software*. De outra forma, o usuário pode expor explicitamente estas partes utilizando-se de recursos da linguagem fonte (como os *pragmas* da linguagem C). Uma vez feita esta detecção automática ou manual, a descrição é mapeada nos elementos de *hardware* representados em alto nível. Geralmente estes elementos estão descritos em uma linguagem de descrição de *hardware* que permite maior abrangência em relação aos dispositivos reconfiguráveis alvos. Depois, através de ferramentas de fabricante, esta nova representação é compilada e encaminhada para o dispositivo reconfigurável.

2.5.1 Modelo de Co-Projeto *Hardware/Software*

Na maioria dos sistemas computacionais reconfiguráveis os dispositivos de *reconfigware* (*hardware* reconfigurável) trabalham em conjunto com os sistemas de *software* para se aproveitar as potencialidades de ambos os tipos de computação. Em tal modelo, chamado de co-projeto *hardware/software* (*Hardware/Software Co-Design*), parte (ou partes)

da aplicação é traduzida em circuito digital para execução em um FPGA. O resto da aplicação é executado sob a forma de *software* convencional sob um microprocessador. Tais sistemas, por residirem em uma plataforma reconfigurável, são também chamados de SOPCs (*Systems On a Programmable Chip*).

O predomínio de tal modelo se deve ao fato de que, além da dificuldade dos dispositivos reconfiguráveis implementarem operações como ciclos e desvio de execução condicional (daí a tradução somente de partes apropriadas do programa para *hardware*), a utilização de sistemas reconfiguráveis exige trabalhos que, além de serem longos e difíceis, requerem conhecimentos específicos em projeto de *hardware*. Tal situação acaba por não despertar interesse nos programadores de *software* devido principalmente à necessidade de conhecimento de projeto de *hardware*.

Para a utilização deste modelo, tornou-se necessária a construção de ferramentas de automação que gerem o circuito para *reconfigware*, partindo-se preferencialmente de uma representação de mais alto nível, como por exemplo, as linguagens de programação de *software*, que podem ser mais facilmente entendidas e utilizadas pelos programadores. Desta forma, um modelo misto de *hardware* e *software*, em que a parte *hardware* é automaticamente gerada e abrange apenas uma pequena parte do programa, acaba por atrair mais os programadores pois livra-os de lidar com detalhes sobre o *hardware*.

Existem hoje várias ferramentas com este propósito de automação. A maioria delas parte de uma linguagem de programação de uso comum pelos programadores para a geração do *hardware* específico. Tais ferramentas são especializadas na análise do código fonte da aplicação e a geração do circuito digital relativo à(s) parte(s) do programa a ser(em) lançada(s) em *reconfigware*. Estas ferramentas, em sua maioria, traduzem tais partes (especificadas pelo programador ou deduzidas através de heurísticas) para uma representação intermediária, geralmente sob a forma de grafos de fluxo, que mantém suas instruções e os

possíveis caminhos de execução do subprograma. Através desta representação intermediária, essas ferramentas podem realizar uma série de otimizações antes de gerar a representação final do circuito. Tal processo de geração de circuitos de sistemas digitais a partir de descrições de alto nível é denominado de *síntese de alto nível* ou *síntese arquitetural* [Spark 2000; Gajski 1992].

Otimizações do programa (em sua representação intermediária) feitas por ferramentas de síntese de alto nível são necessárias porque muitas vezes os circuitos gerados não competem em qualidade com projetos feitos manualmente (fato muito comum entre as propostas mais antigas). Desta forma, existe um gargalo entre os modelos representados nas linguagens de nível de sistema e a implementação do componente de *hardware* [Gupta *et al.* 2004]. Tal problema ocorre porque a maioria das transformações de otimização que foram propostas durante os anos são transformações ao nível de operações da linguagem. As otimizações ao nível de linguagem, como por exemplo, remoção de ciclos, movimentação de código ciclo-invariante, que permitem a mudança da descrição do circuito no nível de sua descrição fonte, ainda não foram suficientemente exploradas.

A linguagem de programação de uso comum pelos programadores mais utilizada como fonte de descrição do programa a ser gerado *hardware* específico é a linguagem *C* [Cardoso 2000]. Existem também propostas para *C++* e até mesmo *JAVA*. As propostas mais recentes procuram fazer otimizações no nível de linguagem. Uma abordagem em que o compilador executa as otimizações em cima de uma representação intermediária criada a partir dos *bytecodes* do *JAVA* é apresentada por Cardoso em [Cardoso 2000]. Até onde sabemos, todos os compiladores para síntese de alto nível são restritivos quanto a certos recursos da linguagem. Por exemplo, alguns são restritivos quanto ao uso de ponteiros e muitos nem mesmo suportam chamadas de funções dentro do código a ser traduzido. O compilador de *JAVA* apresentado por Cardoso não suporta referências a objetos *JAVA* dentro

do código a ser tratado.

2.6 Conclusão

A computação reconfigurável surgiu com o objetivo de preencher a lacuna existente entre os sistemas baseados em execução de *hardware* dedicado (*ASICs*) e execução de instruções *software* (microprocessadores de propósito geral) antes existentes e bem distintos. Tais sistemas eram de certa forma inflexíveis. A possibilidade de modificação do *hardware* durante o tempo de vida do dispositivo permitida pelos dispositivos reconfiguráveis provê maior adaptabilidade do processo de computação permitindo alternativas como a adaptação do *hardware* à aplicação e a execução de partes da aplicação como *hardware*. A computação reconfigurável provê maior performance que *software* enquanto mantém flexibilidade para o *hardware*. A flexibilidade provida pela computação reconfigurável permitiu ainda a introdução de um modelo de execução misto de *software/hardware* adaptável. Neste modelo, partindo-se de uma descrição do circuito em alto nível, pode-se gerar automaticamente o circuito para o dispositivo reconfigurável.

O suporte de arquiteturas para este paradigma de computação é bem diversificado, permitindo generalidade de suporte para as aplicações. O campo de utilização desta tecnologia é imenso, abrangendo todas as áreas da computação, desde jogos de computador até exploração robótica espacial.

Capítulo 3

Compilação para *Reconfigware*

3.1 Introdução

Este capítulo aborda o processo de criação de circuitos para *reconfigware* a partir de descrições em alto nível do *hardware* desejado. É descrito sucintamente o fluxograma genérico do processo de síntese de alto nível de circuitos digitais para dispositivos reconfiguráveis sendo ilustrados os passos, estruturas de dados e algoritmos geralmente utilizados. São apresentados ainda alguns dos trabalhos mais recentes relativos ao processo de síntese de alto nível de circuitos digitais. Por fim é apresentada uma conclusão.

3.2 Processo de Compilação para *Reconfigware*

De uma forma geral, o processo de compilação para *reconfigware* se assemelha ao processo de compilação tradicional onde é gerado código para processadores reais ou virtuais. Neste processo, um programa descrito em linguagem de alto nível é traduzido para uma representação intermediária de onde pode ser feita uma série de otimizações. Esta melhoria pode ser no âmbito da velocidade de execução do código e/ou no tamanho do código gerado. Após as otimizações, é feita a geração de código para a arquitetura alvo. A figura 3.1 ilustra este processo.

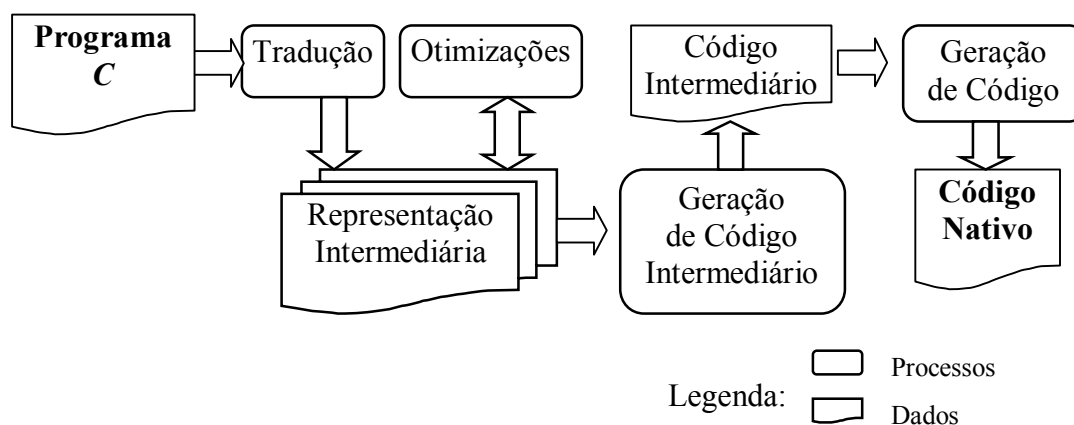


Figura 3.1 – Processo de compilação tradicional.

No processo de compilação tradicional, a descrição do programa fonte é traduzida para a representação intermediária durante a fase de análise sintática. Nesta fase, os *tokens*, reconhecidos pelo analisador léxico, são agrupados hierarquicamente em coleções aninhadas com significado coletivo. A estrutura hierárquica é usualmente expressa por regras recursivas através de gramáticas livres de contexto e a representação intermediária através de árvores sintáticas e instruções de três endereços. Durante a fase de análise sintática são feitas certas verificações para assegurar que os componentes do programa combinam de acordo com seu significado (semântica). Durante este processo de verificação parcial de corretude, chamado de análise semântica, são verificados erros semânticos no programa fonte e são capturadas informações de tipos para a fase subsequente de geração de código. Um componente importante desta análise é a verificação de tipos, onde é verificada se o tipo de uma construção e o tipo de seus operandos ou argumentos é compatível. Uma vez criada a representação intermediária da estrutura do programa e suas instruções, podem ser aplicadas otimizações em cima do código nesta representação visando ganhos de desempenho na execução do programa. Existem otimizações tanto triviais quanto otimizações que exigem uma análise profunda das instruções do programa. A geração de código cria o respectivo

código nativo de uma máquina específica ou código para posterior montagem. As instruções em formato intermediário são traduzidas numa seqüência de instruções de máquina que realizam a mesma tarefa.

No caso da compilação para *reconfigware*, pode ser gerada uma representação intermediária mais extensa constituindo-se de vários grafos que armazenam informações de fluxo de controle e de dados além de informações de dependências de ordem de execução entre as instruções e os blocos básicos do programa. O objetivo é reduzir o tamanho e aumentar o desempenho do circuito, reestruturando ciclos e localização de instruções, e identificando o maior número possível de operações que podem ser executadas em paralelo. A representação intermediária pode ser reorganizada significativamente durante este processo.

Em alguns casos é gerada uma representação na forma de grafo que representa, em alto nível, todo o circuito. Nesta representação podem estar incluídas todas as operações, organizadas de acordo com seu tempo de execução, as condições para a execução de tais operações, o controle do fluxo de dados para os registradores através de multiplexadores, as funções lógicas destes multiplexadores, entre outros.

Sem perda de generalidade, a compilação para *reconfigware* segue os passos apresentados nas seções seguintes.

3.2.1 Geração da Representação Intermediária

Nesta fase são gerados os grafos de fluxo de controle (CFG) e o grafo de fluxo de dados (DFG) [Gajski 1992] durante a tradução do programa fonte. O grafo de fluxo de controle consiste em nós que encapsulam os blocos de instruções do programa ligados de acordo com os possíveis caminhos de execução do mesmo. O grafo de fluxo de dados representa o fluxo de dados entre as operações do programa. Desta forma, cada nó do grafo representa uma operação, sendo ligado a uma outra operação (nó) se existir um fluxo de

algum dado para a mesma. Além destes grafos, são criados os grafos de dependência de controle e de dependência de dados. Nestes grafos as ligações entre os nós representam dependências de ordem de execução em relação ao fluxo de controle e ao fluxo de dados. A figura 3.2 ilustra todos estes grafos para um pequeno programa ilustrativo na linguagem C.

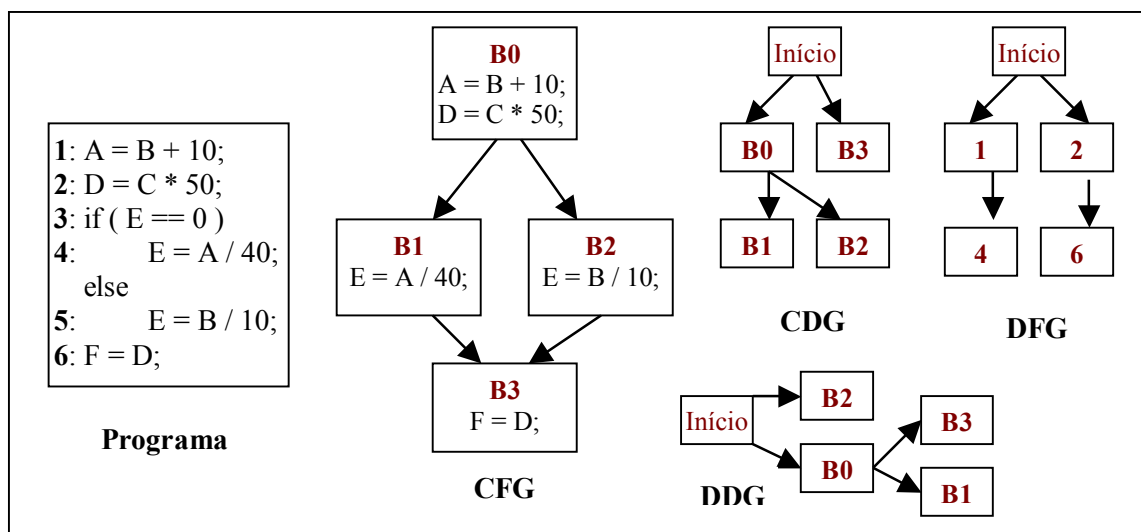


Figura 3.2 – Grafos de fluxo de controle, dependência de controle, fluxo de dados e dependência de dados para um programa exemplo.

Na figura 3.2, o CFG mostra os possíveis caminhos de execução para o programa em questão, que no caso são os dois caminhos possíveis decorrentes do bloco condicional existente. O CDG mostra que os blocos B1 e B2 são dependentes em termos de controle do bloco B0, pois a partir deste, pode ser feito um desvio condicional de execução para qualquer um deles. O DFG mostra que existe um fluxo de dados (variável A) da primeira instrução para a quarta instrução. Existe também um fluxo de dados da segunda para a sexta instrução (variável D). O DDG mostra que existe um fluxo de dados do bloco B0 para o bloco B1 pelo fato de B1 utilizar o valor da variável A que foi definida (atribuída) em B0. O mesmo ocorre para o bloco B3 em relação à variável D. Desta forma, pode-se concluir que os blocos B1 e B3 devem ser executados após a execução do bloco B0, visto a dependência de dados. O

bloco B2 não depende de nenhum outro bloco em termos de dependências de dados e desta forma poderia ser executado em paralelo com o bloco B0, por exemplo.

Algumas vezes, além de serem criados os grafos descritos, são criados grafos que encapsulam hierarquias do programa fonte, como por exemplo, blocos condicionais (*if-else*) e ciclos. Nestes grafos estas hierarquias são representadas em um único nó, permitindo mais flexibilidade para certos tipos de transformações de otimização.

3.2.2 Otimização dos Grafos

A otimização do programa para fins de geração de *hardware* se concentra em cima da análise dos grafos gerados no passo anterior. Estas transformações podem ser em nível dos blocos básicos, buscando paralelismo entre as instruções neste contexto, ou em nível de linguagem, alterando a estrutura original do programa no contexto das estruturas permitidas pela linguagem. As otimizações em nível dos blocos básicos mais comuns são: eliminação de sub-expressões comuns, propagação de cópias, eliminação de código morto e transposição para constantes. As otimizações em nível de linguagem mais comuns são: movimentação de código ciclo-invariante, remoção de ciclos, movimentação de código especulativa e encadeamento de operações.

3.2.3 Mapeamento de Componentes *Hardware*

Nesta fase, as operações do programa são mapeadas nos componentes em *hardware* que executam a mesma função. Geralmente estes componentes são representados em alto nível, não sendo assim específicos a determinado dispositivo. Tais componentes são também geralmente agrupados em uma biblioteca chamada de Biblioteca Tecnológica. Esta biblioteca contém componentes de *hardware* para soma, subtração, multiplicação e divisão de números inteiros e flutuantes, que abstraem as operações da linguagem, além de componentes para

acesso a memórias externas para as operações de carregamento e armazenamento. O mapeamento das operações consiste na busca na Biblioteca Tecnológica do respectivo componente que possui suporte ao processamento com o número de *bits* dos operandos da operação em questão. Muitas vezes, após o processo de mapeamento, o mesmo é refinado, procurando-se atribuir componentes de menor área e maior velocidade para otimizar o circuito final.

3.2.4 Escalonamento

O escalonamento é o processo de temporização das operações. Neste processo, para cada operação é atribuído o ciclo de relógio em que iniciará sua execução. Dois escalonamentos muito utilizados são o ASAP (*As Soon As Possible*) e ALAP (*As Late As Possible*). Os escalonamentos são aplicados por atravessamento no DFG de cima para baixo e de baixo para cima respectivamente. O ASAP segue a regra básica de que um nó sucessor pode executar apenas após a execução de seu nó pai e cria o escalonamento mais rápido possível para as operações. O ALAP cria o escalonamento mais lento possível para as operações. O escalonamento leva em conta o tempo de atraso de cada componente *hardware* mapeado para as operações na fase anterior.

3.2.5 Geração do Circuito

O circuito representativo do programa é composto de duas partes. Uma parte, chamada de unidade de dados, representa o circuito que implementa o fluxo de dados do programa original. A outra parte consiste na unidade que controla o fluxo dos dados na unidade de dados, recebendo daí o nome de unidade de controle. A construção da unidade de dados é feita através da varredura do grafo de fluxo de dados otimizado. Nesta varredura são atribuídos registradores às variáveis existentes no grafo, são inseridos circuitos especializados

para acesso a memórias externas quando for o caso e são inseridos multiplexadores (ou estruturas equivalentes) nos pontos de seleção de dados para as operações. A construção da unidade de controle ocorre pela criação de uma ou mais máquinas de estado finito. As máquinas de estado finito são criadas após o processo de escalonamento. Para cada passo de escalonamento criado, é criado um respectivo estado na máquina de estados. Cada estado mantém informações sobre qual é o próximo estado da transição e a condição para que esta transição seja feita. A representação do circuito final usualmente é feita em uma HDL, pois desta forma, o circuito pode ser sintetizado para diversas plataformas reconfiguráveis através de ferramentas de fabricante. A representação da máquina de estados pode também ser feita da mesma maneira. Existem compiladores em que a geração do circuito é específica para um dispositivo reconfigurável e desta forma é gerada diretamente a cadeia de configuração (*bitstream*) para a reconfiguração do dispositivo.

A figura 3.3 ilustra o fluxograma geral do processo de síntese de alto nível.

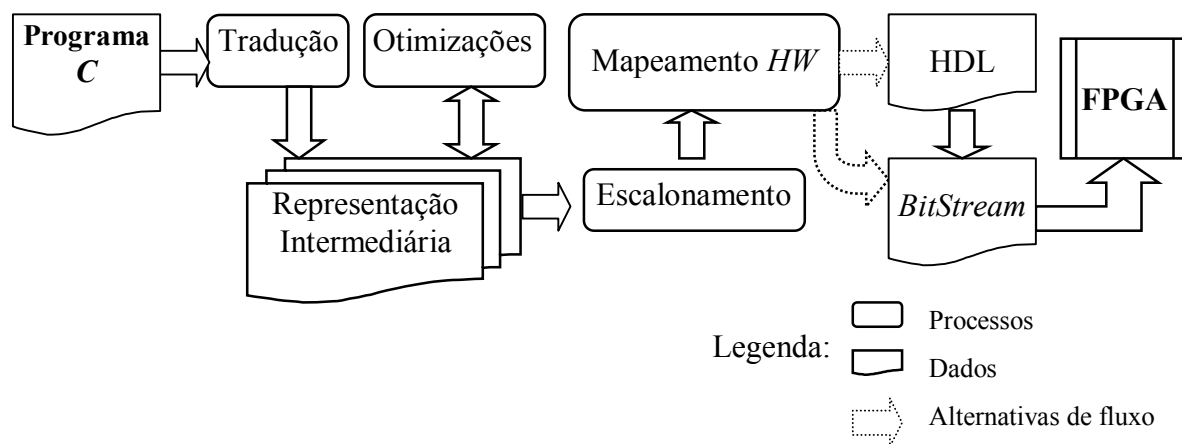


Figura 3.3 – Fluxograma genérico da síntese de alto nível de circuitos digitais.

3.3 Compiladores de *Reconfigware*

Um compilador, em seu sentido mais abrangente, é um tradutor. Um conversor de texto em Português para Inglês, por exemplo, é um tipo de compilador. Outro exemplo seria a geração de código de máquina (real ou virtual) a partir de linguagens de alto nível. Quando esta máquina tem uma arquitetura pré-associada, temos o processo de compilação tradicional. Quando a máquina alvo não tem nenhuma arquitetura pré-associada (como é o caso de um circuito ASIC), o processo é denominado de síntese de *hardware* (inicialmente denominada de compilação de silício). Os máquinas para computação reconfigurável, também chamadas de máquinas de computação personalizadas no campo (*Field-Custom Computing Machines - FCCMs*), se situam entre estes dois tipos de modelos de computação, pois possuem uma arquitetura pré-definida (matriz de blocos lógicos configuráveis) e também a flexibilidade de poderem teoricamente implementar qualquer circuito digital assim como os ASICs [Cardoso 2000].

Os compiladores de silício tiveram sua origem em meados da década de 80 assim como a compilação de linguagens de alto-nível em hardware específico. Depois destas abordagens iniciais a investigação foi centrada em linguagens capazes de descrever *hardware* (HDLs). A geração de circuitos digitais através da compilação de sua descrição em linguagens de programação (C principalmente) tem ganhado cada vez mais adeptos, principalmente devido à enorme quantidade de algoritmos especificados nestas linguagens. Vários compiladores foram criados tentando resolver problemas relativos à síntese arquitetural, tais como otimização do circuito final, diminuição do tamanho do circuito, tempo de compilação, partição temporal, entre outros. No restante deste capítulo são apresentados alguns dos trabalhos mais recentes relativos à compilação de programas em linguagens com nível de abstração elevado para sistemas de computação reconfigurável. São apresentados os compiladores para o *Garp* [Callahan *et al* 2000], *Galadriel* e *Nénia* [Cardoso 2000], *Spark*

[Gupta *et al.* 2003] e *Impulse C* [Impulse 2006] que são posteriores ao ano de 1999. Cardoso [Cardoso 2000] faz uma excelente revisão sobre trabalhos anteriores ao ano de 2000.

3.3.1 *Garp*

Garp é uma arquitetura proposta em 2000 que combina um microprocessador MIPS e um *hardware* reconfigurável e foi idealizada para computação de propósito geral que inclui execução de programas estruturados, bibliotecas de sub-rotinas, mudança de contextos, memória virtual e múltiplos usuários. O módulo reconfigurável da arquitetura atua como um co-processador. Dados são movidos do processador MIPS para o dispositivo reconfigurável através de instruções especiais de movimentação de dados. A arquitetura foi concebida com o propósito de acelerar a execução dos ciclos de programas através da transformação e execução destes como *hardware* otimizado. A arquitetura permite a reconfiguração do dispositivo em tempo de execução através de estruturas de reconfiguração armazenadas em memória *cache*. Durante a execução do programa, quatro passos são executados toda vez que é encontrado um ciclo que foi transformado em *hardware*:

4. É lida a configuração correspondente à implementação do ciclo. Se a configuração já se encontra em *cache*, o processo de leitura da mesma dura cerca de 5 ciclos de relógio.
5. São copiados os dados para o co-processador através das instruções especiais.
6. É iniciada a execução do circuito, sendo deixado o processador MIPS em estado de espera.
7. Ao término da execução no coprocessador, os resultados são retornados.

É provido um compilador para a linguagem C chamado de *Garpcc*. Este compilador procura por ciclos de execução intensa no código e gera uma estrutura, chamada de hiperbloco [Mahlke *et al.* 1992], para cada um deles. Esta representação é comumente utilizada por compiladores para processadores do tipo VLIW (*Very Long Instruction Word*). Operações que não podem ser diretamente implementadas no dispositivo reconfigurável como chamadas à função *printf*, são mantidas separadamente à estrutura do hiperbloco. O compilador gera a estrutura de DFG, mas nela os blocos básicos são unidos utilizando-se da técnica de predicação, que elimina a necessidade de ramificação condicional. A computação é feita por todos os caminhos incluídos no grafo e predicados (valores booleanos resultantes das condições que originalmente controlavam os ramos condicionais) controlam os multiplexadores para selecionar os valores apropriados nos pontos de união de controle de execução. Operações que tem efeito fora do dispositivo reconfigurável, como acessos à memória, tem um predicado específico que habilita a operação quanto o caminho de controle da mesma é válido. O processo de síntese da representação é feito através do mapeamento direto dos nós no DFG para os módulos do dispositivo. A geração da configuração a partir do mapeamento é feita por um *software* específico para o dispositivo reconfigurável. O compilador permite que algumas técnicas avançadas sejam utilizadas como leitura especulativa, onde a leitura de um dado é feita antes do instante que deveria ser realmente feita, *pipelining*, e filas de memória que permitem otimizações especiais para aplicações que lidam com *streams* de dados.

3.3.2 *Galadriel e Nenia*

Galadriel e *Nenia* são dois compiladores para síntese de alto nível que trabalham em conjunto para criar SOPCs a partir do código objeto (*bytecodes*) gerado pelos compiladores JAVA. *Galadriel* é um compilador de anteguarda que, a partir dos *bytecodes* do programa

JAVA, constrói a representação intermediária. Esta representação consiste no grafo de fluxo de controle, o grafo de dependência de controle, o grafo de fluxo de dados, o grafo de dependência de dados, o grafo de dependências de fusão (MDG) e o grafo hierárquico de dependências de programa (HPDG), sendo que os dois últimos grafos foram propostos pelo autor. *Galadriel* não suporta todo o conjunto JAVA, não permitindo, por exemplo, que o código a ser analisado possua invocação de métodos. Consequentemente, a criação de objetos também não pode ser feita no mesmo trecho de código.

Na construção da representação intermediária, *Galadriel* primeiramente constrói o CFG a partir da análise das instruções da JVM, onde tais instruções são agrupadas em blocos e cada bloco é conectado com os outros blocos de acordo com o fluxo de controle. Na criação destes blocos, instruções que geram exceções não foram consideradas como delimitadoras dos mesmos.

Após a construção do CFG, é criado o grafo de dependência de controle. Para isto é criada a árvore de pós-dominâncias a partir do CFG. Em seguida, o CDG é criado a partir da árvore de dominância através de um algoritmo baseado nas seguintes considerações:

Um nó $v1$ é dependente em termos de controle do nó $v2 \Leftrightarrow$

1. Existir um caminho não nulo de $v2$ a $v1$, de modo a que $v1$ pós-domina todos os nós após $v2$ no caminho.
2. $v1$ não pós-domina estritamente o nó $v2$.

A figura 3.4 ilustra a árvore de pós-dominâncias e o CDG gerado para um programa.

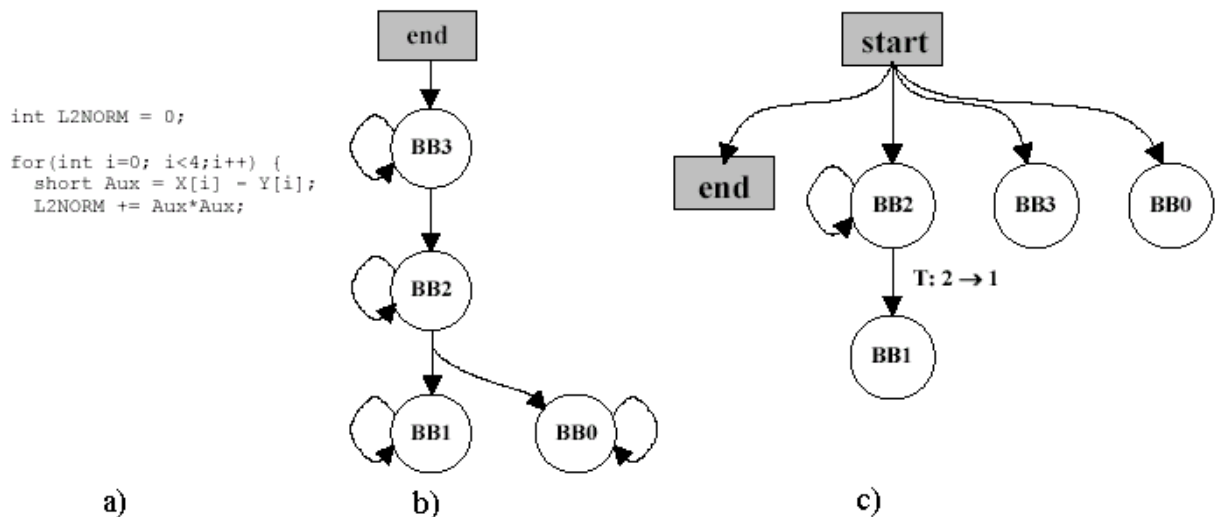


Figura 3.4 – Exemplo de grafo de pós-dominâncias (b) e CDG (c) para um programa (a).

A construção do DDG envolve a análise das dependências de dados do programa em consideração. De acordo com Aho *et al* [Aho *et al.* 1986], as dependências de dados podem ser classificadas em quatro tipos:

1. **Dependências de fluxo:** ocorrem quando uma instrução define uma posição de memória em um ponto P_X no fluxo de execução do programa e depois é lida (usada) por outra instrução em um ponto P_Y sendo que P_Y é sucessivo à P_X no fluxo de execução do programa.
2. **Antidependências:** ocorrem quando uma instrução lê uma posição de memória que será definida por outra instrução subsequente.
3. **Dependências de entrada:** ocorrem quando uma instrução lê uma posição de memória que será lida por outra instrução subsequente.
4. **Dependências de saída:** ocorrem quando uma instrução grava em uma posição de memória que será gravada posteriormente por outra instrução.

De acordo com Cardoso [Cardoso 2000], existem quatro fontes de dependências de

dados oriundas da JVM:

1. Uso de variáveis locais.
2. Uso da pilha de operandos.
3. Uso de variáveis do tipo *arrays*.
4. Uso de atributos de um objeto.

O estudo da utilização de variáveis locais pela JVM é feito pela análise de cada bloco básico, onde, para cada um, são identificadas as variáveis locais cujos valores são utilizados internamente ao bloco. Para as dependências de fluxo de dados entre blocos, são determinadas as cadeias uso-definição e definição-uso.

O estudo da dependência de dados originado pela pilha de operandos da JVM é feito através da simulação de execução de cada instrução no CFG. Após o estado da pilha ser determinado para cada instrução, é executado um algoritmo que retorna a árvore de dependências entre blocos básicos em relação à utilização da pilha de operandos.

As dependências de dados relativas ao uso de *arrays* são computadas da mesma forma que as das variáveis locais, com a diferença que as definições e os usos são determinados por inspeção das instruções da JVM de acesso a elementos do *array*. Ao contrário da análise de dependências de dados relativa as variáveis locais, são também consideradas as antidependências e as dependências de saída. A versão do compilador apresentada até então não distingue acessos a elementos diferentes do mesmo *array*.

A análise das dependências de dados oriundas do uso de atributos de um objeto foi reservada para trabalhos futuros.

As informações sobre dependências de dados de todas as fontes descritas formam juntas o grafo de dependência de dados. A partir deste grafo são identificados os pontos de seleção. Um ponto de seleção é um ponto do programa ou da representação intermediária onde é necessário colocar unidades de seleção de entre vários dados oriundos de caminhos

mutuamente exclusivos. A figura 3.5 ilustra um DDG gerado pelo compilador com dois pontos de seleção.

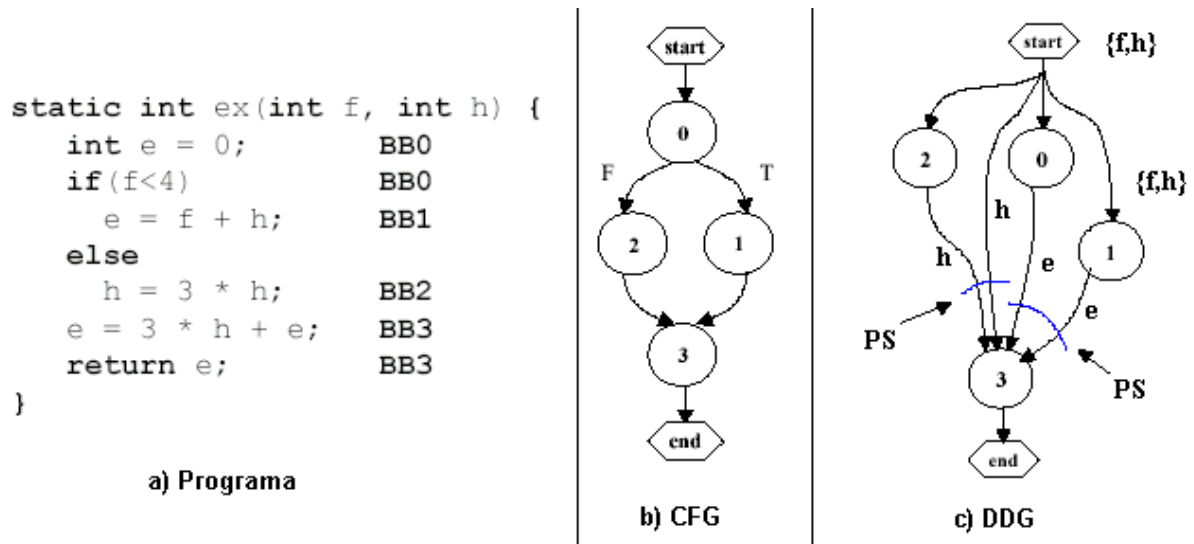


Figura 3.5 – Exemplo de pontos de seleção em um programa.

Na figura 3.5, nota-se que as variáveis h e e são utilizadas na instrução $e = 3 * h + e$. A variável e possui duas definições resultantes de instruções na própria função. A variável h possui uma definição resultante da terceira instrução da função e outra decorrente de uma definição feita anteriormente à chamada da função (h é um argumento da função ex). Desta forma, cada uma destas variáveis possui duas definições a atingir a instrução $e = 3 * h + e$. Este fato é ilustrado no DDG pelas quatro ligações (duas para cada variável) que atingem o bloco básico 3. A alternativa das definições que incidem sobre o bloco básico 3 decorre dos dois fluxos de controle do bloco condicional *if-then-else*. Dependendo da condição $f < 4$ ser verdadeira ou não, os valores das respectivas variáveis que atingirão o bloco básico 3 poderão variar. Desta forma é preciso selecionar quais destas definições vão realmente atingir o bloco básico 3. Para isto são criados os pontos de seleção que são responsáveis por escolher qual o verdadeiro valor de cada variável que irá atingir o bloco básico 3. Estes pontos de seleção serão mapeados posteriormente para dois multiplexadores, cuja função lógica será a condição

do *if-then-else* ($f < 4$) presente no bloco básico 0. A figura 3.6 ilustra este fato.

Os pontos de seleção são controlados pela lógica decisória do programa (embutida na unidade de controle) que por si é criada através do DDG e CDG. A lógica decisória é uma função lógica que representa a condição para qual determinada definição possa atingir um ponto de seleção.

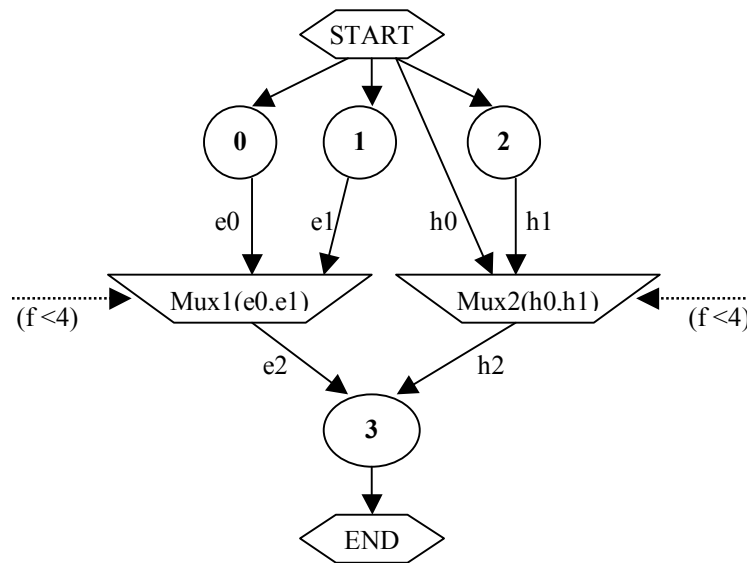


Figura 3.6 – Exemplo de pontos de seleção com multiplexadores.

O MDG é um grafo direcionado cujos laços entre nós, que por si são os blocos básicos do CFG, representam as dependências de controle de seleção ou de execução do programa. A construção do MDG é feita através dos grafos CDG e DDG.

A construção do HPDG, proposto por Cardoso, é feita através dos grafos DDG e MDG. O grafo é derivado do grafo hierárquico de tarefas (HTG) [Girkar e Polychronopoulos 1992] e é uma representação eficiente para lidar com paralelismo funcional. O nível mais baixo da representação do grafo é ilustrado sob a forma de um DFG, no qual os nós representam operações. Cada nó do HPDG sem hierarquia representa uma região do DFG.

O DFG global incorpora todos os DFGs construídos para os blocos básicos. Sua construção se dá com auxílio do HPDG. O DFG global representa todo o programa, tendo nos

nós a representação de operações, variáveis, constantes, e escrita e leitura em memória. Seus laços representam o fluxo de dados entre nós ou simplesmente precedências de execução de operações. O grafo contém ainda mecanismos que implementam os fluxos de controle sempre que haja necessidade de preservação dos mesmos. A figura 3.7 ilustra um DFG global obtido pelo compilador para o programa exemplo.

Após a construção da representação intermediária pelo compilador *Galadriel*, entra em cena o compilador *Nenya* que é responsável pela geração de *hardware* re-configurável a partir dos modelos de representação anteriormente gerados. *Nenya* tem como alvo a compilação para uma arquitetura constituída por um FPGA acoplado a uma ou mais memórias RAM via barramentos independentes. O compilador atua sobre o DFG global, mapeando operações em unidades funcionais e fazendo atribuição de registros às variáveis para geração do *reconfigware* representativo do programa.

O *reconfigware* gerado pelo *Nenya* é composto de duas unidades que interagem entre si: a unidade de dados e a unidade de controle. A unidade de dados é responsável pelo fluxo de dados do programa original, enquanto que a unidade de controle é responsável pelo controle dos acessos à memórias externas, pelo controle de escritas em registros e pela execução correta dos ciclos. Resumidamente, a unidade de controle guia o fluxo de execução e de dados da unidade de dados, enquanto que a unidade de dados envia à unidade de controle informações sobre sua situação. Uma biblioteca de macrocélulas parametrizáveis, definidas na forma de geradores de circuitos, é utilizada para a confecção da unidade de dados. Cada operador encontrado no DFG global é mapeado em um componente correspondente da biblioteca de macro-células. Após, são executados os escalonamentos ASAP e ALAP, onde é determinado o tempo que cada operação pode usar sem afetar o atraso do programa. A construção da unidade de controle se dá através da construção do grafo de transição de estados que é depois implementado em hardware específico sob a forma de uma máquina de

estados finita. Ambas as unidades de dados e de controle são descritas em VHDL.

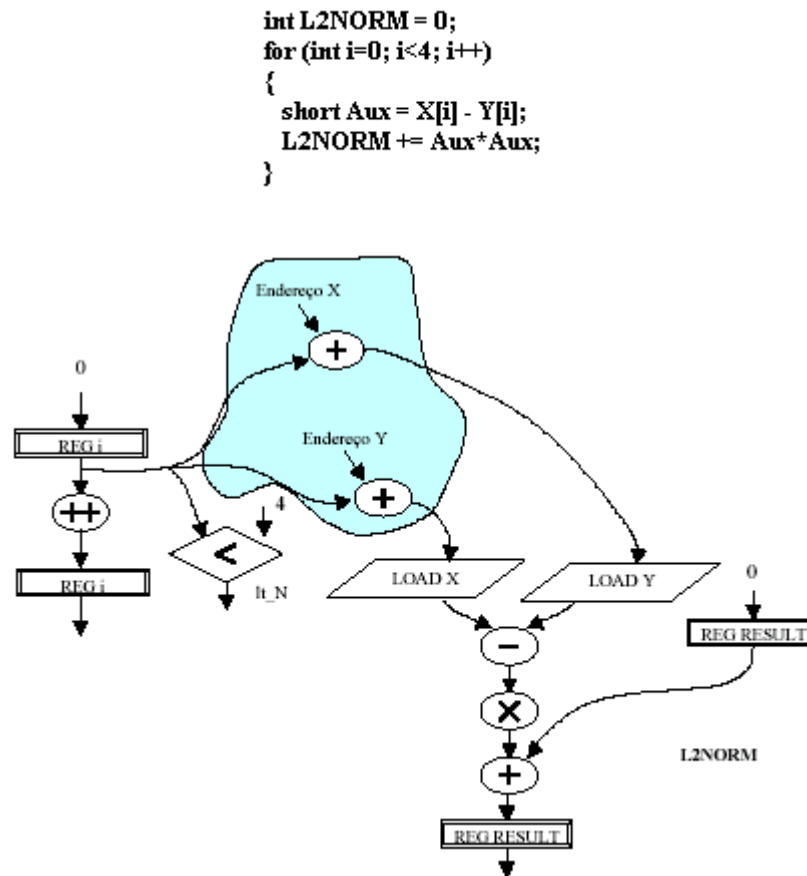


Figura 3.7 – Exemplo de um DFG global

Após a geração da representação intermediária pelo *Galadriel*, *Nênya* aplica uma série de transformações nos grafos gerados a fim de se otimizar o circuito. Estas transformações são:

1. **Re-associação de operações:** permite reordenar um DFG de modo a reduzir a distância do início até o fim (e com isto diminuir o atraso de execução do programa), expondo um maior grau de paralelismo. A transformação baseia-se nos casos mais simples no balanceamento de uma árvore, ou na utilização das propriedades associativa, comutativa e distributiva, em casos mais complexos. A re-associação de operações é efetuada apenas em cadeias de adições,

multiplicações, ANDs e ORs.

2. **Redução do Custo de Operações:** permite a substituição de operações com custos elevados (área, atraso, etc) por operações de custos menores que preservem a funcionalidade inicial. Os casos mais simples de redução do custo de uma operação ocorrem quando existe a presença de divisões ou multiplicações por constantes do tipo 2^N , em que N representa um número inteiro positivo. Quando a constante não é do tipo 2^N , a redução do custo da operação (multiplicação ou divisão) tem de ser feita com base em adições, subtrações e deslocamentos.
3. **Aferição do Número de *Bits* de Representação:** o compilador extrai para cada operador o número de *bits* suficiente de modo a preservar a funcionalidade original.
4. **Propagação de Padrões de Constantes ao Nível do Bit:** consiste em determinar os padrões de *bits* para cada operando da operação, a fim de determinar quantos *bits* o operador deverá ter para suportar a operação. Por exemplo, a comparação $00uuu0 == u0uuuu$, onde “u” pode ser tanto 0 ou 1, pode ser realizada com um comparador de 5 *bits* em vez de 6, pois o bit 5 de ambos os operandos tem o mesmo valor. Isto permite reduzir a área e o atraso do circuito gerado através da análise dos *bits* dos dados envolvidos na operação. Isto permite o uso de um menor número de portas lógicas, reduzindo a área e o atraso geral.
5. **Aferição do Número de *Bits* em Regiões Cíclicas:** consiste em deduzir a quantidade de *bits* necessários para se representar somas ou multiplicações existentes dentro de um ciclo, no caso em que as somas ou multiplicações são

acumulativas. Como as regiões cíclicas de um programa são normalmente as partes de computação mais intensivas, é nestas regiões que pequenas reduções no tamanho de *bits* de uma operação podem implicar em ganhos elevados no nível de atraso total do ciclo, pelo fato do operador ser executado várias vezes.

De acordo com Cardoso [Cardoso 2000], *Nenya* foi, até a data de sua publicação, o primeiro compilador para *reconfigware* a implementar a partição temporal. A partição temporal permite que um programa, cujo tamanho do circuito gerado exceda a capacidade do FPGA, seja completamente executado. Em tal proposta, são obtidas frações do circuito (chamadas de frações temporais) em que suas execuções sejam possíveis de serem feitas individualmente no dispositivo (uma de cada vez), sendo preservada a funcionalidade global do circuito. De acordo com Cardoso, os incentivos para tal abordagem são:

- FPGAs com menores tempos de configuração.
- FPGAs com suporte de armazenamento de várias configurações, que permitem a mutação de contextos em alguns nano-segundos, com armazenamento na inicialização do dispositivo ou paralelo a execução de outra configuração.
- Muita pesquisa já feita relativa à comunicação eficiente entre frações temporais.
- Comutação entre esquemas de codificação e decodificação em sistemas de comunicação, vídeo ou áudio.

A comunicação entre as partições geradas pode ser feita por três formas:

1. Uso de registros e de uma tarefa em execução no microprocessador para controlar a escrita/leitura de dados entre frações temporais. Tal abordagem necessita de um microprocessador ou micro-controlador que pode gerir as

reconfigurações e comunicações.

2. Uso de um conjunto de registros na unidade de processamento reconfigurável quando o dispositivo pode ser reconfigurado parcialmente.
3. O uso de memórias acopladas ao FPGA para guardar os dados oriundos da execução entre frações temporais. Esta forma tem a vantagem de permitir transferência de grandes quantidades de dados.

Os compiladores *Galadriel* e *Nenya* foram os primeiros compiladores a trabalhar a partir dos *bytecodes* do JAVA para geração de *reconfigware*. Além disto, foi proposto um novo algoritmo de partição temporal. Tal algoritmo foi confrontado com uma versão implementada baseada no *simulated annealing* para confrontação com resultados ótimos obtidos pelo mesmo. Os resultados obtidos pelo *simulated annealing* foram em média superados pelo novo algoritmo de partição temporal proposto por Cardoso. Foi a primeira vez que o *simulated annealing* foi utilizado para resolver o problema da partição temporal e, baseado nos resultados, não se mostrou uma alternativa razoável quando o objetivo é velocidade de compilação.

Foram criados também algoritmos originais que permitem determinar estaticamente, para cada operando, o número de *bits* suficientes para satisfazer a funcionalidade inicial, um algoritmo que realiza a propagação de constantes ao nível do bit, um algoritmo de escalonamento baseado em regiões que permite efetuar o escalonamento de estruturas de representação hierárquicas, a implementação de ciclos em *hardware* específico e o controle das seqüências de acesso à mesma memória RAM.

3.3.3 *Spark*

Assim como *Galadriel* e *Nenya*, *Spark* é um projeto recente relativo à síntese de alto

nível que visa obter resultados competitivos com os projetos de circuitos feitos manualmente. *Spark* permite otimizações em nível de operação e em nível de linguagem visando plataformas reconfiguráveis de granularidade grossa. *Spark* explora a aplicação de otimizações ao nível de linguagem, re-estruturando o código original para conseguir o máximo de paralelismo possível em programas com intenso fluxo de controle. A plataforma permite ainda ao usuário o total controle sobre as transformações aplicadas. A figura 3.8 ilustra o fluxograma do *Spark*.

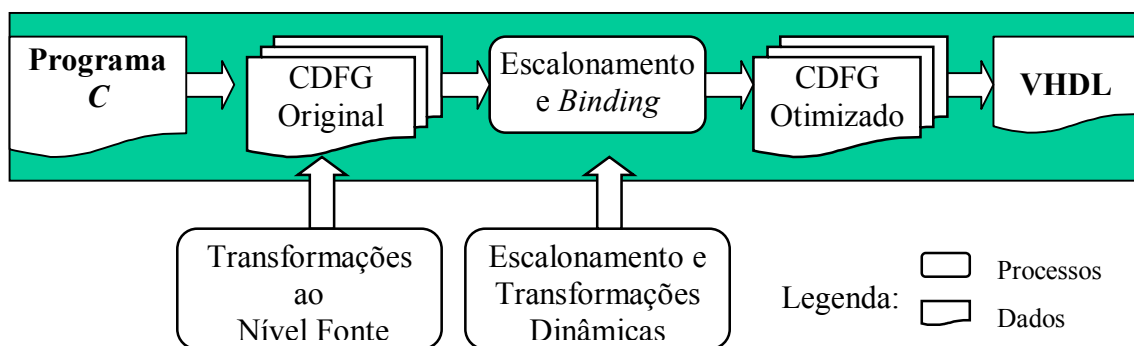


Figura 3.8 – Fluxograma do *Spark*

Como mostra a figura, as transformações de paralelização e otimização são aplicadas ao nível fonte da descrição e durante a fase de escalonamento. As transformações de paralelização e otimização foram organizadas em quatro grupos [*Spark* 2004]:

1. **Otimizações de pré-síntese:** consiste na aplicação de movimentação de código ciclo-invariante, desenrolamento de ciclos e eliminação de sub-expressões comuns.
2. **Otimizações de escalonamento:** consistem na aplicação de movimentação de código especulativa, *multi-cycling*, encadeamento de operações e *pipelining* de ciclos.

3. **Otimizações dinâmicas:** consistem em transformações aplicadas dinamicamente durante o escalonamento. Tais transformações são: eliminação dinâmica de sub-expressões comuns, propagação dinâmica de cópias, propagação dinâmica de ramos e balanceamento dinâmico de ramos.
4. **Transformações básicas de compiladores:** consistem em transformações como propagação de cópias e eliminação de código morto.

A representação do programa fonte, assim como no *Galadriel* e *Nenya*, foi feita através de grafos de fluxo. No caso do *Spark*, foi escolhido o HTG como representação de topo, pois:

1. O HTG retém informações importantes sobre a estrutura do código tais como, fluxo de controle, hierarquia de controle e construções de ciclos, características de ciclos, entre outros.
2. A manutenção destas informações torna possível fazer otimizações no código em múltiplos níveis, desde transformações de pré-síntese (tais como desenrolamento de ciclos e fusão de ciclos) até transformações de compilação tradicionais de muito baixo nível como, por exemplo, eliminação de código morto.

Além disto, a escolha do HTG como representação do código fonte ajudou na implementação e análise de transformações de paralelização, como *Trailblazing* [Nicolau e Novac 1993] e *pipelining* de ciclo direcionado ao recurso [Novac e Nicolau 1996].

Adicionalmente ao HTG, é utilizado o grafo de fluxo de controle-dados (CDFG) como representação intermediária do programa. O CDFG mantém ambas as informações de fluxo de controle e de fluxo de dados. Apesar de manter todas estas informações, este grafo não mantém informações relativas às hierarquias do código fonte, não sendo assim suficiente para

aplicação de transformações ao nível de linguagem. O CDFG era a representação de topo mais utilizada em projetos mais antigos relativos a síntese de alto nível, sendo a principal causa da não obtenção de bons resultados comparados a projetos manuais [Cardoso 2000; *Spark* 2004].

A ferramenta *Spark* permite total controle sobre as otimizações a serem executadas sobre o programa fonte. A ferramenta possui opções para habilitação e desabilitação da aplicação dos vários algoritmos propostos para fins de testes e comparações. O sistema funciona com base em *scripts* de fácil criação e opções que podem ser especificadas em linha de comando.

Os resultados mostram que a aplicação das otimizações implementadas permite um ganho de desempenho em torno de 80%, com um modesto incremento de 5% da área total do circuito. Desta forma, as otimizações desenvolvidas foram capazes de melhorias de alta performance com mínima ou nenhuma degradação da qualidade do circuito (área). Foi percebido também que dois fatores conflitantes emergem quando uma transformação de otimização é aplicada. À medida que a utilização de recursos aumenta, a quantidade de lógicas de navegação (multiplexadores e de-multiplexadores) conectadas às unidades funcionais e o tamanho da lógica de controle associada também aumentam. Do outro lado, à medida que o número de estados do controlador diminui, o tamanho e complexidade do mesmo diminuem. Dependendo da efetividade da transformação em um determinado projeto, um destes fatores pode sobressair-se ao outro.

3.3.4 *Impulse C*

Impulse C [Impulse 2006] é uma ferramenta comercial para síntese de alto nível que permite gerar SOPCs para vários tipos de FPGAs de diversos fabricantes. Consiste de uma biblioteca de extensões à linguagem C (na forma de novos tipos de dados e chamadas de

função pré-definidas) que permitem vários segmentos de programas paralelos serem descritos, interconectados e sincronizados. O compilador *Impulse C* traduz e otimiza programas em representações de baixo nível apropriadas, incluindo descrições lógicas em VHDL-RTL. CoDeveloperTM da *Impulse Corporation* é um ambiente de desenvolvimento que integra a biblioteca *Impulse C* permitindo a criação de projetos para plataformas específicas, provendo ferramentas para medição de performance (*profiling*) e co-simulação com outros ambientes. A ferramenta é projetada para aplicações orientadas a fluxo de dados, mas é também flexível o suficiente para suportar modelos de programação alternativos, incluindo uso de memórias compartilhadas como um mecanismo de comunicação. O modelo de programação do *Impulse C* é baseado no modelo de comunicação sequencial de processos descrito por C. A. R. Hoare e é voltado para aplicações mistas *hardware* e *software* orientadas a *streams*. No *Impulse C*, os elementos (ou processos) *software* e *hardware* das aplicações se comunicam primariamente através de *streams* de dados *bufferizados* que são implementados diretamente em *hardware*.

Esta *bufferização* de dados, que é implementada utilizando-se de filas configuradas pelo programador, torna possível escrever aplicações paralelas em um nível de abstração relativamente alto, sem a sincronização a partir de ciclos de relógio que seria de outro modo necessária. Os processos são independentemente sincronizados e com isto operam os segmentos da aplicação concorrentemente. Os dados que são processados pela aplicação fluem de processo para processo através de *streams*, ou em alguns casos através de mensagens e/ou memórias compartilhadas. A figura 3.9 ilustra o modelo de programação do *Impulse C*.

Como os outros compiladores de síntese de alto nível que partem da linguagem C como fonte de descrição do programa, a ferramenta de compilação do *Impulse C* não suporta todos os tipos de instruções e construções da linguagem. Tais limitações são as seguintes:

1. Chamadas de função dentro do código a ser tratado.
2. Chamadas de função recursivas.

3. Ponteiros (devem ser resolvidos em tempo de compilação para referências de localização estáticas de memória)
4. Suporte a *structs* e *unions*.
5. Suporte a variáveis globais.

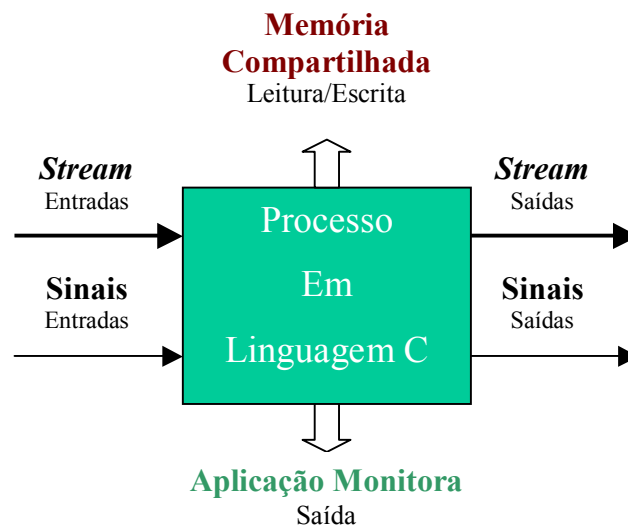


Figura 3.9 – Modelo de programação do *Impulse C*

3.4 Conclusão

A síntese de alto nível evoluiu animadoramente nos últimos seis anos. Neste período surgiram ferramentas poderosas capazes de produzir circuitos com performance compatível com os obtidos em projetos criados por engenheiros especializados.

A representação do programa fonte através de grafos que permitem armazenar as hierarquias da estrutura do código (HPDG e HTG) mostra-se, através dos resultados obtidos pelos compiladores apresentados, eficiente em tempo e espaço.

Embora existam outras abordagens para linguagens de alto nível para descrição do programa a ser sintetizado, como por exemplo, JAVA, a utilização da linguagem C como

fonte de descrição para o programa a ser transformado em circuito específico é predominante, mesmo nas ferramentas mais recentes. Mas em todos os projetos já implementados pela comunidade de síntese de alto nível existe a falta de suporte total aos recursos da linguagem utilizada como fonte de descrição, qualquer que seja esta linguagem. Algumas limitações ocorrem devido às limitações do próprio *hardware* em se implementar determinados tipos de construções das linguagens de alto nível, principalmente construções de desvio condicionais e recursão de funções. Outras decorrem de uma certa falta de equivalência semântica entre o recurso da linguagem e o mesmo recurso implantado em *hardware*. Como exemplo, em um programa um ponteiro muitas vezes pode “apontar” para o endereço de uma variável que, ao final do processo de síntese de alto nível, acabou sendo atribuída a um registrador. Como vemos, não existe sentido em uma variável do tipo ponteiro (esteja esta atribuída a um registro) “apontar” para uma que reside em um registrador em um circuito, pois teríamos um “apontador para um registrador”. Outras limitações de linguagem talvez decorram do alto tempo gasto na implementação de todo o *software* necessário para a validação de novas propostas, sendo que o suporte total à linguagem acaba sendo deixado para trabalhos futuros.

Cardoso aponta ainda que a criação de ferramentas de arquitetura aberta que permitam fácil integração de implementações de propostas futuras relativas à síntese de alto nível de circuitos digitais mostra-se uma contribuição interessante para a área.

Embora a motivação principal da criação do *Phoenix* seja sua utilização no projeto ARCHITECT+, a criação e disponibilização do *framework* como uma arquitetura de compilação aberta supre esta necessidade apontada por Cardoso no que tange a compilação a partir da linguagem C. O *framework* contribui também no aspecto da falta de suporte total aos recursos da linguagem C por parte dos compiladores anteriormente criados, provendo um mecanismo de suporte automático às estruturas de dados do tipo *struct* da linguagem C..

Capítulo 4

O Framework Phoenix

4.1 Introdução

Este capítulo tem como objetivo descrever a arquitetura do *framework Phoenix*. O *framework* compõe o projeto ARCHITECT+, em desenvolvimento no Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo – Campus São Carlos. A principal motivação do projeto do ARCHITECT+ é o desenvolvimento rápido e flexível de aplicações para dispositivos reconfiguráveis. A utilização de um modelo de computação reconfigurável permite o desenvolvimento de aplicações através de um modelo de Co-Projeto *Hardware/Software*. Este modelo permite flexibilidade para a aplicação, pois esta pode ser focada no *hardware* ou no *software*, dependendo das necessidades do projeto. Estas vantagens tornam simples o difícil processo de desenvolvimento de sistemas de robótica móvel, processo este que envolve a construção de algoritmos complexos e o projeto tanto de *hardware* quanto de *software*.

O desenvolvimento do *Phoenix* objetiva a tradução de programas C para uma representação intermediária adequada tanto para o processo de geração de código nativo quanto para o processo de síntese de alto nível de circuitos digitais. Objetiva-se também o desenvolvimento da arquitetura do compilador de um modo que permita uma fácil expansão de seus recursos para pesquisas futuras na área. No âmbito deste trabalho, objetiva-se ainda criar o processo de geração de código nativo para o processador Nios IITM da Altera Corporation. Um suporte automático para estruturas de dados do tipo *struct* também será implementado, representando uma contribuição para a área.

Este capítulo contém uma breve descrição do projeto do ARCHITECT+. É

apresentado o fluxograma geral do sistema, sendo apresentadas também cada uma de suas partes. Posteriormente é apresentado o processador Nios IITM com uma descrição de suas características mais importantes. É feita uma descrição detalhada dos aspectos mais importantes da arquitetura do compilador *Phoenix*.

4.2 ARCHITECT+

O ARCHITECT+ é uma ferramenta para a geração de *hardware* e *software* para robôs móveis que integra computação de alto desempenho e algoritmos de aprendizado de máquina. O objetivo da ferramenta é fornecer um ambiente para co-projeto de *hardware/software* em plataformas de FPGAs com aplicação em robótica móvel. A partir de um algoritmo de aplicação em robótica móvel escrito em linguagem C, o sistema gerará automaticamente um SOPC. A motivação para o desenvolvimento deste projeto é que os FPGAs permitem a implementação de circuitos que executam programas com maior velocidade que em *software* em processadores de propósito geral enquanto possibilitam uma maior flexibilidade que os ASICs, permitindo desta forma, melhor performance e adaptação aos sistemas computacionais atuais. Tais vantagens tornam mais simples a construção de sistemas de robótica móvel, objetivo principal da ferramenta. A figura 4.1 ilustra o diagrama de blocos do sistema.

Para a especificação dos programas, o sistema conta com uma biblioteca de unidades reconfiguráveis de processamento (URPs) em VHDL e a respectiva descrição de cada URP em linguagem C. Estas URPs implementam vários algoritmos utilizados em robótica, incluindo funções de processamento de sinais, algoritmos probabilísticos, redes neurais e algoritmos genéticos. O sistema funciona da seguinte forma:

- Dada uma aplicação, seu código é analisado pelo sistema para a detecção dos trechos que consomem o maior tempo de execução. O sistema conta com

algoritmos (estimadores) responsáveis por detectá-los automaticamente.

- O resultado desta análise é mostrado ao usuário que decide se o resultado é considerado satisfatório. Se não é, o usuário pode requerer uma nova análise do código, podendo ele passar ao programa alguns novos aspectos (características) a se considerar na nova análise.
- Este processo se repete até a satisfação do usuário. Neste caso, é gerado o *hardware* (em HDL) específico de cada trecho e o *software* para o resto da aplicação.

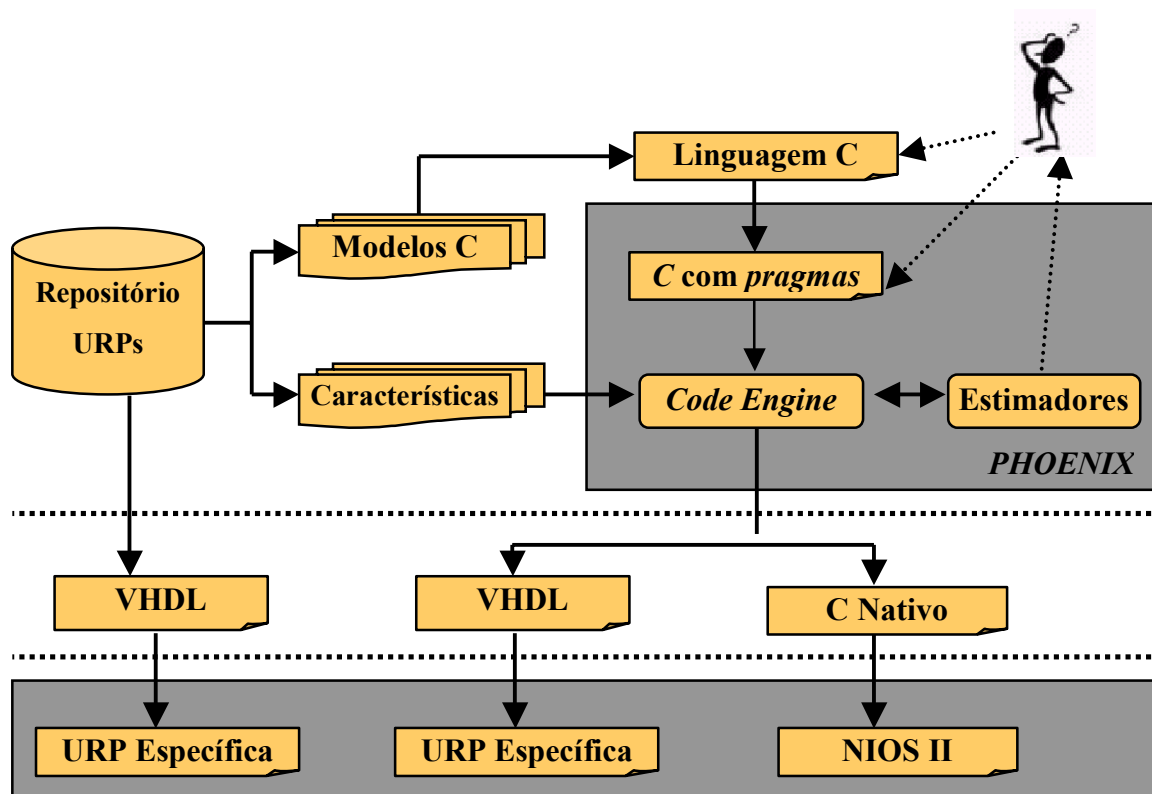


Figura 4.1 – Diagrama de blocos do ARCHITECT+.

Em algumas situações, as URPs podem ser mapeadas completamente para o FPGA, não necessitando passar pelo processo de análise pelos estimadores. Considerando-se os meios de suporte à computação reconfigurável, o *reconfigure* gerado funcionará como um

núcleo de co-processamento. A figura 4.2 ilustra esta organização dentro do FPGA.

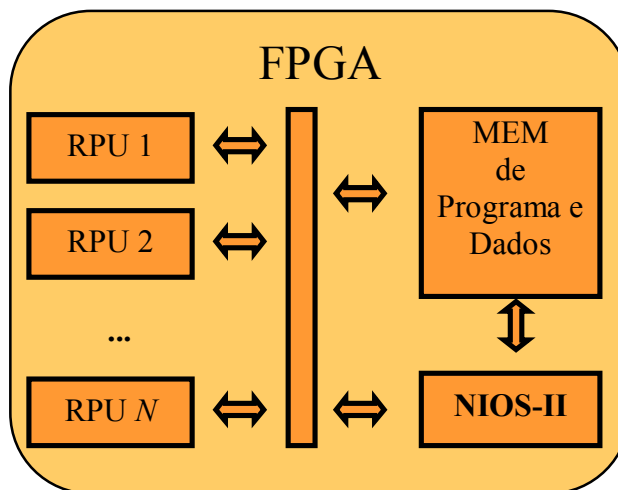


Figura 4.2 – Organização do SOPC gerado pelo ARCHITECT+ no FPGA.

4.3 Nios II

Nios II [Altera 2005a] é um processador RISC de propósito geral virtual. A principal característica deste processador é ser um processador *soft-core*, o que significa que ele é provido na forma de *software* e não como uma placa de silício. Nios II é fornecido em código HDL que pode ser sintetizado através de ferramentas do fabricante para qualquer família de FPGAs da Altera. Isto permite flexibilidade, pois o projetista pode configurá-lo de acordo com suas necessidades e até mesmo estender suas funcionalidades. O projetista pode também criar seus próprios periféricos customizados e integrá-los aos sistemas do Nios II. Outras características do Nios II são:

- Conjunto de instruções, caminho de dados e espaço de endereçamento de 32 *bits*.
- 32 registradores de propósito geral.
- 32 fontes de interrupção externa.

- Instruções simples para multiplicação e divisão de 32 *bits* produzindo resultado de 32 *bits*.
- Instruções dedicadas para multiplicação de 64 e 128 *bits*.
- Acesso a vários periféricos em *chip* e interfaces para acesso a memórias e interfaces fora do *chip*.
- Módulo de depuração assistido por *hardware* que permite depuração do processador por uma aplicação de desenvolvimento integrada.
- Arquitetura do conjunto de instruções compatível com todas as versões disponíveis do Nios II.
- Alta performance executando sob FPGA.

A figura 4.3 ilustra um sistema do processador Nios II.

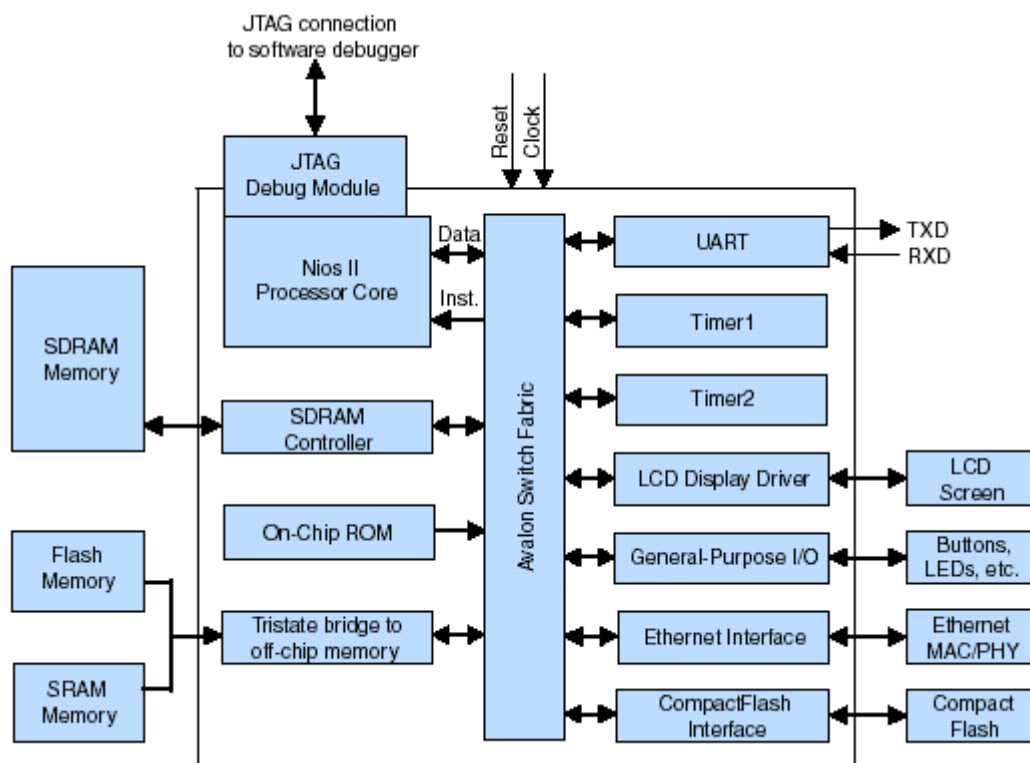


Figura 4.3 – Exemplo de um sistema do processador Nios II.

4.3.1 Conjunto de Instruções

O processador Nios II possui 94 instruções de propósito geral divididas em 7 categorias:

- **Instruções de transferência de dados:** são as instruções de transferência de dados entre memória e registradores (*load/store*). Existem instruções específicas para lidar com dados de 8, 16 e 32 *bits* com e sem sinal.
- **Operações aritméticas e lógicas:** implementam operações como adição, subtração, multiplicação, divisão e *and*, *or*, *xor* e *nor*.
- **Instruções de movimentação:** provêm movimentação de dados entre registradores.
- **Instruções de comparação:** implementam todas as operações relacionais e de equivalência.
- **Instruções de deslocamento e rotação:** implementam deslocamento e rotação de *bits*.
- **Instruções de controle:** implementam instruções de desvio incondicional e chamada de sub-rotinas, desvio de execução em caso de exceções, além de instruções de manipulação de informações da CPU. Implementam ainda instruções de desvio condicional que comparam diretamente valores entre registradores e desviam para o endereço provido se a condição for verdadeira. Estas instruções também permitem as comparações de equivalência e relação.
- **Instrução sem-operação:** implementa a instrução sem-operação (*nop*).

O conjunto de instruções do Nios II está dividido em 3 formatos de palavra mostrados

na figura 4.4:

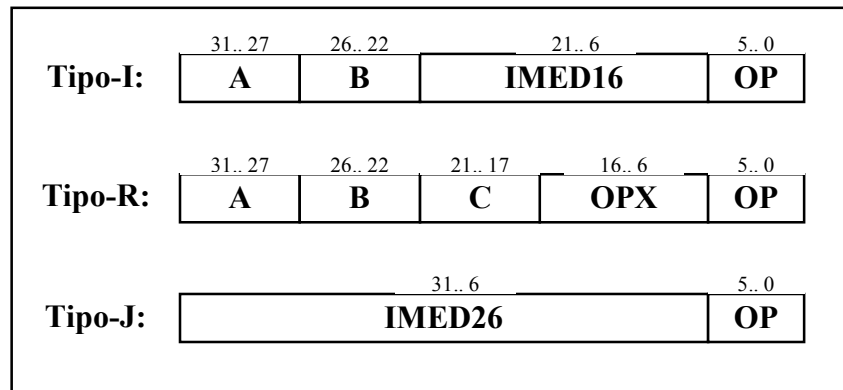


Figura 4.4 – Formatos de instruções do Nios II.

- **Tipo-I:** este formato permite que um valor de 16 *bits* (IMED16) seja embutido na instrução. Este tipo de instrução permite ainda a especificação de dois registradores (5 *bits* cada) e o código da instrução (6 *bits*). Em muitos casos, os campos *A* e IMED16 especificam os operandos fonte e o campo *B* especifica o registrador de destino do resultado da operação. Este formato inclui algumas das instruções aritméticas e lógicas, desvio condicional, transferência de dados e operações de gerenciamento de *cache*.
- **Tipo-R:** neste formato, todos os argumentos e resultados são especificados como registradores. A instrução possui três campos para registradores (5 *bits* cada), o código da instrução (6 *bits*) e um campo de código extra (OPX) de 11 *bits*. Este formato inclui algumas as instruções aritméticas e lógicas, instruções de comparação, a instrução customizada, e outras operações que envolvem somente registradores.
- **Tipo-J:** este formato inclui o código da instrução (6 *bits*) e um valor imediato de

26 *bits* (IMED26). A única instrução tipo-J é a instrução *call* (chamada de sub-rotina).

Além das instruções de propósito geral, o Nios II oferece suporte às instruções customizadas [Altera 2005b]. As instruções customizadas são instruções com lógica definível pelo programador. Este pode integrar um circuito lógico ao processador e associá-lo a uma instrução customizada específica. Da mesma forma que as outras instruções do processador, as instruções customizadas possuem dois registradores de entrada e um de saída. O formato da instrução customizada permite criar até 256 instruções diferentes. A figura 4.5 ilustra o formato deste tipo de instrução.

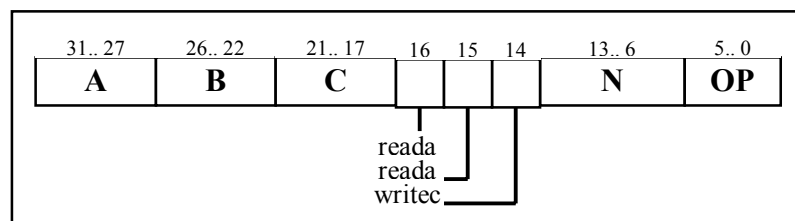


Figura 4.5 – Formato da instrução customizada do Nios II.

Na figura 4.5, *A* e *B* são os registradores de entrada e *C* é o registrador de saída. Estes registradores podem ser tanto os registradores internos do módulo de suporte às instruções customizadas quanto os registradores de propósito geral do Nios II. *N* é o número da instrução customizada a executar. Os *bits* 16 (*readra*), 15 (*readrb*) e 14 (*writerc*) especificam se os registradores a serem utilizados serão os registradores internos ou se serão os registradores de propósito geral do Nios II.

4.4 *Phoenix*

Phoenix é um compilador que tem como entrada a linguagem C, padrão *ANSI*, que cria uma representação intermediária para geração de código para processadores e para o processo

de síntese de alto nível de circuitos digitais. Os objetivos principais no desenvolvimento deste compilador são:

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

A figura 4.6 ilustra o fluxograma geral do *Phoenix*.

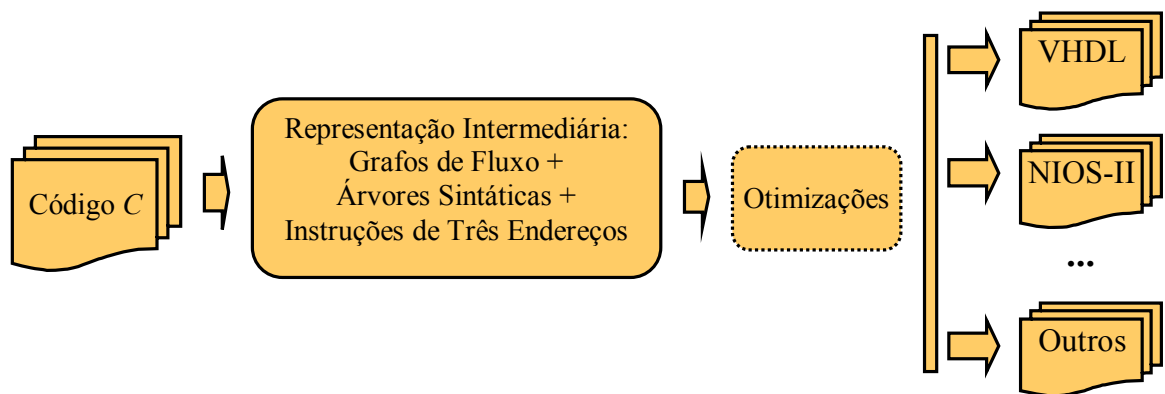


Figura 4.6 – Fluxograma do *Phoenix*.

A representação intermediária gerada pelo compilador consiste de uma série de grafos. Estes encapsulam os fluxos de controle do programa original, suas instruções (representadas em um outro formato intermediário) e as dependências de controle e de dados. Através desta mesma representação intermediária pode ser gerado código para processadores de arquitetura CISC ou RISC. O compilador permite esta geração de código para o processador Nios II. A geração de código para outros processadores pode ser implementada com a descrição das propriedades da máquina em uma linguagem própria especialmente projetada para o compilador.

Embora indicada na figura 4.6, a fase de otimizações dos grafos gerados foi deixada para implementações futuras. O compilador opera em quatro fases:

- **Fase 1:** nela é feita a análise do programa fonte. São gerados os grafos de fluxo de controle e de hierarquia de tarefas para cada função. São geradas também as árvores sintáticas das expressões e sua representação na forma de instruções de três endereços. Durante a análise sintática é também feita a análise semântica das expressões da linguagem. Durante esta fase são relatados os erros sintáticos e semânticos para o usuário.
- **Fase 2:** após a geração dos grafos de fluxo de controle e hierarquia de tarefas, das árvores sintáticas e das instruções de três endereços, é feita a análise das dependências de dados das operações. São gerados os grafos auxiliares ao processo de geração de código e síntese de alto nível. Entre estes grafos destacam-se o grafo de hierarquia de tarefas (HTG), o grafo de fluxo de controle (CFG), o grafo de dependência de controle (CDG), o grafo de fluxo de dados (DFG), o grafo de dependência de dados (DDG) e os grafos de dominância e pós-dominância. É feita também a análise das variáveis vivas [Aho *et al.* 1986] para a geração de código.
- **Fase 3:** É relativa à otimizações e foi deixada para trabalhos futuros. Nesta fase serão feitas as otimizações exigidas para o processo de geração de código nativo ou síntese de alto nível de circuitos digitais.
- **Fase 4:** nesta fase é gerado o código para o alvo específico. No caso da geração de código, é feita a alocação de registradores para as operações do programa, sendo estas depois mapeadas para as devidas instruções de máquina. Faz parte deste processo a geração do prólogo e epílogo para cada função do programa. Se

requisitado pelo usuário, pode ser ainda gerado o código *assembly* da respectiva arquitetura e a visualização gráfica de todos os grafos criados, através do *software Graphviz* [Graphviz 2005].

O compilador *Phoenix* foi desenvolvido em linguagem C++, em plataforma Windows™ com a ferramenta Microsoft Visual C++ versão 6.0. O código criado é independente das bibliotecas básicas da linguagem. O código é orientado a objetos (com exceção de poucas funções utilizadas no esquema de tradução do analisador sintático) permitindo alto grau de extensibilidade. As estruturas de dados utilizadas como base de implementação do compilador foram criadas como classes C++ parametrizadas, recurso que permite a utilização de tais classes com qualquer outro tipo de objeto.

As seções seguintes descrevem cada um dos componentes e processos do compilador.

4.5 Componentes

4.5.1 Tabela de Símbolos

A tabela de símbolos é a estrutura de dados que armazena os identificadores encontrados no programa. Estes identificadores podem ser nomes de variáveis, tipos ou funções, entre outros. A tabela de símbolos do *Phoenix* foi implementada como duas tabelas *hash*, uma para os símbolos globais e outra para os símbolos locais. Para cada tipo de símbolo armazenado nestas tabelas foi implementada uma classe com atributos específicos. No caso da linguagem C, estes símbolos podem ser declarações de funções e variáveis, declaração de tipos estruturados (*structs* e *unions*), declaração de tipos, declaração de tipos enumerados e declaração de parâmetros de uma função.

As declarações de funções foram encapsuladas na classe *CDDeclFuncao*. Nesta classe estão definidos possíveis atributos de uma função (*extern*, *static*, *const*) e também seus

parâmetros (um *array* de objetos *CItemTabSimb*). As declarações de variáveis foram encapsuladas na classe *CDeclVariavel*. Nesta classe estão definidas as informações de dependências de fluxo de dados para a variável como usos e definições. As declarações de tipos estruturados foram encapsuladas na classe *CDeclTipoEstruturado*, sendo nela definidas as informações sobre seus membros (campos), e seu tipo (*struct* ou *union*). A estrutura de dados de um tipo estruturado forma uma árvore. Neste caso, um tipo estruturado possui membros. Cada um destes membros possuirá membros se este também for um tipo estruturado, e assim sucessivamente. Declarações de novos tipos são encapsuladas na classe *CDeclTipo*. Nela é definido um ponteiro para o tipo do qual o novo tipo deriva. As declarações de tipos enumerados são encapsuladas na classe *CDeclEnumerado* que provê uma lista de seus itens numéricos. Um item da tabela de símbolos, representado pela classe *CItemTabSimb*, encapsula qualquer um destes possíveis tipos de símbolos (suas classes). A classe *CHashTB* implementa uma tabela *hash* para itens da tabela de símbolos como uma especialização da classe base parametrizada *CHash*, que implementa *hashing* aberto (sem limite no número de entradas). A classe *CTabelaSimbolos* encapsula duas tabelas provendo funções para inserção, busca de acordo com escopo, e liberação de símbolos. Estas funções encapsulam as respectivas funções da classe *CHashTB*, sendo portanto de mais alto nível. Uma ilustração completa do diagrama de classes da tabela de símbolos do *Phoenix* pode ser encontrada no apêndice 1.

A função de *hash* implementada foi a função de *hash* proposta por P. J. Weinberger quando da construção de seu compilador C. Em um conjunto de experimentos, Weinberger observou que sua função funcionou consistentemente para vários tamanhos de tabelas. Abaixo segue o código C da função *hash*, aqui implementada para 211 entradas.

A função começa com a variável *H* inicializada com o valor 0. Para cada caractere do símbolo a ser inserido na tabela, desloca-se os *bits* de *H* quatro posições à esquerda. Se

qualquer um dos quatro *bits* mais significativos for 1, os mesmos são deslocados 24 posições para a direita, fazendo-se logo após o ou exclusivo dos mesmos com H e zerando-se em seguida os 4 *bits* mais significativos.

```
#define PRIMO 211
int Função_Hash ( char * Símbolo )
{
    char *P;
    unsigned H = 0, G;

    H = 0;
    for ( P = Símbolo; *P != '\0'; P++ ) {
        H = (H << 4) + (*P);
        G = H & 0xF0000000;
        if ( G != 0 ) {
            H = H ^ (G >> 24);
            H = H ^ G;
        }
    }
    return H % PRIMO;
}
```

Figura 4.7 – Função de *hash* utilizada no *Phoenix*.

A escolha da representação da tabela de símbolos como uma tabela de *hash* aberto resulta nas seguintes propriedades:

- Permite um número ilimitado de entradas, implicando em maior flexibilidade de alocação;
- Permite realizar busca sobre e entradas da tabela e n nomes num tempo proporcional a $n(n+e)/m$ para qualquer constante m , onde m é o número de entradas;
- O tempo de acesso a uma entrada da tabela é praticamente constante;
- O espaço ocupado pela tabela depende somente de m e o espaço ocupado pelas entradas da tabela depende somente de e ;

A tabela de símbolos utiliza-se de duas classes parametrizadas que provêm uma

estrutura de dados de *array* e de lista respectivamente. Estas classes, chamadas de *CArray* e *CLista* respectivamente, são utilizadas também em vários outros componentes do sistema.

4.5.2 Árvores Sintáticas

As árvores sintáticas constituem a representação intermediária para uma sentença de uma linguagem. As árvores sintáticas podem representar totalmente um programa incluindo, além das instruções, as outras estruturas permitidas pela linguagem, como desvios condicionais e ciclos.

Optou-se por uma representação intermediária através de árvores sintáticas somente para as expressões (instruções) da linguagem. A justificativa é que uma representação abrangendo as estruturas da linguagem como desvios condicionais e ciclos é mantida através de grafos de fluxo, não sendo desta forma necessário representá-las em árvores sintáticas. Esta representação do programa através de grafos é necessária para a aplicação dos algoritmos de otimização de código mais comuns.

Embora as instruções de três endereços sejam geradas no passo seguinte à geração e verificação das árvores sintáticas, a manutenção das árvores sintáticas permite a manutenção da representação sintática original do programa. Desta forma a descrição do programa original pode ser re-obtida se houver necessidade.

Para cada elemento passível de pertencer a uma expressão da linguagem C foi criada uma classe para encapsulá-lo. A classe *CNoArvoreSintatica* encapsula qualquer um dos elementos possíveis de pertencer a uma árvore sintática. Estes elementos podem ser uma chamada de função (invocada pelo operador “()”), um identificador (nome de variável), uma constante numérica ou do tipo *string*, um tipo C (classe *CConstante_Tipo*), uma operação unária, um *typecast*, e operações binárias e ternárias. Embora um *typecast* seja uma operação unária, por fins de implementação resolveu-se especializá-lo a partir da classe

COperacao_Unaria. Uma ilustração completa do diagrama de classes das árvores sintáticas do *Phoenix* encontra-se descrita no apêndice 2.

Uma árvore sintática consiste de nós dinâmicos do tipo *CNoArvoreSintatica*. As árvores sintáticas do *Phoenix* são construídas diretamente no esquema de tradução através do uso de atributos sintetizados permitidos pela análise sintática *bottom-up*.

4.5.3 Instruções de Três Endereços

As instruções de três endereços são uma representação intermediária para as instruções do programa. Nela, as instruções e declarações da linguagem são representadas em operações menores com no máximo três operandos. A representação segue o formato genérico $x := y \text{ op } z$, onde x , y e z são nomes, constantes ou objetos de dados temporários criados pelo compilador, e op é uma operação. Tal representação é útil para o processo de otimização de código, pois representa a operação da linguagem em um formato simplificado, e também para o processo de geração de código, pois representa a operação em um nível de instrução de máquina.

As instruções de três endereços foram a forma escolhida para a representação das instruções do programa para fins de otimização de código. Elas são geradas a partir das árvores sintáticas, logo após a verificação de tipo sobre as mesmas. O processo consiste em varrer a árvore em ordem pós-fixa, gerando-se as respectivas instruções de três endereços que encapsulam o resultado das sub-árvores. A Fig. 4.8 ilustra a construção dos códigos de três endereços para a expressão $A = B * C + 10 / D + -E$.

Na figura 4.8, quando é encontrado um nó correspondente a um operador durante a varredura da árvore, uma variável temporária é criada a fim de armazenar o resultado da operação. Como uma instrução de três endereços só pode conter uma operação, a operação unária $-E$ tem que ser contida em um código de três endereços próprio.

Embora a geração das instruções de três endereços possa ser feita diretamente no esquema de tradução utilizando-se de atributos sintetizados, optou-se por esta abordagem de geração a partir das árvores sintáticas para fins de legibilidade e simplificação para trabalhos futuros. Embora também esta abordagem represente uma segunda travessia na árvore sintática, optou-se por esta separação pelos mesmos motivos anteriores.

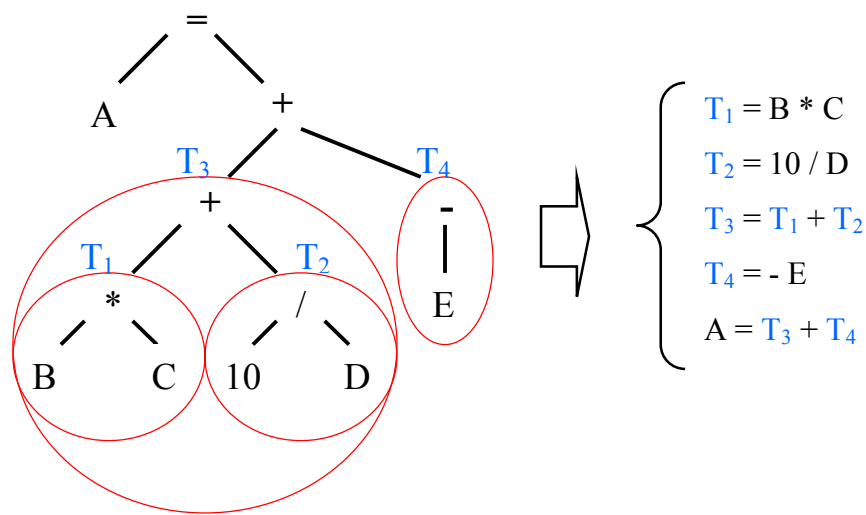


Figura 4.8 – Exemplo de construção de instruções de três endereços a partir da árvore de derivação sintática da expressão $A = B * C + 10 / D + -E$.

4.5.4 Representação de Tipos da Linguagem C

A linguagem C possui um grande poder de expressão de tipos. Como exemplo, considere a seguinte declaração de uma variável em linguagem C:

```
char * (* (* (* (*Var) (char C, int I) ) (long L, float F) ) [10] ) [5]
```

Figura 4.9 – Exemplo de uma declaração de uma variável em linguagem C.

Traduzindo esta expressão temos “*Var é um ponteiro para uma função de argumentos C e I que por si retorna um ponteiro para uma função de argumentos L e F que por sua vez*

retorna um ponteiro para um array (de 10 posições) de ponteiros para arrays (de 5 posições) de ponteiros para objetos de tipo char”.

Embora se possa expressar qualquer tipo possível da linguagem C utilizando-se do poder dos declaradores da mesma, esta notação é pouco utilizada na prática, pois tais declarações para tipos complexos são de difícil interpretação e sua criação exige o conhecimento da gramática dos declaradores da linguagem. A abordagem utilizada nesta situação é a definição da hierarquia de tipos através de declarações utilizando-se da palavra reservada *typedef*.

Embora a criação e a interpretação de tipos como o da figura 4.9 possam ser relativamente complicados, a representação dos mesmos para fins de compilação não o é. Pelo menos este foi o resultado obtido com a representação criada e utilizada no *Phoenix*. A representação de tipos da linguagem C do compilador consiste em uma lista (no formato de *array*) que armazena as informações possíveis de cada componente do tipo. De acordo com a gramática de declaradores C, estes componentes de um tipo da linguagem podem ser um tipo primitivo da linguagem, um tipo construído a partir dos construtores *struct* e *union*, uma função (com seus argumentos), um ponteiro ou um *array*.

Para cada um destes componentes foi criada uma classe que encapsula suas informações. O primeiro componente (um tipo primitivo ou agregado) é implementado através da classe *CNoTipo*. Esta classe provê informações sobre o tipo primitivo ou um ponteiro para o tipo estruturado na tabela de símbolos. Uma declaração de função é encapsulada na classe *CNoFuncao* e esta classe provê um *array* dos parâmetros da mesma. Um ponteiro é encapsulado na classe *CNoPonteiro*. Um *array* por sua vez é encapsulado pela classe *CNoArray* sendo que a classe provê a dimensão do mesmo. A classe *CNoListaTipo* encapsula qualquer objeto de uma das classes descritas anteriormente. Enfim, um tipo da linguagem C no *Phoenix* é uma lista de objetos do tipo *CNoListaTipo*, sendo que cada

elemento desta lista representa um dos componentes possíveis de um tipo C descritos anteriormente representados por objetos das respectivas classes. Como exemplo, a declaração exemplificada na figura 4.9 seria representada pelo *Phoenix* pela seguinte lista:

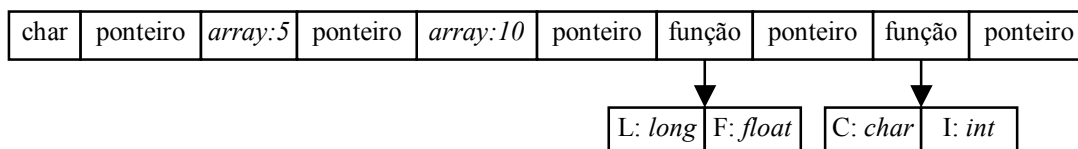


Figura 4.10 – Exemplo da lista que representa a declaração da figura 4.9.

A construção da lista que representa uma declaração em C é feita no esquema de tradução através de um atributo sintetizado específico. Em cada regra da gramática responsável pelo reconhecimento de um componente da declaração, uma função insere o respectivo objeto representativo na lista. Cada novo componente da declaração (objeto da lista) é inserido na lista na posição zero, sendo que desta forma, o último componente reconhecido estará no início da lista. Analisando-se a tradução feita para a declaração apresentada anteriormente (fig. 4.9), perceber-se-á que a ordem dos componentes da lista é inversa à ordem na qual aparecem na tradução. Tal fato decorre da ordem em que os componentes são reconhecidos pela gramática de declaradores C.

A verificação de igualdade entre dois tipos pode ser facilmente feita comparando-se os componentes das duas listas que representam os tipos. Outros tipos de verificações como, por exemplo, se o tipo é um ponteiro, ou se é um ponteiro para uma função, ou se é um *array*, ou se é um *array* de ponteiros, entre outros, podem ser feitas verificando se os respectivos componentes existem na lista e se estão na ordem que representariam tal tipo.

A representação de declarações da linguagem C no *Phoenix* por tal abordagem mostrou-se ser:

- Simples de visualizar.

- Fácil de se implementar no esquema de tradução.
- Fácil de se implementar todas as necessidades relativas à manipulação de tipos.
- Conveniente, visto as facilidades apontadas anteriormente.

4.6 Processos do Compilador *Phoenix*

4.6.1 Análise Léxica e Sintática

A tarefa de traduzir um programa para sua representação intermediária é realizada por dois processos: de análise léxica e sintática do código fonte.

A principal tarefa de um analisador léxico é ler um fluxo de caracteres e agrupá-los em *tokens* para o processo de análise sintática. Ele realiza outras tarefas como eliminação de comentários e espaços em branco do programa fonte e aviso de erros de análise léxica como, por exemplo, caracteres inválidos na entrada (não reconhecidos pelo analisador léxico). Os analisadores léxicos geralmente implementam os padrões de *tokens* através de diagramas de transição de estados. Os diagramas de transição guiam as ações a serem tomadas a cada caractere lido do *buffer* de entrada.

A análise sintática é responsável pela construção e validação da hierarquia produzida de acordo com as regras de sintaxe. A descrição da sintaxe de construção da maioria das linguagens é feita através de gramáticas livre de contexto. As gramáticas oferecem as seguintes vantagens adicionais tanto para os projetistas de linguagens como para os escritores de compiladores:

1. Especificação precisa e fácil de entender.
2. Pode-se construir automaticamente, para certas classes de gramáticas, um analisador sintático, que garantidamente aceita todas as sentenças pertencentes à linguagem e rejeitam toda sentença não pertencente à linguagem.

3. É útil na tradução correta de programas-fonte em código-objeto e também à detecção de erros.
4. Novas construções para as linguagens podem ser facilmente adicionadas às gramáticas.

Os métodos de análise sintática mais comumente usados pelos compiladores são classificados em duas categorias: *top-down* e *bottom-up*. No método *top-down*, os analisadores sintáticos constroem árvores gramaticais do topo (raiz) para o fundo (folhas). Já no *bottom-up*, os analisadores sintáticos constroem árvores gramáticas partindo das folhas para a raiz.

O método de análise sintática escolhido para a construção do *Phoenix* foi o método *bottom-up* LALR [DeRemer 1969 e 1971]. As justificativas para tal escolha são:

- Este método produz tabelas sintáticas menores que os outros métodos *bottom-up*.
- Ele permite expressar a maioria das construções sintáticas comuns nas linguagens de programação.
- O analisador sintático pode ser gerado automaticamente a partir da gramática por ferramentas especializadas, como por exemplo, o YACC [Johnson 1975].
- As vantagens descritas anteriormente relativas à especificação de analisadores sintáticos através de gramáticas.

O analisador léxico utilizado foi construído através de uma variante da ferramenta Lex [Lesk 1975] chamada *Flex*. O analisador sintático utilizado foi construído utilizando-se de uma variante da ferramenta YACC chamada *Bison*. A justificativa principal do uso destes dois compiladores é que são mais estáveis e os analisadores gerados por eles podem ser facilmente integrados. Tanto *Bison* quanto *Flex* usam uma representação simples para a

especificação de suas entradas. Através de opções de linha de comando, permitem gerar em arquivo os respectivos analisadores. Estes arquivos podem ser facilmente incluídos em um projeto de um compilador.

A geração automática de analisadores léxicos e sintáticos permite um ganho de velocidade considerável quanto ao desenvolvimento de um compilador. Em caso de necessidade de modificações dos analisadores, basta repetir o processo de compilação através das ferramentas.

4.6.2 Análise Semântica

A análise semântica garante que um programa fonte a ser compilado seja verificado quanto a erros de tipo semântico antes do processo de geração de código. Uma verificação muito importante nesta fase é a verificação de tipos. Neste processo é verificado se os operandos satisfazem as exigências feitas pelos operadores ao qual estão associados. Ainda nesta fase, atributos relativos aos símbolos tais como tipo, escopo, entre outros, podem ser verificados e posteriormente armazenados na tabela de símbolos.

No *Phoenix*, a análise semântica das expressões da linguagem foi implementada por verificação através da árvore sintática por uma única função recursiva. Neste processo, após o término da geração da árvore sintática, a mesma é varrida em ordem pós-fixa e os tipos resultantes das sub-árvores são retornados para os níveis superiores da árvore e verificados de acordo como os operadores que são aplicados. A função e a descrição dos padrões de verificação de tipos das operações são mostradas na figura 4.11.

```
CTipo Verifica_Expressão( CArvoreSintatica *No, BOOL Global, CItemTabSimb *UltNivel )
{
    CTipo ROP1;    // Tipo resultante da operação da sub-árvore esquerda
    CTipo ROP2;    // Tipo resultante da operação da sub-árvore direita
    CTipo ROP3;    // Tipo resultante extra para a operação ternária

    if ( ! No ) return CTipo( TIPO_NULO );
}
```

```

switch ( No->Tipo )
{
    case UN_AS_VARIAVEL:
        if ( UltNivel == NULL ) {
            /* Busca variável no nível raiz da tabela de símbolos. Se
            a variável não existir, reportar erro. Se existir, retorna
            seu tipo. */
        }
        else
        {
            /* Se entramos aqui quer dizer que estamos acessando
            variável por "." ou "->". UltNivel é igual ao último
            componente (operando direito à "." ou "->" do
            nível anterior) analisado.
            Verificamos se operando do lado direito do "." ou "->" é
            membro da estrutura do lado esquerdo (função
            Busca_Simbolo_Membro() ). Se tudo OK, retorna o tipo
            do componente da direita.*/
            Senão, reporta erro.
        }

    case UN_AS_CONSTANTE:
        /* Retorna seu tipo */

    case UN_AS_CONSTANTE_TIPO:
        /* Retorna seu tipo */

    case UN_AS_CHAMADA_FUNCAO:
        /* Verifica se tipos dos argumentos correspondem aos tipos
        dos parâmetros da função. Retorna o tipo retornado pela função. */

    case UN_AS_TYPE_CAST:
        /* Verifica se conversão é possível. Se sim, retorna o novo tipo. */

    case UN_AS_OPERACAO_UNARIA:
        ROP1 = Verifica_Expressao( No->Operacao, Global, NULL );

        /* Verifica se a operação unária pode ser aplicada sobre o tipo ROP1.
        Se não, reportar erro.
        Se sim, retorna o tipo resultante da operação.*/

    case UN_AS_OPERACAO_BINARIA:
        ROP1 = Verifica_Expressao( No->OpEsq, Global, NULL );
        /* Se os operadores forem “.” ou “->”, passar para a sub-árvore direita
        o ponteiro para o item da tabela de símbolos que representa o tipo
        da sub-árvore (sub-expressão) esquerda. */
        ROP2 = Verifica_Expressao( No->OpDir, Global, Ptr_Simb );
        /* Se não, passar NULL. */
        ROP2 = Verifica_Expressao( No->OpDir, Global, NULL );
        /* Verifica se a operação binária pode ser aplicada sobre os tipos ROP1
        e ROP2. Verifica por typecasts implícitos.
        Se não, reportar erro.
        Se sim, retorna o tipo resultante da operação (de acordo com o typecast
        se for o caso).*/

```

```

case UN_AS_OPERACAO_TERNARIA:
    ROP1 = Verifica_Expressao( No->Condicao, Global, NULL );
    ROP2 = Verifica_Expressao( No->OpEsq, Global, NULL );
    ROP3 = Verifica_Expressao( No->OpDir, Global, NULL );
    /* Verifica se ROP1 é do tipo booleano. Verifica se ROP2 e ROP3
    possuem tipos iguais.
    Se não, reportar erro.
    Se sim, retorna o tipo resultante da operação (pode ser um valor-L
    ou valor-R) */
    }
}

```

Figura 4.11 – Função de verificação de tipos.

Como se pode notar no pseudocódigo da função, além fazer a verificação de tipos, a mesma função faz a verificação de acesso a componentes de *structs* e *unions*. No caso de uma expressão como $A.B=0$, a função verifica se B é realmente um membro do tipo estruturado instanciado por A . Tal recurso é implementado nos seguintes passos:

- A sub-árvore esquerda da operação binária “.” e “→” é visitada e o tipo do último nível da sub-árvore é retornado.
- Para a visita da sub-árvore direita, é passado um ponteiro para o item da tabela de símbolos que representa o tipo da sub-árvore (sub-expressão) esquerda e que deverá ser um ponteiro para um objeto do tipo *CDeclTipoEstruturado*. Este ponteiro está presente na informação de tipo retornado quando da visita à sub-árvore esquerda.
- Quando da visita da sub-árvore direita, é verificado se o componente direito pertence a um dos campos existentes neste objeto. Esta verificação é feita quando o nó da árvore sintática é do tipo *UN_AS_VARIAVEL* (fig 4.11).
- O processo é repetido recursivamente para níveis superiores da árvore.

Para exemplificação da verificação de tipos e de membro de agregado, suponhamos a seguinte expressão $A = B * C + D.F + -E$ com suas variáveis tendo os seguintes tipos da linguagem C:

A	int
B	float
C	char
D	struct { int F; }
E	short

Figura 4.12 – Variáveis e tipos associados à expressão $A = B * C + D.F + -E$.

A árvore sintática da expressão será como se segue:

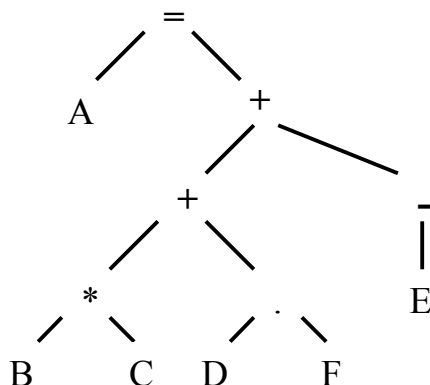


Figura 4.13 – Árvore sintática da expressão $A = B * C + D.F + -E$.

Aplicando-se a função de verificação semântica da figura 4.11 sobre a árvore sintática da figura 4.13, é feito um percurso pós-fixado, resolvendo-se primeiro a sub-árvore $B * C$. Neste caso, o nó B será visitado primeiro sendo retornado o tipo da variável para o nível superior da recursão. O mesmo será feito para o nó C . Uma vez feita a visita de cada um destes nós, têm-se em $ROP1$ e $ROP2$ os tipos de B e C respectivamente. A verificação de tipos para a operação de multiplicação é então feita. Olhando-se os tipos de B e C em $ROP1$ e $ROP2$

constata-se que está sendo feita uma multiplicação entre um tipo *float* e um tipo *char*. O ANSI C especifica que, no caso de operações binárias (com exceção da atribuição) aplicadas entre operandos de tipos diferentes, o tipo de menor nº de *bits* de representação deve ser convertido para o tipo de maior nº de *bits* de representação. Neste caso, a ordem de conversão de tipos para as operações na linguagem C segue a seguinte convenção:

$$char \rightarrow short \rightarrow (int, long) \rightarrow float \rightarrow double,$$

onde, entre os tipos *int* e *long*, a conversão dependerá do número de *bits* da representação do tipo *int* que é dependente de máquina.

Na figura 4.13, teremos então a conversão da variável *C* para o tipo *float* e o resultado da operação de multiplicação será do tipo *float*. Logo após será resolvida a sub-árvore *D.F*. Neste caso, a operação binária se refere à seleção de campo de um agregado, que na geração de código será o cálculo de deslocamento entre o campo *F* e a base do agregado *D*. Deverá neste caso ser feita a verificação de existência do campo *F* no agregado *D*. Para isto, quando da visita do nó da variável *D* será transmitido para a visita do nó *F* um ponteiro para a entrada da tabela de símbolos que descreve o tipo de *D*. Este tipo deverá ser um tipo estruturado. Quando da visita ao nó da variável *F*, será verificado se a variável pertence a um dos campos do respectivo tipo estruturado. Se tudo estiver certo, o tipo deste campo será retornado como resultado da operação. O próximo passo é a resolução da operação de adição das sub-árvores *B * C* e *D.F*. Neste caso, uma vez visitadas cada sub-árvore, *ROP1* e *ROP2* conterão os respectivos tipos de cada sub-expressão. Constatar-se-á que o tipo contido em *ROP1* é *float* e em *ROP2* é *int* (tipo de *F*, componente de *D*). Um novo *typecast* implícito será necessário, transformando o tipo *int* para *float*, sendo que a operação resultará também em um tipo *float*. Aplicando-se o mesmo procedimento, teremos o resultado da sub-árvore *-E* como sendo tipo *short*. A operação de adição entre as sub-árvores *B * C + D.F* e *-E* necessitará também de um *typecast* para a sub-árvore *-E* do tipo *short* para o tipo *float*. O resultado da operação será

então do tipo *float*. A operação final, que é a atribuição deste resultado à variável *A* sofre também de um *typecast* implícito. Como o tipo da variável *A* é *int*, um *typecast* de *float* para *int* deverá ser feito sobre o resultado da sub-árvore $B * C + D.F + -E$.

Embora uma abordagem mais simples para a implementação de verificação de tipos seja fazê-la diretamente no esquema de tradução, a verificação que usa as árvores sintáticas mostra-se bem mais legível para o programador, de fácil expansão, tão eficiente quanto a abordagem no esquema de tradução no que tange os resultados, e ainda, menos complexa que a primeira, pois esta tem que dividir espaço com a geração da representação intermediária do programa e da manipulação de atributos herdados e sintetizados.

4.6.3 Geração da Representação Intermediária

Um processo importante do compilador *Phoenix* é a geração da representação intermediária do código fonte que poderá ser utilizada para a geração de código nativo de algum processador ou para o processo de geração de circuitos para *reconfigware*. Esta representação consiste em uma série de grafos que encapsulam os fluxos de controle do programa original, suas instruções (representadas em um outro formato intermediário) e as dependências de controle e de dados. Os grafos gerados para o suporte a geração de *reconfigware* são:

1. grafo de hierarquia de tarefas (HTG);
2. grafo de fluxo de controle (CFG);
3. grafo de dependência de controle (CDG);
4. grafo de fluxo de dados (DFG);
5. grafo de dependência de dados (DDG).

Além destes grafos, são criadas as árvores de dominância e pós-dominância que auxiliam na construção do grafo de dependência de controle. A justificativa da escolha da

construção de tais grafos é que analisando-se trabalhos anteriores relativos à síntese de alto nível [Athanas 1992; Athanas e Silverman 1993; Chichkov e Almeida 1997; Chichkov 1998; Callahan *et al.* 2000; Cardoso 2000; Gupta *et al.* 2003], eles se mostram os mais comumente utilizados como representação intermediária para otimizações para a geração de *reconfigware*. De acordo com Aho *et al* [Aho *et al.* 1986], a maioria destes grafos é também utilizada nos algoritmos tradicionais de otimização para geração de código nativo. No *Phoenix*, para cada um destes grafos pode ser gerado um arquivo com a representação do mesmo no formato suportado pelo *software Graphviz* para posterior compilação e visualização. As seções a seguir descrevem cada um dos grafos gerados.

4.6.3.1 Grafo de Fluxo de Controle

O grafo de fluxo de controle encapsula os possíveis caminhos de execução do programa. O grafo consiste de nós que encapsulam os blocos básicos do programa e as ligações entre eles mostram os possíveis caminhos de execução entre os nós. O desvio de execução de um bloco (nó) para outro é associado a uma função lógica que pode ser sempre verdadeira, ou ser resultado de uma condição representada por uma instrução do programa. De acordo com Cardoso [Cardoso 2000], o grafo de fluxo de controle representa a estrutura sequencial (um único fluxo de controle) da função original, sendo necessário desta forma expor as dependências de controle e de dados para que a função original possa ser representada mais eficientemente e sejam expostos, sempre que possível, vários fluxos de controle. O grafo constitui assim a base para a geração dos grafos de dependências para exploração do máximo de paralelismo do programa.

De acordo com Gupta *et al* [Gupta *et al.* 2004] e Cardoso [Cardoso 2000], a definição do grafo de fluxo de controle é como se segue.

Definição 4.1:

O grafo de fluxo de controle é um grafo dirigido $G = (V, E, START, END)$, com um conjunto de nós V que representam os blocos básicos do programa e um conjunto de ligações E que representam a dependência de execução entre estes blocos. O grafo inclui o nó $START$, início do CFG, e o nó END , término do CFG. O nó $START$ não possui predecessores e o nó END não possui sucessores. Cada nó está contido no caminho entre o nó $START$ e o nó END . Existe uma ligação entre dois nós (v_1, v_2) , ambos pertencentes a V , se e somente se a instrução do nó v_2 pode ser executada imediatamente após a execução da instrução em v_1 .

O grafo de fluxo de controle é criado durante a fase de tradução do código fonte, sendo gerado através de atributos sintetizados específicos. A geração do grafo desta forma não apresentou problemas significativos de desenvolvimento no que tange a maioria das estruturas de desvio de controle comuns provida pela linguagem C. As instruções de desvio de controle *continue*, *break*, *goto* e *return* foram implementadas utilizando-se de estruturas de dados auxiliares como filas e pilhas que armazenam informações para que as ligações entre os nós envolvidos possam ser feitas mais tardiamente durante o processo de tradução.

4.6.3.2 Grafo de Hierarquia de Tarefas

O grafo de hierarquia de tarefas é um grafo que mantém a estrutura hierárquica do programa como, por exemplo, blocos *if-then-else* e ciclos (*for*, *while*, etc). De acordo com Nicolau e Novack [Nicolau e Novack 1993] e Girkar e Polychronopoulos [Girkar e Polychronopoulos 1992], a definição do grafo de hierarquia de tarefas é como se segue.

Definição 4.2:

O grafo de hierarquia de tarefas (HTG) é um grafo direcionado

acíclico $HTG=(HV,HE)$, com nós $START$ e $STOP$ únicos pertencendo a HV tal que existe um caminho a partir do nó $START$ para todo nó em HV e um caminho de qualquer nó em HV para o nó $STOP$. As ligações (HE) entre os nós do HTG representam fluxo de controle. Cada nó $HV \in HTG$ pode ser de qualquer um dos seguintes três tipos $T_{HTG}=\{NS,NC,CN\}$, correspondendo à nós simples, nós compostos e nós de ciclos.

- 1- Nós simples: representam nós que não possuem sub-nós e são utilizados para representar blocos básicos*
- 2- Nós compostos: são hierárquicos por natureza, contendo outros nós HTG . São utilizados para conter estruturas como blocos *if-then-else*, blocos *switch-case* ou uma série de nós HTG .*
- 3- Nós de ciclos. São usados para representar os vários tipos de ciclos (*for*, *while*, *do-while*).*

No *Phoenix*, cada nó HTG simples possui uma correspondência com um nó CFG . O grafo de hierarquia de tarefas, devido à sua característica hierárquica, permite manter informações de alto nível sobre a estrutura do programa e também informações ao nível de operação e ao nível de blocos básicos. De acordo com Gupta *et al* [Gupta *et al.* 1993], isto auxilia em transformações de otimizações em que existe uma re-estruturação do código em nível de linguagem, como por exemplo, transformações de ciclos (desenrolamento de ciclos, movimentação de código ciclo-invariante, entre outros).

4.6.3.3 Grafo de Fluxo de Dados

O grafo de fluxo de dados mostra o fluxo de dados entre as diversas operações do programa. Desta forma, seus nós representam operações e uma ligação entre dois nós indica que existe uma dependência de fluxo entre eles. De acordo com Gupta [Gupta *et al.* 2004], a

definição do grafo de fluxo de dados é como se segue.

Definição 4.3:

O grafo de fluxo de dados é um grafo dirigido acíclico $DFG=(Ops, E_{data})$, onde o conjunto de nós $Ops=\{op_i; i=1,...,n_{ops}\}$ é o conjunto de operações n_{ops} do programa e o conjunto de ligações $E_{data}=\{(op_i, op_j); i,j=1,...,n_{ops}\}$ representa a dependência de fluxo de dados. Uma ligação direcionada $e_{ij}=(op_i, op_j)$ existe em E_{data} se um dado produzido pela operação op_i é lido pela operação op_j .

O grafo de fluxo de dados é construído no *Phoenix* através da inspeção das cadeias uso-definição (cadeias-*ud*) e definição-uso (cadeias-*du*). Cada uso de uma variável u_n de uma variável tem um conjunto associado $UD(u_n)$ que representa todas as definições que atingem aquele uso u_n da variável. Por sua vez, cada definição d_n de uma variável tem um conjunto associado $DU(u_n)$ que representa todos os usos que são atingidos pela definição d_n da variável. Para o cômputo das cadeias *ud* e *du* é feito primeiramente o cômputo das definições incidentes através do algoritmo proposto em Aho *et al* [Aho *et al.* 1986]. O algoritmo se baseia nas seguintes equações de fluxo de dados:

$$\begin{aligned} Entrada[B] &= \cup Saída[P], \text{ para todo } P \text{ predecessor de } B. \\ Saída [B] &= Geradas[B] \cup (Entrada[B]-Mortas[B]) \end{aligned}$$

Figura 4.14 – Equações de fluxo de dados utilizadas no cômputo da cadeias *ud* e *du*.

Na figura 4.14, B é um bloco básico e P é um predecessor de B . O conjunto *Entrada* corresponde a todas as definições que atingem um bloco básico. O conjunto *Saída* corresponde a todas as definições que estão disponíveis ao fim do bloco básico. O conjunto *Geradas* corresponde a todas as definições que foram criadas dentro do bloco básico. O

conjunto *Mortas* corresponde a todas as definições que não atingem o final do bloco básico, mesmo que tenham atingido o início do bloco.

No *Phoenix* os conjuntos *Entrada*, *Saída*, *Geradas* e *Mortas*, foram todos representados por um vetor de *bits*, existindo um *bit* para cada definição encontrada no programa. As operações de união e interseção comuns nas equações de fluxo de dados podem ser facilmente executadas por operações *bitwise OR* e *AND* nestes vetores, respectivamente.

Utilizando-se das equações da figura 4.14, o algoritmo executa iterativamente sobre o grafo propagando as definições no mesmo num sentido que simula todas as possíveis execuções do programa. Para cada bloco básico *B*, o conjunto *Entrada* é computado de acordo com a primeira equação. Logo após, o conjunto saída é calculado com base na segunda equação. O algoritmo executa até que não seja detectada mudança no conjunto *Saída*.

Os conjuntos *Geradas* e *Mortas* são criados antes da execução do algoritmo. O conjunto *Mortas* de um bloco básico é a união de todos os conjuntos de todas as definições das variáveis definidas no bloco “menos” o conjunto *Geradas* do mesmo bloco. A equação correspondente pode ser deduzida a partir das equações propostas por Aho *et al* [Aho *et al.* 1986] e é como se segue:

$$Mortas(BB) = \cup Dx - G(BB),$$

Figura 4.15 – Equação utilizada no cômputo do conjunto *Mortas*.

onde *Dx* é o conjunto de todas as definições de uma variável *x* definida no bloco. A mesma equação pode ser computada com operações *bitwise* em um vetor de *bits* com a seguinte equação:

$$Mortas(BB) = \cup Dx \text{ AND NOT}(Geradas(BB))$$

Figura 4.16 – Equação da figura 4.15 definida em termos de operações *bitwise*.

O conjunto *Geradas* de um bloco básico é o conjunto de todas as definições que foram criadas no bloco básico. Desta forma, para cada uma das variáveis definidas no bloco, insere-se no conjunto sua última definição desta variável dentro do bloco.

Após a varredura das definições incidentes, são geradas as cadeias-*ud*. O algoritmo varre todos os usos de variáveis existentes no programa. Para cada uso u_i de uma variável v é verificado se existe uma definição inambígua (não gerada por uma função que tem como argumento a variável, sendo esta passada por referência) da mesma no bloco básico no qual u_i é encontrado (é considerada somente a última definição no bloco). Se houver, esta definição faz parte da cadeia-*ud* de u_i . Senão, todas as definições do conjunto *entrada* do bloco básico que sejam definições da variável v pertencem à cadeia-*ud* de u_i . Logo após são inseridas na cadeia-*ud* de u_i todas as definições ambíguas de v que não caiam entre a última definição ambígua (se existir uma) e o uso u_i de v .

O cômputo das cadeias-*du* é feito através da relação $u_i \in DU(d_j) \Leftrightarrow d_j \in UD(u_i)$ disposta por [Bik e Gannon 1997].

A Fig. 4.17 ilustra as cadeias *ud* e *du* obtidas para um trecho de código com hierarquias de controle e cinco variáveis (A, B, C, D, E).

Na figura 4.17, as definições existentes são mostradas no lado esquerdo das instruções enquanto que os usos existentes são colocados em subscrito ao lado da respectiva variável. Nota-se que a cadeia *ud* para a variável D é vazia, pois não existe nenhuma definição da mesma variável que atinja o respectivo uso (u_l). Desta forma, tem-se um erro de programação, pois a variável foi usada sem antes ser definida. Já a cadeia *du* para a variável E é vazia, pois a variável é definida, mas não é usada em momento algum sendo, portanto, código morto.

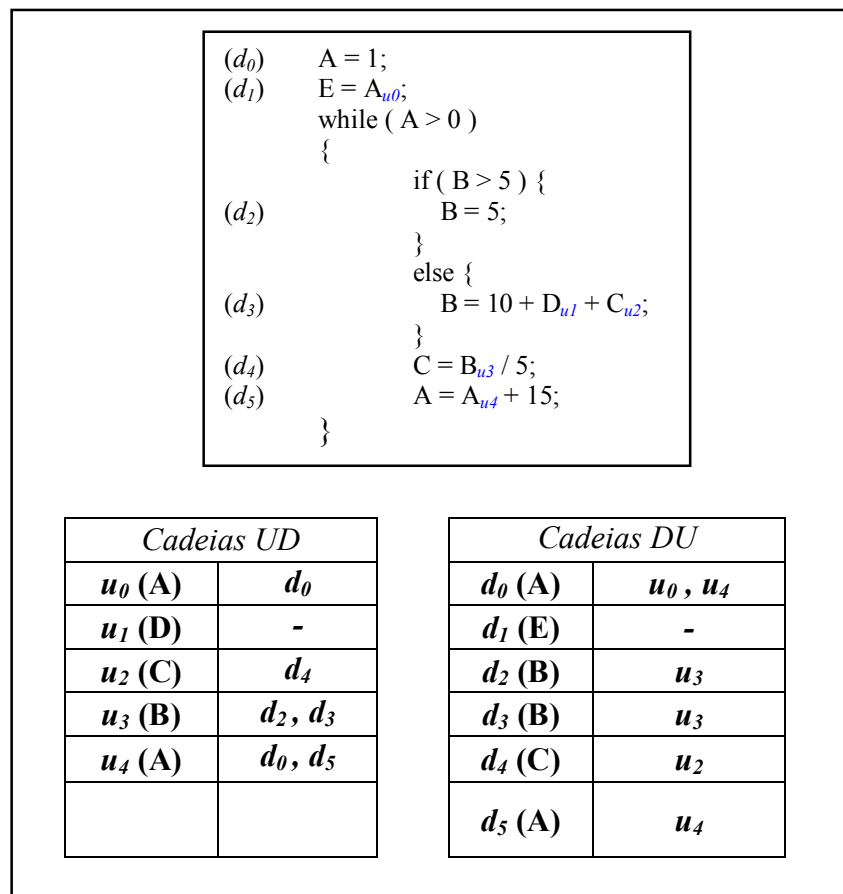


Figura 4.17 – Exemplo de cadeias *ud* e *du* de um programa.

No *Phoenix* é feita somente a análise de fluxo de dados intraprocedimental, sendo que a análise de fluxo de dados entre funções foi deixada para trabalhos futuros. As informações sobre as cadeias *ud* e *du* são mantidas em nível de blocos básicos, variáveis (na respectiva entrada da tabela de símbolos) e funções. Desta forma, as cadeias *ud* e *du* existentes para cada um dos três níveis podem ser acessadas mais facilmente.

Uma vez geradas as cadeias *ud* e *du*, o grafo de fluxo de dados é criado por inspeção nas cadeias-*du*. Para cada instrução é criado um nó no grafo. Para cada definição existente é feita a ligação com seus respectivos usos.

4.6.3.4 Grafo de Dependência de Dados

O grafo de dependência de dados expressa as dependências de dados em nível de

blocos básicos. Para cada nó do grafo de fluxo de controle existe um nó correspondente no grafo de dependência de dados. No caso do DDG, as ligações entre os nós representam dependência de ordem de execução em termos de dependências de dados. Este grafo permite explorar o paralelismo a nível de blocos básicos. Se dois blocos são independentes em termos de fluxo de dados, os mesmos podem ser executados em paralelo.

Definição 4.4:

O DDG é um grafo dirigido que engloba o conjunto de nós do CFG de onde foi obtido, no qual as ligações entre os blocos básicos representam dependências de dados que restringem a ordem de execução destes blocos de forma a preservar a funcionalidade do programa ao nível de dependências de dados. Cada ligação é anotada com a informação sobre o tipo de dependência (de fluxo, de entrada, de saída ou anti-dependência).

No *Phoenix*, o DDG é criado por inspeção das cadeias-*du* contidas nos blocos básicos. Para cada uso atingido por uma definição existente em um bloco básico i , é feita a ligação com o bloco básico j atingido (com $i \neq j$). No DDG criado pelo *Phoenix*, são consideradas somente as dependências de dados do tipo fluxo. Todos os outros tipos de dependências podem ser obtidos também por análise das cadeias *ud* e *du*.

4.6.3.5 Grafo de Dependência de Controle

O grafo de dependência de controle expressa as dependências de dados em nível de controle. Para cada nó do grafo de fluxo de controle existe um nó correspondente no grafo de dependência de controle.

Definição 4.5:

O CDG é um grafo dirigido que engloba o conjunto de nós do CFG de

onde foi obtido, no qual as ligações entre os blocos básicos representam dependências de controle que restringem a ordem pela qual estes blocos devem ser executados de forma a preservar a funcionalidade do programa ao nível controle. Cada ligação é anotada com a informação da condição (verdadeira ou falsa) usada para o desvio de execução para o nó sucessor.

No *Phoenix*, o grafo de dependência de controle é criado através do algoritmo proposto por Cytron *et al* [Cytron *et al.* 1991]. Para a criação do CDG é primeiro criado a árvore de pós-dominâncias por inspeção no CFG. Um nó v_2 pós-domina um nó v_1 se todos os caminhos inversos do nó END até v_1 contêm o nó v_2 . Por definição, um nó pós-domina a si próprio. No *Phoenix*, a árvore de dominâncias foi criada de acordo com o algoritmo iterativo proposto por Aho *et al* [Aho *et al* 1986].

Com a árvore de pós-dominâncias e o CFG é criado o CDG com base nos seguintes fatos:

Um nó v_1 é dependente em termos de controle do nó v_2 , se e somente se:

1. Existir um caminho não nulo de v_2 a v_1 , de modo a que v_1 pós-domina todos os nós após v_2 no caminho, e;
2. v_1 não pós-domina estritamente o nó v_2 .

4.6.4 Decomposição de *Structs* e *Unions* ao Nível Primitivo ou *Array*

De acordo com Compton e Hauck [Compton e Hauck 2000], a grande maioria dos compiladores criados para o processo de síntese de alto nível de circuitos digitais que tem como entrada a linguagem C não possui suporte total aos recursos da linguagem. Geralmente estes compiladores não implementam suporte a ponteiros, agregados de dados, objetos

dinâmicos, e algumas vezes nem mesmo *arrays*. A figura 4.18 ilustra alguns compiladores para síntese de alto nível de circuitos digitais e o suporte de cada um quanto aos recursos da linguagem C. Na figura pode-se perceber que todos são restritivos quanto a algum recurso da linguagem C.

Compilador	PRISM II [Athanas 1992]	Transmo- grifierC [Galloway 1995]	Chichkov [Chichkov 1998]	Garpcc [Callahan <i>et al</i> 2000]	SPARK [Gupta <i>et al</i> 2003]	<i>Phoenix</i>
1ª Publicação	1993	1995	1998	2000	2003	2006
<i>if-then-else</i>	V	V	V	V	V	*
Ciclos	V	<i>while</i>	V	V	V	*
<i>Arrays</i>	-	-	-	V	V	*
Estruturas	-	-	-	V	-	V
Ponteiros	-	-	-	V	-	*
Chamada de Funções	-	-	-	V	V	*
Recursividade	-	-	-	-	-	*
Representação Intermediária	DFG+CFG	AST	CDG+ DDG+ AST	DFG+ Hiperbloco	HTG+ DDG+ CDFG	HTG+ CFG+CDG+ DFG+DDG
Saída	VHDL	<i>Bitstreams</i>	VHDL	<i>Bitstreams</i>	VHDL	*

Tabela 4.1 – Alguns compiladores de síntese de alto nível de circuitos digitais e seus suportes à linguagem C.

Embora o processo de análise do programa fonte por parte do *Phoenix* permita suporte de todas estas construções da linguagem C para a geração de código nativo, a utilização destas construções na geração de circuitos deve receber tratamentos específicos. No caso dos ponteiros, por exemplo, muitas vezes não existe uma equivalência semântica em sua utilização em nível de circuito, como explicado na conclusão do capítulo 3. A utilização de objetos dinâmicos pode exigir a utilização de um gerenciador de memória, que no caso

deveria ser também implementado em nível de circuito, o que aumenta a área do circuito final. Variáveis de um programa, por questões de performance, são mapeadas para registradores quando da geração do circuito do respectivo programa. Como agregados de dados possuem tamanhos variáveis, estes devem ser mapeados para memória. Mas neste caso existirá uma perda de performance considerável devido aos acessos à memória e cálculos de endereços. Desta forma o mapeamento de agregados para memória não se mostra tão atrativo, pois fere o objetivo principal da construção do circuito representativo do programa que é o ganho de velocidade de execução.

Visando contribuir com a síntese de alto nível de circuitos digitais a partir da linguagem C, foi implementado no *Phoenix* um recurso que permite o compilador decompor automaticamente agregados de dados em variáveis simples ou *arrays*. As justificativas para a implementação de tal recurso são:

- Completude quanto ao suporte à linguagem.
- Possibilidade de mapeamento direto dos agregados de dados para registradores, visto que estes são transformados em variáveis simples.
- A modificação automática da representação dos agregados dispensa modificações manuais por parte do programador.
- De acordo com Cytron *et al* [Cytron *et al.* 1991], a decomposição de agregados em seus componentes mais primitivos mostra-se mais atraente para a criação do formato estático de atribuição única (SSA) através do algoritmo por eles proposto.
- Uma vez que o agregado foi transformado em variáveis de tipo primitivo ou *arrays*, elas conseqüentemente se mostram mais adequadas aos algoritmos de

análise de fluxo de dados e otimização. Além disto, a decomposição permite explorar um nível maior de paralelismo quanto ao fluxo de dados.

Embora o compilador *Garpcc* suporte tais tipos de estruturas, até onde sabemos, o compilador não implementa o processo de decomposição (ou outro semelhante) implementado no *Phoenix*. Ele mapeia os agregados para a memória do dispositivo, usando-se de instruções *load* e *store* para acessar as estruturas.

No *Phoenix*, a decomposição dos agregados é feita somente em agregados criados com o construtor *struct*. O construtor *union* não é considerado, pois seus campos não são disjuntos. Mas o compilador permite que agregados do tipo *union* sejam tratados como disjuntos (*struct*) através de uma opção de linha de comando. O processo de decomposição é feito simultaneamente com a criação das instruções de três endereços. Os nomes dos operandos para o operador “.” são concatenados árvore sintática acima até o aparecimento de qualquer outro operador. Neste ponto é definido o tipo e a entrada da tabela de símbolos para o símbolo. A figura 4.18 ilustra este procedimento para a expressão $S1.S2.A=0$, considerando as seguintes declarações que declaram duas estruturas e uma variável *S1* do tipo *Struct1*:

```
struct Struct2 { char A; };
struct Struct1 { Struct2 S2; } S1;
```

Varrendo-se a árvore sintática, ocorrerá a concatenação dos nomes *S1* com *S2*, e posteriormente com *A*. Quando do encontro do operador “=” a concatenação é terminada. Neste ponto toda a sub-árvore à esquerda do operador “=” é substituída por um nó correspondente a um nó de variável, cujo nome é *S1.S2.A*. Sua entrada na tabela de símbolos será uma nova entrada que é uma cópia (exceto o nome do símbolo) da entrada correspondente ao símbolo *A*. É gerada uma instrução de três endereços da forma $S1.S2.A = 0$, onde *S1.S2.A* é o nome do símbolo (o ponto é inserido como parte da *string* que representa o nome). Se o processo de decomposição não fosse executado, seriam geradas as seguintes

instruções de três endereços: $T_0 = S1.S2$, $T_1 = T0.A$ e $T_2 = 0$.

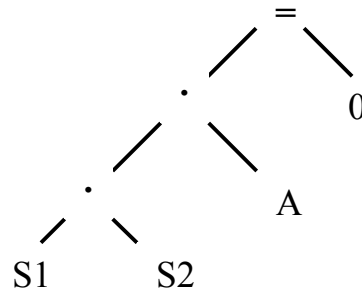


Figura 4.18 – Árvore sintática para a expressão $S1.S2.A = 0$.

A figura 4.19 ilustra um exemplo mais complexo envolvendo as seguintes declarações que declaram três estruturas e uma variável $S3$ do tipo *Struct3*:

```

struct Struct1 { int A; };
struct Struct2 { Struct1 S1; };
struct Struct3 { Struct2 *S2; } S3;

```

Na figura 4.19, varrendo-se a árvore sintática, ocorrerá a concatenação dos nomes $S3$ com $S2$. Uma vez encontrado o operador “ $\rightarrow$ ” a concatenação é terminada. Neste ponto toda a sub-árvore à esquerda do operador “ $\rightarrow$ ” é substituída por um nó correspondente a um nó de variável, cujo nome é $S3.S2$. Sua entrada na tabela de símbolos será uma nova entrada que é uma cópia (exceto o nome do símbolo) da entrada correspondente ao símbolo $S2$. Na sub-árvore direita ao operador “ $\rightarrow$ ” ocorrerá a concatenação dos nomes $S1$ com A . Uma vez reencontrado o operador “ $\rightarrow$ ” a concatenação é terminada. Neste ponto toda a sub-árvore à direita ao operador “ $\rightarrow$ ” é substituída por um nó correspondente a um nó de variável, cujo nome é $S1.A$. Sua entrada na tabela de símbolos será uma nova entrada que é uma cópia (exceto o nome do símbolo) da entrada correspondente ao símbolo A . Logo após, devido ao operador “ $\rightarrow$ ”, é criada a instrução de três endereços $T_0 = S3.S2 \rightarrow S1.A$. Terminando-se a varredura da árvore sintática é criada a última instrução de três endereços: $T_0 = 0$.

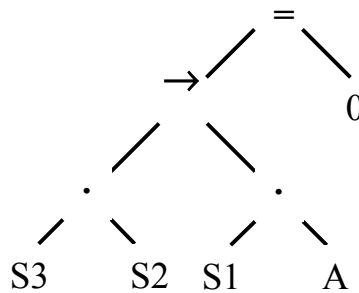


Figura 4.19 – Árvore sintática para a expressão $S3.S2 \rightarrow S1.A = 0$.

Se o processo de decomposição não fosse executado, seriam geradas as seguintes instruções de três endereços: $T_0 = S3.S2$, $T_1 = S1.A$, $T_2 = T_0 \rightarrow T_1$ e $T_2 = 0$.

O processo de decomposição pára também quando da detecção de um *array* (operador `[]`). Isto decorre pelo fato de que neste caso este *array* será mapeado para a memória do dispositivo reconfigurável, como é o que é feito pela maioria dos compiladores de *reconfigware* analisados. Uma decomposição do *array* poderia também ser feita se este se apresentasse simples como, por exemplo, um *array* de 5 posições de números inteiros. Embora esta decomposição também fosse útil, este recurso foi deixado para estudos e implementações futuras.

Resultados observados na decomposição de agregados ao nível primitivo ou *array* mostraram que esta decomposição pode reduzir bastante a quantidade de instruções de três endereços na representação do programa, acelerando desta forma a geração de código. Embora tal recurso apresente-se extremamente útil para o processo de síntese de alto nível de circuitos digitais, sua utilização na geração de código não se mostra muito atraente, pois uma vez que os agregados são decompostos, o programador perde a visão das hierarquias e das estruturadas de dados criadas, o que pode dificultar o processo de depuração do programa. Para suprir este problema, este suporte de decomposição pode ser habilitado ou desabilitado via linha de comando do compilador.

4.6.5 Interface de Retaguarda

A geração de código é feita pela interface de retaguarda do compilador, também chamada de gerador de código. O processo de geração de código consiste na tradução da representação intermediária do programa fonte em um programa alvo equivalente onde, tradicionalmente, cada operação da linguagem é mapeada em uma respectiva instrução de máquina. A geração de código ótimo é matematicamente insolucionável [Aho *et al.* 1977a]. Mas apesar disto, pode-se conseguir um código de boa qualidade através de aplicação de heurísticas. A escolha de tais heurísticas deve ser cuidadosa, pois podem afetar drasticamente o desempenho do programa. A produção de um código correto é o critério mais importante para a geração de código. De forma resumida, o processo de geração de código consiste no gerenciamento da memória em tempo de execução, seleção de instruções da máquina e alocação de registradores para as operações. A geração de representações em HDLs a partir da representação intermediária segue um padrão diferente, sendo parte dela o processo de escalonamento das operações do grafo de fluxo dados, o mapeamento das operações para componentes de circuitos específicos, o mapeamento de variáveis para registradores ou memória e a geração desta representação na HDL específica.

4.6.5.1 Compilação para Múltiplos Alvos

Um importante aspecto que deve ser observado quando da construção de um compilador é a da possibilidade de se adaptar o compilador para geração de código para vários tipos de máquinas. Um projeto de um compilador de arquitetura fixa para uma determinada máquina não é vantajoso, pois a velocidade de projeto de novos computadores nos dias atuais é muito grande. Desta forma, uma nova versão de uma máquina recente pode estar disponível em pouco tempo. Se esta nova versão possuir um novo conjunto de instruções ou outro tipo de suporte às operações de linguagens, o novo compilador deveria ser criado

praticamente a partir de seu início. O projeto de um compilador reorientável é um projeto em que a interface de vanguarda do compilador pode ser aproveitada para a geração de código para várias CPUs com nenhuma ou poucas mudanças nesta interface. Para isto deve existir um encaixe entre a interface de vanguarda e a de retaguarda do compilador que permita o reaproveitamento do projeto sem muitas modificações. Este encaixe pode ser feito através da representação do programa original em uma forma intermediária independente da máquina alvo. Muitos compiladores foram propostos com infraestrutura voltada para o reuso dos componentes de compilação e tecnologia. Algumas das propostas mais relevantes de compiladores deste tipo são PCC2 [Johnson 1980], IMPACT [Chang *et al.* 1991], Zephyr [Andrew *et al.* 2005] e GCC.

PCC2 gera código de montagem a partir de árvores sintáticas. A geração deste código se dá através da geração de código por casamento de padrões de árvores e programação dinâmica proposta por Aho *et al* [Aho *et al.* 1989]. O casamento de padrões é feito através de *templates* de geração de código que devem ser especificados pelo programador quando da construção do compilador. Cada *template* especifica a substituição de uma sub-árvore da expressão (operação) por uma instrução de máquina correspondente.

IMPACT utiliza-se de dois níveis de representação da máquina. Uma é orientada ao usuário, de mais alto nível, sendo de fácil leitura e modificação, enquanto que a outra é de mais baixo nível voltada para o próprio compilador e seus processos. As duas linguagens de descrição de máquinas são independentes das estruturas internas do compilador.

Zephyr e GCC utilizam-se da representação de máquina através de linguagem específicas que abstraem o nível de transferência de registros (RTL), que é a representação intermediária do programa. A linguagem de descrição de máquina utilizada pelo GCC tem sintaxe baseada nas listas LISP e provê 114 instruções padrão, sendo este conjunto fixo para qualquer alvo. Cada instrução RTL casa com pelo menos uma instrução de máquina. Zephyr é

uma ferramenta comercial e utiliza-se de uma linguagem RTL que possui uma semântica independente de qualquer máquina. Esta linguagem é fortemente tipada, baseada na linguagem ML e não é muito voltada para o usuário, sendo de difícil legibilidade. De um modo geral, Zephyr é difícil de ser diretamente adaptado para produzir compiladores práticos [Guilan *et al.* 2002b].

Guilan *et al* [Guilan *et al.* 2002a] faz um panorama da atual situação da compilação objetivando múltiplos alvos. De acordo com o trabalho, embora resultados relevantes tenham sido conseguidos no que tange a computação para múltiplos alvos, as pesquisas sobre interfaces de retaguarda flexíveis ainda estão longe da maturidade tanto no aspecto teórico quanto no aspecto prático. Para permitir a construção de um compilador para múltiplos alvos, a infraestrutura do compilador precisa abstrair as diferenças e semelhanças entre as diversas linguagens fonte (se for o caso), processadores e sistemas operacionais. Técnicas especiais são necessárias para descrever o ambiente alvo, para a criação de geradores de código, e para construir as interfaces entre as descrições do ambiente alvo e os geradores de código. As representações intermediárias devem suportar abstrações em níveis apropriados tal que elas não possam ser apenas mapeadas de múltiplas linguagens fonte, mas também serem adaptadas à múltiplas plataformas. Um meio que permita uma descrição detalhada dos recursos da máquina alvo é também requerido para este tipo de projeto. A maioria dos compiladores existentes voltados para esta reorientabilidade possui uma forma própria de especificação de máquina. Muitas vezes essa forma não é de propósito geral, sendo voltada para arquiteturas específicas. Além disto estas formas sofrem de falta de regras gerais para abstração e conteúdo das descrições. Fica difícil descrever uma nova arquitetura ou saber o quê descrever. Desta forma, existe uma necessidade de completude e padronização na forma de descrição de uma máquina. Ainda, em algumas situações, geradores automáticos de geradores de códigos são também necessários para simplificar o processo de desenvolvimento. Entretanto, a

geração automática de geradores de geradores de código ainda é um desafio. No que tange a interface de ligação entre a vanguarda e a retaguarda, esta não pode ser nem simples demais, pois pode prover muito pouca informação para a interface de retaguarda, e nem pode ser complexa demais, pois complicaria demais a interface de retaguarda [Fraser e Hanson 1995]. A fim de criar compiladores reorientáveis, pesquisas futuras devem se focar sobre capacidade de inserir a descrição de máquinas, construção de geradores de código e interfaces de ligação entre a vanguarda e retaguarda do compilador.

4.6.5.2 Geração de Código no *Phoenix*

No *Phoenix*, a geração de código consiste na tradução das instruções de três endereços para as respectivas instruções de máquina. O gerador de código consiste em um objeto que recebe a representação intermediária de uma função e gera o código para a respectiva função. A interface de geração de código foi projetada para ser adaptável ao processo de vanguarda do compilador cujo elo de ligação é a representação intermediária já descrita. Desta forma pode-se gerar código para múltiplos alvos sendo necessário somente ligar o gerador de código à vanguarda. A figura 4.20 ilustra este esquema.

Para o gerador de código é passado somente o grafo HTG da função. Desta forma, o gerador de código é invocado para compilar uma função de cada vez. Embora seja transmitido para o gerador de código somente o HTG, através deste grafo pode-se acessar toda a representação intermediária, pois o conjunto de todos os grafos do *Phoenix* representa um sistema de várias camadas, havendo ligações entre as diversas camadas. Cada nó do HTG (nó simples) possui um apontador para o respectivo nó CFG que por sua vez possui apontadores para a estrutura do bloco básico (instruções de três endereços, cadeias *ud* e *du*, etc) e para os grafos de dependência de controle e de dependência de dados.

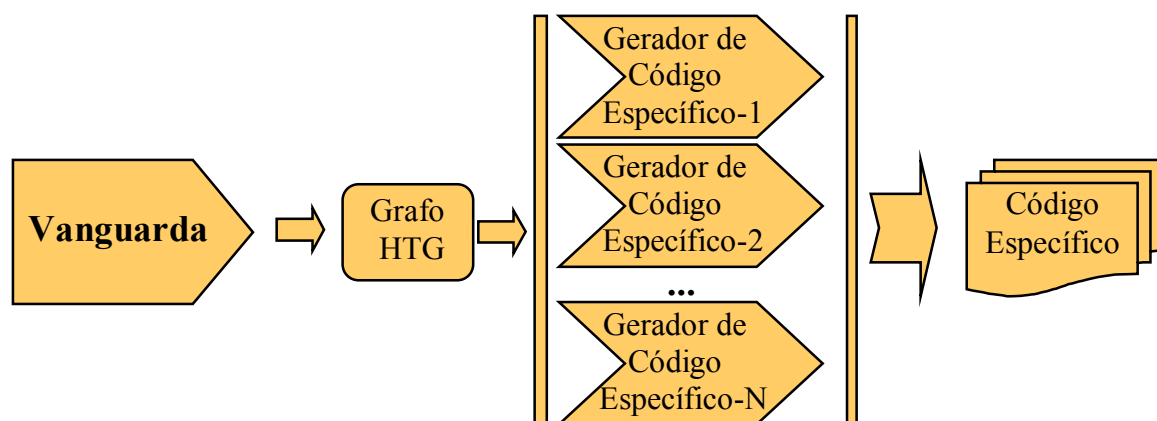


Figura 4.20 – Modelo do *Phoenix* para a geração de código para múltiplos alvos.

A geração de HDLs neste modelo pode ser feita para cada função separadamente. O mesmo ocorre com a geração de código nativo. Mas, neste caso, os códigos individuais de funções existentes no mesmo arquivo em processo de compilação podem ser agrupados para posterior ligação, como é comumente feito pelos compiladores C.

Para a geração de código para processadores, o compilador possui um motor responsável pela alocação de registradores para as operações e pelo mapeamento das operações para as respectivas instruções de máquina. As instruções de máquina no *Phoenix* são descritas em uma linguagem de descrição de máquinas própria, tendo sido desenvolvida com o propósito de poder especificar em alto nível boa parte das máquinas CISC e RISC. Seu desenvolvimento ficou focado também na correspondência com os tipos de dados e operadores existentes na linguagem C. Desta forma, o desenvolvimento da linguagem não visou a questão da completude apontada anteriormente. Mas, apesar disto, a criação da linguagem visou bastante a questão da simplicidade da linguagem com o objetivo de facilitar a especificação por parte do programador. Isto reforça um dos objetivos principais do compilador *Phoenix* que é a facilidade para expansão de seus recursos. Tal simplicidade, de acordo com Guilan *et al* [Guilan *et al.* 2002a], não é encontrada em boa parte das linguagens de descrição de máquinas existentes. A linguagem de descrição de máquinas construída

mostrou-se satisfatória neste aspecto, sendo bastante simples para a construção das especificações.

A descrição de máquina é feita em uma linguagem simples de alto nível que pode ser compilada para uma representação a ser utilizada pelo gerador de código em tempo de execução. Nela são especificadas propriedades da máquina necessárias para a geração de código. Exemplos de tais informações são: o conjunto de registradores, tamanho das instruções, tamanho da palavra (largura de dados do processador), largura de endereçamento, o alinhamento para os diversos tipos de dados e estruturas da linguagem C (funções, ponteiros, tipos primitivos, agregados), comportamento do crescimento da pilha (se cresce para os endereços de memória mais baixos, ou o contrário), entre outros. Outro tipo de informação importante é o conjunto de instruções da máquina. A linguagem suporta 7 tipos de instruções:

1. Instrução de operação unária que utiliza operando em registrador (palavra reservada *INST_REG*).
2. Instrução de operação unária que utiliza operando em memória (palavra reservada *INST_MEM*).
3. Instrução de operação binária que utiliza operadores em registradores (palavra reservada *INST_REG_REG*).
4. Instrução de operação binária que utiliza operando esquerdo em registrador e operando direito em memória (palavra reservada *INST_MEM_REG*).
5. Instrução de operação binária que utiliza operando direito em registrador e operando esquerdo em memória (palavra reservada *INST_REG_MEM*).
6. Instrução de operação binária que utiliza operando esquerdo do tipo valor imediato

e operando direito em registrador (palavra reservada *INST_REG_IMMEDIATO*).

7. Instrução de operação binária que utiliza operando direito do tipo valor imediato e operando esquerdo em registrador (palavra reservada *INST_MEM_IMMEDIATO*).

As instruções que envolvem operando em memória podem ser instruções que identificam valores por endereçamento por *base* ou por *base* somado a *deslocamento*, sendo que neste último caso *base* será um registrador e *deslocamento* será um valor imediato. A linguagem suporta também dez tipos de dados:

1. **Tipo com 8 bits sinalizado:** corresponde à palavra reservada da linguagem *SIGNED_8_BITS*. Este tipo de dado é mapeado para o tipo *signed char* da linguagem C.
2. **Tipo com 8 bits não sinalizado:** corresponde à palavra reservada da linguagem *UNSIGNED_8_BITS*. Este tipo de dado é mapeado para o tipo *unsigned char* da linguagem C.
3. **Tipo com 16 bits sinalizado:** corresponde à palavra reservada da linguagem *SIGNED_16_BITS*. Este tipo de dado é mapeado para o tipo *signed short* da linguagem C.
4. **Tipo com 16 bits não sinalizado:** corresponde à palavra reservada da linguagem *UNSIGNED_16_BITS*. Este tipo de dado é mapeado para o tipo *unsigned short* da linguagem C.
5. **Tipo com 32 bits sinalizado:** corresponde à palavra reservada da linguagem *SIGNED_32_BITS*. Este tipo de dado é mapeado para o tipo *signed long* da linguagem C.

6. **Tipo com 32 *bits* não sinalizado:** corresponde à palavra reservada da linguagem *UNSIGNED_32_BITS*. Este tipo de dado é mapeado para o tipo *unsigned long* da linguagem C.
7. **Tipo sinalizado com a largura de *bits* igual ao tamanho da palavra da máquina:** corresponde à palavra reservada da linguagem *SIGNED_WORD_SIZE*. Este tipo de dado é mapeado para o tipo *signed int* da linguagem C pelo fato do tipo *int* ser relativo à arquitetura.
8. **Tipo não sinalizado com a largura de *bits* igual ao tamanho da palavra da máquina:** corresponde à palavra reservada da linguagem *UNSIGNED_WORD_SIZE*. Este tipo de dado é mapeado para o tipo *unsigned int* da linguagem C.
9. **Tipo flutuante com a largura de *bits* igual ao tamanho da palavra da máquina:** corresponde à palavra reservada da linguagem *FLOAT_WORD_SIZE*. Este tipo de dado é mapeado para o tipo *float* da linguagem C.
10. **Tipo flutuante com o dobro da largura de *bits* igual ao tamanho da palavra da máquina:** corresponde à palavra reservada da linguagem *FLOAT_DWORD_SIZE*. Este tipo de dado é mapeado para o tipo *double* da linguagem C.

Para cada um dos seguintes operadores da linguagem C, pelo menos uma instrução de qualquer tipo deve ser especificada para cada tipo de dado para que seja possível o processo de geração de código:

- **Operadores unários:** *!*, *~*, *-*, *+*.
- **Operadores binários:** *+*, *-*, ***, */*, *%*, *||*, *&&*, *&*, *^*, *|*, *<<*, *>>*, *<*, *<=*, *>*, *>=*, *!=*, *==*.

Ainda devem ser especificadas, para cada tipo de dado, pelo menos uma instrução correspondente às operações de movimentação de dados da memória para registrador (*load*), de dados de registrador para memória (*store*) e de constantes para registrador. Devem ser especificadas também as instruções de desvio condicional (desviar se igual, diferente, menor, menor ou igual, maior, maior ou igual) e incondicional, chamada de sub-rotina e de movimentação de blocos de memória.

As instruções são representadas em uma forma que lembra as declarações de funções da própria linguagem C. A figura 4.21 mostra o formato geral das instruções.

<pre> INSTRUCTION [Type] [SubType] ([Arguments, ExtraArguments]) ([asm(String, Arguments)] return (Value₁,..., Value_N)) </pre>
--

Figura 4.21 – Formato geral para descrição das instruções.

Na figura 4.21:

- *Type* é qualquer um dos sete tipos de instrução descritos anteriormente.
- *SubType* é qualquer um dos dois modos de endereçamento: *Base*, *Base_Imed*.
Algumas instruções não precisam do subtipo da instrução.
- *Arguments* especifica os argumentos passados para codificar a instrução. Os argumentos possíveis são *Reg_x*, um registrador qualquer, um registrador declarado nas declarações iniciais da especificação ou uma construção do tipo *Constant : Size*, onde *Constant* é uma palavra reservada que pode assumir um endereço de memória ou um valor imediato (uma constante) e *Size* é o tamanho do dado em *bits*. O número de *bits* utilizado na codificação dos registradores é

especificado nas declarações iniciais.

- *ExtraArguments* especifica registradores extras necessário para codificar a instrução. Neste caso, a operação a ser codificada pode não ser implementável em uma única instrução, podendo-se precisar de registradores extras para sua codificação.
- *asm* especifica o código *assembly* a ser gerado. *String* é a *string* a ser formatada de acordo com *Arguments*, que pode ser qualquer um dos argumentos especificados no protótipo da instrução.
- *return* especifica o código de toda instrução que pode ser um único valor (instrução) ou uma sequência de valores (instruções). Cada um dos valores (uma instrução única) é especificado em termos de concatenação de *bits*. O operador “::” aplica uma concatenação entre *bits*. O tamanho do código total é a soma de todas as instruções criadas e é calculado em tempo de compilação da descrição.

A figura 4.22 ilustra parte da declaração de arquitetura do processador PentiumTM [Intel 1997c]. As cinco declarações iniciais descrevem a largura de representação de vários atributos da máquina como tamanho da instrução, tamanho da palavra de máquina, largura de endereçamento, número total de registradores, e número de *bits* usados para a representação dos registradores. Além disto são especificados os registradores de propósito geral e os registradores de propósito específico. Logo após são definidos o alinhamento para os diversos tipos de dados da linguagem C e o comportamento da pilha. Na figura estão definidas ainda instruções para a operação de soma e multiplicação.

Para a operação de adição, são especificadas três instruções para adição de dados de 8 *bits* com ou sem sinal. Na primeira instrução a operação pode ser feita entre dois registradores quaisquer (*Reg1* e *Reg2*). Uma vez feito o processo de atribuição de registradores para a

operação pelo gerador de código, o mesmo codifica a instrução fazendo a concatenação de *bits* expressa pela mesma (0000::0001::11::Reg1::Reg2). Supondo que *Reg1* seja o registrador EAX e *Reg2* seja EBX, teríamos a instrução 0000::0001::11::EAX::EBX. Na segunda instrução a operação pode ser feita entre memória e registrador por endereçamento por base. O argumento *Constant : 32* indica que a largura do endereço é de 32 *bits*. A terceira pode ser feita entre um valor imediato de 32 *bits* e um registrador.

Para a operação de multiplicação foi especificada uma instrução para multiplicação de dados de 32 *bits* com ou sem sinal, que exige o registrador EAX como operando esquerdo e qualquer outro como operando direito.

INSTRUCTION_SIZE	-	:: Em bits
WORD_SIZE	32	:: Em bits
ADDRESS_RANGE	32	:: Em bits
TOTAL_REGS	32	:: Em bits
REGS_RANGE	3	:: 3 bits para cada registrador
REGS	{ EAX=000, ECX=001, EDX=010, EBX=011, ESP=100, EBP=101, ESI=110, EDI=111 }	
STACK_POINTER	ESP	
BASE_POINTER	EBP	
;; Registradores específicos		
GENERAL_REGS	{ EAX, ECX, EDX, EBX, ESI, EDI }	
TEMP_REGS	{ EAX, ECX, EDX, EBX, ESI, EDI }	
ARGS_REGS	{ EAX, ECX, EDX, EBX, ESI, EDI }	
RETURN_REGS	{ EAX, EDX }	
CALLER_SAVED_REGS	{ }	
EXCEPTION_RETURN_REG	-	
RETURN_ADRESS_REG	-	
;; Alinhamento dos dados		
CHAR_ALIGN	8	
SHORT_ALIGN	16	
INT_ALIGN	WORD_SIZE	
LONG_ALIGN	WORD_SIZE	
FLOAT_ALIGN	WORD_SIZE	
DOUBLE_ALIGN	64	
STRUCT_ALIGN	WORD_SIZE	
POINTER_ALIGN	WORD_SIZE	
FUNCTION_ALIGN	-	
STACK	TOP_DOWN	:: Cresce dos endereços altos para os baixos
;; Operações		
ADD	;; Soma	
SIGNED_8_BITS		
UNSIGNED_8_BITS		

```

INSTRUCTION INST_REG_REG ( Reg1, Reg2 )
(
    asm( "add %p, %p", Reg1, Reg2 )
    return ( 0000::0001::11::Reg1::Reg2 )
)

INSTRUCTION INST_MEM_REG Base ( Constant : 32, Reg )
(
    asm( "add [%p], %p", Constant, Reg )
    return( 0000::0011::10::Reg1::100::Constant )
)

INSTRUCTION INST_IMMEDIATO_REG (Constant: 32, Reg )
(
    asm( "add %p, %p", Reg2, Constant )
    return( 1000::0001::1100::Reg::Constant )
)

(...)

MUL                                ;; Multiplicação

SIGNED_32_BITS
UNSIGNED_32_BITS

INSTRUCTION INST_REG_REG ( EAX, Reg )
(
    asm( "mul %p, %p", EAX, Reg )
    return ( 1111::0111::11::100::Reg )
)

(...)

```

Figura 4.22 – Exemplo de parte da especificação do Pentium™.

O mapeamento entre as instruções de três endereços e as instruções presentes na descrição da máquina é feito pelo gerador de código seguindo os seguintes passos:

1. A instrução de três endereços é decodificada de acordo com o operador.
2. São alocados registradores para o(s) operando(s) envolvido(s). Em caso de operandos que são resultados de uma operação anterior, é utilizado o registrador associado ao resultado.
3. A instrução para a operação é obtida por indexação primeiramente no *array* relativo às operações e depois no *array* relativo ao tipo de dado dos operandos. Neste ponto existem sete entradas para as sete instruções possíveis e pelo menos uma delas deve estar preenchida.

4. É gerado o código da instrução através da análise dos argumentos requeridos pela instrução e a concatenação de todos os *bits* e argumentos envolvidos.
5. A instrução é inserida em um *stream* que contém todo o código.

Como exemplo, considere a expressão $S.A[B+C].D = S.E$ com as seguintes declarações das variáveis:

<pre>struct Struct1 { int D; }; int B, C; Struct2 S;</pre>	<pre>struct Struct2 { int E; Struct1 A[10]; };</pre>
---	--

Figura 4.23 – Declarações das variáveis da expressão $S.A[B+C].D = S.E$.

Para a expressão, seria gerada a seguinte sequência de instruções de três endereços:

<pre>1: T₀ = S.A 2: T₁ = B+C 3: T₂ = T₀ [] T₁</pre>	<pre>4: T₃ = T₂.D 5: T₄ = S.E 6: T₃ = T₄</pre>
---	---

Figura 4.24 – Instruções de três geradas para a expressão $S.A[B+C].D = S.E$.

Seguindo os cinco passos descritos anteriormente e considerando-se todas as três variáveis envolvidas como sendo globais, a geração de código seria como se segue:

- A primeira operação é decodificada de acordo com o operador. O operador “.” calcula o endereço (valor-L) de memória de A em relação a S . Como a variável S é global, o endereço é um valor absoluto. Se S fosse local, o endereço precisaria ser calculado com base no ponteiro atual do frame. Um registrador é alocado

para armazenar o resultado da operação que será do tipo $Reg \leftarrow Endereço$. O registrador (que contém o endereço calculado) e o tipo do campo A são associados à instrução de três endereços para uso futuro. Uma instrução de carregamento de constante para registrador é buscada na descrição da máquina na respectiva entrada. Os argumentos da montagem desta instrução serão o registrador alocado e a constante. A instrução é então gerada e enfileirada.

- Na segunda operação são alocados dois registradores para A e B respectivamente. Se uma ou ambas as variáveis já estiverem em outros registradores, estes serão então utilizados. Se não, será necessário carregar uma ou as duas variáveis em seus registradores alocados. Será necessário então gerar uma ou duas instruções do tipo $Reg_{Var} \leftarrow [Endereço_{Var}]$, onde Var é a variável que teve o registrador alocado. Uma instrução de carregamento de variáveis para registrador (*load*) é buscada na descrição da máquina na respectiva entrada e devidamente codificada e enfileirada. Em seguida, uma operação do tipo $Reg_1 \leftarrow Reg_1 + Reg_2$ terá que ser gerada. Uma instrução de soma entre registradores é buscada na descrição da máquina na entrada correspondente ao operador “+”. Os argumentos da montagem desta instrução serão os dois registradores associados às variáveis. A instrução é então gerada e enfileirada.
- A terceira operação é decodificada de acordo com o operador. O operador “[]” calcula um endereço de memória (valor-L) em relação a T_0 a partir de um deslocamento gerado por T_1 . Como ambos os operandos são temporários, quer dizer que eles correspondem a resultados de operações anteriores. Desta forma, o endereço é computado com base nos resultados prévios. O cálculo do endereço é feito com base na fórmula $End_{final} = End(T_1) + End(T_0) * sizeof(Tipo(T_0))$.

Desta forma serão necessárias duas instruções para implementar a operação. A primeira do tipo $Reg \leftarrow Reg * Constante$, onde o registrador de destino é o mesmo que o registrador envolvido na operação e corresponde ao registrador do resultado da operação T_1 . A constante corresponde ao tamanho do objeto que é determinado em tempo de compilação. O tamanho do objeto é encontrado na informação de tipo que foi associada à operação T_0 . A instrução de máquina que implementa esta instrução é buscada e codificada. A segunda instrução será do tipo $Reg_1 \leftarrow Reg_1 + Reg_2$, onde Reg_1 um é o registrador associado à operação T_0 e Reg_2 é o registrador do resultado da primeira instrução gerada ($Reg \leftarrow Reg * Constante$). A instrução de máquina que implementa esta instrução é buscada e codificada. Neste ponto, Reg_1 e o tipo da operação T_0 (que é o tipo de A) são associados à operação T_2 para uso futuro.

- A quarta operação é decodificada. Novamente o operador “.” está envolvido. Mas nesta situação, o primeiro operando é o resultado da última operação (T_2). O resultado da operação é novamente um endereço (valor-L) que será calculado entre os dois operandos. Como o primeiro operando é o resultado de uma operação prévia, o endereço associado correspondente representa a base de cálculo. Desta forma precisa-se apenas somar a este endereço o deslocamento gerado pelo operando D . Para isto será gerada uma instrução do tipo $Reg \leftarrow Reg + Constante$. Onde Reg é o resultado da operação T_2 e $Constante$ é o deslocamento gerado por D . Tal deslocamento pode ser obtido em tempo de compilação subtraindo-se o endereço relativo de D sobre o tipo associado à operação T_2 , que seguindo-se para trás, é o tipo da operação T_0 que por si é o tipo de A (tipo *Struct1*). Neste caso o endereço relativo de D sobre a base da estrutura é 0 (pois é o único componente da estrutura). Desta forma não é

necessário gerar nenhuma instrução. Mas se o endereço relativo fosse diferente de 0, a instrução deveria ser buscada e codificada.

- A quinta operação segue os procedimentos descritos para a primeira operação (T_0). Neste caso, o registrador associado ao resultado da operação T_4 conterá o endereço do campo E .
- Na sexta operação é feita a atribuição da expressão. Neste ponto, T_3 e T_4 possuem um endereço de memória (valor-L). O valor existente no endereço de memória de T_4 será copiado para o endereço de T_3 . Neste caso serão geradas duas instruções. Uma de carregamento de memória para registrador por endereçamento por registrador ($Reg_{T4} \leftarrow [Reg_{T4}]$), e uma de registrador para memória também por endereçamento por registrador ($[Reg_{T3}] \leftarrow Reg_{T3}$). As instruções são então buscadas e codificadas.

Visto que o HTG encapsula em nós próprios todas as hierarquias do código, a geração de código das instruções de desvios condicionais e incondicionais e de todo o resto a partir do grafo fica bastante facilitada.

A decomposição de agregados ao nível primitivo ou *array* poderia eliminar duas das seis instruções de três endereços geradas para o exemplo. As instruções geradas seriam:

1: $T_1 = B + C$	4: $T_3 = T_2.D$
2: $T_2 = \underline{S.A} [] T_1$	5: $T_3 = \underline{S.E}$
3: $T_3 = T_2.D$	

Figura 4.25 – Instruções de três geradas para a expressão $S.A[B+C].D = S.E$ com o processo de decomposição de *structs* ativado.

Observa-se que o ponto nos símbolos sublinhados não representa mais o respectivo

operador, sendo colocado apenas como parte do nome do novo símbolo criado.

O motor de alocação de registros utiliza um algoritmo de geração de código semelhante ao algoritmo descrito por *Aho et al* [Aho et al. 1986]. No caso de necessidade de alocação de um registrador que esteja ocupado, é dada prioridade na alocação de registradores que contenham constantes ou registradores com dados já utilizados várias vezes, nesta ordem.

Até o presente estado de implementação do gerador de código não foi considerada nenhuma otimização especial. Otimização *Peephole*, eliminação de sub-expressões comuns, propagação de constantes, entre outras otimizações, foram reservadas para trabalhos futuros. Como a representação das operações da linguagem C foi feita através de instruções de três endereços, a otimização *Peephole* pode ser executada sobre a mesma. Tanenbaum et al [Tanenbaum et al. 1982] faz uma discussão sobre a aplicação desta otimização sobre operações representadas na forma de instruções de três endereços. As outras otimizações podem ser também facilmente aplicadas sobre esta representação.

4.6.5.3 Simulador para o Nios II

A fim de verificação do código gerado para o Nios II, foi criado um simulador funcional para o processador. Uma simulação funcional de um processador simula a decodificação e execução de suas instruções. Esta simulação não atinge o nível do *hardware*, não simulando desta forma o funcionamento dos componentes do processador. Visto que o Nios II é um processador RISC, seu conjunto de instruções se apresenta simples, facilitando a decodificação das instruções em *software*. A maioria das instruções do processador podem ser simuladas. Somente as instruções de manipulação de *cache* não fazem sentido em uma simulação deste tipo. A depuração do código no simulador aliada à análise do código de montagem gerado durante a compilação se mostrou suficiente para a verificação da corretude do código. Testes no FPGA foram reservados para trabalhos futuros.

4.7 Conclusões

Neste capítulo foi apresentado o *framework Phoenix*, parte do projeto ARCHITECT+, e o processador Nios II.

O compilador foi desenvolvido em C++ em um projeto orientado a objetos visando fácil extensão. Foi construída uma base para seu desenvolvimento que consiste de classes parametrizadas que implementam as estruturas de dados mais comuns, como *arrays*, filas, listas, pilhas, tabelas *hash* e árvores, e também funções comumente utilizadas nestes tipos de estruturas de dados como, por exemplo, busca binária. Esta base permitiu um grau elevado de velocidade na implementação do compilador, sem perda de eficiência do código.

Phoenix gera uma representação intermediária que é utilizada na geração de código nativo e que apresenta boa parte das estruturas necessárias para se implementar a síntese de alto nível de circuitos digitais. Neste aspecto, a utilização do grafo HTG como fonte de representação de alto nível de uma função mostrou-se muito útil na geração de código dos desvios condicionais e incondicionais existentes no programa. Além disto, a representação criada e utilizada na representação de tipos na linguagem C mostrou-se extremamente simples nos aspectos de visualização e implementação e eficaz.

O compilador permite um suporte maior quanto às construções da linguagem C para o processo de síntese de alto nível de circuitos digitais, implementando um recurso que permite decompor automaticamente agregados de dados em variáveis simples ou *arrays*. Desta forma, permitiu-se maior completude quanto o suporte à linguagem e maior adequação à criação da forma SSA e aos algoritmos de análise de fluxo de dados tradicionais.

O compilador permite gerar código para processadores utilizando-se de um modelo de geração de código para múltiplos alvos. Este modelo consiste na descrição dos atributos da máquina em uma linguagem de alto nível para encaixe com a interface de vanguarda e geração do código específico. Uma linguagem de descrição de máquinas foi construída

especificamente para o compilador. Ela foi construída para fornecer a especificação da arquitetura de duas máquinas, uma RISC (Nios II) e uma CISC (Pentium IV). A linguagem de descrição de máquinas se mostrou bastante simples e legível para o processo de descrição. Até o momento, a geração de código nativo foi implementada e verificada para o processador Nios IITM da Altera.

Capítulo 5

Resultados

O *Phoenix* é capaz de gerar uma representação intermediária com suporte aos processos de síntese de alto nível de circuitos digitais e geração de código para processadores. A decomposição de *structs* ao nível primitivo ou *array* permite que tais estruturas de dados possam ser automaticamente utilizadas quando a compilação de circuitos vier a ser implementada. Tal recurso é comparado em relação a outros compiladores na tabela 1.1. A geração de código para múltiplos alvos garante o suporte por parte do compilador para futuros processadores da linha Nios e também outros processadores CISC e RISC.

Testes feitos com o *framework* mostraram que ele gera adequadamente todas as estruturas descritas.

A geração de código nativo sofre de certa perda de performance devido a utilização da geração de código para múltiplos alvos. Existe neste caso uma sobrecarga de execução no mapeamento das instruções de três endereços para as instruções de máquina durante a geração de código. Se a geração de código tivesse sido implementada especificamente para uma CPU qualquer, esta sobrecarga de execução relativa a este processo de mapeamento não existiria.

Com fins de comparação com ferramentas de compilação já existentes, abaixo segue algumas comparações entre o *Phoenix* e o GCC:

1. *Phoenix* constitui-se hoje de aproximadamente 1 mega byte de código fonte, contra os 43 mega bytes do GCC.
2. O código do *Phoenix* se constitui de aproximadamente 32000 linhas de código, de onde aproximadamente 8800 linhas são relativas aos analisadores léxicos e

sintáticos gerados pelas ferramentas Flex e Bison para a linguagem C e a linguagem de descrição de máquinas construída.

3. *Phoenix* é orientado a objetos enquanto que GCC não.
4. *Phoenix* possui código simples e mais legível, utilizando-se raramente de macros, ao contrário do GCC.
5. *Phoenix* representa as operações da linguagem em um formato tradicionalmente utilizado pela comunidade de compiladores (instruções de três endereços).
6. *Phoenix* gera o grafo de hierarquia de tarefas para o processo de síntese de alto nível, ao contrário do GCC.
7. *Phoenix* implementa geração de código nativo visando múltiplos alvos assim como o GCC. Mas no caso do *Phoenix*, o encaixe da vanguarda com a retaguarda pode ser feito em tempo de execução, enquanto que no GCC o encaixe é feito em tempo de compilação da ferramenta.
8. *Phoenix* implementa o processo de decomposição de *structs* para suporte destes tipos de agregados de dados para o processo de síntese de alto nível de circuitos digitais.

Obviamente, não se pode desconsiderar o valor do GCC. Ele é uma ferramenta excelente no que tange seu propósito: fácil criação de um compilador para uma determinada CPU a partir de uma das linguagens suportadas. Entretanto sua utilização no projeto ARCHITECT+ não se mostrou viável, visto a complexidade que emerge em se modificar seu código fonte para suprir as necessidades exigidas pelo projeto. Desta forma a criação de um *framework* de compilação sem reutilização de qualquer outro software foi necessária.

Para ilustração de alguns resultados do *Phoenix*, segue abaixo alguns exemplos de pequenos programas C e as respectivas estruturas geradas pelo compilador.

5.1 Exemplo 1

As figuras 5.2 e 5.3 mostram, respectivamente, o grafo HTG e o grafo CFG criados pelo *Phoenix* para o programa da figura 5.1. O programa exemplo possui algumas hierarquias permitidas pela linguagem C.

```
void funcao( int Arg )
{
    char A, B, C, D, E;

    A = 1;
    E = A;
    while ( A > 0 )
    {
        if ( B > 5 )
        {
            B = 5;
        }
        else
        {
            B = 10 + D + Arg;
        }
        C = B / 5;
        A = A + 15;
    }
}
```

Figura 5.1 – Programa exemplo para a geração do CFG e HTG.

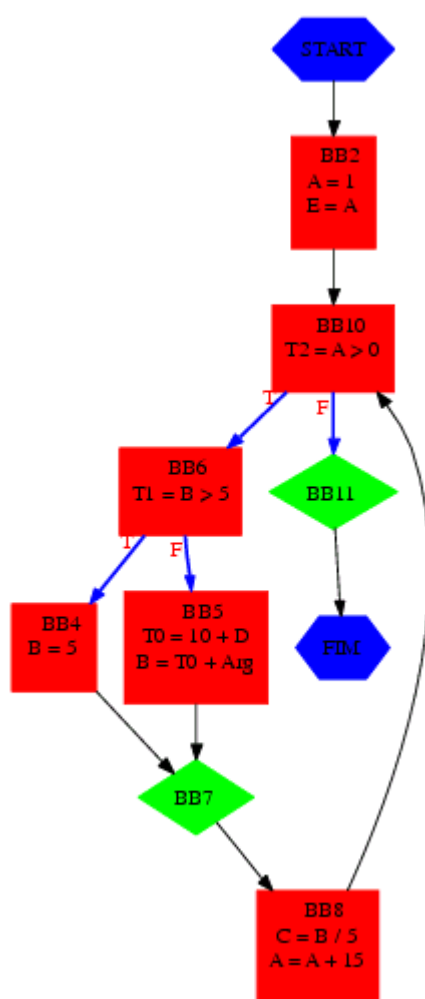


Figura 5.2 – Grafo CFG do programa da figura 5.1.

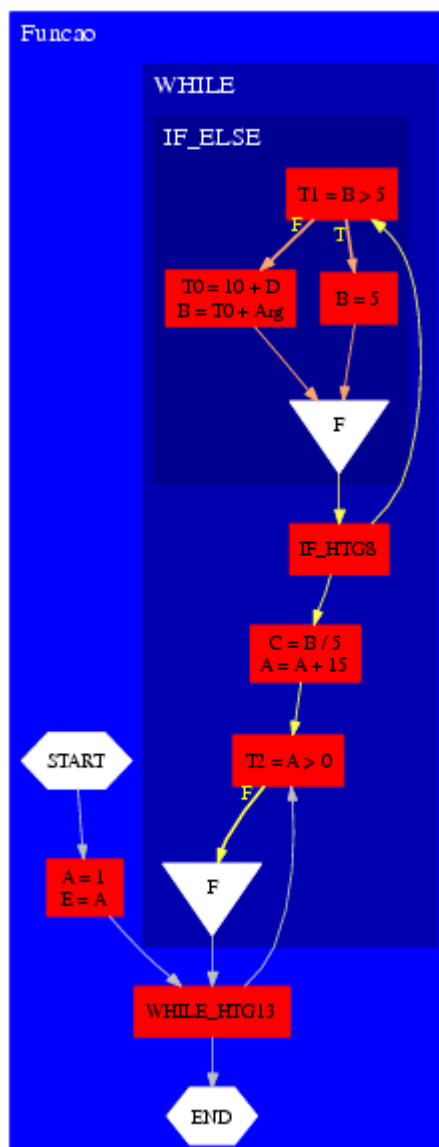


Figura 5.3 – Grafo HTG do programa da figura 5.1.

	subi	sp, sp, 1	:: Prólogo
	stw	fp, 0(sp)	:: Prólogo
	mov	fp, sp	:: Prólogo
	sub	sp, sp, 6	:: Prólogo
:: 16			
	ldb	r8, -5(fp)	:: r8 = B
	movi	r9, 10	:: r9 = 10
	add	r8, r8, r9	:: r8 = (A = B+10)
	stb	r8, -6(fp)	:: Store A
:: 32			
	ldb	r10, -4(fp)	:: r10 = C
	movi	r11, 50	:: r11 = 50
	mul	r10, r10, r11	:: r10 = (D=C*50)
	stb	r10, -3(fp)	:: Store D
:: 48			
	ldb	r12, -2(fp)	:: r12 = E
	movi	r13, 0	:: r13 = 0
	cmpeq	r12, r12, r13	:: r12 = (E == 0)
:: 60			
	movi	r14, 0	:: r14 = 0
	be	r12, r14, 88	:: r12 = ((E==0)==0)
:: 68			
	movi	r15, 40	:: r15 = 40
	div	r8, r8, r15	:: r8 = (E=A/40)
	stb	r8, -2(fp)	:: Store E
:: 80			
	movi	r16, 120	
	jmp	r16	
:: 88			
	ldb	r17, -5(fp)	:: r17 = B
	movi	r18, 10	:: r18 = 10
	div	r17, r17, r18	:: r17 = (E=B/10)
	stb	r17, -2(fp)	:: Store E
:: 120			
	stb	r10, -1(fp)	:: F=D (r10 = D, Store F)
:: 124			
	mov	sp, fp	:: Epílogo
	ldw	fp, 0(sp)	:: Epílogo
	addi	sp, sp, 1	:: Epílogo
	ret		:: Epílogo

Figura 5.4 – Código gerado para o programa da figura 5.1.

5.2 Exemplo 2

As figuras 5.5 e 5.6 mostram, respectivamente o grafo DDG e o grafo CFG com as definições incidentes criados pelo *Phoenix* para o programa da figura 5.4.

```
void funcao()
{
    char A, B, C, D, E, F;

    A = B + 10;
    D = C * 50;
    if ( E == 0 )
        E = A / 40;
    else
        E = B / 10;
    F = D;
}
```

Figura 5.5 – Programa exemplo para a geração do DDG.

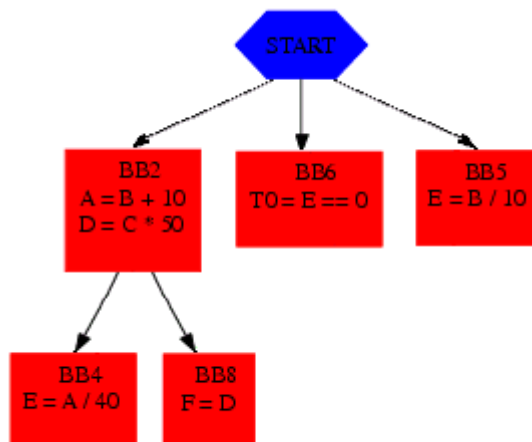


Figura 5.6 – Grafo DDG do programa da figura 5.5.

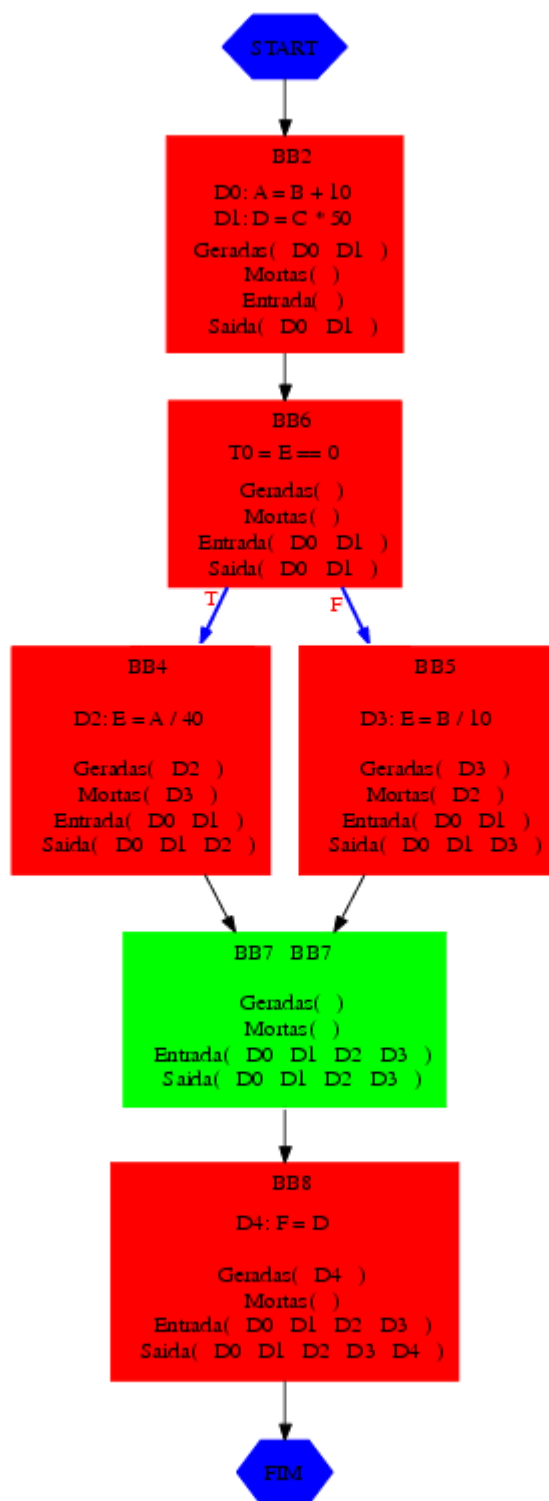


Figura 5.7 – Grafo CFG com as definições incidentes do programa da figura 5.5.

	subi	sp, sp, 1	:: Prólogo
	stw	fp, 0(sp)	:: Prólogo
	mov	fp, sp	:: Prólogo
	sub	sp, sp, 5	:: Prólogo
:: 32			
	movi	r8, 1	:: r8 = 1
	stb	r8, -5(fp)	:: A = r8 = 1
:: 40			
	stb	r8, -1(fp)	:: E = A = r8
:: 44			
	ldb	r9, -5(fp)	:: r9 = A
	movi	r10, 0	:: r10 = 0
	cmpgt	r9, r9, r10	:: r9 = (A>0)
:: 56			
	movi	r11, 0	:: r11 = 0
	be	r9, r11, 156	:: r12 = ((A>0)==0)
:: 64			
	ldb	r12, -4(fp)	:: r12 = B
	movi	r13, 5	:: r13 = 5
	cmpgt	r12, r12, r13	:: r12 = (B>5)
:: 76			
	movi	r14, 0	:: r11 = 0
	be	r12, r14, 100	:: r12 = ((B>5)==0)
:: 84			
	movi	r15, 5	:: r15 = 5
	stb	r8, -4(fp)	:: B = r15 = 5
:: 92			
	movi	r16, 124	
	jmp	r16	
:: 100			
	ldb	r17, -2(fp)	:: r17 = D
	ldw	r18, +4(fp)	:: r18 = Arg
	add	r17, r17, r18	:: r17 = (D+Arg)
	movi	r19, 10	:: r19 = 10
	add	r17, r19, r17	:: r17 = (10+D+Arg)
	stb	r17, -4(fp)	:: B = r17 = (10+D+Arg)
:: 124			
	movi	r20, 5	:: r21 = 5
	div	r17, r17, r20	:: r17 = (B/5)
	stb	r17, -3(fp)	:: C = r17 = (B/5)
:: 136			
	ldb	r21, -5(fp)	:: r21 = A
	addi	r21, r21, 15	:: r21 = A + 15
	stb	r21, -5(fp)	:: A = r21 = (A+15)
:: 148			
	mov	r22, 44	
	jmp	r22	
:: 156			
	mov	sp, fp	:: Epílogo
	ldw	fp, 0(sp)	:: Epílogo
	addi	sp, sp, 1	:: Epílogo
	ret		:: Epílogo

Figura 5.8 – Código gerado para o programa da figura 5.5.

Capítulo 6

Conclusão

Neste trabalho foi apresentada uma proposta de um *framework* de compilação ANSI C denominado *Phoenix*. Esta ferramenta provê suporte para o processo de síntese de alto nível de circuitos digitais e para o processo de geração de código para processadores reais ou virtuais.

A construção do compilador *Phoenix*, disponibilizado para uso do Projeto ARCHITECT+, torna possível a definição de arquiteturas especialmente criadas para aplicações específicas bem como para a geração de código para a execução nestas arquiteturas.

Phoenix gera as estruturas internas necessárias para a transformação de um programa escrito em ANSI C para código HDL que pode ser posteriormente sintetizado em cadeias de configuração de dispositivos FPGA. Sua construção mostrou a viabilidade da produção destes complexos sistemas no ambiente de parceria institucional instalado entre a Faculdade de Computação a UFU e o Instituto de Ciências Matemáticas e de Computação da USP São Carlos.

A principal contribuição deste trabalho é disponibilização de uma ferramenta de arquitetura aberta que permite suporte para Co-Projeto *hardware/software*. O compilador *Phoenix*, no estágio em que se encontra, oferece suporte automático ao construtor *struct* da linguagem C para o processo de síntese de alto nível. Desta forma, agregados de dados deste tipo podem ser utilizados neste processo sem modificações do código fonte por parte do programador. Este suporte expande os recursos tradicionalmente implementados nos compiladores que o antecederam (tabela 1.1). A geração de código para múltiplos alvos

implementada no compilador permite flexibilidade de geração de código para outros processadores bem como para novas versões do processador virtual Nios.

6.1 Trabalhos Futuros

Como trabalhos futuros relativos ao *Phoenix* pode-se destacar:

1. Pesquisar o tratamento de recursos complexos da linguagem C, tais como, tratamento de ponteiros, suporte à recursividade e objetos dinâmicos para o processo de síntese de alto nível.
2. Testar o suporte de objetos dinâmicos com a implementação de um gerenciador de memória em *hardware*.
3. Otimizar os grafos para o processo de síntese de alto nível de circuitos digitais. Tais otimizações podem se basear nos trabalhos de Cardoso [Cardoso 2000] e *Spark* [Gupta *et al.* 2004].
4. Gerar VHDL (ou Verilog) a partir da representação gerada. Cardoso [Cardoso 2000] e Gajski [Gajski 1992] explicam como este processo é feito.
5. Pesquisar a construção de máquinas virtuais (como a JVM) através do *framework* criado.
6. Suportar aos tipos de dados *float* e *double* da linguagem C no código nativo gerado para o Nios II.
7. Executar testes do código gerado no processador Nios II. Para isto será necessário gerar a imagem do programa executável no formato específico exigido para o carregamento do programa no processador.

8. Implementar otimizações tradicionais na geração de código nativo. Entre estas otimizações destacam-se: eliminação de sub-expressões comuns, propagação de cópias, eliminação de código morto e transposição para constantes, movimentação de código ciclo-invariante e otimização *Peephole*. Todas estas otimizações podem ser feitas com base em Aho *et al* [Aho *et al.* 1986].

Referências Bibliográficas

- [Aho *et al.* 1989] Aho, A. V.; Ganapathi, M.; Tjiang, Steven W. K. (1989). *Code Generation Using Tree Matching and Dynamic Programming*. ACM Transactions on Programming Languages and Systems, Vol. 11, No. 4, October, Pages 491-516.
- [Aho *et al.* 1986] Aho, A. V.; Sethi, R.; Ullman, J. D. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass.
- [Aho *et al.* 1977a] Aho, A. V.; Johnson, S. C.; Ullman, J. D. (1977). *Code generation for expressions with common subexpressions*. J. ACM 24, 1 (1977), 146-160.
- [Altera 2005a] Altera Inc. (2005a). *Nios II Processor Reference Handbook*. www.altera.com.
- [Altera 2005b] Altera Inc. (2005b). *Nios II Custom Instruction - User Guide*. www.altera.com.
- [Altera 2006] Altera Inc. (2006). Internet: www.altera.com. Acesso em Jan. 2006.
- [Altera 2005a] Altera Inc. (2005a). *Nios II Processor Reference Handbook*. www.altera.com.
- [Andrew *et al.* 2005] Andrew, A.; *et al.* *The Zephyr Compiler Infrastructure*. <http://www.cs.virginia.edu/zephyr/>. Acesso em Dezembro 2005.
- [Annapolis 1998] Annapolis Microsystems, Inc. (1998). *Wildfire Reference Manual*. Annapolis Microsystems, Inc, Annapolis, MD.
- [Athanas 1992] Athanas, Peter Mark (1992). *An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration*. PhD Thesis, Brown University.
- [Athanas e Silverman 1993] Athanas, P. M.; Silverman, H. (1993). *Processor Reconfiguration through Instruction-Set Metamorphosis: Architecture and Compiler*. IEEE Computer, vol. 26, n. 3, pp. 11-18. Março 1993.
- [Bik e Gannon 1997] Bik, Aart J. C.; Gannon, D. B. (1997). “*Javab - A Prototype Bytecode Parallelization Tool*”. Computer Science Department Technical Report. Indiana University. USA. URL: <http://www.extreme.indiana.edu/hpjava/>, pp. 46.
- [Brown 1996] Brown, S.; Rose, J. (1996). “*PGA and CPLD Architectures: A Tutorial*. IEEE Design & Test of Computers, Summer 1996, Vol. 13, No. 2, pp. 42-57.
- [Callahan *et al.* 2000] Callahan, T. J.; Hauser, J.; Wawrzynek, J. (2000), *The Garp Architecture and C Compiler*. IEEE Computer, Vol.33, No. 4, Abril, pp. 62-69.
- [Cardoso 2000] Cardoso, J. M. P. (2000) *Compilação de Algoritmos em Java TM para Sistemas Computacionais Reconfiguráveis com Exploração do Paralelismo ao Nível das Operações*. Tese de Doutorado. Universidade Técnica de Lisboa.
- [Chameleon 2000] Chameleon Systems, Inc. (2000). *CS2000 Advance Product Specification*. Chameleon Systems, Inc., San Jose, CA.

- [Chang *et al.* 1991] Chang, P. P.; *et al* (1991). *IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors*. Proceedings of the 18th Annual Int'l Symposium on Computer Architecture. Toronto, Canada. 28(5): 266-275.
- [Chichkov e Almeida 1997] Chichkov, A. V.; Almeida, C. B. (1997). *An Hardware/Software Partitioning Algorithm for Custom Computing Machines*. Published in Springer-Verlag Lecture Notes in Computer Science 1304, W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Berlin, Germany, pp.274-283.
- [Chichkov 1998] Chichkov, A. V, *Metodologia de Co-Projecto Hardware/Software Para Arquitecturas Computacionais Reconfiguráveis Utilizando uma Partição Baseada no Paralelismo Implícito*, Dissertação para a obtenção do grau de doutor em Engenharia Electrotécnica e de Computadores, Instituto Superior Técnico (IST) da Universidade Técnica de Lisboa (UTL), Lisboa. Janeiro.
- [Compton e Hauck 2000] Compton, K.; Hauck, S. (2000). *Reconfigurable computing: A survey of systems and software*. ACM Computing Surveys, 34(2): 171–210.
- [Compton e Hauck 2000b] Compton, K.; Hauck, S. (2000). *An Introduction to Reconfigurable Computing*. IEEE Computer, April.
- [Cytron *et al.*, 1991] Ron Cytron, *et al.* (1991). *Efficiently Computing static single assignment form and the control dependence graph*. In *ACM Transactions on Programming Languages and Systems*, 13(4), October, pp. 451-490.
- [DeRemer 1971] DeRemer, F. (1971). *Simple LR(k) grammars*. *Comm. ACM* 14:7, 453-460.
- [DeRemer 1969] DeRemer, F. (1969). *Practical Translators for LR(k) Languages*. Ph. D. Thesis, M.I.T, Cambridge, Mass.
- [Estrin *et al.* 1963] Estrin, G.; Bussel, B.; Turn, R.; Bibb, J. (1963). *Parallel processing in a restructurable computer system*. *IEEE Trans. Elect. Computing*. 747–755.
- [Fraser e Hanson 1995] Fraser, C. W.; Hanson, D. R (1995). *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub Co, Redwood City, CA, USA.
- [Gajski 1992] Gajski, D. *et al.* (1992). *High-Level Synthesis, Introduction to Chip and System Design*. Kluwer Academic Publishers.
- [Galloway 1995] Galloway D. (1995). *The Transmogripher C Hardware Description Language and Compiler for FPGAs*. In *Proc. of the 3rd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'95)*, Napa Valley, CA, USA, April 19-21, 1995, Published by the IEEE Computer Society Press, Peter Athanas and Kenneth L. Pocek, editors, Los Alamitos, CA, USA, pp. 136-144.
- [Girkar e Polychronopoulos 1992] Girkar, M.; Polychronopoulos, C. D. (1992). *Automatic extraction of functional parallelism from ordinary programs*. *IEEE Trans. on Parallel & Distributed Systems*. Março.
- [Graphviz 2005] AT&T Research Labs. *Graphviz - Open source graph drawing software*. <http://www.research.att.com/sw/tools/graphviz/>. Acesso em Julho. 2005.

- [Guilan *et al.* 2002a] Guilan, Dai; Suqing, Zhang; Jinlan, Tian; Weidu, Jiang (2002). *A Study of Compiler Techniques for Multiple Targets in Compiler Infrastructures*. ACM SIGPLAN Notices, 2002, 36(7): 45~51.
- [Guilan *et al.* 2002b] Guilan, Dai; Suqing, Zhang; Jinlan, Tian; Weidu, Jiang; Jun, Dai; (2002). *Retargetable Cross Compilation Techniques - Comparison and Analysis of GCC and Zephyr*. ACM SIGPLAN Notices, 2002, 37(6): 38~43.
- [Gupta *et al.* 2004] Gupta, S.; Gupta, R.; Dutt, N.; Nicolau, A. (2004). *Spark: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers. 234 p.
- [Gupta *et al.* 2003] Gupta, S.; Gupta, R.; Dutt, N.; Gupta, R.; Nicolau, A. (2003). *SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations*. In International Conference on VLSI Design.
- [Hauck 1998] Hauck S. (1998). *The Roles of FPGAs in Reprogrammable Systems*. In *Proceedings of the IEEE*, Vol. 86, No. 4, pp. 615-638, April.
- [Hauck *et al.* 1997] Hauck, S.; Fry, T. W.; Hosler, M. M.; Kao, J. P. (1997). *The Chimaera Reconfigurable Functional Unit*. In *Proc. of the 5th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'97)*, Napa Valley, CA, USA, April 16-18, 1997, Published by the IEEE Computer Society Press, Kenneth L. Pocek and Jeffrey Arnolds, editors, Los Alamitos, CA, USA, pp. 87-96.
- [Impulse 2006] Impulse Accelerated Technologies, Inc. (2006). Internet: www.impulse.com. Acesso em Jan. 2006.
- [Intel 1997c] Intel Inc. (1997c). *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*. www.intel.com.
- [Johnson 1980] Johnson, S. C. (1980). *A Tour Through The Portable C Compiler (1980)*. UNIX Programmer's Manual. U.C. Berkeley.
- [Johnson 1975] Johnson, S. C. (1975). *Yacc – yet another compiler compiler*. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J.
- [Laufer *et al.* 1999] Laufer, R.; Taylor, R. R.; Schmit, H. (1999). *PCI-PipeRench and the SwordAPI: A system for stream-based reconfigurable computing*. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 200–208.
- [Lesk 1975] Lesk, M. E. (1975). *Lex – a lexical analyzer generator*. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J.
- [Mahlke *et al.* 1992] Mahlke, S. A.; Lin, D. C.; Chen, W. Y.; Hank, R. E.; Bringmann, R. A. (1992). *Effective Compiler Support for Predicated Execution Using the Hyperblock*. In *Proc. of the 25th International Symposium on Microarchitecture*, Dez., pp. 45-54.
- [Nicolau e Novack 1993] Nicolau, A.; Novack, S. (1993). *Trailblazing: A hierarchical approach to Percolation Scheduling*. In *Internacional Conference on Parallel Processing*.
- [Novack e Nicolau 1996] Novack, S.; Nicolau, A. (1996). *An efficient, global resource-*

directed approach to exploiting instruction level parallelism. In Conference on Parallel Architectures e Compilation Techniques.

- [Quickturn 1999a] Quickturn, A Cadence Company. (1999). *System Realizer*™. Disponível em <http://www.quickturn.com/products/systemrealizer.htm>. Quickturn, A Cadence Company, San Jose, CA.
- [Quickturn 1999b] Quickturn, A Cadence Company.. (1999). *Mercury*™ *Design Verification System Technology Backgrounder*. Disponível em http://www.quickturn.com/products/mercury_backgrounder.htm. Quickturn, A Cadence Company, San Jose, CA.
- [Razdan e Smith 1994] Razdan, R.; Smith, M. D. (1994). *A high performance micro architecture with hardware programmable functional units*. In *International Symposium on Microarchitecture*, 172–180.
- [Russel *et al.* 2004] - Russel, S.; Norvig, P. (2004). *Artificial Inteligence: A Modern Approach*. 2 Ed. Prentice Hall. p 113.
- [Stallman 2001] Stallman, R. M. (2001). *Using and Porting GNU Compiler Collection*. Free Software Foundation, Inc.
- [Tanenbaum *et al.* 1982] Tanenbaum, A. S.; Staveren, V. H.; Stevenson, J. W. (1982). *Using peephole optimization on intermediate code*. *TOPLAS* 4:1, 21-36.
- [Vuillemin *et al.* 1996] Vuillemin, J.; Bertin, P.; Roncin, D.; Shand, M.; Touati, H.; Boucard, P. (1996). *Programmable active memories: Reconfigurable systems come of age*. *IEEE Trans. VLSI Syst.* 4, 1, 56–69.
- [Wittig e Chow 1996] Wittig, R. D.; Chow, P. (1996). *OneChip: An FPGA processor with reconfigurable logic*. *IEEE Symposium on FPGAs for Custom Computing Machines*, 126–135.
- [Xilinx 2006] Xilinx Inc. (2005). Internet: www.xilinx.com. Acesso em Jan. 2006.

Bibliografia

- [Abid *et al.* 1996] Abid, M.; Changuel, A.; Jerraya, A. (1996). *Exploration of Hardware/Software Design Space through a Codesign of Robot Arm Controller*. IEEE 0-89791-848-7/96.
- [Aho e Ganapathi 1985] Aho, A. V.; Ganapathi, M. (1985). *Efficient tree pattern matching: An aid to code generation*. In Proceedings of the 12th ACM Symposium on Principles of Programming Languages. ACM, New York, 1985, pp. 334-340.
- [Aho e Johnson 1976] Aho, A. V.; Johnson, S. C. (1976). *Optimal code generation for expression trees*. J. ACM 23, 3 (1976), 488-501.
- [Aho *et al.* 1977b] Aho, A. V.; Johnson, S. C.; Ullman, J. D. (1977). *Code generation for machines with multi-register operations*. In Proceedings of the 4th ACM Symposium on Principles of Programming Languages. ACM, New York, pp. 21-28.
- [Aho *et al.* 1975] Aho, A. V.; Johnson, S. C.; e Ullman, J. D. (1975). *Deterministic parsing of ambiguous grammars*. Commun. ACM 18,s August., pp 441-452.
- [Barth 1977] Barth, J. M. (1977). *An Interprocedural Data Flow Analysis Algorithm+*. 1977 ACM 0-12345-678-9
- [Bruno e Lassagne, 1975] Bruno, J. L.; Lassagne, T. (1975). *The Generation of Optimal Code for Stack Machines*. Journal of the Association for Computing Machinery, Vol. 22, No. 3, July 1975, pp. 382-396.
- [Burke e Cytron 1986] Burke, M.; Cytron, R. (1986). *Interprocedural Dependence Analysis and Parallelization*. ACM 0-89791-197-O/86/0600-0162
- [Cardoso 1999a] Cardoso, J. M. P.; Neto, H. C. (1999). *Macro-Based Hardware Compilation of JavaTM Bytecodes into a Dynamic Reconfigurable Computing System*, In Proc. of the IEEE 7th Symposium on Field-Programmable Custom Computing Machines (FCCM'99), Napa Valley, CA, USA, April 21-23, 1999, Published by the IEEE Computer Society Press, Kenneth L. Pocek and Jeffrey Arnolds, editors, Los Alamitos, CA, USA, pp. 2-11.
- [Cardoso 1999b] Cardoso, J. M. P.; Neto, H. C. (1999). *Fast Hardware Compilation of Behaviors into an FPGA-Based Dynamic Reconfigurable Computing System*, In Proc. of the XII Symposium on Integrated Circuits and Systems Design (SBCCI'99), (Co-) Sponsored by the Brazilian Computer Society and the IFIP WG 10.5, Natal-RN, Brazil, Sept. 29-Oct. 2, 1999, In Vladimir C. Alves, Marcelo Lubaszewski, and Ivan S. Silva (Editors), IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 150-153.
- [Cardoso 1999c] Cardoso, J. M. P.; Neto, H. C. (1999). *An Enhanced Static-List Scheduling Algorithm for Temporal Partitioning onto RPU's*, In Proc. of the IFIP TC10 WG10.5 X International Conference on Very Large Scale Integration (VLSI'99), Lisbon, December 1-3, 1999. *VLSI: Systems on a Chip*, Luis M. Silveira, Srinivas Devadas and Ricardo Reis (Editors), Kluwer Academic Publishers, pp. 485-496.

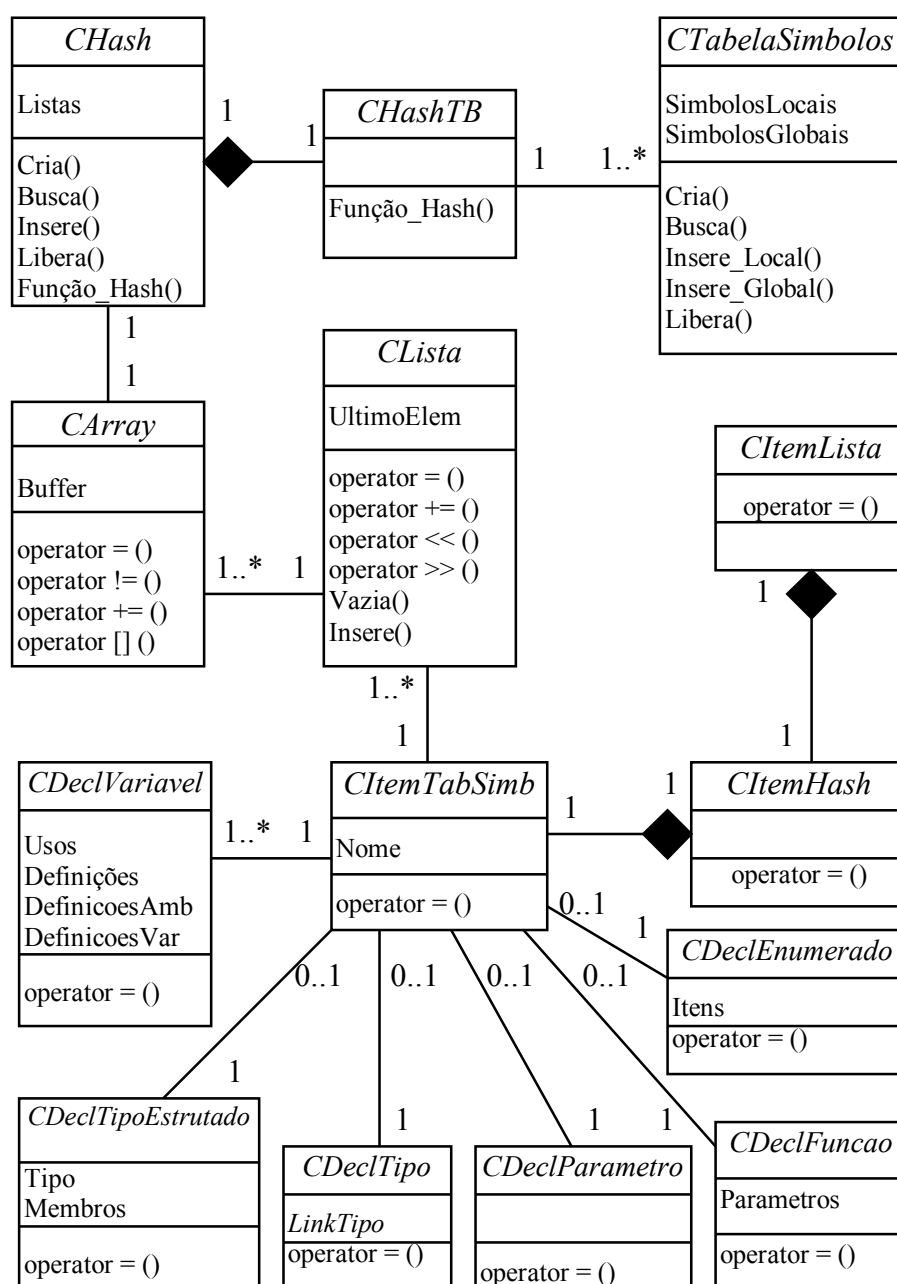
- [Cardoso 1999d] Cardoso, J. M. P.; Neto, H. C. (1999). *The Library of macro-cells used by NENYA: Circuit generators for the Xilinx XC6000 FPGA series*, INESC Technical Report, December 1998, revised April.
- [Cardoso 1999e] Cardoso, J. M. P.; Véstias, M. P. (1999). *Architectures and Compilation Techniques to Support the Reconfigurable Computing Concept*, In Crossroads, the Association for Computing Machinery (ACM) Student Magazine, topic: Computer Architectures, Spring 1999, Issue 5.3, pp. 15-22. Version online at: <http://turing.acm.org/crossroads/xrds5-3/rconconcept.html>.
- [Cardoso 1998a] Cardoso, J. M. P.; Neto, H. C. (1999). *An Approach to Hardware Synthesis from a System Java(tm) Specification*, In *Proc. of the 1st International Workshop on Design, Test and Applications (WDTA'98)*, Sponsored by IEEE region 8 and IEEE TTTC, ISBN 953-184-009-1, Dubrovnik, Croatia, June 8-10, pp. 149-152.
- [Cardoso 1998b] Cardoso, J. M. P.; Neto, H. C. (1999). *Towards an Automatic Path from Java(tm) Bytecodes to Hardware Through High-Level Synthesis*, In *Proc. of the 5th IEEE International Conference on Electronics, Circuits and Systems (ICECS-98)*, Lisbon, Portugal, September 7-10, Vol. 1, pp. 85-88.
- [Cartwright e Felleisen 1989] Cartwright, R.; Felleisen, M. (1989). *The Semantics of Program Dependence*. ACM 0-89791-306-X/89/CCG5/0013
- [Edwards 2005] Edwards, S. A. (2005). *The Challenges of Hardware Synthesis from C-like Languages*. Department of Computer Science, Columbia University, New York. IEEE 1530-1591/05.
- [Ferrante et al. 1987] Ferrante, J.; Ottenstein, K. J.; Warren, J. D. (1987). *The Program Dependence Graph and Its Use in Optimization*. ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3, July, 319-349.
- [Ganapathi et al. 1982] Ganapathi, M.; Fischer C. N.; Hennessy, J. L. (1982). *Retargetable Compiler Code Generation*. ACM Computing Surveys. Vol. 14, No. 4, December.
- [Gupta 2003] Gupta, S. (2003). *Coordinated Coarse-Grain and Fine-Grain Optimizations for High-Level Synthesis*. PhD thesis, University of California, Irvine.
- [Gupta et al. 2003b] Gupta, S.; Dutt, N.D.; Gupta, R.K.; Nicolau A. (2003) *Dynamic conditional branch balancing during the highlevel synthesis of control-intensive designs*. In *Design, Automation and Test Conference*.
- [Gupta et al. 2003c] Gupta, S.; Dutt, N.D.; Gupta, R.K.; Nicolau A. (2003) *Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits*. *Invited Paper in Special Issue of IEE Proceedings: Computers and Digital Technique: Best of DATE 2003*.
- [Gupta et al, 2002a] Gupta, S, Reshadi, M.; Savoiu, N.; Dutt, N.D.; Gupta, R.K.; Nicolau A. (2002). *Dynamic common sub-expression elimination during scheduling in high-level synthesis*. In *International Symposium on System Synthesis*.
- [Gupta et al. 2002b] Gupta, S.; Savoiu, N.; Dutt, N.D.; Gupta, R.K.; Nicolau A. (2002). *Using global code motions to improve the quality of results for high-level synthesis*. Technical

Report CECS-TR-02-29, Center for Embedded Computer Systems, Univ. of California, Irvine.

- [Gupta *et al.* 2001a] Gupta, S.; Savoiu, N.; Kim, S.; Dutt, N.D.; Gupta, R.K.; Nicolau A. (2001). *Speculation techniques for high level synthesis of control intensive designs*. In *Design Automation Conference*.
- [Gupta *et al.* 2001b] Gupta, S.; Savoiu, N.; Dutt, N.D.; Gupta, R.K.; Nicolau A. (2001). *Conditional speculation and its effects on performance and area for high-level synthesis*. In *International Symposium on System Synthesis*.
- [Gyllenhaal 1991] Gyllenhaal, J. C. (1991). *A Machine Description Language for Compilation*. Master's Degree Thesis.
- [Hauck e Agarwal 1996] Hauck S.; Agarwal, A. (1996). *Software Technologies for Reconfigurable Systems*. Northwestern University. Dept. of ECE, Technical Report.
- [Intel 1997a] Intel Inc. (1997a). *Pentium® Processor Family Developer's Manual*. www.intel.com.
- [Intel 1997b] Intel Inc. (1997b). *Intel Architecture Optimization Manual*. www.intel.com.
- [Lengauer e Tarjan 1979] Lengauer, T.; Tarjan, R. E. (1979). *A Fast Algorithm for Finding Dominators in a Flowgraph*. ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, July.
- [Levine *et al.* 1995] Levine, John R.; Mason, Tony; Brown, Doug (1995). *Lex e Yacc*. O'Reilly.
- [Minchew e Tai 1982] Minchew, C. H.; Tai, Kuo-Chung (1982). *Experience With Porting the Portable C Compiler*. North Carolina State University. ACM.
- [Myers 1981] Myers, E. M. (1981). *A Precise Inter-Procedural Data Flow**. ACM 0-89791-029-X.
- [Navabi 1993] Navabi, Z (1993). *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill.
- [Ramsey *et al.* 2001?] Ramsey, N.; Davidson, J. W.; Fernández, M. F. (2000?). *Design Principles for Machine-Description Languages*.
- [Sethi e Ullman 1970] Sethi, R.; Ullman, J. D. (1970). *The Generation of Optimal Code for Arithmetic Expressions*. Journal of the Association for Computing Machinery, Vol. 17, No. 4, October, pp. 715-728.
- [So *et al.* 2002] So, B.; Hall, M. W.; Diniz, P. C. (2002). *A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems*. ACM 1-58113-463-0/02/0006.
- [Wegman e Zadeck 1991] Wegman, M. N.; Zadeck, F. K. (1991). *Constant Propagation with Conditional Branches*. ACM Transactions on Programming Languages and Systems, Vol. 13, No. 2, April, Pages 181-210.

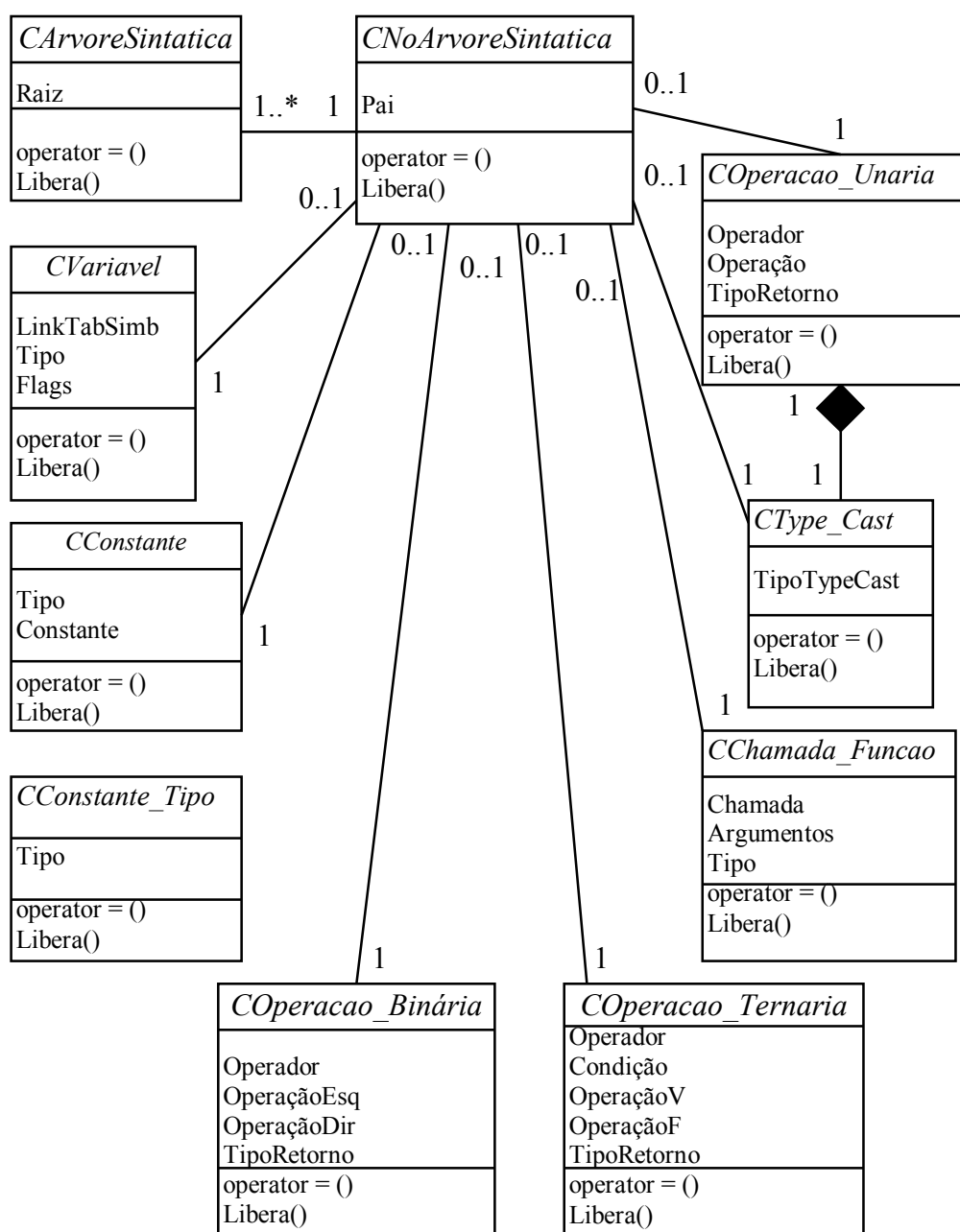
Apêndice 1

Diagrama de Classes da Tabela de Símbolos do *Phoenix*



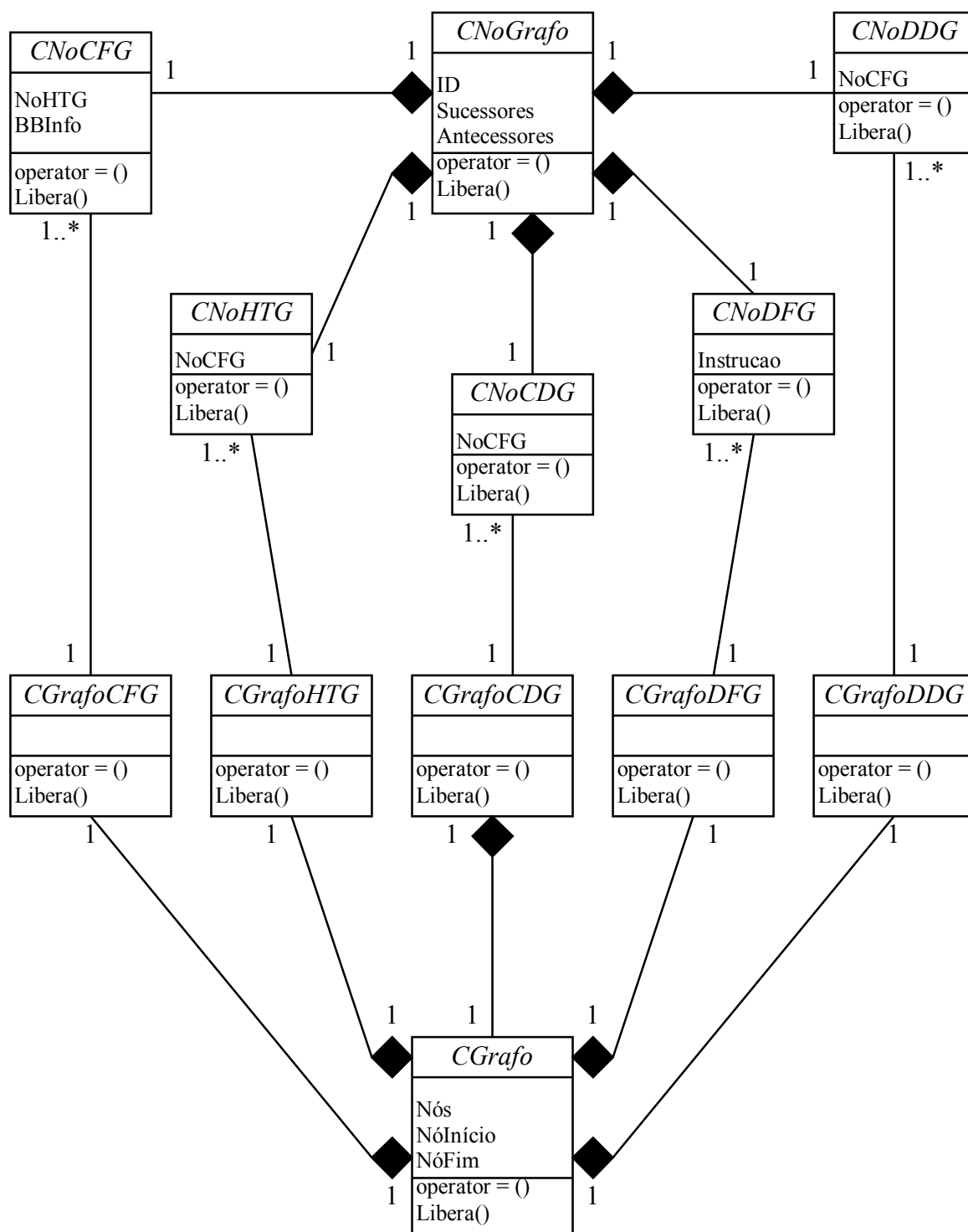
Apêndice 2

Diagrama de Classes das Árvores Sintáticas do *Phoenix*



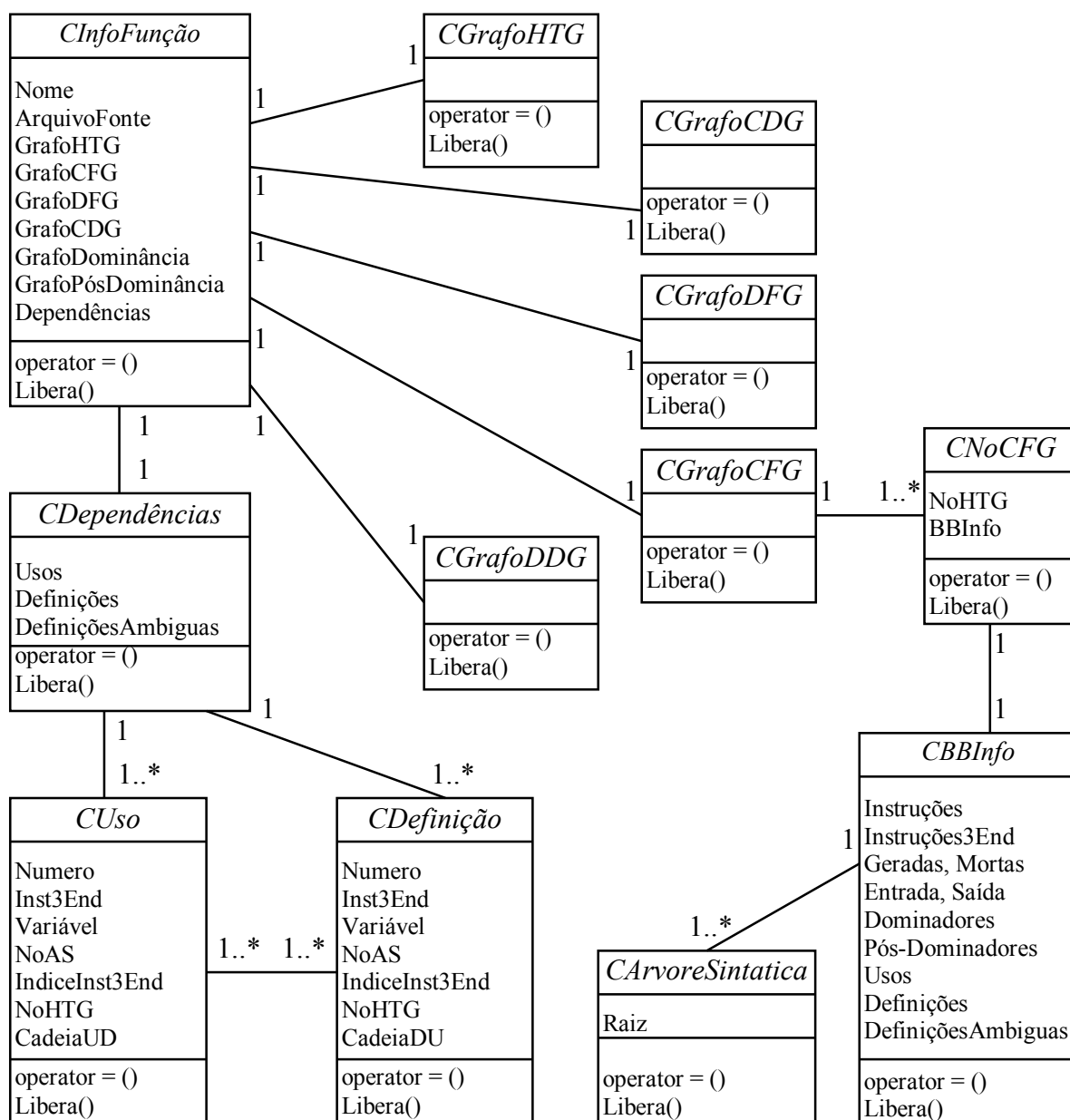
Apêndice 3

Diagrama de Classes da Representação Intermediária do *Phoenix*



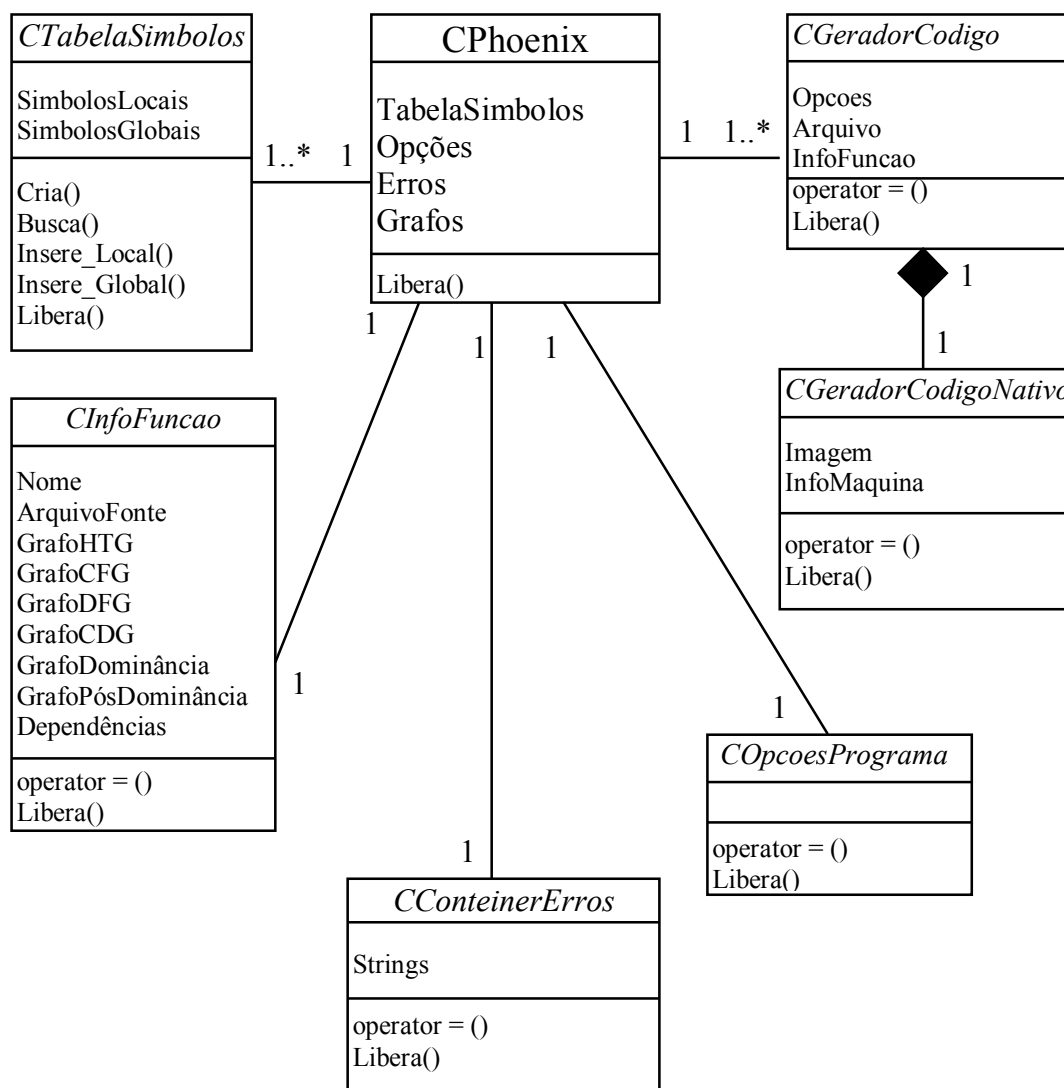
Apêndice 4

Diagrama de Classes da Representação Intermediária do *Phoenix* (continuação)



Apêndice 5

Diagrama de Classes do *Phoenix*



Apêndice 6

Descrição da Arquitetura do Nios II

```

INSTRUCTION_SIZE 32    ;; Em bits
WORD_SIZE        32    ;; Em bits
ADDRESS_RANGE    32    ;; Em bits
TOTAL_REGS       32    ;; Em bits
REGS_RANGE       5     ;; 5 bits para cada registrador

REGS              { R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15,
                   R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29,
                   R30, R31
                   }

STACK_POINTER     R27
BASE_POINTER      R28

;; Registradores específicos
GENERAL_REGS      { R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21,
                   R22, R23 }
TEMP_REGS         { R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21,
                   R22, R23 }
ARGS_REGS         { R4, R5, R6, R7 }
RETURN_REGS       { R2, R3 }
CALLER_SAVED_REGS { R8, R9, R10, R11, R12, R13, R14, R15 }
EXCEPTION_RETURN_REG R29
RETURN_ADRESS_REG  R31

;; Alinhamento dos dados
CHAR_ALIGN        8
SHORT_ALIGN       16
INT_ALIGN         WORD_SIZE
LONG_ALIGN        WORD_SIZE
FLOAT_ALIGN       WORD_SIZE
DOUBLE_ALIGN      64
STRUCT_ALIGN      WORD_SIZE
POINTER_ALIGN     WORD_SIZE
FUNCTION_ALIGN     32

STACK             TOP_DOWN    ;; Cresce dos endereços altos para os baixos

;; -----
;; -----
;; Operações

LOAD_VAR
;; Carrega para registrador uma variável

;; -----
SIGNED_8_BITS

INSTRUCTION INST_REG_MEM Base_Imed (Reg1, Reg2, Constant : 16 )
(
    asm( "load %p, [%p+%p]", Reg1, Reg2, Constant )

```

```

        return ( Reg2::Reg1::Constant::07H )
    )

;; -----
UNSIGNED_8_BITS

INSTRUCTION INST_REG_MEM Base_Imed ( Reg1, Reg2, Constant : 16 )
(
    asm( "load %p, [%p+%p]", Reg1, Reg2, Constant )
    return ( Reg2::Reg1::Constant::03H )
)

;; -----
SIGNED_16_BITS

INSTRUCTION INST_REG_MEM Base_Imed (Reg1, Reg2, Constant : 16 )
(
    asm( "load %p, [%p+%p]", Reg1, Reg2, Constant )
    return ( Reg2::Reg1::Constant::0FH )
)

;; -----
UNSIGNED_16_BITS

INSTRUCTION INST_REG_MEM Base_Imed ( Reg1, Reg2, Constant : 16 )
(
    asm( "load %p, [%p+%p]", Reg1, Reg2, Constant )
    return ( Reg2::Reg1::Constant::0BH )
)

;; -----
SIGNED_WORD_SIZE                <<--- Mapeado para o tipo "int"
UNSIGNED_WORD_SIZE
SIGNED_32_BITS
UNSIGNED_32_BITS

INSTRUCTION INST_REG_MEM Base_Imed ( Reg1, Reg2, Constant : 16 )
(
    asm( "load %p, [%p+%p]", Reg1, Reg2, Constant )
    return ( Reg2::Reg1::Constant::17H )
)

;; -----
FLOAT_WORD_SIZE                  <<--- Mapeado para o tipo "float"
FLOAT_DOUBLE_WORD_SIZE          <<--- Mapeado para o tipo "double"

    ;; Sem suporte do processador!!

;; -----
LOAD_CONST
;; Carrega para registrador uma constante de qualquer comprimento

;; -----
SIGNED_8_BITS
SIGNED_16_BITS

INSTRUCTION INST_REG ( Reg, Constant : 16 )
(
    asm( "movi %p, %p", Reg, Constant )
    return ( R0::Reg::Constant::04H )        ;; movi = addi rd, R0, IMM16
)

```

```

)

;; -----
UNDEFINED_8_BITS
UNDEFINED_16_BITS

INSTRUCTION INST_REG ( Reg, Constant : 16 )
(
    asm( "movi %p, %p", Reg, Constant )
    return ( R0::Reg::Constant::14H )      ;; movui = ori rd, R0, IMM16
)

;; -----
SIGNED_WORD_SIZE
UNDEFINED_WORD_SIZE
SIGNED_32_BITS
UNDEFINED_32_BITS

#define LO_32( Valor )      ((Valor & 0x0000FFFF))
#define HIADJ_32( Valor )   ((Valor >> 16) & 0x0000FFFF) + ((Valor >> 15) & 0x00000001)

INSTRUCTION INST_REG ( Reg, Constant : 32 )
(
    ;; movi = orhi rd, R0, IMM16, addi rd, R0, IMM16

    asm( "orhi %p, %p", Reg, HIADJ(Constant) )
    asm( "movi %p, %p", Reg, LO_32(Constant) )
    return (R0::Reg:: HIADJ (Constant)::34H, R0::Reg:: LO_32(Constant)::04H )
)

;; -----
FLOAT_WORD_SIZE
FLOAT_DOUBLE_WORD_SIZE

    ;; Sem suporte do processador!!

;; -----
STORE
;; Carrega para memória uma variável de qualquer comprimento

;; -----
SIGNED_8_BITS
UNDEFINED_8_BITS

INSTRUCTION INST_REG_MEM Base_Imed ( Reg1, Reg2, Constant : 16 )
(
    asm( "store [%p+%p], %p", Reg2, Constant, Reg1 )
    return ( Reg2::Reg1::Constant::05H )
)

;; -----
SIGNED_16_BITS
UNDEFINED_16_BITS

INSTRUCTION INST_REG_MEM Base_Imed ( Reg1, Reg2, Constant : 16 )
(
    asm( "store [%p+%p], %p", Reg2, Constant, Reg1 )
    return ( Reg2::Reg1::Constant::0DH )
)

```

```
;; -----
SIGNED_WORD_SIZE
UNSIGNED_WORD_SIZE
SIGNED_32_BITS
UNSIGNED_32_BITS
```

```
INSTRUCTION INST_REG_MEM Base_Imed ( Reg1, Reg2, Constant : 16 )
(
    asm( "store [%p+%p], %p", Reg2, Constant, Reg1 )
    return ( Reg2::Reg1::Constant::0DH )
)
```

```
FLOAT_WORD_SIZE
FLOAT_DOUBLE_WORD_SIZE
```

```
;; Sem suporte do processador!!
```

```
;; -----
MOVE
;; Carrega de registrador para registrador
```

```
INSTRUCTION INST_REG_REG ( Reg1, Reg2 )
(
    asm( "mov %p, %p", Reg1, Reg2 )          ;; mov = add rd, rs, R0
    return ( Reg2::R0::Reg1::110001::00000::111010 )
)
```

```
;; -----
NOP
;; "No Operation"
```

```
INSTRUCTION ()
(
    asm( "nop" )                             ;; nop = add R0, R0, R0
    return ( R0::R0:: R0::110001::00000::111010 )
)
```

```
;; -----
ADD                                     ;; Soma
```

```
SIGNED_8_BITS
UNSIGNED_8_BITS
```

```
INSTRUCTION INST_REG_REG ( Reg1, Reg2 )
(
    asm( "add %p, %p, %p", Reg1, Reg1, Reg2 )  ;; add rc, ra, rb
    return ( Reg1::Reg2::Reg1::110001::00000::111010 )
)
```

```
INSTRUCTION INST_IMMEDIATO_REG ( Constant: 16, Reg )
(
    asm( "addi %p, %p, %p", Reg1, Reg1, Constant )  ;; addi rc, ra, IMM16
    return ( Reg1::Reg1::Constant::000100 )
)
```

```
(...)
```

```
;; -----
MUL                                     ;; Multiplicação
```

SIGNED_32_BITS

UNSIGNED_32_BITS

INSTRUCTION INST_REG_REG (Reg1, Reg2)

```
(
    asm( "mul %p, %p, %p", Reg1, Reg1, Reg2 )    ;; mul rc, ra, rb
    return ( Reg1::Reg2::Reg1::100111::00000::111010 )
)
```

INSTRUCTION INST_IMMEDIATO_REG (Constant: 16, Reg)

```
(
    asm( "muli %p, %p, %p", Reg1, Reg1, Constant )    ;; muli rc, ra, IMM16
    return ( Reg1::Reg1::Constant::000100 )
)
```

(...)

```
;; -----
JMP                ;; Incondicional
```

INSTRUCTION INST_REG (Reg)

```
(
    asm( "jmp %p", Reg )                ;; jmp ra
    return ( Reg::00000::00000:: 001101::00000::111010 )
)
```

```
;; -----
JMPEQ              ;; Sem sinal e com sinal
```

INSTRUCTION INST_REG_REG (Reg1, Reg2, Constant: 16)

```
(
    asm( "beq %p, %p, %p", Reg1, Reg2, Constant )    ;; beq
    return ( Reg1::Reg2::Constant:: 100110 )
)
```

```
;; -----
JMPNE             ;; Sem sinal e com sinal
```

INSTRUCTION INST_REG_REG (Reg1, Reg2, Constant: 16)

```
(
    asm( "bne %p, %p, %p", Reg1, Reg2, Constant )    ;; bne
    return ( Reg1::Reg2::Constant:: 011110 )
)
```

```
;; -----
JMPGT             ;; Sem sinal e com sinal
```

INSTRUCTION INST_REG_REG (Reg1, Reg2, Constant: 16)

```
(
    asm( "bgt %p, %p, %p", Reg1, Reg2, Constant )    ;; bgt = blt com registradores invertidos
    return ( Reg2::Reg1::Constant:: 010110 )
)
```

```
;; -----
JMPGE            ;; Sem sinal e com sinal
```

INSTRUCTION INST_REG_REG (Reg1, Reg2, Constant: 16)

```
(
    asm( "bge %p, %p, %p", Reg1, Reg2, Constant )    ;; bge
```

```

        return ( Reg1::Reg2::Constant:: 001110 )
    )

;; -----
JMPLT          ;; Sem sinal e com sinal

INSTRUCTION INST_REG_REG ( Reg1, Reg2, Constant: 16 )
(
    asm( "bne %p, %p, %p", Reg1, Reg2, Constant )    ;; blt
    return ( Reg1::Reg2::Constant:: 010110 )
)

;; -----
JMPLE          ;; Sem sinal e com sinal

INSTRUCTION INST_REG_REG ( Reg1, Reg2, Constant: 16 )
(
    asm( "ble %p, %p, %p", Reg1, Reg2, Constant )    ;; ble = bge com registradores invertidos
    return ( Reg2::Reg1::Constant:: 001110 )
)

;; -----
CALL

INSTRUCTION ( Constant: 26 )
(
    asm( "call %p", Constant )        ;; call
    return (Constant:: 000000 )
)

INSTRUCTION INST_REG ( Reg )
(
    asm( "call %p", Reg )              ;; callr
    return ( Reg::00000:: 11111::011101::00000::111010 )
)

;; -----
RETURN

INSTRUCTION ()
(
    asm( "ret" )                      ;; ret
    return (11111::00000:: 11111:: 000101::00000::111010 )
)

(...)

```

Apêndice 7

Especificação ANSI C

Pode ser obtida nestes sites, acessados em Janeiro 2006:

- <http://webstore.ansi.org/ansidocstore/product.asp?sku=AS+3955%2D1991>
- http://eli-project.sourceforge.net/c_html/c.html

Apêndice 8

Gramática ANSI C para o YACC

Pode ser obtida nos sites: *ftp.uu.net/usenet/net.sources/ansi.c.grammar.Z* e <http://www.lysator.liu.se/c/ANSI-C-grammar-y.html>. Acessados em Janeiro 2006.

Apêndice 9

Analizador Léxico ANSI C para o Lex

Pode ser obtido no site: *http://www.lysator.liu.se/c/ANSI-C-grammar-l.html*. Acessado em Janeiro 2006.