**UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS**

KELTON AUGUSTO PONTARA DA COSTA

**Uma Ferramenta para Execução de Algoritmos Utilizando o Modelo a
Fluxo de Dados Dinâmico em Hardware Reconfigurável para a arquitetura
ChipCflow : Módulo de Conversão C em Grafo a Fluxo de Dados**

São Carlos
2009

KELTON AUGUSTO PONTARA DA COSTA

**Uma Ferramenta para Execução de Algoritmos Utilizando o Modelo a
Fluxo de Dados Dinâmico em Hardware Reconfigurável para a arquitetura
ChipCflow : Módulo de Conversão C em Grafo a Fluxo de Dados**

Tese apresentada à Escola de Engenharia de
São Carlos da Universidade de São Paulo para
obtenção do título de Doutor em Engenharia
Elétrica.

Orientador: Prof. Dr. Valentin Obac Roda

São Carlos
2009

AUTORIZO A REPRODUÇÃO E DIVULGAÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

Costa, Kelton Augusto Pontara da.

C837c

Uma Ferramenta para Execução de Algoritmos Utilizando o Modelo a Fluxo de Dados Dinâmico em Hardware Reconfigurável para a arquitetura ChipCflow : Módulo de Conversão C em Grafo a Fluxo de Dados

São Carlos, 2009.

Tese (Doutorado – Programa de Pós-Graduação em Engenharia Elétrica e Área de Concentração em Processamento de Sinais) – Escola de Engenharia de São Carlos da

Universidade de São Paulo, 2009.

1. Introdução a DFG. 2. Características do FPGA. 3. Compiladores e suas estruturas.

4. Ferramentas de programação C para plataformas FPGA. 5. Modelo a fluxo de dados. 6.

Ferramenta de conversão. I. Título.

FOLHA DE APROVAÇÃO

Kelton Augusto Pontara da Costa

Uma Ferramenta para Execução de Algoritmos Utilizando o Modelo a Fluxo de Dados Dinâmico em Hardware Reconfigurável para a arquitetura ChipCflow : Módulo de Conversão C em Grafo a Fluxo de Dados

Tese apresentada à Escola de Engenharia de São Carlos da Universidade de São Paulo para obtenção do título de Doutor em Engenharia Elétrica.

Área de Concentração: Processamento de Sinais

Aprovado em:

Banca Examinadora

Prof. Dr.

Instituição:

Assinatura: _____

Prof. Dr.

Instituição:

Assinatura: _____

Prof. Dr.

Instituição:

Assinatura: _____

Prof. Dr.

Instituição:

_____Assinatura:_____

Prof. Dr.

Instituição:

_____Assinatura:_____

DEDICATÓRIA

A minha esposa e filho, pais e familiares, com amor, admiração e gratidão por sua compreensão, carinho, presença e incansável apoio ao longo do período de elaboração deste trabalho.

AGRADECIMENTOS

Ao Prof. Dr. Valentin Obac Roda, pela atenção e apoio durante o processo de definição e orientação, e por confiar em minha capacidade.

Ao Prof. Dr. Jorge Luiz e Silva, que nos anos de convivência, muito me ensinou, contribuindo para o meu crescimento científico, intelectual e pessoal além de ser um grande amigo.

Aos funcionários do departamento de Engenharia Elétrica de São Carlos, pelo apoio e amizade.

Aos amigos e colegas, em especial ao grande amigo Prof. Especialista Luis Alexandre da Silva pelo apoio e força durante o processo final para a defesa, momento em que mais precisava devido ao cansaço e complexidade do projeto, isto nunca será esquecido ou deixado para trás.

Ao meu amigo e cunhado Prof. Mestre Miguel José das Neves pela apoio e participação constante durante todo o processo de início e término do doutoramento.

A minha família em Cristo, os casais da Equipe de Nossa Senhora (ENS), pelo apoio e amizade.

À Deus pelo Dom da Vida e pela grande paciência que ele têm comigo.

À Nossa Senhora que sempre está e me ilumina em todos os meus caminhos.

RESUMO

COSTA, K. A. P. Uma Ferramenta para Execução de Algoritmos Utilizando o Modelo a Fluxo de Dados Dinâmico em Hardware Reconfigurável para a arquitetura ChipCflow : Módulo de Conversão C em Grafo a Fluxo de Dados. 2009. 103f. Tese (Doutorado) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2009.

O *ChipCflow* é o projeto de uma ferramenta para execução de algoritmos utilizando o modelo a fluxo de dados dinâmico em hardware reconfigurável. Tem como principal objetivo utilizar o modelo de arquitetura a fluxo de dados, associado ao conceito de arquiteturas reconfiguráveis, para acelerar programas de aplicação escritos em C. Essa aceleração vai acontecer nas partes mais intensas de processamento (exemplo: *loops*), através da execução direta em hardware, aproveitando ao máximo o paralelismo considerado natural do modelo a fluxo de dados. O Módulo de Conversão C em Grafo a Fluxo de Dados, desenvolvido neste trabalho de doutoramento, é parte fundamental no desenvolvimento do projeto *ChipCflow* e se trata de um pré-compilador C que gera código *VHDL* para os operadores a fluxo de dados a partir de módulos do programa de aplicação escrito originalmente em C. É possível considerar como principal resultado dessa tese, a implementação de algoritmos a partir de uma linguagem C em grafo a fluxo de dados no modelo proposto no projeto *ChipCflow*, onde os operadores a fluxo de dados são os já tradicionalmente conhecidos, e até mesmo a forma de interconexão também, entretanto a forma de abordar essa implementação, em *VHDL*, e na forma de fluxo de dados dinâmico, torna o modelo proposto, tanto para o hardware como para o compilador, uma característica única existente até o momento.

COSTA, K. A. P. A Tool for Algorithms Implementation using the Dynamic Data Flow Model in Reconfigurable Hardware for the ChipCflow architecture - C conversion to Data Flow Graph module. 2009. 103f. Thesis (Doctoral) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2009.

ChipCflow is a tool for algorithms execution using a data dynamic flow in reconfigurable hardware. The main purpose of the work is to use the data flow architecture model, associated to the concept of reconfigurable architectures, to speed up C written applications. The program acceleration happens in the most intensive processing parts (example: *loops*), through direct hardware execution, using to advantage of the natural parallelism of the data flow model. The C Conversion Module in Data Flow Graph is the basic part in the development of the *ChipCflow* project and the specific objective of the developed work. In this work we present details of the C precompiler that generates *VHDL* code for the data flow operators modules of the written program of the original application in C. It is possible to consider as the main result of this thesis, the implementation of algorithms from a C language in the data flow graph model proposed in the project ChipCflow, where the operators to the data stream are typically already known, and even the way interconnection too, but how to approach the implementation in VHDL, and as a dynamic data flow, makes the proposed model for both the hardware and the compiler, a feature only available so far.

LISTA DE ILUSTRAÇÕES

Figura 1- Proposta da Ferramenta ChipCflow	20
Figura 2- Arquitetura geral de roteamento de um FPGA	23
Figura 3- Exemplo de reconfiguração dinâmica	25
Figura 4 - Árvore Sintática para uma atribuição.....	39
Figura 5 – Possíveis implementações geradas por <i>Pico Express</i>	47
Figura 6 - Visão geral do Pico Express	48
Figura 7 – Visão detalhada do Pico Express	48
Figura 8 – Arquitetura do hardware gerado pelos compiladores Galadriel e Neny.....	49
Figura 9 – Fluxo de compilação feitos por Galadriel e Neny	51
Figura 10 – Estrutura utilizada pelo compilador Neny para implementar o hardware gerado pelo compilador Neny.....	54
Figura 11 – Colocação relativa de células no FPGA	55
Figura 12 - Arquitetura de configuração do compilador	57
Figura 13 – Visão geral de HLS.....	59
Figura 14 – Ambiente ANSI-C e ambiente Handel-C.....	61
Figura 15 - Um exemplo de Grafo a Fluxo de Dados.....	68
Figura 16 - Sub-grafo F a ser incluído no grafo original T.....	69
Figura 17 - Resultado da concatenação do sub-grafo F ao grafo T.....	70
Figura 18 - Sub-grafo a ser removido do grafo original	71
Figura 19 - Grafo Resultante da remoção de sub-grafo.....	71
Figura 20 - Instâncias diferentes para o operador "*"	73
Figura 21 - Operadores do Modelo DDFG	73
Figura 22 - Grafo representando comandos em C++	74
Figura 23 - Formato dos dados contendo Tags.....	75
Figura 24 - Exemplo de um programa com construtores iterativos	75
Figura 25 - Instância com circuito de "Matching" e variável comum	77
Figura 26 - Visão da ferramenta utilizando técnicas de Análise Léxica para posterior conversão em VHDL	78
Figura 27 – Grafo extraído do escopo "for"	80
Figura 28 - Identificação dos tokens pela análise léxica	81

Figura 29 - Tela inicial da ferramenta de conversão	82
Figura 30 - Janela de busca dos tokens gerados em arquivo com extensão “.c”	83
Figura 31 - Primeiro resultado da execução para identificação dos operandos DFG “FOR” e dependências	84
Figura 32 - Segundo resultado da execução para identificação do mapa binário “FOR”	85
Figura 33 - Definição de bits e operandos DFG para a ferramenta.....	86
Figura 34 - Grafo extraído do escopo “if”	87
Figura 35 - Primeiro resultado da execução para identificação dos operandos DFG “IF” e dependências	87
Figura 36 - Segundo resultado da execução para identificação do mapa binário “IF”	88
Figura 37- Grafo extraído do escopo “While”	89
Figura 38 - Primeiro resultado da execução para identificação dos operandos DFG “While” e dependências	89
Figura 39 - Segundo resultado da execução para identificação do mapa binário “While”	90
Figura 40 - Grafo extraído do escopo (IF, FOR, WHILE)	92
Figura 41 - Execução para identificação dos operandos DFG (IF, FOR, WHILE) parte1	92
Figura 42 - Execução para identificação dos operandos DFG (IF, FOR, WHILE) parte2	93
Figura 43 - Identificação mapa binário parte1	93
Figura 44 - Identificação mapa binário parte 2	94
Figura 45- Identificação do mapa binário parte 3	95
Figura 46 - Ferramenta ChipCFlow com o novo item inserido: TXT::VHDL	96
Figura 47 - DFG de um escopo FOR gerado em linguagem VHDL após análise do mapa binário.....	98
Figura 48 - DFG de um escopo IF gerado em linguagem VHDL após análise do mapa binário	99
Figura 49 - DFG de um escopo WHILE gerado em linguagem VHDL após análise do mapa binário.....	100
Figura 50 - Escopo IF, FOR e WHILE gerado em linguagem VHDL após análise do mapa binário.....	101

LISTA DE TABELAS

Tabela 1 – Tokens, lexemas e descrição.....30

Tabela 2 – Tipo de dados, operações e controles suportados pelo compilador Galadriel.....51

LISTA DE SIGLAS

ACM Association for Computing Machinery
ASIC Application Specific Integrated Circuit
ANSI American National Standards Institute
ALU Arithmetic and Logic Unit
CI Circuito Integrado
CSP Communicating Sequential Processes
DVD Digital Versatile Disc
DFG DataFlow Graph
EDA Eletronic Design Automation
EPROM Erasable PROM
FPGA Field Programmable Gate Array
FPSOC Field Programmable System On Chip
FCCM Field-Programmable Custom Computing Machines
GNU GNU is Not Unix
GCC GNU C Compiler
HDL Hardware Description Language
HPDG Hierarchical Program Dependence Graph
HLS High level Synthesis
IEEE Institute of Electric and Electronic Engineers
IP Intellectual Property
JVM Java Virtual Machine
LL Linguagem Livre de Contexto
LR Linguagem Recursiva
MPGA Mask Programmable Gate Array
MPEG Moving Picture Experts Group
NTM New Tag Manager
NIG New Iteration Generation
NTD New Tag Destructors
PLD Programmable Logic Device
PCT Pilha de Controle de Tipos

PCO Pilha de Controle de Operandos
PHLS Parallelizing Highlevel Synthesis
PPA Pipeline of Processor
RTR Run-Time Reconfiguration
RPU Reconfigurable Processing Units
RTL Register Transfer Level
TS Tabela de Símbolos
VHDL VHSIC Hardware Description Language
VLSI Very Large Scale Integration
VLIW Very Long Instruction Width

SUMÁRIO

INTRODUÇÃO	17
CAPÍTULO 2- CARACTERÍSTICAS DOS DISPOSITIVOS FPGA	21
2.1 Arquitetura de Roteamento.....	23
2.2 Hardware que permite reconfiguração dinâmica.....	23
2.3 Reconfiguração de FPGAs	24
2.3.1 Reconfiguração Parcial	24
2.3.2 Reconfiguração Dinâmica.....	24
CAPÍTULO 3 - COMPILADORES E SUAS ESTRUTURAS	28
3.1 Análise Léxica.....	29
3.1.1 Tokens, Padrões e Lexemas	29
3.1.2 Erros Léxicos	31
3.2 Análise Sintática	31
3.2.1 Tratamento dos Erros de Sintaxe.....	32
3.2.2 Estratégia de recuperação de erros	32
3.2.3 Regras Gramaticais	33
3.2.4 Análise sintática top-down.....	33
3.2.5 Análise Sintática bottom-up.....	33
3.3 Análise Semântica.....	34
3.3.1 Entradas nas tabelas de símbolos	34
3.3.2 Alocação de posição de memória.....	34
3.3.3 Implementação da tabela de símbolos.....	35
3.4 Compatibilidade de tipos.....	36
3.4.1 Expressões Aritméticas	36
3.5 Geração de código intermediário.....	39

3.5.1 Árvores de sintaxe	39
3.5.2 Triplas de código intermediário	40
3.5.3 Expressão aritmética e atribuição	40
3.5.4 Comandos de controle de fluxo	41
3.6 Geração de Código-objeto	41

CAPÍTULO 4 - FERRAMENTAS DE PROGRAMAÇÃO C PARA PLATAFORMAS FPGA

4.1 Introdução a Ferramentas de Programação C para Plataformas FPGA	44
4.2 Pico Express	46
4.3 Galadriel e NENYA.....	49
4.4 Compilador apresentado por Chichkov	55
4.5 Spark	58
4.5.1 Síntese a partir de linguagem de alto nível	58
4.5.2 HLS paralelo	60
4.6 Handel-C	60
4.7 Impulse C	62

CAPÍTULO 5 - MODELO A FLUXO DE DADOS DINÂMICO PARA HARDWARE RECONFIGURÁVEL UTILIZADO NESTE PROJETO

5.1 Estrutura do Modelo a Fluxo de Dados.....	68
5.1.1 O Modelo de Instâncias	69
5.1.2. A estrutura tagged-token	72
5.1.3 Operadores utilizados no Modelo.....	73
5.1.4 Construções Iterativas no Modelo	74
5.2 Estrutura "Matching" de Dados.....	75

CAPÍTULO 6 – FERRAMENTA DE CONVERSÃO C EM GRAFO A FLUXO DE DADOS.....	78
6.1 O Analisador Léxico gerado na Ferramenta.....	79
6.1.1 Implementação do Analisador Léxico em código C++	79
6.2 A Ferramenta para conversão.....	81
6.3 O Processo para conversão de código C em VHDL.....	96
CAPÍTULO 7 – CONCLUSÃO	102
7.1 Trabalhos futuros	104
REFERÊNCIAS	106

INTRODUÇÃO

Durante os anos 70 até meados dos anos 90, existiram algumas implementações envolvendo arquiteturas a fluxo de dados (ARVIND, 2005; DENNIS, 1975; VEEN, 1986).

A diferença fundamental entre as arquiteturas tradicionais von Neumann e as arquiteturas a fluxo de dados está exatamente no controle do fluxo de informações que no caso das máquinas von Neumann é baseado no controle seqüencial, enquanto que nas máquinas a fluxo de dados é executado pela presença dos dados (CAPPELLI et al., 2004; SWANSON et al., 2003).

Um grafo a fluxo de dados é um conjunto de operadores interligados por arcos. A presença dos dados em cada arco de um operador é que irá disparar a execução desse operador.

Existem dois modelos de arquiteturas a fluxo de dados: estático e dinâmico. No modelo estático apenas um dado pode estar presente em um arco esperando pelos seus dados parceiros. Um protocolo deve garantir o sincronismo entre os operadores para garantir que apenas um dado esteja presente em cada arco. Conseqüentemente o paralelismo fica limitado ao conjunto de operadores cujos arcos possuem apenas um dado. No modelo dinâmico mais que um dado pode estar presente no arco. Um protocolo deve assegurar que os dados presentes nos arcos disparem as operações cujos dados parceiros também estejam presentes. O "*Tagged-Token*" é usado no protocolo para controlar os dados parceiros em cada arco (ARVIND, 2005). Neste caso, o paralelismo acontece quando da execução de vários operadores cujos dados estejam todos disponíveis.

Um programa sendo executado por um microprocessador pode possuir trechos que demandam mais tempo de execução que outros. Se considerarmos por exemplo comandos em loops, e dependendo de como esses loops estão construídos, a execução dos mesmos pode ocupar um longo tempo de processamento. Esses trechos podem ser considerados pontos de processamento intenso.

Computação Reconfigurável por sua vez, ainda é uma área em desenvolvimento, diferente de arquiteturas de computadores na qual os conceitos e mecanismos já vêm sendo

largamente testados e comprovados. Validar novos e diferentes tópicos em arquiteturas reconfiguráveis tem sido um dos desafios dos pesquisadores na área, nos últimos 20 anos.

Tendo como base os dispositivos programáveis *FPGA* (*Field Programmable Gate Array*), uma das definições para computação reconfigurável é a dada por Dehon:

Quando se pensa em computadores de uso geral, pensamos em um microprocessador executando um programa seqüencial. Entretanto, uma característica simples que define um processador de uso geral seria a de um pedaço de silício, configurado após a fabricação como um circuito integrado (CI) para resolver qualquer tarefa computacional. Isso significa, que até por motivos econômicos, muitas aplicações poderiam ser executadas sobre um único CI, e também que o mesmo CI poderia resolver problemas diferentes em diferentes instantes de tempo. Significando ainda que as pessoas poderiam programar esses componentes para fazerem coisas que o produtor original do CI nunca teria imaginado fazer. Computação Reconfigurável, compartilha todas essas características de um computador de uso geral, mas organiza a computação de maneira muito diferente. Ao invés de computar uma função seqüencialmente através de um conjunto de instruções de tempo, igual a um processador, as arquiteturas reconfiguráveis geralmente computam as funções através de unidades funcionais configuradas no espaço (unidades em diferentes blocos lógicos dentro das *FPGA*'s). Portanto, computação paralela envolvendo diferentes unidades funcionais que geram e consomem resultados intermediários. A diferença está na computação totalmente espacial/seqüencial das tarefas em computador de uso geral e a computação totalmente espacial/paralela em computador reconfigurável. (Dehon,1996)

Originalmente dispositivos *FPGAs* vinham sendo utilizado como dispositivo "*glue logic*", substituindo sistemas lógicos construídos com vários circuitos integrados, reduzindo significativamente o número de componentes em um sistema, e diminuindo os riscos de desenvolvimento de projetos, ao mesmo tempo em que proporcionavam a flexibilidade para correções e atualizações de forma rápida e mais segura. Atualmente os *FPGAs* são utilizados no projeto e controle de inúmeros sistemas com aplicações específicas, significativamente complexas, e dedicadas.

Muitos recursos têm sido aplicados para desenvolver sistemas reconfiguráveis baseados em *FPGAs*, tornando computação reconfigurável uma realidade.

Diversas arquiteturas reconfiguráveis foram propostas nas últimas 2 décadas, dentre elas a máquina Matrix (*Multiply Alu architecture with Reconfigurable InterConnect eXperiment*) proposta pelo próprio Dehon (Dehon, 1996) e Mirsky (Mirsky,1996); o projeto GARP proposto por Hauser (Hauser,1997); a máquina DISC proposto por Wirthlin e Eldredge (Wirthlin,1994), (Wirthlin,2001) e Eldredge (Eldredge,1994); a máquina SPLASH proposta por Arnold e Buell (Arnold, 1993) e um grande número de aplicações sendo desenvolvidas na área (algumas dessas máquinas estão descritas no capítulo 2). Poderíamos referenciar, por

exemplo, os artigos apresentados nos últimos quatorze anos nos congressos de *FPGAs* patrocinados pela ACM e o FCCM patrocinado pela *IEEE*.

A grande experiência adquirida no passado no desenvolvimento de máquinas a fluxo de dados, a experiência também adquirida nos últimos 20 anos em computação reconfigurável, e mais recentemente o aparecimento de grupos desenvolvendo sistemas a fluxo de dados em computação reconfigurável e compiladores para arquiteturas reconfiguráveis foram os motivadores para desenvolver este trabalho.

Podemos destacar o projeto WaveScalar, proposto por Swanson (SWANSON, 2003), (SWANSON, 2006); a Unidade de Controle Dataflow para Compilação Pipeline Configurável a partir de programação C, por CAPPELLI (CAPPELLI, 2004); o projeto SPARK proposto por Gupta (GUPTA, et al., 2004) no "Center for Embedded Computer Systems"; o Stream-C (GOKHALE, 2006) e (STREAM-C, 2005) desenvolvido pelo "Alamos National Laboratory"; *Pico Express* comercializado pela empresa *Synfora* (SYNFORA, 2007); o compilador de *Chichkov* (CHICHKOV, et al., 1997); *DK Design Suit da* (CELOXICA Ltd, 2002); *Handel-C* (Chappell, et al., 2002); *ImpulseC* (PELLERIN, et al., 2005), etc.

O *ChipCflow* é o projeto de uma ferramenta para execução de algoritmos utilizando o modelo a fluxo de dados dinâmico em hardware reconfigurável. Tem como principal objetivo utilizar o modelo de arquitetura a fluxo de dados, associado ao conceito de arquiteturas reconfiguráveis, para acelerar programas de aplicação escritos em C.

Essa aceleração vai acontecer nas partes mais intensas de processamento (exemplo: loops), através da execução direta em hardware, aproveitando ao máximo o paralelismo considerado natural do modelo a fluxo de dados (SILVA, MARQUES, 2006).

A Figura 1 descreve a sequência de atividades para que uma aplicação seja utilizada na ferramenta *ChipCflow*.

Figura 1- Proposta da Ferramenta ChipCflow

Fonte: (CORREIA, 2007)

A partir do programa de aplicação escrito originalmente na linguagem C, se extrai os grafos *CDFG* do programa de aplicação tendo como referencia para os operadores, uma base de grafos *CDFG* a ser gerada especificamente para essa ferramenta. O grafo *CDFG* gerado já estará otimizado. O grafo então é convertido em *VHDL*, tendo como base todo o conjunto de operadores propostos para o *ChipCflow*, previamente implementados em *VHDL*. O código *VHDL* é então sintetizado e simulado em ferramenta *EDA* comercial, em particular trabalharemos com *ISE* da *Xilinx*, e o bitstream é então gerado para serem executados direto no hardware.

Baseado nos modelos de compiladores já existentes, como: *SPARK*; *Stream-C*; *Pico Express*; *DK Design Suit*; *Handel-C*; *ImpulseC*, etc., foi desenvolvido um pré-compilador que a partir do programa de aplicação escrito em linguagem C, extrai os grafos *CDFG* do programa, já otimizado, e os converte para *VHDL* pronto para que seja implementados em ferramentas *EDA* e executado direto no hardware.

No capítulo 2 são abordados conceitos e características principais do *FPGA*, o capítulo 3 descreve conceitos, técnicas e as fases de desenvolvimento de um compilador, o capítulo 4 mostra as ferramentas de programação C para plataformas *FPGA*, o capítulo 5 explica o modelo a fluxo de dados dinâmicos para hardware reconfigurável, já no capítulo 6 é descrita a ferramenta de conversão C em grafo a fluxo de dados desenvolvida, no capítulo 7 encontra-se a conclusão e em seguida as referências da tese.

CAPÍTULO 2- CARACTERÍSTICAS DOS DISPOSITIVOS FPGA

Os circuitos digitais têm sofrido grande evolução nas últimas décadas. As constantes mudanças na tecnologia têm transformado de forma radical todo o processo de projeto de hardware.

Os componentes dos circuitos digitais evoluíram de transístores individuais para circuitos integrados *VLSI* (*Very Large Scale Integration*). A utilização de ferramentas *EDA* (*Electronic Design Automation*) tem simplificado e acelerado todo o ciclo de projeto. Atualmente, não é mais necessário desenhar portas lógicas individuais e planejar todas as suas interconexões. As linguagens de descrição de *hardware* (*HDLs*) estão hoje consolidadas no meio acadêmico e industrial como forma padrão na elaboração de projetos. Há, também, ferramentas de síntese lógica automática, disponíveis para mapear circuitos em diversas tecnologias (CHAN; MOURAD, 1994).

Todas essas mudanças na tecnologia exigem uma prototipação cada vez mais rápida, pois o ciclo de vida dos produtos modernos está se tornando cada vez mais curto em relação ao tempo necessário para o projeto e o desenvolvimento dos mesmos.

Os CIs (*Circuitos Integrados*) digitais podem ser construídos utilizando-se de diversas tecnologias diferentes. A escolha da tecnologia adequada deve ser realizada com base no tipo de projeto que se pretende executar.

Segundo Wolf (2001), as implementações de circuitos podem ser agrupadas em diversas categorias de projeto, tais como, CIs customizados ou *ASICs* (*Application Specific Integrated Circuits* – Circuitos Integrados com Aplicação Específica), *MPGAs* (*Mask Programmable Gate Arrays*), *Standard Cells* e *PLDs* (*Programmable Logic Devices* – Dispositivos Lógicos Programáveis). Os CIs customizados, ou *ASICs*, são aqueles que necessitam de um processo de fabricação especial, que requerem metodologias específicas para cada projeto. Características desse tipo de implementação são os custos de projeto extremamente altos e o tempo de desenvolvimento longo. Em aplicações que requerem um grande volume de produção, o alto custo do projeto e dos testes é amortizado.

Segundo Brown e Rose (1996):

Um FPGA consiste de um grande arranjo de células configuráveis (ou blocos lógicos) contidos em um único *chip*. Cada uma dessas células contém certa capacidade computacional para implementar funções lógicas, e/ou realizar roteamento para permitir a comunicação entre

células. Essas operações podem acontecer simultaneamente no arranjo das células.

2.1 Arquitetura de Roteamento

A arquitetura de roteamento de um *FPGA* é a maneira como os comutadores programáveis e segmentos de trilha são posicionados para permitir a interconexão entre as células lógicas (TEIXEIRA, 2002). Esta arquitetura pode ser descrita a partir do modelo geral ilustrado na Figura 2.

Figura 2- Arquitetura geral de roteamento de um FPGA

Fonte: (VERONESI, 2005)

O modelo contém duas estruturas básicas. A primeira é o bloco de conexão que aparece em todas as arquiteturas. O bloco de conexão permite a conectividade das entradas e saídas de um bloco lógico com os segmentos de trilhas nos canais. A segunda estrutura é o bloco de comutação que permite a conexão entre os segmentos de trilhas horizontais e verticais. Em algumas arquiteturas, o bloco de comutação e o bloco de conexão são distintos; em outras, estão combinados numa mesma estrutura. Nem todas as arquiteturas seguem este modelo. Há arquiteturas que apresentam uma estrutura hierárquica e outras que possuem somente estruturas horizontais.

2.2 Hardware que permite reconfiguração dinâmica

Segundo Veronesi (2005), o mercado oferece uma ampla gama de dispositivos *FPGAs*, de diversos fabricantes; em geral, a estrutura de roteamento, os blocos lógicos e a forma de

programação variam. Dependendo da aplicação, uma arquitetura pode ter características mais desejáveis para a aplicação em questão.

2.3 Reconfiguração de FPGAs

Os dispositivos *FPGAs* podem ser classificados de acordo com suas configurabilidades, (LYSAGHT; DUNLOP, 1993). Todos os dispositivos *FPGAs* são, por definição, programáveis, ou seja, configuráveis pelo menos uma única vez, como qualquer PLD. Um pequeno subconjunto destes dispositivos pode ser reconfigurado várias vezes por alguma operação que recarrega completamente a configuração do dispositivo.

2.3.1 Reconfiguração Parcial

Segundo Veronesi (2005), um dispositivo é definido como reconfigurável parcialmente se é possível reconfigurá-lo seletivamente, enquanto o resto do dispositivo permanece inativo, mas retém sua configuração.

Caso o dispositivo só possa ser reconfigurado por completo, mesmo que na própria placa do sistema, ou fora dela, como com dispositivos EPROMs, é considerado apenas reconfigurável, pois, obviamente, não permanece ativo em nenhum dos casos, por não permitir configuração seletiva.

2.3.2 Reconfiguração Dinâmica

FPGAs são classificados como dinamicamente reconfiguráveis se seus circuitos internos de armazenamento da configuração podem ser atualizados seletivamente, sem prejudicar o funcionamento da lógica restante que pode estar em operação. Estes dispositivos podem assim ser reconfigurados seletivamente enquanto estiverem ativos. Outros termos atualmente encontrados na literatura que se referem à reconfiguração dinâmica são, *run-time reconfiguration (RTR)*, *real-time reconfiguration*, *on-the-fly reconfiguration* e *in-circuit reconfiguration*. Todas essas expressões podem ser traduzidas também como reconfiguração em tempo de execução (RIBEIRO, 2002).

Veronesi (2005) afirma que estes FPGAs possuem características comuns: eles são *arrays* de blocos lógicos, de granularidade fina ou média, cuja configuração é controlada por escrita de dados em uma memória estática. Para um FPGA ser reconfigurável dinamicamente, a definição implica que ele deve ser capaz de ser reconfigurado parcialmente enquanto ativo (energizado e em operação). Em nível de sistema, um módulo que contenha múltiplos FPGAs reconfiguráveis, total ou parcialmente, pode ser classificado como reconfigurável dinamicamente se os componentes FPGAs são reconfigurados individualmente.

Como os tempos de configuração não são desprezíveis, a habilidade de intercalar execução e reconfiguração, sem prejuízo do desempenho, é uma questão que ainda merece atenção e esforços de pesquisa (RIBEIRO, 2002). Um sistema de reconfiguração dinâmica inclui pelo menos uma área de reconfiguração cujos blocos lógicos podem ser carregados em tempo de execução. A Figura 3 ilustra a reconfiguração dinâmica de um sistema composto de cinco circuitos ou tarefas. A tarefa A é uma função de entrada que alimenta as tarefas B, C ou D, que alimentam a tarefa E, que é a função de saída do sistema. As tarefas de entrada e saída são permanentemente residentes no *FPGA*, enquanto as três tarefas dinâmicas alternam-se sob o controle de um sinal de *Swap*.

Figura 3- Exemplo de reconfiguração dinâmica

Fonte: (RIBEIRO, 2002)

A capacidade de reconfiguração dinâmica permite o compartilhamento ao longo do tempo de tarefas diferentes, o que pode reduzir significativamente a área de silício exigida. Esta tecnologia torna possível o conceito de *hardware* ilimitado ou “*Hardware Virtual*”

(CARDOSO; VÉSTIAS, 1999). Este conceito é similar ao de memória virtual e está sendo adotado para permitir que múltiplos conjuntos de configuração sejam armazenados para serem usados pelas RPU's (*Reconfigurable Processing Units*), quando necessário.

O campo da computação reconfigurável avançou amplamente na década passada, utilizando FPGAs como base para sistemas reprogramáveis de alto desempenho (GUCCIONE, 2000). Muitos desses sistemas alcançaram altos níveis de performance e demonstraram sua aplicabilidade à resolução de uma grande variedade de problemas. Contudo, apesar de os autores desses sistemas os classificarem como reconfiguráveis, eles são tipicamente configurados uma vez antes de iniciarem a execução da aplicação.

As áreas de um programa que podem ser aceleradas através do uso de *hardware* reconfigurável freqüentemente são muito numerosas ou complexas para serem carregadas simultaneamente no *FPGA* disponível. Para esses casos, é interessante que seja possível trocar entre diferentes configurações no mesmo dispositivo.

Com a complexidade citada no parágrafo anterior, é visto portanto, a necessidade de utilização de algum tipo de sistema que trate com um maior controle as áreas que podem ser aceleradas com o uso de *hardware* reconfigurável, para esta situação foi encontrado como um método de controle, a utilização de Compiladores bem como suas fases de desenvolvimento e que irá ser abordado no capítulo seguinte.

CAPÍTULO 3 – COMPILADORES E SUAS ESTRUTURAS

Um tradutor é um programa que recebe como dado de entrada um programa escrito em uma linguagem de programação (a linguagem-fonte) e produz como saída de seu processamento um programa escrito em outra linguagem (a linguagem-objeto). Como importante parte desse processo de tradução, o compilador relata a seu usuário a presença de erros no programa-fonte. Os compiladores são, algumas vezes, classificados como de uma passagem; de passagens múltiplas; de carregar e executar; depuradores ou otimizantes, dependendo de como tenham sido construídos ou que função se suponha que devam realizar.

A construção de compiladores se estende através dos temas de linguagens de programação como arquitetura de máquina, teoria das linguagens, algoritmos e engenharia de software.

Segundo Aho (1995), um compilador é dividido em seis fases:

Análise Léxica: tem como função separar os símbolos significativos dentro do programa-fonte e verificar a categoria a que estes pertencem.

Análise Sintática: verifica se a seqüência dos símbolos escritos no Programa Fonte é válida ou não.

Análise Semântica: verifica, através da árvore sintática (montada depois de passar pelas duas primeiras fases), se os aspectos semânticos do programa têm coerência.

Gerador de código intermediário: usa as estruturas produzidas pelo analisador sintático e verificadas pelo analisador semântico para criar uma seqüência de instruções simples, dita código intermediário (está entre a linguagem de alto nível e a linguagem de baixo nível).

Otimização de código: independentemente de máquina, é um módulo opcional (presente na grande maioria dos compiladores) que objetiva melhorar o código intermediário de modo que o programa-objeto produzido ao fim da compilação seja menor (ocupe menos espaço de memória) e/ou mais rápido (tenha tempo de execução menor). A saída do otimizador de código é um novo código intermediário.

Gerador de código: produz o código-objeto final, tomando decisões com relação à alocação de espaço para os dados do programa, selecionando a forma de acessá-los, definindo que registradores da UCP serão usados, etc. Projetar um gerador de código que produza

programas objeto verdadeiramente eficientes é uma das tarefas mais difíceis no projeto de um compilador.

Já Delamaro (2004), descreve um compilador com os seguintes componentes: o analisador léxico, o analisador sintático, o analisador semântico e o gerador de código.

Cada autor caracteriza as fases de um compilador de uma forma pessoal, uns com uma quantidade maior de fases que outros, mas é perceptível que maioria conclui e obtêm os mesmos níveis de descrição existentes em todas as literaturas existentes sobre compiladores, alguns autores descrevem todas as fases detalhadamente, enquanto outros autores ocultam ou retiram fases, mas acabam inserindo na fase seguinte a função da fase retirada.

3.1 Análise Léxica

O analisador léxico é a primeira fase de um compilador. Sua tarefa principal é a de ler os caracteres de entrada e produzir uma seqüência de *tokens*, que a segunda fase de um compilador chamado de análise sintática ou *parser* posteriormente utiliza para a análise sintática. Ao receber do *parser* um comando “obter o próximo *token*”, o analisador léxico lê os caracteres de entrada até que possa identificar o próximo *token* (através do caráter espaço em branco por exemplo).

Segundo AHO (1995), o analisador léxico também realiza algumas tarefas secundárias. Uma delas é remover do programa-fonte os comentários e os espaços em branco, as tabulações e os caracteres de avanço de linha. Uma outra é a de correlacionar as mensagens de erro do compilador com o programa-fonte.

Algumas vezes, os analisadores léxicos são divididos em duas fases em cascata, a primeira chamada de “varredura” (*scanning*) e, a segunda, de “análise léxica”. O *scanner* é responsável por realizar tarefas mais simples, enquanto o analisador léxico propriamente dito realiza as tarefas mais complexas. Por exemplo, um compilador Fortran poderia usar um *scanner* para eliminar os espaços de entrada.

3.1.1 Tokens, Padrões e Lexemas

Quando se fala sobre análise léxica, usamos os termos “*token*”, “padrão” e lexema com significados específicos. Em geral, existe um conjunto de cadeias de entrada para as quais os

mesmos *tokens* são produzidos como saída. Esse conjunto de cadeias é descrito por uma regra chamada de padrão associado ao *token* de entrada. O padrão é assim chamado por reconhecer cada cadeia do conjunto. Um lexema é um conjunto de caracteres no programa-fonte e é reconhecido pelo padrão de algum *token*. Por exemplo, no enunciado Pascal “*const pi = 3.1416*”, a subcadeia *pi* é um lexema para o *token* “identificador”.

Tratamos os *tokens* como símbolos terminais na gramática para a linguagem-fonte, usando nomes em negrito para representá-los. Os lexemas reconhecidos pelo padrão do *token* representam cadeias de caracteres no programa-fonte e podem receber um tratamento conjunto, como instâncias de uma mesma unidade léxica (por exemplo, instâncias de identificadores, números, etc.) (AHO, 1995).

Na maioria das linguagens de programação, as seguintes construções são tratadas como *tokens*: palavras-chave; operadores; identificadores; constantes; literais e cadeias e símbolos de pontuação, como parênteses, vírgulas e ponto-e-vírgulas. No exemplo anterior, quando a seqüência de caracteres *pi* aparece no programa-fonte, um *token* representando um identificador é repassado ao *parser*. O repasse de um *token* é freqüentemente implementado transmitindo-se um inteiro associado ao *token*. É justamente esse inteiro que é designado por *id* na tabela 1 (AHO, 1995).

Tabela 1 – Tokens, lexemas e descrição

Fonte: (AHO, 1995)

Token	Lexemas: exemplo	Descrição informal do Padrão
const if relação id num literal	const if <, <=, =, <>, >, >= pi, contador, D2 3.1416, 0, 6.02E23 conteúdo da memória	const if < ou <= ou = ou <> ou > ou >= letra seguida por letras e/ou dígitos qualquer constante numérica Quaisquer caracteres entre aspas, exceto aspas

3.1.2 Erros Léxicos

Poucos erros são distinguíveis somente em nível léxico, uma vez que um analisador léxico possui uma visão muito local do programa-fonte. Mas suponhamos uma situação em que o analisador léxico seja incapaz de prosseguir, porque nenhum dos padrões reconheça um prefixo na entrada remanescente (DELAMARO, 2004). Talvez a estratégia mais simples de recuperação seja a da “modalidade pânico”. Removemos sucessivos caracteres da entrada remanescente até que o analisador léxico possa encontrar um *token* bem formado. Essa técnica de recuperação pode, ocasionalmente, confundir o *parser*, mas num ambiente de computação interativo pode ser razoavelmente adequada.

Outras possíveis ações de recuperação de erros são remover um caráter estranho, inserir um caráter ausente, substituir um caráter incorreto por um correto ou transpor dois caracteres adjacentes.

3.2 Análise Sintática

O analisador sintático obtém uma cadeia de *tokens* proveniente do analisador léxico e verifica se a mesma pode ser gerada pela gramática da linguagem-fonte. Esperamos que a

análise sintática relate quaisquer erros de sintaxe de uma forma inteligível. Deve, também, recuperar-se dos erros que ocorram mais comumente, a fim de poder continuar processando o resto de sua entrada.

Os métodos de análise mais eficientes, tanto *top-down* quanto *bottom-down*, trabalham somente em determinadas subclasses de gramáticas, mas várias dessas subclasses, como as das gramáticas LL (Linguagem Livre de Contexto) e LR (Linguagem Recursiva), são suficientemente expressivas para descrever a maioria das construções sintáticas das linguagens de programação. Os analisadores implementados manualmente trabalham freqüentemente com gramáticas LL. Os da classe mais ampla das gramáticas LR são usualmente construídos através de ferramentas automatizadas (AHO,1995).

3.2.1 Tratamento dos Erros de Sintaxe

Se um compilador tivesse que processar programas corretos, seu projeto e sua implementação seriam grandemente simplificados. Mas programadores freqüentemente escrevem programas incorretos e um bom compilador deveria assistir o programador na identificação e localização de erros. É gritante que, apesar dos erros serem lugares-comuns, poucas linguagens sejam projetadas, tendo-se o tratamento de erros em mente. A maioria das especificações das linguagens de programação não descreve como um compilador deveria responder aos erros; tal tarefa é deixada para o projetista do compilador. O planejamento do tratamento de erros exatamente desde o início poderia tanto simplificar a estrutura de um compilador quanto melhorar sua resposta aos erros.

Sabe-se que os programas podem conter erros em muitos níveis diferentes. Por exemplo, os erros podem ser:

- léxicos, tais como errar a grafia de um identificador, palavra-chave ou operador;
- sintáticos, tais como uma expressão aritmética com parênteses não balanceados;
- semânticos, tais como um operador aplicado a um operando incompatível;
- lógicos, tais como uma chamada infinitamente recursiva.

3.2.2 Estratégia de recuperação de erros

Há muitas estratégias gerais diferentes que um analisador sintático pode empregar para se recuperar de um erro sintático. Apesar de nenhuma delas ter comprovado ser universalmente aceitável, uns poucos métodos têm ampla aplicabilidade. Introduzimos aqui as seguintes estratégias: modalidade do desespero; nível de frase; produção de erros; correção global (LOUDEN; KENNETH 2004).

3.2.3 Regras Gramaticais

As gramáticas podem descrever a maioria das sintaxes das linguagens de programação. Uma parte da análise sintática é realizada pelo analisador léxico, na medida em que produz uma sequência de *tokens* a partir dos caracteres de entrada. As seqüências de *tokens* aceitas por um analisador sintático formam um subconjunto de uma linguagem de programação; as fases subseqüentes precisam analisar a sua saída para assegurar a concordância com as regras que não são checadas pelo analisador sintático.

3.2.4 Análise sintática top-down

A análise sintática *top-down* pode ser vista como uma tentativa de se encontrar uma derivação mais à esquerda para uma cadeia de entrada. Equivalentemente, pode ser vista como uma tentativa de se construir uma árvore gramatical, para a cadeia de entrada, a partir da raiz, criando os nós da árvore gramatical em pré-ordem (AHO, 1995).

A análise sintática *top-down* permite o processo de esquadrinhamento repetido da entrada, chamado de descendência recursiva sendo solicitado para analisar sintaticamente construções de linguagens de programação.

3.2.5 Análise Sintática bottom-up

A análise sintática *bottom-up* é conhecida como análise de empilhar e reduzir. Uma forma fácil de implementá-la é chamada de análise sintática de precedência de operadores (AHO, 1995).

A análise gramatical de empilhar e reduzir tenta construir uma árvore gramatical para uma cadeia de entrada começando pelas folhas (o fundo) e trabalhando árvore acima em direção à raiz (o topo). Pode-se pensar neste processo como o de “reduzir” uma cadeia W ao símbolo de partida de uma gramática. A cada passo de redução, uma subcadeia particular, que reconheça o lado direito de uma produção, é substituída pelo símbolo à esquerda daquela produção e, se a subcadeia tiver sido escolhida corretamente a cada passo, uma derivação mais à direita terá sido rastreada na ordem inversa.

3.3 Análise Semântica

Um compilador usa uma tabela de símbolos (ts) para guardar informações sobre os nomes declarados em um programa. A ts é pesquisada cada vez que um nome é encontrado no programa-fonte. Alterações são feitas na ts sempre que um novo nome ou nova informação sobre um nome já existente é obtido. A gerência da ts de um compilador deve ser implementada de forma a permitir inserções e consultas da forma mais eficiente possível, além de permitir o crescimento dinâmico da mesma.

3.3.1 Entradas nas tabelas de símbolos

Cada entrada na ts é a declaração de um nome. O formato das entradas pode não ser uniforme (embora seja mais fácil manipular entradas uniformes) porque as informações armazenadas para cada nome podem variar de acordo com o tipo/uso do nome. Cada entrada na ts pode ser implementada como um registro (*record* ou *struc*) contendo campos (nome, tipo, classe, tamanho, escopo, etc.) que a qualificam.

Se o número máximo de caracteres em um nome for limitado e pequeno (até 16 caracteres), o campo *NOME* de uma entrada da ts pode ser simplesmente um arranjo de caracteres (alocação estática).

Se o número de caracteres de um nome for grande (maior que 16), o campo *NOME* de uma entrada da ts deve ser um apontador para uma área alocada dinamicamente.

3.3.2 Alocação de posição de memória

A informação sobre que endereços de memória serão associados aos nomes (variáveis) deve ser guardada na *ts*. No caso de nomes globais, a estratégia é muito simples: ao primeiro nome global, associa-se o endereço relativo 0 (zero); ao segundo, o endereço 0 + ‘espaço ocupado pelo primeiro nome global’; e assim sucessivamente.

No caso de nomes locais (e parâmetros de subprogramas), a alocação é feita geralmente em uma pilha e o endereçamento pode ser feito de forma semelhante ao usado com os nomes globais, lembrando-se de que o endereço relativo 0 (zero) é, na realidade, 0 + valor do apontador de pilha na hora de execução do programa.

3.3.3 Implementação da tabela de símbolos

Pode-se implementar uma tabela de símbolos como uma lista linear de registros cujos nomes novos são inseridos no fim e a pesquisa sempre se processa do fim para o início. Convém fazer algumas observações com relação à busca e inserção de nomes da *ts* (AHO, 1995). Sempre que encontramos um nome de identificador em um programa-fonte, ele pode estar sendo declarado ou sendo usado pelo programador.

Se estiver sendo declarado é necessário analisar duas situações:

- em uma linguagem com estrutura de blocos, devemos pesquisar a *ts* do fim até o ponto onde começaram a ser inseridos os símbolos do bloco mais interno;
- em uma linguagem sem estrutura de blocos, devemos pesquisar a *ts* do fim até o início.

Em ambos os casos, o nome não pode estar na *ts* (se estiver, é um símbolo sendo redeclarado), devendo ser inserido. Se estiver sendo usado pelo programador, devemos pesquisar a *ts* do fim até o início; se não for encontrado, é um símbolo indefinido.

Em uma tabela com n entradas, podemos considerar que, na média, pesquisamos $n / 2$ nomes para verificar se um dado nome está ou não na *ts*.

Dado que inserções e consultas à *ts* levam um tempo proporcional a n , o tempo total para inserir n nomes e fazer e consultas à *ts* pode ser considerado $c * n * (n + e)$, em que c é uma constante representando o tempo médio necessário para uma inserção ou consulta (o tempo de pesquisar $n / 2$ entradas).

Em um programa de tamanho médio, poderíamos ter $n = 100$ e $e = 1000$, de modo que $c * 100 * (100 + 1.000) = c * 1.100.00$. Se c fosse, por exemplo, 0,001 segundo, então o tempo total gasto na gerência da ts seria de 110 segundos. Pode parecer pouco, mas considerando agora que n e e foram multiplicados por 10, teríamos o custo multiplicado por 100 (LOUDEN; KENNETH, 2004).

Por essa razão, em geral, tabelas de símbolos são implementadas como tabelas de acesso direto (*hash table*).

Uma tabela de acesso direto consiste basicamente de um arranjo de tamanho fixo com m apontadores para entradas de tabela (idealmente um número primo) e entradas de tabela organizadas em m listas encadeadas separadas, chamadas buques (*buckets*). Cada entrada da tabela de símbolos aparece em somente uma dessas listas.

Em um tabela hash, o tempo de realização de n inserções e e consultas pode ser considerado $c * n * (n + e) / m$, em que c é uma constante representando o tempo médio necessário para uma inserção ou consulta.

Desde que m pode ser feito tão grande quanto desejado, até mesmo n , esse método é, via de regra, mais eficiente que listas lineares e é usado na maioria das implementações de compiladores.

3.4 Compatibilidade de tipos

Já foi visto como é feita a verificação semântica de identificadores com relação à tabela de símbolos. A seguir, será visto como se faz a verificação semântica de compatibilidade de tipos em expressões e comandos de atribuição.

3.4.1 Expressões Aritméticas

Em uma expressão aritmética, do tipo $b + c * d$, precisamos verificar se:

- a) $c * d$ pode ser feito e qual o tipo do resultado produzido;
- b) $b + (c * d)$ pode ser feito e qual o tipo do resultado produzido.

Uma forma simples de fazer tal verificação é associar ações semânticas às regras que geram expressões de tal forma que, a cada aplicação de uma regra que gera uma operação aritmética básica (soma, subtração, multiplicação e divisão, por exemplo), seja verificada a compatibilidade dos tipos dos operandos envolvidos e se determine o tipo do resultado para uso posterior. O armazenamento dos tipos resultantes intermediários é feito em uma estrutura de dados tipo pilha, chamada de Pilha de Controle de Tipos (PcT) (HUNTER, 1986).

A seguir, um exemplo com a gramática:

$EA ::= T E'$

$E' ::= + T E' \mid \&$

$T ::= F T'$

$T' ::= * F T' \mid \&$

$F ::= \text{ident} \mid \text{número_inteiro} \mid \text{número_real} \mid (EA)$

Quando o analisador sintático aplicar uma das regras, sabe-se o tipo de cada operando possível (de identificador, obtém-se com consulta à tabela de símbolos; de número, é informado pelo analisador léxico).

$F ::= \text{ident}$

$F ::= \text{número_inteiro}$

$F ::= \text{número_real}$

Simplesmente, deve-se, então, empilhar o tipo do operando na PcT. A ação semântica associada a cada regra seria empilhar o tipo do operador na PcT.

Quando o analisador sintático aplicar uma das regras, sabe-se que uma operação básica de adição/multiplicação foi reconhecida e pode-se verificar a compatibilidade de tipos analisando o topo e o subtopo da PcT.

$E' ::= + T E'$

$T' ::= * F T'$

A ação semântica associada às operações $+$ e $*$ poderia ser:

se topo = inteiro E subtopo = inteiro, então

resultado = inteiro

senão

se topo = real E subtopo = real, então

resultado = real

senão Erro: incompatibilidade de tipos para linguagens de programação que não admitem misturas de tipos como Pascal.

3.5 Geração de código intermediário

Segundo Louden e Kenneth (2004), tomando um compilador como um modelo de análise-e-síntese, pode-se considerar que seus módulos iniciais (*front-end modules*) traduzem um programa-fonte em uma representação (linguagem) intermediária, a partir da qual seus módulos finais (*back-end modules*) geram o código-objeto final.

Embora um programa-fonte possa ser traduzido diretamente para a linguagem-objeto, o uso de uma representação intermediária, independentemente de máquina, tem as seguintes vantagens:

- Reaproveitamento de código, facilitando o transporte de um compilador para diversas plataformas de *hardware*; somente os módulos finais precisam ser refeitos a cada transporte;
- Um otimizador de código independente de máquina pode ser usado no código intermediário.

AHO (1995) afirma que há várias formas de representação de código intermediário. Duas das mais comuns são árvores de sintaxe e triplas de código intermediário (ou instruções de três endereços).

3.5.1 Árvores de sintaxe

Segundo Aho (1995), uma árvore de sintaxe mostra conforme Figura 4, a estrutura hierárquica de um programa-fonte. Por exemplo, a atribuição $a = b * c + b * -d$ seria representada graficamente por:

Figura 4 - Árvore Sintática para uma atribuição

Fonte: (AHO, 1995)

3.5.2 Triplas de código intermediário

Triplas de código intermediário são seqüências de comandos da forma:

$$x := y \text{ op } z$$

em que x , y e z são variáveis ou constantes definidas pelo programador ou variáveis temporárias geradas pelo compilador, op define um operador (aritmético, relacional ou lógico) e $:=$ define atribuição.

A escolha dos operadores disponíveis é de fundamental importância no projeto de uma linguagem intermediária.

O conjunto de operadores deve ser claro e rico o suficiente para implementar todas as operações da linguagem-fonte.

Um conjunto pequeno, se por um lado simplifica o transporte do compilador para novas plataformas, por outro força o gerador de código intermediário a gerar longas seqüências de triplas para comandos da linguagem-fonte e o otimizador de código intermediário trabalha muito para produzir um bom código final (LOUDEN; KENNETH, 2004).

3.5.3 Expressão aritmética e atribuição

Iniciando com expressão aritmética e comando de atribuição, considerando a gramática abaixo:

$$A ::= \text{id} := E$$

$$E ::= T E'$$

$$E' ::= + T E' \mid \&$$

$$T ::= F T'$$

$$T' ::= * F T' \mid \&$$

$$F ::= \text{id} \mid \text{nro_int} \mid \text{nro_real} \mid (E)$$

uma forma simples de geração de código intermediário aproveita o mesmo mecanismo usado na verificação de tipos, em que uma pilha de controle de tipo (PcT) é usada durante o reconhecimento de uma expressão, além de verificar o tipo dos operandos, gerar código intermediário simultaneamente, criando uma pilha de controle de operandos (PcO) que vai conter os operandos usados nas operações realizadas em uma expressão aritmética.

Quando o analisador sintático aplicar as regras simplesmente empilhar-se-ão na PcO o identificador, número inteiro ou número real correspondente (ou indicador de onde se encontra esse operando).

$$F ::= \text{id} \text{ ou } F ::= \text{nro_int} \text{ ou } F ::= \text{nro_real}$$

Quando o analisador sintático aplicar as regras, sabe-se que ele reconheceu uma operação de adição ou multiplicação e que, não havendo problemas de compatibilidade de tipos, pode gerar código intermediário para a realização de tais operações. Para isso, basta utilizar os operandos que estiverem no topo e subtopo da PcO, deixando o resultado em uma variável temporária que deve ser deixada na PcO.

3.5.4 Comandos de controle de fluxo

Comandos de controle de fluxo podem ser basicamente resumidos aos seguintes:

```
<cmdo> ::= if <cond> then <cmdo> <pelse> fi
| while <cond> do <cmdo> endwhile
| for id = <expr-início> to <expr-limite> <step>
<pelse> ::= else <cmdo>
| &
<step> ::= step <expr-incremento>
| &
```

Para tratá-los, usa-se uma pilha de controle de rótulos (*labels*) e uma tabela de rótulos, necessários para resolver o endereço de um rótulo referenciado antes de ser definido.

Considera-se que, ao final da geração do código intermediário, a tabela de rótulos contenha somente entradas resolvidas (com rótulo e endereço) para que se faça a respectiva substituição no código intermediário.

3.6 Geração de Código-objeto

Enquanto a fase de análise (léxica, sintática, semântica) é essencialmente dependente da linguagem de programação em questão, a fase de geração de código é, ao contrário, dependente principalmente da máquina-alvo, a máquina em que o código gerado deverá ser executado eventualmente. Em princípio, esta não precisa ser a máquina em que a compilação se realiza, porque poderemos precisar gerar código para máquinas que, por seu tamanho ou por outras características, não servem como plataforma para execução do compilador.

A divisão de um compilador em *front-end* (análise) e *back-end* (geração e otimização de código) reflete estas características: em princípio, é possível escrever um *front-end* para cada linguagem de programação e um *back-end* para cada máquina-alvo.

O gerador de código é a parte do compilador que gera o programa objeto ou *assembler*. O código é gerado sempre para determinadas unidades sintáticas, sendo utilizadas informações fornecidas pelo analista de contexto.

Fazer um gerador de código que produza programas-objetos realmente eficientes é uma das partes mais difíceis da construção de um compilador, tanto do ponto de vista prático como teórico.

Por exemplo,

a) *Código comum:*

temp1 := inttoreal(60)

temp2 := id3 * temp1

temp3 := id2 + temp2

id1 := temp3

b) *Código otimizado:*

temp1 := id3 * 60.0

id1 := id2 + temp1

c) *Código objeto:*

MOVF id3, R2

MULF #60.0, R2

MOVF id2, R1

ADDF R2, R1

MOVF R1, id1

Vale ressaltar que, no código gerado em *assembler*, o 1.º operando da instrução representa a fonte; o 2.º operando da instrução representa o destino; o carácter F indica que a instrução trabalha com ponto flutuante e o símbolo # indica que 60.0 deve ser tratado como constante.

Segundo Grune (2001), a compilação produz o código-objeto a partir da árvore de código intermediário e que a base da geração de código é a substituição sistemática de nós e subárvores de uma árvore sintática por segmentos de código de destino, de tal forma que a semântica seja preservada.

Para complementar o capítulo que aborda conceitos, técnicas e fases de um Compilador, o Capítulo 4 contempla algumas ferramentas em C para execução em *FPGA* que são estudadas nesta tese e de grande valia como apoio para o processo de desenvolvimento do projeto proposto. As ferramentas descritas a seguir utilizam de métodos de compilação apoiando-se nos conceitos descritos pelo Capítulo 3.

CAPÍTULO 4 – FERRAMENTAS DE PROGRAMAÇÃO C PARA PLATAFORMAS FPGA

Segundo Lopes (2007), ferramentas já disponíveis, como o *ISE* (Xilinx, 2007) e *EDK* (Xilinx, 2007), permitem implementar, testar, simular e corrigir projetos eletrônicos computacionais com grande facilidade sem que se tenham gastos elevados como no desenvolvimento de *ASICs*. É possível simular a funcionalidade do projeto antes de produzi-lo, evitando assim desperdícios de recursos.

Lopes (2007) ainda ressalta que hoje, é possível desenvolver rapidamente um projeto de sistema digital empregando-se novas metodologias como linguagens de descrição de *hardware*, as HDLs, ferramentas de síntese e lógica e simulação. Não é mais necessário desenhar portas lógicas individuais e montar diferentes componentes. As linguagens de descrição de *hardware* estão se consolidando como forma padrão de descrição de projetos.

Ferramentas de síntese lógica automática estão disponíveis para mapear circuitos em diversas tecnologias. Além das mudanças na tecnologia, o ciclo de vida dos produtos modernos está tornando-se mais curto que os tradicionais ciclos de projeto, exigindo uma rápida prototipação.

Apesar de novas arquiteturas terem sido propostas e novos conceitos apresentados (RADUNOVIC, et al., 1998), a utilização dos sistemas computacionais reconfiguráveis requer etapas morosas e complexas e conhecimentos específicos de projeto de *hardware* (CARDOSO, et al., 2003).

Este capítulo destina-se a investigar algumas ferramentas comerciais e acadêmicas, disponíveis para conversão de algoritmos escritos em linguagens de alto nível, como C, C++ e *Java* para linguagens de descrição de *hardware*.

Serão apresentadas algumas ferramentas, funcionalidades e objetivos.

4.1 Introdução a Ferramentas de Programação C para Plataformas FPGA

Muitas aplicações embutidas, geralmente aplicações científicas, de processamento de imagens, entre outras, são escritas em linguagem C e são executadas por processadores DSP.

Quando detectados gargalos nessas aplicações, os trechos de código mais críticos são escritos em linguagens assembler, aumentando assim o desempenho da aplicação (LOPES, 2007).

Há casos em que a aplicação exige tempo crítico para executar, por exemplo, aplicações de processamento de imagens de tempo real, a solução então, é reescrever essa aplicação ou parte dela em linguagem de descrição em *hardware* e implementar essa aplicação em um *CI*, por exemplo, um *FPGA*.

O problema é que para reescrever essa aplicação em linguagem de descrição de *hardware* exige-se o conhecimento específico para tal finalidade, além do conhecimento exigido em ferramentas *EDA* para prototipação em *FPGA*, como dito acima.

Para Lopes (2007) as ferramentas de programação C para plataformas *FPGA* podem ajudar na diminuição do tempo para reescrever esses códigos em linguagem C para HDLs, além do tempo, elas diminuem também os custos do projeto, já que não há necessidade de se contratar uma equipe ou empresa especializada em projetos de *hardware*.

Os projetistas de *software*, com a ajuda dessas ferramentas, poderão ver os *FPGAs* como apenas mais um sistema programável, por exemplo, um processador. (PELLERIN, SCOTT, 2005)

As HDLs oferecem ao projetista de *hardware* o controle explícito e preciso sobre o *hardware* gerado. Porém, esse controle explícito sobre a geração de *hardware* nem sempre é necessário. Do mesmo modo que um programa é escrito em linguagem C e posteriormente em assembler, contendo possíveis trechos críticos do sistema, um novo sistema poderia ser escrito em linguagem HDL utilizando ferramentas automáticas e depois serem reescritas por um projetista de *hardware*, todos os trechos que causassem gargalo, caso fosse necessário. (PELLERIN, SCOTT, 2005)

De qualquer forma, essas ferramentas permitem ao projetista de *hardware* explorar o espaço do projeto, por exemplo, movendo porções do código para a parte reconfigurável do *FPGA* ou para um processador embarcado, tal como *PowerPC*. Isso permite ao projetista fazer protótipos e explorar a melhor alternativa de implementação do sistema na plataforma *FPSoC*. Para a computação reconfigurável ser adotada como um paradigma de computação é necessário o desenvolvimento de ferramentas que acelerem o desenvolvimento de *hardware* e que eleve o nível de abstração necessário ao projeto. (LOPES, 2007)

4.2 Pico Express

Pico Express é comercializado pela empresa *Synfora*, fundada em janeiro de 2003. (SYNFORA, 2007)

Segundo o fabricante, a ferramenta possibilita a geração automática de arquiteturas otimizadas e sintetizáveis *RTL* (*Register Transfer Level*) a partir de algoritmos escritos em linguagem *ANSI-C*, e combina *IP* (*Intellectual Property*) que são blocos lógicos pré-fabricados, sintetizados e testados que podem ser utilizados na construção de projetos de sistemas em *FPGA* configuráveis que são projetados para uma implementação eficiente de algoritmos complexos.

O *Pico Express* consegue a partir de um algoritmo, produzir múltiplos projetos com diferentes características, tais como custo e desempenho. Um exemplo pode ser vista na Figura 5.

Um algoritmo *MPEG* (*Moving Picture Experts Group*) pode ser implementado de um modo para um *DVD* (*Digital Versatile Disc*) player e de outro modo para uma câmera de vídeo. O algoritmo é o mesmo, porém, a característica da arquitetura muda. (SYNFORA, 2007)

A ferramenta tem como entrada um algoritmo em linguagem *C* e um arquivo de restrições sobre custo de área e tempo de ciclos, definidos pelo usuário; explora o paralelismo em vários níveis do algoritmo para criar um grupo de alternativas de implementação com diferentes graus de paralelismo. Cada grupo de implementações geradas tem uma característica diferente, em relação à área, desempenho e energia consumida pelo *hardware* gerado, implementações são construídas com a ajuda dos *Ips* configuráveis. Após isso, o programa determina a área e o desempenho de cada implementação e permite ao usuário decidir qual a melhor implementação para o seu sistema. Com isso, ao usuário são oferecidas algumas alternativas de implementações, cada uma tendo a melhor área de implementação e desempenho. Essas alternativas de implementações ficam em uma lista de possíveis implementações. (LOPES, 2007)

Figura 5 – Possíveis implementações geradas por *Pico Express*

Fonte: (LOPES, 2007)

Para uma implementação selecionada pelo usuário é criado o arquivo *RTL* e arquivos *testbenches*, necessários para testar em simuladores o arquivo *RTL* gerado e implementar o *hardware* no *FPGA*.

O resultado da construção do *RTL* é correto, porque *Pico Express* tem *IPs* pré-configurados para adequar as funcionalidades requeridas. Cada *IP* configurável é projetado para atingir as necessidades de um estilo computacional específico e para ser eficiente em área/performance para o dado estilo. Cada *IP* é projetado para ser escalonável e suportar diferentes níveis de paralelismo.

Lopes (2007) mostra que há dois tipos de *IPs* inclusos no *Pico Express*: os não-programáveis (fixos) e programáveis e são altamente paralelos. Esses *IPs* são totalmente compatíveis com outros blocos no projeto. O *IP* não-programável é um *PPA* (*Pipeline of Processor Arrays*) projetado para computações intensivas do código C, tais como multiplicações. O *IP* programável é uma arquitetura *VLIW* (*Very Long Instruction Width*) que é usada para acelerar o controle intensivo do código C.

Pico permite uma junção do *PPA* e do *VLIW* para criar uma solução completa para algoritmos mais complexos, como por exemplo, *MPEG*.

A Figura 6 mostra o funcionamento geral do *Pico Express*. Como pode ser observado, *Pico Express* analisa o algoritmo escrito em linguagem C e transforma as partes que necessitam de um processamento mais pesado, tais como multiplicação, em blocos que executam diretamente no *FPGA* – “*accelerator 1 e 2*” as partes do código que não necessitam

de um processamento tão intenso, são compiladas e executadas em um processador de propósito geral embarcado no *FPGA*. (SYNFORA, 2007)

Figura 6 - Visão geral do Pico Express

Fonte: (SYNFORA, 2007)

Figura 7 – Visão detalhada do Pico Express

Fonte: (SYNFORA, 2007)

Na Figura 7 pode ser visto detalhadamente o funcionamento do *Pico Express*. A ferramenta analisa um código escrito em *ANSI-C* e, ao apontar as partes críticas do código, gera o *hardware* para elas em *FPGA* e o driver com as interfaces do *hardware*. Da parte menos crítica do *software*, o compilador gera um arquivo objeto que pode ser executado diretamente no processador embarcado, além de gerar o driver em *software*. Os *drivers* servem para a comunicação entre as partes do programa que executam no *FPGA* e no processador.

O *hardware* customizado é então chamado pelo programa no processador sob a forma de instrução. Pode se dizer que o *Pico Express* gera um ambiente *hw/sw codesign*.

No ambiente *hw/sw codesign*, como explicado acima, o código original é dividido. A parte do *software* que necessita de mais desempenho é sintetizada no *FPGA* e executada como *hardware* puro, enquanto que a parte que não necessita de tanto desempenho é compilada e executada sob a forma de programa no processador de propósito geral embarcado no *FPGA*. (LOPES, 2007)

4.3 Galadriel e NENYA

Galadriel e *NENYA* são compiladores que atuam em série e tem como objetivo gerar um *hardware* reconfigurável constituído por um *FPGA* acoplado a uma ou mais memórias a partir de um algoritmo representado por *bytecodes* de *Java*. (CARDOSO, 2000). Um exemplo da arquitetura do *hardware* gerado pelos compiladores *Galadriel* e *NENYA* pode se visto na Figura 8. As memórias permitem que pequenas partes da execução de um programa possam ser computadas diretamente no *FPGA*, sem intervenção do processador. Desse modo, não são necessárias as transferências dos dados entre memória e processador. Na Figura 8, sistema de hospedagem é um computador, por exemplo, o computador ligado a uma placa de prototipação.

Figura 8 – Arquitetura do hardware gerado pelos compiladores Galadriel e NENYA

Fonte: (LOPES, 2007)

Segundo Cardoso (2000), a adoção da linguagem *Java* como linguagem de programação de sistemas computacionais reconfiguráveis se deve a:

- Simplicidade da linguagem;
- Nível de abstração elevado;
- Suporte a programação concorrente, pela utilização de *threads*;
- Portabilidade entre diversos sistemas operacionais;
- Sintaxe parecida com a da linguagem C/C++.

E a utilização dos *bytecodes* de *Java*, em particular, se deve a:

- Manter quase todas as informações do programa fonte;
- É um modelo de execução na *internet*;
- Suporta várias linguagens de programação de *software*. Além do fato da representação em *bytecodes* de uma aplicação ser fruto de algumas etapas iniciais de compilação, foi um dos pontos-chaves, segundo (CARDOSO, 2000), que originou a adoção desse modelo como formato de entrada para o compilador *Galadriel*.

O compilador *Galadriel* recebe um algoritmo em *bytecodes* do *Java*, efetua várias etapas que produzem alguns arquivos intermediários. O objetivo do compilador *Galadriel* é extrair dos *bytecodes* do *Java*, o paralelismo existente através de alguns métodos empregados pelo compilador.

Nenya é um compilador que gera a arquitetura mostrada na Figura 8, a partir dos arquivos intermediários gerados pelo compilador *Galadriel*.

A Figura 9 representa o fluxo completo de compilação executados por *Galadriel* e *Nenya*. Nessa Figura também é possível ver que os *bytecodes* do *Java* são as entradas do compilador de anteguerra (*Galadriel*) e que esse gera como produtos o *HPDG* (Hierarchical Program Dependence Graph) e o *DFG* (*DataFlow Graph*), que irão em seguida servir de entrada para o compilador *Nenya*, o qual produzirá o *hardware* reconfigurável, como na Figura 8.

Vários graus de paralelismo são expostos pelo compilador *Galadriel* para que se obtenham implementações eficientes em *hardware* reconfigurável. Também é feita a análise de inferência do número de bits suficientes para cada operando de forma a preservar a funcionalidade original. (LOPES, 2007)

Figura 9 – Fluxo de compilação feitos por Galadriel e Nenya

Fonte: (LOPES, 2007)

Na Tabela 2 a seguir é mostrado o subconjunto de instruções correspondentes ao *JVM* (*Java Virtual Machine*) do *Java* que são corretamente suportadas pelo compilador *Nenya*. (CARDOSO, 2000)

Tabela 2– Tipo de dados, operações e controles suportados pelo compilador Galadriel

Fonte: (CARDOSO, 2000)

TIPOS DE DADOS	OPERAÇÕES	CONTROLE E OUTROS MECANISMOS
----------------	-----------	------------------------------

Boolean	/, %, *,	Tratamento de exceções
Byte	+, -, +, -	Invocação de métodos
Short	, < <, > >,	Criação de objetos
Int	> > >,	Criação de <i>arrays</i> <i>unidimensionais</i>
Char	, &&, &,	
Long	, ^, ~,	While, for, do while
Float, double	<, >, <=,	Break, continue
Referências a <i>arrays</i> <i>unidimensionais</i>	>=, ==, !=	If, ? :, switch
Referências a objetos		Conversões (cast)

Cada tipo primitivo da linguagem *Java* (*byte, short, int, boolean, long e char*) é traduzido para o tipo correspondente em *VHDL*. (CARDOSO, 2000)

Nos compiladores *Galadriel* + *Nenya* o programador escolhe uma função ou segmento de código para ser gerado o *hardware* correspondente a ela.

Basicamente o compilador *Galadriel* através de passos de compilação, gera arquivos intermediários, os quais exibem explicitamente o paralelismo em nível de operação, entre os blocos básicos e entre ciclos considerando sempre a granulosidade da operação e permite a exploração eficiente de implementações em *hardware* específico. Para tal, são utilizados dois modelos de representação: *HPDF* e *DFG*, os quais permitem lidar com a hierarquia definida pelos ciclos no programa fonte e com o paralelismo ao nível da operação nos agrupamentos de blocos básicos. (CARDOSO, 2000)

O compilador *Galadriel* considera o escopo do bloco de código especificado pelo programador para ser compilado para *hardware* reconfigurável. O usuário pode utilizar rótulos especiais (*pragmas*) para identificar a região do código *Java* que deseja compilar para *hardware* reconfigurável. (CARDOSO, 2000)

Dos arquivos intermediários com representações, gerados pelo compilador *Galadriel*, o *HPDG* é um grafo que exhibe explicitamente as dependências de dados, fluxo de controles e os ciclos existentes no programa fonte.

O *DFG* global incorpora explicitamente os predicados (condições segundo as quais determinada instrução é executada) para operações cuja execução dependa de construções condicionais acíclicas. Um predicado de uma operação define se essa operação será executada com base na avaliação de uma função lógica cujas variáveis são resultados de comparações. Ou seja, o *DFG* global representa as operações do programa original e a forma como estão interligadas e com algumas transformações, permite fornecer a imagem da unidade de dados específica.

Segundo Cardoso (2000), “as análises consideradas pelo compilador *Galadriel* são *intraprocedimentais*, ou seja, apenas são realizadas ao longo da estrutura interna dos métodos e não incluem análises *interprocedimentais* (entre métodos/funções)”.

O compilador *Nenya* gera um *hardware* reconfigurável especializado a partir dos modelos de representação apresentados pelos arquivos gerados pelo compilador *Galadriel*. *Nenya* suporta a compilação para uma arquitetura constituída por um *FPGA* acoplado a uma ou mais memórias *RAM* via barramentos independentes, a Figura 10 é um exemplo da arquitetura gerada pelo compilador *Nenya*. (CARDOSO, 2000)

Segundo Lopes (2007), *Nenya* pode ser usado para compilar um programa completo ou segmento de código para *hardware* reconfigurável, desde que o programa respeite o subconjunto de instruções *Java*, apresentados na Tabela 2.

Antes da execução, o compilador *Nenya* necessita das especificações do ciclo de *clock* do *hardware* reconfigurável, das memórias externas, da área máxima do *FPGA*, e da caracterização dos componentes responsáveis pelos operadores existentes no programa fonte em linguagem *Java*.(CARDOSO, 2000)

O compilador *Nenya* recebe os grafos gerados pelo compilador *Galadriel* para um dado método em *Java* e a identificação do segmento de código desse método que o programador deseja compilar para *hardware* reconfigurável.

A Figura 10 como descrito acima, mostra a estrutura utilizada pelo compilador *Nenya* para implementar o *hardware* reconfigurável relativo a uma dada descrição em linguagem *Java*. Nessa Figura é possível ver duas unidades: unidade de controle e unidade de dado.

Essas unidades interagem entre si. A unidade de dados pode incorporar, por exemplo, componentes que acessem dispositivos externos ao *FPGA* – quadrado preto na Figura 10. Um desses componentes é a macrocélula de acesso a uma memória externa. (CARDOSO, 2000)

O endereçamento de dados armazenados em uma memória é realizado por circuitos especializados (constituídos por macrocélulas) presentes na unidade de dados. O compilador baseia-se numa biblioteca de macrocélulas parametrizáveis, definidas sob a forma de geradores de circuitos, para produzir a unidade de dados.

A unidade de controle é responsável pelo controle dos acessos a memórias externas, pelo controle de escritas em registros e pela execução correta dos ciclos. (CARDOSO, 2000)

Figura 10 – Estrutura utilizada pelo compilador NENYA para implementar o hardware gerado pelo compilador NENYA

Fonte: (CARDOSO, 2000)

O compilador NENYA baseia-se em uma biblioteca de macrocélulas sob a forma de geradores de circuitos para produzir a unidade de dados. (CARDOSO, 2000)

A biblioteca inclui operadores inteiros (aritméticos, lógicos, comparadores, deslocadores), uma macrocélula de acesso a memórias externas acopladas ao *FPGA*, multiplexadores, entre outros. (LOPES, 2007)

Cada operador tem uma descrição estrutural em linguagem *VHDL* com as seguintes propriedades:

- Parametrização: de acordo com o número de bits dos operandos de entrada/saída é gerado um circuito especializado.
- Colocação relativa: A colocação do elemento no *FPGA* é sempre relativa ao posicionamento do nível imediatamente a seguir na hierarquia de componentes, Figura 11.

Figura 11 – Colocação relativa de células no FPGA

Fonte: (CARDOSO, 2000)

A Figura 11 mostra um exemplo da colocação relativa de elementos lógicos no *FPGA*. A Figura 11-a mostra um exemplo da descrição do programa de usuário em *Java*. Na Figura 11-b é mostrado o exemplo da colocação relativa. Como mostra a Figura 11, a porta *OR* de duas entradas foi especificada depois da porta *AND* de duas entradas. A propriedade de colocação relativa, a porta *OR* é ligada logo e após a porta *AND*.

Segundo Cardoso (2000), a colocação relativa permite encaminhamentos com menores variações, e por isso maiores previsibilidades nos atrasos das ligações, e reduz os tempos de colocação e encaminhamento globais.

4.4 Compilador apresentado por Chichkov

O compilador apresentado por *Chichkov*, em (CHICHKOV, BELTRÁN, 1998) teve a sua primeira publicação em 1997. (CHICHKOV, BELTRÁN, 1997)

Em Chichkov, Beltrán (1998) é apresentado um compilador que utiliza um subconjunto de comandos da Linguagem *C* e que ajuda no processo de *hardware/software codesign* de sistemas digitais.

O compilador apresentado por *Chichkov* apresenta grandes limitações no suporte a construções da linguagem *C*. (CARDOSO, 2000)

Esse compilador tem por objetivo traduzir uma especificação escrita em linguagem em *ANSI-C* e transformar essa descrição em dois arquivos: um *VHDL* e um programa *C* que juntos executarão a mesma função de entrada, mas com um significativo ganho de desempenho. (CHICHKOV, BELTRÁN, 1998)

A metodologia completa do compilador é mostrada na Figura 12 e consiste das seguintes partes principais: um *front-end GNU C*; um compilador paralelizável (*parallelizing compiler*); algoritmo de particionamento em *hardware/software codesign* e uma ferramenta de implementação.

Primeiramente a linguagem de programação *ANSI-C* é usada como entrada do compilador. Durante a compilação do programa escrito em linguagem C, o código no formato *RTX* (*three-address code*) é gerado. Este código é usado na fase de modelagem e análise do sistema, para gerar o arquivo *VHDL* comportamental da parte do código original que é implementado em *hardware* específico. O compilador usa o modelo de fluxo de grafos para capturar o comportamento do sistema. Esse modelo de fluxo de grafos tem a capacidade de descrever as estruturas seqüenciais e paralelas, correspondentes a clássica descrição *hardware* e *software*. (CHICHKOV, BELTRÁN, 1998)

Figura 12 - Arquitetura de configuração do compilador

Fonte: (CHICHKOV; BELTRÁN, 1998)

Após ter gerado o código no formato *RTX*, esses resultados intermediários gerados pelo *GCC Front-end* vão servir de entrada para a fase “*Parallelism Extration*”, que como o nome diz, tem a função de extrair o paralelismo dos arquivos correspondentes aos resultados intermediários, gerados na fase anterior. Os arquivos correspondentes aos resultados da fase “*Parallelism Extration*” então, servirão como arquivos de entrada para a fase “*System Partitioning*”, o qual é responsável por fazer o particionamento em *hw/sw codesign*. Após essas fases, os arquivos originados na fase anterior serão utilizados na fase “*System Synthesis*”, no qual o arquivo *VHDL* será gerado. (CHICHKOV, BELTRÁN, 1998)

O compilador possui algumas limitações, como por exemplo, não aceita como entrada, descrições em linguagem C, tais como *arrays*, ponteiros, chamadas a funções, recursividade, entre outras. (CARDOSO, 2000)

O compilador GCC foi usado como *front-end* para: assegurar a correlação léxica e sintática do código fonte, gerar o “*three-address code*” do arquivo de entrada e aproveitar os algoritmos de otimização do compilador.

O compilador assume que a implementação em *hardware* de cada bloco básico executa em um ciclo de *clock*, fato que degrada a execução global, pois o ciclo de *clock* do *hardware* reconfigurável para uma região tem de ter período igual ao maior dos atrasos dos blocos básicos pertencentes a essa região. (CARDOSO, 2000)

Para implementar o *hardware* reconfigurável no *FPGA* são utilizadas ferramentas comerciais de síntese lógica.

A falta de informação de tempo na especificação inicial define a implementação de um ramo de um programa C em *hardware* como um problema de síntese de alto nível.

A implementação de um ramo de um programa C em *hardware* inclui três passos principais: atribuição de recursos, tradução do código intermediário GCC para *VHDL* e definição de tempo. A atribuição de recursos consiste na definição do local exato para armazenamento dos componentes (registradores). A definição de tempo é a atribuição de um período de *clock* apropriado para o sistema. A realização desses dois passos requer a análise de dependência de controle e dados, análise do tempo de vida das variáveis e a estimação do desempenho. A análise de dependência de dados e controle provê as informações da implementação do caminho de dados e controle da estrutura. A análise do tempo de vida das variáveis determina a atribuição de recursos, enquanto a estimação de desempenho decide o ciclo de *clock* para o sistema. (LOPES, 2007)

4.5 Spark

Spark é uma ferramenta acadêmica que gera arquivos *VHDL* a partir de algoritmos escritos em linguagem *ANSI-C*. (GUPTA, 2004)

4.5.1 Síntese a partir de linguagem de alto nível

HLS (High-level synthesis) é um processo na ferramenta *Spark* para se construir circuitos eletrônicos a partir de descrições comportamentais, essas descrições são freqüentemente escritas em linguagens de programação. (GUPTA, 2004)

Figura 13 – Visão geral de HLS

Fonte: (GUPTA, 2004)

Na Figura 13 pode ser visto um exemplo de HLS. Mais detalhadamente, HLS é um processo para conversão de uma descrição comportamental em um circuito digital que consiste de um *data path*, um controle e elementos de memória. (GUPTA, 2004).

O primeiro passo de HLS é capturar as representações de fluxo de controle e fluxo de dados a partir da descrição comportamental, como descrito acima e depois essas representações intermediárias são passadas para outras atividades, tais como:

Alocação (allocation): Esse passo consiste em determinar o número de recursos que tem que ser alocados para sintetizar o circuito. Os recursos consistem não apenas de unidades funcionais, tais como somadores e multiplicadores, mas podem incluir registradores e componentes de interconexão, tais como multiplexadores e barramentos;

Sincronização (Scheduling): o problema de sincronização é determinar o tempo ou ciclo de *clock*, o qual cada operação pode ser executada;

Seleção de Módulo (Module Selection): Nesse passo se determina os tipos de recursos da biblioteca de recursos que uma operação pode executar. Por exemplo: uma adição pode ser executada em um somador ou em uma *ALU (Arithmetic and Logic Unit)*. A escolha de um componente determina desempenho, gasto de energia e gasto de área no *chip*.

Binding: determina o mapeamento entre as operações, variáveis e transferências de dados e controle no projeto. Nesse passo as operações são mapeadas para especificar as

unidades funcionais, variáveis e registradores e transferência de dado ou controle para componentes interconectados;

Geração de controle e otimização: Síntese de controle gera uma unidade de controle que implementa o sincronismo (*schedule*). Essa unidade de controle gera sinais que controlam o fluxo de dados através do *data path*, por exemplo, através de multiplexadores. Otimizações no controle tentam minimizar o tamanho da unidade de controle.

Cada uma dessas tarefas é interconectada e são freqüentemente dependentes uma das outras. (GUPTA, 2004)

4.5.2 HLS paralelo

Spark permite também uma nova abordagem ao *HLS* que é o *PHLS* (*Parallelizing highlevel synthesis*). Segundo (GUPTA, 2004), essa abordagem emprega códigos agressivos de paralelização, através de compiladores paralelizáveis e técnicas de movimentação de códigos para otimizações de circuitos, mais do que no modelo tradicional *HLS*.

Essa ferramenta é particularmente interessante para aplicações multimídia e aplicações de processamento de imagens. Essas aplicações tipicamente consistem de operações aritméticas embutidas em *loops* alinhados com uma complexa mistura de construções condicionais, tais como *if-then-else*. (GUPTA, 2004) Essas construções têm um grande efeito na qualidade dos resultados de síntese de *hardware*.

4.6 Handel-C

A linguagem *C* tem 20 anos de história e domina o campo de projeto para sistemas embarcados, aplicações de *software* e sistemas operacionais. (CHAPPELL, et al., 2002) Porém, a linguagem *C* não tem construções para descrever projetos em *hardware*.

Handel-C, é baseada no padrão *ANSI-C* e é uma linguagem de alto nível para implementação de projeto de *hardware* em alto-nível, ou seja, *Handel-C* traz extensões a linguagem *C*, com a finalidade de projetar *hardware*. (CELOXICA Ltd., 2004)

Handel-C adiciona instruções de paralelismo e instruções especiais para descrição em *hardware* na sintaxe de *C*. Podem-se escrever programas sequenciais em *Handel-C*, mas para

ganhar o máximo benefício em desempenho em *hardware*, devem-se usar construções paralelas.

Na Figura 14 pode ser visto a intersecção da linguagem *ANSI-C* com o *Handel-C*, ou seja, o que é suportado pela linguagem *ANSI-C*, o que é suportado pelo *Handel-C* e o que é suportado por ambos (na intersecção). (CHAPPELL, et al., 2002)

Handel-C é um subconjunto de *Occam2*. *Handel-C* permite paralelização de operações, permitindo ao programador especificar concorrência a níveis de abstração baixos. (CARDOSO, 2000)

Figura 14 – Ambiente ANSI-C e ambiente Handel-C

Fonte: (CARDOSO, 2000)

Com o *Handel-C* os algoritmos podem ser codificados no estilo seqüencial, porém, nas partes que devem ser executadas em paralelo, utiliza a construção “*par*” para programar o paralelismo. (CHAPPELL, et al., 2002)

A instrução “*chan*” forma um canal e permite a comunicação e sincronização entre ramos paralelos do programa.

Segundo Chappell, et al. (2002), em *Handel-C* cada tarefa infere um registrador e leva um ciclo de *clock* para completar. Uma simples expressão de sincronização no *Handel-C* da ao projetista controle completo sobre a exatidão no ciclo de *clock* da implementação, permitindo exploração rápida e eficiente de diferentes arquiteturas de projetos. Uma vantagem significativa é o determinismo do resultado.

4.7 Impulse C

Há um problema fundamental quando se tenta programar um *hardware* de propósito geral (maquinas não *Von Neumann*), usando linguagem C. O problema é como expressar paralelismo. Processamento paralelo e programação de sistemas paralelos requerem suporte para concorrência na linguagem a ser utilizada e um entendimento pelo programador de como administrar múltiplos elementos computacionais quase independentes. (PELLERIN, SCOTT, 2005)

Linguagens de descrição de *hardware*, por outro lado, são projetadas para descrever sistemas altamente paralelos, embora possuam baixo nível de abstração. (PELLERIN, SCOTT, 2005)

Segundo (PELLERIN, SCOTT, 2005), para fazer sentido programar *hardware* baseado em *FPGA*, utilizando-se linguagem de alto nível, é necessário criar um modelo de máquina abstrata e escolher um modelo de programação de *software* apropriada para essa máquina abstrata.

O modelo de programação usado pelo *Impulse C* é o modelo de processo de comunicação sequencial, *CSP* (*Communicating Sequential Processes*) (PELLERIN, SCOTT, 2005)

Segundo (PELLERIN, SCOTT, 2005), *CSP* foi descrito por *Anthony Hoare* em 1978. *CSP*, segundo *Hoare* é um modelo de programação e uma linguagem para descrever padrões das interações entre processos (componentes operacionais independentes). Os processos em um sistema podem representar programas com *software* tradicional (que opera sequencialmente), mas são limitados em termos de comunicação, uma vez que só podem comunicar uns com os outros através de canais de comunicação bem definidos.

Ainda segundo (PELLERIN, SCOTT, 2005), os programas em um *CSP* podem ser executados em seu processo de *hardware*, independente uns dos outros e atuar como máquinas tradicionais *Von Neumann* atuariam. Em lugar de ter um processo principal controlando a chamada de outros processos como subrotinas, a aplicação como um todo é projetada de tal forma que os movimentos dos dados através do sistema (e através de vários processos na aplicação) são feitos via canais de dados buferizados (*buffered*). Enquanto cada processo em execução tiver recursos de acesso à memória local (para fazer o processamento e

armazenar resultados intermediários), há pouca ou nenhuma comunicação entre os processos independentes, exceto via canais de dados, os quais são chamados algumas vezes de *streams*.

O modelo *CSP*, segundo (PELLERIN, SCOTT, 2005), é conceitualmente similar ao modelo de programação *dataflow*, mas menos restritivo e mais fácil de suportar sincronização de processos (através de *buffer* de dados e passagem de mensagens), com isso é simplificada a expressão de algoritmos paralelos em alto nível através do uso bem definido de comunicação de dados, passagem de mensagens e mecanismos de sincronização.

Impulse C estende o *ANSI C*, usando funções de bibliotecas para suporte à comunicação sequencial de processos.

O *Impulse C* teve sua raiz filosófica na pesquisa realizada no laboratório *Los Alamos National Laboratories* sobre a direção do Dr. Maya Gokhale. Esta pesquisa culminou na disponibilidade pública do compilador *Streams-C* (GOKHALE, 2006) e se tornou uma metodologia para aplicações baseadas em plataformas *FPGA*. Aplicações desenvolvidas usando *Streams-C* estão no domínio da criptografia de dados, processamento de imagens, astrofísica entre outras. (PELLERIN, SCOTT, 2005)

Funções de monitoramento incluídas com as bibliotecas do *Impulse C* permitem especificar processos em uma aplicação paralela para ser utilizada na depuração. Os resultados dos processamentos são mostrados em janelas, enquanto a aplicação esta em execução sobre o controle do depurador padrão de *C*.

Para a geração de *hardware*, o *Impulse C* inclui um compilador de linguagem *C* que é baseado na parte pública disponível da ferramenta *SUIF* (*Stanford Universal Intermediate Format*) (Stanford Compiler Group, 1994), as quais são combinadas com otimizações proprietárias e ferramentas de geração de código desenvolvida pela empresa *Impulse Accelerated Technologies*. (PELLERIN, SCOTT, 2005)

O coração do modelo de programação do *Impulse C*, é composto por processos e *streams*.

Os processos são independentemente sincronizados e executam de forma concorrente. Esses processos são como subprogramas que aceitam vários dados, executam um processamento específico e geram saídas, nas *streams* de dados, nas memórias ou outro recurso de sistema. Diferentemente dos subprogramas tradicionais de *software*, esses processos não são “chamados” (repetidamente invocados durante a execução de um

programa), mas estão sempre ativos (em execução) e constantemente respondem quando dados aparecem em suas entradas. (PELLERIN, SCOTT, 2005)

As bibliotecas de *Impulse C* suportam plataformas de *FPGA*, como os da família *Virtex* da *Xilinx*, com os processadores *MicroBlaze* e *PowerPC*, bem como plataformas de *FPGAs Altera*, com os processadores *Nios* e *Nios II*. *Impulse C* também pode ser usado para gerar módulos de *hardware* que não fazem interface com processos de *software*.

Com a opção *scalarize array variables* habilitada, o compilador tentará substituir vetores locais em variáveis, de modo que elas possam ser implementadas em registradores ao invés de memórias. Quando um vetor é implementado em registradores, qualquer número de elementos podem ser escritos ou lidos simultaneamente, permitindo um alto nível de paralelismo.

A opção *loop unrolling* move certas expressões para fora do escopo do *loop*. A opção *unrolling* pode ser habilitada pela instrução *#pragma CO UNROLL*. Um exemplo da opção *loop unrolling* pode ser visto abaixo:

```
for (i=0; i < 10; i++) {
    #pragma CO UNROLL
    sum += A[i];
}
```

Com a opção *loop unrolling* o compilador elimina o bloco do loop e duplica o corpo dele para todos os valores de “i”. No exemplo acima, com a opção *loop unrolling* habilitada, o resultado é mostrado abaixo:

```
sum += A[0];
sum += A[1];
sum += A[2];
sum += A[3];
sum += A[4];
sum += A[5];
sum += A[6];
sum += A[7];
sum += A[8];
sum += A[9];
```

Como visto no exemplo acima, 10 somadores foram gerados e cada um é usado apenas uma vez durante a execução do *loop*. Segundo (PELLERIN, SCOTT, 2005), na maioria das vezes, como no exemplo acima, a opção *loop unrolling* sozinha não traz melhorias significativas, porque apenas um valor do vetor “A” pode ser lido em um dado ciclo, ou seja, nesse exemplo, mesmo com a opção *loop unrolling*, o exemplo acima requer 10 ciclos de *clock* para executar e 10 somadores tem que ser gerados. Porém, se a opção *scalarize array variables* for usada junto com a opção *loop unrolling*, os elementos do vetor são substituídos por variáveis escalares. Então o resultado é equivalente a:

```
sum += A_0;
sum += A_1;
sum += A_2;
sum += A_3;
sum += A_4;
sum += A_5;
sum += A_6;
sum += A_7;
sum += A_8;
sum += A_9;
```

Ao invés de ser gerada uma memória para o vetor “A”, registradores são gerados para cada um dos 10 elementos do vetor. Então, os 10 registradores podem ser lidos simultaneamente e assim esse *loop* inteiro pode ser executado com apenas 1 ciclo de *clock*. Segundo (PELLERIN, SCOTT, 2005), o uso da opção *unrolling* requer alguns cuidados, porque uma grande quantidade de lógica pode ser gerada e o *cycle delay* pode aumentar, diminuindo assim a máxima frequência do *clock* (*maximum clock frequency*).

No desenvolvimento do projeto desta tese, os compiladores em linguagem C para plataformas *FPGA* pesquisados, principalmente as metodologias utilizadas por cada um, são de grande apoio, complementando todos os estudos das áreas abordadas por esta tese e principalmente abrindo possibilidades de desenvolvimento da proposta deste estudo utilizando os conceitos e métodos na aplicação do Modelo a Fluxo de Dados para o processo de conversão até a sua execução em hardware para o desenvolvimento da ferramenta proposta neste trabalho que converte C em grafo a fluxo de dados e que serão os posteriores capítulos a serem tratados.

CAPÍTULO 5 – MODELO A FLUXO DE DADOS DINÂMICO PARA HARDWARE RECONFIGURÁVEL UTILIZADO NESTE PROJETO

Durante os anos 70 até meados dos anos 90, existiram algumas implementações envolvendo arquiteturas a fluxo de dados (ARVIND, 2005; DENNIS, 1975; VEEN, 1986).

A diferença fundamental entre as arquiteturas tradicionais Von Neumann e as arquiteturas a fluxo de dados está exatamente no controle do fluxo das informações que no caso das máquinas Von Neumann é baseado no controle seqüencial, enquanto que nas máquinas a fluxo de dados é executado pela presença dos dados .

Um grafo a fluxo de dados é um conjunto de operadores interligados por arcos. A presença de dados em cada arco de um operador é que irá disparar a execução desse operador.

Existem dois modelos de arquiteturas a fluxo de dados: estático e dinâmico. No modelo estático, apenas um dado pode estar presente em um arco esperando pelos seus parceiros. Um protocolo deve garantir o sincronismo entre os operadores para garantir que apenas um dado esteja presente em cada arco. Conseqüentemente o paralelismo fica limitado ao conjunto de operadores cujos arcos possuem apenas um dado.

No modelo dinâmico, mais que um dado pode estar presente no arco. Um protocolo deve assegurar que os dados presentes nos arcos disparem as operações cujos dados parceiros também estejam presentes.

Uma metodologia como "*Tagged-token*" pode ser utilizada no protocolo para controlar os dados parceiros em cada arco. Neste caso o paralelismo acontece quando da execução dos vários operadores cujos dados estejam todos disponíveis.

Como vimos, algumas arquiteturas reconfiguráveis foram propostas durante os últimos 20 anos. Entretanto a motivação para este projeto foi a grande experiência adquirida no passado no desenvolvimento de máquinas a fluxo de dados, a experiência também adquirida nos últimos 20 anos em computação reconfigurável, e mais recentemente o aparecimento de grupos desenvolvendo sistemas a fluxo de dados em computação reconfigurável e compiladores para arquiteturas reconfiguráveis.

Podemos destacar o projeto WaveScalar, proposto por Swanson (SWANSON,2003), (SWANSON,2006); a Unidade de Controle Dataflow para Compilação Pipeline Configurável a partir de programação C, por CAPPELLI (CAPPELLI,2004); e os projetos voltados para compiladores que fazem uso do conceito de fluxo de dados mas que não

implementam arquiteturas a fluxo de dados, é o caso dos projetos: SPARK proposto por Gupta (Gupta, et al., 2004) no "Center for Embedded Computer Systems"; o *Stream-C* (GOKHALE, 2006) desenvolvido pelo "Alamos National Laboratory"; Pico Express comercializado pela empresa Synfora (SYNFORA,2007); o compilador de *Chichkov* (CHICHKOV, et al., 1997); *DK Design Suit da* (CELOXICA Ltd, 2002); *Handel-C* (CHAPPELL, et al., 2002); *ImpulseC* (PELLERIN, et al., 2005), etc.

5.1 Estrutura do Modelo a Fluxo de Dados

O Modelo de programação a fluxo de dados proposto neste trabalho, segue a linha proposta por Veen (VEEN,1986) que está baseada em uma linguagem gráfica contendo operadores e arcos interligando esses operadores. Na Figura 15 é descrito um exemplo de um grafo a fluxo de dados representando a execução de uma equação.

$$T = ((x + y) * z) + (k * p)$$

Figura 15 - Um exemplo de Grafo a Fluxo de Dados

Fonte: (CORREIA, 2007)

Neste caso existem três estágios de paralelismo. O primeiro corresponde às operações $(x + y)$, e $(k * p)$. O segundo corresponderia ao resultado $(x + y)$ multiplicado por z , e finalmente, no terceiro nível $((x + y) * z)$ somado ao resultado de $(k * p)$ gerando T .

Como citado anteriormente, existem dois modelos de arquiteturas a fluxo de dados: o modelo estático e o dinâmico. No modelo estático, somente um dado pode estar presente em cada arco à cada vez. No modelo dinâmico mais que um dado pode estar presente em um arco.

Como o modelo escolhido para este trabalho foi o modelo dinâmico, que permite a existência de mais que um dado em cada arco, a implementação do modelo é tal que novas instâncias dos operadores são geradas para cada dado que chega em um arco. Assim, várias instâncias de um operador podem ser geradas, esperando por um dado parceiro. O modelo de instâncias é discutido a seguir.

5.1.1 O Modelo de Instâncias

Para implementar um grafo a fluxo de dados que é executado de forma dinâmica, foi criado um processo para inserir e remover sub-grafos. Na Figura 16 é descrito um grafo T e um sub-grafo F que será inserido no grafo original T.

O modelo para inserir e retirar sub-grafos foi baseado em uma estrutura de concatenação. Na Figura 16 parte *a*, o grafo original T tem dois pontos de concatenação entre dois operadores onde o sub-grafo F parte *b* será concatenado.

Figura 16 - Sub-grafo F a ser incluído no grafo original T

Fonte: (CORREIA, 2007)

Figura 17 - Resultado da concatenação do sub-grafo F ao grafo T

Fonte: (CORREIA, 2007)

Em termos de grafo, precisamos apenas ligar as linhas existentes entre os pontos do grafo T e os arcos do sub-grafo F, mas em termos do modelo, é necessário implementar um protocolo de comunicação que informa "se" e de "onde" um dado estará vindo e "se" e para "onde" um dado estará indo. O resultado da concatenação é descrito na Figura 17.

Figura 18 - Sub-grafo a ser removido do grafo original

Fonte: (CORREIA, 2007)

Para remover sub-grafos do grafo T conforme mostra a Figura 18, define-se pontos onde a concatenação deverá ocorrer e o mesmo processo acima deverá ser executado mas agora concatenando os pontos onde o sub-grafo foi removido conforme a Figura 19.

Figura 19 - Grafo Resultante da remoção de sub-grafo

Fonte: (CORREIA, 2007)

5.1.2. A estrutura tagged-token

Como vimos "tagged-tokens" estão associados ao modelo a fluxo de dados dinâmico onde diferentes dados podem estar presentes em um arco. Esse modelo será implementado para que os dados que possuam o mesmo "tagged-token" façam parte das mesmas instâncias de um operador e portanto fiquem prontos para serem executados. Um exemplo de aplicação que caracteriza o uso do "tagged-token" é o loop expresso pela seguinte instrução "for":

```
z=0;
for (i=0; i<N; i++)
  z=z+(x*y);
```

O operador "*" utilizado na expressão $z = z + (x * y)$ fará parte de um grafo a fluxo de dados podendo receber N dados diferentes e em tempos diferentes para arcos representando "x" e "y". Cada vez que existir um novo dado "x", e seu correspondente "y" ainda não estiver presente, uma nova instância do operador "*" deverá ser utilizada esperando a chegada do "y" correspondente. O mesmo acontece no caso do dado "y" estar presente antes do dado "x". Assim, diferentes instâncias de um operador deverão ser utilizadas para cada dado que aparecer em um arco, esperando pelo seu par correspondente.

A utilização de uma nova instância no grafo a fluxo de dados se assemelha ao modelo de concatenação. Um sub-grafo de um operador é concatenado tendo como pontos de concatenação os mesmos pontos de entrada e saída (arcos de dados) do operador. No caso do comando "for" descrito acima, nova instância do operador "*" e pontos de concatenação para arcos "x" e "y" serão utilizados.

Tão logo uma instância de um operador seja executado, um dado resultante é gerado e enviado para ser utilizado em outras instância.

Na Figura 20 estão descritas várias instâncias de um operador "*" em um grafo a fluxo de dados dinâmico.

Figura 20 - Instâncias diferentes para o operador "*"

Fonte: (CORREIA, 2007)

5.1.3 Operadores utilizados no Modelo

Os operadores utilizados no modelo são: "Decisão", "Merge Não Determinístico", "Merge Determinístico", "Ramificação", "Cópia" e "Operador", estes operadores estão descritos na Figura 21 (SILVA,1989)(SILVA,1992). Utilizando os operadores descritos podemos construir grafos como os apresentado na Figura 22 que expressam alguns comando em C++.

Figura 21 - Operadores do Modelo DDFG

Fonte: (CORREIA, 2007)

Figura 22 - Grafo representando comandos em C++

Fonte: (CORREIA, 2007)

5.1.4 Construções Iterativas no Modelo

As construções iterativas são estruturas que necessitam um sincronismo toda vez que um novo dado circula por elas. Para isso foram definidos *tags* que serão utilizados para sincronizar os dados no grafo. O formato dos *tags* proposto neste projeto está descrito na Figura 23.

Como as operações iterativas podem ser aninhadas, é necessário que o *tag* contenha também o nível de aninhamento (campo *nesting* na Figura 23).

Finalmente, ao final de cada função, procedure or programa, o *tag* será modificado informando que aquela ativação terminou.

Os operadores específicos para o controle dos construtores iterativos são: "new tag manager (NTM)"; "new iteration generation (NIG)"; e "new tag destructors (NTD)". O operador NTM é responsável pela alocação de um novo *tag*. O operador NIG modifica o *tag* gerando um novo valor para o campo *iteration* e *nesting*.

O operador NTD modifica os *tags* no campo ativação, quando esse deixa uma função, procedure ou programa.

O exemplo de um programa utilizando construções iterativas é descrito na Figura 24. A Figura demonstra o grafo de um "FOR" contendo como códigos *for* ($i=0; i < N; i++$)/ $x = x + i * x$). Para cada execução de um programa, função e procedimento, é gerado um *tag*, representando uma ativação (campo *activation* na Figura 23). No caso de existirem *loops* como é o caso da Figura 23 que é possível identificar estruturas de repetição pelos grafos de

Merge Determinístico seguidos do grafo *Cópia, Decisão e Operador*, todos conectados para executar a sintaxe “FOR”, implementados com operações iterativas (“WHILE, REPEAT e FOR”), a entrada em cada uma dessas operações irá alterar o *tag* (campo *iteration* na Figura 23), especificando nova iteração.

4 bits	4 bits	8 bits	32 bits
Activation	Nesting	Iteration	DATA

Figura 23 - Formato dos dados contendo Tags

Fonte: (CORREIA, 2007)

Figura 24 - Exemplo de um programa com construtores iterativos

Fonte: (CORREIA, 2007)

5.2 Estrutura "Matching" de Dados

Considerando um grafo a fluxo de dados, seus operadores e arcos, acrescido de operadores específicos para construções iterativas, seria necessário desenvolver uma estrutura para identificar os dados parceiros, a partir dos tags de cada dado, a serem manipulados em uma nova instância de um operador. Esta estrutura é chamada de "Matching" de dados.

Uma vez definida essa estrutura, ainda seria necessário a sincronização dos dados presentes em cada arco.

Um protocolo para enviar e receber dados nos arcos foi implementado. Esse protocolo, além de informar para cada operador que um dado está presente em algum arco de entrada naquele operador, também é utilizado para disparar o processo de "*Matching*" (SILVA,2006a; SILVA,2006b; ASTOLFI,2007).

O protocolo implementado entre os operadores é simples a partir de procedimentos "send" e "receive", um sinal específico de "strobe" informando ao operador destino que chegou um dado, finalizando com sinais de reconhecimento "*ack*".

Uma vez implementado o protocolo, o "*Matching*" será acionado a partir da chegada de algum dado de entrada em um operador, detectado pelo sinal de "strobe" de envio de dados.

A partir do "tag" de cada dado, um circuito foi implementado para identificar se existe um dado parceiro em alguma instância do operador, ou se será necessário utilizar uma nova instância daquele operador.

Para que essa busca fosse rápida, o circuito de "*Matching*" foi inserido em cada instância de um operador e a busca é feita simultaneamente em cada uma delas. Se uma instância possui um dado com o mesmo "tag", esse novo dado será inserido naquela instância e se a mesma já possuir todos os dados para execução, ela então é executada e seus dados são retirados daquela instância. Caso contrário, se não houver nenhuma instância com aquele dado, uma nova instância será ocupado com esse dado e ficará aguardando a chegada de dados parceiros.

Quando um dado é encontrado em alguma das instâncias, uma variável é "setada" indicando que existe uma instância com dado parceiro e portanto não será necessária utilizar uma nova instância para aquele dado. Caso contrário, se não ocorrer o "matching" daquele dado, essa mesma variável será "resetada", informando que uma nova instância deve ser utilizada. O circuito "*Matching*" é descrito na Figura 25.

Figura 25 - Instância com circuito de "Matching" e variável comum

Fonte: (CORREIA, 2007)

Na Figura 25, M representa o circuito de "*Matching*" para cada instância, e a variável BitM é a variável que informa se ocorreu ou não o "*matching*" dos dados parceiros.

CAPÍTULO 6 - FERRAMENTA DE CONVERSÃO C EM GRAFO A FLUXO DE DADOS

A ferramenta proposta neste estudo, referente à análise Léxica e para posterior geração do código VHDL, foi desenvolvida em linguagem C++. A linguagem Delphi está sendo utilizada para a parte visual da ferramenta e também para identificar os operadores DFG para elaboração e estruturação do código em mapa binário que traduz partes de códigos considerados mais intensos, do ponto de vista de processamento, para o modelo a fluxo de dados. Na Figura 26 abaixo é descrito como a ferramenta irá gerar o código no modelo a fluxo de dados proposto pela tese.

Figura 26 - Visão da ferramenta utilizando técnicas de Análise Léxica para posterior conversão em VHDL

Fonte: (AUTOR, 2009)

Na Figura 26 a partir do código considerado mais intenso selecionado, é feita a análise léxica. O processo de varredura do código também chamado de *scanning*, é o responsável em identificar letras, números e palavras reservadas do código em análise e, a partir deste processo, cada palavra reservada, letras e números encontrados são chamados de tokens.

Com base no arquivo de tokens gerado através de uma ferramenta desenvolvida em linguagem Delphi que possui extensão em *txt*, é realizado o processo de relacionamento dos tokens e da sintaxe do código com os operadores do *DFG*. De posse do arquivo com a cadeia

de bits, a ferramenta então pode gerar os respectivos códigos VHDL, segundo um conjunto de operadores implementados em VHDL.

Para a implementação da ferramenta descrita na Figura 26 uma parte fundamental do processo é criar um analisador léxico, que é a primeira fase denominada de núcleo principal responsável pelo perfeito desenvolvimento e execução de um compilador.

Concluída essa parte, o passo seguinte é criar rotinas em C++ para que compreenda as cadeias de bits geradas e converta-as em códigos VHDL.

A seguir é detalhado cada um dos processos desenvolvidos e processos a serem desenvolvidos.

6.1 O Analisador Léxico gerado na Ferramenta

Como foi visto, uma parte fundamental do processo foi criar um analisador léxico. Esta fase é responsável em analisar as entradas de linhas de caracteres para buscar e averiguar os padrões pertencentes a uma linguagem. Neste caso o analisador léxico deverá informar se as expressões regulares, ou seja, o alfabeto analisado é referente à linguagem C, linguagem padrão para análise nesta tese. À medida que essa análise vai sendo realizada, uma sequência de símbolos vão sendo gerados, normalmente denominados tokens, que serão utilizados na geração das dependências dos grafos através de uma ferramenta visual e como consequência gerando o mapa binário e o código VHDL.

Essa parte inicial do projeto já se encontra implementado, sendo como proposta final da tese o desenvolvimento do código que converterá o mapa binário gerado pela ferramenta em códigos VHDL.

6.1.1 Implementação do Analisador Léxico em código C++

A seguir é descrito um exemplo de código fonte a ser analisado pelo analisador léxico também denominado de análise linear, especificamente a fase inicial do processo. Trata-se de um comando “for” contendo as variáveis “i”, “n” e “z”. O analisador linear foi desenvolvido em linguagem C++.

```
z=2;
for (i=0, i < n, i++) {
```

```

z += i;
}

```

Com base no escopo “FOR” descrito, traduzindo em grafos via estrutura DFG o resultado ficará de acordo com a Figura 27.

Figura 27 – Grafo extraído do escopo “for”

Fonte: (CORREIA, 2007)

Para esse projeto, consideram-se programas escritos corretamente. Dentro do processo de análise linear, também são gerados os tokens para cada elemento (letras, números, palavras reservadas) do comando, tokens esses que serão utilizados na análise do grafo de dependência. A Figura 28 mostra os resultados iniciais do comando “for” e todos os tokens relacionados. A análise foi gerada através de linha de comando (*projeto [nome_do_arquivo].txt > [nome_do_arquivo].bin.txt*) em modo DOS definido na programação.

Figura 28 - Identificação dos tokens pela análise léxica

Fonte: (AUTOR, 2009)

6.2 A Ferramenta para conversão

A ferramenta gerada como parte da proposta feita pela tese conforme demonstra a Figura 29 foi implementada em linguagem Delphi apresentando, portanto uma interface amigável para o processo a ser descrito.

Figura 29 - Tela inicial da ferramenta de conversão

Fonte: (AUTOR, 2009)

A Figura 29 exhibe a tela padrão do software. O sistema possui campos importantes para demonstração do processo de conversão e dependência dos tokens com os operadores *DFG*.

O programa desenvolvido e denominado “Analizador *ChipCflow*”, possui uma interface simples para interação e que praticamente apresenta somente uma etapa a ser realizada pelo usuário, esta etapa é a responsável pela execução de todo o processo das fases até a geração do mapa binário.

No canto superior esquerdo localiza-se um item titulado de “Código Fonte”, este item, ao ser acionado, abre uma janela para a seleção de um arquivo com extensão “txt” contendo um código fonte exemplo da linguagem C; vale ressaltar que a janela ao ser aberta direciona automaticamente o caminho da pasta onde estão localizados os arquivos em linguagem C conforme Figura 30.

O campo “Analizador Léxico” exhibe o processo já realizado pela primeira parte desta ferramenta. A primeira parte da ferramenta é o Analizador Léxico desenvolvido em C++ e

que não possui interface, responsável em identificar os tokens do escopo em linguagem C contidos no arquivo em “txt”.

O campo “Programa Fonte” é somente informativo, pois exibe o código fonte que foi anteriormente selecionado e executado pela análise Léxica. O código fonte escolhido para execução pela ferramenta de Análise Léxica e posteriormente pela ferramenta Delphi deverá sempre se encontrar em um arquivo com extensão “.c” como é exibido na Figura 30.

Figura 30 – Janela de busca dos tokens gerados em arquivo com extensão “.c”

Fonte: (AUTOR, 2009)

Na Figura 31 é mostrada a execução da iteração baseada totalmente nos operandos do DFG que é a primeira fase mais importante dentro da ferramenta desenvolvida em Delphi e que é indicada pelo campo “Analizador DFG”, pois este procedimento executa o processo de relacionamento e dependência dos tokens gerados com os operadores do *DFG*. O processo de análise de relação entre tokens e DFG foi elaborado via código em linguagem Delphi. A ferramenta trata qualquer tipo de iteração através do tratamento da equivalência entre palavras reservadas da linguagem C (*IF*, *FOR*, *WHILE*, etc) com os operandos do *DFG* apresentados pelo capítulo 5.

Analizador ChipCflow - v. Experimental

Código Fonte

Analizador Léxico::

```

z
=
2
;
for
(
i
=
0
;
i
<
n
;
i
++
)
{
z
+=
i
;
}

```

Analizador DFG::

Operador DFG	Input(1)	Input(2)	Input(3)	Output(1)	Output(2)
(101)Merge Deterministic	Z	2	S3	S1	
(101)Merge Deterministic	i	0	S3	S2	
(010)Copy	S2			S2	S2
(011)Decider	n	S2		S3	
(010)Copy	S3			S3	S3
(010)Copy	S3			S3	S3
(010)Copy	S3			S3	S3
(001)Branch	S2	S3		i	S4
(010)Copy	S4			S4	S4
(110)Operator	S4	1		i	
(001)Branch	S1	S3		z	S5
(110)Operator	S5	S4		z	

Programa Fonte::

```

z=2;
for(i=0,i<n,i++){
z+=i;
}

```

[Tabela::Mapa Binário](#) [TXT::Mapa Binário](#) [TXT::VHDL](#)

Figura 31 – Primeiro resultado da execução para identificação dos operandos DFG “FOR” e dependências
Fonte: (AUTOR, 2009)

Após o processo de análise de relação relatada pelo parágrafo anterior, a segunda fase mais importante da ferramenta desenvolvida em Delphi é indicada pelo link com nome de “TXT.Mapa Binário”. Este link após seu acionamento gera um arquivo com extensão *txt* exibindo os tokens gerados pela primeira ferramenta, e em sequência exibe também o código fonte analisado convertido em mapa binário baseado na análise da relação dos tokens com os operandos *DFG* como mostra a Figura 32.


```

analise_DDFG_for.txt - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda

=====
|      EESC-USP Projeto ChipCFlow      |
=====

Token: z
Token: =
Token: 2
Token: :
Token: for
Token: (
Token: i
Token: =
Token: 0
Token: :
Token: i
Token: <
Token: n
Token: :
Token: i
Token: ++
Token: )
Token: {
Token: z
Token: +=
Token: i
Token: :
Token: }

Operador,Input(1),Input(2),Input(3),Output(1),Output(2)
00000101;00001010;00010010;00001110;00010000;
00000101;00001011;00010011;00001110;00001100;
00000010;00001100;          ;          ;00001100;00001100
00000011;00001101;00001100;          ;00001110;
00000010;00001110;          ;          ;00001110;00001110
00000010;00001110;          ;          ;00001110;00001110
00000010;00001110;          ;          ;00001110;00001110
00000001;00001100;00001110;          ;00001011;00001111
00000010;00001111;          ;          ;00001111;00001111
00000110;00001111;00010100;          ;00001011;
00000001;00010000;00001110;          ;00010101;00010001
00000110;00010001;00001111;          ;00010101;

```

Figura 32 – Segundo resultado da execução para identificação do mapa binário “FOR”

Fonte: (AUTOR, 2009)

O processo de conversão para mapa binário é partido da relação de dependência dos tokens gerados com os operadores do *DFG* identificando cada palavra reservada relacionada a iterações (*IF*, *FOR*, *WHILE*, etc.), presente no código em análise e identificando os bits representativos pré-definidos armazenados em uma tabela comparativa via programação de acordo com a palavra reservada identificada em tempo de execução.

Com a identificação da palavra reservada para relacionar ao seu devido operador *DFG*, é obtida, portanto a cadeia binária representando o operador *DFG* (3bits), a cadeia binária das entradas do operador *DFG* (8bits) denominadas de *Input* pela ferramenta e a cadeia binária das saídas do operador *DFG* (8bits) denominadas de *Output* pela ferramenta.

A Figura 33 também demonstra a definição dos *bits* para cada operando *DFG* e que se encontram pré-definidos via código de programação.

Figura 33 – Definição de bits e operandos DFG para a ferramenta

Fonte: (AUTOR, 2009)

A seguir será descrito um exemplo de todo o processo até sua geração de bits. O código a ser analisado para este processo será o escopo “*IF*”:

```
if (x > 0)
{
  z = a + b;
}
else
{
  z = c - d;
}
```

Com base no escopo “*IF*” descrito, traduzindo em grafos via estrutura DFG o resultado ficará de acordo com a Figura 34:

Figura 34 - Grafo extraído do escopo “if”

Fonte: (CORREIA, 2007)

A Figura 35 exibe a execução da ferramenta visual para os processos de equivalência de DFG e geração do mapa binário e suas interligações com os demais operadores.

Analisador DFG:

Operador DFG	Input(1)	Input(2)	Input(3)	Output(1)	Output(2)
(011)Decider	x	0		S1	
(110)Operator	A	B		S2	
(110)Operator	C	D		S3	
(101)Merge Deterministic	S2	S3	S1	S4	

Programa Fonte:

```

if ( x > 0 )
{
  z = A + B;
}
else
{
  z = C - D;
}

```

Links: [Tabela::Mapa Binário](#) [TXT::Mapa Binário](#)

Figura 35 - Primeiro resultado da execução para identificação dos operandos DFG “IF” e dependências

Fonte: (AUTOR, 2009)

Conforme exibido na Figura 35 com o código fonte VHDL gerado, a ferramenta automaticamente identifica o outro arquivo que contém os tokens que foram previamente gerados pelo Analisador Léxico e principalmente a execução da relação de dependência dos tokens com os operadores *DFG* e, portanto mostra seis colunas como resultado da análise de relação. A primeira coluna identifica os tipos de operadores *DFG* presentes no código e a

cadeia de bits para cada operador e também para melhor compreensão, exibe o nome de cada operador, as demais colunas denominadas de *Input* e *Output* são responsáveis por realizar todo o processo restante do código, interligando cada *Output* de um operador ao devido *Input* do outro operador. As colunas de *Input* e *Output* são fixas para melhor visualização do processo, pois um operador *DFG* possuirá no máximo 3 entradas e 2 saídas.

Após o processo de dependência de operadores, é necessário gerar o mapa binário através do link “TXT.Mapa Binário”, através deste procedimento é exibido na tela o resultado da conversão mostrada pela grid dos operadores com as 6 colunas em mapa binário, o mapa binário gerado é realizado através da análise comparativa de bits representativos definidos e armazenados em código de programação da ferramenta visual. (Figura 36)

```

analise_DDFG - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda
=====
| EESC-USP Projeto ChipCFlow |
=====
Token: if
Token: (
Token: x
Token: >
Token: 0
Token: )
Token: {
Token: z
Token: =
Token: A
Token: +
Token: B
Token: ;
Token: }
Token: else
Token: {
Token: z
Token: =
Token: C
Token: -
Token: 0
Token: ;
Token: }

operador, Input(1), Input(2), Input(3), output(1), output(2)
00000011; 00001010; 00001110;          ; 00010010;
00000110; 00001011; 00001111;          ; 00001101;
00000110; 00001100; 00010000;          ; 00010001;
00000101; 00001101; 00010001; 00010010; 00010011;
  
```

Figura 36 - Segundo resultado da execução para identificação do mapa binário “IF”

Fonte: (AUTOR, 2009)

Para o terceiro exemplo de conversão de DFG para mapa binário, o código a ser analisado para este processo será o escopo “*WHILE*”:

```

x = 3;
while (x>1)
{
x = x+1;
  
```

```

y=y+x;
z=x+y;
}

```

Com base no escopo “*WHILE*” descrito, traduzindo em grafos via estrutura DFG o resultado ficará de acordo com a Figura 37:

Figura 37- Grafo extraído do escopo “While”

Fonte: (AUTOR, 2009)

Com base no DFG da Figura 37, a ferramenta gera suas dependências como primeira fase do processo: (Figura 38)

Analisador ChipCFlow - v. Experimental

Código Fonte

```

x
=
3
while
(
x
>
1
)
{
x
=
x
+
1
;
y
=
y
+
x
;
z
=
x
+
y
;
}

```

Programa Fonte:

```

x=3;
while(x>1)
{
x=x+1;
y=y+x;
z=x+y;
}

```

Analisador DFG:

Operador DFG	Input(1)	Input(2)	Input(3)	Output(1)	Output(2)
(101)Merge Deterministic	3	x		S1	
(010)Copy	S1			S1	S1
(011)Decider	S1	1		S2	
(010)Copy	S2			S2	S2
(101)Merge Deterministic	S1		S2	S3	
(110)Operator	S3	1		S4	
(010)Copy	S4			S4	S4
(110)Operator	S4	y		S5	
(110)Operator	S4	S5		z	

Tabela: Mapa Binário TXT: Mapa Binário TXT: VHDL

Figura 38 - Primeiro resultado da execução para identificação dos operandos DFG “While” e dependências

Fonte: (AUTOR, 2009)

Após o processo de identificação dos operandos DFG e dependências localizadas no escopo “*While*”, é, portanto gerado o relatório final através do item “TXT.Mapa Binário” contendo os tokens gerados e principalmente o mapa binário do DFG. (Figura 39)

```

=====
|      EESC-USP Projeto ChipCFlow      |
=====
Token: x
Token: =
Token: 3
Token: ;
Token: while
Token: (
Token: x
Token: >
Token: 1
Token: )
Token: {
Token: x
Token: =
Token: x
Token: +
Token: 1
Token: ;
Token: y
Token: =
Token: y
Token: +
Token: x
Token: ;
Token: z
Token: =
Token: x
Token: +
Token: y
Token: ;
Token: }

Operador, Input(1), Input(2), Input(3), Output(1), Output(2)
00000101; 00001010; 00001111; ; 00001011;
00000010; 00001011; ; ; 00001011; 00001011
00000011; 00001011; 00010000; ; 00001100;
00000010; 00001100; ; ; 00001100; 00001100
00000101; 00001011; ; 00001100; 00001101;
00000110; 00001101; 00010000; ; 00001110;
00000010; 00001110; ; ; 00001110; 00001110
00000110; 00001110; 00010001; ; 00010010;
00000110; 00001110; 00010010; ; 00010011;
  
```

Figura 39 - Segundo resultado da execução para identificação do mapa binário “While”

Fonte: (AUTOR, 2009)

Para o quarto exemplo de conversão de DFG para mapa binário, o código a ser analisado para este processo será um pouco mais detalhado, apresentando um código mais complexo e a interação entre estruturas de repetição IF, FOR e WHILE.

a = 3;

b = 0;

y = 1;

z=2;

```
for (i=0, i < n, i++) {  
  
    if (z > 1)  
    {  
        z = a+b;  
    }  
    else {  
        x = 3;  
        while (x > 1)  
        {  
            for (j=0, j < n, j++) {  
                x = x+1;  
                y=y+x;  
                z+=i;  
            }  
        }  
    }  
}
```

Com base no escopo descrito, traduzindo em grafos via estrutura DFG o resultado ficará de acordo com a Figura 40:

Figura 40 - Grafo extraído do escopo (IF, FOR, WHILE)

Fonte: (AUTOR, 2009)

Com base no DFG da Figura 40, a ferramenta gera suas dependências como primeira fase do processo: (Figura 41 e Figura 42)

Analizador ChipCflow - v.Experimental

Código Fonte

Analizador Léxico:

```

a = 3;
b = 0;
y = 1;
z = 2;
for (
  i = 0;
  i < n;
  i ++
)

```

Programa Fonte:

```

a = 3;
b = 0;
y = 1;
z = 2;
for (i=0, i<n, i++) {

```

Analizador DFG:

Operador DFG	Input(1)	Input(2)	Input(3)	Output(1)	Output(2)
(101)Merge Deterministic	0	i		S1	
(010)Copy	S1			S1	S1
(011)Decider	n	S1		S2	
(010)Copy	S2			S2	S2
(101)Merge Deterministic	S1	S2		S3	
(110)Operator	S3	1		S4	
(101)Merge Deterministic	z	2	S2	S5	
(011)Decider	S5	1		S6	
(010)Copy	S6			S6	S6
(110)Operator	a	b		S7	
(101)Merge Deterministic	S7	z	S6	S8	
(101)Merge Deterministic	3	x	S6	S9	
(010)Copy	S9			S9	S9
(011)Decider	S9	1		S10	
(101)Merge Deterministic	0	i	S10	S11	
(010)Copy	S11			S11	S11
(011)Decider	S11	n		S12	
(010)Copy	S12			S12	S12
(010)Copy	S12			S12	S12

Tabela: Mapa Binário TXT: Mapa Binário

Figura 41 - Execução para identificação dos operandos DFG (IF, FOR, WHILE) parte1

Fonte: (AUTOR, 2009)

Analizador ChipCflow - v.Experimental

Código Fonte

Analizador Léxico:

```
while
(
x
>
1
)
{
for
(
j
0
j
<
n
:
j
++
)
{
x
=
x
+
1
:
y
}
}
a = 3;
b = 0;
y = 1;
z = 2;
for (i=0, i<n, i++) {
```

Analizador DFG:

Operador DFG	Input(1)	Input(2)	Input(3)	Output(1)	Output(2)
(010)Copy	S12			S12	S12
(010)Copy	S12			S12	S12
(010)Copy	S12			S12	S12
(010)Copy	S12			S12	S12
(101)Merge Deterministic	S11		S12	S13	
(110)Operator	S13	1		S14	
(010)Copy	S14			S14	S14
(101)Merge Deterministic	S10	x	S12	S15	
(001)Branch	S15		S12	S16	S16
(110)Operator	S16	1		S17	
(010)Copy	S17			S17	S17
(101)Merge Deterministic	1	y	S12	S18	
(001)Branch	S18		S12	S19	S19
(110)Operator	S19	S17		S20	
(101)Merge Deterministic	2	z	S12	S21	
(001)Branch	S21		S12	S22	S22
(110)Operator	S22	S14		S23	

Programa Fonte:

```
a = 3;
b = 0;
y = 1;
z = 2;
for (i=0, i<n, i++) {
```

Tabela: Mapa Binário TXT: Mapa Binário

Figura 42 - Execução para identificação dos operandos DFG (IF, FOR, WHILE) parte2

Fonte: (AUTOR, 2009)

Após o processo de identificação dos operandos DFG e dependências localizadas no escopo, é, portanto gerado o relatório final através do item “TXT.Mapa Binário” contendo os tokens gerados e principalmente o mapa binário do DFG conforme demonstrado pelas Figuras 43, 44 e Figura 45).

analise_DDFG.txt - Bloco de notas

Arquivo Editar Formatar Exibir Ajuda

EESC-USP Projeto ChipCflow

```
Token: a
Token: =
Token: 3
Token: ;
Token: b
Token: =
Token: 0
Token: y
Token: =
Token: 1
Token: ;
Token: z
Token: =
Token: 2
Token: ;
Token: for
Token: {
Token: =
Token: 0
Token: ;
Token: 1
Token: <
Token: n
Token: ;
Token: 1
Token: ++
Token: }
Token: {
Token: if
Token: {
Token: z
Token: >
Token: 1
Token: }
Token: {
Token: z
Token: =
Token: a
Token: +
Token: b
Token: ;
Token: else
```

Figura 43 - Identificação mapa binário parte1

Fonte: (AUTOR, 2009)

```

analise_DDFG.txt - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda

Token: {
Token: x
Token: =
Token: 3
Token: ;
Token: while
Token: (
Token: x
Token: >
Token: 1
Token: )
Token: {
Token: for
Token: (
Token: j
Token: =
Token: 0
Token: ,
Token: j
Token: <
Token: n
Token: ,
Token: j
Token: ++
Token: }
Token: {
Token: x
Token: =
Token: x
Token: +
Token: 1
Token: ;
Token: y
Token: =
Token: y
Token: +
Token: x
Token: ;
Token: z
Token: +=
Token: i
Token: ;
Token: }
Token: }
Token: }
Token: }

```

Figura 44 - Identificação mapa binário parte 2

Fonte: (AUTOR, 2009)

```

analise_DDFG.txt - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda

Operador,Input(1),Input(2),Input(3),Output(1),Output(2)
00000101;00001010;00100100;      ;00001011;
00000010;00001011;      ;      ;00001011;00001011
00000011;00001100;00001011;      ;00001101;
00000010;00001101;      ;      ;00001101;00001101
00000101;00001011;00001101;      ;00001110;
00000110;00001110;00011110;      ;00101001;
00000101;00001111;00100001;00001101;00010000;
00000011;00010000;00011110;      ;00010001;
00000010;00010001;      ;      ;00010001;00010001
00000110;00010010;00100101;      ;00010011;
00000101;00010011;00001111;00010001;00101010;
00000101;00010100;00100110;00010001;00010101;
00000010;00010101;      ;      ;00010101;00010101
00000011;00010101;00011110;      ;00011010;
00000101;00001010;00100111;00011010;00010110;
00000010;00010110;      ;      ;00010110;00010110
00000011;00010110;00001100;      ;00010111;
00000010;00010111;      ;      ;00010111;00010111
00000010;00010111;      ;      ;00010111;00010111
00000010;00010111;      ;      ;00010111;00010111
00000010;00010111;      ;      ;00010111;00010111
00000010;00010111;      ;      ;00010111;00010111
00000101;00010110;      ;00010111;00011000;
00000110;00011000;00011110;      ;00011001;
00000010;00011001;      ;      ;00011001;00011001
00000101;00011010;00100110;00010111;00011011;
00000001;00011011;      ;00010111;00011000;00011100
00000110;00011100;00011110;      ;00011101;
00000010;00011101;      ;      ;00011101;00011101
00000010;00011110;00101000;00010111;00011111;
00000001;00011111;      ;00010111;00100000;00100000
00000110;00100000;00011101;      ;00101011;
00000101;00100001;00001111;00010111;00100010;
00000001;00100010;      ;00010111;00100011;00100011
00000110;00100011;00011001;      ;00101100;

```

Figura 45- Identificação do mapa binário parte 3

Fonte: (AUTOR, 2009)

6.3 O Processo para conversão de código C em VHDL

De posse dos códigos binários gerados durante o processo de análise léxica, uma função gera o arquivo fonte *VHDL*, que inclui todos os componentes necessários para a execução desse arquivo e todas as instâncias necessárias para a montagem do grafo a fluxo de dados gerado a partir do código C analisado.

Como descrito anteriormente, no conjunto de 0's e 1's do arquivo a ser convertido em *VHDL*, um token representa uma palavra reservada tais como *IF*, *WHILE*, *FOR*, *DO*, que quando identificado, chama um método que passa como parâmetro o binário correspondente para a estrutura que irá gerar o código *VHDL*. Dentro de uma classe *VHDL*, o método irá comparar o valor do parâmetro com as constantes do valor das estruturas. Ao entrar em uma das condições que identificam uma operação, são definidos os atributos nome da entidade, correspondendo como o tipo de grafo, “*Decisor*”, “*Merge Deterministic*”, “*Non Merge Deterministic*”, “*Operator*”, “*Branch*”, “*Copy*” e os números de portas de entrada e saída que a entidade *VHDL* irá conter, respectivamente.

Na ferramenta de conversão, já utilizada para geração de Tabela:Mapa Binário, e TXT:Mapa Binário, foi inserida mais um uma opção chamada de TXT::VHDL, (Figura 46), responsável por gerar o código fonte em linguagem *VHDL* baseado no código binário gerado anteriormente pela ferramenta de análise de *DFG*.

Figura 46 – Ferramenta ChipCFlow com o novo item inserido: TXT::VHDL

Fonte: (AUTOR, 2009)

Tomando como base o grafo descrito na Figura 32 que é o resultado da conversão para o mapa binário “FOR” descrito na Figura 27, e aplicando a função de conversão TXT:VHDL, tem-se como resultado o arquivo descrito na Figura 47, já compilado em ambiente ISE (Xilinx 10.1).

<pre> 1. ----- 2. -- Company: Projeto ChipCflow 3. -- Engineer: kelton Augusto Pontara da Costa 4. -- 5. -- Create Date: 15:30:40 04/07/2009 6. -- Design Name: 7. -- Module Name: VHDL_FOR - Behavioral 8. -- Project Name: 9. -- Target Devices: 10. -- Tool versions: 11. -- Description: 12. -- 13. -- Dependencies: 14. -- 15. -- Revision: 16. -- Revision 0.01 - File Created 17. -- Additional Comments: 18. ----- 19. library ieee; 20. use ieee.std_logic_1164.all; 21. use ieee.std_logic_arith.all; 22. use ieee.std_logic_signed.all; 23. 24. ---- uncomment the following library declaration if instantiating 25. ---- any xilinx primitives in this code 26. --library UNISIM 27. --use UNISIM.VComponents.all 28. 29. ENTITY circuito_final IS 30. port(a : in std_logic_vector(31 downto 0); 31. b : out std_logic_vector(31 downto 0); 32. END; 33. 34. ARCHITECTURE behavior OF circuito_final IS 35. 36. Component merge_deterministic 37. port(38. a: in std_logic_vector(31 downto 0); 39. b: in std_logic_vector(31 downto 0); 40. c: in std_logic_vector(31 downto 0); 41. z: out std_logic_vector(31 downto 0) 42.); 43. END Component; 44. 45. Component copy 46. port(47. a: in std_logic_vector(31 downto 0); 48. z: out std_logic_vector(31 downto 0); 49. y: out std_logic_vector(31 downto 0) 50.); 51. END Component; 52. 53. Component decider 54. port(55. a: in std_logic_vector(31 downto 0); 56. b: in std_logic_vector(31 downto 0); 57. z: out std_logic_vector(31 downto 0) 58.); 59. END Component; 60. 61. Component branch 62. port(63. a: in std_logic_vector(31 downto 0); 64. b: in std_logic_vector(31 downto 0); 65. z: out std_logic_vector(31 downto 0); 66. y: out std_logic_vector(31 downto 0) </pre>	<pre> 70. Component operator 71. port(72. a: in std_logic_vector(31 downto 0); 73. b: in std_logic_vector(31 downto 0); 74. z: out std_logic_vector(31 downto 0) 75.); 76. END Component; 77. 78. signal i10, i18, i14, i11, i19, i13, i20, i16, i12, i15, i21, i17, i121, i122, i141, i142, i141, i142, i141, i142, i151, i152 : std_logic_vector(31 downto 0); 79. 80. BEGIN 81. u1: merge_deterministic port map (i10,i18,i141,i16); 82. u2: merge_deterministic port map (i11,i19,i142,i12); 83. u3: copy port map (i12,i121,i122); 84. u4: decider port map (i13,i121,i14); 85. u5: copy port map (i14,i141,i142); 86. u6: copy port map (i14,i141,i142); 87. u7: copy port map (i14,i141,i142); 88. u8: branch port map (i122,i141,i11,i15); 89. u9: copy port map (i15,i151,i152); 90. u10: operator port map (i151,i20,i11); 91. u11: branch port map (i16,i142,i21,i17); 92. u12: operator port map (i17,i152,i21); 93. END behavior; </pre>
--	---

67.);	
68. END Component;	
69.	

Figura 47 – DFG de um escopo FOR gerado em linguagem VHDL após análise do mapa binário

Fonte: (AUTOR, 2009)

Pode-se observar na figura 47 a descrição dos componentes: *merge_deterministic*; *branch*; *decider*, *copy* e *operator*, iniciando nas linhas 37, 46, 55, 63 e 71 respectivamente. Todos identificados pelos tokens a partir dos 0's e 1's do arquivo TXT:Mapa Binário. A interconexão entre esses componentes é gerada na descrição das instâncias correspondentes, de u1 à u10, nas linhas 81, 82, 83, 84, 85, 86, 87, 88, 89, 90 e 91 respectivamente, que juntos compõem o grafo *FOR* descrito anteriormente.

Outro exemplo de conversão binária de *DFGs* para linguagem *VHDL* é descrito na Figura 48, tendo como base o comando *IF* descrito na Figura 36, cujo arquivo TXT:Mapa Binário é descrito na Figura 34.

Da mesma forma que no exemplo anterior, os componentes estão todos presentes, as instâncias é que mudam, formando a interconexão do grafo para representar o comando *IF*.

1. -----	
2. -- Company: Projeto ChipCflow	
3. -- Engineer: kelton Augusto Pontara da Costa	
4. --	
5. -- Create Date: 17:41:27 03/26/2009	
6. -- Design Name:	
7. -- Module Name: Circuito_IF - Behavioral	
8. -- Project Name:	
9. -- Target Devices:	
10. -- Tool versions:	
11. -- Description:	
12. --	
13. -- Dependencies:	
14. --	
15. -- Revision:	
16. -- Revision 0.01 - File Created	
17. -- Additional Comments:	
18. -----	
19. library ieee;	
20. use ieee.std_logic_1164.all;	
21. use ieee.std_logic_arith.all;	
22. use ieee.std_logic_signed.all;	
23.	
24. ---- uncomment the following library declaration if instantiating	
25. ---- any xilinx primitives in this code	
26. --library UNISIM	
27. --use UNISIM.VComponents.all	
28.	
29. ENTITY circuito_final IS	
30. port(a : in std_logic_vector(31 downto 0);	
31. b : out std_logic_vector(31 downto 0);	
32. END;	
33.	
34. ARCHITECTURE behavior OF circuito_final IS	
35.	
36. Component decider	
	44. Component operator
	45. port(
	46. a: in std_logic_vector(31 downto 0);
	47. b: in std_logic_vector(31 downto 0);
	48. z: out std_logic_vector(31 downto 0)
	49.);
	50. END Component;
	51.
	52. Component merge_deterministic
	53. port(
	54. a: in std_logic_vector(31 downto 0);
	55. b: in std_logic_vector(31 downto 0);
	56. c: in std_logic_vector(31 downto 0);
	57. z: out std_logic_vector(31 downto 0)
	58.);
	59. END Component;
	60.
	61. signal i10, i14, i11, i15, i12, i16, i18, i13, i17, i19 :
	std_logic_vector(31 downto 0);
	62.
	63. BEGIN
	64. u1: decider port map (i10,i14,i18);
	65. u2: operator port map (i11,i15,i13);
	66. u3: operator port map (i12,i16,i17);
	67. u4: merge_deterministic port map (i13,i17,i18,i19);
	68. END behavior;

<pre> 37. port(38. a: in std_logic_vector(31 downto 0); 39. b: in std_logic_vector(31 downto 0); 40. z: out std_logic_vector(31 downto 0) 41.); 42. END Component; 43. </pre>	
---	--

Figura 48 - DFG de um escopo IF gerado em linguagem VHDL após análise do mapa binário

Fonte: (AUTOR, 2009)

No exemplo descrito na Figura 49, tem-se como base o comando *WHILE* descrito na Figura 37 cujo arquivo TXT:Mapa Binário é descrito na Figura 39. Da mesma forma que no exemplo anterior, os componentes estão todos presentes, e as instâncias é que mudam, formando a interconexão do grafo para representar o comando *WHILE*.

<pre> 1. ----- 2. -- Company: Projeto ChipCflow 3. -- Engineer: kelton Augusto Pontara da Costa 4. -- 5. -- Create Date: 12:45:28 04/02/2009 6. -- Design Name: 7. -- Module Name: VHDL_WHILE - Behavioral 8. -- Project Name: 9. -- Target Devices: 10. -- Tool versions: 11. -- Description: 12. -- 13. -- Dependencies: 14. -- 15. -- Revision: 16. -- Revision 0.01 - File Created 17. -- Additional Comments: 18. -- 19. ----- 20. library ieee; 21. use ieee.std_logic_1164.all; 22. use ieee.std_logic_arith.all; 23. use ieee.std_logic_signed.all; 24. 25. ---- uncomment the following library declaration if instantiating 26. ---- any xilinx primitives in this code 27. --library UNISIM 28. --use UNISIM.VComponents.all 29. 30. ENTITY circuito_final IS 31. port(a : in std_logic_vector(31 downto 0); 32. b : out std_logic_vector(31 downto 0); 33. END; 34. 35. ARCHITECTURE behavior OF circuito_final IS 36. 37. Component merge_deterministic 38. port(39. a: in std_logic_vector(31 downto 0); 40. b: in std_logic_vector(31 downto 0); 41. z: out std_logic_vector(31 downto 0) 42.); 43. END Component; 44. 45. Component copy 46. port(47. a: in std_logic_vector(31 downto 0); 48. z: out std_logic_vector(31 downto 0); 49. y: out std_logic_vector(31 downto 0) 50.); 51. END Component; 52. 53. Component decider 54. port(55. a: in std_logic_vector(31 downto 0); </pre>	<pre> 61. Component operator 62. port(63. a: in std_logic_vector(31 downto 0); 64. b: in std_logic_vector(31 downto 0); 65. z: out std_logic_vector(31 downto 0) 66.); 67. END Component; 68. 69. signal i10, i15, i16, i17, i11, i12, i13, i14, i18, i19, i111, i112, i121, i122, i141, i142 : std_logic_vector(31 downto 0); 70. 71. BEGIN 72. u1: merge_deterministic port map (i10,i15,i11); 73. u2: copy port map (i11,i111,i112); 74. u3: decider port map (i111,i16,i12); 75. u4: copy port map (i12,i121,i122); 76. u5: merge_deterministic port map (i112,i121,i13); 77. u6: operator port map (i13,i16,i14); 78. u7: copy port map (i14,i141,i142); 79. u8: operator port map (i141,i17,i18); 80. u9: operator port map (i142,i18,i19); 81. END behavior; </pre>
---	--

56. b: in std_logic_vector(31 downto 0); 57. z: out std_logic_vector(31 downto 0) 58.); 59. END Component; 60.	
---	--

Figura 49 - DFG de um escopo WHILE gerado em linguagem VHDL após análise do mapa binário

Fonte: (AUTOR, 2009)

No último exemplo (Figura 50), tem-se como base a junção de três comandos em alto nível, os comandos *IF*, *FOR* e *WHILE*, cujo arquivo TXT:Mapa Binário, é descrito na Figura 40. Neste exemplo as instâncias, em grande número, é que muda, formando a interconexão do grafo para representar o grafo envolvendo os três comandos *IF*, *FOR* e *WHILE*.

82. ----- 83. -- Company: Projeto ChipCflow 84. -- Engineer: kelson Augusto Pontara da Costa 85. -- 86. -- Create Date: 08:43:20 03/27/2009 87. -- Design Name: 88. -- Module Name: Circuito_completo - Behavioral 89. -- Project Name: 90. -- Target Devices: 91. -- Tool versions: 92. -- Description: 93. -- 94. -- Dependencies: 95. -- 96. -- Revision: 97. -- Revision 0.01 - File Created 98. -- Additional Comments: 99. -- 100. ----- 101. library ieee; 102. use ieee.std_logic_1164.all; 103. use ieee.std_logic_arith.all; 104. use ieee.std_logic_signed.all; 105. 106. ---- uncomment the following library declaration if instantiating 107. ---- any xilinx primitives in this code 108. --library UNISIM 109. --use UNISIM.VComponents.all 110. 111. ENTITY circuito_final IS 112. port(a : in std_logic_vector(31 downto 0); 113. b : out std_logic_vector(31 downto 0); 114. END; 115. 116. ARCHITECTURE behavior OF circuito_final IS 117. 118. Component merge_deterministic 119. port(120. a: in std_logic_vector(31 downto 0); 121. b: in std_logic_vector(31 downto 0); 122. c: out std_logic_vector(31 downto 0); 123. z: out std_logic_vector(31 downto 0) 124.); 125. END Component; 126. 127. Component copy 128. port(129. a: in std_logic_vector(31 downto 0); 130. z: out std_logic_vector(31 downto 0); 131. y: out std_logic_vector(31 downto 0) 132.); 133. END Component; 134. 135. Component decider 136. port(145. a: in std_logic_vector(31 downto 0); 146. b: in std_logic_vector(31 downto 0); 147. z: out std_logic_vector(31 downto 0) 148.); 149. END Component; 150. 151. Component branch 152. port(153. a: in std_logic_vector(31 downto 0); 154. c: in std_logic_vector(31 downto 0); 155. z: out std_logic_vector(31 downto 0); 156. y: out std_logic_vector(31 downto 0) 157.); 158. END Component; 159. 160. signal i10, i36, i12, i30, i15, i33, i18, i37, i20, i38, i39, i40, i11, i13, i14, i41, i16, i17, i19, i42, i21, i26, i22, i23, i24, i25, i27, i28, i29, i31, i32, i43, i34, i35, i44, i111, i112, i131, i132, i171, i172, i211, i212, i221, i222, i231, i232, i231, i232, i231, i232, i231, i232, i231, i232, i231, i232, i251, i252, i291, i292 : std_logic_vector(31 downto 0); 161. 162. BEGIN 163. u1: merge_deterministic port map (i10,i36,i11); 164. u2: copy port map (i11,i11,i12); 165. u3: decider port map (i12,i11,i13); 166. u4: copy port map (i13,i13,i132); 167. u5: merge_deterministic port map (i112,i131,i14); 168. u6: operator port map (i14,i30,i41); 169. u7: merge_deterministic port map (i15,i33,i132,i16); 170. u8: decider port map (i16,i30,i17); 171. u9: copy port map (i17,i171,i172); 172. u10: operator port map (i18,i37,i19); 173. u11: merge_deterministic port map (i19,i15,i171,i42); 174. u12: merge_deterministic port map (i20,i38,i172,i21); 175. u13: copy port map (i21,i211,i212); 176. u14: decider port map (i211,i30,i26); 177. u15: merge_deterministic port map (i10,i39,i26,i22); 178. u16: copy port map (i22,i221,i222); 179. u17: decider port map (i221,i12,i23); 180. u18: copy port map (i23,i231,i232); 181. u19: copy port map (i23,i231,i232); 182. u20: copy port map (i23,i231,i232); 183. u21: copy port map (i23,i231,i232); 184. u22: copy port map (i23,i231,i232); 185. u23: copy port map (i23,i231,i232); 186. u24: merge_deterministic port map (i222,i231,i24); 187. u25: operator port map (i24,i30,i25); 188. u26: copy port map (i25,i251,i252); 189. u27: merge_deterministic port map (i26,i38,i232,i27); 190. u28: branch port map (i27,i233,i28,i28); 191. u29: operator port map (i28,i30,i29); 192. u30: copy port map (i29,i291,i292); 193. u31: merge_deterministic port map (i30,i40,i234,i31); 194. u32: branch port map (i31,i235,i32,i32);
--

137. a: in std_logic_vector(31 downto 0); 138. b: in std_logic_vector(31 downto 0); 139. z: out std_logic_vector(31 downto 0) 140.); 141. END Component; 142. 143. Component operator 144. port(	195. u33: operator port map (i32,i291,i43); 196. u34: merge_deterministic port map (i33,i15,i236,i34); 197. u35: branch port map (i34,i237,i35,i35); 198. u36: operator port map (i35,i251,i44); 199. END behavior;
--	---

Figura 50 - Escopo IF, FOR e WHILE gerado em linguagem VHDL após análise do mapa binário

Fonte: (AUTOR, 2009)

Com a geração dos arquivos fontes *VHDL* e consequente compilação dos mesmos, entende-se ter concluído o principal objetivo deste projeto que foi gerar um grafo a fluxo de dados, a partir de um arquivo fonte originalmente escrito em linguagem C e gerando ao final, um arquivo fonte em *VHDL*, que pudesse ser executado diretamente em ferramentas comerciais, como foi o caso da ferramenta *ISE* da *Xilinx*.

Diferentes exemplos foram implementados, permitindo assim validar o processo de conversão C para linguagem gráfica baseada no modelo a fluxo de dados.

CAPÍTULO 7 – CONCLUSÃO

O Projeto *ChipCflow* vem sendo desenvolvido nos últimos 2 anos, e trata-se de uma ferramenta para execução de algoritmos utilizando o modelo a fluxo de dados dinâmico em hardware reconfigurável. Tem como principal objetivo utilizar o modelo de arquitetura a fluxo de dados, associado ao conceito de arquiteturas reconfiguráveis, para acelerar programas de aplicação escritos em C. Essa aceleração vai acontecer nas partes mais intensas de processamento (exemplo: *loops*), através da execução direta em hardware, aproveitando ao máximo o paralelismo considerado natural do modelo a fluxo de dados (SILVA, 2006; SILVA, 2007; COSTA, 2009; SILVAa, 2009; SILVAb, 2009).

O projeto *ChipCflow* foi dividido em vários módulos, descritos a seguir:

- O módulo de conversão C em Grafo a Fluxo de Dados, que foi o objetivo específico desta tese de doutoramento, e trata-se de um pré-compilador C que gera código *VHDL* para os operadores a fluxo de dados a partir de módulos do programa de aplicação escrito originalmente em C;
- O módulo de execução do modelo a fluxo de dados no *FPGA*, a partir da conversão do programa escrito em *VHDL* para hardware executável (outra tese de doutoramento dentro do projeto);
- O módulo de reconfiguração parcial, que prevê instâncias sendo geradas dinamicamente para cada operador que tiver dados a espera de parceiros e neste caso em particular, algumas frentes de trabalho foi gerado como: o tratamento da reconfiguração parcial (dissertação de mestrado); a partição do grafo original para que a reconfiguração parcial não cause um *overhead* muito grande (dissertação de mestrado); a implementação de um protocolo entre as partições para que haja interação entre as partições (dissertação de mestrado);
- Finalmente o módulo de análise de desempenho, que prevê a execução de *benchmarks* para diferentes programas de aplicação, utilizando toda a estrutura do *ChipCflow*, inclusive com a reconfiguração parcial em execução (dissertação de mestrado).

Para o módulo de conversão C em Grafo a Fluxo de Dados, objetivo desta tese inicialmente, foram realizados estudos sobre compiladores, mais especificamente os métodos de análise sintática e semântica, destacando principalmente a análise feita quando da geração

de grafo de dependência uma vez que esses grafos seriam utilizados como base para a arquitetura a fluxo de dados no projeto.

A partir de toda a análise feita sobre as diferentes partes que envolvem o desenvolvimento de um compilador, passou-se a adequação das partes, diretamente voltadas para a proposta do projeto. Neste caso, uma adequação da geração do grafo de dependência, já otimizado e já direcionado para a arquitetura a fluxo de dados. Essa adequação, basicamente, partiu da proposta dos diferentes operadores existentes no modelo a fluxo de dados do projeto *ChipCflow*, e a combinação prévia entre esses operadores, que permitiram gerar comandos em C como “*IF ... THENELSE*”; *WHILE...DO*; *FOR.....DO*; etc.

Considerando-se os modelos pré-concebidos dos comandos C e utilizando os operadores padrões a fluxo de dados do projeto, foi gerado um arquivo objeto composto basicamente por uma representação binária, em um formato pré-definido, que caracteriza os diferentes operadores e suas interconexões. Isto porque, comandos em alto nível como os descritos acima, acabam por combinar diferentes operadores a fluxo de dados, que ao final acabam configurando o grafo completo de um programa inicialmente escrito em C, mas agora convertido totalmente para uma representação em binário de grafo a fluxo de dados daquele programa.

Como parte fundamental ainda deste módulo, estava a implementação propriamente dita dos operadores em linguagem *VHDL*, e como o grafo a fluxo de dados é uma combinação dos operadores, a simples instanciação dos operadores, com os devidos elementos de interconexão, permitiram a montagem em *VHDL* de um grafo a fluxo de dados pré-concebido. A partir então da representação em binário, do grafo a fluxo de dados do programa originalmente escrito em C, foi relativamente simples converter esse grafo, em um arquivo fonte original em *VHDL*. Assim sendo, o resultado final desse módulo, foi um arquivo fonte em *VHDL*, que pode ser executado em alguma plataforma com ferramentas *EDA* (*Electronic Design Automation*) convertendo então um programa originalmente escrito em linguagem C diretamente para hardware.

Pode-se considerar como principal resultado dessa tese a implementação de algoritmos a partir de uma linguagem C em grafo a fluxo de dados no modelo proposto no projeto *ChipCflow*, onde os operadores a fluxo de dados são os já tradicionalmente conhecidos, e até mesmo a forma de interconexão também, para comandos em alto nível como os citados

acima, entretanto a forma de abordar essa implementação, em *VHDL*, e na forma de fluxo de dados dinâmico, torna o modelo proposto, tanto para o hardware como para o compilador, uma característica única existente até o momento.

A forma de implementar algoritmos a partir de *C* em Grafo a Fluxo de Dados no modelo proposto para o *ChipCflow*, é portanto, uma contribuição original, pois a forma como os operadores são tratados, mesmo sendo já propostos por *Veen*, é que torna o procedimento todo original, utilizando todas as análises feitas para compiladores, gerando um arquivo binário que representa o modelo original de execução da máquina a fluxo de dados do projeto *ChipCflow*.

Devido a complexidade no desenvolvimento da ferramenta uma série de parâmetros ainda necessitam ser tratados no compilador, mais precisamente os operadores NTM (new tag mannager), NIG (new iteration generation) e NTD (new tag destruction), que são operadores específicos voltados para a reconfiguração dinâmica, e como esses operadores são parte integrantes de um grafo a fluxo de dados dinâmico, os mesmo devem ser incluídos no compilador para tratar as construções iterativas que são as estruturas que necessitam de um sincronismo toda vez que um novo dado circula por elas.

São nesses operadores NTM, NIG e NTD que estão embarcados o tratamento dos valores iniciais de algumas variáveis, dos dados de entrada em um loop ou rotina, dos dados que circulam por um loop, e dos dados que saem de um loop ou rotina, uma vez que serão esses operadores que irão tratar dos *tagged tokens* existentes para cada dado que circulam pelo grafo.

Vale ressaltar que este estudo já possui 2 artigos publicados pelo *V Southern Programmable Logic Conference SPL 2009* realizado em São Carlos e um journal pelo *World Scientific and Engineering Academy and Society WSEAS 2009 Conferences* a ser realizado em Tbilisi na Georgia.

7.1 Trabalhos futuros

Como descrito acima, o projeto *ChipCflow* foi dividido em vários módulos, entretanto no caso específico deste módulo de conversão *C* em Grafo a Fluxo de Dados e se tratando de uma nova ferramenta não desenvolvida até a presente tese, uma série de parâmetros ainda precisam ser tratados no compilador. Dentre eles, estão os operadores

NTM (new tag mannager), NIG (new iteration generation) e NTD (new tag destruction), que são operadores específicos voltados para a reconfiguração dinâmica, e como esses operadores são parte integrantes de um grafo a fluxo de dados dinâmico que é executado em um ambiente com reconfiguração parcial, os mesmo devem ser incluídos do grafo final após conversão do programa originalmente escrito em linguagem C, para a representação binária de onde é gerado o grafo e o programa fonte em VHDL para execução direta no hardware.

São nesses operadores NTM, NIG e NTD que estão embarcados o tratamento dos valores iniciais de algumas variáveis, dos dados de entrada em um loop ou rotina, dos dados que circulam por um loop, e dos dados que saem de um loop ou rotina, uma vez que serão esses operadores que irão tratar dos *tagged tokens* existentes para cada dado que circulam pelo grafo.

Finalmente uma integração entre o pré-compilador gerando o código VHDL e uma ferramenta EDA, para que o ambiente seja completo, desde a concepção do programa escrito originalmente em C até a geração final em hardware, nos moldes de programas como *StreamC*, *Spark*, etc.

REFERÊNCIAS

- ACKLAND, B; et al. A Single-Chip 1.6 Billion 16-b MAC/s Multiprocessor DSP. In: **IEEE Journal of Solid-State Circuits**, p. 412-424, Mar. 2000.
- ALTERA – “**System On Programmable Chip**” – San Jose, California, 2004
- ATHANAS, P. M. - "An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration", Ph.D. thesis, Brown University, USA, 1992
- ATHANAS, P. M.; SILVERMAN, H. F. **Processor Reconfiguration through Instruction-Set Metamorphosis**; IEEE Computer, v. 26, no. 3, Mar. 1993
- ARNOLD, J. The SPLASH 2 Software Environment. **Ieee Workshop on FPGAS For Custom Computing Machines**, p. 88-93, April, 1993.
- ARNOLD, J.; BUELL, D; DAVIS, E. SPLASH 2. In: **Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures**, p. 316-324, 1992.
- ARQUITETURA PIC. Instituto de Computação. **Universidade de Campinas**, Campinas. Disponível em: <http://www.dcc.unicamp.br/~celio/mc404/pic/pic_arquitetura.html>. Acesso em: 10 jan. 2003.
- ARVIND, K.; NIKHIL, R.S. **Executing a Program on the MIT Tagged-Token Dataflow Architecture**. IEEE Transactions on Computers, volume 39 issue 3, Mar.1990.
- AHO, A. V. **Compiladores: princípios, técnicas e ferramentas**. Rio de Janeiro: LTC, 1995.
- ARVIND Dataflow: **Passing the token**. In: ISCA Keynote, 2005.
- BANNATYNE, R.; VIOT, G. Introduction to Microcontrollers – Part 1 and 2. **Embedded Systems Conference**, San Francisco, 1998.
- BROWN, S.; ROSE, J. **Architecture of FPGA and CPLDs: A tutorial**; IEEE Design and Test of Computers, v. 13, 1996.
- BROWN, S.; ROSE, J.; FRANCIS, R.; VRANESIC, Z. **Field Programmable Gate Array**. Kluwer Academic Publishers, Jun. 1992.
- CAPPELLI, A.; LODI, A.; MUCCI, C.; TOMA, M.; CAMPI, F. **A dataflow control unit for c-to configurable pipelines compilation flow**. In: IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, p. 332-333.

CAPPELLI, A.; LODI, A.; MUCCI, C.; TOMA, M.; CAMPI, F. **A Dataflow Control Unit for C-to-Configurable Pipelines Compilation Flow.** ARCES - University of Bologna, Bologna, Italy. Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04), Apr. 2004.

CARRILLO, J. E.; CHOW, P. The Effect of Reconfigurable Units in Superscalar Processors. **Acm/Sigda International Symposium on Field Programmable Gate Array - FPGA 2001**, Fev. 2001. p. 141-150.

CARDOSO J. M. P.; VESTIAS M. P. **Architectures and compilers to support reconfigurable computing**; ACM Crossroads Student Magazine, 2000. Disponível em: <<http://www.acm.org/crossroads/xrds5-3/rcconcept.html>>; Acessado em: 15 fev. 2004.

CARDOSO, J. Compilação de Algoritmos em Java para Sistemas Computacionais Reconfiguráveis com Exploração do Paralelismo ao Nível das Operações. Dissertação Apresentada ao Programa de Pós-Graduação a Universidade Técnica de Lisboa, Lisboa, outubro, 2000.

CARDOSO, J.; NETO, H. **Compilation for FPGA-Based Reconfigurable Hardware.** IEEE Design & Test of Computers Magazine, 2003.

CARDIM, M.H.C. **“RtraSSoc51 - Adaptável Superescalar Reconfigurável Sistema em Chip - O Processador Harward Superescalar”** - Programa de Pós-Graduação em Ciência da Computação do UNIVEM para obtenção do título de Mestre, Marília, S.P., 2005.

CELOXICA Ltd. **Software Product Description for DK version 4.0 SP2. Technical Library - White Papers**, 2006. Disponível em: <http://www.celoxica.com/support/view_article.asp?ArticleID=521>. Acesso em 13 dez 2007.

CELOXICA Ltd. **Introducing Software Paradigms to Hardware Design. Technical Library - White Papers**, 2002. Disponível em: <<http://www.celoxica.com/techlib/default.asp?Action=1&CatID=23&CatType=1&OrderBy=1>>. Acesso em: 10 dez 2007.

CELOXICA Ltd. **The Application of Retiming to the Synthesis of C Based Languages Using the Celoxica DK Design Suit. Technical Library - White Papers**, 2004. Disponível em: <<http://www.celoxica.com/techlib/default.asp?Action=1&CatID=23&CatType=1&OrderBy=1>>. Acesso em: 13 dez 2007.

CELOXICA Ltd. **Handel-C Language Reference Manual. Technical Library - Handel-C**, 2004. Disponível em: <<http://www.celoxica.com/techlib/default.asp?Action=1&CatID=9&CatType=2&OrderBy=1>>. Acesso em: 13 dez 2007.

CORREIA, V.M. - **Execução de Algoritmos Utilizando o Modelo a Fluxo de Dados Dinâmico para Hardware Reconfigurável - Prova de Conceito do Modelo a Fluxo de Dados.** Programa de Pós-Graduação ao Instituto de Ciências Matemáticas e de Computação, ICMC/USP, São Paulo, 2007.

COSTA, V. S. **Estudo Comparativo de Projetos de hardware Orientado a Controle em Níveis Comportamental e de Transferência entre Registradores.** Recife – 2006.

COSTA, R. M. **Microcontroladores em Computação Reconfigurável – Uma Aplicação.** Mestrado em Ciências da Computação – Universidade Federal de São Carlos, São Carlos, 2000.

CHAN, P. K.; MOURAD, S. *Digital design using field programmable gate arrays.* Prentice Hall, Abr. 1994.

CHAPPELL, S. Et al. **Handel-C for co-processing & co-design of Field Programmable System on Chip.** Setembro 2002- Novembro 2006.
<<http://www.celoxica.com/techlib/default.asp?Action=1&CatID=23&CatType=1&OrderBy=1>>. Acesso em: 10 dez 2007.

CHICHKOV, A.; BELTRÁN, A.C. **A Hardware/Software Partitioning Algorithm For Custom Computing Machines.** Lecture Notes In Computer Science; Vol. 1304. Germany: [s.n.], 1997. pp. 274-283.

CHICHKOV, A.; BELTRÁN, A.C. **A Tool for Mapping a C Description into a Hardware Configuration using VHDL at RTL.** DCIS'98 – XIII Design Of Circuits and Integrated Systems Conference. Madrid: [s.n.], 1998, pp.17-20.

DANDALIS, A.; PRASANA, V. K. **Configuration Compression for FPGA-based Embedded Systems.** *Acm/Sigda International Symposium on Field Programmable Gate Array – FPGA 2001*, Fev. 2001. p. 173-182.

DAVIS, A. L. **The architecture and system method of ddm1: A recursively structured data driven machine.** In: ISCA '78: Proceedings of the 5th annual symposium on Computer architecture, New York, NY, USA: ACM Press, 1978, p. 210–215.

DEHON, A. **DPGA Utilization and Applications**, Proc. ACM International Symposium on FPGAs, 1996, pp. 115-121. Disponível em
<<http://citeseer.ist.psu.edu/article/dehon96dpga.html>>. Acesso em: 10 dez 2007.

DELAMARO, M. E. **Como construir um Compilador: utilizando ferramentas Java.** São Paulo: Novatec, 2004.

DENNIS, J. B.; MISUNAS, D. P. **A preliminary architecture for a basic dataflow processor.** In: in Proceedings of the 2nd Annual Symposium on Computer Architecture, 1974, p. 126-132.

DENNIS, J. B. Data Flow Supercomputer. **IEEE Computer**, v. 13, n. 11, p. 48-56, Nov. 1980.

DENNIS, J. B. **First Version of a data Flow Produces Language Lecture Notes in Computer Science.** In: in Proceedings of the 2nd Annual Symposium on Computer Architecture, v. 19, p. 362-372, 1975.

DOUCET, F.; GUPT, R. K. **Microelectronic System-on-Chip Modeling using Objects and their Relationships,** [s.l.][s.d.]. Disponível em: <<http://www.cecs.uci.edu/~balboa/pubs/doucet-osee1.pdf>>. Acesso em: 22 jan. 2003.

DUBOIS, M. et al. **Rapid hardware prototyping on RPM-2;** IEEE Design and Test of Computers, v.15, no. 3, Set. 1998.

ELDREDGE, J.G.; HUTCHINGS, B.L. **Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration.** In D. A. Buell and K. L. Pocek, editors, Second Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '94, pages 180- 188, Napa Valley, California, USA, April 10-13 1994. IEEE, Inc.

FERRARI, J. D. Et al. **Aplicação de Loop Pipelining e Loop Unrolling à Síntese de Alto Nível.** Disponível em: <<http://www.laps.inf.ufsc.br/publications.htm>>. Acesso em: 21 dez 2006.

FERRARI, D. J. **Aplicações de Loop Pipelining e Loop Unrolling à síntese de alto nível,** Universidade de Santa Catarina – UFSC – 2001.

GONZÁLES, J. A. Q. **Algoritmo de Otimização de planos de teste de unidades funcionais para circuitos bits.** São Paulo – 2001.

GOKHALE, M. et al. Building and Using a Highly Parallel Programmable Logic Array. In: **IEEE COMPUTER**, Jan. 1991. p. 81-89.

GOKHALE, M. et al. **SPLASH: A reconfigurable Linear Logic Array;** in Proceedings of the International Conference on Parallel Processing (ICPP'97), v. 9, Ago. 1990.

GOKHALE, M and W. Holmes and A. Kopser and S. Lucas and R. Minnich and D. Sweely and D. Lopresti – **“Building and Using a Highly Parallel Programmable Logic Array”** – IEEE Computer, pp 81-89, Jan. 1991.

GONÇALVES R. A.; Wolf D. F.; Coelho F. A. S.; Teixeira M. A.; Ribeiro A. A. L.; Marques E. **Architect: Um sistema de computação reconfigurável**. In CORE2000 Workshop de Computação Reconfigurável, 2000.

GUCCIONE, S. **List of FPGA-based computing machine**, [s.l.], Ago. 2000. Disponível em: <http://www.io.com/~guccione/HW_list.html> . Acesso em: 22 jan. 2003.

GUPTA SUMIT et al. **Spark - A Parallelizing Approach to the High_Level Synthesis of Digital Circuits**. Kluwer Academic Publishers, 2004.

HUNTER, R. **Compilers**. Their Design and Construction using Pascal. New York: Editora Newnes Books, 1986.

HAUCK, S. The Roles of FPGAs in Reprogrammable Systems. **Proceedings of the IEEE**. V. 86, p.615-638, Abr. 1998.

HAUSER, J.R.; WAWRZYNEK, J. **Garp: a MIPS processor with a reconfigurable coprocessor**, fccm, p. 12, 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97). Disponível em: <<http://doi.ieeecomputersociety.org/10.1109/FPGA.1997.624600>> . Acesso em: 10 dez 2007.

HWANG, K.; BRIGGS, F. A. **Computer Architecture and Parallel Processing**. New York: McGraw-Hill Book Company, 1984. 846p.

HENESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 2 ed. San Francisco: Morgan Kaufmann, 1996.

ISELI, C. **Spyder - A Reconfigurable Processor Development System**. Tese (Doutorado) - École Polytechnique Fédérale de Lausanne, Lausanne, 1996.

JASINSKI, R. P.; COPETTI, L. F.; SCHEIDER, F. K. **Introdução Da Linguagem Vhdl No Ciclo De Desenvolvimento De Um Circuito Digital**. Centro Federal de Educação Tecnológica do Paraná / Núcleo de Pesquisa em Engenharia Simultânea, [s.l.], [s.d.]. Disponível em: <<http://www.nupes.cefetpr.br/jasinski.pdf>> . Acesso em: 16 jan. 2003.

KOZYRAKIS et al., **Scalable Processors in the Billion-Transistor Era: IRAM**. In: IEEE Computer, v. 30, n. 9, p. 75-78, Set. 1997.

LAHIRI , K.; RAGHUNATHAN, A.; DEY, S. **Evaluation of the Traffic-Performance Characteristics of System-on-Chip Communication Architectures**, [s.l.][s.d.]. Disponível em <<http://www.computer.org/proceedings/vlsid/0831/0831toc.htm>> . Acesso em: 25 jan. 2003.

LEWIS, D. M. et al. *The Transmogripher-2: A 1 million gate rapid prototyping System*; IEEE Transactions on VLSI Systems, Jun 1998.

LYSAGHT, P.; DUNLOP, J. **Dynamic Reconfiguration of Field Programmable Gate Arrays**; in Proceedings of the 3o. International Workshop on Field Programmable Logic and Applications (FPL '93), Oxford, UK, Abingdon EE&CS Books, Set. 1993.

LI, Y.; WOLF, W. H. **Hardware/Software Co-Synthesis with memory Hierarchies**. In: IEEE Transaction Computer-Aided Design of Integrated Circuits and Systems, v. 18, n. 10, out. 1999. P. 1405-1417.

LOPES, J.J. **Estudos e Avaliações de Compiladores para Arquiteturas Reconfiguráveis**. Instituto de Ciências Matemáticas e de Computação ICMC-USP para obtenção do título de Mestre, São Carlos, S.P., 2007.

LOUDEN, K. C. **Compiladores – Princípios e práticas**. São Paulo: Thomson Pioneira, 2004.

MARCHI, L. Projeto e Implementação de uma Unidade Lógica Arimética (ULA) utilizando VHDL e PLD's. **Universidade Católica Dom Bosco**, Campo Grande, 2002. Disponível em: <<http://www.Ec.ucdb.br/engenharia/projeto-graduação/projetos-2002/projetos/projg2.pdf>>. Acesso em: 16 jan. 2003.

MARQUES, F. S. Microcontrolador BDLC – Uma Implementação para Aplicação na Indústria Automobilística. **Universidade Federal de Pelotas**, Pelotas 2000. Disponível em: <<http://www.ufpe.tche.br/prg/sisbi/bibct/acervo/info/2000/Mono-Felipe.pdf>>. Acesso em: 16 jan. 2003.

MASSIMO, B. et al., **HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform**. ACM/SIGDA. In: 10th International Workshop on Hardware/Software Codesign, pp 151-156, 2002.

MESQUITA, D.; MORAES, F.; MOLLER, L.; CALAZANS, N. **Reconfiguração Parcial e Remota de Dispositivos FPGA da Família Virtex**. [s.l.[s.d.]. Disponível em: <http://www.inf.pucrs.br/~moraes/papers/2001_scr_reconf.pdf>. Acesso em: 10 jan. 2004.

MENSCH Jr., W. D.; SILAGE, D. **System-on-Chip design methodology in engineering education**, [s.l.][s.d.]. Disponível em: < <http://www.temple.edu/scdc/icee2000.pdf> >. Acesso em: 22 jan. 2003.

MERINO, P.; LOPES, J. C.; JACOME, M. **A Hardware Operating System for Dynamic Reconfiguration of FPGAs**, in Proceedings of the 8o. Internacional Workshop on Field Programmable Logic and Applications (FPL '98), Tallin, Estonia, Springer (LNCS 1482), Set. 1998.

Mirsky, E.; DeHon, A. **MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources**. IEEE Symposium on

FPGAs for Custom Computing Machines. Disponível em: <<http://citeseer.ist.psu.edu/mirsky96matrix.html>> . Acesso em: 10 dez 2007.

MORAES, F.; MESQUITA, D. Tendências em Reconfiguração dinâmica de FPGAs. **Faculdade de Informática - PUCRS**. [s.l.][s.d.] Disponível em: <<http://www.inf.pucrs.br/~moraes/papers/2001-scr2001-palestra.pdf>> . Acesso em: 20 Jan. 2003.

NAKAMURA, T. **Preface Special Issue on System-on-Chip (SOC)**, [s.l.], 2000. Disponível em: <<http://www.temple.edu/scdc/icee2000.pdf>> . Acesso em: 22 jan. 2003.

OKAMOTO Jr, J.; SASAKI, T. M. Utilização de Processadores Digitais de Sinais na Implementação de Sistemas de Controle de Robôs. **Departamento de Engenharia Mecânica da Escola Politécnica da Universidade de São Paulo**, São Paulo, Jun. 1998. Disponível em: <<http://www.mcca.epusp.br/~jokamoto/9806.pdf>> . Acesso em: 20 dez. 2002.

ORDONEZ, E. et al. **Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs)**. 1. ed. Pompéia: Bless, 2003.

PANDA, P. R. et al., **Memory Data Organization for Improved Cache Performance In Embedded Processor Applications**. In: ACM Transactions on Design Automation of Electronic, v. 2, n. 4, p. 384-409, Out. 1997.

PARMA, G. G. Introdução à VHDL. **Universidade Federal de Minas Gerais**, Minas Gerais Disponível em: <<http://www.cpdee.ufmg.br/~parma/spp/Prat02.pdf>> . Acesso em: 15 Jan. 2003.

PELLERIN, D.; SCOTT, T. **Practical FPGA Programming in C**. [s.l.]: Prentice Hall PTR, 2005.

PIACENTINO, M. R.; GOOITZEN, S. V. W.; HANSEN, M. W. **Reconfigurable Elements for a Video Pipeline Processor**. In: Ieee Symposium on FPGAS For Custom Computing Machines, 1999. p. 1-10.

PONTARA, K. A. ; RODA, V. O. ; SILVA, J. L. . **C Conversion Module for a Dynamic Dataflow Tool**. In: 5th. IEEE Southern Programmable Logic Conference - SPL 2009, 2009, São Carlos. SPL - V Southern Programmable Logic Conference, 2009. p. 1-1.

QUENOT, G. et al. **A Reconfigurable Computing Engine for Real-Time Vision Automata Prototyping**. In: IEEE Workshop on FPGAs ofr Custom Computing Machine, Abr. 1994.

RADUNOVIC, B.; MILUTINOVIC, V. **A Survey of Reconfigurable Computing Architectures**. Tutorial in 8th Int. Workshop on Field Programmable Logic and Applications (FPL'98). Published in Springer-Verlag LNCS 1482, Hartenstein and Keevallik eds, Berlin,

Germany, 1998, pp. 376-385.

RATHA, N.; JAIN, A.; ROVER, D. **Convolution on SPLASH 2**. In: IEEE Workshop on FPGA for Custom Computing Machine, Abr. 1995, p. 204-213.

RIBEIRO, A. A. L. **Reconfigurabilidade dinâmica e remota de FPGAs**. Tese de Mestrado, Universidade de São Paulo – ICMC, São Carlos –SP, Jul. 2002.

RODRIGUES, L. C. R. **Computação Reconfigurável – FPGA**. Especialização em Computação Avançada – Fundação Eurípides Soares da Rocha, Marília, 2000.

RODRIGUES, L. C. R. **Microcontroladores Intel 8051**. Especialização Computação Avançada – Fundação Eurípides Soares da Rocha, Marília, 2000.

SCHEITRUM, M. **Existing System Design Methods to Easy SOC Design**, [s.l.], mai. 2000. Disponível em: <http://www.ictpld.com/jobs/prkit/simplify_soc.pdf> . Acesso em: 25 jan. 2003.

SHALAN, M.; MOONEY III, V. J. **Hardware Support for Real-Time Embedded Multiprocessor System-on-a-Chip Memory Management**. [s.l.], 2002. Disponível em: <<http://codesign.ece.gatech.edu/publications/shalan/paper/CODES2002.pdf>> . Acesso em: 10 Fev. 2003.

SILVA, J. et al. **RtrASSoc – An Adaptable Superscalar Reconfigurable System on Chip – The Simulator**. In: The 3rd IEEE International Workshop on System-on-Chip for Real Time Applications, IWSOC 2003, Calgary, Alberta, 2003.

SILVA, J. L. **Processamento a Fluxo de Dados Tolerante a Falhas em um Computador Paralelo**. Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica, Departamento de Telemática. Tese Doutorado, 1992.

SILVA Jr., V. P. **Aplicações Práticas do Microcontrolador 8051: Teoria Geral Detalhada**. 9. ed. São Paulo: Érica, 1998.

SILVA, J. L. **RtrASSoc51-Adaptable Superscalar Reconfigurable Programmable System on Chip -The Reconfigurable tools for DSR a Development System**. In: **IP-SOC 2005 IP Based SoC Design Conference**, Grenoble, 2005.

SILVA, J. L.; PONTARA, K. A. **RtrASSoc - Adaptable, Superscalar, Reconfigurable, Programmable System on Chip - The Embedded Operating System - EOS**. In: The 4th IEEE International Workshop on System-on-Chip for Real-Time Applications, 2004, Alberta. **Proceedings of the 4th IEEE International Workshop on System-on-Chip for Real-Time Applications**, v. 4. p. 283-286, 2004.

SILVA, K. G. Projeto de um Controlador de Rede com Tolerância a Falhas Baseado em Computação Reconfigurável. **Universidade de São Paulo**, São Paulo, [s.d]. Disponível em: <<http://www.dedalus.usp.br:4500/aleph/por/usp/usp/dedalus/full/1224054>>. Acesso em: 20 jan. 2003.

SILVA, J., COSTA, K. A. P. **RtrASSoc - An Adaptable Superscalar Programmable Reconfigurable System on Chip - The Operational System Inlaid - Storage and Vectorial Allocation**. The 4th IEEE International Workshop on System-on-Chip for Real-Time Applications, IWSOC, Banff, Alberta, 2004.

SILVAa, J. L. ; LOPES, J. J. ; PONTARA, K. A.; RODA, V. O. **ChipCflow- A Dynamic Dataflow Machine using Dynamic Reconfigurable Hardware**. In: 5th. IEEE - Southern Programmable Logic Conference - SPL 2009, 2009, São Carlos. SPL - V Southern Programmable Logic Conference, 2009. p. 1-4.

SILVAb, J. L. PONTARA, K. A. C., RODA, V. P. - **The C Compiler generating a Source file in VHDL for a Dynamic Dataflow Machine**. WSEAS Conferences in Tbilisi, Georgia, June 26-28, 2009.

SILVA, J. L.; MARQUES, E. Executing algorithms for dynamic dataflow reconfigurable hardware - the operators protocol. In: RECONFIG '06: The 3rd International Conference on Reconfigurable Computing and FPGAs, 2006.

SYNFORA Pico Express [online]. **Synfora Revolutionizing Complex SoC Design**. Synfora, 2007. Disponível em: <<http://www.synfora.com/products/picoexpress.html>>. Acesso em: 10 dez 2007.

SOUZA, D. J. **Desbravando o PIC Baseado no Microcontrolador PIC 16F84**. 2. ed. São Paulo: Érica, 2000.

SWANSON, S.; OSKIN, M.; MICHELSON, K. **WaveScalar (2003)**. Disponível em: <citeseer.ist.psu.edu/article/swanson03wavescalar.html>. Acesso em: 10 dez 2007.

SWANSON, S.; MERCALDI, M.; PETERSEN, A.; PUTNAM, A.; SCHWERIN, A.; OSKIN, M.; EGGERS, S. J.; MICHELSON, K. **Area-performance trade-off in tiled dataflow architectures**. In: International Symposium on Computer Architecture, 2006.

TEIXEIRA, M. A. **Técnicas de Reconfigurabilidade dos FPGAs da Família APEX 20K - Altera**. 2002. Mestrado em Ciências da Computação - Universidade de São Paulo, São Carlos.

TRELEAVEN, P. C.; BROWNBRIDGE, D. R.; HOPKINS, R. P. **Data-driven and demand-driven computer architecture**. ACM Comput. Surv., v. 14, n. 1, p. 93-143, 1982.

TEXAS INSTRUMENTS. [s.l.][s.d.], Disponível em:

<http://dspvillage.ti.com/docs/catalog/dspplatform/details.jhtml?templateId=5121&path=templatedata/cm/dspdetail/data/vil_getstd_whatIs>. Acesso em: 5 Dez. 2002.

TRELEVEAN, P. C.; BROWNBRIDGE, D. R.; HOPKINS, R. P. Data-Driven and Demand-Driven Computer Architecture. **ACM Computer Surveys**, v. 14, n.1, p. 93-143, Mar. 1982.

THOMPSON, T. **ComputerWorld An IDG Company**. Digital Signal Processor, [s.l.], 2001. Disponível em: <http://www.computerworld.com/cwi/story/0,1199,nav47-68-85-1950_sto58487,00.html>. Acesso em: 5 jan. 2003.

VEEN, A. H. **Dataflow machine architecture**. In: **ACM Computing Surveys**, 1986, p. 365-396.

VERONESI, R.L.M. - **RTRASSOC51 - Módulo de Comunicação I2c Reconfigurável rI2C**. Defesa de Dissertação Apresentada ao Programa de Pós-Graduação em Ciência da Computação do UNIVEM para obtenção do título de Mestre, Marília, S.P., 2005.

ASTOLFI, V. F.; SILVA, J. L. - **Execution of algorithms using a Dynamic Dataflow Model for Reconfigurable Hardware - Commands in Dataflow Graph**. The 3th IEEE - Southern Conference on Programmable Logic - SPL 2007, Mar del Plata, Argentina, 2007.

VIRTEX-II Pro Platform FPGAs.[s.l.][s.d.]. Disponível em: <<http://www.xilinx.com/virtex2pro>>. Acesso em 20 Fev. 2003.

VIRTEX-5 Platform FPGAs.[s.l.][s.d.]. Disponível em: <www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/index.htm>. Acesso em 04 Set. 2007.

WIRTHLIN, M.; HUTCHINGS, B. L. **A Dynamic Instruction Set Computer**. In proceedings of the 3rd IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '95), Napa Valley, California, IEEE Computer Society Press, Abr 1995.

Wirthlin, M.; Hutchings, B.L. **Dynamically-Configurable Digital Processor Using Method For Relocating Logic**. Brigham Young University (Provo, UT). Disponível em: <<http://www.google.com/patents?hl=pt-BR&lr=&vid=USPAT6173434&id=dXB8AAAAEBAJ&oi=fnd>>. Acesso em: 10 dez 2007.

XILINX Inc. **Virtex II 1.5v Field Programmable Gate Arrays**. Disponível em: <<http://www.xilinx.com/partinfo/ds031.pdf>>, Out. 2001.

XILINX - **"Embedded Operating System"**- San Jose, California, 2004.

_____. The Varieties of Data Flow Computer. In: **Proc. Of the First International Conference on Distributed Computing Systems**, p. 430-439, Oct. 1979.