SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 05/05/2010

Assinatura: _____

LALP: uma linguagem para exploração do paralelismo de *loops* em computação reconfigurável

Ricardo Menotti

Orientador: *Prof. Dr. Eduardo Marques*

Co-orientador: *Prof. Dr. João M. P. Cardoso*

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação (ICMC/USP), como parte dos requisitos para obtenção do título de Doutor em Ciências da Computação e Matemática Computacional.

USP - São Carlos

Maio de 2010

Dedico este trabalho aos meus queridos pais, José e Elda

Agradecimentos

Aos meus pais, José e Elda, pelo carinho, dedicação e pelas orações. Aos meus irmãos Rodrigo e Regiane, pelo apoio em todos os momentos. À Ana Rubia, pelo incentivo e compreensão.

Aos meus orientadores, professor Eduardo Marques e professor João Cardoso da FEUP, pelo apoio, confiança, orientações acadêmicas e pessoais, e pela amizade. Ao professor Marcio Fernandes da UFSCar, pela ajuda nos trabalhos realizados em cooperação. Ao professor João Lima da UAlg, pelo companheirismo e pela acolhida no Algarve.

Aos amigos “de” São Carlos, pelas alegrias e dificuldades compartilhadas: André Domingues, Carlos Almeida, Cassio Oishi, Fabiano Ferrari, Mário Pazoti, Otávio Lemos, Reginaldo Ré, Renato Ishii, Rodrigo Pedra, Rogério Garcia e Vanderlei Bonato; a todos os demais colegas do ICMC que contribuíram direta ou indiretamente para realização deste trabalho, perdoem-me pelo esquecimento neste momento.

Ao pessoal da UTFPR em Campo Mourão, pela colaboração durante os períodos em que tive que me ausentar. Em especial aos colegas André Kawamoto, Celso Gandolfo (*in memoriam*), Ivanilton Polato, Narci Nogueira, Radames Halmeman e Reginaldo Ré, pelo apoio na concessão do meu afastamento do país.

Aos colegas e agregados da Residência dos Baldaques, em Lisboa, pelos momentos compartilhados longe de casa.

Ao Banco Santander pelo suporte financeiro que proporcionou o período de estágio em Lisboa. Ao CNPq e à FCT pelo suporte financeiro concedido nos convênios de cooperação internacional Brasil/Portugal.

porque sem mim nada podeis fazer
(João 15:5)

Sumário

Lista de Figuras	iv
Lista de Tabelas	v
Lista de Algoritmos	viii
Lista de Abreviaturas	ix
Resumo	xi
Abstract	xiii
1 Introdução	1
1.1 Contextualização	1
1.2 Motivação	3
1.3 Objetivo	4
1.4 Contribuições	5
1.5 Organização	5
2 Computação Reconfigurável	7
2.1 Conceitos Básicos	7
2.2 Fluxo de Desenvolvimento	13
2.3 Recursos dos FPGAs Atuais	14
2.4 <i>Softcore Processors</i> e Co-projeto	17
3 Técnicas de Compilação	19
3.1 Notação Informal de Algoritmo para Compilador	20
3.2 Fluxo Básico de Compilação	21
3.3 Compiladores Otimizantes	23
3.4 Representações Intermediárias	24
3.5 Técnicas de Otimização	28
3.6 <i>Loop Pipelining</i>	33
3.7 Considerações Finais	41

4	Trabalhos Relacionados	45
4.1	Por que Ferramentas de Geração de <i>Hardware</i> ?	47
4.2	C2H	48
4.3	SPARK	50
4.4	C-to-Verilog	52
4.5	ROCCC	53
4.6	Análise Comparativa	54
5	A Linguagem LALP	57
5.1	Especificação da Linguagem	58
5.2	Limitações Impostas pela Linguagem	72
5.3	A Linguagem LALP-S	75
5.4	Extensões Possíveis	76
6	<i>Mapeamento de LALP em FPGAs</i>	79
6.1	Abordagem	80
6.2	Biblioteca de Componentes	82
6.3	Representação Intermediária	83
6.4	Algoritmos	85
6.5	Visualização	94
6.6	Interface Gráfica do Usuário	96
7	Resultados	99
7.1	Conjunto de <i>Benchmarks</i>	100
7.2	Resultados Preliminares	102
7.3	Resultados com <i>ADPCM Coder/Decoder</i>	104
7.4	LALP Comparado ao C-to-Verilog	106
7.5	Impacto dos Algoritmos de Escalonamento	108
7.6	LALP Comparado ao ROCCC e C-to-Verilog	111
7.7	LALP Comparado a Processadores Embarcados	115
7.8	Exploração do Espaço de Projeto	116
7.9	Consumo de Potência	121
8	Conclusão	123
8.1	Trabalhos Futuros	125
	Referências Bibliográficas	127
A	Especificação Formal da Linguagem	137
B	Códigos Fonte dos <i>Benchmarks</i>	145

Lista de Figuras

2.1	Computação reconfigurável comparada às soluções de <i>hardware</i> e <i>software</i> . . .	8
2.2	Relações de mercado de lógica digital (Hamblen e Furman, 2001)	9
2.3	Relação entre flexibilidade e desempenho (Bobda, 2007)	10
2.4	Estrutura básica de um FPGA	10
2.5	Uso de LUTs para implementação de funções lógicas	11
2.6	Circuito de uma LUT de três entradas	12
2.7	Aumento do número de transistores (Bondalapati e Prasanna, 2002)	12
2.8	Fluxo de desenvolvimento para FPGAs	13
2.9	Estrutura dos LABs nos FPGA da família Stratix IV	15
3.1	Estrutura em alto nível de um compilador simples (Muchnick, 1997)	23
3.2	DFG representando um bloco básico	27
3.3	Ganho de desempenho obtido com <i>loop pipelining</i>	36
3.4	Exemplo de <i>loop pipelining</i>	37
3.5	Exemplo de <i>unroll-and-compact</i>	40
3.6	Exemplo de <i>window scheduling</i>	41
4.1	Vendas de ferramentas para síntese de alto nível (Martin e Smith, 2009)	46
4.2	Integração de um módulo gerado pelo C2H ao sistema (AlteraURL, 2008a)	48
4.3	Fluxo da ferramenta desenvolvida no projeto SPARK (Gupta <i>et. al.</i> , 2004b)	51
4.4	Visão geral do compilador ROCCC (Guo <i>et. al.</i> , 2005)	54
5.1	Simulação do componente contador	62
5.2	Escalonamento para os Códigos 5.10 e 5.11	66
5.3	<i>Sobel</i> : (a) arquitetura original; (b) arquitetura melhorada	69
5.4	Escalonamento para o exemplo <i>ADPCM Coder</i>	70
5.5	Escalonamento para o exemplo <i>ADPCM Decoder</i>	73
6.1	Exemplo de ALP: (a) trecho de código; (b) estruturas de <i>hardware</i>	81
6.2	Fluxo de desenvolvimento com ALP	82

6.3	Diagrama das classes usadas para representação intermediária	84
6.4	Grafo com componentes fortemente conectados em destaque	86
6.5	Exemplo <i>ADPCM Coder</i> com diferentes escalonamentos	90
6.6	Caminhos desbalanceados no grafo	93
6.7	Algoritmos usados na compilação	94
6.8	Visualizações geradas pelo compilador ALP com auxílio do Graphviz	95
6.9	Interface gráfica do usuário	97
7.1	Número de ciclos de <i>clock</i> necessários para execução	103
7.2	Comparação dos recursos ocupados no FPGA Stratix III (EP3SE50F484C2) . .	106
7.3	Ganho no tempo de execução (<i>speedup</i>)	107
7.4	Tempo de execução normalizado em relação a LALP	108
7.5	Tempo de execução normalizado em relação a LALP	114
7.6	Comparação do <i>throughput</i> em relação ao ROCCC e C2Verilog	115
7.7	<i>Dotprod</i> : (a) arquitetura original; (b) arquitetura melhorada	118
7.8	Exploração do espaço de projeto para o exemplo <i>Dotprod</i>	119
7.9	Fronteira de Pareto considerando <i>Slices</i> e tempo de execução	120
7.10	Frequência máxima relativa por estágios no multiplicador	120
7.11	Consumo de potência dinâmico por frequência de operação	122

Lista de Tabelas

2.1	Memória interna dos FPGAs da família Stratix IV da Altera	16
2.2	Número de blocos DSP e máximo de operações implementáveis por tipo	16
2.3	Principais característica dos FPGAs Stratix IV e Virtex 6 (Assumpção Jr., 2010)	18
3.1	Construtores permitidos em ICAN	21
3.2	Somador em <i>pipeline</i> : 4 ciclos necessários para realizar a operação	34
3.3	Multiplicador em <i>pipeline</i> : 6 ciclos necessários para realizar a operação	34
3.4	Alocação de recursos para o Algoritmo 3.14	34
3.5	Alocação de recursos para o Algoritmo 3.15	35
3.6	Alocação de recursos para o Algoritmo 3.16	36
3.7	Alocação de recursos para o Algoritmo 3.17	37
3.8	Tabela comparativa dos algoritmos existentes (Allan <i>et. al.</i> , 1995)	41
4.1	Comparativo dos projetos encontrados na literatura	55
5.1	Genéricos e portas do componente contador	62
7.1	Lista dos <i>benchmarks</i> por ferramenta	101
7.2	Características dos <i>benchmark</i> usados	102
7.3	Frequência e recursos no FPGA Stratix (EP1S10F780C6)	104
7.4	Frequência e recursos no FPGA Stratix III (EP3SE50F484C2)	105
7.5	Frequência e recursos no FPGA Virtex 5 (XC5VLX303FF324)	107
7.6	Diretivas de sincronização	110
7.7	Frequência e recursos no FPGA Virtex 6 (XC6VLX75T3FF484)	111
7.8	LALP: frequência e recursos no FPGA Virtex 6 (XC6VLX75T3FF484)	112
7.9	ROCCC: frequência e recursos no FPGA Virtex 6 (XC6VLX75T3FF484)	112
7.10	C2Verilog: frequência e recursos no FPGA Virtex 6 (XC6VLX75T3FF484)	113
7.11	LALP comparado a microprocessadores embarcados	116
B.1	Lista dos <i>benchmarks</i> e respectivos códigos fonte	145

Lista de Algoritmos

3.1	Exemplo de um procedimento em ICAN (Muchnick, 1997)	20
3.2	Código de três endereços: código inicial	25
3.3	Código de três endereços: código modificado	25
3.4	Bloco básico de instruções	27
3.5	SSA: código inicial	28
3.6	SSA: código modificado	28
3.7	Loop unrolling: repetição inicial	31
3.8	Loop unrolling: repetição desenrolada em um fator 4	31
3.9	Expansão de variáveis: repetição inicial	31
3.10	Expansão de variáveis: repetição com variável expandida	32
3.11	Renomeação de registradores: código inicial	32
3.12	Renomeação de registradores: código modificado	32
3.13	Renomeação de registradores: escalonamento alternativo	32
3.14	Loop pipelining: código inicial	34
3.15	Loop pipelining: código modificado	35
3.16	Loop pipelining: código modificado novamente	35
3.17	Loop pipelining: código modificado com loop unrolling	36
4.1	Uma entrada válida para o compilador C-to-Verilog	53
5.1	Forma geral de um programa descrito em LALP	58
5.2	Declaração de constantes e tipos de dados	59
5.3	Declaração do programa com interfaces de entrada/saída	59
5.4	Declaração de variáveis escalares e arranjos	60
5.5	Repetições do exemplo <i>FDCT</i> descritas em C	61
5.6	Repetições do exemplo <i>FDCT</i> descritas em LALP	61
5.7	Operador ternário em LALP	61
5.8	Exemplo <i>Dotprod</i> descrito em C	63
5.9	Exemplo <i>Dotprod</i> descrito em LALP	63
5.10	Exemplo <i>Fibonacci</i> descrito em LALP	64

5.11	Exemplo <i>Fibonacci</i> descrito em LALP com reuso de dados	65
5.12	Exemplo <i>Sobel</i> descrito em LALP	67
5.13	Exemplo <i>Sobel</i> descrito em LALP com reuso de dados	68
5.14	Exemplo <i>ADPCM Coder</i> descrito em LALP	71
5.15	Exemplo <i>ADPCM Decoder</i> descrito em LALP	72
5.16	Forma geral de um programa descrito em LALP-S	75
5.17	Exemplo <i>Dotprod</i> descrito em LALP-S	76
5.18	Exemplo <i>Sobel</i> descrito em LALP com modularização	77
6.1	Exemplo de componente parametrizável da biblioteca VHDL	83
6.2	Exemplo <i>Dotprod</i> descrito diretamente no código Java	85
6.3	Computação dos componentes fortemente conectados (SCC)	87
6.4	Detecção de arestas recorrentes	88
6.5	Escalonamento ASAP modificado	89
6.6	Sincronização de contadores	91
6.7	Sincronização de operações	92
6.8	Balanceamento de arestas	94
7.1	Exemplo <i>Max</i> descrito em C	102
7.2	Exemplo <i>ADPCM Decoder</i> descrito em LALP com diretivas de sincronização .	109
7.3	Exemplo <i>Dotprod</i> modificado para melhor desempenho	117
B.1	Exemplo <i>ADPCM Coder</i> descrito em C	146
B.2	Exemplo <i>ADPCM Decoder</i> descrito em C	147
B.3	Exemplo <i>Autocorrelation</i> descrito em C	148
B.4	Exemplo <i>Bubble Sort</i> descrito em C	149
B.5	Exemplo <i>FDCT</i> descrito em C	149
B.6	Exemplo <i>Fibonacci</i> descrito em C	152
B.7	Exemplo <i>Fibonacci</i> descrito em C com reuso de dados	152
B.8	Exemplo <i>Pop Count</i> descrito em C	152
B.9	Exemplo <i>Sobel</i> descrito em C	152
B.10	Exemplo <i>Vector Sum</i> descrito em C	153
B.11	Exemplo <i>Autocorrelation</i> descrito em LALP	153
B.12	Exemplo <i>FDCT</i> descrito em LALP	154
B.13	Exemplo <i>Bubble Sort</i> descrito em LALP	157
B.14	Exemplo <i>Max</i> descrito em LALP	158
B.15	Exemplo <i>Pop Count</i> descrito em LALP	158
B.16	Exemplo <i>Vector Sum</i> descrito em LALP	161

Lista de Abreviaturas

ALAP	As L ate As P ossible
ALM	Adaptive L ogic M odule
ALP	Aggressive L oop P ipelining
ASAP	As S oon As P ossible
ASIC	Application Specific Integrated C ircuit
BRAM	B lock R andom Access M emory
C2H	C-to-H ardware Acceleration
CDFG	Control D ata F low G raph
CFG	Control F low G raph
CIRRF	Compiler Intermediate R epresentation for R econfigurable F abrics
CSoC	Configurable System- on-a-Chip
DDG	D ata D ependence G raph
DFA	D ata F low Analysis
DFG	D ata F low G raph
DFI	D ata- F low Intensive
DFS	D epth- F irst Search
DIL	D ataflow Intermediate L anguage
DMA	D irect M emory Access
DSP	D igital Signal P rocessing
EBNF	Extended B ackus- N aur F orm
EMS	Enhanced M odulo S cheduling
EPS	Enhanced P ipeline S cheduling
ESL	Electronic System L evel
FDCT	F ast D iscrete C osine T ransform
FPGA	F ield- P rogrammable G ate A rray
FSM	F inite State M achine
GCC	G NU Compiler C ollection
GNU	G NU's N ot U nix!

GPP	General Purpose Processor
HDL	Hardware Description Language
HLS	High-Level Synthesis
HTG	Hierarchical Task Graph
ICAN	Informal Compiler Algorithm Notation
ICMC	Instituto de Ciências Matemáticas e de Computação
ILP	Instruction-Level Parallelism
IMS	Iterative Modulo Schedule
JavaCC	Java Compiler Compiler
JPEG	Joint Photographic Experts Group
LAB	Logic Array Block
LALP	Language for Aggressive Loop Pipelining
LALP-S	Language for Aggressive Loop Pipelining Structural
MLAB	Memory Logic Array Block
PDF	Portable Document Format
PNG	Portable Network Graphics
RAM	Random Access Memory
RaPiD	Reconfigurable Pipelined Datapath
ROCCC	Riverside Optimizing Configurable Computing Compiler
RTL	Register Transfer Level
SoC	System-on-a-Chip
SoPC	System-on-a-Programmable-Chip
SRAM	Static Random Access Memory
SSA	Static Single Assignment
SCC	Strongly Connected Component
SUIF	Stanford University Intermediate Format
SVG	Scalable Vector Graphics
USP	Universidade de São Paulo
UTFPR	Universidade Tecnológica Federal do Paraná
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

Resumo

A computação reconfigurável tem se tornado cada vez mais importante em sistemas computacionais embarcados e de alto desempenho. Ela permite níveis de desempenho próximos aos obtidos com circuitos integrados de aplicação específica (ASIC), enquanto ainda mantém flexibilidade de projeto e implementação. No entanto, para programar eficientemente os dispositivos, é necessária experiência em desenvolvimento e domínio de linguagens de descrição de *hardware* (HDL), tais como VHDL ou Verilog. As técnicas empregadas na compilação em alto nível (por exemplo, a partir de programas em C) ainda possuem muitos pontos em aberto a serem resolvidos antes que se possa obter resultados eficientes.

Muitos esforços em se obter um mapeamento direto de algoritmos em *hardware* se concentram em *loops*, uma vez que eles representam as regiões computacionalmente mais intensivas de muitos programas. Uma técnica particularmente útil para isto é a de *loop pipelining*, a qual geralmente é adaptada de técnicas de *software pipelining*. A aplicação destas técnicas está fortemente relacionada ao escalonamento das instruções, o que frequentemente impede o uso otimizado dos recursos presentes nos FPGAs modernos.

Esta tese descreve uma abordagem alternativa para o mapeamento direto de *loops* descritos em uma linguagem de alto nível para FPGAs. Diferentemente de outras abordagens, esta técnica não é proveniente das técnicas de *software pipelining*. Nas arquiteturas obtidas o controle das operações é distribuído, tornando desnecessária uma máquina de estados finitos para controlar a ordem das operações, o que permitiu a obtenção de implementações eficientes. A especificação de um bloco de *hardware* é feita por meio de uma linguagem de domínio específico (LALP), especialmente concebida para suportar a aplicação das técnicas. Embora a sintaxe da linguagem lembre C, ela contém certas construções que permitem intervenções do programador para garantir ou relaxar dependências de dados, conforme necessário, e assim otimizar o desempenho do *hardware* gerado.

Abstract

RECONFIGURABLE computing is becoming increasingly important in embedded and high-performance computing systems. It allows performance levels close to the ones obtained with Application-Specific Integrated Circuits (ASIC), while still keeping design and implementation flexibility. However, to efficiently program devices, one needs the expertise of hardware developers in order to master hardware description languages (HDL) such as VHDL or Verilog. Attempts to furnish a high-level compilation flow (e.g., from C programs) still have to address open issues before broader efficient results can be obtained.

Many efforts trying to achieve a direct mapping of algorithms into hardware concentrate on loops since they represent the most computationally intensive regions of many application codes. A particularly useful technique for this purpose is loop pipelining, which is usually adapted from software pipelining techniques. The application of this technique is strongly related to instruction scheduling, which often prevents an optimized use of the resources present in modern FPGAs.

This thesis describes an alternative approach to direct mapping loops described in high-level languages onto FPGAs. Different from other approaches, this technique does not inherit from software pipelining techniques. The control is distributed over operations, thus a finite state machine is not necessary to control the order of operations, allowing efficient hardware implementations. The specification of a hardware block is done by means of LALP, a domain specific language specially designed to help the application of the techniques. While the language syntax resembles C, it contains certain constructs that allow programmer interventions to enforce or relax data dependences as needed, and so optimize the performance of the generated hardware.

Introdução

1.1 Contextualização

A computação há basicamente dois métodos primários para a execução de algoritmos. O primeiro deles é baseado em circuitos integrados ou combinações deles, construídos especificamente para executar a tarefa (ASIC¹) e, por esse motivo, a realizam com eficiência e rapidez. No entanto, o circuito é pouco flexível, não podendo ser alterado após sua fabricação. Mudanças em sistemas desse tipo resultam em altos custos na substituição dos circuitos. O segundo método é baseado em *software* executado em processadores de propósito geral (GPP²), constituindo uma solução muito mais flexível, pois é possível alterar as instruções do programa e, conseqüentemente, sua funcionalidade sem alterar o *hardware* do microprocessador. Apesar da flexibilidade, o método baseado em *software* apresenta uma grande desvantagem de desempenho, pois o processador precisa buscar cada instrução na memória, decodificá-la e depois executá-la (Compton e Hauck, 2002). A solução específica (*hardware*) realiza a computação de forma espacial, calculando muitas operações ao mesmo tempo em diferentes regiões do chip.

¹Application Specific Integrated Circuit

²General Purpose Processor

A computação reconfigurável pode ser considerada uma metodologia intermediária às duas anteriores. Os chips usados nessa tecnologia, dentre os quais os FPGAs³ são os mais difundidos (Chan e Mourad, 1994; Murgai *et. al.*, 1995; Oldfield e Dorf, 1995), permitem sua configuração após a fabricação. Dessa maneira, é possível construir soluções baseadas em *hardware* reconfigurável, oferecendo desempenho de *hardware* com flexibilidade de *software*. Os sistemas reconfiguráveis são projetados de maneira semelhante aos baseados em ASICs, por meio de projetos esquemáticos ou linguagens de descrição de *hardware* (HDL⁴) como VHDL⁵ e Verilog. A vantagem é que os projetos podem ser modificados a qualquer momento e o sistema pode ser reconfigurado para atender novas necessidades.

As melhorias contínuas em densidade e desempenho tornaram as arquiteturas baseadas em FPGA candidatas para construção de sistemas complexos. O uso de arquiteturas reconfiguráveis é visto como uma solução alternativa para atender a demanda de alto desempenho em vários sistemas de computação, em especial os sistemas embarcados. Mas, embora os FPGAs possibilitem a implementação de sistemas especializados, pesquisas em técnicas de compilação ainda são necessárias para permitir o mapeamento de regiões de *software* computacionalmente intensivas, tais como *loops*, sem a ajuda de especialistas em *hardware*.

A exploração automática de paralelismo é um objetivo fundamental em sistemas de computação não convencionais, voltadas para alto desempenho. Grandes esforços foram realizados no desenvolvimento de compiladores paralelizantes para linguagens convencionais, bem como para desenvolver linguagens especializadas que pudessem expor ao programador as características de paralelismo destas arquiteturas. Segundo Hauck (1998), para a computação reconfigurável ser usada com sucesso é necessário criar uma metodologia que possa mapear automaticamente sistemas descritos em linguagem de alto nível para *hardware*. Essa necessidade é justificada pelo fato de que os FPGAs modernos permitem a implementação de sistemas com um alto nível de complexidade, o que inviabiliza o projeto desses sistemas por métodos tradicionais de desenvolvimento de *hardware* em função do tempo necessário para sua realização. A maioria dos problemas de uso prático da computação tem sido implementado em *software*

³Field-Programmable Gate Array

⁴Hardware Description Language

⁵VHSIC (Very High Speed Integrated Circuits) Hardware Description Language

para execução em GPPs. Em uma tentativa de usar essa grande quantidade de algoritmos, foram desenvolvidas técnicas e ferramentas para a síntese de alto nível (Cardoso e Diniz, 2008; Densmore *et. al.*, 2006). Seu objetivo é a geração de blocos especializados de *hardware* ou de arquiteturas a partir de algoritmos descritos em linguagem de alto nível, como C.

Um dos fatores determinantes para a obtenção da melhor relação entre custo e desempenho em um sistema é a otimização de *loops*, por meio da técnica de *loop pipelining*. Usada com frequência em compiladores tradicionais, a técnica consiste basicamente reorganizar as instruções do *loop* para que se possa sobrepor instruções, de diferentes iterações, de forma a obter melhor proveito do paralelismo em nível de instruções (ILP⁶). A obtenção de uma solução ótima para o escalonamento é extremamente complexa, pois este deve ser realizado sem violar as dependências de dados e sem causar conflitos de recursos. Por esta razão, algumas abordagens desprezam as restrições de recursos para simplificar o problema, tornando sua aplicação propícia aos dispositivos reconfiguráveis onde os recursos são mais abundantes e diversificados.

1.2 Motivação

Considerando a compilação de linguagens de alto nível (por exemplo C e Java) para FPGAs, a maioria das abordagens para se realizar *loop pipelining* foram desenvolvidas a partir de técnicas consolidadas de *software pipelining* (Allan *et. al.*, 1995). Grande parte dos compiladores para arquiteturas reconfiguráveis, por exemplo, SPARK (Gupta *et. al.*, 2004b), Garp-C (Callahan, 2002; Callahan *et. al.*, 2000; Callahan e Wawrzynek, 2000) e MATCH (Banerjee *et. al.*, 2000; Haldar *et. al.*, 2000, 2001) usam versões do algoritmo IMS⁷ criado por Rau (1994), um dos mais eficientes para este propósito. Embora o uso deste algoritmo tenha algumas vantagens, ele se baseia no escalonamento estático das operações e na latência do caminho crítico do *loop* para criar um prólogo, um epílogo e um *kernel*, podendo resultar em arquiteturas não otimizadas. Algumas técnicas de vetorização foram também aplicadas aos FPGAs (Weinhardt e Luk, 2001), mas elas requerem repetições normalizadas e bem comportadas para atingir projetos com alto desempenho.

Recentemente, pesquisas foram dedicadas a abordagens dinâmicas para mapear *loops* in-

⁶Instruction-Level Parallelism

⁷Iterative Modulo Schedule

ternos, chamados *innermost loops* (Cardoso, 2005), e sequências de *loops* com dependências de dados entre eles (Rodrigues *et. al.*, 2007). Estas abordagens sugerem que os recursos em arquiteturas reconfiguráveis podem ser usados para aumentar o desempenho. Além disso, as experiências com o estado da arte nas ferramentas de síntese de alto nível mostraram que os resultados obtidos não são ideais e que há espaço para melhorias importantes, justificando a realização desta pesquisa. Por meio da análise de algumas ferramentas, foram constatadas fortes evidências de que as técnicas atuais de mapeamento não são capazes de explorar adequadamente os recursos dos FPGAs (Menotti *et. al.*, 2007). As técnicas baseadas em *software pipelining* estão fortemente ligadas ao escalonamento de recursos em arquiteturas do tipo von-Neumann, o que limita a obtenção de mapeamentos eficientes em dispositivos reconfiguráveis.

1.3 Objetivo

Nesta perspectiva, este trabalho de doutorado teve como objetivo propor técnicas inovadoras de mapeamento de *loops* em FPGAs, usando arquiteturas específicas que fornecessem melhorias em relação às técnicas existentes. A aplicação de técnicas extensivas para a obtenção de arquiteturas de alto desempenho visaram aproveitar efetivamente a sinergia entre os recursos disponíveis nos FPGAs atuais, explorando sua vasta gama de recursos (memórias *on-chip* reconfiguráveis, blocos de DSP⁸ etc.). Para atingir este objetivo, foi realizada a pesquisa e o desenvolvimento de uma linguagem de alto nível para a geração de *hardware* a partir de descrições comportamentais.

A linguagem desenvolvida, denominada LALP⁹ (Menotti *et. al.*, 2010a), teve como objetivo permitir a programação de aceleradores eficientes, usando *loop pipelining* agressivamente, para que os sistemas resultantes operassem com o melhor desempenho possível, buscando sempre o melhor aproveitamento dos recursos disponíveis. A linguagem, considerada de propósito específico, foi concebida para suportar operações com alto grau de paralelismo, mas oferecendo diretivas de sincronização de baixo nível capazes de orientar a geração do *pipeline*. Além disso, as descrições em LALP devem ser, em geral, muito próximas às aquelas equivalentes em C ou Java.

⁸Digital Signal Processing

⁹Language for Aggressive Loop Pipelining

1.4 Contribuições

A comparação com outras ferramentas mostrou que a abordagem adotada nesta pesquisa permite explorar o espaço de projeto de forma eficiente, oferecendo uma alternativa quando as técnicas tradicionais de síntese de alto nível não apresentam resultados satisfatórios. As arquiteturas obtidas com LALP proporcionaram *speedups* consideráveis, além de permitirem reduções do espaço ocupado no dispositivo.

Além gerar arquiteturas otimizadas em termos de desempenho e recursos ocupados, o compilador proporciona um ambiente favorável ao avanço da pesquisa em outros aspectos.

1.5 Organização

Para contextualizar a pesquisa, descrever as técnicas desenvolvidas e apresentar os resultados e conclusões, o presente documento está organizado como segue:

- No [Capítulo 2](#) é apresentada uma visão geral da computação reconfigurável, com enfoque nos dispositivos FPGA, que são comparados às soluções de *software* e *hardware* tradicionais. São apresentados os recursos existentes nos FPGAs atuais, e as possibilidades de uso desta tecnologia;
- No [Capítulo 3](#) é apresentada uma breve descrição do ciclo de compilação tradicional de *software* e as otimizações possíveis neste processo. São descritas as técnicas de *software pipelining*, em especial a de escalonamento módulo;
- No [Capítulo 4](#) é apresentada uma revisão bibliográfica dos projetos relacionados à síntese de alto nível de sistemas. São descritas as características de ferramentas pesquisadas e usadas para comparação dos resultados;
- No [Capítulo 5](#) é descrita a linguagem usada na criação das arquiteturas. São discutidos os aspectos da linguagem, suas limitações e apresentados exemplos para facilitar a compreensão;
- No [Capítulo 6](#) são descritas as técnicas desenvolvidas para a geração de *hardware* neste projeto, e implementadas no protótipo de um compilador. São apresentados a biblioteca

de componentes do compilador, a representação intermediária usada, as visualizações geradas para orientar o processo de desenvolvimento, bem como os algoritmos aplicados nas otimizações;

- No [Capítulo 7](#) são expostos os resultados obtidos com a linguagem e com as técnicas desenvolvidas. Os dados são comparados aos obtidos com outras ferramentas e com execuções em *software*.
- Finalmente, no [Capítulo 8](#) são apresentadas as conclusões deste trabalho e apontadas as limitações, além de sugestões para continuidade da mesma abordagem em trabalhos futuros.

Computação Reconfigurável

O objetivo deste capítulo é apresentar uma visão geral da computação reconfigurável, tecnologia usada neste trabalho para implementação das arquiteturas de *hardware* propostas. Na [Seção 2.1](#) são apresentados os conceitos básicos desta tecnologia. Na [Seção 2.2](#) é apresentado o fluxo de desenvolvimento usado na computação reconfigurável. Na [Seção 2.3](#) são apresentados os recursos disponíveis nos dispositivos atuais. Finalmente, na [Seção 2.4](#) é discutido o uso desta tecnologia em soluções *hardware/software*.

2.1 Conceitos Básicos

Segundo [Bobda \(2007\)](#) a computação reconfigurável pode ser definida como o estudo da computação envolvendo dispositivos reconfiguráveis, incluindo arquiteturas, algoritmos e aplicações. O objetivo da computação reconfigurável é preencher o espaço que há entre o *software* e o *hardware*, atingindo um desempenho muito maior que o da solução por *software*, enquanto mantém um nível de flexibilidade maior que o do *hardware*, como apresentado na [Figura 2.1](#).

O primeiro cenário, refere-se a um circuito integrado de aplicação específica (ASIC), desenvolvido especialmente para realizar determinada tarefa. Esta abordagem proporciona ótimo desempenho e nenhuma flexibilidade. O segundo cenário, refere-se ao uso de *software* execu-

Figura 2.1: Computação reconfigurável comparada às soluções de *hardware* e *software*

tando em um GPP. Esta abordagem é a mais flexível, pois é possível modificar a funcionalidade do sistema apenas ajustando o *software*. No entanto, as características destes processadores não permitem alto desempenho comparado ao *hardware* dedicado. A computação reconfigurável, apresentada no terceiro cenário, é capaz de atingir o desempenho oferecido pelo *hardware*, enquanto mantém a flexibilidade oferecida pelo *software*

Os FPGAs, introduzidos pela empresa Xilinx Inc. no ano de 1985 (Chan e Mourad, 1994), consistem em dispositivos lógicos programáveis que suportam a implementação de circuitos lógicos relativamente grandes (Brown e Vranesic, 2000) e são os principais dispositivos reconfiguráveis usados atualmente. Os recursos presentes nos FPGAs atuais permitem a construção de sistemas extremamente complexos em um único chip e têm permitido acelerar uma variedade de aplicações. Além disso, os dispositivos reconfiguráveis podem aproveitar melhor sua densidade uma vez que usam a mesma área do circuito integrado para realizar tarefas diferentes (DeHon, 2000).

Embora os FPGAs tenham suas vantagens quanto ao custo de engenharia e ao tempo de desenvolvimento quando comparado aos ASICs, projetos desenvolvidos com tecnologia VLSI¹, como processadores e memórias RAM² usadas nos PCs, apresentam maior velocidade, densidade e complexidade, mas precisam ser produzidos em um volume muito maior. Na Figura 2.2 é demonstrada a relação dos FPGAs com os ASICs e os projetos VLSI (Hamblen e Furman,

¹Very Large Scale Integration

²Random Access Memory

2001).

Figura 2.2: Relações de mercado de lógica digital (Hamblen e Furman, 2001)

Outro aspecto importante diz respeito a flexibilidade das soluções implementadas, sobre o qual os GPPs são os mais vantajosos. Os DSPs oferecem boa flexibilidade, mas são adotados para um classe específica de aplicações. Os dispositivos reconfiguráveis atingem alto grau de flexibilidade combinado ao bom desempenho, enquanto os circuitos integrados de aplicação específica oferecem pouca ou nenhuma flexibilidade. A relação entre flexibilidade e desempenho destes dispositivos é apresentada na Figura 2.3.

Dispositivos reconfiguráveis como os FPGAs são formados por um arranjo de células configuráveis, também chamado de bloco lógico, que pode ser usado para a implementação de funções lógicas. Um FPGA é composto, principalmente, por três tipos de recurso: blocos lógicos, blocos de entrada e saída e chaves de interconexão programáveis, conforme mostrados na Figura 2.4.

Os blocos lógicos formam um arranjo bidimensional e as chaves de interconexão são organizadas como canais de roteamento horizontais e verticais entre as linhas e as colunas de blocos lógicos. Cada um desses canais possui chaves também programáveis, que permitem conectar os blocos lógicos de maneira conveniente, segundo a necessidade de cada algoritmo (Compton, 1999).

A forma mais usada para se construir o bloco lógico é por meio de *lookup table* (LUT),

Figura 2.3: Relação entre flexibilidade e desempenho (Bobda, 2007)

Figura 2.4: Estrutura básica de um FPGA

que contém células de armazenamento usadas para implementar uma função lógica com poucas entradas e uma saída. Cada célula é capaz de armazenar um bit, que pode ser a saída da função dependendo da entrada, conforme [Figura 2.5\(a\)](#). É possível criar LUTs de vários tamanhos,

sendo o tamanho definido pela quantidade de entradas. Como a tabela verdade de uma função de duas variáveis tem quatro linhas, é possível implementar qualquer função de duas variáveis com uma LUT de quatro células. As variáveis de entrada da LUT são usadas como entradas de seleção de multiplexadores, determinando qual célula deve fornecer a saída do circuito. Na Figura 2.5(c) é apresentado o uso de uma LUT para implementar a função da Figura 2.5(b).

Figura 2.5: Uso de LUTs para implementação de funções lógicas

Na Figura 2.6 é mostrada uma LUT de três entradas, que possui oito células de armazenamento de acordo com a tabela verdade de uma função de três entradas. Os FPGAs comerciais normalmente possuem LUTs de quatro, cinco ou seis entradas e incluem elementos extra, como *flip-flops*, em seus blocos lógicos (Bout e E., 1999). Existem diversas tecnologias de programação para FPGAs, sendo as mais comuns baseadas em LUTs voláteis, carregadas por intermédio de *programmable read-only memorys* (PROMs) quando o circuito é ligado.

O crescimento constante do número de transistores por área, cuja previsão é apresentada na Figura 2.7, tem permitido a construção de sistemas embarcados cada vez mais complexos, denominados SoCs³. As principais características que diferenciam esses sistemas da computação tradicional são suas restrições de consumo de energia e baixo custo de produção, além das exigências de desempenho.

A computação reconfigurável, mais especificamente os FPGAs, têm sido usada com sucesso na construção de sistemas embarcados, pois oferecem um equilíbrio entre o desempenho e a flexibilidade (Compton e Hauck, 2002). Os FPGAs são dispositivos que podem proporcionar

³System-on-a-Chip

Figura 2.6: Circuito de uma LUT de três entradas

Figura 2.7: Aumento do número de transistores (Bondalapati e Prasanna, 2002)

ganhos significativos de desempenho nos sistemas embarcados se comparados aos sistemas baseados em microprocessadores tradicionais.

Os SoCs desenvolvidos com essa tecnologia são denominados SoPCs⁴ ou CSoCs⁵ e suas características de configuração apresentam também vantagens sobre os ASICs, especialmente para prototipação ou produção em baixa e média escala.

Na área de computação de alto desempenho (HPC) os FPGAs vêm propiciando um aumento na capacidade computacional superior ao obtido com microprocessadores, por permitirem a criação de arquiteturas massivamente paralelas e especializadas. Entre as aplicações que fazem uso desta tecnologia estão as de criptografia de dados (Elbirt *et. al.*, 2001, 2000), aplicações financeiras (Herbordt *et. al.*, 2007), computação científica e outras, inclusive as que necessitam de operações de ponto flutuante (Castillo *et. al.*, 2009; DuBois *et. al.*, 2009; Dubois *et. al.*, 2010; Lanzagorta *et. al.*, 2009; de Souza, 2008; Underwood, 2004; Zhuo e Prasanna, 2007).

2.2 Fluxo de Desenvolvimento

O fluxo de desenvolvimento tradicional para FPGAs é apresentado na Figura 2.8. Todos os passos podem ser realizados por uma única ferramenta, fornecida pela fabricante do dispositivo, ou podem ser usadas ferramentas específicas para cada parte do processo. Inicialmente, o circuito é especificado na forma de um diagrama esquemático ou por uma linguagem de descrição de *hardware* como VHDL e Verilog. Nesta fase, podem ser usados *cores* e *templates* uma vez que a especificação pode ser hierárquica tanto na forma de diagrama como nas linguagens.

Figura 2.8: Fluxo de desenvolvimento para FPGAs

⁴System-on-a-Programmable-Chip

⁵Configurable System-on-a-Chip

Durante o processo de síntese é verificada a consistência da especificação e após o seu término é possível realizar uma simulação comportamental do sistema. Por meio desta simulação é possível verificar se as funções do projeto foram implementadas corretamente. O processo de síntese pode ser dividido em etapas, sendo as primeiras independentes da tecnologia alvo e as últimas responsáveis por determinar quais e quantos elementos do dispositivo alvo serão usados para implementar o circuito.

A implementação propriamente dita consiste em posicionar os elementos e rotear as ligações entre eles, mapeando o circuito no dispositivo alvo. Após este processo, é possível realizar simulações mais precisas, capazes de determinar o desempenho do sistema, pois consideram propriedades físicas como o tempo de propagação do sinal elétrico no meio. O resultado final da implementação é um *bitstream* que descreve a configuração para o dispositivo alvo. O processo de configuração poder ser realizado na ferramenta por meio de um cabo para transmitir o *bitstream*. Outra possibilidade é a gravação do conteúdo em uma memória não volátil para posterior transferência no FPGA. A vantagem deste processo é a possibilidade de se embarcar o aparato de configuração no mesmo sistema.

2.3 Recursos dos FPGAs Atuais

Para exemplificar os tipos e a quantidade dos recursos presentes nos FPGAs atuais são descritas nesta seção algumas propriedades da família Stratix IV da Altera. Tal linha de dispositivos será usada por representar atualmente os dispositivos mais modernos e por possuir ampla documentação. Para uma referência completa consulte [AlteraURL \(2009\)](#).

O componente principal desta família de FPGAs é o LAB⁶, apresentado na [Figura 2.9](#), que pode ser configurado para executar funções lógicas e aritméticas e atuar como registrador. Cada LAB é formado por dez ALMs⁷ e cada ALM possui duas LUTs de seis entradas, dois somadores e dois *flip-flops*, além de multiplexadores e sinais de controle para ligações em cadeia. As interconexões locais transferem dados entre os ALMs do mesmo LAB e adjacentes e servem para aliviar as interconexões de linhas e colunas. Existe ainda uma variação do LAB, denomi-

⁶Logic Array Block

⁷Adaptive Logic Modules

nado MLAB⁸, que possui as mesmas funcionalidade mas é acrescido de 64 bits de memória em cada ALM e podem ser usados como memórias em configurações de 64x10 ou 32x20.

Figura 2.9: Estrutura dos LABs nos FPGA da família Stratix IV

Os dispositivos desta família possuem também grande quantidade de memória interna, organizados em diferentes tamanhos e que podem operar em até 600 MHz de frequência. Os MLABs são otimizados para implementar *shift-registers* e pequenas FIFOs. Os M9K são blocos de 9 Kbits e são ideais para memórias de propósito geral. Os M144K são blocos de 144 Kbits e são mais indicados para armazenamento de código e para *buffers* maiores como os de vídeo. Na [Tabela 2.1](#) é apresentada a disponibilidade de memórias de cada tipo por dispositivo.

Uma característica importante dos FPGAs atuais é a presença de blocos de DSP, usados para implementar algoritmos matematicamente intensivos e minimizar a alocação de elementos reconfiguráveis do dispositivo. A família Stratix IV possui blocos capazes de realizar operações de multiplicação, adição, subtração e deslocamento dinâmico. Na [Tabela 2.2](#) é apresentada

⁸Memory Logic Array Block

Tabela 2.1: Memória interna dos FPGAs da família Stratix IV da Altera

Dispositivo	MLABs	Blocos M9K	Blocos M144K	Dedicado♥	Total RAM◇
EP4SE230	4560	1235	22	14.283	17.133
EP4SE360	7072	1248	48	18.144	22.564
EP4SE530	10.624	1280	64	20.736	27.376
EP4SE820	16.261	1610	60	23.130	33.294
EP4SGX70	1452	462	16	6462	7370
EP4SGX110	2112	660	16	8244	9564
EP4SGX180	3515	950	20	11.430	13.627
EP4SGX230	4560	1235	22	14.283	17.133
EP4SGX290	5824	936	36	13.608	17.248
EP4SGX360	7072	1248	48	18.144	22.564
EP4SGX530	10.624	1280	64	20.736	27.376
EP4S40G2	4560	1235	22	14.283	17.133
EP4S40G5	10.624	1280	64	20.736	27.376
EP4S100G2	4560	1235	22	14.283	17.133
EP4S100G3	5824	936	36	13.608	17.248
EP4S100G4	7072	1248	48	18.144	22.564
EP4S100G5	10624	1280	64	20.736	27.376

♥ Total de memória dedicada em Kbits ◇ Total de memória incluindo MLABs em Kbits

a quantidade de blocos de cada dispositivo (coluna DSPs), bem como o número máximo de operações possíveis que podem ser implementadas com estes recursos.

Tabela 2.2: Número de blocos DSP e máximo de operações implementáveis por tipo

Família	Dispositivo	DSPs	Operações Independentes					♥	◇
			Mult. 9x9	Mult. 12x12	Mult. 18x18	Comp. 18x18	Mult. 36x36	MAC 18x36	MAC 18x18
Stratix IV E	EP4SE230	161	1288	966	644	322	322	644	1288
	EP4SE360	130	1040	780	520	260	260	520	1040
	EP4SE530	128	1024	768	512	256	256	512	1024
	EP4SE820	120	960	720	480	240	240	480	960
Stratix IV GX	EP4SGX70	48	384	288	192	96	6	192	384
	EP4SGX110	64	512	384	256	128	128	256	512
	EP4SGX180	115	920	690	460	230	230	460	920
	EP4SGX230	161	1288	966	644	322	322	644	1288
	EP4SGX290	104	832	624	416	208	208	416	832
	EP4SGX360	130	1040	780	520	260	260	520	1040
	EP4SGX360	128	1024	768	512	256	256	512	1024
	EP4SGX530	128	1024	768	512	256	256	512	1024
Stratix IV GT	EP4S40G2	161	1288	966	644	322	322	644	1288
	EP4S40G5	128	1024	768	512	256	256	512	1024
	EP4S100G2	161	1288	966	644	322	322	644	1288
	EP4S100G3	104	832	624	416	208	208	416	832
	EP4S100G4	128	1024	768	512	256	256	512	1024
	EP4S100G5	128	1024	768	512	256	256	512	1024

♥ Multiplicador/Somador de alta precisão ◇ Multiplicador/Somador

Na estrutura básica do bloco DSP é encontrado um par de multiplicadores 18x18, seguidos de uma unidade somadora/subtratora de 37 bits (primeiro estágio) que permite a realização de operações no formato $P[36..0] = A_0[17..0] \times B_0[17..0] \pm A_1[17..0] \times B_1[17..0]$. Após estes

componentes existem ainda registradores de *pipeline*, somadores (segundo estágio) e registradores de saída. Cada bloco de DSP possui quatro destas estruturas e é dividido em duas partes de igual funcionalidade. Cada parte pode ser combinada diretamente para realizar operações em diferentes formatos. Tal modularização permite aos blocos prover as seguintes funcionalidades:

- Suporte nativo para operações em 9, 12, 18 e 36 bits;
- Suporte nativo para multiplicação de números complexos em 18 bits;
- Implementação eficiente de operações de ponto flutuante de precisão simples (24 bits) e dupla (53 bits);
- Suporte a números com e sem sinal (em complemento de dois);
- Somadores, subtratores e acumuladores integrados aos multiplicadores;
- Saídas em cascata para propagar resultados de um bloco a outro sem o uso de lógica externa;
- Unidades de arredondamento e saturação;
- Capacidade de retroalimentação para suportar filtros adaptáveis.

A lista de recursos, protocolos suportados e tecnologias envolvidas nos FPGAs atuais é extensa e específica para cada modelo e fabricante. Muito detalhes de implementação são determinados pelas ferramentas de síntese sem que o projetista necessite especificar cada um individualmente. Outros podem ser escolhidos de forma global conforme o objetivo da síntese (menor área ou melhor desempenho, por exemplo). Além dos recursos mencionados, certos dispositivos possuem ainda *transceivers* e *hardware* dedicados para comunicação em diversos padrões da indústria. Na [Tabela 2.3](#) são apresentadas as principais características dos FPGAs Stratix IV e Virtex 6, fabricadas pelas empresas Altera e Xilinx, respectivamente.

2.4 Softcore Processors e Co-projeto

A densidade dos FPGAs atuais permitem o uso de processadores *softcore*, como o MicroBlaze da Xilinx ([XilinxURL, 2009](#)) e o Nios II da Altera ([AlteraURL, 2008b](#)). Existe, ainda, a

Tabela 2.3: Principais característica dos FPGAs Stratix IV e Virtex 6 (Assumpção Jr., 2010)

	Stratix IV	Virtex 6
Tecnologia	40nm	40nm
Células lógicas	72K a 803K	75K a 588K
Block RAMs	7M a 33M	6M a 33M
DSP	384 a 1024	288 a 864
Transceiver	Até 48	Até 48+24
E/S	372 a 920	380 a 720
Frequência	600MHz	600MHz

possibilidade de se introduzir um ou mais processadores à pastilha, como no caso de algumas famílias de FPGAs da Xilinx que possuem processadores *hardcore* PowerPC internamente.

Esse tipo de combinação permite que sistemas possam ser desenvolvidos de forma híbrida, com parte da aplicação em *software* e parte em *hardware*. Ao projeto de sistemas com componentes de *hardware* e *software* dá-se o nome de *codesign*. O particionamento *hardware/software* é uma etapa importante desse projeto (de Micheli e Sami, 1996; Wolf e Staunstrup, 1997). Assim, inúmeras aplicações já disponíveis em *software* podem ser aproveitadas e o *hardware* dedicado pode ser usado para melhorar as partes mais críticas em termos de desempenho.

Embora a combinação FPGA/processador seja propícia para o desenvolvimento de sistemas extremamente complexos, a síntese desses sistemas, a partir de descrições de alto nível, ainda possui muitos pontos em aberto, conforme descrito no Capítulo 4, sobre as ferramentas disponíveis para esse fim. O particionamento *hardware/software* e a geração de *hardware* para os trechos críticos em termos de desempenho requerem conhecimentos aprofundados das arquiteturas envolvidas e são difíceis de serem automatizadas. Apesar disso, compiladores de *hardware* podem ser usados no desenvolvimento de aceleradores separados para posterior integração ao sistema.

Técnicas de Compilação

NESTE capítulo são apresentados os conceitos encontrados na literatura ([Aho et. al., 1986](#); [Leupers e Marwedel, 2001](#); [Muchnick, 1997](#)) relativos aos sistemas de compilação e a divisão em compiladores *frontend* e *backend*. Na [Seção 3.1](#) é apresentada a notação adotada para representação dos algoritmos desenvolvidos. Na [Seção 3.2](#) é apresentado o fluxo básico de compilação de programas. Na [Seção 3.3](#) são apresentadas as características dos compiladores otimizantes. Na [Seção 3.4](#) são discutidas algumas representações intermediárias usadas nos compiladores. Na [Seção 3.5](#) são apresentadas as técnicas de otimização e as transformações realizadas no código dos programas para aplicação destas otimizações. Um enfoque maior é dado nas otimizações realizadas em código de baixo nível, as quais necessitam de informações dependentes de máquina, já que o trabalho está direcionado aos sistemas reconfiguráveis. Na [Seção 3.6](#) é apresentada a técnica principal adotada pelas ferramentas de geração de *hardware*, denominada *loop pipelining*. Por fim, na [Seção 3.7](#) são realizadas algumas considerações finais sobre as técnicas encontradas na literatura para este fim.

3.1 Notação Informal de Algoritmo para Compilador

Para demonstrar os algoritmos aplicados na otimização de programas será adotada uma notação denominada ICAN¹, proposta por Muchnick (1997). A notação é derivada de algumas linguagens de programação como C, Pascal e Modula-2, e possui extensões para representar conjuntos, tuplas, sequências, funções, arranjos e tipos específicos usados nos compiladores. O Algoritmo 3.1 é um exemplo de uma declaração global e um procedimento em ICAN.

Algoritmo 3.1: Exemplo de um procedimento em ICAN (Muchnick, 1997)

```

1  Struc : Node → set of Node
2
3  procedure Example_1(N, r)
4    N: in set of Node
5    r: in Node
6  begin
7    change := true: boolean
8    D, t: set of Node;
9    n, p: Node
10   Struc(r) := {r}
11   for each n ∈ N (n ≠ r) do
12     Struct(n) := N
13   od
14   while change do
15     change := false
16     for each n ∈ N - {r} do
17       t := N
18       for each p ∈ Pred[n] do
19         t ⊇ Struc(p)
20       od
21       D := {n} ∪ t
22       if D ≠ Struct(n) then
23         change := true; Struc(n) := D
24       fi
25     od
26   od
27 end    || Example_1

```

Na sintaxe usada pela ICAN, os comandos de bloco devem ser fechados (por exemplo, `if` com `fi` e `do` com `od`) e, por isso, não é necessário finalizar uma linha, a não ser que se use dois comandos em uma mesma linha (linha 23 do Algoritmo 3.1). Nesse caso, deve-se separá-los por `;` (ponto e vírgula). Comentário são iniciados por `||`.

Um programa em ICAN consiste em uma série de definições de tipo, seguida por uma série de declaração de variáveis, seguido por uma série de declaração de procedimentos e, opcio-

¹Informal Compiler Algorithm Notation

nalmente, por um programa principal. As variáveis podem ser de tipos simples (`boolean`, `integer`, `real` e `character`) ou construídos pelo comando `type`. Na [Tabela 3.1](#) são apresentados os construtores permitidos.

Tabela 3.1: Construtores permitidos em ICAN

Construtor	Nome	Exemplo de declaração
<code>enum</code>	Enumeração	<code>enum{left, right}</code>
<code>array of</code>	Arranjo	<code>array[1..10] of integer</code>
<code>set of</code>	Conjunto	<code>set of MIRInst</code>
<code>sequence of</code>	Sequência	<code>sequence of boolean</code>
<code>×</code>	Tupla	<code>integer × set of real</code>
<code>record</code>	Registro	<code>record {x:real, y:real}</code>
<code>U</code>	União	<code>integer U boolean</code>
<code>→</code>	Função	<code>integer → set of real</code>

Por sua simplicidade, a linguagem ICAN será usada para exemplificar as técnicas de otimização de *loops* existentes nos compiladores, descritas neste capítulo, e para apresentar os algoritmos desenvolvidos durante a pesquisa, descritos no [Capítulo 6](#). Para uma referência completa da notação consulte [Muchnick \(1997\)](#).

3.2 Fluxo Básico de Compilação

Um compilador consiste basicamente em um sistema de *software* capaz de transformar um programa escrito em uma linguagem de programação de alto nível em um programa equivalente em linguagem de máquina para ser executado em um computador. Também é considerado compilador um *software* que realiza outros tipos de transformações como traduzir um código fonte de um linguagem para outra ou um código objeto de uma arquitetura para outra.

Esse processo de transformação, consiste em uma série de fases que analisam um dado formato e o sintetizam em um novo, partindo de uma sequência de caracteres do código fonte e resultando em um código objeto para um computador na maioria dos casos. É possível que esse código objeto seja realocável e possa ser ligado com outros códigos antes de estar pronto para ser executado na memória. O processo de compilação, em geral, é composto pelos seguintes passos:

- **análise léxica:** analisa o texto apresentado e o divide em pequenos pedaços, chamados *tokens*, os quais podem ser membros válidos da linguagem em que o código foi escrito.

Essa fase pode resultar em um erro caso alguma parte do texto não possa ser convertida em um *token* válido;

- **análise sintática:** processa os *tokens* e gera uma representação intermediária sequencial ou em árvore, além de uma tabela de símbolos formada pelos identificadores usados no programa e seus atributos. Nesse momento podem ocorrer erros de sintaxe, caso alguma seqüência de *tokens* não seja reconhecida;
- **análise semântica:** processa a representação intermediária para verificar se o programa atende a semântica exigida pela linguagem de origem, como por exemplo, a verificação de todos os identificadores usados e suas respectivas declarações com os tipos compatíveis. Durante essa análise podem ocorrer erros, caso o programa não atenda os requisitos de semântica da linguagem usada;
- **geração de código:** transforma a representação intermediária em um código de máquina equivalente, o qual pode estar na forma de um módulo realocável ou diretamente em um programa executável.

Além dos quatro componentes básicos citados, um compilador inclui também uma tabela de símbolos e de acesso a rotinas e uma interface para o sistema operacional, apresentados na [Figura 3.1](#). Essa interface é usada para realizar a leitura e gravação de arquivos, para fazer a comunicação com o usuário e para facilitar a portabilidade de um compilador entre sistemas operacionais.

Embora se possa reunir algumas ou até mesmo as quatro fases do processo de compilação em um único passo, e isso pode ser necessário em alguns casos, a estrutura modular é vantajosa, pois favorece a construção de compiladores para linguagens e plataformas diferentes. As fases iniciais do processo, que recebem uma seqüência de caracteres e geram uma representação intermediária, são chamadas também de compilador *frontend*. As fases posteriores, que partem da representação intermediária e geram o código equivalente para a arquitetura alvo, são chamadas de compilador *backend*. A estrutura modular permite que se substitua o compilador *frontend* por outro que suporte uma linguagem de programação diferente. Da mesma maneira, é possível

manter o compilador *frontend* e substituir o compilador *backend* para se obter códigos objetos para outras arquiteturas.

Figura 3.1: Estrutura em alto nível de um compilador simples (Muchnick, 1997)

3.3 Compiladores Otimizantes

Os resultados obtidos por um compilador simples, efetuando sequencialmente e somente as transformações apresentadas na seção anterior, deixam muito a desejar. Isso ocorre porque ao gerar o código, expressão por expressão, sem considerar as instruções vizinhas, obtêm-se um programa sem nenhuma otimização e que pode ser melhorado na maioria dos casos.

As principais otimizações possíveis de se realizar durante o processo de compilação são as que atuam em repetições (doravante denominadas *loops*), alocação de registradores e escalonamento de instruções. Apesar disso, cada tipo de programa pode tirar mais ou menos proveito das otimizações dependendo de suas características.

Os compiladores otimizantes podem atuar em uma representação intermediária de nível médio, que não considera os detalhes específicos de cada arquitetura computacional ou em uma representação de baixo nível, com informações fortemente ligadas à arquitetura alvo. Em

qualquer um dos casos, o objetivo é analisar a representação do programa e transformá-la de maneira que ela realize a mesma tarefa de forma mais eficiente. As otimizações implementadas em um nível médio de abstração podem ser facilmente portadas de uma arquitetura para outra, enquanto as otimizações em baixo nível exploram melhor as características da máquina, como os modos de endereçamento suportados pela arquitetura.

Muchnick (1997) classifica as otimizações nas categorias a seguir:

- **A:** Otimizações tipicamente aplicadas ao código fonte ou alguma representação intermediária de alto nível que preserve a estrutura geral do programa (sequência das instruções, repetições, formas de acesso aos arranjos etc). Normalmente, são as primeiras otimizações a serem executadas, já que a tendência é baixar o nível das representações à medida que se avança no processo de compilação.
- **B,C:** Otimizações tipicamente aplicadas à representação intermediária de nível médio ou baixo.
- **D:** Otimizações realizadas em código de baixo nível que necessitem de informações dependentes de máquina.
- **E:** Otimizações realizadas em tempo de ligação² que operam no código objeto realocável.

Este trabalho concentrou-se na categoria de otimizações **D**, já que foi aplicado a computação reconfigurável, na qual se tem total domínio da arquitetura alvo e flexibilidade para modificá-la, se necessário.

3.4 Representações Intermediárias

A saída de um compilador *frontend* é uma representação intermediária do código fonte de entrada. O propósito dessa representação é prover uma estrutura de dados simples, na qual se possa aplicar as transformações necessárias e, posteriormente, gerar o código objeto para a arquitetura alvo. Alguns formatos de representações importantes são descritos a seguir.

²Em inglês: *linking*

3.4.1 Código de três endereços

A representação em código de três endereços fornece uma visão mais simplificada do programa comparada às representações usadas nas linguagens de alto nível. Todas as instruções do programa são convertidas em operações de dois operandos e um resultado, inserindo variáveis temporárias quando necessário. Considere o [Algoritmo 3.2](#).

Algoritmo 3.2: Código de três endereços: código inicial

```
1 x := a + b - c * d
```

A tradução em código de três endereços é feita com variáveis auxiliares, resultando no [Algoritmo 3.3](#)

Algoritmo 3.3: Código de três endereços: código modificado

```
1 t1 := a + b  
2 t2 := c * d  
3 x := t1 - t2
```

Essa representação é mais conveniente para analisar o fluxo dos dados e para realizar otimizações do que a representação usada pelas linguagens de alto nível. Para certos propósitos, é necessário manter estruturas de controle, como repetições e decisões, para realizar otimizações. O código de três endereços também pode ser importante para mapear estruturas diretamente para unidades funcionais da arquitetura alvo de dois operandos e um resultado. À medida que se transforma o código para esse formato, o processo de decomposição em primitivas suportadas pela arquitetura está praticamente realizado.

3.4.2 Grafo de fluxo de controle/dados

Analisar o fluxo de controle de um programa é essencial para realizar otimizações no código. Para representar melhor esse fluxo, podem ser criados blocos básicos de instruções que são executadas sempre sequencialmente. Em um bloco básico $B = (s_1, \dots, s_n)$, quando a instrução s_1 é executada, tem-se a certeza de que a instrução s_n será executada.

Para identificar os blocos básicos em um programa representado por código de três endereços é necessário localizar as instruções que podem alterar o fluxo do programa, como rótulos

(*labels*), instruções de desvio (*goto*), chamadas de procedimentos (*call*) e retorno (*return*).

O grafo de fluxo de controle (CFG³) é uma estrutura de dados que representa todas as possibilidades de fluxo de controle entre blocos básicos de um programa ou função. Para uma função F , o CFG é um grafo direcionado $G_F = (V, E)$, no qual cada nó $b_i \in V$ representa um bloco básico de F e cada aresta $e = (b_i, b_j) \in E \subseteq V \times V$ representa que o bloco b_j deve ser executado logo após b_i . Após a identificação dos blocos básicos, o CFG pode ser construído facilmente.

Em um CFG, os nós que possuem duas saídas representam blocos básicos que terminam com instruções de salto condicional, enquanto blocos com apenas uma saída terminam com instruções de *goto* para outro bloco ou simplesmente necessitam ser seguidos por outro bloco. Os nós que não possuem saídas devem terminar com uma instrução *return*. Caso o CFG não seja totalmente conectado, os blocos isolados podem ser eliminados sem prejudicar o comportamento do programa.

Outra análise importante para realizar otimizações é o fluxo dos dados de um programa e suas dependências. Para um bloco $B = (s_1, \dots, s_n)$ se diz que s_j é dependente de dados de s_i , com $i < j$, se s_i define um valor usado por s_j , que precisa ser executado depois de s_i no código de máquina. A análise do fluxo dos dados (DFA⁴) consiste em calcular essas dependências e é relativamente simples de ser realizada quando se considera somente as variáveis locais de uma função.

O resultado da DFA é o grafo de fluxo de dados (DFG⁵). O DFG para um bloco básico B é um grafo direcionado e acíclico $G_B = (V, E)$, no qual cada nó $n \in V$ representa uma entrada primária, uma operação ou uma saída. Uma aresta $e = (op_i, op_j) \in E \subset V \times V$ representa que o valor definido por op_i é usado por op_j . Essa dependência pode ocorrer das seguintes maneiras:

- **Dependência de fluxo (*read-after-write*):** op_j lê o resultado escrito por op_i ;
- **Antidependência (*write-after-read*):** op_j escreve em uma variável após ela ter sido lida por op_i ;
- **Dependência de saída (*write-after-write*):** op_j escreve a mesma variável escrita por op_i ;

³Control Flow Graph

⁴Data Flow Analysis

⁵Data Flow Graph

- **Dependência de entrada (*read-after-read*):** op_j lê a mesma variável lida por op_i .

Na [Algoritmo 3.4](#) é apresentado um bloco básico de instruções, cujo DFG é representado na [Figura 3.2](#).

Algoritmo 3.4: Bloco básico de instruções

```

1  t1 := a + b
2  t2 := c * d
3  x  := t1 - t2

```

Figura 3.2: DFG representando um bloco básico

É possível combinar CFGs e DFGs de maneira que cada nó do CFG seja representado por um DFG, dando origem ao grafo de fluxo de dados e controle (CDFG⁶). O uso do CDFG não está limitada à representação intermediária e pode ser empregado no compilador *backend* com os nós do DFG representando instruções de máquina.

O formato SSA

O formato mais usado para realizar análises e otimizações de dependência de dados é o SSA⁷ ([Alpern et. al., 1988](#)), que representa o programa de forma que cada variável seja atribuída uma única vez em todo o código. Esse formato pode ser construído a partir da representação de três endereços ou a partir do código original. Como a atribuição única não acontece na maioria dos

⁶Control/Data Flow Graph

⁷Static Single Assignment

programas, o código deve ser modificado toda vez que uma variável é alterada, criando-se novas versões para a mesma variável de forma que esta seja a única atribuição. Variáveis usadas do lado direito das expressões também são modificadas de forma a obter a versão mais recente da variável original.

No [Algoritmo 3.5](#) é apresentada uma situação na qual pode-se observar claramente que a primeira atribuição não é necessária. A mudança para o formato SSA apresentada no [Algoritmo 3.6](#) facilita essa identificação por parte do compilador e favorece as otimizações posteriormente.

Algoritmo 3.5: SSA: código inicial

```
1 a := 1
2 a := 2
3 b := a
```

Algoritmo 3.6: SSA: código modificado

```
1 a1 := 1
2 a2 := 2
3 b := a2
```

Em algumas situações é impossível saber em tempo de compilação qual versão da variável é a mais atualizada devido aos desvios que podem ser tomados em tempo de execução. Nesses casos, é necessário criar uma nova versão para esta variável e realizar a atribuição baseada na decisão que foi tomada anteriormente. A implementação detalhada para se criar a representação em SSA de forma eficiente, denominada *dominance frontiers*, é descrita por [Cytron et. al. \(1991\)](#).

3.5 Técnicas de Otimização

Esta seção apresenta técnicas empregadas na otimização do código, dentre elas as de escalonamento de instruções e as de transformações que podem ser realizadas durante o processo para se obter um melhor desempenho. Os métodos descritos podem ser aplicados ao escalonamento de blocos básicos, de desvios ou entre blocos. As principais transformações possíveis são o desenrolamento de repetições (*loop unrolling*), a expansão de variáveis (*variable expansion*) e

a renomeação de registradores (*register renaming*). O objetivo é reorganizar as instruções de um programa de forma a obter melhor proveito do ILP, explorando a capacidade de algumas arquiteturas de se usar unidades de processamento simultaneamente. Ao modificar a ordem das instruções, é necessário observar possíveis problemas (*hazards*) que podem ser de dependência dos dados, estruturais ou de salto. Algumas arquiteturas possuem mecanismos de *interlock* para evitar tais problemas, outras deixam essa tarefa a cargo do compilador.

O escalonamento de blocos básicos consiste na busca do melhor arranjo entre as instruções do bloco de modo a obter o menor tempo de execução com o mesmo resultado do bloco inicial. O escalonamento de saltos pode se referir a duas coisas: preencher o espaço dos atrasos que existem após um desvio com instruções úteis; ou cobrir o atraso entre realizar uma comparação e estar pronto para realizar o desvio baseado em seu salto. Alguns programas possuem blocos básicos muito pequenos e, nesses casos, o escalonamento consegue pouca ou nenhuma melhora no desempenho. Nesses casos, é interessante deixar os blocos básicos maiores ou estender o escalonamento aos blocos vizinhos, realizando escalonamento entre blocos.

As técnicas de escalonamento tratam na maioria dos casos de cobrir o atraso entre buscar um dado em uma cache e obter o valor disponível no registrador. Estas não levam em consideração a possibilidade do dado não estar na cache, o que causa um atraso considerável e imprevisível. A interação entre a alocação de registradores e o escalonamento de instruções é um problema complexo. Para resolver esse problema, alguns compiladores realizam a alocação de registradores simbólicos, realizam o escalonamento e, posteriormente, alocam os registradores físicos.

3.5.1 Análise de dependência dos dados

As informações de dependência dos dados obtidas pelos compiladores otimizantes são essenciais para produzir códigos com alto grau de ILP e que conservem as características do código original, ou seja, que gerem resultados exatos. Os testes de dependência dos dados são importantes para determinar quais transformações podem ser realizadas no programa sem que ele deixe de gerar resultados corretos e são a chave para paralelização de repetições em programas.

De modo geral, duas instruções de um programa são dependentes em termos de dados se as duas acessam a mesma informação (posição de memória ou registrador) e pelo menos uma

delas escreve essa informação. Instruções que não são dependentes podem ser executadas em qualquer ordem e, portanto, podem ser paralelizadas.

Análise de dependência dos dados em arranjos

Para variáveis escalares, as análises tradicionais de fluxo dos dados (Aho *et. al.*, 1986) podem determinar as relações de dependência. Já no caso dos arranjos é necessário observar o índice das variáveis, o que torna o problema muito mais complexo (Banerjee, 1988).

Instruções que fazem parte de uma repetição são executadas várias vezes e a dependência de dados pode ocorrer de uma iteração para outra qualquer. Uma dependência de dados que ocorre entre iterações diferentes de uma determinada repetição é denominada *loop carried dependence*. Uma repetição que não possui esse tipo de dependência pode ser completamente desenrolada (*loop unrolling*) e paralelizada.

O problema de dependência dos dados não pode ser resolvido em tempo polinomial, mas é possível encontrar soluções aceitáveis para algumas instâncias do problema. Diversos testes de dependência de dados foram propostos e se diferenciam na relação entre exatidão e eficiência da solução (Psarris e Kyriakopoulos, 2004). Os algoritmos usados adotam uma política conservadora, ou seja, se determinada dependência não pode ser provada, então as instruções são consideradas dependentes. Isso garante que a análise resultante não possibilitará a construção de programas com erros posteriormente.

3.5.2 Transformações auxiliares

Para realizar o escalonamento de instruções de forma a obter um maior proveito das arquiteturas com ILP é necessário realizar algumas transformações no código. Essas transformações, isoladamente, não trazem nenhuma melhoria de desempenho, no entanto, elas podem tornar o código mais aproveitável para outras modificações. A seguir são relacionadas algumas transformações dessa natureza.

Loop unrolling

Em certos casos é possível desenrolar uma repetição para obter blocos básicos maiores e, portanto, com mais possibilidades de escalonamento. A ideia é substituir o corpo de uma repetição por réplicas (a quantidade de replicações determina o fator) deste mesmo corpo,

ajustando-se o controle da repetição. Considerando o exemplo do [Algoritmo 3.7](#), este pode ser desenrolado por um fator 4, resultando no [Algoritmo 3.8](#). Nesse exemplo, a repetição poderia ser desenrolada completamente, porém, é importante lembrar que essa transformação teria um impacto significativo no tamanho do código.

Algoritmo 3.7: Loop unrolling: repetição inicial

```
1 for i := 1 to 100 do  
2   a[i] := a[i] + b[i]  
3 od
```

Algoritmo 3.8: Loop unrolling: repetição desenrolada em um fator 4

```
1 for i := 1 by 4 to 100 do  
2   a[i] := a[i] + b[i]  
3   a[i+1] := a[i+1] + b[i+1]  
4   a[i+2] := a[i+2] + b[i+2]  
5   a[i+3] := a[i+3] + b[i+3]  
6 od
```

Expansão de variáveis

Ao se desenrolar uma repetição em um fator n , é possível criar n cópias de algumas variáveis usadas para minimizar a dependência entre as instruções e obter um maior grau de paralelismo no código. O [Algoritmo 3.9](#) apresenta uma repetição usada para acumular um vetor a .

Algoritmo 3.9: Expansão de variáveis: repetição inicial

```
1 soma := 0;  
2 for i := 1 to 100 do  
3   soma := soma + a[i]  
4 od
```

Nesse exemplo, a variável `soma` pode ser expandida e depois combinada ao final da repetição. O [Algoritmo 3.10](#) apresenta o programa resultante, o qual aumenta as possibilidades de paralelização em alguns casos.

Renomeação de registradores

A renomeação de registradores é uma técnica usada para aumentar a flexibilidade durante o escalonamento, diminuindo as dependências entre as instruções. No [Algoritmo 3.11](#), o regis-

Algoritmo 3.10: Expansão de variáveis: repetição com variável expandida

```
1 soma := 0;  
2 somal := 0;  
3 for i := 1 by 2 to 100 do  
4   soma := soma + a[i]  
5   somal := somal + a[i+1]  
6 od  
7 soma := soma + somal;
```

tradutor `r1` é usado em todas as instruções e, portanto, a ordem dessas não pode ser modificada.

Algoritmo 3.11: Renomeação de registradores: código inicial

```
1 r1 := r2 + 1.0  
2 r52 := r1  
3 r1 := r3 * 2.0  
4 r40 := r1
```

É possível modificar esse registrador sem alterar o resultado final, conforme apresentado no [Algoritmo 3.12](#).

Algoritmo 3.12: Renomeação de registradores: código modificado

```
1 r17 := r2 + 1.0  
2 r52 := r17  
3 r1 := r3 * 2.0  
4 r40 := r1
```

Após a mudança, é possível realizar um escalonamento alternativo para o conjunto de instruções, conforme apresentado no [Algoritmo 3.13](#).

Algoritmo 3.13: Renomeação de registradores: escalonamento alternativo

```
1 r17 := r2 + 1.0  
2 r1 := r3 * 2.0  
3 r52 := r17  
4 r40 := r1
```

Durante essa transformação, é necessário observar o tipo de dado dos registradores envolvidos e o fluxo dos dados para garantir que um valor usado posteriormente não seja alterado.

3.6 Loop Pipelining

As técnicas de *loop pipelining*, como optou-se por denominar neste texto, são também conhecidas na área de *hardware* e arquiteturas por *loop folding* (Gajski *et. al.*, 1992), e na área de compiladores por *software pipelining* (Allan *et. al.*, 1995; Charlesworth, 1981; Goossens *et. al.*, 1989; Patel e Davidson, 1976). Tais técnicas merecem atenção especial entre as outras técnicas de escalonamento, pois em geral o tempo de execução de repetições (*loops*) domina o tempo de execução dos programas.

A ideia básica do *loop pipelining* é sobrepor instruções de iterações diferentes sem violar as dependências de dados e sem causar conflitos de recursos, iniciando uma iteração antes que a anterior termine e aumentando, assim, o paralelismo do código como um todo.

Embora as instruções de uma repetição possam ser paralelizadas em um escalonamento local, mais paralelismo pode ser obtido se for considerado o escalonamento entre iterações. Para tal, considera-se que uma repetição AB^n , na qual n representa o número de iterações executadas, possa ser modificada para a forma $A\{BA\}^{n-1}B$. As operações contidas em A são chamadas prólogo, e são executadas uma única vez. Em seguida está o *kernel*, representado pelas operações BA e que são executadas repetidamente. Por último está o epílogo, representado nesse exemplo pelas operações contidas em B .

As tabelas de reserva de recursos (Rau, 1994) podem ser usadas para gerenciar conflitos durante o escalonamento das operações. Na Tabela 3.2 são apresentados os recursos usados por um somador que opera em *pipelining*. Os operandos são carregados simultaneamente em um ciclo, a operação propriamente dita é realizada em dois ciclos e é requerido um ciclo adicional para salvar o resultado. Na Tabela 3.3 são apresentados os recursos usados por um multiplicador que também opera em *pipelining*.

Considerando o Algoritmo 3.14, que consiste em realizar uma soma seguida de uma multiplicação em dois vetores distintos, a combinação das operações é apresentada na Tabela 3.4. Durante a execução dessa repetição, diversas unidades funcionais ficam subutilizadas, pois a multiplicação só pode iniciar após o término da soma devido à dependência direta de dados entre as operações.

Tabela 3.2: Somador em *pipeline*: 4 ciclos necessários para realizar a operação

Tempo	Fonte 0	Fonte 1	ULA		Multiplicador				Resultado
			Estágio 0	Estágio 1	Estágio 0	Estágio 1	Estágio 2	Estágio 3	
0	A	A							
1			A						
2				A					
3									A

Tabela 3.3: Multiplicador em *pipeline*: 6 ciclos necessários para realizar a operação

Tempo	Fonte 0	Fonte 1	ULA		Multiplicador				Resultado
			Estágio 0	Estágio 1	Estágio 0	Estágio 1	Estágio 2	Estágio 3	
0	M	M							
1					M				
2						M			
3							M		
4								M	
5									M

Algoritmo 3.14: Loop pipelining: código inicial

```

1  for i := 1 to MAX do
2    a[i] := a[i] + 2;
3    b[i] := a[i] * 3;
4  od

```

Tabela 3.4: Alocação de recursos para o Algoritmo 3.14

Tempo	Fonte 0	Fonte 1	ULA		Multiplicador				Resultado
			Estágio 0	Estágio 1	Estágio 0	Estágio 1	Estágio 2	Estágio 3	
0	Ai	Ai							
1			Ai						
2				Ai					
3									Ai
4	Mi	Mi							
5					Mi				
6						Mi			
7							Mi		
8								Mi	
9									Mi

Considerando a transformação apresentada no Algoritmo 3.15, na qual a soma da primeira

iteração (prólogo) e a multiplicação da última iteração (epílogo) são movidas para fora da repetição, é possível diminuir a quantidade de ciclos necessários para a execução do *kernel* da repetição. A nova alocação dos recursos é apresentada na [Tabela 3.5](#).

Algoritmo 3.15: Loop pipelining: código modificado

```

1 a[1] := a[1] + 2;
2 for i := 1 to MAX-1 do
3   a[i+1] := a[i+1] + 2;
4   b[i] := a[i] * 3;
5 od
6 b[MAX] := a[MAX] * 3;

```

Tabela 3.5: Alocação de recursos para o [Algoritmo 3.15](#)

Tempo	Fonte 0	Fonte 1	ULA		Multiplicador				Resultado
			Estágio 0	Estágio 1	Estágio 0	Estágio 1	Estágio 2	Estágio 3	
0	Ai+1	Ai+1							
1	Mi	Mi	Ai+1						
2				Ai+1	Mi				
3						Mi			Ai+1
4							Mi		
5								Mi	
6									Mi

Como a multiplicação necessita de mais ciclos para terminar, é possível iniciá-la antes da soma, reduzindo ainda mais a quantidade de ciclos necessários ao *kernel*. O [Algoritmo 3.16](#) apresenta essa modificação, cuja alocação de recursos é apresentada na [Tabela 3.6](#).

Algoritmo 3.16: Loop pipelining: código modificado novamente

```

1 a[1] := a[1] + 2;
2 for i := 1 to MAX-1 do
3   b[i] := a[i] * 3;
4   a[i+1] := a[i+1] + 2;
5 od
6 b[MAX] := a[MAX] * 3;

```

Em certos casos, a técnica é combinada com a de *loop unrolling* para melhorar as possibilidades de escalonamento. O ganho de desempenho em função do tempo é apresentado na [Figura 3.3](#).

Tabela 3.6: Alocação de recursos para o Algoritmo 3.16

Tempo	Fonte 0	Fonte 1	ULA		Multiplicador				Resultado
			Estágio 0	Estágio 1	Estágio 0	Estágio 1	Estágio 2	Estágio 3	
0	Mi	Mi							
1	Ai+1	Ai+1			Mi				
2			Ai+1			Mi			
3				Ai+1			Mi		
4								Mi	Ai+1
5									Mi

Figura 3.3: Ganho de desempenho obtido com *loop pipelining*

Para o exemplo mencionado anteriormente, é possível desenrolar a repetição em um fator de dois (conforme apresentado no Algoritmo 3.17) e escaloná-la de forma diferente. Na Tabela 3.7 é apresentada a alocação de recursos para essa configuração.

Algoritmo 3.17: Loop pipelining: código modificado com loop unrolling

```

1 a[1] := a[1] + 2;
2 for i := 1 to MAX-2 do
3   b[i] := a[i] * 3;
4   a[i+1] := a[i+1] + 2;
5   b[i+1] := a[i+1] * 3;
6   a[i+2] := a[i+2] + 2;
7 od
8 b[MAX] := a[MAX] * 3;
```

Considerando uma arquitetura com 3 unidades funcionais independentes, o escalonamento apresentado na Figura 3.4 pode apresentar um ganho em relação a repetição original, inclusive

Tabela 3.7: Alocação de recursos para o Algoritmo 3.17

Tempo	Fonte 0	Fonte 1	ULA		Multiplicador				Resultado
			Estágio 0	Estágio 1	Estágio 0	Estágio 1	Estágio 2	Estágio 3	
0	Ai+1	Ai+1							
1	Mi	Mi	Ai+1						
2	Ai+2	Ai+2		Ai+1	Mi				
3	NOP		Ai+2			Mi			Ai+1
4	Mi+1	Mi+1		Ai+2			Mi		
5					Mi+1			Mi	Ai+2
6						Mi+1			Mi
7							Mi+1		
8								Mi+1	
9									Mi+1

sem a necessidade de desenrolar o *loop*.

Figura 3.4: Exemplo de *loop pipelining*

O intervalo de iniciação (*II*) é a quantidade de ciclos de *clock* entre a primeira instrução de iterações subseqüentes do *kernel*. O tempo total de execução do *loop* é dado por: $\text{tamanho}(\text{prólogo}) + II \times \# \text{iterrações} + \text{tamanho}(\text{epílogo})$. O problema central está em determinar qual o melhor *kernel* para a execução, ou seja, o mínimo intervalo de iniciação (*MII*). A complexidade para a solução ótima é um problema da classe NP-completo (Lam, 1988).

Os algoritmos de *loop pipelining* existentes seguem, em geral, duas linhas: os de escalonamento módulo, proposto pela primeira vez por Rau (1994); e os de identificação de *kernel*, proposto pela primeira vez por Aiken e Nicolau (1988). Os algoritmos de escalonamento módulo procuram um *kernel* escalonável com base nas restrições de recursos e de dependências, e buscam melhorar o *kernel* reduzindo o *II*. Já os algoritmos de identificação de *kernel* desen-

rolam completamente a repetição e buscam por padrões de execução para construir um novo *kernel*.

3.6.1 O algoritmo IMS

Um dos algoritmos mais usados para a realização de *loop pipelining* são os do tipo escalonamento módulo. Proposto por Rau (1994)⁸ de uma forma iterativa, denominada IMS, possui inúmeras variações propostas posteriormente (Fernandes *et. al.*, 1999; Lam, 1988; Rong *et. al.*, 2005; Schreiber *et. al.*, 2002; Snider, 2002; So e Dean, 2005; Stoodley e Lee, 1996). No escalonamento módulo as instruções são escalonadas de forma cíclica, de modo a evitar conflitos de recursos e dependências entre as iterações, formando assim um *pipeline* regular. Cada iteração de uma instrução é executada em um mesmo momento, determinado pelo resto (módulo) da divisão do tempo pelo *II* (daí o nome escalonamento módulo). Dessa maneira, cada iteração de uma repetição é iniciada em um intervalo fixo, de mesmo valor *II*.

Como não é possível determinar em tempo polinomial o *II* ideal, esse valor inicial é restrito basicamente pelo maior dos dois limites a seguir: o tempo mínimo de execução considerando os recursos disponíveis (*ResMII*); e o tempo mínimo de execução considerando as dependências entre as operações (*RecMII*). A partir desse valor, tenta-se realizar o escalonamento das instruções nesse intervalo. Caso não seja possível, o valor de *II* é incrementado e inicia-se uma nova tentativa até se obter sucesso. A principal característica desse método é que ao se escalonar uma determinada instrução, consideram-se os conflitos de recursos e as dependências em todas as iterações, e não somente na iteração escalonada.

A técnica de escalonamento módulo tem sido muito usada, inclusive com suporte de *hardware* para se reduzir a expansão do código. A única desvantagem dessa técnica é que ganhos fracionais só podem ser obtidos se a repetição for desenrolada uma ou mais vezes antes do escalonamento. Por exemplo, um código cujo *kernel* executa em três ciclos, pode ser desenrolado em um fator dois e se for escalonado em cinco ciclos, apresenta um ganho fracionário, pois cada iteração será realizada em 2,5 ciclos.

Diversos melhoramentos ao escalonamento módulo foram propostos por Lam (1988). Ela usou a expansão de variáveis, denominada *modulo variable expansion*, para eliminar as depen-

⁸Segundo Allan *et. al.* (1995), as ideias não foram publicadas anteriormente por considerações proprietárias.

dências de determinadas variáveis entre iterações. Essa técnica pode ser assistida por *hardware* na forma de rotação de registradores.

Mas a principal melhoria apresentada foi a possibilidade de tratar repetições que possuem desvios condicionais. O modelo proposto permite que um conjunto de instruções, que englobam o desvio, sejam representadas no grafo como um único nó. As dependências e recursos alocados por esse nó são calculados por meio da soma de todas as dependências e recursos usados por cada instrução.

Como os caminhos de um desvio podem possuir tamanhos diferentes, é necessário introduzir atrasos (NOPs) para compensar a diferença. Outra desvantagem é que, como instruções são escalonadas em conjunto, o *II* tende a aumentar mais do que o necessário para acomodar as instruções.

3.6.2 Identificação de *kernel*

Os algoritmos de identificação de *kernel*, também denominados *unroll-and-compact*, consistem em desenrolar diversas cópias de uma repetição e buscar um padrão de repetição onde as instruções iniciem em um mesmo tempo. Se o padrão encontrado executa em poucos ciclos de *clock*, este é selecionado e um *kernel* é construído com base neste padrão. As instruções vizinhas são usadas para formar o prólogo e o epílogo da repetição. A [Figura 3.5](#) apresenta o escalonamento de um conjunto de instruções de uma determinada repetição que vão de *A* até *I*.

Uma das técnicas usadas para realizar o reconhecimento de *kernel* é a de *window scheduling*. Dado um DDG⁹, que representa o corpo de uma repetição ([Figura 3.6\(a\)](#)), é criado um novo DDG com duas cópias do primeiro conectadas contendo uma janela (*window*) que engloba as instruções necessárias para uma iteração completa. Essa janela é deslocada de forma a buscar o melhor escalonamento para o *kernel*. As instruções anteriores e posteriores à janela representam o prólogo e o epílogo respectivamente, conforme apresentado na [Figura 3.6\(b\)](#).

[Hwang et. al. \(1991, 1993\)](#) propõem um escalonamento baseado nesse método. Inicialmente, é estimada uma latência mínima para o *kernel* de forma semelhante ao escalonamento módulo, baseada nos recursos disponíveis e na dependência das operações. Após isso, é realizado um escalonamento em duas etapas:

⁹Data Dependence Graph

Tempo	Iterações				
	1	2	3	4	...
1	A C F				
2	B E				
3					
4					
5	D	A C F			
6		B E			
7					
8	G				
9	H	D	A C F		
10	I		B E		
11					
12		G			
13		H	D	A C F	
14		I		B E	
15					
16			G		
17			H	D	
18			I		
19					
20				G	
21				H	
22				I	

Figura 3.5: Exemplo de *unroll-and-compact*

- *forward scheduling*: usa o esquema ASAP¹⁰ e dá prioridade as instruções de entrada de dados para diminuir o atraso entre as instruções, deixando a maior quantidade de recursos possíveis para a próxima instrução a ser escalonada.
- *backward scheduling*: modifica as instruções escalonadas para o esquema ALAP¹¹, deixando a maior quantidade de recursos possíveis para o escalonamento da próxima iteração.

Essas duas etapas são realizadas iterativamente, aumentando-se a latência a cada iteração, até que se obtenha um escalonamento válido.

A abordagem adotada por [Chao et. al. \(1997\)](#) é parecida, porém o escalonamento é realizado em grafos cíclicos ao invés de desenrolar o *kernel*. As instruções são rotadas no grafo e reescalonadas, buscando-se um intervalo menor.

¹⁰As Soon As Possible

¹¹As Late As Possible

Figura 3.6: Exemplo de *window scheduling*

3.7 Considerações Finais

A seguir são comparadas diversas técnicas de escalonamento para *loop pipelining*. Na [Tabela 3.8](#) são apresentadas características dos principais algoritmos.

Tabela 3.8: Tabela comparativa dos algoritmos existentes ([Allan et. al., 1995](#))

	Lam	Path	Pred	EMS	Perfect	Petri	Vegdahl	EPS
Ganho fracionário	não	não	não	não	sim	sim	sim	não
Restrições de recursos	sim	não	sim	sim	sim	sim	np	np
Complexidade	poli	poli	poli	2^n	poli	poli	2^n	poli
Expansão de código	sim	não	não	sim	sim	sim	sim	sim
Ganho limitado	sim	sim	sim	sim	não	sim	não	sim
Regular	sim	sim	sim	sim	não	não	–	não
Reconhecimento de <i>kernel</i>	não	não	não	não	sim	sim	sim	não
Módulo	sim	sim	sim	sim	não	não	não	não

A coluna **Lam** refere-se ao algoritmo proposto por [Lam \(1988\)](#), o qual realiza redução hierárquica. O algoritmo referente à coluna **Path** foi proposto por [Zaky \(1989\)](#) e formula o problema em termos matemáticos. A abordagem é interessante mas difícil de ser usada na prática, pois não considera os requisitos de recursos. A coluna **Pred** refere-se ao uso do escalonamento módulo explorando a capacidade de execução predicativa de algumas arquiteturas para acomodar os desvios. A coluna **EMS**¹² representa um algoritmo proposto por [Warter et. al. \(1992\)](#), no qual os desvios condicionais a serem escalonados são eliminados por meio de uma técnica denominada *if-conversion*.

A coluna **Perfect** refere-se ao algoritmo de [Aiken e Nicolau \(1988\)](#), no qual as instruções

¹²Enhanced Modulo Scheduling

são escalonadas sucessivamente até que se consiga uma uniformidade de padrões entre elas, obtendo assim o *kernel*. Aprimoramentos nesse algoritmo foram realizado por Aiken *et. al.* (1995), nos quais foi dada ênfase especial nas restrições de recursos. O algoritmo da coluna **Petri** (Allan *et. al.*, 1993) é similar ao anterior, porém usa a fundamentação e os algoritmos de redes de Petri para reconhecer o melhor *kernel*. A coluna **Vegdahl** representa um método exaustivo, proposto por Vegdahl (1982), no qual todas as possibilidades de escalonamento são criadas. Devido a sua complexidade, o método é impraticável em situações reais.

O algoritmo **EPS**¹³ (Ebcioğlu, 1987) não se enquadra nas duas categorias mais difundidas, pois mantém o *loop* original ao invés de buscar um novo *kernel*, porém movimenta instruções entre iterações diferentes. A principal limitação desse algoritmo é sua incapacidade em lidar com recursos persistentes, ou seja, aqueles que permanecem alocados por alguns ciclos após o início da instrução.

A linha **Ganho fracionário** diz respeito aos algoritmos capazes de obter *II* fracionais em relação à repetição original sem ter que replicá-la antes do escalonamento. Por meio de replicação (*loop unrolling*), todos os algoritmos são capazes de obter ganhos fracionais, mas são incapazes de decidir o melhor fator de replicação. **Restrições de recursos** são consideradas pela maioria dos algoritmos e **np** significa controle apenas dos recursos não persistentes.

Em termos de complexidade, os algoritmos comparados são equivalentes, exceto para o algoritmo EMS no seu pior caso e no algoritmo exaustivo de Vegdahl. A linha **Expansão do código** indica quais algoritmos podem aumentar o tamanho do código resultante em função das transformações criadas. É importante lembrar que o simples fato de se criar um prólogo e um epílogo para o *kernel* já resulta em um pequeno incremento no tamanho do código.

A linha **Ganho limitado** indica quais algoritmos usam um valor mais específico para o ganho obtido com as transformações. O escalonamento módulo, parte de um valor inicial para o *II* e incrementa esse valor a medida que não é possível obter todo esse ganho. Os escalonamentos baseados em reconhecimento de *kernel* não possuem um valor alvo e, portanto, resultam em um tempo de compilação maior. A linha **Regular** indica em quais algoritmos o prólogo e o epílogo são executados no mesmo intervalo do *kernel*, ou seja, são exatamente partes do *kernel*

¹³Enhanced Pipeline Scheduling

usadas para preencher o *pipeline* e para completar a execução, respectivamente.

Embora a solução ótima para o problema de *software pipelining* não possa ser obtida em tempo polinomial, muitos algoritmos são capazes de oferecer bons resultados com um nível de complexidade relativamente baixo. A restrição dos recursos é uma parte complexa dos algoritmos, sendo até desprezada por alguns deles, o que motivou o uso desses algoritmos em arquiteturas reconfiguráveis onde os recursos disponíveis são diferentes dos encontrados nas arquiteturas convencionais, nas quais os algoritmos foram aplicados inicialmente.

Trabalhos Relacionados

A síntese da alto nível (HLS¹) ou síntese ESL² consiste no uso de descrições com altos níveis de abstração para geração de *hardware* na forma RTL³ para posterior síntese de portas lógicas. As primeiras iniciativas nesta área de pesquisa começaram na década de 70, mas após um período de aparente declínio por volta de 2000 muitos trabalhos têm sido realizados na área acadêmica (Gupta e Brewer, 2008; Sangiovanni-Vincentelli, 2010). Os investimentos com ferramentas para este propósito na indústria de automação são apresentados na Figura 4.1.

Devido a complexidade dos sistemas atuais, a exploração adequada do espaço de projeto para obtenção de arquiteturas Pareto-ótimas (Pareto, 1909) só é possível com o uso de ferramentas automáticas de geração de hardware. No entanto, a adoção destas ferramentas ainda não se tornou uma prática comum. A baixa aceitação no mercado se deve a diversos fatores como o desejo dos projetistas de controlar em detalhes os projetos ou a ausência de uma linguagem universal para este fim (Coussy e Morawiec, 2008). Além do mais, a otimização dos circuitos em termos da potência dissipada tem se tornado importante e difícil de ser realizada

¹High-Level Synthesis

²Electronic System Level

³Register Transfer Level

Figura 4.1: Vendas de ferramentas para síntese de alto nível (Martin e Smith, 2009)

em todos os níveis de abstração (Cong e Rosenstiel, 2009).

Embora muitas ferramentas estejam disponíveis, tanto academicamente quanto comercialmente, pesquisas adicionais são fundamentais para possibilitar a exploração automática do espaço de projeto para diferentes requisitos e tecnologias (Cheng, 2009; Coussy *et. al.*, 2009). As iniciativas na área abordam o problema das mais variadas formas, com diferentes propósitos e aplicações. O objetivo pode ser a construção de sistemas completos, inclusive realizando o co-projeto de *software/hardware*, ou partes de sistemas como aceleradores e coprocessadores (Ahuja *et. al.*, 2009).

Atualmente, alguns dos maiores usos das ferramentas de síntese de alto nível são a prototipação e exploração de arquiteturas. Elas proporcionam uma maneira rápida para modelar a funcionalidade de diferentes arquiteturas e, a partir destas, estabelecer limites mínimos/máximos para área, desempenho e potência. Diferentes arquiteturas são modeladas, a melhor é escolhida e remodelada manualmente para se atingir um sistema eficiente (Sarkar *et. al.*, 2009).

Nesse capítulo são descritos alguns projetos na área de compiladores encontrados na literatura, mais especificamente os trabalhos voltados à síntese de alto nível para FPGAs comerciais. Apesar de haver muitos trabalhos na área, procurou-se abordar os projetos mais recentes e os de maior repercussão na comunidade científica em termos de publicações. As ferramentas descritas aqui se propõem a fazer, mesmo que parcialmente, a síntese de alto nível de sistemas. Na Seção 4.1 é discutida a necessidade do uso de ferramentas automáticas na geração de *hardware*.

Na [Seção 4.2](#) é apresentada a ferramenta C2H⁴ da Altera, usada para geração de aceleradores para o processador Nios II. Na [Seção 4.3](#) é apresentado o compilador SPARK, desenvolvido na Universidade da Califórnia San Diego. Na [Seção 4.4](#) é apresentado o compilador C-to-Verilog, que consiste em uma ferramenta *online* para compilação de programas em C para Verilog. Na [Seção 4.5](#) é apresentado o compilador ROCCC⁵, desenvolvido na Universidade da Califórnia Riverside. Na [Seção 4.6](#) é realizada uma breve análise comparativa das ferramentas abordadas.

4.1 Por que Ferramentas de Geração de *Hardware*?

Conforme apresentado no [Capítulo 2](#), os avanços tecnológicos na microeletrônica vêm permitindo a construção de muitos sistemas embarcados complexos em um único chip. Juntamente com o aumento da capacidade de processamento e comunicação desses sistemas, aumentam também a complexidade de seus projetos. Para acompanhar as mudanças nos sistemas que ocorrem rapidamente e atender a demanda por modificações em padrões e protocolos em tempo hábil é necessário realizar projetos em um curto espaço de tempo.

As ferramentas que permitem o projeto de sistemas em altos níveis de abstração ajudam a reduzir o tempo de projeto e diminuem a necessidade por projetistas com habilidades específicas de *hardware*. Além disso, existe uma infinidade de algoritmos já implementados em linguagens de alto nível que podem ser reaproveitados.

Na tentativa de facilitar a especificação de sistemas completos de *hardware* e *software* em uma única linguagem, foram propostas diversas novas linguagens, algumas delas extensões baseadas em C e C++. As linguagens SystemC, SystemVerilog e PSL/Sugar já possuem padrões estabelecidos pelo IEEE e são objetos de pesquisa em linguagens para projeto de sistemas.

Uma outra linha de pesquisa busca alternativas para realizar a síntese de alto nível a partir da descrição do sistema em linguagens de programação de *software* como C/C++ e Java, sem o uso de extensões ou bibliotecas próprias para *hardware*. Serão descritos nas seções seguintes alguns projetos e ferramentas para esse fim.

⁴C-to-Hardware Acceleration

⁵Riverside Optimizing Configurable Computing Compiler

4.2 C2H

O compilador C2H foi desenvolvido pela Altera para acelerar algoritmos desenvolvidos em C para o processador Nios II. A ferramenta não é destinada a construção de *hardware* a partir de uma descrição em linguagem C, mas sim à melhoria de desempenho dos programas executados no processador por meio de aceleradores construídos em *hardware*. Os módulos gerados pela ferramenta são automaticamente conectados ao barramento do processador, denominado Avalon. As porções do código devem ser escolhidas pelo usuário e devem ser isoladas em uma função separada no código fonte.

A Figura 4.2 apresenta como o acelerador gerado é integrado ao sistema por meio do barramento. O processador Nios II possui barramentos dedicados para dados e para instruções, modelo baseado na arquitetura de Harvard. O acelerador realiza operações de DMA⁶ para acessar uma ou mais memórias de dados, compartilhando dados com o processador.

Figura 4.2: Integração de um módulo gerado pelo C2H ao sistema (AlteraURL, 2008a)

⁶Direct Memory Access

Os passos realizados pela ferramenta são os seguintes:

- Pré-processa o código com o compilador GCC⁷;
- Cria um DDG;
- Realiza algumas otimizações;
- Determina a melhor sequência para a execução das operações;
- Gera um arquivo HDL para o módulo acelerador;
- Gera um *wrapper* para a função selecionada que será invocada no lugar da função original, ocultando do desenvolvedor os detalhes de como a função será executada efetivamente.

O módulo gerado pela ferramenta possui uma ou mais máquinas de estado que controlam a execução das operações, uma ou mais portas mestras que buscam e gravam os dados na memória e uma porta escrava mapeada para registradores, que são usados pelo processador para controlar o módulo. A documentação da ferramenta apresenta orientações ao desenvolvedor que permitem influenciar na geração do *hardware*, modificando-se as construções no *software* de origem.

As funções mais indicadas para serem transformadas em *hardware* são aquelas que possuem repetições, de um ou mais níveis, pequenas e simples, atuando em uma determinada quantidade de dados. Por outro lado, não são boas para transformação as funções que possuem muitas operações sequenciais, as que possuem chamadas às bibliotecas de funções do sistema operacional e as que possuem alguma sintaxe não suportada pela ferramenta. As construções não suportadas atualmente são as que possuem tipos de dados de ponto flutuante, funções recursivas, `structs` e `unions` que usam declarações *bit-field*, instruções `goto`, entre outras.

O processo de desenvolvimento usando a ferramenta envolve os seguintes passos:

1. Desenvolver e depurar a aplicação em C para o processador Nios II;

⁷GNU (GNU's Not Unix!) Compiler Collection

2. Obter o perfil do código (*profiling*) para determinar as áreas que podem ser beneficiadas com a ferramenta;
3. Isolar o código desejado em funções separadas e se possível também em arquivos separados;
4. Especificar as funções que deseja acelerar na ferramenta;
5. Reconstruir o projeto;
6. Obter novamente o perfil do código;
7. Se os resultados não atenderem os requisitos de desempenho, modificar o código C e a arquitetura do sistema;
8. Retornar ao passo 5.

A documentação dessa ferramenta não descreve as técnicas de otimização usadas, no entanto, o compilador pôde ser usado para comparar a eficiência das técnicas propostas neste projeto.

4.3 SPARK

O projeto SPARK (Gupta *et. al.*, 2004b) apresenta uma metodologia para a síntese de alto nível de sistemas digitais baseada na paralelização. O *framework* desenvolvido pelos autores do projeto recebe códigos em linguagem C, com algumas restrições, e gera códigos em VHDL para serem sintetizados.

A ferramenta apresenta algumas restrições quanto ao código de entrada usado, decorrentes da dificuldade de se sintetizar certas estruturas da linguagem C em *hardware*. Entre os recursos da linguagem que não são suportados estão o uso de ponteiros, a recursão de funções e os saltos irregulares (`goto`). Outras limitações são impostas simplesmente porque ainda não foram implementadas na ferramenta. Entre elas estão o uso de estruturas (`struct`), os *loops* que usam internamente os comandos `break` e `continue` e o suporte a arranjos multidimensionais.

O fluxo de compilação do SPARK é apresentado na Figura 4.3. Inicialmente, o código de entrada é transformado em uma representação intermediária que conserva toda sua estrutura.

Essa representação usa grafos hierárquicos de tarefas (HTG⁸), CFGs e DFGs formando uma estrutura de camadas que irá possibilitar a realização de otimizações no programas e a geração do código ao final do processo.

Figura 4.3: Fluxo da ferramenta desenvolvida no projeto SPARK (Gupta *et. al.*, 2004b)

As primeiras transformações a serem realizadas são aquelas ao nível de código fonte descritas na literatura (Aho *et. al.*, 1986; Muchnick, 1997), tais como eliminação de subexpressões, propagação de cópias, eliminação de código morto (*dead code*) etc. Além disso, são realizados os desenrolamentos das repetições possíveis (*loop unrolling*) para que se possa aplicar o escalonamento na maior parte do código.

A próxima fase é a do escalonamento, na qual são realizadas movimentações no código para aumentar o desempenho e tentar extrair o paralelismo inerente das aplicações. Para tal, algumas operações são reordenadas, movimentadas entre blocos básicos e até mesmo duplicadas entre estruturas condicionais durante o escalonamento. A ferramenta realiza um tipo de transformação de ciclos denominada *loop shifting*, para explorar o paralelismo entre repetições (Gupta

⁸Hierarchical Task Graph

et. al., 2004a). Os autores afirmam que o impacto das transformações no *hardware* resultante necessita de mais estudos e que é possível aplicar outros tipos de transformações para explorar melhor o paralelismo das aplicações.

As transformações realizadas no código aumentam significativamente a complexidade das conexões do *hardware* a ser gerado, tais como barramentos e multiplexadores. Para minimizar essa complexidade, as operações que possuem as mesmas entradas ou saídas são agrupadas em uma mesma unidade funcional. De maneira semelhante, as variáveis são agrupadas em um mesmo registrador quando representam entradas ou saídas de uma mesma unidade funcional.

Ao final do processo, é gerada uma máquina de estados finitos (FSM⁹) para controlar as partes do projeto. Este controlador irá disparar cada operação no tempo correto, gerando sinais de controle para tal. São gerados também os arquivos VHDL das unidades funcionais para serem sintetizados por ferramentas específicas de cada fabricante.

O projeto SPARK é uma referência importante para esse trabalho, pois a ferramenta desenvolvida é gratuita e pôde ser obtida no sítio dos autores. Apesar de não possuir código fonte aberto, foi possível realizar comparações de desempenho de sistemas e de arquiteturas geradas com a ferramenta.

4.4 C-to-Verilog

O C-to-Verilog (Ben-Asher e Rotem, 2008; C-to-VerilogURL, 2009) é uma ferramenta *on-line* para síntese de alto nível capaz de gerar código Verilog a partir de programas escritos em linguagem C. Apesar de mencionar claramente objetivos comerciais e não apresentar publicações científicas que detalhem as metodologias empregadas em sua operação, o compilador oferece uma boa alternativa de comparação, pois usa escalonamento módulo para a geração do *pipeline*. Os dados obtidos com esta ferramenta são apresentados nos gráficos e tabelas neste texto como C2Verilog ou simplesmente C2V.

Entre as limitações impostas ao código C para a compilação estão o uso de funções recursivas, estruturas, ponteiros para funções e chamadas à funções de biblioteca, tais como `printf` e `malloc`. É possível definir múltiplas funções no código desde que sejam declaradas com os

⁹Finite State Machine

modificadores `static inline`. O compilador suporta os tipos de dados inteiros e arranjos unidimensionais, que devem ser passados por parâmetro para a função principal. No [Algoritmo 4.1](#) é apresentado um programa que constitui uma entrada válida para esta ferramenta.

Algoritmo 4.1: Uma entrada válida para o compilador C-to-Verilog

```

1 unsigned int fib(unsigned int A[], unsigned int n) {
2     unsigned int last, prelast, curr;
3     prelast = 0;
4     last = 1;
5     unsigned int i;
6     for (i = 2; i < n; i++) {
7         curr = prelast + last;
8         prelast = last;
9         last = curr;
10        A[i] = curr;
11    }
12    return curr;
13 }
```

4.5 ROCCC

O projeto ROCCC ([Buyukkurt et. al., 2006](#); [Guo et. al., 2005](#)) tem como objetivo desenvolver um compilador para sistemas reconfiguráveis *on-chip* (CSoC) a partir de programas em linguagens de alto nível como C e FORTRAN. Uma visão geral da ferramenta é apresentada na [Figura 4.4](#), o compilador é baseado na plataforma SUIF¹⁰ ([SuifURL, 2008](#)).

A ferramenta desenvolvida analisa o perfil do código de entrada para identificar as partes executadas com mais frequência. O objetivo é transformar essas partes em código VHDL para serem executadas em *hardware*. As otimizações são realizadas em *loops* e aplicadas a uma representação intermediária gerada pelo compilador. A representação usada foi proposta pelos autores e foi denominada CIRRF¹¹. As restrições no código de entrada, incluem funções recursivas e o uso de ponteiros.

O compilador foi construído para otimizar aplicações de fluxo intenso de dados (DFI¹²). Portanto, opera melhor com aplicações sem muitos desvios no fluxo de controle. Para otimizar determinadas operações, seus dados são movidos de uma memória externa para um bloco de

¹⁰Stanford University Intermediate Format

¹¹Compiler Intermediate Representation for Reconfigurable Fabrics

¹²Data-Flow Intensive

Figura 4.4: Visão geral do compilador ROCCC (Guo *et. al.*, 2005)

memória RAM interno ao FPGA. Os circuitos gerados pelo compilador processam esses dados que são movidos para outro bloco de memória interno. São usados *buffers* e controladores para alimentar e coordenar as operações realizadas.

Em testes com alguns *benchmarks*, os autores compararam o *hardware* gerado pelo ROCCC com *cores* existentes no mercado. O *hardware* gerado consegue em geral operar na mesma frequência dos *cores*, mas ocupa de duas a três vezes mais área do FPGA. Cabe ressaltar que o uso da ferramenta também reduz o tempo de desenvolvimento.

4.6 Análise Comparativa

Na Tabela 4.1 é apresentada um comparativo dos projetos encontrados na literatura nos últimos anos, cujas técnicas e desempenho pudessem ser comparados a este trabalho.¹³

Os trabalhos descritos neste capítulo exploram de maneira superficial as técnicas de *loop pipelining*, principalmente por não usarem os recursos disponíveis nos FPGAs atuais. Em alguns

¹³Procurou-se abordar os projetos mais recentes e de maior repercussão na comunidade científica em termos de publicações, além de uma ferramenta comercial.

Tabela 4.1: Comparativo dos projetos encontrados na literatura

Projeto	Referência	Representação Intermediária	Loop Pipelining
SPARK	Gupta et. al. (2004b)	CFG, DFG e HTG	<i>Loop Shifting</i>
C-to-Verilog	C-to-VerilogURL (2009)	GCC	<i>Modulo Scheduling</i>
ROCCC	Buyukkurt et. al. (2006)	CIRRF	Não Divulgado
C2H	AlteraURL (2008a)	DDG	Não Divulgado◇

◇ por se tratar de uma aplicação comercial, a documentação não menciona a técnica usada

casos, os compiladores foram desenvolvidos para arquiteturas específicas, cuja comunicação entre o processador e as células reconfiguráveis era limitada.

Linguagens de domínio específico foram mais aplicadas a arquiteturas próprias. Como exemplo, pode-se citar a linguagem DIL¹⁴ para o PipeRench ([Goldstein et. al., 1999, 2000](#)) e a Rapid-C para o RaPiD¹⁵ ([Cronquist et. al., 1998](#); [Ebeling, 2002](#); [Ebeling et. al., 1996](#)). As exceções mais conhecidas são as linguagens Handel-C ([Handel-CURL, 2007](#)) e Haydn-C ([Coutinho e Luk, 2003](#); [Coutinho et. al., 2005](#)). Enquanto a Handel-C permite a descrição das operações ao nível de ciclos, a Haydn-C torna possível especificar explicitamente etapas de controle e operações para cada fase.

¹⁴Dataflow Intermediate Language

¹⁵Reconfigurable Pipelined Datapath

A Linguagem LALP

NESTE capítulo é descrita a linguagem desenvolvida, durante a execução do trabalho, para descrição de programas em alto nível (Menotti *et. al.*, 2009a,b, 2010c), denominada LALP. Para demonstrar as funcionalidades da linguagem serão usados alguns trechos de código isolados e exemplos de programas completos (*benchmarks*). Uma especificação formal da linguagem é apresentada no [Apêndice A](#).

As abordagens de desenvolvimento envolvendo síntese de alto nível prometem diminuir o esforço no processo, considerando a compilação automática de programas para os recursos do FPGA. Embora tenha havido muitas técnicas de otimização e diferentes abordagens para este fim, a deficiência dos compiladores em obter sistemas otimizados é evidente em muitos casos. Considerando estes aspectos, foi desenvolvida uma nova linguagem que oferece a possibilidade de interferir no escalonamento das operações em termos de ciclos de *clock* sem a necessidade de se usar linguagens de baixo nível como VHDL e Verilog. O objetivo foi oferecer uma alternativa no desenvolvimento de seções críticas de código quando as ferramentas de alto nível são incapazes de atingir os resultados desejados, seja em termos de recursos usados ou desempenho obtido. Na [Seção 5.1](#) é apresentada a especificação da linguagem LALP por meio de exemplos de uso. Na [Seção 5.2](#) são discutidas as limitações impostas pela linguagem. Na [Seção 5.3](#) é

apresentada a linguagem LALP-S¹, usada para descrever arquiteturas de forma estrutural. Finalmente, na [Seção 5.4](#) são discutidas algumas extensões possíveis para LALP e LALP-S.

5.1 Especificação da Linguagem

LALP pode ser considerada uma linguagem de propósito específico e foi concebida para suportar operações com alto grau de paralelismo, mas oferecendo diretivas de sincronização de baixo nível. Desta maneira, é possível explicitar, por exemplo, em quantos ciclos uma multiplicação deve ser concluída, ou ainda, quantos ciclos uma operação deve aguardar antes de ser iniciada. Além disso, o esforço usado para converter um programa escrito em C ou Java para LALP não deve ser maior do que o mesmo usado para modificar o *hardware* gerado por uma ferramenta capaz de interpretar diretamente estas linguagens. O compilador da linguagem foi desenvolvido com a ajuda do JavaCC² (Copeland, 2007; JavaCCURL, 2010; Kodaganallur, 2004), uma ferramenta para a criação de *parsers* e compiladores em Java.

No [Algoritmo 5.1](#) é apresentada a forma geral de um programa descrito em LALP. Inicialmente, são declaradas constantes e tipos de dados definidos pelo usuário. Em seguida, é descrita a interface da entidade a ser gerada, atribuindo-se um nome e os pinos de entrada e saída que esta irá possuir. Finalmente, são declaradas as variáveis em um bloco mais interno e, posteriormente, as instruções do programa são listadas.

Algoritmo 5.1: Forma geral de um programa descrito em LALP

```

1  declaração de constantes
2  declaração de tipos
3  entidade(pinos de entrada/saída) {
4      {
5          declaração de variáveis
6      }
7      instruções
8  }
```

A seguir são descritas as partes do programa em detalhes.

¹Language for Aggressive Loop Pipelining Structural

²Java Compiler Compiler

5.1.1 Constantes e tipos de dados

Em LALP é possível definir constantes que podem ser usadas tanto para declaração de tipos e variáveis escalares quanto para valores no código. No [Algoritmo 5.2](#) é apresentada a declaração de uma constante inteira (linha 1) e de dois tipos de dados definidos pelo usuário (linhas 2 e 3). O primeiro tipo é capaz de representar números inteiros de 32 bits com sinal e o segundo sem sinal.

Algoritmo 5.2: Declaração de constantes e tipos de dados

```
1 const DATA_WIDTH = 32;  
2 typedef fixed(DATA_WIDTH, 1) int;  
3 typedef fixed(DATA_WIDTH, 0) u_int;
```

5.1.2 Interfaces de entrada e saída

Após as declarações iniciais, deve ser descrito um nome para o programa que será usado na entidade VHDL correspondente. Opcionalmente, podem ser descritos parâmetros de entrada e saída que corresponderão à portas na entidade. O [Algoritmo 5.3](#) apresenta um programa com dois pinos de entrada e um pino de saída. Os parâmetros de entrada podem ser usados do lado direito das expressões do programa e os de saída do lado esquerdo.

Algoritmo 5.3: Declaração do programa com interfaces de entrada/saída

```
1 typedef fixed(16, 1) int;  
2 soma(in int a, in int b, out int c) {  
3     c = a + b;  
4 }
```

5.1.3 Variáveis e arranjos

A declaração de variáveis e arranjos unidimensionais pode ser realizada com tipos de dados definidos anteriormente ou descrevendo-se diretamente as propriedades por meio de parâmetros na cláusula `fixed`. No [Algoritmo 5.4](#) é apresentada a declaração de três variáveis distintas. Na linha 2 é declarada uma variável de um tipo já definido, e especificado um valor inicial que será atribuído ao registrador correspondente no momento da inicialização do circuito. Este

valor é especificado no código VHDL por meio de um genérico. Na linha 3 é declarada uma variável de 6 bits sem sinal e na linha 4 um vetor de N posições. Somente vetores unidimensionais são permitidos. Cada vetor declarado irá gerar uma memória correspondente e o código VHDL da memória irá conter os valores de inicialização, caso estes sejam especificados na declaração. Caso o programador deseje mapear vários arranjos em uma mesma memória deverá agrupá-los manualmente. O mapeamento de múltiplos arranjos em memória pode ser adicionado até mesmo por um pré-processamento do código em versões futuras.

Algoritmo 5.4: Declaração de variáveis escalares e arranjos

```

1  {
2    int a = 3;
3    fixed(6, 0) b;
4    int v[N] = {0, 1};
5  }
```

5.1.4 Controle de fluxo

Em LALP é possível descrever contadores aninhados em vários níveis ou em série, sem a necessidade de se usar blocos. No [Algoritmo 5.5](#) são apresentadas as repetições do algoritmo da FDCT³ descritas em linguagem C e no [Algoritmo 5.6](#) as mesmas repetições são descritas em LALP. A linha 16 indica que o contador k só será iniciado 17 ciclos após o contador i terminar, obtendo-se assim o mesmo efeito de seqüência. Note que em LALP é possível representar contadores concorrentes, o que é uma vantagem em certos casos como neste exemplo. O contador i_1 realiza o acesso a uma matriz que está armazenada em um vetor de maneira transposta. Nas linguagens com repetições de uma única variável esta estrutura teria que ser construída com uma variável independente do contador.

Na [Tabela 5.1](#) são listados parâmetros genéricos e as portas do componente contador, que é a chave para geração de arquiteturas usando o compilador ALP⁴. Assim como os demais componentes da biblioteca, o contador tem largura de bits variável. O parâmetro `step` é usado para modificar o número de ciclos por iteração em casos em que dependências no código impeçam

³Fast Discrete Cosine Transform

⁴Aggressive Loop Pipelining

Algoritmo 5.5: Repetições do exemplo *FDCT* descritas em C

```

1  i_1 = 0;
2  for (i = 0; i < num_fdcts; i++) {
3      for (j = 0; j < N; j++) {
4          f0 = dct_io_ptr[ 0+i_1 ];
5          f1 = dct_io_ptr[ 8+i_1 ];
6          ...
7          i_1++;
8      }
9      i_1 += 56;
10 }
11 i_1 = 0;
12 for (k = 0; k < N*num_fdcts; k++) {
13     ...
14 }

```

Algoritmo 5.6: Repetições do exemplo *FDCT* descritas em LALP

```

1  counter (i=0; i<num_fdcts; i+=64@72);
2  i.clk_en = init;
3  i_plus_8 = i + 8;
4  counter (j=i; j<i_plus_8; j++@9);
5  j.clk_en = init;
6  j.load = i.step;
7  j_plus_64 = j + 64;
8  counter (i_1=j; i_1<j_plus_64; i_1+=8);
9  i_1.clk_en = init@2;
10 i_1.load = j.step;
11 dct_io_ptr.address = i_1;
12 f0 = dct_io_ptr.data_out when (j.step@3);
13 f1 = dct_io_ptr.data_out when (j.step@4);
14 ...
15 counter (k=0; k<num_fdcts; k++);
16 k.clk_en = i.done@17;
17 ...

```

a execução de uma iteração por ciclo. O componente é capaz de contar para frente ou para trás com incremento também parametrizável. As condições de parada possíveis, na comparação do valor de output com `termination`, são `<`, `<=`, `>`, `>=`, `=` e `/=`.

As estruturas condicionais devem usar o operador ternário `? :`, presente também nas linguagens C e Java. A atribuição do Algoritmo 5.7 irá resultar em valores diferentes para `sign` caso a variável `diff` seja ou não menor do que zero.

Algoritmo 5.7: Operador ternário em LALP

```

1  sign = diff < 0 ? 8 : 0;

```

Tabela 5.1: Genéricos e portas do componente contador

	Nome	Função
Genéricos	bits	Largura das portas input, output e termination
	steps	Número de ciclos por iteração (padrão: 1)
	increment	Valor do incremento/decremento por iteração (padrão: 1)
	down	Direção da contagem (padrão: 0, incrementar)
	condition	Condição de parada (padrão: 1, <=)
Portas	input	Usada para carga do valor inicial
	termination	Valor para término da contagem
	clk	Clock do sistema
	clk_en	Habilita o funcionamento
	reset	Reinicia a contagem
	load	Realiza a carga do valor inicial
	step	Usado para sincronização das operações
	done	Indica o término da contagem
	output	Valor atual da contagem

Uma maneira alternativa de atribuir condicionalmente um valor a uma variável é usando a cláusula *when* na atribuição, que será realizada somente se a expressão que segue a cláusula for verdadeira. Esta funcionalidade da linguagem é muito usada para sincronizar as operações a partir do sinal *step* do contador. Nas iterações realizadas em mais de um ciclo de *clock* este sinal tem valor um no primeiro ciclo e zero nos demais. Na [Figura 5.1](#) é apresentada a simulação de um componente contador gerado a partir do código `counter (i=0; i<5; i++@3); .`

Figura 5.1: Simulação do componente contador

5.1.5 Exemplos de programas

Para demonstrar algumas funcionalidades da linguagem, será usado um exemplo simples que calcula a soma do produto de dois vetores. No [Algoritmo 5.8](#) é apresentado o código fonte deste exemplo para a linguagem C.

No [Algoritmo 5.9](#) é apresentada a descrição em LALP para geração de uma arquitetura capaz de calcular a mesma soma. Nas linhas 1 e 2 são declaradas constantes, usadas para o

Algoritmo 5.8: Exemplo *Dotprod* descrito em C

```

1 #define N 2048
2
3 int dotprod() {
4     int x[N], y[N];
5     int i, sum = 0;
6     for (i=0; i<N; i++)
7         sum += x[i] * y[i];
8     return sum;
9 }

```

número de bits de cada valor e para o número de iterações, respectivamente. Na linha 4 é definido um tipo de dado usado para 32 bits de ponto fixo com sinal e na linha seguinte um tipo de um único bit para sinais de controle. A linha 7 inicia com um nome que será usado na criação da entidade em VHDL, seguido de sinais de entrada e saída desta entidade. O bloco que vai da linha 8 até a linha 12 contém declarações de variáveis escalares e arranjos. Nota-se que a variável *i* é declarada com um tipo de dado não definido anteriormente.

Algoritmo 5.9: Exemplo *Dotprod* descrito em LALP

```

1 const DATA_WIDTH = 32;
2 const N = 2048;
3
4 typedef fixed(DATA_WIDTH, 1) int;
5 typedef fixed(1, 0) bit;
6
7 dotprod(out int sum, out bit done, in bit init) {
8     {
9         int x[N], y[N];
10        int acc;
11        fixed(16, 0) i;
12    }
13    counter (i=0; i<N; i++@1);
14    i.clk_en = init;
15    x.address = i;
16    y.address = i;
17    acc += x.data_out * y.data_out when i.step@1;
18    sum = acc;
19    done = i.done@2;
20 }

```

As instruções propriamente ditas iniciam com o contador na linha 13. A diretiva @1 indica que o componente irá gerar um novo valor para *i* a cada ciclo de *clock* e poderia ser omitida, pois este é o valor padrão. Em casos em que ocorrem dependências entre as iterações, um valor diferente pode ser necessário. Na linha 14 o sinal externo de inicialização *init* é usado para

habilitar a contagem. As linhas seguintes indicam que o endereçamento dos vetores será determinado pela variável `i`. Estas instruções podem ser facilmente substituídas por uma macro nas formas `x[i]` e `y[i]`. A linha 17 descreve as operações principais do código que devem aguardar um ciclo após o início da contagem para obtenção dos valores da memória. Finalmente, a linha 18 indica que o sinal `sum` irá externar a soma dos valores e o sinal `done` do contador também será apresentado como um pino de saída da entidade, indicando o término das operações dois ciclos após o contador terminar.

No [Algoritmo 5.10](#) é listado um outro exemplo de programa em LALP, usado para calcular e armazenar em uma memória os 32 primeiros números da sequência de Fibonacci. Cada variável escalar declarada no código deverá gerar um componente na arquitetura, dependendo das atribuições realizadas nesta variável. O compilador ALP assume que cada componente recebe valores de uma única origem. Caso haja mais de uma atribuição para a mesma variável, será necessário informar por meio da cláusula `when` o momento das atribuições seguintes para que seja gerado um componente multiplexador.

Algoritmo 5.10: Exemplo *Fibonacci* descrito em LALP

```

1  const DATA_WIDTH = 32;
2  const N = 32;
3
4  typedef fixed(DATA_WIDTH, 1) int;
5  typedef fixed(1, 0) bit;
6
7  fibonacci_alp(in bit init, out int output, out bit done) {
8      {
9          fixed(6, 0) i, v_addr;
10         int v[N] = {0, 1};
11         int a, b;
12     }
13     counter (i=2; i<N; i++@3);
14     i.clk_en = init;
15     i.load = !init@1;
16     v.address = v_addr;
17     v_addr = i - 2;
18     v_addr = i - 1 when i.step@1;
19     v_addr = i when i.step@2;
20     a = v.data_out when i.step@1;
21     b = v.data_out when i.step@2;
22     v.data_in = a + b when i.step@3;
23     output = v;
24     done = i.done@3;
25 }
```

Para este exemplo cada iteração é realizada em três ciclos de *clock*, dois para a leitura dos valores e um para a escrita. A diretiva @3 indica que o contador *i* irá produzir um novo valor a cada três ciclos. Caso um contador tenha valor inicial diferente de zero ou controlado por outra variável, é necessário usar o sinal *load* para indicar o momento da inicialização (linha 15). Nas linhas 17 a 19 são atribuídos valores diferentes à variável *v_addr* que será transformada em um multiplexador para controlar o endereçamento do vetor *v*. As diretivas *when* indicam o momento em que cada valor deve ser atribuído.

Esta versão do algoritmo não faz reuso de dados, pois faz a leitura na memória dos valores anteriores da sequência para calcular os novos valores, consumindo portanto três ciclos a cada iteração. Uma versão otimizada do mesmo algoritmo é apresentada no [Algoritmo 5.11](#), na qual os valores anteriores são mantidos em registradores e apenas os novos valores são gravados na memória.

Algoritmo 5.11: Exemplo *Fibonacci* descrito em LALP com reuso de dados

```

1  const DATA_WIDTH = 32;
2  const N = 32;
3
4  typedef fixed(DATA_WIDTH, 1) int;
5  typedef fixed(1, 0) bit;
6
7  fibonacci_alt_alp(in bit init, out int output, out bit done) {
8      {
9          fixed(6, 1) i;
10         int v[N];
11         int a=0, b=1;
12     }
13     counter (i=0; i<N; i++);
14     i.clk_en = init;
15     v.address = i;
16     v.data_in = a;
17     b += a when i.step;
18     a = b when i.step;
19     output = v;
20     done = i.done@1;
21 }
```

O escalonamento resultante das duas versões é apresentado na [Figura 5.2](#). A primeira versão do código, cujo escalonamento é apresentado na [Figura 5.2\(a\)](#), resulta em uma solução ineficiente, na qual a memória é acessada três vezes para cada iteração. Na segunda versão, apresentada na [Figura 5.2\(b\)](#), as operações das linhas 16 a 18 do [Algoritmo 5.11](#) são executadas

Figura 5.2: Escalonamento para os Códigos 5.10 e 5.11

em paralelo, o que proporciona a conclusão dos cálculos em quase um terço dos ciclos de *clock*.

No Algoritmo 5.12 encontra-se a descrição do filtro *Sobel* em LALP, usado para detecção de contornos em imagens. Neste exemplo, são realizados oito acessos à memória, para cada iteração. Após o cálculo, o valor é gravado na memória de saída e os valores de entrada são desprezados. Embora as operações de leitura (linhas 19 a 35), processamento (linhas 36 a 42) e gravação (linha 43) sejam efetuadas em *pipeline* a arquitetura é ineficiente, pois despreza valores que serão usados logo a seguir.

No Algoritmo 5.13 é apresentada uma implementação alternativa, na qual os valores são lidos uma única vez na memória e armazenados em registradores auxiliares.

Enquanto a implementação inicial realiza oito acessos à memória para cada iteração e realiza o cálculo logo a seguir (Figura 5.3(a)), o código modificado necessita de vinte e três acessos antes de calcular a primeira iteração (Figura 5.3(b)), mas a partir daí, cada iteração necessita de apenas mais um acesso, já que os dados estão disponíveis para os cálculos subsequentes.

Neste caso em particular, onde a memória é usada para armazenar uma imagem, a largura

Algoritmo 5.12: Exemplo *Sobel* descrito em LALP

```

1  const DATA_WIDTH = 16;
2  const ROWS = 10;
3  const COLS = 10;
4  const SIZE = 100; // ROWS*COLS
5
6  typedef fixed(DATA_WIDTH, 1) int;
7  typedef fixed(1, 0) bit;
8
9  sobel_alt(in bit init, out bit done, out int result) {
10   {
11     int H, O, V, Hpos, Vpos, Otrunk, i, addr;
12     int i00, i01, i02;
13     int i10, i12;
14     int i20, i21, i22;
15     int input[SIZE];
16     int output[SIZE];
17   }
18   counter (i=0; i<78; i+=1@8);
19   addr = i;
20   addr = (i) + 1 when i.step@1;
21   addr = (i) + 2 when i.step@2;
22   addr = (i) + COLS when i.step@3;
23   addr = ((i) + COLS) + 2 when i.step@4;
24   addr = ((i) + COLS) + COLS when i.step@5;
25   addr = (((i) + COLS) + COLS) + 1 when i.step@6;
26   addr = (((i) + COLS) + COLS) + 2 when i.step@7;
27   input.address = addr;
28   i00 = input when i.step@2;
29   i01 = input when i.step@3;
30   i02 = input when i.step@4;
31   i10 = input when i.step@5;
32   i12 = input when i.step@6;
33   i20 = input when i.step@7;
34   i21 = input when i.step@8;
35   i22 = input when i.step@9;
36   H = ((-i00) + (-2 *@6 i01)) + (((-i02) + i20) + (2 *@6 i21 + i22));
37   V = ((-i00) + i02) + (((-2 *@6 i10) + 2 *@6 i12) + ((-i20) + i22));
38   Hpos = H < 0 ? -H : H;
39   Vpos = V < 0 ? -V : V;
40   O = Hpos + Vpos;
41   Otrunk = O;
42   Otrunk = 255 when O > 255;
43   output.data_in = Otrunk;
44   output.address = i;
45   result = output;
46 }

```

da imagem tem influência direta na número de registradores necessários para implementar a otimização. Nos casos em que a largura da entrada possa inviabilizar esta abordagem, seja pela complexidade do código ou pela quantidade de recursos necessários, seria possível ainda realizar o reuso em apenas uma dimensão. Para isso, seriam necessários três acessos à memória para

Algoritmo 5.13: Exemplo *Sobel* descrito em LALP com reúso de dados

```

1  const DATA_WIDTH = 16;
2  const ROWS = 10;
3  const COLS = 10;
4  const SIZE = 100; // ROWS*COLS
5
6  typedef fixed(DATA_WIDTH, 1) int;
7  typedef fixed(1, 0) bit;
8
9  sobel_alt(in bit init, out bit done, out int result) {
10 {
11     int H, O, V, Hpos, Vpos, Otrunk, i;
12     int i00, i01, i02, i03, i04, i05, i06, i07
13     int i08, i09, i10, i11, i12, i13, i14, i15
14     int i16, i17, i18, i19, i20, i21, i22;
15     int i00h, i01h, i02h, i20h, i21h, i22h;
16     int i00v, i10v, i20v, i02v, i12v, i22v;
17     int input[SIZE];
18     int output[SIZE];
19 }
20 counter (i=0; i<78; i++);
21 input = i; i22 = input; i21 = i22;
22 i20 = i21; i19 = i20; i18 = i19;
23 i17 = i18; i16 = i17; i15 = i16;
24 i14 = i15; i13 = i14; i12 = i13;
25 i11 = i12; i10 = i11; i09 = i10;
26 i08 = i09; i07 = i08; i06 = i07;
27 i05 = i06; i04 = i05; i03 = i04;
28 i02 = i03; i01 = i02; i00 = i01;
29 // Synchronization Barrier
30 i00h = -i00 when i.step@25;
31 i01h = -(i01 + i01) when i.step@25;
32 i02h = -i02 when i.step@25;
33 i20h = i20 when i.step@25;
34 i21h = i21 + i21 when i.step@25;
35 i22h = i22 when i.step@25;
36 i00v = -i00 when i.step@25;
37 i02v = i02 when i.step@25;
38 i10v = -(i10 + i10) when i.step@25;
39 i12v = i12 + i12 when i.step@25;
40 i20v = -i20 when i.step@25;
41 i22v = i22 when i.step@25;
42 H = ((i00h + i01h) + i02h) + ((i20h + i21h) + i22h);
43 V = ((i00v + i10v) + i20v) + ((i02v + i12v) + i22v);
44 Hpos = H < 0 ? -H : H;
45 Vpos = V < 0 ? -V : V;
46 O = Hpos + Vpos;
47 Otrunk = O;
48 Otrunk = 255 when O > 255;
49 output.data_in = Otrunk;
50 output.address = i;
51 result = output;
52 }

```

cada iteração, e apenas um registrador adicional para armazenar o valor central. Embora o compilador não seja capaz de detectar e realizar automaticamente o reúso dos dados, este exemplo demonstra a potencialidade da linguagem em obter diferentes arquiteturas para a exploração do espaço de projeto. No [Capítulo 7](#) são apresentados os recursos ocupados em cada um dos casos e o desempenho obtido.

Figura 5.3: *Sobel*: (a) arquitetura original; (b) arquitetura melhorada

No [Algoritmo 5.14](#) é apresentada a descrição do algoritmo *ADPCM Coder* em LALP. Os vetores `indexTable[16]` e `stepSizeTable[98]` possuem valores constantes em suas declarações que foram omitidas por questões de espaço. Em LALP todas as instruções de um programa serão supostamente executadas em paralelo, a não ser que haja dependências entre elas. Neste caso, elas serão escalonadas para execução posterior às operações predecessoras.

Em alguns casos, existem dependências circulares entre as instruções, impedindo que uma iteração inicie antes do término de parte da iteração anterior. O escalonamento do [Algoritmo 5.14](#) é apresentado na [Figura 5.4](#), onde cada operação é executada em um único ciclo de *clock*, exceto a leitura de dados em memória, realizada em dois ciclos. Nota-se que as instruções são posicionadas de acordo com a dependência entre elas e que a segunda iteração do algoritmo não pode buscar o valor de `stepSizeTable` antes de calcular o valor de `index` da iteração anterior. O mesmo ocorre com as operações `valpred` e `diff`, respectivamente.

As instruções compreendidas entre os ciclos de *clock* 17 e 28, em destaque na [Figura 5.4](#), representam o caminho crítico da dependência que limita a execução em uma iteração a cada

Ciclo	Iteração 0				Iteração 1				Iteração 2					
0	len													
1	indata													
2														
3	val													
4	diff													
5	stepSizeTable			sign										
6														
7	diff2			step										
8	diff3	vpdiff	delta	step2										
9	delta2	vpdiff2	diff4	step3										
10	delta3			vpdiff3										
11	delta4			vpdiff4										
12	outputbuffer	valpred2	indexTable	len										
13	valpred3			indata										
14	index2													
15	index3	i	valpred	val										
16	bufferstep	outdata	index	diff										
17					stepSizeTable		sign							
18														
19					diff2		step							
20					diff3	vpdiff	delta	step2						
21					delta2	vpdiff2	diff4	step3						
22					delta3		vpdiff3							
23					delta4		vpdiff4							
24	outputbuffer	valpred2	indexTable	len										
25	valpred3			indata										
26	index2													
27	index3	i	valpred	val										
28	bufferstep	outdata	index	diff										
29					stepSizeTable			sign						
30														
31					diff2			step						
32					diff3	vpdiff	delta	step2						
33					delta2	vpdiff2	diff4	step3						
34					delta3			vpdiff3						
35					delta4			vpdiff4						
36	outputbuffer	valpred2	indexTable											
37	valpred3													
38	index2													
39	index3	i	valpred											
40	bufferstep	outdata	index											

Figura 5.4: Escalonamento para o exemplo *ADPCM Coder*

Algoritmo 5.14: Exemplo *ADPCM Coder* descrito em LALP

```

1  const DATASIZE = 1024;
2
3  typedef fixed(32,1) int;
4  typedef fixed(1,0) bit;
5
6  adpcm_coder(in bit init, out int output, out bit done) {
7      len.clk_en = init;
8      counter (len=0; len<DATASIZE; len++@12);
9      indata.address = len;
10     val = indata.data_out;
11     diff = val - valpred;
12     sign = diff < 0 ? 8 : 0;
13     stepSizeTable.address = index;
14     step = stepSizeTable.data_out;
15     diff2 = sign != 0 ? -diff : diff;
16     delta = diff2 >= step ? 4 : 0;
17     diff3 = diff2 >= step ? diff2 - step : diff2;
18     step2 = step >> 1;
19     vpdiff = (step >> 3);
20     delta2 = diff3 >= step2 ? delta | 2 : delta;
21     diff4 = diff3 >= step2 ? diff3 - step2 : diff3;
22     vpdiff2 = diff2 >= step ? vpdiff + step : vpdiff;
23     step3 = step2 >> 1;
24     delta3 = diff4 >= step3 ? delta2 | 1 : delta2;
25     vpdiff3 = diff3 >= step2 ? vpdiff2 + step2 : vpdiff2;
26     vpdiff4 = diff4 >= step3 ? vpdiff3 + step3 : vpdiff3;
27     delta4 = delta3 | sign;
28     valpred2 = sign != 0 ? valpred - vpdiff4 : valpred + vpdiff4;
29     indexTable.address = delta4;
30     outdata.address = i;
31     outputbuffer = (delta4 << 4) & 0xf0 when bufferstep & (len.step@11);
32     valpred3 = valpred2 > 32767 ? 32767 : valpred2;
33     index2 = index + indexTable.data_out;
34     index3 = index2 < 0 ? 0 : index2;
35     valpred = valpred3 < -32768 ? -32768 : valpred3;
36     i += 1 when !bufferstep & (len.step@14);
37     index = index3 > 88 ? 88 : index3;
38     outdata.data_in = (delta4 & 0x0f) | outputbuffer when !bufferstep
39                                                             & (len.step@15);
40     bufferstep = !bufferstep;
41     output = outdata.data_out;
42 }

```

12 ciclos de *clock*. O exemplo *ADPCM Decoder* é apresentado no [Algoritmo 5.15](#).

Apesar de possuir características muito semelhantes ao *ADPCM Coder* este algoritmo possui um número menor de instruções no caminho crítico mencionado, permitindo o escalonamento com um grau de *pipelining* maior. Enquanto o primeiro algoritmo permite a execução simultânea de apenas duas iterações, este permite que sejam realizadas até 5 iterações ao mesmo tempo, pois uma nova iteração é iniciada a cada 3 ciclos. Nota-se, no entanto, que número de

Algoritmo 5.15: Exemplo *ADPCM Decoder* descrito em LALP

```

1  const DATASIZE = 1024;
2
3  typedef fixed(32,1) int;
4  typedef fixed(1,0) bit;
5
6  adpcm_decoder(in bit init, out int output, out bit done) {
7      len.clk_en = init;
8      counter (len=0; len<DATASIZE; len++@3);
9      indata.address = i;
10     inputbuffer = indata.data_out when !bufferstep & (len.step@1);
11     delta = bufferstep ? inputbuffer & 0xf : (inputbuffer >> 4) & 0xf;
12     i += 1 when !bufferstep & (len.step@2);
13     sign = delta & 8;
14     indexTable.address = delta;
15     delta2 = delta & 7;
16     bufferstep = !bufferstep;
17     index2 = index + indexTable.data_out;
18     stepSizeTable.address = index;
19     index3 = index2 < 0 ? 0 : index2;
20     step = stepSizeTable.data_out;
21     index = index3 > 88 ? 88 : index3;
22     vpdiff = step >> 3;
23     vpdiff2 = ((delta2) & 1) > 0 ? vpdiff + (step >> 2) : vpdiff;
24     vpdiff3 = ((delta2) & 2) > 0 ? vpdiff + (step >> 1) : vpdiff2;
25     vpdiff4 = ((delta2) & 4) > 0 ? vpdiff + (step) : vpdiff3;
26     valpred2 = (sign) > 0 ? valpred - vpdiff4 : valpred + vpdiff4;
27     valpred3 = valpred2 > 32767 ? 32767 : valpred2;
28     valpred = valpred3 < -32768 ? -32768 : valpred3;
29     outdata.address = len;
30     outdata.data_in = valpred;
31     output = outdata.data_out;
32     done = len.done;
33 }

```

ciclos de *clock* necessários para cada iteração é o mesmo nos dois exemplos.

Na [Figura 5.5](#) é apresentado o escalonamento para o [Algoritmo 5.15](#). Por análise do código, é possível notar que a mesma dependência ocorre entre as operações `stepSizeTable` e `index`, mas o número de operações entre elas é muito menor. No [Capítulo 6](#), são descritos os algoritmos usados para detectar e resolver estas dependências para obtenção do maior *throughput* possível.

5.2 Limitações Impostas pela Linguagem

A exploração de paralelismo é um objetivo fundamental em sistemas de computação não convencionais, tais como arquiteturas multiprocessadas, processadores vetoriais, máquinas vetoriais e arquiteturas reconfiguráveis ou híbridas voltadas para alto desempenho. Grandes es-

Ciclo	Iteração 0				Iteração 1				Iteração 2			
0	indata		len									
1												
2	inputbuffer											
3	i		delta		indata		len					
4	indexTable	bufferstep	sign	delta2								
5						inputbuffer						
6	stepSizeTable		index2		i		delta		indata		len	
7			index3		indexTable	bufferstep	sign	delta2				
8	step		index									
9	vpdiff				stepSizeTable		index2		i		delta	
10	vpdiff2						index3		indexTable	bufferstep	sign	delta2
11	vpdiff3				step		index					
12	vpdiff4				vpdiff				stepSizeTable		index2	
13	valpred2				vpdiff2						index3	
14	valpred3				vpdiff3				step		index	
15	valpred				vpdiff4				vpdiff			
16	outdata				valpred2				vpdiff2			
17					valpred3				vpdiff3			
18					valpred				vpdiff4			
19					outdata				valpred2			
20									valpred3			
21									valpred			
22									outdata			

Figura 5.5: Escalonamento para o exemplo *ADPCM Decoder*

forços foram realizados no desenvolvimento de compiladores paralelizantes para linguagens convencionais, bem como para desenvolver linguagens especializadas que pudessem expor ao programador as características de paralelismo destas arquiteturas. Muitas destas linguagens refletem o comportamento do sistema para o qual elas foram projetadas e não facilitam a maneira com o programador pensa normalmente na solução dos problemas (Ackerman, 1982).

As linguagens existentes para programação paralela são, em geral, voltadas para sistemas distribuídos ou multiprocessados e não voltadas para a geração de arquiteturas especializadas como seria adequado para os sistemas reconfiguráveis baseados em FPGAs (Bal, 1992; Feo, 1992). São usadas, portanto, linguagens especializadas para criação de *hardware* como VHDL e Verilog que permitem a exploração das menores características possíveis do dispositivo alvo. Qualquer tentativa de descrever uma arquitetura em detalhes a partir de uma linguagem de

programação convencional, por exemplo C, irá certamente esbarrar em limitações impostas por esta linguagem. O objetivo da LALP foi minimizar estas limitações possibilitando um maior controle nas operações do programa sem a complexidade necessária às linguagens de descrição de *hardware*, tais como VHDL e Verilog.

Uma das principais limitações da linguagem desenvolvida está na impossibilidade de descrever acessos à memória em diferentes índices, tais como `a = v[i]; b = v[i+1];`. Nestes casos, considerando memórias com uma única porta, o programador deve escalonar manualmente o acesso à memória gerada para o vetor `v`, como nos exemplos apresentados até aqui. Conforme resultados apresentados no [Capítulo 7](#), as ferramentas para síntese de alto nível não são capazes de gerar arquiteturas eficientes quando tratam algoritmos com este tipo de acesso à memória.

Os contadores usados permitem que o número de iterações seja variável, mas não o número de ciclos em cada iteração. Esta limitação inviabiliza o uso da técnica em *loops* irregulares aninhados, pois a quantidade de iterações no *loop* mais interno (*innermost*) é que determina a quantidade de ciclos do *loop* mais externo (*outermost*). Nestes casos, a solução seria adicionar mecanismos de controle ao componente contador para que o término da contagem interna pudesse disparar uma nova iteração na contagem externa. Os algoritmos de *loop pipelining*, em geral, são aplicados apenas a *loops* regulares ou a *loops* irregulares com suporte específico de *hardware*. Em [Granston et. al. \(2001\)](#) é proposta uma solução para este problema em um domínio específico de aplicações.

Em LALP são aceitos apenas arranjos unidimensionais e cada vetor declarado no código irá gerar um componente de memória independente. A conversão de arranjos multidimensionais em unidimensionais e o mapeamento de múltiplos arranjos em memória poderia ser adicionado por um pré-processamento do código. As demais limitações da linguagem incluem repetições irregulares, blocos de instruções para estruturas de decisão e repetição, modularização por meio de procedimentos ou funções e o uso de ponteiros para alocação dinâmica de recursos.

5.3 A Linguagem LALP-S

Para facilitar modificações na representação intermediária do compilador, durante o processo de desenvolvimento da ferramenta, foi desenvolvida uma linguagem textual que descrevesse os grafos diretamente. A linguagem, denominada LALP-S, permite a descrição da arquitetura de forma estrutural, listando-se os componentes e as ligações entre eles. A forma geral de um grafo descrito nesta linguagem é apresentado no [Algoritmo 5.16](#). Inicialmente, são declaradas constantes, nome da entidade e interfaces de entrada e saída, assim como em LALP, mas com exceção à definição de tipos de dados. A seguir são declarados componentes que devem estar presentes na biblioteca do compilador e as ligações entre estes componentes.

Algoritmo 5.16: Forma geral de um programa descrito em LALP-S

```
1 declaração de constantes
2 entidade(pinos de entrada/saída) {
3   {
4     declaração de componentes
5   }
6   ligações entre os componentes
7 }
```

No [Algoritmo 5.17](#) é apresentado o exemplo *Dotprod* descrito nesta linguagem, cuja tarefa é calcular a soma dos produtos de dois vetores. Nas linhas 5 a 13 são declaradas instâncias dos componentes que serão usados, descrevendo-se os parâmetros necessários. Por exemplo, nas linhas 7 e 8 são instanciadas duas memórias com palavras de 32 bits e com 11 bits de endereço. Da linha 14 em diante são descritas todas as ligações entre os componentes, usando-se uma notação do tipo `componente.porta` na origem e no destino do sinal, ligados pelo operador `<-`. Para alguns componentes não é necessário descrever a porta de ligação pois este pode possuir uma porta padrão ou uma única porta, como no caso das constantes e pinos de entrada/saída. No operador de ligação pode ser fornecido um nome entre parênteses que será usado no sinal que fará a ligação.

Algoritmo 5.17: Exemplo *Dotprod* descrito em LALP-S

```

1  const c14 = 2048;
2  const c13 = 0;
3
4  dotprod_alp(out fixed(1,0) done, out fixed(32,1) sum, in fixed(1,0) init) {
5  {
6      i: counter(16, 1, 1, 0, 0);
7      x: block_ram(11, 32);
8      y: block_ram(11, 32);
9      x_data_out_mult_op_s_y_data_out: mult_op_s(32);
10     acc: add_reg_op_s(32);
11     i_done_delay_op_2: delay_op(1, 2);
12     i_step_delay_op_1: delay_op(1, 1);
13 }
14 i.input <-(s0) c13;
15 i.termination <-(s1) c14;
16 i.clk_en <-(s2) init;
17 x.address <-(s3) i.output;
18 y.address <-(s4) i.output;
19 x_data_out_mult_op_s_y_data_out.I0 <-(s5) x.data_out;
20 x_data_out_mult_op_s_y_data_out.I1 <-(s6) y.data_out;
21 i_step_delay_op_1.a <-(s7) i.step;
22 acc.I0 <-(s8) acc.O0;
23 acc.I1 <-(s9) x_data_out_mult_op_s_y_data_out.O0;
24 acc.we <-(s10) i_step_delay_op_1.a_delayed;
25 sum <-(s11) acc.O0;
26 i_done_delay_op_2.a <-(s12) i.done;
27 done <-(s13) i_done_delay_op_2.a_delayed;
28 }

```

5.4 Extensões Possíveis

Durante a programação dos exemplos usados neste trabalho, foram constatadas algumas extensões para a linguagem que poderiam facilitar a descrição das arquiteturas ou mesmo a exploração do espaço de projeto. A seguir são listadas algumas destas extensões:

- **Modularização:** possibilidade de descrever os programas de maneira hierárquica. Apenas para fins de ilustração, no [Algoritmo 5.18](#) é apresentada uma implementação modular do exemplo *Sobel*. Inicialmente, um módulo é declarado contendo a parte da implementação que aparece mais de uma vez no código original (linhas 1 a 7). A seguir, nas linhas 24 e 25, são criadas duas instâncias deste módulo, similarmente a implementação realizada nas linguagens de descrição de hardware. O código substituído é mantido em comentários entre as linhas 18 e 23 para facilitar o entendimento.

Algoritmo 5.18: Exemplo *Sobel* descrito em LALP com modularização

```

1  calc(in int a, ..., in int f, out int Npos) {
2      {
3          int N;
4      }
5      N = ((-a) + (-2 *@6 b)) + (((-c) + d) + (2 *@6 e + f));
6      Npos = N < 0 ? -N : N;
7  }
8
9  sobel_alt(in bit init, out bit done, out int result) {
10     {
11         int i00, i01, i02;
12         int i10,      i12;
13         int i20, i21, i22;
14         int O, Hpos, Vpos;
15         ...
16     }
17     ...
18     /*
19     H = ((-i00) + (-2 *@6 i01)) + (((-i02) + i20) + (2 *@6 i21 + i22));
20     V = ((-i00) + i02) + (((-2 *@6 i10) + 2 *@6 i12) + ((-i20) + i22));
21     Hpos = H < 0 ? -H : H;
22     Vpos = V < 0 ? -V : V;
23     */
24     H: calc(i00, i01, i02, i20, i21, i22, Hpos)
25     V: calc(i00, i10, i20, i02, i12, i22, Vpos)
26     O = Hpos + Vpos;
27     ...
28 }

```

- **Expressões aritméticas em parâmetros de sincronização:** possibilidade de usar expressões no lugar de constantes, especialmente nos parâmetros de sincronização que representam um número de ciclos. Em certos casos, nos quais diretivas de sincronização se façam necessárias, a simples mudança de uma constante poderia modificar o grau de paralelismo sem que o código deixasse de gerar resultados corretos.
- **Distâncias relativas:** possibilidade de descrever a distância relativa entre duas operações. Durante o escalonamento, sempre que a operação precedente for deslocada a seguinte deveria acompanhar a mudança com o mesmo número de estágios. Isso possibilitaria, por exemplo, forçar uma dependência para que duas operações compartilhassem o mesmo recurso.
- **Dependências entre iterações diferentes:** possibilidade de descrever a distância relativa entre operações de iterações diferentes, facilitando a inferência do paralelismo pelo

compilador.

- **Barreiras de sincronização:** possibilidade de descrever um conjunto de operações que devem terminar ao mesmo tempo, sem a necessidade de indicar cada estágio manualmente.

Mapeamento de LALP em FPGAs

NESTE capítulo são descritas as técnicas empregadas para traduzir diretamente os programas escritos em LALP e LALP-S para VHDL. O compilador ALP possui aproximadamente vinte mil linhas de código Java, das quais pouco mais de oito mil compreendem o *frontend* gerado com o JavaCC. O restante do código está dividido nas seguintes partes:

- algoritmos para geração da representação intermediária a partir da árvore sintática anotada;
- representação intermediária propriamente dita, incluindo as classes descritas na [Figura 6.3](#);
- representação dos componentes da biblioteca VHDL, contendo informações de suas particularidades (genéricos, portas, ciclos de *clock* necessários para operação, etc);
- algoritmos de otimização e escalonamento que modificam a representação intermediária;
- *backend* que realiza a geração de código;

Na [Seção 6.1](#) é apresentada a abordagem adotada neste trabalho para tratar o problema de criação de um *pipeline* otimizado. Na [Seção 6.2](#) é descrita a biblioteca de componentes,

adaptada do compilador NENYA. Na [Seção 6.3](#) é descrita a representação intermediária usada no compilador. Na [Seção 6.4](#) estão relacionados os algoritmos que atuam na representação intermediária para realizar o escalonamento das operações. Na [Seção 6.5](#) são apresentadas as visualizações geradas para auxiliar no processo de desenvolvimento. Por fim, na [Seção 6.6](#) é apresentado o protótipo de uma interface gráfica do usuário para o compilador.

6.1 Abordagem

A abordagem usada neste trabalho, denominada ALP, mostrou que contadores podem ser usados para fornecer o controle para a maioria dos *loops* presentes nos programas. Tais contadores são capazes de operar em altas frequências de *clock*, se forem adequadamente adicionados estágios nas estruturas do *datapath* responsáveis pela computação das operações dos *loops*. Com esta abordagem, mesmo repetições com estruturas condicionais, que podem resultar em caminhos com diferentes latências, podem ser realizadas em *pipelining* se as latências forem devidamente preenchidas, conforme apresentado na [Figura 6.1](#). Desta maneira, as operações no corpo do laço são realizadas nos ciclos de *clock* de acordo com os caminhos tomados durante a execução, permitindo a obtenção de um *pipelining* altamente paralelo. A principal razão para isto é o fato de que as técnicas tradicionais atribuem estaticamente as operações em cada estado da máquina de estados finitos que controla o *loop*. Para isso, é necessário considerar a latência do caminho crítico do corpo da repetição (muito conservador) ou considerar todos os caminhos possíveis (muito complexo). A última opção é muito complexa e, até onde se sabe, não é usada pelos compiladores existentes para este fim. Como em ALP as operações não são estaticamente atribuídas a estados, estes problemas não precisam ser enfrentados ([Menotti et. al., 2007](#)).

Considerando o exemplo da [Figura 6.1](#), quando o número de registradores em cascata, existentes nos caminhos para preencher os estágios do *pipeline* no *datapath*, é muito alto, o sinal *step* pode ser usado no lugar do índice *i*. Neste caso, uma duplicação do contador seria usada, sendo o incremento no segundo contador habilitado pelo sinal *step* do primeiro.

Ferramentas de síntese de alto nível consideram primitivas básicas (componentes lógicos e aritméticos simples) para gerar uma arquitetura específica ([Gupta et. al., 2004b](#)). Em ALP são usados componentes mais complexos, tais como contadores e acumuladores, o que permite a

Figura 6.1: Exemplo de ALP: (a) trecho de código; (b) estruturas de *hardware*

construção conjunta de estruturas de dados e controle.

O fluxo de desenvolvimento com ALP é apresentado na Figura 6.2. A partir de um programa descrito em qualquer linguagem de alto nível deve ser realizada a tradução manual para o código LALP, passo indicado pelas linhas tracejadas. A tradução direta para a linguagem de grafos (LALP-S) só indicada para geração de arquitetura muito simples, nos casos em que a ligação de poucos componentes seja suficiente para representar o programa equivalente.

O *parser* da linguagem recebe o arquivo .ALP ou .ALPS e gera a representação intermediária do programa. Por meio da linha de comandos ou da interface gráfica do usuário é possível executar algoritmos para escalonamento, balanceamento e sincronização das operações. Estes algoritmos irão modificar o grafo da representação intermediária, inserindo registradores de deslocamento e sinais de controle entre os componentes.

Existem três classes para geração de código: LALP-S, Graphviz e VHDL. O formato de saída LALP-S é uma representação textual do grafo e não possui variações. O gerador de códigos para Graphviz possui dois formatos distintos, descritos adiante. A geração de código VHDL resulta em um arquivo com o nome usado internamente no programa. Opcionalmente é

possível gerar um arquivo de testes (*testbench*) com estímulos básicos de *clock* e *reset* que pode ser complementado. Caso o programa contenha declarações de arranjos com valores iniciais o compilador não usará o componente padrão de memória da biblioteca. Um único arquivo contendo todas as memórias que necessitam de inicialização é criado com componentes gerados dinamicamente.

Figura 6.2: Fluxo de desenvolvimento com ALP

6.2 Biblioteca de Componentes

Assim como na maioria dos compiladores para *hardware*, foi adotada uma biblioteca de componentes para a geração de código. No [Algoritmo 6.1](#) é apresentada a entidade VHDL de um dos componentes desta biblioteca. Em geral, os parâmetros determinam a largura de bits das entradas e saídas (linhas 4 a 6) e outros aspectos configuráveis de cada operação. Neste exemplo, o parâmetro *stages* é usado para determinar o número de estágios (ciclos de *clock*) usados em uma operação de multiplicação.

A biblioteca usada como ponto de partida foi a do compilador NENYA ([Cardoso, 2000](#); [Cardoso e Neto, 2003](#)), acrescida de novos componentes para suportar a técnica ALP. Entre os componentes estão operações lógicas e aritméticas com inteiros, além de operações de deslo-

Algoritmo 6.1: Exemplo de componente parametrizável da biblioteca VHDL

```

1  — signed pipelined multiply operation
2  entity mult_op_s_p is
3      generic (
4          w_in1  : integer := 16;
5          w_in2  : integer := 16;
6          w_out   : integer := 32;
7          stages : integer := 5
8      );
9      port (
10         clk : in    std_logic;
11         I0  : in    std_logic_vector(w_in1-1 downto 0);
12         I1  : in    std_logic_vector(w_in2-1 downto 0);
13         O0  : out   std_logic_vector(w_out-1 downto 0)
14     );
15 end mult_op_s_p;

```

camento, comparadores, multiplexadores, memórias e contadores. Cada operação possui uma versão com sinal e outra sem sinal, mas apenas as operações com sinal são usadas pelo compilador.

A geração de código VHDL a partir de um biblioteca como esta é vantajosa, pois cada componente da biblioteca está diretamente associado a um elemento da representação intermediária, permitindo que a tradução seja realizada diretamente sem nenhuma análise do grafo.

6.3 Representação Intermediária

Para representar internamente as arquiteturas foi usado um *framework* denominado JUNG (O'Madadhain *et. al.*, 2003), que possui inúmeros algoritmos para criação de manipulação de grafos. No entanto a categoria de grafo, vértice e aresta escolhidos limitou a quantidade destes algoritmos que puderam ser usados. A escolha destas classes foi direcionada para a representação de *hardware* na forma de componentes, o que demanda por arestas direcionadas e paralelas quando, por exemplo, um componente é conectado a outro por mais de um sinal. A maioria dos algoritmos de busca do *framework* não funcionam com estes tipos de grafo e seus métodos tiveram que ser sobrecarregados no compilador.

Uma visão macro da estrutura é apresentada na [Figura 6.3](#). Cada projeto do compilador é representado por uma especialização da classe `Design` que por sua vez é uma especialização da classe `SparseGraph`. Os projetos possuem componentes, representados pela classe `Component` herdada de `SparseVertex`, além de sinais representados pela classe `Signal`

herdada de `DirectedSparseEdge`. Para cada componente da biblioteca NENYA existe uma classe, derivada de `Component`, correspondente.

Figura 6.3: Diagrama das classes usadas para representação intermediária

Os componentes podem ser instanciados e conectados diretamente no código Java, assim como foram construídas as primeiras arquiteturas para analisar a viabilidade da técnica. No [Algoritmo 6.2](#) é listado o código Java capaz de descrever o exemplo *Dotprod* usando as classes da representação intermediária.

A classe `dotprod_hw` representa a arquitetura (grafo), que irá conter componentes (nós) e ligações (arestas) entre eles. Os pinos `clk` e `reset` são pinos de entrada especiais, pois são automaticamente conectados a qualquer componente que possua tais entradas. Entre as linhas 7 e 9 são instanciados e adicionados os demais pinos, `init`, `sum` e `done`. Entre as linhas 11 e 23 os componentes necessários são instanciados e adicionados ao grafo. Os construtores destes componentes possuem parâmetros que podem determinar o número de bits, os estágios de *pipeline* e outras configurações necessárias. Entre as linhas 25 e 34 são realizadas as conexões entre os componentes. Após a criação da linguagem para descrição das arquiteturas a criação e ligação dos componentes foi automatizada, tornando descrições deste tipo não mais necessárias.

Algoritmo 6.2: Exemplo *Dotprod* descrito diretamente no código Java

```

1 public class dotprod_hw extends Design {
2     public dotprod_hw() {
3         super("dotprod");
4         // input/output pins
5         this.addGlobalComponent(new input("clk"));
6         this.addGlobalComponent(new input("reset"));
7         input init = (input) this.addComponent(new input("init"));
8         output sum = (output) this.addComponent(new output("sum", 32));
9         output done = (output) this.addComponent(new output("done"));
10        // intermediate components
11        module_for for1 =
12            (module_for) this.addComponent(new module_for("for1", 11, 2048));
13        block_ram ramx =
14            (block_ram) this.addComponent(new block_ram("ramx", 11));
15        block_ram ramy =
16            (block_ram) this.addComponent(new block_ram("ramy", 11));
17        mult_op_s mult1 = (mult_op_s) this.addComponent(new mult_op_s("mult1"));
18        add_reg_op_s acc1 =
19            (add_reg_op_s) this.addComponent(new add_reg_op_s("acc1"));
20        delay_op doneDelayed =
21            (delay_op) this.addComponent(new delay_op("doneDelayed", 1, 2));
22        delay_op stepDelayed =
23            (delay_op) this.addComponent(new delay_op("stepDelayed", 1, 1));
24        try { //connections
25            init.connectComponent(for1, "clk_en");
26            for1.connectComponent("done", doneDelayed);
27            doneDelayed.connectComponent(done);
28            for1.connectComponent("step", stepDelayed);
29            for1.connectComponent(ramx);
30            for1.connectComponent(ramy);
31            mult1.connectComponentIOI1(ramx, ramy);
32            acc1.connectComponentIOI1(acc1, mult1);
33            acc1.connectComponent(sum);
34            stepDelayed.connectComponent(acc1, "we");
35        } catch (Exception e) {
36            e.printStackTrace();
37        }
38    }
39 }

```

6.4 Algoritmos

A seguir são descritos os algoritmos usados para extrair características importante do grafo ou aplicar transformações. A execução destes algoritmos é facultativa e em alguns casos a execução de determinado algoritmo inviabiliza o uso de outro para o mesmo fim. A escolha dos algoritmos é feita por meio da linha de comandos ou da interface gráfica do usuário. Para facilitar a compreensão, as descrições apresentadas em ICAN são simplificadas e não descrevem todos os detalhes da implementação em Java.

Uma característica importante do grafo que representa o programa é a conectividade entre seus elementos. Um componente fortemente conectado (SCC¹) de um grafo $G(V, E)$ é um conjunto maximal de vértices $C \in V$ tal que para cada $u, v \in C$ existe $u \rightsquigarrow v$ e $v \rightsquigarrow u$ em G . Ou seja, u e v são alcançáveis mutuamente em G . O grafo apresentado na Figura 6.4 possui quatro SCCs em destaque.

Figura 6.4: Grafo com componentes fortemente conectados em destaque

O algoritmo listado no Algoritmo 6.3 é usado para detectar os SCCs de um grafo². Após a detecção os nós são representados em coloração diferente, facilitando a análise em programas grandes. Inicialmente, é criado um grafo transposto (linha 13) que é uma cópia do grafo original com todas as arestas invertidas. Depois é realizada uma busca em profundidade recursiva a partir de um nó inicial e gerada a lista `sccMap` dos nós visitados no procedimento `visit`. Finalmente o grafo transposto é percorrido em profundidade novamente, mas em ordem inversa da lista `sccMap`.

Uma das tarefas realizadas pelo *parser* da linguagem é anotar em cada nó da árvore a linha do código fonte em que é feita a atribuição. Esta informação é usada para determinar arestas recorrentes nos casos em que existem dependências circulares entre as operações. As dependências circulares determinam o ciclo de execução dos *loops*, ou seja, o número de ciclos de *clock* necessários para a execução de cada iteração do *loop*.

O Algoritmo 6.4 pode ser usado para assinalar as arestas recorrentes e tornar possível o escalonamento das operações em casos onde ocorrem ciclos. O algoritmo percorre todas as

¹Strongly Connected Component

²Em ICAN o operador @ é usado para acessar os elementos de uma tupla e não deve ser confundido com a diretiva de sincronização usada em LALP.

Algoritmo 6.3: Computação dos componentes fortemente conectados (SCC)

```

1  level := 0: integer
2  dfs := 0: integer
3  sccMap: set of integer × Component
4  sccLevels: set of integer
5  transpose: Design
6
7  procedure detectStrongConnectedComponents(design, initial)
8      design: in Design
9      initial: in Component
10  begin
11      component: Component
12      me: integer × Component
13      transpose := design.getTranspose()
14      visit(initial)
15      for each me ∈ sccMap do
16          level++
17          component := me@2
18          if component.getLevel() = 0 then
19              visitTranspose(design, component)
20          fi
21      od
22  end
23
24  procedure visit(component)
25      component: in Component
26  begin
27      successor: Component
28      component.setVisit(dfs++)
29      for each successor ∈ component.getSuccessors() do
30          if successor.getVisit() = 0 then
31              visit(successor)
32          if
33      do
34      sccMap ∪= ⟨-dfs, component.getEqualVertex(transpose)⟩
35  end
36
37  procedure visitTranspose(design, component)
38      design: in Design
39      component: in Component
40  begin
41      successor: Component
42      component.getEqualVertex(design).setLevel(level)
43      for each successor ∈ component.getSucessors() do
44          if successor.getLevel() = 0 then
45              visitTranspose(design, successor)
46          elif successor.getLevel() = level then
47              sccLevels ∪= level
48          fi
49      od
50  end

```

arestas do grafo (linha 6) e as seleciona como recorrentes nos seguintes casos:

- Componente autoconectado: nos casos em que um componente tem uma saída conectada

a uma entrada (linha 9);

- Componentes conectados entre si: nos casos em que um par de arestas formam um ciclo entre dois componentes (linha 12);
- Ordem das atribuições: nos casos em que um componente recebe um valor que será calculado em uma linha posterior (linha 15);

O último caso indica que a atribuição irá usar um valor da iteração anterior do *loop*, causando uma dependência entre as operações que irá limitar, por exemplo, o cálculo de uma iteração por ciclo de *clock*. A maior distância entre os componentes ligados por estas arestas determina o ritmo de execução do ciclo.

Algoritmo 6.4: Detecção de arestas recorrentes

```

1 procedure detectBackwardEdges ( design ) || based on source code attributions
2   design: in Design
3 begin
4   source , dest: Component
5   signal: Signal
6   for each signal ∈ design.getSignals () do
7     source := signal.getSource ()
8     dest := signal.getDest ()
9     if source = dest then
10      signal.setBackEdge (true);
11    fi
12    if source.isSuccessorOf (dest) then
13      signal.setBackEdge (true);
14    fi
15    if source.getLine () > dest.getLine () then
16      signal.setBackEdge (true);
17    fi
18  od
19 end

```

Após a remoção das arestas recorrentes, pode ser realizado o escalonamento das operações. Este processo irá definir em que ciclo de *clock* cada operação será efetuada. No [Algoritmo 6.5](#) é descrito o escalonamento ASAP de forma iterativa e sem restrição de recursos. Para cada nó do grafo são percorridos os predecessores e calculado o maior escalonamento. Caso este seja maior do que o escalonamento atual ele é substituído e a mudança é sinalizada determinando uma nova iteração global do algoritmo até que não ocorram mais mudanças.

Algoritmo 6.5: Escalonamento ASAP modificado

```

1 procedure ASAP(design , attributions)
2   design: in Design
3   attributions: in set of integer  $\times$  Component
4 begin
5   source , dest: Component
6   signal: Signal
7   me: integer  $\times$  Component
8   max, pred: integer
9   change := true: boolean
10  while change do
11    for each dest  $\in$  design.getComponents() do
12      max := pred := 0
13      for each source  $\in$  dest.getPredecessors() do
14        pred := source.getASAP() + source.getDelay();
15        if pred > max then
16          max := pred
17        fi
18      od
19      if dest  $\in$  attributions then || additional condition
20 inner: for each me  $\in$  attributions do
21      line := me@1
22      source := me@2
23      if line > dest.getLine() then
24        break inner;
25      fi
26      pred := source.getASAP()
27      if pred > max then
28        max := pred
29      fi
30    od
31    fi
32    if dest.getASAP()  $\neq$  max then
33      dest.setASAP(max)
34      change := true
35      if max > design.getMaxSchedulingTime() then
36        design.setMaxSchedulingTime(max)
37      fi
38    fi
39  od || for each dest
40  od || while change
41 end

```

Uma característica diferenciada neste processo está nas instruções compreendidas entre as linhas 19 e 31. Além de considerar a dependência dos predecessores de uma operação, o escalonamento determina que toda atribuição realizada em uma linha anterior deve ser escalonada no mesmo ciclo ou antes da atribuição atual. Essa característica impede que operações que recebem apenas arestas recorrentes, as quais não são consideradas para buscar os predecessores, sejam escalonadas no início da execução e causem uma distância grande entre as operações. As operações são escalonadas na ordem em que aparecem no código sem que o paralelismo seja

comprometido.

Para exemplificar o impacto desta mudança, são apresentadas duas versões de escalonamento do algoritmo *ADPCM Coder* na Figura 6.5. Os ciclos que limitam o ritmo de execução estão destacados. Na Figura 6.5(a) foi aplicado o escalonamento ASAP original. A memória *stepSizeTable*, apenas de leitura, é endereçada pelo registrador *index*. A ligação entre eles é uma aresta recorrente, pois o valor de *index* em uma iteração irá determinar o valor de *stepSizeTable* na iteração seguinte. Esta memória é então escalonada no instante 0, gerando uma aresta recorrente com distância de 16 ciclos de *clock*. O escalonamento na versão ASAP modificada, apresentado no Figura 6.5(b), agrega a restrição de que a memória *stepSizeTable* deve ser escalonada após ou junto com certas atribuições que foram declaradas em linhas anteriores (neste caso no mesmo ciclo que o registrador *sign*). O mesmo ocorre com o registrador *bufferstep*.

Figura 6.5: Exemplo *ADPCM Coder* com diferentes escalonamentos

Além do escalonamento ASAP, foi implementado também o escalonamento ALAP. A diferença no ciclo em que uma operação é escalonada nos dois algoritmos é chamada de mobilidade e pode ser usada para balancear operações em alguns casos.

Após determinar o ciclo em que cada operação será executada o algoritmo descrito no [Algoritmo 6.6](#) é aplicado para balancear as saídas dos contadores, normalmente usadas como endereços de memórias. O algoritmo recebe uma lista dos contadores encontrados no código. Para cada contador encontrado no código são analisados todos os componentes conectados ao valor de saída deste contador e inseridos atrasos necessários conforme o escalonamento do componente de destino. Supondo que um mesmo contador forneça os endereços para uma memória no início da arquitetura e para outra no final, o conjunto de componentes compreendido entre estas memórias irá determinar quantos atrasos serão necessários entre o contador e a memória final.

Algoritmo 6.6: Sincronização de contadores

```

1 procedure synchroizeCounters( counters )
2   counters: in set of integer × Component
3 begin
4   c, counter := nil, delay, dest, source: Component
5   counterSched, destSched, distance, mii := 0: integer
6   me: integer × Component
7   s, signal: Signal
8   for each me ∈ counters do
9     counter := me@2
10    counterSched := counter.getAsAP();
11    s := counter.getDefaultOutput().getSignal() || counter value (e.g. i)
12    while s ≠ nil do
13      dest := s.getDest()
14      distance := dest.getAsAP() - counterSched
15      if distance > i then
16        s.insertDelay( distance - 1 )
17      fi
18      s := s.getSignal()
19    od
20  od || for each me
21 end

```

Outro papel fundamental dos contadores é sincronizar as operações de acordo com o ciclo calculado no escalonamento. Novamente, é considerada a ordem das atribuições no código fonte e cada componente é sincronizado pelo sinal `step` do contador imediatamente anterior a sua atribuição. Essa abordagem permite a construção de contadores em série, paralelos ou

aninhados conforme descrito a seguir.

O Algoritmo 6.7 recebe como parâmetro um conjunto de componentes que sofrem atribuições no código, cada um com seu contador imediatamente anterior. A condição listada na linha 11, serve para verificar se o componente já possuiu uma restrição de execução especificada pelo usuário. Caso esta restrição não exista, é calculada a distância entre o contador e o componente, criado um registrador de deslocamento com o mesmo número de passos e inserido entre a porta *step* do contador e a porta *we* da componente. Esta porta é responsável por habilitar a escrita do componente.

Algoritmo 6.7: Sincronização de operações

```

1 procedure synchrhize( attributions )
2   attributions: in set of Component  $\times$  Component
3 begin
4   comp, counter := nil, delay, dest, source: Component
5   counterSched, destSched, distance, mii := 0: integer
6   me: integer  $\times$  Component
7   s, signal: Signal
8   for each me  $\in$  attributions do
9     comp = me@1
10    counter = me@2
11    if comp.getPort("we").getSignal() = nil then
12      distance := comp.getASAP() - counter.getASAP()
13      if distance > 1 then
14        delay := new delay_op(1, distance - 1)
15        counter.connectComponent("step", delay)
16        delay.connectComponent(comp, "we")
17      fi
18    fi
19  od
20 end

```

Finalmente, as operações atreladas aos contadores possuem ciclos exatos para a execução, permitindo identificar se os sinais de entrada de cada componente estão balanceados. Na Figura 6.6 são apresentados escalonamentos com valores em cada nó para os algoritmos ASAP e ALAP. Na Figura 6.6(a) a operação $\subset$ tem mobilidade um e fornece o operando em um tempo diferente do outro lado da operação de soma. O mesmo problema ocorre na Figura 6.6(b), mas sem que os nós neste caso tenham mobilidade.

Nenhuma interferência seria necessária para balancear estes sinais se os componentes de origem não tivessem seus valores modificados a cada ciclo, ou seja, nos casos em que o ritmo de execução do *loop* fosse maior do que a diferença entre a origem e o destino destes sinais.

Figura 6.6: Caminhos desbalanceados no grafo

No entanto, o compilador possui um algoritmo de balanceamento conservador para evitar que valores possam ser perdidos por diferenças de escalonamento dos operandos dos componentes.

O algoritmo descrito no [Algoritmo 6.8](#) percorre cada aresta do grafo identificando se o sinal correspondente está balanceado. Caso seja necessário, um atraso é inserido por meio de um registrador de deslocamento.

A [Figura 6.7](#) contém o fluxo de execução dos algoritmos até a obtenção da arquitetura correta. Para determinar as arestas recorrentes, além do algoritmo baseado na ordem das atribuições, pode ser usado o algoritmo de busca em profundidade (DFS³) ou o baseado em dominadores. Estes algoritmos foram testados e se mostraram eficientes em alguns casos. No entanto, para grafos irredutíveis, eles não são capazes de determinar estas arestas corretamente.

³Depth-First Search

Algoritmo 6.8: Balanceamento de arestas

```

1 procedure balance ( design )
2   design: in Design
3 begin
4   dest, source: Component
5   counterSched, destSched, distance: integer
6   me: integer × Component
7   signal: Signal
8   for each signal ∈ design.getSignals() do
9     source := signal.getSource()
10    dest := signal.getDest()
11    sourceSched := source.getASAP() + source.getDelay()
12    destSched := dest.getASAP()
13    distance := destSched - sourceSched
14    if distance > 1 then
15      signal.insertDelay ( distance - 1 )
16    fi
17  od
18 end

```

Figura 6.7: Algoritmos usados na compilação

6.5 Visualização

Durante o processo de desenvolvimento de um arquitetura, a visualização dos grafos gerados constitui um aspecto importante, uma vez que a intervenção do usuário é requerida com uma certa frequência, conforme mencionado anteriormente. A seguir são descritas as principais formas de visualização usadas neste trabalho, no qual foi usado o *software* Graphviz ([GraphvizURL, 2006](#)) pelas seguintes razões:

- Geração de visualizações a partir de descrições textuais simples;
- Poderosos algoritmos para posicionamento automático dos nós;

- Exportação para inúmeros formatos de bitmap e vetoriais, tais como JPEG⁴, PNG⁵, SVG⁶ e PDF⁷, entre outros;

Os principais tipos de visualizações gerados pelo compilador ALP são apresentado na [Figura 6.8](#). Em ambos os casos, cada nó está diretamente relacionado à instância de um componente da biblioteca VHDL e cada aresta à uma ligação entre elas.

Figura 6.8: Visualizações geradas pelo compilador ALP com auxílio do Graphviz

O primeiro tipo de grafo, apresentado na [Figura 6.8\(a\)](#), possui nós mais simples e está mais voltado para o posicionamento dos nós do que para os detalhes de cada componente. Neste tipo de visualização os pinos de entrada/saída e as constantes são representados por triângulos, os componentes não registrados por elipses e os componentes registrados por octógonos. As arestas são direcionadas e representam ligações entre os componentes. Em alguns casos,

⁴Joint Photographic Experts Group

⁵Portable Network Graphics

⁶Scalable Vector Graphics

⁷Portable Document Format

são geradas arestas tracejadas para representar uma característica especial, por exemplo arestas recorrentes, e nós com colorações diferentes para representar um conjunto específico, por exemplo componentes fortemente conectados. Neste tipo de representação, é possível incluir também o escalonamento por ciclos de execução, que facilita a identificação de dependências entre as operações.

O segundo tipo de grafo, apresentado na [Figura 6.8\(b\)](#), possui nós mais detalhados e mostra alguns parâmetros internos da instância de cada componente. A partir desta visualização é possível construir o código VHDL completo da arquitetura, pois todos os nomes e tipos usados no código são representados. Os pinos de entrada/saída são representados por formas retangulares e as constantes por elipses. Os componentes registrados, possuem coloração cinza enquanto os não registrados são brancos. Dentro de cada componente as portas de entrada são apresentadas na parte superior e as de saída na parte inferior. A coloração e a nomenclatura (nomes com colchetes) são usadas nas arestas para diferenciar sinais simples (STD_LOGIC) de sinais compostos (STD_LOGIC_VECTOR), embora sinais compostos possam ter somente um bit. As arestas recorrentes são representadas de forma tracejada.

6.6 Interface Gráfica do Usuário

Para demonstrar as potencialidades da linguagem na exploração de níveis de *pipelining* das arquiteturas foi desenvolvido o protótipo para uma interface gráfica do usuário, apresentado na [Figura 6.9](#). O código LALP escrito no painel esquerdo da ferramenta é compilado e gera uma visualização do lado direito, permitindo que pequenos parâmetros do programa sejam modificados quando não há a necessidade de simulações dos resultados imediatamente. Além de apresentar graficamente a arquitetura, esta interface permite a geração de código VHDL e a exportação do grafo na linguagem descrita anteriormente.

Figura 6.9: Interface gráfica do usuário

Resultados

NESTE capítulo, são apresentados os resultados experimentais obtidos com o compilador ALP para mapear algoritmos, com diferentes graus de complexidade, em FPGAs. Sempre que possível, as implementações baseadas em ALP foram comparadas à implementações obtidas com outras ferramentas. O objetivo das comparações foi apenas verificar o desempenho das técnicas desenvolvidas, e não teve a intenção de concluir a eficácia de outras ferramentas. Para as descrições em LALP, realizadas em um estágio posterior, os mesmos algoritmos apresentados em linguagem C foram usados. Isto é importante, pois reflete o grau de complexidade para uma possível conversão automática entre C e LALP. Memórias *on-chip* (BRAM¹) foram usadas para armazenar os dados de entrada e saída para todos os experimentos. Para cada memória, foi permitido apenas uma leitura/gravação por ciclo de *clock*. Os resultados poderiam ser ainda melhores, caso fossem usadas memórias com múltiplas portas para leituras/gravações simultâneas.

O conjunto de *benchmarks* usado possui complexidade variada e foi expandido à medida que o trabalho se desenvolveu. Desta forma, optou-se por descrever os resultados em ordem

¹Block Random Access Memory)

cronológica, para que se possa ter uma ideia da evolução da pesquisa. Foram usadas diferentes ferramentas e plataformas de execução, tanto de *hardware* quanto de *software*, para comparação dos resultados. Na [Seção 7.1](#) é apresentado o conjunto de *benchmarks* e suas características. Na [Seção 7.2](#) são apresentados resultados preliminares, obtidos na comparação de dois exemplos simples, especificados manualmente. Na [Seção 7.3](#) são apresentados resultados para dois exemplos mais complexos, já com o uso da linguagem para geração das arquiteturas. Na [Seção 7.4](#) são apresentados resultados obtidos com o uso da linguagem em um conjunto de *benchmarks*. Na [Seção 7.5](#) é discutido o impacto dos algoritmos de escalonamento automático. Na [Seção 7.6](#) são apresentadas novas comparações para as arquiteturas obtidas no estágio final de desenvolvimento do trabalho. Na [Seção 7.7](#) é apresentada uma breve comparação com processadores embarcados. Na [Seção 7.8](#) são discutidas as possibilidades de exploração do espaço de projeto usando LALP. Na [Seção 7.9](#) é realizada uma breve discussão sobre o consumo de potência e energia nos dispositivos FPGA.

7.1 Conjunto de *Benchmarks*

Além dos exemplos já apresentados, outros exemplos presentes em repositórios de *benchmarks* públicos foram considerados. A lista é composta pelos algoritmos de imagem *FDCT* e *Sobel* ([TexasURL, 2003b](#)); os algoritmos *Autocorrelation*, *Dotprod*, *Max* e *Vector Sum*, comumente usados em DSPs ([TexasURL, 2003a](#)); os algoritmos de áudio *ADPCM Coder* e *ADPCM Decoder* ([Guthaus et. al., 2001](#)); além dos algoritmos *Bubble Sort* e *Pop Count*, usados para demonstração pela ferramenta [C-to-VerilogURL \(2009\)](#); e o algoritmo *Fibonacci*, por apresentar uma dependência de dois níveis entre as iterações.

Na [Tabela 7.1](#) é apresentada a lista dos *benchmarks* e as ferramentas com as quais cada um deles foi gerado. O compilador C2H da Altera ([AlteraURL, 2008a](#)) foi usado nos primeiros testes, no entanto, os aceleradores obtidos com esta ferramenta precisam ser acoplados ao processador Nios II. Não foram realizadas experimentos com os demais *benchmarks* neste compilador. A ferramenta SPARK ([Gupta et. al., 2004b](#)) possui vasta documentação e foi usada com alguns exemplos, mas o VHDL gerado possui arranjos nas interfaces que impedem que os componentes sejam instanciados para simulação. Além disso, o SPARK assume que todas

as posições do arranjo estão disponíveis ao mesmo tempo. Isso faz com que sejam criados inúmeros pinos de entrada/saída, incompatíveis com a quantidade disponível nos dispositivos.

Apesar da escassa documentação, a ferramenta C-to-Verilog ([C-to-VerilogURL, 2009](#)) foi capaz de compilar a maioria dos exemplos, com exceção do *FDCT*. A mensagem informa que o GCC compilou o programa de entrada, mas que o compilador não foi capaz de gerar o código Verilog correspondente.

A maioria dos exemplos presentes no conjunto de *benchmarks* avaliado não pôde ser compilada com o ROCCC ([Buyukkurt et. al., 2006](#); [Guo et. al., 2005](#)) em função das restrições de entrada do código C. Alguns exemplos tiveram que ser modificados e outros geraram resultados incorretos.

Tabela 7.1: Lista dos *benchmarks* por ferramenta

<i>Benchmark</i>	Altera C2H	SPARK	C2Verilog	ROCCC	LALP
<i>ADPCM Coder</i>		✓	✓		✓
<i>ADPCM Decoder</i>		✓	✓		✓
<i>Autocorrelation</i>			✓		✓
<i>Bubble Sort</i>			✓		✓
<i>Dotprod</i>	✓	✓	✓		✓
<i>FDCT</i>					✓
<i>Fibonacci</i>			✓	✓	✓
<i>Max</i>	✓	✓	✓		✓
<i>Pop Count</i>			✓		✓
<i>Sobel</i>			✓	✓	✓
<i>Vector Sum</i>			✓	✓	✓

Na [Tabela 7.2](#) são apresentadas algumas características dos *benchmarks*. Em geral, a tradução dos programas de C para LALP causou um leve aumento no número de linhas de código, com algumas excessões. A abordagem baseada em componentes para a geração de VHDL, comum a muitos compiladores de *hardware*, resulta em longas descrições estruturais. Embora descrições comportamentais em VHDL possam ser bem menores, este fator não influencia na eficiência do *hardware* resultante, seja em termos de desempenho ou área ocupada no FPGA. Adicionalmente, a [Tabela 7.2](#) apresenta o número de *loops*, arranjos e testes condicionais para cada exemplo. Nos exemplos em que são encontrados dois *loops* estes estão sempre aninhados. O *FDCT* possui dois *loops* aninhados e um em sequência.

Tabela 7.2: Características dos *benchmark* usados

<i>Benchmark</i>	Linhas de código			<i>Loops</i>	<i>Arrays</i>	<i>Ifs</i>
	C	LALP	VHDL			
<i>ADPCM Coder</i>	83	71	1718	1	4	10
<i>ADPCM Decoder</i>	70	60	1352	1	4	9
<i>Autocorrelation</i>	16	29	470	2	2	0
<i>Bubble Sort</i>	15	31	418	2	1	1
<i>Dotprod</i>	10	18	225	1	1	0
<i>FDCT</i>	145	175	5290	3	3	0
<i>Fibonacci</i>	10	19	202	1	1	0
<i>Max</i>	10	18	225	1	1	1
<i>Pop Count</i>	11	118	2294	2	2	0
<i>Sobel</i>	36	52	1298	1	2	3
<i>Vector Sum</i>	12	20	234	1	1	0

7.2 Resultados Preliminares

Os primeiros resultados obtidos com as técnicas pesquisadas foram avaliados em um FPGA da família Stratix da Altera (EP1S10F780C6) usando o Quartus II 6.1 para síntese. Foram usados dois exemplos simples: *Dotprod*, listado anteriormente no [Algoritmo 5.8](#); e *Max*, listado no [Algoritmo 7.1](#). O primeiro calcula e retorna a soma do produto de dois vetores de inteiros e o segundo retorna o maior valor encontrado em um vetor de inteiros.

Algoritmo 7.1: Exemplo *Max* descrito em C

```

1  #define N 2048
2
3  int max() {
4      int v[N];
5      int i, maxval = 0;
6      for (i=0; i<N; i++)
7          if (maxval > v[i])
8              maxval = v[i];
9      return maxval;
10 }
```

Para cada exemplo, foi gerado manualmente um *hardware* com a técnica ALP, outro com o compilador SPARK, um sistema de *software* executando o código em um processador Nios II e outro híbrido, usando o compilador C2H da Altera para acelerar a plataforma de *software*. Na [Figura 7.1](#) é apresentada a latência, em número de ciclos de *clock* necessárias para a execução dos algoritmos em cada plataforma, considerando 10, 100 e 1000 iterações para cada exemplo. As arquiteturas geradas com ALP permitiram executar os exemplos *Dotprod* e *Max*

com quase o mesmo número de ciclos necessários para percorrer os vetores, 1003 e 1002 ciclos respectivamente, para 1000 iterações.

Figura 7.1: Número de ciclos de *clock* necessários para execução

A plataforma de *software* e a híbrida operaram a 100 MHz, limitação imposta pelo processador Nios II neste dispositivo, e ocuparam uma grande área (mais de três mil elementos lógicos), tornando a comparação injusta em termos de frequência e recursos consumidos. O *hardware* obtido com o compilador SPARK não pode ser sintetizado, pois este compilador assume que todos os dados do código estão disponíveis, resultando em um número de pinos de E/S superior ao disponível no FPGA. A quantidade de ciclos neste caso, foi estimada por análise da máquina de estados finitos do código. Foram medidas ainda as frequências máximas e os recursos ocupados no FPGA para o mesmo número de iterações, com dados de 16 e 32 bits, para as arquiteturas obtidas com ALP. Os dados apurados são apresentados na [Tabela 7.3](#).

A frequência máxima de execução é semelhante nos dois exemplos e praticamente não varia com o aumento do número de bits de dados ou iterações, bem como o número de elementos lógicos usados. A quantidade de memória ocupada nos exemplos varia diretamente com o

Tabela 7.3: Frequência e recursos no FPGA Stratix (EP1S10F780C6)

<i>Benchmark</i>	Bits	Dados	Freq. Max. (MHz)	Elem. Lógicos	Memória (Kbits)	DSP (9-bits)
<i>Dotprod</i>	16	10	224,87	36	0	2
		100	180,21	39	4	2
		1000	178,16	45	32	2
	32	10	134,86	52	0	8
		100	134,86	55	8	8
		1000	134,86	61	64	8
<i>Max</i>	16	10	179,82	56	0	0
		100	128,78	54	2	0
		1000	111,21	60	16	0
	32	10	184,37	55	0	0
		100	112,08	85	4	0
		1000	107,25	92	32	0

número de vetores de dados, dois para o *Dotprod* e um *Max*, o número de bits e o número de iterações. O exemplo *Max* não realiza nenhuma operação aritmética e portanto não necessita de blocos DSP. Para o exemplo *Dotprod* o número de blocos DSP ocupados varia com a número de bits da multiplicação (dois blocos de 9 bits para dados de 16 bits e oito para dados de 32 bits).

Embora muito simples, estes exemplos puderam demonstrar a potencialidade da técnica (Menotti *et. al.*, 2007) e a dificuldade das ferramentas comparadas em obter escalonamentos, e consequentemente, arquiteturas otimizadas.

7.3 Resultados com *ADPCM Coder/Decoder*

Em uma nova etapa da pesquisa, foi desenvolvida a linguagem para facilitar a construção das arquiteturas. Conforme descrito nos capítulos anteriores, os algoritmos *ADPCM* possuem características interessantes para a avaliação da geração de arquiteturas *pipelining*, pois possuem dependências circulares em suas instruções. Para comparação, foram geradas arquiteturas para os dois algoritmos usando as ferramentas C-to-Verilog (C2Verilog), SPARK e com o compilador ALP (Menotti *et. al.*, 2010b). Na Tabela 7.4 são apresentadas as frequências máximas de operação, bem como os recursos ocupados no FPGA Stratix III (EP3SE50F484C2) usando o Quartus II 9.0 da Altera para implementação.

Para as arquiteturas geradas com o SPARK, foi possível completar o processo de síntese e verificar a frequência e os recursos ocupados, mas não a implementação completa no disposi-

Tabela 7.4: Frequência e recursos no FPGA Stratix III (EP3SE50F484C2)

<i>Benchmark</i>	<i>Resultados</i>	<i>C2Verilog</i>	<i>SPARK</i>	<i>ALP</i>
ADPCM Coder	Freq. Max.	299,4	342,1	293,6
	Throughput	0,025	0,023~0,028	0,100
	ALUTs	572	726	460
	Registers	1033	298	465
ADPCM Decoder	Freq. Max.	466,0	351,8	438,4
	Throughput	0,040	0,031~0,043	0,332
	ALUTs	471	502	331
	Registers	757	333	409

tivo. Isso ocorre sempre que são usadas memórias nos programas, pois o compilador assume que todas as posições de memória estão disponíveis ao mesmo tempo. Tal abordagem resulta em um código VHDL com muitos pinos de entrada e saída que não podem ser mapeados no dispositivo alvo por exceder os pinos disponíveis. Por análise da máquina de estados gerada é possível estimar o *throughput* para o melhor e para o pior caso. Nota-se que estes valores aproximam-se muito os obtidos com o compilador C-to-Verilog.

Em termos de recursos ocupados, apresentados na [Figura 7.2](#), por sua complexidade o *ADPCM Coder* requer mais recursos do que o *ADPCM Decoder*, atingindo também uma frequência máxima de operação menor. A variação é consistente e pôde ser observada nas arquiteturas geradas com as três ferramentas.

O ganho em tempo de execução (*speedup*), apresentado na [Figura 7.3](#), teve como referência o tempo medido durante a execução dos algoritmos em um processador IBM PowerPC 405 (integrado em um FPGA Xilinx Virtex-II Pro) a 300 MHz e usando um barramento de sistema a 100 MHz. Os códigos foram gerados com o compilador GCC usando parâmetros para melhor desempenho. As latências medidas podem ser consideradas mínimas, pois foram usadas memórias *on-chip* para dados e instruções.

É possível notar que o *throughput* das arquiteturas obtidas com ALP é significativamente mais alto do que os obtidos com C-to-Verilog e SPARK. Isto se deve ao fato de que o compilador ALP pôde gerar melhores implementações em *pipelining*, necessitando de um número menor de ciclos para a execução total do código. As técnicas permitiram ganhos no tempo de execução sobre os arquiteturas obtidas com C-to-Verilog e SPARK na ordem de 4x e 8x, para os exemplos *ADPCM Coder* e *ADPCM Decoder*, respectivamente. Relativo ao tempo de execução obtido com o PowerPC o desempenho das arquiteturas geradas com ALP é equivalente ao de um

Figura 7.2: Comparação dos recursos ocupados no FPGA Stratix III (EP3SE50F484C2)

processador PowerPC hipotético executando a 2GHz e 7,5GHz para *ADPCM Coder* e *ADPCM Decoder*, respectivamente.

7.4 LALP Comparado ao C-to-Verilog

Para comprovar a eficiência de LALP e das técnicas de compilação foram descritos *benchmarks* representativos com duas metodologias. Os exemplos foram compilados com ALP e com a ferramenta [C-to-VerilogURL \(2009\)](#), disponível *on-line*.

Na [Tabela 7.5](#) são apresentadas as frequências máximas de operação e os recursos ocupados no FPGA Virtex 5 (XC5VLX303FF324) da Xilinx, usando a ferramenta ISE 9.2i para síntese.

São descritos o número de *flip-flops* (FFs), *lookup tables* (LUTs), *Slices* e blocos DSP, bem como a frequência máxima de operação para as arquiteturas obtidas com o compilador C-to-Verilog e ALP, respectivamente, seguidos de uma comparação entre eles. Os exemplos obtidos com LALP consomem consideravelmente menos recursos, nunca ultrapassando a metade dos gerados com C-to-Verilog, apenas 26% na média comparando o número de *Slices*

Figura 7.3: Ganho no tempo de execução (*speedup*)

Tabela 7.5: Frequência e recursos no FPGA Virtex 5 (XC5VLX303FF324)

Benchmark	Resultados obtidos com C-to-Verilog					Resultados obtidos com LALP					Melhorias (LALP / C2Verilog)	
	FFs	LUTs	Slices	DSPs	Freq. Max. (MHz)	FFs	LUTs	Slices	DSPs	Freq. Max. (MHz)	Slices	Freq. Max. (MHz)
ADPCM Coder	1078	711	827	0	427,92	605	783	279	0	383,30	0,34	0,90
ADPCM Decoder	743	603	590	0	446,25	650	499	251	0	383,30	0,43	0,86
Bubble Sort	2353	2471	971	0	239,45	219	105	79	0	353,16	0,08	1,47
Dotprod	758	578	285	3	249,36	97	69	32	3	213,14	0,11	0,85
Fibonacci	73	108	69	0	297,81	104	41	30	0	505,08	0,43	1,70
Max	496	392	164	0	435,90	50	39	20	0	484,97	0,12	1,11
Pop Count	1023	872	384	0	411,22	350	215	115	0	503,73	0,29	1,22
										Média	0,26	1,16

ocupados. Em alguns casos, o número de recursos consumidos pela ferramenta comparada é muito maior. A documentação da ferramenta oferece uma possível explicação para isso, devido ao número de acessos à memória o compilador pode gerar arquiteturas não otimizadas.

Com relação a frequência de operação, os exemplos obtidos com LALP apresentaram em média 16% de ganho em relação a C-to-Verilog. No entanto, este parâmetro é extremamente dependente do número de estágios de *pipelining* inseridos em cada arquitetura. Além disso, todos os exemplos gerados com LALP tiveram um tempo de execução menor, em função do número

de ciclos de *clock* necessários para completar a computação. Na Figura 7.4 são apresentados os tempos de execução normalizados (considerando a frequência máxima e o número de ciclos requeridos). Foram obtidos *speedups* de 1,43 a 7,06 nas comparações com média de 4,01.

Estes resultados comprovaram a eficiência da técnica na obtenção de arquiteturas otimizadas para execução em *pipelining*. Mesmo nos exemplos em que a frequência máxima obtida foi menor, o número de ciclos requeridos para a execução foi sempre inferior, resultando em um *throughput* mais alto em todos os exemplos.

Figura 7.4: Tempo de execução normalizado em relação a LALP

7.5 Impacto dos Algoritmos de Escalonamento

Os resultados descritos até aqui referem-se a arquiteturas geradas automaticamente a partir de descrições em LALP. No entanto, muitas diretivas de sincronização foram necessárias para se obter o funcionamento correto. Para que a descrição do programa em LALP fosse a mais próxima possível das linguagens de alto nível, foram desenvolvidos algoritmos para inferir automaticamente estas diretivas. A seguir é descrito o impacto da aplicação dos algoritmos de

escalonamento, balanceamento e sincronização, apresentados no [Capítulo 6](#), em termos de área ocupada no dispositivo e frequência máxima de operação.

Para se ter uma ideia do nível de automatização alcançado, o [Algoritmo 7.2](#) apresenta o código LALP para o algoritmo *ADPCM Decoder* com as diretivas comentadas. Vinte e quatro diretivas de sincronização são necessárias para geração da arquitetura correta. Já com os algoritmos automáticos, são necessárias apenas três. Embora o número de ciclos por iteração deva ser informado no contador, o compilador é capaz de calcular este número e exibir uma advertência caso ele seja omitido pelo usuário.

Algoritmo 7.2: Exemplo *ADPCM Decoder* descrito em LALP com diretivas de sincronização

```

1  const DATASIZE = 1024;
2
3  typedef fixed(32,1) int;
4  typedef fixed(1,0) bit;
5
6  adpcm_decoder(in bit init, out int output, out bit done) {
7      len.clk_en = init;
8      counter (len=0; len<DATASIZE; len++@3);
9      indata.address = i;
10     inputbuffer = indata.data_out when !bufferstep & (len.step@1);
11     delta = bufferstep ? inputbuffer & 0xf : (inputbuffer >> 4) & 0xf; /*
12         when len.step@2; */
13     i += 1 when !bufferstep & (len.step@2);
14     sign = delta & 8; // when len.step@3;
15     indexTable.address = delta;
16     delta2 = delta & 7; // when len.step@3;
17     bufferstep = !bufferstep; // when len.step@3;
18     index2 = index + indexTable.data_out; // when len.step@4;
19     stepSizeTable.address = index;
20     index3 = index2 < 0 ? 0 : index2; // when len.step@5;
21     step = stepSizeTable.data_out; // when len.step@5;
22     index = index3 > 88 ? 88 : index3; // when len.step@6;
23     vpdiff = step >> 3; // when len.step@6;
24     vpdiff2 = ((delta2/*@3*/) & 1) > 0 ? vpdiff + (step >> 2) : vpdiff; /*
25         when len.step@7; */
26     vpdiff3 = ((delta2/*@4*/) & 2) > 0 ? vpdiff + (step >> 1) : vpdiff2; /*
27         when len.step@8; */
28     vpdiff4 = ((delta2/*@5*/) & 4) > 0 ? vpdiff + (step) : vpdiff3; /*
29         when len.step@9; */
30     valpred2 = (sign) > 0 ? valpred - vpdiff4 : valpred + vpdiff4; /*
31         when len.step@10; */
32     valpred3 = valpred2 > 32767 ? 32767 : valpred2; // when len.step@11;
33     valpred = valpred3 < -32768 ? -32768 : valpred3; // when len.step@12;
34     outdata.address = len/*@12*/;
35     outdata.data_in = valpred; // when len.step@13;
36     output = outdata.data_out;
37     done = len.done/*@11*/;
38 }

```

Na [Tabela 7.6](#) são apresentadas, para cada exemplo, o número de diretivas de sincronização necessárias para a descrição em LALP com processo manual de sincronização, o número de diretivas necessárias com o uso dos algoritmos automáticos, e o número de diretivas adicionadas automaticamente por estes algoritmos. Alguns algoritmos possuem uma implementação alternativa com reúso dos dados para minimizar os acessos à memória. O exemplo *Sobel*◇ foi incluído após o uso dos algoritmos automáticos. Nos exemplos *Dotprod*, *Fibonacci*◇, *Pop Count* e *Vector Sum* nenhuma diretiva é necessária para geração das arquiteturas corretas. As arquiteturas obtidas para os exemplos *Autocorrelation*, *Bubble Sort* e *Fibonacci* necessitam que todas as diretivas sejam informadas para a geração de resultados corretos.

Tabela 7.6: Diretivas de sincronização

<i>Benchmark</i>	Manual	Automático	
	Inseridas	Inseridas	Adicionadas
<i>ADPCM Coder</i>	28	4	28
<i>ADPCM Decoder</i>	24	3	22
<i>Autocorrelation</i>	3	3	-
<i>Bubble Sort</i>	5	5	-
<i>Dotprod</i>	2	-	2
<i>FDCT</i>	119	39	118
<i>Fibonacci</i>	5	5	-
<i>Fibonacci</i> ◇	1	-	1
<i>Max</i>	2	1	1
<i>Pop Count</i>	1	-	1
<i>Sobel</i>	26	16	13
<i>Sobel</i> ◇	-	12	15
<i>Vector Sum</i>	3	-	3

◇ versão com reúso de dados

Em geral, são inseridas mais diretivas pelo algoritmo do que durante o processo manual, isso ocorre porque em muitos casos é possível omitir certas diretivas e permitir que uma operação seja calculada o tempo todo, mesmo que seu valor não seja necessário posteriormente. O compilador usa uma estratégia conservadora e, portanto, insere restrições para que cada operação seja efetuada apenas no momento necessário.

Na [Tabela 7.7](#) são apresentadas as frequências máximas de operação e os recursos ocupados no FPGA Virtex 6 (XC6VLX75T3FF484) da Xilinx, usando a ferramenta ISE 11.4 para síntese. Na tabela são comparados os dados obtidos com a sincronização manual e os obtidos com os algoritmos automáticos, descritos no [Capítulo 6](#). É possível notar que o impacto dos algoritmos é muito pequeno em termos de recursos, e que o desempenho obtido também não varia, já

que a arquitetura resultante é praticamente a mesma. Apenas o exemplo *FDCT* sofreu um aumento significativo em termos de recursos ocupados. Este exemplo possui muitas operações executadas em sequência, e que não necessitam de qualquer diretiva de sincronização na versão manual. Conforme mencionado, os algoritmos automáticos são conservadores e adicionam sinais de controle que garantam que as operações serão executadas apenas nos ciclos exatos.

Tabela 7.7: Frequência e recursos no FPGA Virtex 6 (XC6VLX75T3FF484)

<i>Benchmark</i>	Escalonamento manual				Algoritmos automáticos				Comparação			
	FFs	LUTs	Slices	DSPs	Freq. Max. (MHz)	FFs	LUTs	Slices	DSPs	Freq. Max. (MHz)	Slices	Freq. Max. (MHz)
<i>ADPCM Coder</i>	635	868	246	0	523,37	798	916	257	0	535,02	1,04	1,02
<i>ADPCM Decoder</i>	572	625	190	0	535,02	543	596	181	0	535,02	0,95	1,00
<i>Autocorrelation</i>	312	161	44	0	669,70	312	161	44	0	669,70	1,00	1,00
<i>Bubble Sort</i>	225	197	61	0	423,89	225	197	61	0	423,89	1,00	1,00
<i>Dotprod</i>	93	79	23	3	530,79	93	79	23	3	530,79	1,00	1,00
<i>FDCT</i>	3900	3411	943	56	483,68	5960	5110	1322	56	483,68	1,40	1,00
<i>Fibonacci</i>	113	101	31	0	514,03	113	101	31	0	514,03	1,00	1,00
<i>Fibonacci</i> ◇	107	81	23	0	623,36	107	81	23	0	623,36	1,00	1,00
<i>Max</i>	88	67	18	0	464,17	88	67	18	0	464,17	1,00	1,00
<i>Pop Count</i>	30	24	11	0	849,62	30	24	11	0	849,62	1,00	1,00
<i>Sobel</i>	39	34	14	0	589,38	45	36	14	0	589,38	1,00	1,00
<i>Sobel</i> ◇	39	34	14	0	589,38	45	36	14	0	589,38	1,00	1,00
<i>Vector Sum</i>	177	113	34	0	546,54	177	113	34	0	546,54	1,00	1,00
◇ versão com reúso de dados										Média	1,04	1,00

7.6 LALP Comparado ao ROCCC e C-to-Verilog

Após o desenvolvimento dos algoritmos de escalonamento, novas comparações foram realizadas com os compiladores ROCCC e C-to-Verilog. Na [Tabela 7.8](#) são apresentados os dados consolidados para as versões geradas com os algoritmos automáticos de escalonamento. Foi usado o dispositivo Virtex 6 (XC6VLX75T3FF484) da Xilinx e a ferramenta ISE 11.4 para síntese. A coluna **Dados** refere-se ao número de iterações usado em cada *benchmark*. A coluna **Latência** refere-se ao número total de ciclos de *clock* necessários para completar a computação. A coluna **Throughput** representa o número de iterações processadas por ciclo de *clock*. Cabe ressaltar que as implementações foram obtidas diretamente a partir de descrições em LALP sem mudanças manuais na estrutura dos grafos (por meio de descrições em LALP-S).

Os algoritmos cujo *throughput* se aproxima de um são aqueles que possuem apenas um *loop* e nenhuma dependência entre as operações de iterações diferentes, permitindo praticamente o cálculo de uma iteração por ciclo de *clock*. Números inferiores são obtidos para algoritmos com vários *loops*, especialmente se estes forem aninhados, ou para aqueles com dependências entre

Tabela 7.8: LALP: frequência e recursos no FPGA Virtex 6 (XC6VLX75T3FF484)

<i>Benchmark</i>	<i>Dados</i>	<i>FFs</i>	<i>LUTs</i>	<i>Slices</i>	<i>DSPs</i>	<i>Freq Max. (MHz)</i>	<i>Latência</i>	<i>Tempo de Execução (us)</i>	<i>Throughput</i>
<i>ADPCM Coder</i>	1024	798	916	257	0	535,017	12288	22,97	0,083
<i>ADPCM Decoder</i>	1024	543	596	181	0	535,017	3089	5,77	0,331
<i>Autocorrelation</i>	1600	312	161	44	0	669,703	3010	4,49	0,532
<i>Bubble Sort</i>	32	225	197	61	0	423,889	4224	9,96	0,008
<i>Dotprod</i>	2048	93	79	23	3	530,786	2050	3,86	0,999
<i>FDCT</i>	640	5960	5110	1322	56	483,676	1430	2,96	0,448
<i>Fibonacci</i>	32	113	101	31	0	514,033	96	0,19	0,333
<i>Fibonacci</i> ◇	32	107	81	23	0	623,364	32	0,05	1,000
<i>Max</i>	2048	88	67	18	0	464,165	2050	4,42	0,999
<i>Pop Count</i>	1024	30	24	11	0	849,618	1029	1,21	0,995
<i>Sobel</i>	100	45	36	14	0	589,379	643	1,09	0,156
<i>Sobel</i> ◇	100	777	627	177	0	423,711	109	0,26	0,917
<i>Vector Sum</i>	2048	177	113	34	0	546,538	2050	3,75	0,999

◇ versão com reúso de dados

as iterações. O menor *throughput* obtido foi o do exemplo *Bubble Sort*, por sua complexidade quadrática com notória ineficiência.

A abordagem usada com LALP permitiu a geração de arquiteturas otimizadas, capazes de operar em altas frequências e com o maior *throughput* possível. Com exceção das versões modificadas para reúso dos dados e do exemplo *Pop Count*, os códigos foram estritamente baseados no original em C. Para se obter melhor desempenho é possível realizar mudanças no código ou adotar memórias com múltiplas portas.

Os resultados obtidos com o ROCCC são apresentados na [Tabela 7.9](#), embora a ferramenta não pudesse compilar a maioria dos exemplos. Para os exemplos *Fibonacci* e *Sobel* a ferramenta foi capaz de realizar o reúso dos dados. Nestes casos, os resultados obtidos foram melhores do que aqueles atingidos com LALP sem reúso dos dados. As implementações LALP com reúso dos dados atingiram desempenhos melhores do que as do ROCCC. Para os exemplos mostrados na [Tabela 7.9](#), o compilador LALP obteve um *speedup* médio de 1,17x. Com relação aos recursos do dispositivo, as implementações LALP ocuparam em média 66% menos *Slices* do que o ROCCC.

Tabela 7.9: ROCCC: frequência e recursos no FPGA Virtex 6 (XC6VLX75T3FF484)

<i>Benchmark</i>	<i>Dados</i>	<i>FFs</i>	<i>LUTs</i>	<i>Slices</i>	<i>DSPs</i>	<i>Freq Max. (MHz)</i>	<i>Latência</i>	<i>Tempo de Execução (us)</i>	<i>Throughput</i>	<i>Melhorias (LALP / ROCCC)</i>	
										<i>Slices</i>	<i>Speedup</i>
<i>Fibonacci</i>	32	130	208	54	0	364,786	33	0,09	0,970	0,57	0,48
<i>Fibonacci</i> ◇	32	130	208	54	0	364,786	33	0,09	0,970	0,43	1,76
<i>Sobel</i>	100	1371	1308	351	0	313,587	114	0,36	0,877	0,04	0,33
<i>Sobel</i> ◇	100	1371	1308	351	0	313,587	114	0,36	0,877	0,50	1,41
<i>Vector Sum</i>	2048	554	751	200	0	291,979	2052	7,03	0,998	0,17	1,87
◇ versão com reúso de dados									Média	0,34	1,17

Os resultados obtidos com o compilador C-to-Verilog são apresentados na [Tabela 7.10](#). Neste caso, foi possível avaliar e comparar quase todos os *benchmarks* considerados. Assim como o ROCCC, o C-to-Verilog também foi capaz de realizar o reúso dos dados para o *Fibonacci* e parcialmente para o *Sobel*. As implementações LALP ocuparam em média 47% menos *Slices* no dispositivo do que as implementações obtidas com o C-to-Verilog. O *speedup* obtido com LALP comparado a esta ferramenta foi em média 5,93x. É importante ressaltar que em alguns casos o número de recursos requerido por esta ferramenta foram muito maiores. Uma possível explicação para isto é que o número de acessos à memória, presentes no código C original, possa gerar estruturas não otimizadas. A documentação da ferramenta orienta a minimizar os acessos em arranjos com índices diferentes, e portanto, seriam necessárias mudanças no código original.

Tabela 7.10: C2Verilog: frequência e recursos no FPGA Virtex 6 (XC6VLX75T3FF484)

<i>Benchmark</i>	<i>Dados</i>	<i>FFs</i>	<i>LUTs</i>	<i>Slices</i>	<i>DSPs</i>	<i>Freq Max. (MHz)</i>	<i>Latência</i>	<i>Tempo de Execução (us)</i>	<i>Melhorias</i>		
									<i>Throughput (LALP / C2Verilog)</i>		
									<i>Slices</i>	<i>Speedup</i>	
<i>ADPCM Coder</i>	1024	990	701	522	0	515,623	40963	79,44	0,025	0,49	3,46
<i>ADPCM Decoder</i>	1024	750	611	528	0	529,633	25346	47,86	0,040	0,34	8,29
<i>Autocorrelation</i>	1600	6296	6729	1738	3	161,009	10631	66,03	0,151	0,03	14,69
<i>Bubble Sort</i>	32	2353	4904	1369	0	261,766	4098	15,66	0,008	0,04	1,57
<i>Dotprod</i>	2048	744	582	476	0	303,261	14348	47,31	0,143	0,05	12,25
<i>Fibonacci</i>	32	45	38	11	0	688,208	34	0,05	0,941	2,82	0,26
<i>Fibonacci</i> ◇	32	45	38	11	0	688,208	34	0,05	0,941	2,09	0,96
<i>Max</i>	2048	490	474	127	0	521,839	6151	11,79	0,333	0,14	2,67
<i>Pop Count</i>	1024	1023	1036	328	0	630,456	4107	6,51	0,249	0,03	5,38
<i>Sobel</i>	100	4616	6322	1716	0	433,149	1730	3,99	0,058	0,01	3,66
<i>Sobel</i> ◇	100	4616	6322	1716	0	433,149	1730	3,99	0,058	0,10	15,53
<i>Vector Sum</i>	2048	664	660	169	0	665,38	6150	9,24	0,333	0,20	2,46
◇ versão com reúso de dados									Média	0,53	5,93

Na [Figura 7.5](#) é apresentado o gráfico de tempo de execução normalizado, comparando as arquiteturas obtidas com LALP às obtidas com ROCCC e C-to-Verilog (C2Verilog). O *speedup* relativo ao C-to-Verilog é em média 5,9 vezes maior e ao ROCCC 1,2 vezes maior. Embora as arquiteturas obtidas com LALP não ofereça em todos os casos a maior frequência de operação, os ganhos em função do *throughput* permitem que se obtenha um tempo de execução menor. As únicas exceções são os exemplos *Fibonacci* e *Sobel* que tiveram, inicialmente, desempenho pior. Foram necessárias novas implementações destes algoritmos para que se atingisse resultados satisfatórios.

Na [Figura 7.6](#) é apresentado o gráfico com o *throughput* obtido para cada exemplo, comparando as arquiteturas obtidas com LALP às obtidas com ROCCC e C-to-Verilog (C2Verilog).

Figura 7.5: Tempo de execução normalizado em relação a LALP

No caso das arquiteturas geradas com LALP, o *throughput* é limitado apenas nos casos de leituras e gravações a um mesmo arranjo ou de dependências entre as iterações do programa usado. O problema das dependências nestes exemplos é demonstrado na [Figura 5.4](#) e na [Figura 5.5](#), no [Capítulo 5](#).

O desempenho obtido com implementações em LALP foi em média superior àqueles obtidos com o ROCCC e o C-to-Verilog, seja em termos de *throughput* ou frequência máxima de operação, mesmo ocupando menor área no dispositivo. Os mesmos ganhos foram observados nas comparações anteriores, usando o SPARK e o C2H, justificando essa metodologia alternativa quando os resultados com ferramentas de síntese de alto nível não são satisfatórios.

A economia de recursos de *hardware* obtida com as implementações em LALP é justificada pelas técnicas agressivas na geração do *pipeline*. As arquiteturas são capazes de atingir altos graus de paralelismo sem a necessidade de redundância de recursos para a construção do epílogo, *kernel* e prólogo. Estes estágios estão presentes na execução dos *loops*, mas não estão presentes como estágios específicos no *hardware*, conforme discutido em ([Cardoso, 2005](#)). Em

Figura 7.6: Comparação do *throughput* em relação ao ROCCC e C2Verilog

LALP, a criação de estágios e a atribuição de operações a eles não é necessária. Isto consumiria mais recursos já que algumas operações precisariam estar em vários estados para melhorar o nível de paralelismo.

7.7 LALP Comparado a Processadores Embarcados

A seguir, são apresentados alguns dados da simulação ou execução dos *benchmarks* em processadores embarcados. Na [Tabela 7.11](#) os tempos de execução obtidos com LALP são comparados aos processadores *hardcore* ARM e PowerPC, e ao *softcore* Nios II da Altera.

Para a plataforma ARM, os dados foram obtidos por meio de simulações com o SimpleScalarARM ([Austin et. al., 2002](#); [SimpleScalarURL, 2004](#)). Os códigos foram compilados com o GCC usando o nível de otimização três. O tempo de execução foi baseado em um processador ARM920T operando a 200MHz, assumindo, de forma otimista, que uma instrução fosse completada a cada ciclo de *clock*.

A plataforma PowerPC teve como referência o tempo medido durante a execução dos algo-

Tabela 7.11: LALP comparado a microprocessadores embarcados

Benchmark	Dados	LALP	ARM (200 MHz)	PowerPC (100 MHz)		Nios II (50 MHz)		
		Tempo (us)	Tempo (us)	LALP Speedup	Tempo (us)	LALP Speedup	Tempo (us)	LALP Speedup
ADPCM Coder	1024	22,97	278,34	12,12	1936,44	84,31	14917,75	649,52
ADPCM Decoder	1024	5,77	216,03	37,42	1465,36	253,80	11246,80	1947,95
Autocorrelation	1600	4,49	178,86	39,80	-	-	4838,34	1076,50
Bubble Sort	32	9,96	46,87	4,70	-	-	2320,97	232,92
Dotprod	2048	3,86	72,65	18,81	656,26	169,92	5810,23	1504,38
FDCT	640	2,96	222,30	75,19	885,56	299,53	7274,47	2460,48
Fibonacci	32	0,19	15,22	81,47	13,50	72,29	100,43	537,74
Fibonacci◇	32	0,05	15,22	296,39	13,50	262,98	100,43	1956,32
Max	2048	4,42	72,60	16,44	533,62	120,82	4012,07	908,42
Pop Count	1024	1,21	11,49	9,49	-	-	70348,01	58084,49
Sobel	100	1,09	22,86	20,95	123,04	112,78	1071,63	982,27
Sobel◇	100	0,26	22,86	88,84	123,04	478,29	1071,63	4165,71
Vector Sum	2048	3,75	83,19	22,18	1434,66	382,49	5932,85	1581,72

◇ versão com reúso de dados

ritmos em um processador IBM PowerPC 405 (integrado em um FPGA Xilinx Virtex-II Pro) a 300 MHz e usando um barramento de sistema a 100 MHz. Os códigos foram gerados com o compilador GCC usando parâmetros para melhor desempenho. As latências medidas podem ser consideradas mínimas, pois foram usadas memórias *on-chip* para dados e instruções.

Os dados para o processador Nios II foram obtidos em um FPGA Stratix II (EP2S60F672C5). O sistema foi sintetizado no Quartus II versão 9.0 e operou a 50 MHz. Um *timer* foi utilizado no barramento para registrar o número exato de ciclos. Foi considerada a média do número de ciclos entre três execuções consecutivas, mas estas praticamente não apresentaram variações.

As implementações em LALP atingiram *speedups* médios de 56x sobre o a plataforma ARM, 224x sobre o PowerPC e 5853x sobre o Nios II. Se as frequências fossem aumentadas para 500MHz nos três processadores, os *speedups* ainda seriam em média 22x, 44x e 585x respectivamente.

7.8 Exploração do Espaço de Projeto

Durante o processo de geração automática de arquiteturas reconfiguráveis, a exploração do espaço de projeto consiste na obtenção de configurações Pareto-ótimas. Considerando múltiplas variáveis ou restrições, tais como área ocupada no dispositivo alvo, desempenho e consumo de potência, uma configuração Pareto-ótima é aquela em que não é possível atingir uma melhora em uma variável sem que isso implique em uma piora em outra (Pareto, 1909). A exploração

automática destas arquiteturas tem sido objeto de estudo tanto em trabalhos acadêmicos como em ferramentas comerciais (Abraham e Rau, 2000; So *et. al.*, 2002).

A variação dos parâmetros da arquitetura pode ser realizada por mudanças no código LALP que favoreçam determinada situação. Por exemplo, a frequência máxima de execução é determinada pelos caminhos críticos calculados pela ferramenta no processo de síntese. Em muitos casos, é possível reduzir estes caminhos por meio da inserção de registradores entre as operações. Este é um dos métodos para exploração do espaço de projeto usando LALP. No Algoritmo 7.3 é apresentado o exemplo *Dotprod* com modificações para melhoria de desempenho em termos de frequência. São adicionados registradores após as memórias x e y , além de seis estágios de *pipeline* no multiplicador.

Algoritmo 7.3: Exemplo *Dotprod* modificado para melhor desempenho

```

1  const DATA_WIDTH = 32;
2  const N = 2048;
3
4  typedef fixed(DATA_WIDTH, 1) int;
5  typedef fixed(1, 0) bit;
6
7  dotprod(out int sum, out bit done, in bit init) {
8      {
9          int x[N], y[N];
10         int acc;
11         fixed(16, 0) i;
12     }
13     counter (i=0; i<N; i++);
14     x.address = i;
15     y.address = i;
16     acc += (x@1) *@6 (y@1);
17     sum = acc;
18 }
```

Na arquitetura original, apresentada na Figura 7.7(a), os dados provenientes das memórias x e y vão diretamente para um multiplicador não registrado. Na versão com melhorias, apresentada na Figura 7.7(b), são inseridos registradores nas saídas das memórias e o multiplicador passa a ter seis estágios. O compilador insere registradores de deslocamento nas demais arestas de controle para acompanhar a variação.

Na Figura 7.8 são apresentadas diferentes configurações para o exemplo *Dotprod* em 2048 iterações. As variações de registradores foram testadas em três locais diferentes nas arquiteturas, apresentadas no eixo X do gráfico com legendas no formato x - y - z . A presença de

Figura 7.7: *Dotprod*: (a) arquitetura original; (b) arquitetura melhorada

registradores antes do multiplicador, na saída das memórias, é denotada por $x=1$ e a ausência por $x=0$. O número de estágios do multiplicador é denotado pelo valor de y , a presença de um registrador após o multiplicador por $z=1$ e a ausência por $z=0$. Desta maneira, as arquiteturas vão da primeira versão, identificada por 0-0-0 no gráfico, a qual não possui registradores em nenhuma posição, até a identificada por 1-6-1, que possui um registrador antes do multiplicador, seis estágios no multiplicador e um registrador após a operação.

A inserção de cada registrador na arquitetura implica em um incremento no número de ciclos necessários para calcular cada iteração. No entanto, as arquiteturas geradas possuem alto grau de paralelismo, o que faz com que este número tenha um impacto muito pequeno no tempo total de execução, inversamente proporcional ao número de iterações do algoritmo. A arquitetura 0-0-0 é capaz de calcular as 2048 iterações em 2050 ciclos de *clock*, enquanto a

Figura 7.8: Exploração do espaço de projeto para o exemplo *Dotprod*

arquitetura 1-6-1 necessita de 2058 ciclos, devido aos oito registradores inseridos na arquitetura.

Por análise do gráfico é possível notar que neste caso a inserção de registradores após o multiplicador tem impacto direto no uso de *FFs* e *Slices*, mas não apresenta melhora significativa em termos de frequência. Os melhores resultados neste dispositivo são obtidos com multiplicadores de seis estágios e com um registrador antes do multiplicador. Na Figura 7.9 é possível verificar quais são as configurações mais vantajosas em termos de tempo de execução e *Slices* ocupados.

As configurações representadas por pontos na linha, indicada por “Fronteira de Pareto”, são denominadas Pareto-ótimas, pois não é possível diminuir o tempo de execução sem que se aumente no número de *Slices* ocupados. As configurações acima desta linha causaram aumento no número de *Slices* ocupados, mas sem benefícios em termos de tempo de execução.

A exploração do espaço de projeto é dependente, entre outros fatores, do dispositivo alvo a ser usado. Na Figura 7.10 é apresentada a frequência máxima relativa do exemplo *Dotprod* com diferentes dispositivos, variando-se apenas o números de estágios no multiplicador.

Os dispositivos Cyclone III e Stratix III da Altera apresentam ganhos de desempenho com

Figura 7.9: Fronteira de Pareto considerando *Slices* e tempo de execução

Figura 7.10: Frequência máxima relativa por estágios no multiplicador

apenas um estágio no multiplicador. Os dispositivos Virtex 5 e Virtex 6 da Xilinx apresentam ganhos significativos com dois estágios, atingindo o melhor desempenho relativo com seis estágios. Essas diferenças ocorrem em função da arquitetura dos blocos de DSP de cada família de dispositivo.

7.9 Consumo de Potência

Os FPGAs em geral apresentam maior consumo de potência do que os ASICs implementados com a mesma tecnologia, especialmente se considerarmos o efeito *inrush* e o tempo de configuração dos dispositivos baseados em memórias SRAM². A potência estática é o consumo requerido para manter o dispositivo ligado e pode variar com a temperatura de operação. Já a potência dinâmica é o consumo requerido para realizar as mudanças de estado no dispositivo e varia diretamente com a frequência de operação, além da temperatura. Na [Figura 7.11](#) é apresentado o consumo de potência dinâmico estimado em Watts com a ferramenta Xilinx XPower Analyzer no dispositivo Virtex 6 (XC6VLX75T3FF484) para os exemplos *Dotprod*, *ADPCM Coder* e *FDCT* em diferentes frequências de operação, considerando uma temperatura constante.

No dispositivo analisado o consumo estático se mantém estável, em torno de 780 miliwatts, para os três exemplos, praticamente independente da frequência de operação. O consumo dinâmico varia diretamente com a área ocupada no FPGA e com a frequência de operação. Embora o consumo dinâmico aumente com frequências mais altas, a energia consumida não varia, pois o tempo de execução é proporcionalmente menor.

Assim como as melhorias de desempenho e da área ocupada no dispositivo, a redução do consumo de potência pode ser atingida em diversos níveis de projeto. Em termos arquiteturais, os níveis de *pipelining* são importantes para a redução do consumo de potência por operação ([Boemo et. al., 1995](#); [Wilton et. al., 2004](#)). Ao adotar uma arquitetura mais eficiente em termos de *throughput*, é possível diminuir a frequência de operação sem comprometer o desempenho requerido em termos de tempo de execução ([Saar, 2010](#)).

²Static Random Access Memory

Figura 7.11: Consumo de potência dinâmico por frequência de operação

Conclusão

NESTE trabalho foi apresentada uma nova abordagem para geração de arquiteturas especializadas, a partir de descrições em alto nível, para execução em dispositivos reconfiguráveis, mais especificamente os FPGAs. As técnicas desenvolvidas permitiram a obtenção de sistemas capazes de realizar a execução de *loops* de forma altamente paralela e, consequentemente, de alto desempenho. Para facilitar a descrição e a exploração de diferentes arquiteturas, foram criadas as linguagens de propósito específico LALP e LALP-S. Para comparação dos resultados obtidos foram usadas diferentes ferramentas e plataformas de execução.

Considerando os compiladores existentes, dentro do contexto apresentado, a maioria dos trabalhos concentram-se no mapeamento das operações descritas em linguagens tradicionais de software, principalmente C, para FPGAs. As razões para isto são a ampla utilização destas linguagens e todo o código legado. Conforme se pode verificar, a abordagem apresentada neste trabalho difere das anteriores, especialmente na identificação das operações paralelas e sequenciais. Ao permitir que o programador controle os ciclos de *clock* relativos entre as operações, o compilador ALP torna possível a exploração do espaço de projeto, favorecendo a obtenção de arquiteturas otimizadas.

As implementações obtidas com LALP obtiveram *speedups* médios de 1,2x e 5,9x com-

paradas àquelas obtidas com o ROCCC e o C-to-Verilog, respectivamente, devido ao maior *throughput* alcançado. Além disso, os exemplos compilados com ALP ocuparam em média 66% e 45% menos *Slices* do que o ROCCC e o C-to-Verilog, respectivamente. Na abordagem usada, maiores graus de paralelismo são atingidos sem a redundância de recursos. Cabe ressaltar que em LALP é possível alterar a implementação sempre que os resultados obtidos não forem satisfatórios, orientando a geração do *pipeline* de forma eficiente. Esta flexibilidade não é diretamente suportada pelos outros compiladores, que oferecem pouca ou nenhuma possibilidade de intervenção do programador.

O compilador ALP permite a exploração da vasta gama de recursos (memórias internas, blocos de DSP, etc.) dos FPGAs, atuais e futuros, de forma que o desempenho e/ou consumo de energia na execução de *loops* sejam otimizados. A exploração do espaço de projeto nestes sistemas envolve muitos parâmetros e é dependente do dispositivo alvo. Conforme demonstrado, pequenas modificações nos programas descritos em LALP permitem a obtenção de arquiteturas diferentes, característica ideal para aceleração de aplicações com componentes de *hardware*.

O compilador desenvolvido proporciona um ambiente favorável à pesquisa de novas otimizações na representação intermediária ou na geração de código para outros formatos e linguagens. O objetivo não foi a construção de uma ferramenta completa para o usuário final, mas sim de um protótipo que pudesse facilitar a validação das técnicas desenvolvidas com o conjunto de *benchmarks* usado.

Em muitos casos, é necessário acelerar apenas partes do programa em FPGAs e manter o restante em *software*, seja por questões de praticabilidade ou por necessidade de integração com outros sistemas. Embora o objetivo da pesquisa tenha se voltado para a obtenção de *kernels* eficientes, o modelo de programação usado se destina apenas a construção de *hardware*, mas pode possibilitar uma integração harmoniosa ao *software*.

Atualmente, as linguagens de programação de alto nível são as mais usadas para a programação de *software* em microprocessadores. Em alguns casos, a programação em linguagem de montagem ainda é usada, quando os recursos computacionais são escassos ou quando é requerido um desempenho excepcional. Os exemplos mais comuns destas necessidades são a programação de jogos e a de sistemas embarcados críticos. No caso dos compiladores de *software*, o

código de máquina resultante, em geral, é de qualidade inferior ao código escrito manualmente. No entanto, a evolução no desempenho dos sistemas fizeram com que esta perda fosse aceitável diante da facilidade de programação oferecida pelas linguagens de alto nível. O surgimento de novas técnicas de compilação para *hardware* e a evolução dos dispositivos reconfiguráveis podem tornar viável a transição para linguagens de alto nível também neste campo. Deste modo, acredita-se que em pouco tempo os compiladores de *hardware* baseados em linguagens de alto nível possam também atingir ampla aceitação, mas este ainda é um ponto em aberto e mais pesquisas são necessárias nesta área. O trabalho desenvolvido nesta tese mostrou que o uso de linguagens de domínio específico pode trazer avanços que favoreçam esta mudança.

8.1 Trabalhos Futuros

A infraestrutura de compilação desenvolvida neste trabalho oferece oportunidades para a realização de outras pesquisas na mesma linha, pois contempla o processo completo de desenvolvimento para arquiteturas reconfiguráveis a partir de descrições em LALP ou LALP-S. A seguir, são relacionadas algumas sugestões para trabalhos futuros que podem aproveitar e estender os avanços obtidos nesta pesquisa.

- Implementar as extensões descritas na [Seção 5.4](#), de modo a facilitar ainda mais a exploração do espaço de projeto com LALP. Incluir suporte direto a tipos de ponto fixo e ponto flutuante, além da possibilidade de tratar *loops* irregulares (*while* e *do-while*). Com recursos adicionais na linguagem, desenvolver novos algoritmos de análise de dependências e sincronismo. Isso permitiria a geração de arquiteturas, mesmo não otimizadas, com pouca ou nenhuma mudança no código original. A partir da implementação inicial, poderiam ser realizadas intervenções no código LALP para melhoria dos resultados.
- Realizar a tradução automática de programas escritos em linguagem C ou Java para LALP, permitindo o uso de uma infinidade de código disponível. As diretivas de sincronização, presentes em LALP, que não puderem ser inferidas automaticamente podem ser inseridas na linguagem de origem por meio de *pragmas* ou diretamente no código gerado em uma etapa posterior a tradução.

- Explorar a possibilidade de descrição dos programas de forma hierárquica, permitindo maior reuso dos códigos escritos em LALP e facilitando a criação de *cores* reaproveitáveis.
- Permitir, a partir das duas sugestões anteriores, a partição *hardware/software* de programas para execução em uma plataforma específica de processador, como por exemplo o Nios II da Altera e o MicroBlaze da Xilinx, ou em processadores *open source* como o LEON3, OpenSPARC e OpenRISC. A opção pela linguagem C com *pragmas*, apresentada na primeira sugestão de pesquisa, facilita a compilação do *software* para estes processadores. A modularização das especificações, sugerida a seguir, favorece a criação de módulos de *hardware* padronizados para os barramentos destes processadores.
- Implementar otimizações que permitam o reuso automático dos dados sem que o programador necessite descrevê-los explicitamente. Os compiladores pesquisados, em especial o C-to-Verilog e o ROCCC, realizam esta tarefa com resultados satisfatórios. No entanto, é necessária uma análise da aplicabilidade destas técnicas em conjunto com as técnicas até aqui desenvolvidas.
- Incluir otimizações ou diretivas que permitam a partição temporal do dispositivo. Embora a principal vantagem dos FPGAs seja a possibilidade de se realizar a computação de um problema de forma espacial, e consequentemente paralela, em certos casos a quantidade e a complexidade das operações podem extrapolar os recursos disponíveis.
- Desenvolver um modelo de simulação para as arquiteturas descritas em LALP, possibilitando a verificação funcional dos programas sem a necessidade de geração de código VHDL para simulação.

Referências Bibliográficas

- ABRAHAM, S. G.; RAU, B. R. Efficient design space exploration in PICO. Em: *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, New York, NY, USA: ACM, 2000, pgs. 71–79.
- ACKERMAN, W. B. Data Flow Languages. *Computer*, vol. 15, no. 2, pgs. 15–25, 1982.
- AHO, A.; SETHI, R.; ULLMAN, J. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
- AHUJA, S.; GURUMANI, S. T.; SPACKMAN, C.; SHUKLA, S. K. Hardware Coprocessor Synthesis from an ANSI C Specification. *IEEE Des. Test*, vol. 26, no. 4, pgs. 58–67, 2009.
- AIKEN, A.; NICOLAU, A. Perfect Pipelining: A New Loop Parallelization Technique. Em: *ESOP '88: Proceedings of the 2nd European Symposium on Programming*, London, UK: Springer-Verlag, 1988, pgs. 221–235.
- AIKEN, A.; NICOLAU, A.; NOVACK, S. Resource-Constrained Software Pipelining. *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 12, pgs. 1248–1270, 1995.
- ALLAN, V. H.; JONES, R. B.; LEE, R. M.; ALLAN, S. J. Software Pipelining. *ACM Computing Surveys*, vol. 27, no. 3, pgs. 367–432, 1995.
- ALLAN, V. H.; RAJAGOPALAN, M.; LEE, R. M. Software Pipelining: Petri Net Pacemaker. Em: *Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Amsterdam, The Netherlands: North-Holland Publishing Co., 1993, pgs. 15–26.
- ALPERN, B.; WEGMAN, M. N.; ZADECK, F. K. Detecting equality of variables in programs. Em: *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA: ACM Press, 1988, pgs. 1–11.
- AlteraURL *Nios II C2H Compiler User Guide*. Altera Corporation, 2008a.
Disponível em <http://www.altera.com/> (Acessado em 05/2008)
- AlteraURL *Nios II Processor Reference Handbook*. Altera Corporation, 2008b.
Disponível em <http://www.altera.com/literature/lit-nio2.jsp> (Acessado em 04/2009)
- AlteraURL *Stratix IV Device Handbook*. Altera Corporation, 2009.
Disponível em <http://www.altera.com/> (Acessado em 02/2010)

- ASSUMPCÃO JR., J. *State of the art in FPGA technology*. 2010.
Disponível em http://www.merlintec.com/download/jecel_osso.pdf
(Acessado em 05/2010)
- AUSTIN, T.; LARSON, E.; ERNST, D. SimpleScalar: An infrastructure for computer system modeling. *Computer*, vol. 35, no. 2, pgs. 59–67, 2002.
- BAL, H. E. A comparative study of five parallel programming languages. *Future Gener. Comput. Syst.*, vol. 8, no. 1-3, pgs. 121–135, 1992.
- BANERJEE, P.; SHENOY, N.; CHOUDHARY, A.; HAUCK, S.; BACHMANN, C.; HALDAR, M.; JOISHA, P.; JONES, A.; KANHARE, A.; NAYAK, A.; *et. AL.* A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. Em: *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society Washington, DC, USA, 2000, pg. 39.
- BANERJEE, U. K. *Dependence Analysis for Supercomputing*. Norwell, MA, USA: Kluwer Academic Publishers, 1988.
- BEN-ASHER, Y.; ROTEM, N. Synthesis for variable pipelined function units. Em: *System-on-Chip, 2008. SOC 2008. International Symposium on*, 2008, pgs. 1–4.
- BOBDA, C. *Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications*. Springer, 2007.
- BOEMO, E.; GONZALEZ DE RIVERA, G.; LÓPEZ-BUEDO, S.; MENESES, J. Some notes on power management on FPGA-based systems. *Lecture Notes in Computer Science*, vol. 975, pgs. 149–157, 1995.
- BONDALAPATI, K.; PRASANNA, V. K. Reconfigurable Computing Systems. *Proceedings of the IEEE*, vol. 90, no. 7, pgs. 1201–1217, 2002.
- BOUT, V. D.; E., D. *The Practical Xilinx Designer Lab Book Version 1.5*. Prentice Hall, 1999.
- BROWN, S.; VRANESIC, Z. *Fundamentals of Digital Logic with VHDL Design*. McGraw Hill, 2000.
- BUYUKKURT, B.; GUO, Z.; NAJJAR, W. A. Impact of Loop Unrolling on Area, Throughput and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs. Em: *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC2006)*, 2006.
- C-to-VerilogURL *C-to-Verilog*. C-to-Verilog.com, 2009.
Disponível em <http://www.c-to-verilog.com/> (Acessado em 05/2009)
- CALLAHAN, T. J. *Automatic compilation of c for hybrid reconfigurable architectures*. Tese de doutorado, University of California, 2002.
- CALLAHAN, T. J.; HAUSER, J. R.; WAWRZYNEK, J. The GARP Architecture and C Compiler. *Computer*, vol. 33, no. 4, pgs. 62–69, 2000.

- CALLAHAN, T. J.; WAWRZYNEK, J. Adapting Software Pipelining for Reconfigurable Computing. Em: *CASES '00: Proceedings of the 2000 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, New York, NY, USA: ACM Press, 2000, pgs. 57–64.
- CARDOSO, J. M. P. *Compilação de Algoritmos em Java para Sistemas Computacionais Reconfiguráveis com Exploração do Paralelismo ao Nível das Operações*. Tese de doutorado, Universidade Técnica de Lisboa, 2000.
- CARDOSO, J. M. P. Dynamic Loop Pipelining in Data-Driven Architectures. Em: *CF '05: Proceedings of the 2nd conference on Computing Frontiers*, New York, NY, USA: ACM Press, 2005, pgs. 106–115.
- CARDOSO, J. M. P.; DINIZ, P. C. *Compilation Techniques for Reconfigurable Architectures*. Springer Publishing Company, Incorporated, 2008.
- CARDOSO, J. M. P.; NETO, H. C. Compilation for FPGA-based reconfigurable hardware. *IEEE Design & Test of Computers*, vol. 20, no. 2, pgs. 65–75, 2003.
- CASTILLO, J.; BOSQUE, J. L.; CASTILLO, E.; HUERTA, P.; MARTINEZ, J. Hardware accelerated montecarlo financial simulation over low cost fpga cluster. *Parallel and Distributed Processing Symposium, International*, vol. 0, pgs. 1–8, 2009.
- CHAN, P. K.; MOURAD, S. *Digital Design Using Field-Programmable Gate Arrays*. Prentice Hall, 1994.
- CHAO, L.-F.; LAPAUGH, A. S.; SHA, E. H.-M. Rotation Scheduling: A Loop Pipelining Algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 3, pgs. 229–239, 1997.
- CHARLESWORTH, A. E. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. *Computer*, vol. 14, no. 9, pgs. 18–27, 1981.
- CHENG, T. From the EIC: Building and verifying hardware at a higher level of abstraction. *IEEE Design and Test of Computers*, vol. 26, pg. 2, 2009.
- COMPTON, K. *Programming Architectures for Run-Time Reconfigurable Systems*. Dissertação de mestrado, Northwestern University, 1999.
- COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, vol. 34, no. 2, pgs. 171–210, 2002.
- CONG, J.; ROSENSTIEL, W. The Last Byte: The HLS tipping point. *IEEE Des. Test*, vol. 26, no. 4, pg. 104, 2009.
- COPELAND, T. *Generating Parsers with JavaCC*. Centennial Books, 2007.
- COUSSY, P.; GAJSKI, D.; MEREDITH, M.; TAKACH, A.; GRAPHICS, M. An Introduction to High-Level Synthesis. *IEEE Design & Test*, vol. 26, no. 4, pgs. 8–17, 2009.
- COUSSY, P.; MORAWIEC, A. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

- COUTINHO, J.; LUK, W. Source-directed transformations for hardware compilation. Em: *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, 2003, pgs. 278 – 285.
- COUTINHO, J. G. F.; JIANG, J.; LUK, W. Interleaving Behavioral and Cycle-Accurate Descriptions for Reconfigurable Hardware Compilation. Em: *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA: IEEE Computer Society, 2005, pgs. 245–254.
- CRONQUIST, D. C.; FRANKLIN, P.; BERG, S. G.; EBELING, C. Specifying and Compiling Applications for RaPiD. Em: *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Washington, DC, USA: IEEE Computer Society, 1998, pg. 116.
- CYTRON, R.; FERRANTE, J.; ROSEN, B. K.; WEGMAN, M. N.; ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pgs. 451–490, 1991.
- DEHON, A. The density advantage of configurable computing. *Computer*, vol. 33, no. 4, pgs. 41–49, 2000.
- DENSMORE, D.; PASSERONE, R.; SANGIOVANNI-VINCENTELLI, A. A Platform-Based Taxonomy for ESL Design. *IEEE Design and Test of Computers*, vol. 23, no. 5, pgs. 359–374, 2006.
- DUBOIS, D.; DUBOIS, A.; BOORMAN, T.; CONNOR, C. Non-Preconditioned Conjugate Gradient on Cell and FPGA Based Hybrid Supercomputer Nodes. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pgs. 201–208, 2009.
- DUBOIS, D.; DUBOIS, A.; BOORMAN, T.; CONNOR, C.; POOLE, S. Sparse matrix-vector multiplication on a reconfigurable supercomputer with application. *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 1, pgs. 1–31, 2010.
- EBCIOGLU, K. A compilation technique for software pipelining of loops with conditional jumps. Em: *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, 1987.
- EBELING, C. *RaPiD-C Manual*. 2002.
- EBELING, C.; CRONQUIST, D. C.; FRANKLIN, P. Rapid - reconfigurable pipelined datapath. Em: *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, London, UK: Springer-Verlag, 1996, pgs. 126–135.
- ELBIRT, A.; YIP, W.; CHETWYND, B.; PAAR, C. An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. Em: *Proceedings of IEEE Transactions on VLSI Systems*, 2001.
- ELBIRT, A. J.; YIP, W.; CHETWYND, B.; PAAR, C. An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. Em: *Proceedings of ACM/SIGDA International Symposium on FPGAs*, 2000, pgs. 13–27.
- FEO, J. T. *Comparative Study of Parallel Programming Languages: The Salishan Problems*. New York, NY, USA: Elsevier Science Inc., 1992.

- FERNANDES, M. M.; LLOSA, J.; TOPHAM, N. Distributed Modulo Scheduling. Em: *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Washington, DC, USA: IEEE Computer Society, 1999, pg. 130.
- GAJSKI, D. D.; DUTT, N. D.; WU, A. C. H.; LIN, S. Y. L. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.
- GOLDSTEIN, S.; SCHMIT, H.; MOE, M.; BUDIU, M.; CADAMBI, S.; TAYLOR, R.; LAUFER, R. PipeRench: a coprocessor for streaming multimedia acceleration. Em: *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, 1999, pgs. 28–39.
- GOLDSTEIN, S. C.; SCHMIT, H.; BUDIU, M.; CADAMBI, S.; MOE, M.; TAYLOR, R. R. PipeRench: A Reconfigurable Architecture and Compiler. *Computer*, vol. 33, no. 4, pgs. 70–77, 2000.
- GOOSSENS, G.; VANDEWILLE, J.; MAN, H. D. Loop optimization in register-transfer scheduling for DSP-systems. Em: *DAC'89: Proceedings of the 26th ACM/IEEE Conference on Design Automation*, New York, NY, USA: ACM Press, 1989, pgs. 826–831.
- GRANSTON, E.; STOTZER, E.; ZBICIAK, J. Software Pipelining Irregular Loops On the TMS320C6000 VLIW DSP Architecture. Em: *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, New York, NY, USA: ACM, 2001, pgs. 138–144.
- GraphvizURL *Graphviz: Graph Visualization Software*. AT&T Research, 2006.
Disponível em <http://www.graphviz.org/> (Acessado em 08/2006)
- GUO, Z.; BUYUKKURT, B.; NAJJAR, W.; VISSERS, K. Optimized generation of data-path from C codes for FPGAs. Em: *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, IEEE Computer Society, 2005, pgs. 112–117.
- GUPTA, R.; BREWER, F. *High-Level Synthesis: From Algorithm to Digital Circuit*, cap. High-Level Synthesis: A Retrospective Springer, pgs. 13–28, 2008.
- GUPTA, S.; DUTT, N.; GUPTA, R.; NICOLAU, A. Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow. Em: *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, Washington, DC, USA: IEEE Computer Society, 2004a, pg. 10114.
- GUPTA, S.; GUPTA, R.; DUTT, N.; NICOLAU, A. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004b.
- GUTHAUS, M. R.; RINGENBERG, J. S.; ERNST, D.; AUSTIN, T. M.; MUDGE, T.; BROWN, R. B. MiBench: A free, commercially representative embedded benchmark suite. Em: *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, Washington, DC, USA: IEEE Computer Society, 2001, pgs. 3–14.
- HALDAR, M.; NAYAK, A.; CHOUDHARY, A.; BANERJEE, P. Scheduling algorithms for automated synthesis of pipelined designs on FPGAs for applications described in matlab. Em: *CASES '00: Proceedings of the 2000 international conference on Compilers, Architectures and Synthesis for Embedded Systems*, New York, NY, USA: ACM Press, 2000, pgs. 85–93.

- HALDAR, M.; NAYAK, A.; CHOUDHARY, A.; BANERJEE, P. A System for Synthesizing Optimized FPGA Hardware from Matlab. Em: *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, Piscataway, NJ, USA: IEEE Press, 2001, pgs. 314–319.
- HAMBLÉN, J. O.; FURMAN, M. D. *Rapid Prototyping of Digital Systems*. Kluwer, 2001.
- Handel-CURL *Handel-C Language Reference Manual*. Agility Design Solutions Inc., 2007. Disponível em <http://www.agilityds.com/> (Acessado em 04/2010)
- HAUCK, S. The Future of Reconfigurable Systems. Em: *Proceedings of the 5th Canadian Conference on Field Programmable Devices*, 1998.
- HERBORDT, M. C.; VANCOURT, T.; GU, Y.; SUKHWANI, B.; CONTI, A.; MODEL, J.; DISABELLO, D. Achieving High Performance with FPGA-Based Computing. *Computer*, vol. 40, no. 3, pgs. 50–57, 2007.
- HWANG, C.-T.; HSU, Y.-C.; LIN, Y.-L. Scheduling for functional pipelining and loop winding. Em: *DAC'91: Proceedings of the 28th ACM/IEEE Conference on Design Automation*, New York, NY, USA: ACM Press, 1991, pgs. 764–769.
- HWANG, C.-T.; HSU, Y.-C.; Y.-L. PLS: A scheduler for pipeline synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 9, pgs. 1279–1286, 1993.
- JavaCCURL *Java Compiler Compiler*. 2010. Disponível em <https://javacc.dev.java.net/> (Acessado em 05/2010)
- KODAGANALLUR, V. Incorporating language processing into Java applications: A JavaCC tutorial. *IEEE Softw.*, vol. 21, no. 4, pgs. 70–77, 2004.
- LAM, M. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. Em: *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, New York, NY, USA: ACM Press, 1988, pgs. 318–328.
- LANZAGORTA, M.; BIQUE, S.; ROSENBERG, R. Introduction to Reconfigurable Supercomputing. *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pgs. 1–103, 2009.
- LEUPERS, R.; MARWEDEL, P. *Retargetable compiler technology for embedded systems: tools and applications*. Kluwer Academic Publishers, 2001.
- MARTIN, G.; SMITH, G. High-Level Synthesis: Past, Present, and Future. *IEEE Design and Test of Computers*, vol. 26, pgs. 18–25, 2009.
- MENOTTI, R.; CARDOSO, J. M. P.; FERNANDES, M. M.; MARQUES, E. Automatic Generation of FPGA Hardware Accelerators Using a Domain Specific Language. Em: *International Conference on Field Programmable Logic and Applications (FPL)*, 2009a, pgs. 457–461.
- MENOTTI, R.; CARDOSO, J. M. P.; FERNANDES, M. M.; MARQUES, E. LALP: A Novel Language to Program Custom FPGA-based Architectures. Em: *Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Los Alamitos, CA, USA: IEEE Computer Society Press, 2009b, pgs. 3–10.

- MENOTTI, R.; CARDOSO, J. M. P.; FERNANDES, M. M.; MARQUES, E. LALP: A Language to Program Custom FPGA-based Acceleration Engines. *International Journal of Parallel Programming*, artigo submetido, 2010a.
- MENOTTI, R.; CARDOSO, J. M. P.; FERNANDES, M. M.; MARQUES, E. On Using LALP to Map an Audio Encoder/Decoder on FPGAs. Em: *Proceedings of the 2010 IEEE International Symposium on Industrial Electronics*, 2010b, pgs. 3063–3068.
- MENOTTI, R.; CARDOSO, J. M. P.; FERNANDES, M. M.; MARQUES, E. Uma Linguagem para Geração Automática de Arquiteturas Baseadas em Computação Reconfigurável. Em: *REC2010: VI Jornadas sobre Sistemas Reconfiguráveis*, 2010c.
- MENOTTI, R.; MARQUES, E.; CARDOSO, J. M. P. Aggressive Loop Pipelining for Reconfigurable Architectures. Em: *International Conference on Field Programmable Logic and Applications (FPL)*, 2007, pgs. 501–502.
- DE MICHELI, G.; SAMI, M. *Hardware/Software Co-Design*. Kluwer Academic Publishers, 1996.
- MUCHNICK, S. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.
- MURGAI, R.; BRAYTON, R. K.; SANGIOVANNI-VINCENTELLI, A. *Logic Synthesis for Field-Programmable Gate Arrays*. Kluwer Academic, 1995.
- OLDFIELD, J. V.; DORF, R. C. *Field-Programmable Gate Arrays*. John Wiley & Sons Inc., 1995.
- O'MADADHAIN, J.; FISHER, D.; WHITE, S.; BOEY, Y.-B. *The JUNG (Java Universal Network/Graph) Framework*. Relatório técnico UCI-ICS 03-17, University of California, 2003.
- PARETO, V. Manuel d'économie politique. *Trans. A. Bonnet. Paris: Giard & Briere*, 1909.
- PATEL, J. H.; DAVIDSON, E. S. Improving the throughput of a pipeline by insertion of delays. *SIGARCH Comput. Archit. News*, vol. 4, no. 4, pgs. 159–164, 1976.
- PSARRIS, K.; KYRIAKOPOULOS, K. An experimental evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 3, pgs. 196–213, 2004.
- RAU, B. R. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. Em: *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, New York, NY, USA: ACM Press, 1994, pgs. 63–74.
- RODRIGUES, R.; CARDOSO, J. M. P.; DINIZ, P. C. A Data-Driven Approach for Pipelining Sequences of Data-Dependent Loops. Em: *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA: IEEE Computer Society, 2007, pgs. 219–228.
- RONG, H.; DOUILLET, A.; GAO, G. R. Register allocation for software pipelined multi-dimensional loops. Em: *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA: ACM Press, 2005, pgs. 154–167.

- SAAR, H. *Minimize FPGA power consumption*. Actel Corp., 2010.
Disponível em http://www.eetasia.com/ART_8800454242_499485_NT_967f82ca.HTM (Acessado em 02/2010)
- SANGIOVANNI-VINCENTELLI, A. *Defining platform-based design*. 2010.
Disponível em <http://www.eetimes.com/news/design/showArticle.jhtml?articleID=16504380> (Acessado em 02/2010)
- SARKAR, S.; DABRAL, S.; TIWARI, P.; MITRA, R. Lessons and Experiences with High-Level Synthesis. *Design and Test of Computers, IEEE*, vol. 26, no. 4, pgs. 34–45, 2009.
- SCHREIBER, R.; ADITYA, S.; MAHLKE, S.; KATHAIL, V.; RAU, B. R.; CRONQUIST, D.; SIVARAMAN, M. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *The Journal of VLSI Signal Processing*, vol. 31, no. 2, pgs. 127–142, 2002.
- SCOWEN, R. Extended BNF: a generic base standard. Em: *Proc. 1993 Software Engineering Standards Symposium (SESS'93)*, Brighton, UK, 1993.
- SimpleScalarURL *SimpleScalar*. SimpleScalar LLC, 2004.
Disponível em <http://www.simplescalar.com/v4test.html> (Acessado em 05/2010)
- SNIDER, G. Performance-constrained pipelining of software loops onto reconfigurable hardware. Em: *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-Programmable Gate Arrays*, New York, NY, USA: ACM Press, 2002, pgs. 177–186.
- SO, B.; HALL, M. W.; DINIZ, P. C. A compiler approach to fast hardware design space exploration in FPGA-based systems. *SIGPLAN Not.*, vol. 37, no. 5, pgs. 165–176, 2002.
- SO, W.; DEAN, A. G. Complementing software pipelining with software thread integration. Em: *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, New York, NY, USA: ACM Press, 2005, pgs. 137–146.
- DE SOUZA, V. L. S. *Implementação de uma arquitetura para multiplicação de matrizes densas em sistemas reconfiguráveis de alto desempenho*. Dissertação de mestrado, Universidade Federal de Pernambuco, 2008.
- STOODLEY, M.; LEE, C. Software pipelining loops with conditional branches. Em: *Proceedings of the MICRO-29 - The 29th Annual International Symposium on Microarchitecture*, 1996.
- SuifURL *SUIF Compiler System*. The Stanford SUIF Compiler Group, 2008.
Disponível em <http://suif.stanford.edu/> (Acessado em 07/2008)
- TexasURL *TMS320C64x DSP Library: Programmer's Reference*. Texas Instruments Incorporated, 2003a.
Disponível em <http://focus.ti.com/lit/ug/spru565b/spru565b.pdf> (Acessado em 05/2010)

- TexasURL *TMS320C64x Image/Video Processing Library: Programmer's Reference*. Texas Instruments Incorporated, 2003b.
Disponível em <http://focus.ti.com/lit/ug/spru023b/spru023b.pdf> (Acessado em 05/2010)
- UNDERWOOD, K. Fpgas vs. cpus: trends in peak floating-point performance. Em: *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, New York, NY, USA: ACM, 2004, pgs. 171–180.
- VEGDAHL, S. R. *Local code generation and compaction in optimizing microcode compilers*. Tese de doutorado, Carnegie-Mellow University, 1982.
- WARTER, N. J.; HAAB, G. E.; BOCKHAUS, J. W. Enhanced modulo scheduling for loops with conditional branches. Em: *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, IEEE Computer Society Press, 1992, pgs. 170–179.
- WEINHARDT, M.; LUK, W. Pipeline Vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pgs. 234–248, 2001.
- WILTON, S. J. E.; ANG, S.-S.; LUK, W. The impact of pipelining on energy per operation in field-programmable gate arrays. Em: *International Conference on Field Programmable Logic and Applications (FPL)*, 2004, pgs. 719–928.
- WOLF, W.; STAUNSTRUP, J. *Hardware/Software Co-Design: Principles and Practice*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.
- XilinxURL *MicroBlaze Processor Reference Guide*. Xilinx Inc., 2009.
Disponível em http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf (Acessado em 04/2009)
- ZAKY, A. M. *Efficient static scheduling of loops on synchronous multiprocessors*. Tese de doutorado, Ohio State University, 1989.
- ZHUO, L.; PRASANNA, V. K. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 4, pgs. 433–448, 2007.

Especificação Formal da Linguagem

*N*ESTE apêndice é listada a gramática livre de contexto que descreve a linguagem LALP usando a notação EBNF¹ (Scowen, 1993). O compilador ALP usa a ferramenta JavaCC para gerar o *parser* da linguagem que está especificada na forma de gramática LL(*k*).

A.1 Símbolos terminais

ANDASSIGN = “&=”;

ASSIGN = “=”;

AT = “@”;

BANG = “!”;

BIT_AND = “&”;

BIT_OR = “|”;

COLON = “:”;

COMMA = “,”;

CONST = “const”;

COUNTER = “counter”;

¹Extended Backus-Naur Form

DECR = “--” ;
DIGIT = “0”..“9” ;
DOT = “.” ;
EQ = “==” ;
FIXED = “fixed” ;
GE = “>=” ;
GT = “>” ;
HOOK = “?” ;
IN = “in” ;
INCR = “++” ;
LBRACE = “{” ;
LBRACKET = “[” ;
LE = “<=” ;
LETTER = “a”..“z” | “A”..“Z” ;
LPAREN = “(” ;
LSHIFT = “<<” ;
LSHIFTASSIGN = “<<=” ;
LT = “<” ;
MINUS = “-” ;
MINUSASSIGN = “- =” ;
NE = “!=” ;
ORASSIGN = “| =” ;
OUT = “out” ;
PLUS = “+” ;
PLUSASSIGN = “+ =” ;
RBRACE = “}” ;
RBRACKET = “]” ;
REM = “%” ;
REMASSIGN = “% =” ;

RPAREN = “)” ;
 RSIGNEDSHIFT = “>>” ;
 RSIGNEDSHIFTASSIGN = “>>=” ;
 RUNSIGNEDSHIFT = “>>>” ;
 RUNSIGNEDSHIFTASSIGN = “>>>=” ;
 SC_AND = “&&” ;
 SC_OR = “||” ;
 SEMICOLON = “;” ;
 SLASH = “/” ;
 SLASHASSIGN = “/ =” ;
 STAR = “*” ;
 STARASSIGN = “* =” ;
 TILDE = “~” ;
 TYPEDEF = “typedef” ;
 WHEN = “when” ;
 XOR = “^” ;
 XORASSIGN = “^=” ;

A.2 Símbolos não terminais

LALPProgram =

*(Const)**

*(Typedef)**

Identifier

LPAREN (*Pin* (COMMA *Pin*)*)? RPAREN

LBRACE *Statements* RBRACE ;

Const =

CONST *Identifier* ASSIGN *IntegerLiteral* SEMICOLON ;

Typedef =

TYPEDEF *Fixed Identifier* SEMICOLON ;

Pin =

(IN | OUT) (*Identifier* | *Fixed*) *Identifier* ;

Statements =

(*Declarations*)?

(*Statement*)⁺ ;

Fixed =

FIXED LPAREN *ConstOrInt* COMMA *ConstOrInt*

(COMMA *ConstOrInt*)? RPAREN ;

Name =

Identifier (DOT *Identifier*)? ;

Declarations =

LBACE (*Declaration*)⁺ RBACE ;

Declaration =

(*Identifier* | *Fixed*) *Variable* (COMMA *Variable*)^{*} SEMICOLON ;

Variable =

Identifier (LBRACKET *ConstOrInt* RBRACKET)?

(ASSIGN (*ConstOrInt* | *MemoryInit*))? ;

MemoryInit =

LBRACE *IntegerLiteral* (**COMMA** *IntegerLiteral*)* **RBRACE** ;

Counter =

COUNTER **LPAREN** *Identifier* **ASSIGN** (*IntegerLiteral* | *Identifier*)

SEMICOLON *Identifier* *CounterComp* (*IntegerLiteral* | *Identifier*)

SEMICOLON *Identifier* (**INCR** | **DECR** | (**PLUSASSIGN** | **MINUSASSIGN**) *ConstOrInt*)

(**AT** *ConstOrInt*)? **RPAREN** ;

CounterComp =

(**LT** | **LE** | **GT** | **GE** | **EQ** | **NE**) ;

Statement =

(*Assignment* | *Counter*) **SEMICOLON** ;

Assignment =

LHS *AssignmentOperator* *RHS* (*When*)? ;

When =

WHEN *Expression* ;

LHS =

Name ;

AssignmentOperator =

(**ASSIGN** | **STARASSIGN** | **SLASHASSIGN** | **REMASSIGN** | **PLUSASSIGN** |

MINUSASSIGN | **LSHIFTASSIGN** | **RSIGNEDSHIFTASSIGN** |

RUNSIGNEDSHIFTASSIGN | **ANDASSIGN** | **XORASSIGN** | **ORASSIGN**) ;

RHS =

Expression ;

Expression =

DelayExpression ;

DelayExpression =

ConditionalExpression AT *ConstOrInt* ;

ConditionalExpression =

ConditionalOrExpression

[HOOK *ConditionalOrExpression* COLON *ConditionalOrExpression*] ;

ConditionalOrExpression =

ConditionalAndExpression (SC_OR *ConditionalAndExpression*)* ;

ConditionalAndExpression =

InclusiveOrExpression (SC_AND *InclusiveOrExpression*)* ;

InclusiveOrExpression =

ExclusiveOrExpression (BIT_OR *ExclusiveOrExpression*)* ;

ExclusiveOrExpression =

AndExpression (XOR *AndExpression*)* ;

AndExpression =

EqualityExpression (AND *EqualityExpression*)* ;

EqualityExpression =

RelationalExpression ((EQ | NE) *RelationalExpression*)* ;

RelationalExpression =

ShiftExpression ((LT | GT | LE | GE) *ShiftExpression*)* ;

ShiftExpression =

AdditiveExpression

((LSHIFT | RSIGNEDSHIFT | RUNSIGNEDSHIFT) *ConstOrInt*)* ;

AdditiveExpression =

MultiplicativeExpression ((PLUS | MINUS) *MultiplicativeExpression*)* ;

MultiplicativeExpression =

UnaryExpression ((STAR [AT *ConstOrInt*] | SLASH | REM) *UnaryExpression*)* ;

UnaryExpression =

((PLUS | MINUS) *UnaryExpression* | *PreIncrementExpression* |

PreDecrementExpression | *UnaryExpressionNotPlusMinus*) ;

PreIncrementExpression =

INCR *PrimaryExpression* ;

PreDecrementExpression =

DECR *PrimaryExpression* ;

UnaryExpressionNotPlusMinus =

((TILDE | BANG) *UnaryExpression* | *PostfixExpression*) ;

PostfixExpression =

PrimaryExpression [INCR | DECR] ;

PrimaryExpression =
(LPAREN *Expression* RPAREN | *Literal* | *Name*) ;

Literal =
IntegerLiteral ;

ConstOrInt =
(*IntegerLiteral* | *Identifier*) ;

Identifier =
LETTER (LETTER | DIGIT)* ;

IntegerLiteral =
(*DecimalLiteral* | *HexLiteral*) ;

DecimalLiteral =
(DIGIT)+ ;

HexLiteral =
“0” (“x” | “X”) (DIGIT | “a” .. “f” | “A” .. “F”)+ ;

Códigos Fonte dos *Benchmarks*

NESTE apêndice são listados os *benchmarks* cujos códigos fonte não foram usados como exemplos no texto principal. Para facilitar a localização no texto, a lista completa dos códigos C e LALP, com referências e números de página, é apresentada na [Tabela B.1](#).

Tabela B.1: Lista dos *benchmarks* e respectivos códigos fonte

<i>Benchmark</i>	Referência (Página)	
	C	LALP
<i>ADPCM Coder</i>	Algoritmo B.1 (146)	Algoritmo 5.14 (71)
<i>ADPCM Decoder</i>	Algoritmo B.2 (147)	Algoritmo 5.15 (72)
<i>Autocorrelation</i>	Algoritmo B.3 (148)	Algoritmo B.11 (153)
<i>Bubble Sort</i>	Algoritmo B.4 (149)	Algoritmo B.13 (157)
<i>Dotprod</i>	Algoritmo 5.8 (63)	Algoritmo 5.9 (63)
<i>Dotprod</i> ♥	Algoritmo 5.8 (63)	Algoritmo 7.3 (117)
<i>FDCT</i>	Algoritmo B.5 (149)	Algoritmo B.12 (154)
<i>Fibonacci</i>	Algoritmo B.6 (152)	Algoritmo 5.10 (64)
<i>Fibonacci</i> ◇	Algoritmo B.7 (152)	Algoritmo 5.11 (65)
<i>Max</i>	Algoritmo 7.1 (102)	Algoritmo B.14 (158)
<i>Pop Count</i>	Algoritmo B.8 (152)	Algoritmo B.15 (158)
<i>Sobel</i>	Algoritmo B.9 (152)	Algoritmo 5.12 (67)
<i>Sobel</i> ◇	Algoritmo B.9 (152)	Algoritmo 5.13 (68)
<i>Vector Sum</i>	Algoritmo B.10 (153)	Algoritmo B.16 (161)

♥ versão otimizada ◇ versão com reúso de dados

B.1 Algoritmos Descritos em C

Algoritmo B.1: Exemplo *ADPCM Coder* descrito em C

```

1  #define DATASIZE 1024
2  short indata[DATASIZE];
3  char outdata[(DATASIZE/2)+1];
4  static int indexTable[16] = {
5      -1, -1, -1, -1, 2, 4, 6, 8,
6      -1, -1, -1, -1, 2, 4, 6, 8,
7  };
8  static int stepsizeTable[89] = {
9      7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
10     19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
11     50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
12     130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
13     337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
14     876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
15     2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
16     5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
17     15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
18 };
19 main() {
20     int i;
21     int len;
22     int sign;
23     int delta;
24     int step;
25     int valpred;
26     int vpdiff;
27     int index;
28     int bufferstep;
29     int outputbuffer;
30     int diff;
31     int val;
32     step = stepsizeTable[index];
33     bufferstep = 1;
34     i=0;
35     for (len = 0 ; len < DATASIZE ; len++ ) {
36         val = indata[len];
37         diff = val - valpred;
38         sign = (diff < 0) ? 8 : 0;
39         if ( sign )
40             diff = (-diff);
41         delta = 0;
42         vpdiff = (step >> 3);
43         if ( diff >= step ) {
44             delta = 4;
45             diff -= step;
46             vpdiff += step;
47         }
48         step >>= 1;
49         if ( diff >= step ) {
50             delta |= 2;
51             diff -= step;
52             vpdiff += step;

```



```

53     }
54     step >= 1;
55     if ( diff >= step ) {
56         delta != 1;
57         vpdiff += step;
58     }
59     if ( sign )
60         valpred -= vpdiff;
61     else
62         valpred += vpdiff;
63     if ( valpred > 32767 )
64         valpred = 32767;
65     else if ( valpred < -32768 )
66         valpred = -32768;
67     delta != sign;
68     index += indexTable[delta];
69     if ( index < 0 )
70         index = 0;
71     if ( index > 88 )
72         index = 88;
73     step = stepsizeTable[index];
74     if ( bufferstep ) {
75         outputbuffer = (delta << 4) & 0xf0;
76     } else {
77         outdata[i] = (delta & 0x0f) | outputbuffer;
78     }
79     bufferstep = !bufferstep;
80 }
81 if ( !bufferstep )
82     outdata[i] = outputbuffer;
83 }

```

Algoritmo B.2: Exemplo *ADPCM Decoder* descrito em C

```

1  #define DATASIZE 1024
2  short indata[(DATASIZE/2)+1];
3  char outdata[DATASIZE];
4  static int indexTable[16] = {
5      -1, -1, -1, -1, 2, 4, 6, 8,
6      -1, -1, -1, -1, 2, 4, 6, 8,
7  };
8  static int stepsizeTable[89] = {
9      7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
10     19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
11     50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
12     130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
13     337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
14     876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
15     2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
16     5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
17     15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
18 };
19 main() {
20     int i;
21     int len;
22     int sign;
23     int delta;
24     int step;

```

```

25  int valpred;
26  int vpdiff;
27  int index;
28  int bufferstep;
29  int inputbuffer;
30  step = stepsizeTable[index];
31  bufferstep = 0;
32  i=0;
33  for (len = 0 ; len < DATASIZE ; len++ ) {
34      if ( bufferstep ) {
35          delta = inputbuffer & 0xf;
36      }
37      else {
38          inputbuffer = indata[i];
39          i++;
40          delta = (inputbuffer >> 4) & 0xf;
41      }
42      bufferstep = !bufferstep;
43      index += indexTable[delta];
44      if ( index < 0 )
45          index = 0;
46      if ( index > 88 )
47          index = 88;
48      sign = delta & 8;
49      delta = delta & 7;
50      vpdiff = step >> 3;
51      if ( delta & 4 )
52          vpdiff += step;
53      else
54          if ( delta & 2 )
55              vpdiff += step>>1;
56          else
57              if ( delta & 1 )
58                  vpdiff += step>>2;
59      if ( sign )
60          valpred -= vpdiff;
61      else
62          valpred += vpdiff;
63      if ( valpred > 32767 )
64          valpred = 32767;
65      else if ( valpred < -32768 )
66          valpred = -32768;
67      step = stepsizeTable[index];
68      outdata[len] = valpred;
69  }
70  }

```

Algoritmo B.3: Exemplo *Autocorrelation* descrito em C

```

1  #define N 160    //—— Length of Input array vector (sd[]) —
2  #define M 10    //(MULTIPLE of 2)
3
4  short ac[M]; // —— Resulting array of autocorrelation
5  short sd[N]; // —— Input array of autocorrelation
6
7  main() {
8      int i,k,sum;
9

```

```

10  for (i = 0; i < M; i++){
11      sum = 0;
12
13      for (k = 0; k < N; k++) {
14          sum += sd[k+M] * sd[k+M-i];
15      }
16
17      ac[i] = (sum >> 15);
18  }
19  }

```

Algoritmo B.4: Exemplo *Bubble Sort* descrito em C

```

1  #define N 32
2
3  int ar[N];
4
5  main() {
6      int i, j;
7      int temp;
8      for (i=0; i<N; i++) {
9          for (j=0; j<N-1; j++) {
10             if (ar[j] > ar[j+1]) {
11                 temp = ar[j+1];
12                 ar[j+1] = ar[j];
13                 ar[j] = temp;
14             }
15         }
16     }
17 }

```

Algoritmo B.5: Exemplo *FDCT* descrito em C

```

1  #define N 8 // fixed
2  #define M N*N
3  #define num_fdcts 10
4  #define SIZE num_fdcts*M
5
6  short    dct_io_ptr[SIZE];
7  short    dct_io_tmp[SIZE];
8  short    dct_o[SIZE];
9
10 int main() {
11     const unsigned short c1 = 0x2C62, c3 = 0x25A0;
12     const unsigned short c5 = 0x1924, c7 = 0x08D4;
13     const unsigned short c0 = 0xB505, c2 = 0x29CF;
14     const unsigned short c6 = 0x1151;
15     short f0, f1, f2, f3, f4, f5, f6, f7;           // Spatial domain samples.
16     int g0, g1, h0, h1, p0, p1;                   // Even-half intermediate.
17     short r0, r1;                                   // Even-half intermediate.
18     int P0, P1, R0, R1;                             // Even-half intermediate.
19     short g2, g3, h2, h3;                           // Odd-half intermediate.
20     short q0a, s0a, q0, q1, s0, s1;                 // Odd-half intermediate.
21     short Q0, Q1, S0, S1;                           // Odd-half intermediate.
22     int F0, F1, F2, F3, F4, F5, F6, F7;           // Freq. domain results.
23     int F0r, F1r, F2r, F3r, F4r, F5r, F6r, F7r;    // Rounded, truncated results.
24     unsigned i, j, i_1;

```

```

25  i_1 = 0;
26  for (i = 0; i < num_fdcts; i++) {
27      for (j = 0; j < N; j++) {
28          f0 = dct_io_ptr[ 0+i_1];
29          f1 = dct_io_ptr[ 8+i_1];
30          f2 = dct_io_ptr[16+i_1];
31          f3 = dct_io_ptr[24+i_1];
32          f4 = dct_io_ptr[32+i_1];
33          f5 = dct_io_ptr[40+i_1];
34          f6 = dct_io_ptr[48+i_1];
35          f7 = dct_io_ptr[56+i_1];
36          g0 = f0 + f7;
37          h2 = f0 - f7;
38          g1 = f1 + f6;
39          h3 = f1 - f6;
40          h1 = f2 + f5;
41          g3 = f2 - f5;
42          h0 = f3 + f4;
43          g2 = f3 - f4;
44          p0 = g0 + h0;
45          r0 = g0 - h0;
46          p1 = g1 + h1;
47          r1 = g1 - h1;
48          q1 = g2;
49          s1 = h2;
50          s0a= h3 + g3;
51          q0a= h3 - g3;
52          s0 = (s0a * c0 + 0x7FFF) >> 16;
53          q0 = (q0a * c0 + 0x7FFF) >> 16;
54          P0 = p0 + p1;
55          P1 = p0 - p1;
56          R1 = c6 * r1 + c2 * r0;
57          R0 = c6 * r0 - c2 * r1;
58          Q1 = q1 + q0;
59          Q0 = q1 - q0;
60          S1 = s1 + s0;
61          S0 = s1 - s0;
62          F0 = P0;
63          F4 = P1;
64          F2 = R1;
65          F6 = R0;
66          F1 = c7 * Q1 + c1 * S1;
67          F7 = c7 * S1 - c1 * Q1;
68          F5 = c3 * Q0 + c5 * S0;
69          F3 = c3 * S0 - c5 * Q0;
70          dct_io_tmp[ 0+i_1] = F0;
71          dct_io_tmp[ 8+i_1] = F1 >> 13;
72          dct_io_tmp[16+i_1] = F2 >> 13;
73          dct_io_tmp[24+i_1] = F3 >> 13;
74          dct_io_tmp[32+i_1] = F4;
75          dct_io_tmp[40+i_1] = F5 >> 13;
76          dct_io_tmp[48+i_1] = F6 >> 13;
77          dct_io_tmp[56+i_1] = F7 >> 13;
78          i_1++;
79      }
80      i_1 += 56;
81  }
82  i_1 = 0;

```

```

83  for (i = 0; i < N*num_fdcts; i++) {
84      f0 = dct_io_tmp[0+i_1];
85      f1 = dct_io_tmp[1+i_1];
86      f2 = dct_io_tmp[2+i_1];
87      f3 = dct_io_tmp[3+i_1];
88      f4 = dct_io_tmp[4+i_1];
89      f5 = dct_io_tmp[5+i_1];
90      f6 = dct_io_tmp[6+i_1];
91      f7 = dct_io_tmp[7+i_1];
92      g0 = f0 + f7;
93      h2 = f0 - f7;
94      g1 = f1 + f6;
95      h3 = f1 - f6;
96      h1 = f2 + f5;
97      g3 = f2 - f5;
98      h0 = f3 + f4;
99      g2 = f3 - f4;
100     p0 = g0 + h0;
101     r0 = g0 - h0;
102     p1 = g1 + h1;
103     r1 = g1 - h1;
104     q1 = g2;
105     s1 = h2;
106     s0a= h3 + g3;
107     q0a= h3 - g3;
108     q0 = (q0a * c0 + 0x7FFF) >> 16;
109     s0 = (s0a * c0 + 0x7FFF) >> 16;
110     P0 = p0 + p1;
111     P1 = p0 - p1;
112     R1 = c6 * r1 + c2 * r0;
113     R0 = c6 * r0 - c2 * r1;
114     Q1 = q1 + q0;
115     Q0 = q1 - q0;
116     S1 = s1 + s0;
117     S0 = s1 - s0;
118     F0 = P0;
119     F4 = P1;
120     F2 = R1;
121     F6 = R0;
122     F1 = c7 * Q1 + c1 * S1;
123     F7 = c7 * S1 - c1 * Q1;
124     F5 = c3 * Q0 + c5 * S0;
125     F3 = c3 * S0 - c5 * Q0;
126     F0r = (F0 + 0x0006) >> 3;
127     F1r = (F1 + 0x7FFF) >> 16;
128     F2r = (F2 + 0x7FFF) >> 16;
129     F3r = (F3 + 0x7FFF) >> 16;
130     F4r = (F4 + 0x0004) >> 3;
131     F5r = (F5 + 0x7FFF) >> 16;
132     F6r = (F6 + 0x7FFF) >> 16;
133     F7r = (F7 + 0x7FFF) >> 16;
134     dct_o[0+i_1] = F0r;
135     dct_o[1+i_1] = F1r;
136     dct_o[2+i_1] = F2r;
137     dct_o[3+i_1] = F3r;
138     dct_o[4+i_1] = F4r;
139     dct_o[5+i_1] = F5r;
140     dct_o[6+i_1] = F6r;

```

```

141     dct_o[7+i_1] = F7r;
142     i_1 += 8;
143 }
144 return F7r;
145 }

```

Algoritmo B.6: Exemplo *Fibonacci* descrito em C

```

1 #define N 32
2
3 main() {
4     int v[N] = {0, 1};
5     int i;
6     for (i=2; i<N; i++) {
7         v[i] = v[i-1] + v[i-2];
8     }
9 }

```

Algoritmo B.7: Exemplo *Fibonacci* descrito em C com reuso de dados

```

1 #define N 32
2
3 main() {
4     int V[N];
5     int a=1, b=0, c, i;
6     for (i = 0; i < N; i++) {
7         c=a+b;
8         V[i]=b;
9         b=a;
10        a=c;
11    }
12 }

```

Algoritmo B.8: Exemplo *Pop Count* descrito em C

```

1 #define N 1024
2
3 int A[N], B[N];
4
5 main() {
6     int i, j, sum, input;
7     for (i=0; i<N; i++) {
8         input = A[i];
9         sum = 0;
10        for (j = 0; j < 32; j++) {
11            sum += (input) & 1;
12            input = input/2;
13        }
14        B[i] = sum;
15    }
16 }

```

Algoritmo B.9: Exemplo *Sobel* descrito em C

```

1 #define cols 10

```

```

2 #define rows 10
3 #define N cols*rows
4
5 int main() {
6     char in[N];
7     char out[N];
8     int H, O, V, i;
9     int i00, i01, i02;
10    int i10, i12;
11    int i20, i21, i22;
12    for (i = 0; i < cols*(rows-2)-2; i++) {
13        i00=in[i]; i01=in[i+1]; i02=in[i+2];
14        i10=in[i+cols]; i12=in[i+cols+2];
15        i20=in[i+2*cols]; i21=in[i+2*cols+1]; i22=in[i+2*cols+2];
16        H = - i00 - 2*i01 - i02 +
17            + i20 + 2*i21 + i22;
18        V = - i00 + i02
19            - 2*i10 + 2*i12
20            - i20 + i22;
21        if (H<0)
22            H = -H;
23        if (V<0)
24            V = -V;
25        O = H + V;
26        if (O > 255) O = 255;
27        out[i + 1] = (char)O;
28    }
29 }

```

Algoritmo B.10: Exemplo *Vector Sum* descrito em C

```

1 #define N 2048
2
3 int A[N];
4 int B[N];
5 int C[N];
6
7 main() {
8     int i;
9     for(i = 0; i < N; i++) {
10        C[i] = A[i] + B[i];
11    }
12 }

```

B.2 Algoritmos Descritos em LALP

Algoritmo B.11: Exemplo *Autocorrelation* descrito em LALP

```

1 const N = 160;
2 const M = 10;
3
4 typedef fixed(32, 1) int;
5 typedef fixed(1, 0) bit;
6

```

```

7 autocor(in bit init, out bit done, out int output) {
8   {
9       int sd [N];
10      int ac[M], a, b;
11      fixed(8, 1) i, k, sd_addr;
12      int sum;
13   }
14   i.clk_en = init;
15   counter (i=0; i<M; i++@301);
16   k.clk_en = init;
17   k.load = i.step;
18   counter (k=M; k<N; k++@2);
19
20   sd.address = sd_addr;
21   sd_addr = k;
22   sd_addr = k - i when k.step@1;
23   a = sd.data_out;
24   b = sd.data_out;
25   sum += i.step ? 0 : a * b;
26
27   ac.address = i;
28   ac.data_in = sum >> 15;
29   output = ac.data_out;
30 }

```

Algoritmo B.12: Exemplo *FDCT* descrito em LALP

```

1  const C0 = 0xB505;
2  const C1 = 0x2C62;
3  const C2 = 0x29CF;
4  const C3 = 0x25A0;
5  const C5 = 0x1924;
6  const C6 = 0x1151;
7  const C7 = 0x08D4;
8
9  const N = 8;
10 const M = 64; //N*N
11 const num_fdcts = 640;
12 const SIZE = 1024; // < 640=num_fdcts*M;
13
14 typedef fixed(32,1) int;
15 typedef fixed(10,0) address;
16 typedef fixed(1,0) bool;
17
18 fdct(in bool init, out bool done, out int output) {
19   {
20       int dct_io_ptr[SIZE], dct_io_tmp[SIZE], dct_o[SIZE];
21       address i, j, i_1, i_plus_8, j_plus_64;
22       int f0, f1, f2, f3, f4, f5, f6, f7;           // Spatial domain samples
23       int g0, g1, h0, h1, p0, p1;                   // Even-half intermediate
24       int r0, r1;                                     // Even-half intermediate
25       int P_0, P_1, R_0, R_1;                         // Even-half intermediate
26       int g2, g3, h2, h3;                             // Odd-half intermediate
27       int q0a, s0a, q0, q1, s0, s1;                   // Odd-half intermediate
28       int Q_0, Q_1, S_0, S_1;                         // Odd-half intermediate
29       int F_0, F_1, F_2, F_3, F_4, F_5, F_6, F_7;    // Freq. domain results
30       int mf, xmf;
31       address dct_io_tmp_address;

```



```

32     address xi, xj;
33     int xf0, xf1, xf2, xf3, xf4, xf5, xf6, xf7;
34     int xg0, xh0, xp0, xr0;
35     int xg1, xh1, xp1, xr1;
36     int xP_0, xP_1, xR_0, xR_1;
37     int xg2, xh2, xg3, xh3;
38     int xq0a, xq0, xq1;
39     int xs0a, xs0, xs1;
40     int xQ_0, xQ_1, xS_0, xS_1;
41     int xF_0, xF_1, xF_2, xF_3, xF_4, xF_5, xF_6, xF_7;
42     int xF0r, xF1r, xF2r, xF3r, xF4r, xF5r, xF6r, xF7r;
43 }
44
45 counter (i=0; i<num_fdcts; i+=64@72);
46 i.clk_en = init;
47
48 i_plus_8 = i + 8;
49
50 counter (j=i; j<i_plus_8; j++@9);
51 j.clk_en = init;
52 j.load = i.step;
53
54 j_plus_64 = j + 64;
55
56 counter (i_1=j; i_1<j_plus_64; i_1+=8);
57 i_1.clk_en = init@2;
58 i_1.load = j.step;
59
60 dct_io_ptr.address = i_1;
61
62 f0 = dct_io_ptr.data_out when (j.step@4);
63 f1 = dct_io_ptr.data_out when (j.step@5);
64 f2 = dct_io_ptr.data_out when (j.step@6);
65 f3 = dct_io_ptr.data_out when (j.step@7);
66 f4 = dct_io_ptr.data_out when (j.step@8);
67 f5 = dct_io_ptr.data_out when (j.step@9);
68 f6 = dct_io_ptr.data_out when (j.step@10);
69 f7 = dct_io_ptr.data_out when (j.step@11);
70
71 g0 = f0 + f7;
72 h2 = f0 - f7;
73 g1 = f1 + f6;
74 h3 = f1 - f6;
75 h1 = f2 + f5;
76 g3 = f2 - f5;
77 h0 = f3 + f4;
78 g2 = f3 - f4;
79
80 p0 = g0 + h0;
81 r0 = g0 - h0;
82 p1 = g1 + h1;
83 r1 = g1 - h1;
84 q1 = g2;
85 s1 = h2;
86 s0a= h3 + g3;
87 q0a= h3 - g3;
88
89 s0 = (s0a *@6 C0 + 0x7FFF) >> 16;

```

```

90  q0 = (q0a *@6 C0 + 0x7FFF) >> 16;
91
92  R_1 = C6 *@6 r1 + C2 *@6 r0;
93  R_0 = C6 *@6 r0 - C2 *@6 r1;
94  P_0 = p0 + p1;
95  P_1 = p0 - p1;
96  Q_1 = q1 + q0;
97  Q_0 = q1 - q0;
98  S_1 = s1 + s0;
99  S_0 = s1 - s0;
100
101  F_1 = C7 *@6 Q_1 + C1 *@6 S_1;
102  F_7 = C7 *@6 S_1 - C1 *@6 Q_1;
103  F_5 = C3 *@6 Q_0 + C5 *@6 S_0;
104  F_3 = C3 *@6 S_0 - C5 *@6 Q_0;
105  F_0 = P_0;
106  F_4 = P_1;
107  F_2 = R_1;
108  F_6 = R_0;
109
110  mf = F_0;
111  mf = F_1 >> 13 when j.step@30;
112  mf = F_2 >> 13 when j.step@31;
113  mf = F_3 >> 13 when j.step@32;
114  mf = F_4 when j.step@33;
115  mf = F_5 >> 13 when j.step@34;
116  mf = F_6 >> 13 when j.step@35;
117  mf = F_7 >> 13 when j.step@36;
118
119  dct_io_tmp_address = xi.step ? xi : (i_1@26) when init;
120
121  dct_io_tmp.address = dct_io_tmp_address;
122
123  dct_io_tmp.data_in = mf when i_1.step@28;
124
125  counter (xi=0; xi<num_fdcts; xi++);
126  xi.clk_en = i.done@28;
127
128  counter (xj=0; xj<num_fdcts; xj+=8@8);
129  xj.clk_en = i.done@28;
130
131  xf0 = dct_io_tmp.data_out when xj.step@3;
132  xf1 = dct_io_tmp.data_out when xj.step@4;
133  xf2 = dct_io_tmp.data_out when xj.step@5;
134  xf3 = dct_io_tmp.data_out when xj.step@6;
135  xf4 = dct_io_tmp.data_out when xj.step@7;
136  xf5 = dct_io_tmp.data_out when xj.step@8;
137  xf6 = dct_io_tmp.data_out when xj.step@9;
138  xf7 = dct_io_tmp.data_out when xj.step@10;
139
140  xg0 = xf0 + xf7;
141  xh2 = xf0 - xf7;
142  xg1 = xf1 + xf6;
143  xh3 = xf1 - xf6;
144  xh1 = xf2 + xf5;
145  xg3 = xf2 - xf5;
146  xh0 = xf3 + xf4;
147  xg2 = xf3 - xf4;

```

```

148
149   xp0 = xg0 + xh0;
150   xr0 = xg0 - xh0;
151   xp1 = xg1 + xh1;
152   xr1 = xg1 - xh1;
153   xq1 = xg2;
154   xs1 = xh2;
155   xs0a= xh3 + xg3;
156   xq0a= xh3 - xg3;
157
158   xq0 = (xq0a *@6 C0 + 0x7FFF) >> 16;
159   xs0 = (xs0a *@6 C0 + 0x7FFF) >> 16;
160
161   xR_1 = C6 *@6 xr1 + C2 *@6 xr0;
162   xR_0 = C6 *@6 xr0 - C2 *@6 xr1;
163   xP_0 = xp0 + xp1;
164   xP_1 = xp0 - xp1;
165   xQ_1 = xq1 + xq0;
166   xQ_0 = xq1 - xq0;
167   xS_1 = xs1 + xs0;
168   xS_0 = xs1 - xs0;
169
170   xF_1 = C7 *@6 xQ_1 + C1 *@6 xS_1;
171   xF_7 = C7 *@6 xS_1 - C1 *@6 xQ_1;
172   xF_5 = C3 *@6 xQ_0 + C5 *@6 xS_0;
173   xF_3 = C3 *@6 xS_0 - C5 *@6 xQ_0;
174   xF_0 = xP_0;
175   xF_4 = xP_1;
176   xF_2 = xR_1;
177   xF_6 = xR_0;
178
179   xF0r = (xF_0 + 0x0006) >> 3;
180   xF1r = (xF_1 + 0x7FFF) >> 16;
181   xF2r = (xF_2 + 0x7FFF) >> 16;
182   xF3r = (xF_3 + 0x7FFF) >> 16;
183   xF4r = (xF_4 + 0x0004) >> 3;
184   xF5r = (xF_5 + 0x7FFF) >> 16;
185   xF6r = (xF_6 + 0x7FFF) >> 16;
186   xF7r = (xF_7 + 0x7FFF) >> 16;
187
188   xmf = xF0r;
189   xmf = xF1r when xj.step@34;
190   xmf = xF2r when xj.step@35;
191   xmf = xF3r when xj.step@36;
192   xmf = xF4r when xj.step@37;
193   xmf = xF5r when xj.step@38;
194   xmf = xF6r when xj.step@39;
195   xmf = xF7r when xj.step@40;
196
197   dct_o.data_in = xmf;
198   dct_o.address = xi;
199   output = dct_o.data_out;
200   done = xi.done;
201 }

```

Algoritmo B.13: Exemplo *Bubble Sort* descrito em LALP

```

1 const DATA_WIDTH = 32;

```

```

2  const ITERATIONS = 32;
3
4  typedef fixed(DATA_WIDTH, 1) int;
5  typedef fixed(1, 0) bit;
6
7  bubble_sort(in bit init , out fixed(DATA_WIDTH, 1) output , out bit done) {
8      {
9          int v[ITERATIONS];
10         int a, b, maior, menor, troca;
11         fixed(6, 0) i, iml, j, v_addr;
12     }
13     i.clk_en = init;
14     counter (i=0; i<31; i++@125);
15     iml = i + 1;
16     j.load = i.step;
17     j.clk_en = init;
18     counter (j=iml; j<32; j++@4);
19     a = v.data_out when j.step;
20     b = v.data_out when j.step@1;
21     maior = a;
22     maior = b when b > a;
23     menor = b;
24     menor = a when a < b;
25     troca = maior;
26     troca = menor when j.step@3;
27     v.address = v_addr;
28     v.data_in = troca;
29     v_addr = i;
30     v_addr = j when j.step | (j.step@2);
31     output = v.data_out;
32     done = i.done;
33 }

```

Algoritmo B.14: Exemplo *Max* descrito em LALP

```

1  const DATA_WIDTH = 32;
2  const ITERATIONS = 2048;
3
4  typedef fixed(DATA_WIDTH, 1) int;
5  typedef fixed(1, 0) bit;
6
7  max(in bit init , out fixed(DATA_WIDTH, 1) maxval , out bit done) {
8      {
9          int v [ITERATIONS];
10         int a, b;
11         fixed(12, 1) i;
12     }
13     counter (i=0; i<ITERATIONS; i+=1);
14     v.address = i;
15     a = v;
16     b = a when a > b;
17     maxval = b;
18 }

```

Algoritmo B.15: Exemplo *Pop Count* descrito em LALP

```

1  const DATA_WIDTH = 32;

```

```

2  const ITERATIONS = 1024;
3
4  typedef fixed(DATA_WIDTH, 1) int;
5  typedef fixed(1, 0) bit;
6
7  pop_cnt_alp(in bit init, out fixed(DATA_WIDTH, 1) cnt, out bit done) {
8      {
9          int a [ITERATIONS];
10         int b[10], sum, word;
11         fixed(11, 1) i;
12         int w00000000001;
13         int w00000000002;
14         int w00000000004;
15         int w00000000008;
16         int w00000000016;
17         int w00000000032;
18         int w00000000064;
19         int w00000000128;
20         int w00000000256;
21         int w00000000512;
22         int w00000001024;
23         int w00000002048;
24         int w00000004096;
25         int w00000008192;
26         int w00000016384;
27         int w00000032768;
28         int w00000065536;
29         int w00000131072;
30         int w00000262144;
31         int w00000524288;
32         int w0001048576;
33         int w0002097152;
34         int w0004194304;
35         int w0008388608;
36         int w0016777216;
37         int w0033554432;
38         int w0067108864;
39         int w0134217728;
40         int w0268435456;
41         int w0536870912;
42         int w1073741824;
43         int w2147483648;
44         int w01, w02, w03, w04, w05, w06, w07, w08, w09, w10;
45         int w11, w12, w13, w14, w15, w16, w17, w18, w19, w20;
46         int w21, w22, w23, w24, w25, w26, w27, w28, w29, w30, w31;
47     }
48     i.clk_en = init;
49     counter (i=0; i<ITERATIONS; i++);
50     word = a.data_out;
51
52     w00000000001 = word & 00000000001;
53     w00000000002 = (word & 00000000002)>>1;
54     w01 = w00000000001 + w00000000002;
55
56     w00000000004 = (word & 00000000004)>>2;
57     w00000000008 = (word & 00000000008)>>3;
58     w02 = w00000000004 + w00000000008;
59     w17 = w01 + w02;

```

```

60
61 w0000000016 = (word & 0000000016)>>4;
62 w0000000032 = (word & 0000000032)>>5;
63 w03 = w0000000016 + w0000000032;
64
65 w0000000064 = (word & 0000000064)>>6;
66 w0000000128 = (word & 0000000128)>>7;
67 w04 = w0000000064 + w0000000128;
68 w18 = w03 + w04;
69 w25 = w17 + w18;
70
71 w0000000256 = (word & 0000000256)>>8;
72 w0000000512 = (word & 0000000512)>>9;
73 w05 = w0000000256 + w0000000512;
74
75 w0000001024 = (word & 0000001024)>>10;
76 w0000002048 = (word & 0000002048)>>11;
77 w06 = w0000001024 + w0000002048;
78 w19 = w05 + w06;
79
80 w0000004096 = (word & 0000004096)>>12;
81 w0000008192 = (word & 0000008192)>>13;
82 w07 = w0000004096 + w0000008192;
83
84 w0000016384 = (word & 0000016384)>>14;
85 w0000032768 = (word & 0000032768)>>15;
86 w08 = w0000016384 + w0000032768;
87 w20 = w07 + w08;
88 w26 = w19 + w20;
89 w29 = w25 + w26;
90
91 w0000065536 = (word & 0000065536)>>16;
92 w0000131072 = (word & 0000131072)>>17;
93 w09 = w0000065536 + w0000131072;
94
95 w0000262144 = (word & 0000262144)>>18;
96 w0000524288 = (word & 0000524288)>>19;
97 w10 = w0000262144 + w0000524288;
98 w21 = w09 + w10;
99
100 w0001048576 = (word & 0001048576)>>20;
101 w0002097152 = (word & 0002097152)>>21;
102 w11 = w0001048576 + w0002097152;
103
104 w0004194304 = (word & 0004194304)>>22;
105 w0008388608 = (word & 0008388608)>>23;
106 w12 = w0004194304 + w0008388608;
107 w22 = w11 + w12;
108 w27 = w21 + w22;
109
110 w0016777216 = (word & 0016777216)>>24;
111 w0033554432 = (word & 0033554432)>>25;
112 w13 = w0016777216 + w0033554432;
113
114 w0067108864 = (word & 0067108864)>>26;
115 w0134217728 = (word & 0134217728)>>27;
116 w14 = w0067108864 + w0134217728;
117 w23 = w13 + w14;

```

```

118
119     w0268435456 = (word & 0268435456)>>28;
120     w0536870912 = (word & 0536870912)>>29;
121     w15 = w0268435456 + w0536870912;
122
123     w1073741824 = (word & 1073741824)>>30;
124     w2147483648 = (word & 2147483648)>>31;
125     w16 = w1073741824 + w2147483648;
126     w24 = w15 + w16;
127     w28 = w23 + w24;
128     w30 = w27 + w28;
129     w31 = w29 + w30;
130
131     a.address = i;
132     b.address = i when i.step@6;
133     b.data_in = w31;
134     cnt = b.data_out;
135 }

```

Algoritmo B.16: Exemplo *Vector Sum* descrito em LALP

```

1  const DATA_WIDTH = 32;
2  const ITERATIONS = 2048;
3
4  typedef fixed(DATA_WIDTH, 1) int;
5  typedef fixed(1, 0) bit;
6
7  vecsum(out int result, out bit done, in bit init) {
8      {
9          int x[ITERATIONS], y[ITERATIONS];
10         int z[ITERATIONS];
11         fixed(16, 0) i;
12     }
13     counter (i=0; i<ITERATIONS; i++);
14     i.clk_en = init;
15     x.address = i;
16     y.address = i;
17     z.address = i;
18     z.data_in = x + y;
19     result = z;
20 }

```