

Universidade Federal de São Carlos  
PPG/CC - Programa de Pós-Graduação em Ciência da Computação  
Mestrado em Ciência da Computação

**RTEV - Ambiente de Desenvolvimento de  
Aplicações Reconfiguráveis com o Kernel de Tempo  
Real Virtuoso**

*Mairum Ceoldo Andrade*

Orientador:

Prof. Dr. José Hiroki Saito

Co-orientador:

Prof. Dr. Célio Estevan Morón

SÃO CARLOS

Julho, 2006

**Ficha catalográfica elaborada pelo DePT da  
Biblioteca Comunitária da UFSCar**

A553ra

Andrade, Mairum Ceoldo.

RTEV - ambiente de desenvolvimento de aplicações reconfiguráveis com o kernel de tempo real Virtuoso / Mairum Ceoldo Andrade. -- São Carlos : UFSCar, 2009.  
109 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2006.

1. Computação reconfigurável. 2. FPGA - Field Programmable Gate Array. 3. Virtuoso (Sistema operacional de computador). 4. Tempo real. I. Título.

CDD: 004.22 (20<sup>a</sup>)

# Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

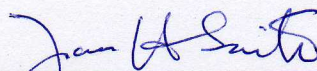
Programa de Pós-Graduação em Ciência da Computação

## *“RTEV – Ambiente de Desenvolvimento de Aplicações Reconfiguráveis com o Kernel de Tempo Real Virtuoso”*

MAIRUM CEOLDO ANDRADE

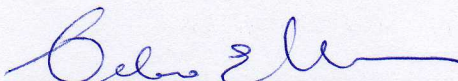
Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:



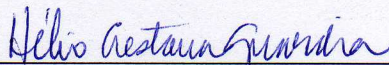
---

Prof. Dr. José Hiroki Saito  
(Orientador – DC/UFSCar)




---

Prof. Dr. Célio Estevan Moron  
(Co-Orientador - DC/UFSCar)



---

Prof. Dr. Hélio Crestana Guardia  
(DC/UFSCar)



---

Prof. Dr. Fábio Gonçalves Jota  
(Depto. de Engenharia Eletrônica/UFMG)

São Carlos  
Agosto/2006

*“Uma vitória que é ganha após uma luta feroz, e é louvada universalmente, não é o apogeu da excelência. Assim como o levantar de um fio de cabelo não é sinal de força, como ver o sol e a lua não é sinal de visão aguçada, tampouco escutar um trovão não é dom de audição aguda.”*

***Sun Tzu***

# *Agradecimentos*

Ao Prof. Dr. José Hiroki Saito, pela confiança e amizade, pelos ensinamentos, pela oportunidade de enveredar nos estudos de componentes reconfiguráveis, e extrema competência na orientação deste trabalho.

Ao Prof. Dr. Célio Estevan Morón, pelo apoio, auxílio no desenvolvimento e competência na coorientação deste trabalho.

Ao Prof. Dr. Hélio Crestana Guardia, pelo apoio, auxílio e considerações na banca de qualificação, que muito contribuíram no desenvolvimento deste trabalho.

A todos amigos do GAPIS e PPG-CC, pela amizade, apoio, convívio, troca de conhecimento, e momentos de descontração.

A minha família pelo amor, apoio, compreensão e incentivo que foram essenciais para conquista deste objetivo, e de todos os outros ao longo de toda a vida.

A todos, que de alguma forma, contribuíram para este objetivo fosse alcançado, direta ou indiretamente.

# *Sumário*

**Lista de Figuras**

**Lista de Tabelas**

**Resumo**

**Abstract**

<b>1</b>	<b>Introdução</b>	<b>14</b>
1.1	Motivação e Relevância . . . . .	14
1.2	Objetivos e Definição do Problema . . . . .	16
1.3	Estrutura do Texto da Dissertação . . . . .	16
<b>2</b>	<b>Revisão Sobre Arquiteturas Reconfiguráveis</b>	<b>18</b>
2.1	Hardware X Software . . . . .	18
2.2	Vantagens da Reconfiguração . . . . .	20
2.2.1	Tolerância a Falhas . . . . .	20
2.2.2	Prototipagem Rápida . . . . .	20
2.2.3	Tempo de Programação . . . . .	21
2.2.4	Área . . . . .	21
2.2.5	Custo Financeiro . . . . .	23
2.2.6	Reconfiguração Parcial/Dinâmica . . . . .	24
2.3	FPGA . . . . .	24
2.4	Características e definições de arquiteturas reconfiguráveis . . . . .	25

2.4.1	Acoplamento . . . . .	25
2.4.2	Granulosidade . . . . .	27
2.4.3	Capacidade de Reconfiguração . . . . .	28
2.5	Considerações Finais . . . . .	28
<b>3</b>	<b>Abordagens para o Desenvolvimento de Aplicações Usando Arquiteturas Recon-</b>	
	<b>figuráveis</b>	<b>30</b>
3.1	Uma visão geral . . . . .	30
3.2	Projetos de Propósito Geral . . . . .	33
3.2.1	Abordagem de Anotações e Abordagem Dirigida por Restrições . . . . .	34
3.2.1.1	SPC . . . . .	34
3.2.1.2	Streams-C . . . . .	34
3.2.1.3	Sea Cucumber . . . . .	35
3.2.1.4	SPARK . . . . .	35
3.2.1.5	Catapult-C . . . . .	36
3.2.2	Compilação Direta do Código Fonte . . . . .	36
3.2.2.1	ASC . . . . .	36
3.2.2.2	Handel-C . . . . .	37
3.2.2.3	Haydn-C . . . . .	37
3.2.2.4	Bach-C . . . . .	38
3.3	Projetos de Propósito Específicos . . . . .	38
3.3.1	Processamento Digital de Sinais . . . . .	38
3.3.2	Otimização da largura de dados . . . . .	39
3.3.3	Outros métodos de desenvolvimento . . . . .	40
3.3.3.1	CHAMPION . . . . .	40
3.3.3.2	IGOL Framework . . . . .	41
3.3.3.3	SA-C . . . . .	41

3.3.3.4	Framework para Firewall . . . . .	41
3.4	Outros Métodos . . . . .	42
3.4.1	Customização em tempo de execução . . . . .	42
3.4.2	Processadores Soft e Hard . . . . .	42
3.4.3	Hardware/software Codesign . . . . .	43
3.4.3.1	Garp . . . . .	43
3.4.3.2	NAPA C . . . . .	43
3.5	Considerações Finais . . . . .	44
<b>4</b>	<b>Sistemas (Hardware e Software) utilizados para o Desenvolvimento Proposto</b>	<b>45</b>
4.1	TEV . . . . .	46
4.1.1	GPP - Editor Gráfico . . . . .	48
4.1.2	GPP - Editor de Texto . . . . .	49
4.1.3	GPP - Gerador de Código . . . . .	50
4.2	Virtuoso . . . . .	50
4.2.1	Tarefa . . . . .	51
4.2.2	Semáforo . . . . .	51
4.2.3	Caixa de Mensagem . . . . .	51
4.2.4	Fila . . . . .	52
4.2.5	Canais . . . . .	52
4.2.6	Memória . . . . .	52
4.2.7	Recursos . . . . .	53
4.2.8	Temporizador . . . . .	53
4.3	Cluster de DSPs Atlas . . . . .	53
4.4	Kit XUP Virtex-II Pro . . . . .	54
4.5	Considerações Finais . . . . .	57



<b>5</b>	<b>Trabalho Desenvolvido - Ambiente R-TEV</b>	<b>58</b>
5.1	Visão Geral . . . . .	58
5.2	R-TEV - Reconfigurable Teaching Environment for Virtuoso . . . . .	59
5.3	Manipulação de Componentes Reconfiguráveis com R-TEV . . . . .	62
5.4	Biblioteca de Funções Reconfiguráveis . . . . .	63
5.5	Protocolo de Comunicação . . . . .	64
5.6	Socket . . . . .	68
5.7	Geração de Código . . . . .	69
5.8	Desenvolvimento das Funções Reconfiguráveis . . . . .	69
5.9	Considerações Finais . . . . .	70
<b>6</b>	<b>Estudo de Caso e Resultados Obtidos</b>	<b>72</b>
6.1	Prewitt . . . . .	72
6.2	Filtro Gaussiano . . . . .	74
6.3	O Desenvolvimento . . . . .	75
6.4	Resultados . . . . .	76
6.5	Implementação nos CLBs . . . . .	77
6.5.1	VHDL para Prewitt . . . . .	78
6.5.2	Prewitt SA-C . . . . .	79
6.6	Considerações Finais . . . . .	79
<b>7</b>	<b>Conclusão e Trabalhos Futuros</b>	<b>80</b>
7.1	Conclusão . . . . .	80
7.2	Trabalhos Futuros . . . . .	81
	<b>Referências</b>	<b>82</b>
	<b>Apêndice A – Tutorial VHDL</b>	<b>88</b>

A.1	INTRODUÇÃO . . . . .	88
A.1.1	Origem . . . . .	88
A.1.2	Vantagens . . . . .	88
A.1.3	Desvantagens . . . . .	88
A.1.4	Características . . . . .	89
A.1.5	Níveis de Abstração . . . . .	89
A.2	COMENTÁRIOS E NOTAÇÕES NA LINGUAGEM VHDL . . . . .	89
A.3	ESTRUTURA DE UM PROGRAMA VHDL . . . . .	90
A.4	LIBRARY . . . . .	90
A.5	ENTITY . . . . .	90
A.6	ARCHITECTURE . . . . .	91
A.7	DESCRIÇÃO DO COMPONENTE . . . . .	91
A.8	DESCRIÇÃO DO SINAL . . . . .	92
A.9	DESCRIÇÃO DA LÓGICA . . . . .	92
<b>Apêndice B – Socket do Virtuoso</b>		<b>95</b>
<b>Apêndice C – Socket do Intermediário</b>		<b>97</b>
<b>Apêndice D – Socket do Hardware Reconfigurável</b>		<b>100</b>
<b>Apêndice E – Código Fonte - Prewitt</b>		<b>104</b>
<b>Apêndice F – Código Fonte - Filtro Gaussiano</b>		<b>107</b>

# *Lista de Figuras*

1	Motivações para computação reconfigurável . . . . .	15
2	Computação Temporal (IDE, 2003) . . . . .	19
3	Computação Espacial (IDE, 2003) . . . . .	19
4	Comparação entre tipos de processadores . . . . .	20
5	Comparação entre <i>Binding Time</i> e Domínio Computacional (IDE, 2003) . . . . .	22
6	Comparação entre tamanho de instrução e tamanho de barramento (IDE, 2003) . . . . .	23
7	Diagrama de custo do elemento lógico ao longo dos últimos anos . . . . .	23
8	Estrutura interna simplificada de uma FPGA . . . . .	24
9	(a) Bloco lógico típico de uma FPGA; (b) Disposição das entradas do bloco lógico . . . . .	25
10	Formas de Acoplamento do Hardware Reconfigurável . . . . .	26
11	Três fluxos possíveis para a implementação de um circuito em um sistema reconfigurável (COMPTON; HAUCK, 2002) . . . . .	31
12	TEV - Ambiente visual para o desenvolvimento de programas paralelos de tempo-real . . . . .	46
13	Componentes do desenvolvimento de aplicações no TEV (RIBEIRO et al., 1998a) . . . . .	47
14	Conexão entre duas tarefas . . . . .	49
15	Cluster de DSPs Atlas . . . . .	54
16	Sistema ATLAS - Cluster de DSPs e processador principal . . . . .	54
17	Kit XUP Virtex-II Pro . . . . .	55
18	Diagrama de blocos do Kit XUP (XILINX, 2006) . . . . .	55
19	Diagrama do FPGA Virtex-II Pro - XC2VP30 (XILINX, 2005c) . . . . .	56
20	Componentes do desenvolvimento de aplicações no R-TEV . . . . .	61

21	Geração de código reconfigurável realizada pelo GPPR . . . . .	61
22	Sistema composto pelo ATLAS e XUP Virtex-II Pro . . . . .	62
23	Janela de Funções reconfiguráveis do R-TEV . . . . .	63
24	Arquivo descritor de função reconfiguráveis do R-TEV . . . . .	64
25	Organização dos dados na memória DDR do HR . . . . .	66
26	Apresentação gráfica dos pacotes do <i>socket</i> . . . . .	67
27	Diagrama de estados do protocolo de comunicação para o computador principal (a) e o HR (b) . . . . .	67
28	Máscaras (a) horizontal e (b) vertical . . . . .	72
29	Máscaras Gaussiana 3x3 . . . . .	74
30	Aplicação com as funções reconfiguráveis <i>Prewitt</i> e filtro <i>Gaussiano</i> . . . . .	75
31	Imagens Processadas: (a) Imagem original; (b) Filtro Prewitt; e (c) Filtro Gaus- siano . . . . .	76
32	Estrutura de um programa em VHDL . . . . .	90
33	Resultado da definição da interface através do exemplo de entity . . . . .	91
34	Elementos da descrição de architecture . . . . .	91
35	Componentes a) XOR_Gate e b) NOT_Gate . . . . .	93
36	Resultado do projeto usando a descrição estrutural: (1) associação dos sinais (port map) sobre os componentes usados; (2) interconexão dos componentes através dos sinais associados . . . . .	94

# *Lista de Tabelas*

1	Compiladores de propósito geral . . . . .	38
2	Valores do <i>Flag</i> . . . . .	65
3	Organização dos ponteiros de controle na memória DDR do HR . . . . .	65
4	Descrição dos pacotes do <i>socket</i> . . . . .	66
5	Resultados do Estudo de Caso . . . . .	77

# *Resumo*

Esta dissertação apresenta um ambiente para o desenvolvimento de aplicações reconfiguráveis em conjunto com o *kernel* de tempo real Virtuoso, denominado RTEV (*Reconfigurable Teaching Environment for Virtuoso*). Este trabalho é baseado no ambiente de desenvolvimento TEV (*Teaching Environment for Virtuoso*), desenvolvido no DC/UFSCar (Departamento de Computação da Universidade Federal de São Carlos). No ambiente RTEV é possível realizar o desenvolvimento de aplicações apenas selecionando uma função reconfigurável disponível na biblioteca de funções e realizar a interconexão com as demais funções da aplicação. Serão apresentados, para melhor entendimento: a computação reconfigurável; métodos de desenvolvimento de funções e/ou aplicações reconfiguráveis que podem ser utilizadas no desenvolvimento das funções da biblioteca reconfigurável; os sistemas utilizados no desenvolvimento; o ambiente RTEV; e um estudo de caso como validação da utilização do ambiente de desenvolvimento. Com este trabalho, busca-se simplificar e facilitar o acesso de programadores sem experiência em computação reconfigurável ao desenvolvimento de aplicações que fazem uso deste tipo de arquitetura.

# *Abstract*

This dissertation presents an environment for the development of reconfigurable applications, with Virtuoso Real-Time Kernel, called RTEV. This work is based on the TEV (Teaching Environment for Virtuoso), developed at DC/UFSCar (Department of Computation of the Federal University of Sao Carlos). In RTEV, it is possible to carry through the development of applications only selecting a reconfigurable function from the library of available functions and making the interconnections with other functions of the applications. It will be presented, for better understanding: a reconfigurable computer review; development methods of functions and/or applications that can be used for the construction of the reconfigurable library; the systems used in the development environment RTEV; and a case study as a way of validation of the development environment. This environment allows the programmers, without any reconfigurable hardware knowledge, to develop applications with reconfigurable components.

# 1 *Introdução*

Neste capítulo serão apresentadas a motivação e relevância para este trabalho, definição do problema e dos objetivos. E será apresentada a estrutura do texto.

## 1.1 **Motivação e Relevância**

O aumento constante na densidade de transistores dos *chips*, tanto para uso geral quanto para sistemas embarcados, é propiciado pela melhoria da tecnologia de produção. Porém, isso faz com que os chips fiquem cada vez mais sofisticados e de custo elevado. A Figura 1a apresenta o gráfico de crescimento da quantidade de transistores (em milhões) por *chip* comparado com a melhoria na tecnologia de fabricação, diminuindo a dimensão de um transistor em nm (nano-metros), ilustrando a afirmação anterior. Além disso o aumento da complexidade dos algoritmos, requerendo cada vez maior densidade de transistores para realizar os cálculos em tempo satisfatório, excede a *lei de Moore*. Segundo a lei de Moore, a cada 18 meses a integração dos circuitos dos computadores dobra, enquanto os custos permanecem constantes. Por outro lado, o aumento da capacidade de processamento exigido pelo aumento da complexidade dos algoritmos esbarra no baixo crescimento da capacidade das baterias dos sistemas, principalmente em sistemas embarcados. Cria-se assim uma severa limitação de consumo de energia, nos sistemas embarcados onde este tipo de problema é crítico. A Figura 1b mostra a taxa de crescimento para a complexidade dos algoritmos, para a tecnologia de computadores (Lei de Moore) e a capacidade de bateria.

Esses problemas dificilmente serão satisfeitos pelos processadores tradicionais nos próximos anos. Uma alternativa aos processadores tradicionais são os computadores, ou arquiteturas, reconfiguráveis. A principal característica dos sistemas reconfiguráveis é a possibilidade de se configurar, ou utilizar, diversos circuitos diferentes, em um único *hardware* reconfigurável. Com isso, supera-se os problemas de necessidade de aumento de quantidade de transistores e consequentemente de consumo de energia, ampliando, desta forma, o leque de aplicações com uma tecnologia de circuitos reconfiguráveis disponível no mercado.



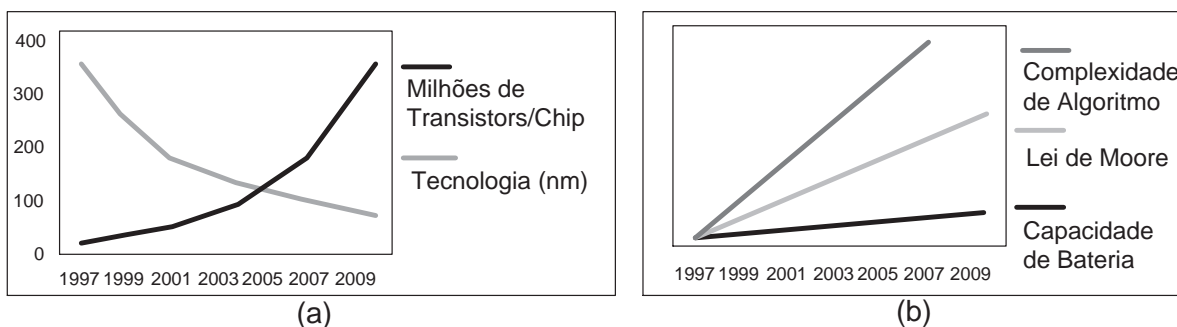


Figura 1: Motivações para computação reconfigurável

A constante evolução no desenvolvimento da computação reconfigurável e conseqüentemente sua maior utilização em sistemas computacionais convencionais, vem apresentando resultados que demonstram o potencial de tal arquitetura e sua viabilidade de projeto, implementação e utilização (DEHON, 2000).

O projeto de aplicações para este tipo de arquitetura requer grande conhecimento e especialização, sendo necessário que o desenvolvedor, além de conhecer bem a aplicação tenha um profundo e detalhado conhecimento da arquitetura reconfigurável específica para a sua aplicação. Devido a esta dificuldade de desenvolvimento e, principalmente, a dificuldade de integração e inclusão de sistemas reconfiguráveis em sistemas convencionais, os desenvolvedores costumam não fazer uso deste tipo de computação.

Um grande número de metodologias, *frameworks*, linguagens e extensões de linguagens tem sido disponibilizado para auxiliar e simplificar o desenvolvimento de aplicações reconfiguráveis. Entretanto, a integração do hardware reconfigurável com sistemas convencionais ainda é complexa. A razão principal é que não é trivial, para desenvolvedores que não possuem conhecimento em computação reconfigurável, a implementação de interação entre os sistemas convencional e reconfigurável ou protocolos de comunicação.

Este trabalho apresenta um ambiente de desenvolvimento amigável para aplicações reconfiguráveis, tornando o hardware reconfigurável quase transparente ao programador. O sistema proposto permite ao programador, sem qualquer conhecimento de hardware reconfigurável, desenvolver aplicações utilizando componentes reconfiguráveis.

O desenvolvimento consistiu em integrar a programação de hardware reconfigurável no TEV (Teaching Environment for Virtuoso), um ambiente visual para o desenvolvimento de aplicações paralelas de tempo real, desenvolvido no Departamento de Computação da Universidade Federal de São Carlos (DC/UFSCar) (MORÓN et al., 2000), para o sistema operacional de tempo real Virtuoso (WIND, 2001). TEV é um ambiente que permite desenvolver aplicações paralelas

de tempo real simplesmente utilizando grafos, com nós representando as funções e linhas denotando a comunicação entre as funções. O sistema proposto é chamado R-TEV (*Reconfigurable-TEV*). Como estudo de caso do R-TEV, é utilizada uma placa XUP (Xilinx University Program) com um FPGA (Field Programmable Gate Array) Virtex-II Pro XC2VP30.

Sendo assim, a motivação deste trabalho é possibilitar o desenvolvimento de aplicações para computadores reconfiguráveis pelos programadores sem conhecimentos específicos deste tipo de arquitetura, explorando diversas formas de processamento, principalmente paralela de granulosidade fina.

## 1.2 Objetivos e Definição do Problema

O objetivo deste trabalho é a implementação de um ambiente de desenvolvimento de aplicações utilizando computações reconfiguráveis. Esse ambiente permite a um projetista realizar o desenvolvimento de aplicações apenas escolhendo os recursos disponibilizados de computação reconfigurável e realizando a interconexão com as demais funções.

No entanto, é necessário que as funções reconfiguráveis tenham sido desenvolvidas previamente e disponibilizadas na biblioteca reconfigurável, composta por um conjunto de funções reconfiguráveis.

## 1.3 Estrutura do Texto da Dissertação

A presente descrição está dividida em 7 partes além das referências bibliográficas e apêndices:

- *Capítulo 1* - Apresenta a motivação e relevância do trabalho, assim como, os objetivos e definição do problema.
- *Capítulo 2* - Apresenta uma revisão sobre as arquiteturas reconfiguráveis, mostrando suas características e motivos pelo qual optamos por este tipo de arquitetura.
- *Capítulo 3* - Apresenta diversas metodologias para o desenvolvimento de aplicações reconfiguráveis, mostra a ampla gama de possibilidades existentes para o desenvolvimento dos mais variados tipos de aplicações.
- *Capítulo 4* - Apresenta os sistemas utilizados no desenvolvimento. Apresenta o ambiente TEV - Teaching Environment for Virtuoso, o Kernel Virtuoso, Cluster de DSPs Atlas

e a placa de desenvolvimento de aplicações reconfiguráveis XUP (*Xilinx University Program*) Virtex-II Pro, dando uma visão geral de suas estruturas, funcionamento e potencial.

- *Capítulo 5* - Apresenta o trabalho desenvolvido e a sua implementação com a adaptação do ambiente de desenvolvimento TEV para o desenvolvimento de aplicações utilizando-se funções reconfiguráveis. Apresenta também o trabalho realizado, detalhando cada etapa.
- *Capítulo 6* - Apresenta um estudo de caso utilizando o ambiente R-TEV.
- *Capítulo 7* - Conclusões e Proposições Futuras.

## 2 *Revisão Sobre Arquiteturas Reconfiguráveis*

No início da década de 1980, surgiram os primeiros circuitos FPGAs (*Field Programmable Gate Array*), componentes reconfiguráveis, inicialmente utilizados para facilitar o desenvolvimento de protótipos de circuitos, devido à sua flexibilidade. Posteriormente, esses circuitos começaram a ser utilizados para aumentar o desempenho de processamento, diminuindo a barreira entre hardware e software, com a implementação diretamente em Hardware (HW) de algoritmos que têm baixo desempenho em Software (SW).

As arquiteturas de computador que fazem uso deste tipo de circuitos e de suas características de reconfiguração são denominadas arquiteturas reconfiguráveis que serão apresentadas neste capítulo.

As duas primeiras seções deste capítulo apresentam algumas características de grande relevância, como tolerância a falhas e prototipagem rápida. Na seqüência, mostram-se as diferenças entre a computação temporal e a computação espacial, e os tipos de computações quanto ao seu tempo de programação e área de implementação utilizada. Apresenta-se ainda a evolução do custo dos elementos lógicos e a possibilidade de reconfiguração parcial e dinâmica.

### 2.1 **Hardware X Software**

Os processadores de propósito geral (microprocessadores) utilizados em computadores pessoais (PCs) são capazes de resolver qualquer tipo de computação, utilizando diferentes programas (SW), cada qual com seu propósito específico. Este tipo de computação é conhecido como computação temporal ou software/temporal, em que os elementos de processamento são distribuídos no tempo, isto é, há um pequeno número de blocos de processamento genéricos que são reutilizados várias vezes no tempo. Na Figura 2, é apresentado um exemplo de computação temporal em que se calcula uma expressão aritmética  $Ax^2 + Bx + C$  e posteriormente atribui-se o resultado à variável  $y$ . A expressão é calculada sequencialmente utilizando variáveis tem-

porárias  $t_1$  e  $t_2$ . Inicialmente,  $t_1$  recebe o valor  $x(t_1 := x)$ ; em seguida  $t_2$  recebe  $A * t_1$ ; seguido de  $t_2 + B$  e  $t_2 * t_1$ ; finalmente,  $y$  recebe  $t_2 + C$ . Os valores  $A$ ,  $B$ ,  $C$ ,  $t_1$  e  $t_2$  são variáveis contidas em registradores e os cálculos são realizados através de uma única ULA (Unidade Lógica e Aritmética). Na grande maioria das aplicações computacionais, este tipo de processador é a solução mais simples e oferece ótimos resultados de custo/benefício. Apresenta facilidade e rapidez no desenvolvimento de aplicações, devido ao uso de linguagens de alto nível.

Este tipo de processador possui uma gama muito grande de microinstruções disponíveis, para os mais variados tipos de aplicação, bem como o HW necessário para realizar todas as execuções de forma sequencial. Em contrapartida, na maior parte do tempo apenas uma pequena porção dos recursos disponíveis está sendo utilizado, levando a um desperdício de recursos e baixando a eficiência.

De lado oposto temos os processadores e circuitos específicos para determinadas aplicações conhecidos como ASICs (*Application Specific Integrated Circuit*). Esses processadores são desenvolvidos para aplicações que necessitam de certas características que não são típicas dos processadores de propósito geral, tais como alta velocidade de processamento, baixo consumo de energia, pequena dissipação de calor, tamanho reduzido e baixo custo.

ASICs são compostos de circuitos específicos desenvolvidos em hardware que executam de forma espacial o cômputo, ou seja, cada operador lógico/aritmético se encontra em uma determinada posição no espaço, configurando a aplicação. Este tipo de computação, conhecida como hardware/espacial (DEHON, 1996) (MIRSKY, 1996), permite a exploração mais eficiente do paralelismo de dados, a fim de obter maior desempenho e menor latência, como ilustrado na Figura 3. Nessa figura é possível verificar o mesmo cálculo mostrado na Figura 2 realizado espacialmente, em paralelo, utilizando 3 multiplicadores e dois somadores dispostos espacialmente.

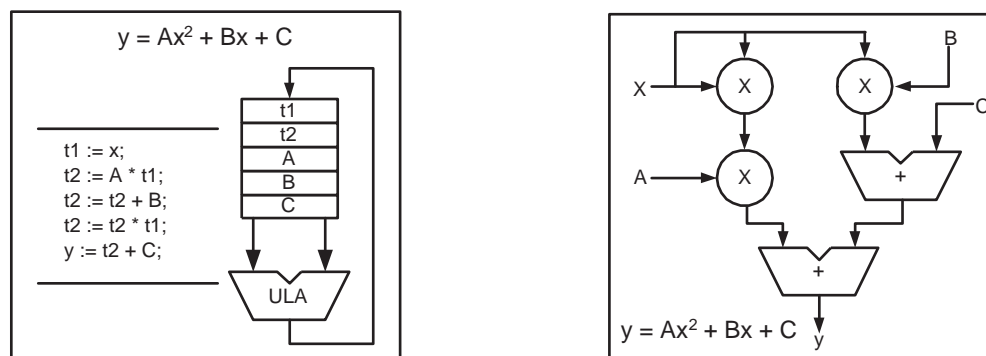


Figura 2: Computação Temporal (IDE, 2003)    Figura 3: Computação Espacial (IDE, 2003)

Devido às características e especificidades de cada aplicação, o desenvolvimento de com-

putadores específicos para cada uma delas é, em geral, complexo e caro, demandando mão de obra muito especializada e tempo de desenvolvimento e custo elevados.

Como já citado na introdução, os FPGAs tornaram possível os Processadores Reconfiguráveis que cobrem o espaço intermediário entre os dois tipos de processadores. A Figura 4 apresenta um gráfico onde se verifica a situação de cada um dos processadores no espaço desempenho versus flexibilidade. As arquiteturas reconfiguráveis tendem a fazer uso da computação espacial, para aplicações específicas, implementando em HW aplicações que possuem baixo desempenho em SW. Além disso possuem flexibilidade para criação de aplicações diversas, utilizando um mesmo chip, e desenvolvimento mais simples, rápido e de custo menor em relação aos ASICs.

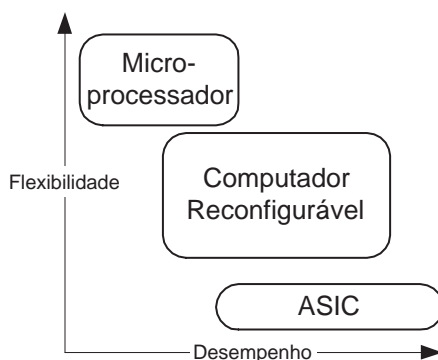


Figura 4: Comparação entre tipos de processadores

## 2.2 Vantagens da Reconfiguração

### 2.2.1 Tolerância a Falhas

Em um sistema computacional composto por um conjunto de unidades de processamento, o tratamento de possíveis falhas é necessário. Esta é uma outra área onde a computação reconfigurável tem sido aplicada. Sistemas tolerantes a falhas reconfiguráveis são capazes de inutilizar a área defeituosa e corrigi-la, enquanto o restante do sistema continua em operação. Já uma falha detectada em arquiteturas convencionais, implementadas em Circuitos Integrados(CI) não-reconfiguráveis, implica em inutilização completa dos CIs (MESQUITA, 2002).

### 2.2.2 Prototipagem Rápida

Alterando-se os dados de reconfiguração de um circuito integrado FPGA é possível programar os componentes e a fiação necessária para interconexão entre os blocos lógicos internos. O

tempo de implementação de um protótipo para uma dada aplicação é bastante curto, consistindo basicamente no tempo de projeto usando algum ambiente de desenvolvimento. Essa é uma clara distinção em relação aos projetos convencionais de VLSI (*Very Large Scale Integration*), onde decorrem muitas semanas entre o projeto e a implementação real devido à necessidade de fabricação do circuito, incluindo todos os processos de microeletrônica, até o encapsulamento final. Esta característica de FPGAs cria oportunidade para novas metodologias de projeto.

### 2.2.3 Tempo de Programação

O tempo de programação, ou *Binding Time*, é uma característica importante para a distinção dos três tipos de computação: (a) *hardware/espacial*; (b) *software/temporal*; e (c) *reconfigurável*. No caso de circuitos integrados convencionais, a programação e a interconexão dos dispositivos são feitas antes e durante o processo de fabricação respectivamente, sob especificação do fabricante. Os componentes ASIC's, que são circuitos integrados específicos para cada aplicação, são programados sob especificação do usuário; entretanto esta programação é fixa e pós-fabricação. Microprocessadores convencionais necessitam de busca/decodificação e execução de instruções da memória a cada ciclo e, portanto, a programação do circuito para a execução de instruções é feita em todos os ciclos. No presente contexto, entende-se como programação o ato de especificar a forma de funcionamento para o circuito. Já em arquiteturas reconfiguráveis, a configuração do dispositivo é programável, através de bits de seleção de contexto, ou "instrução". Cada bit da "instrução" corresponde à função de um determinado operador da arquitetura (DEHON; WAWRZYNEK, 1999). A Figura 5 mostra a distribuição desses modelos de arquitetura de acordo com seu tempo de programação, momento em que ocorre a programação do componente e domínio computacional, sistemas com computação puramente temporal, parte temporal e parte espacial e puramente espacial (HARTENTEIN, 2001). Incluem-se também nessa figura os arranjos de processadores convencionais e os arranjos sistólicos, que são arranjos de pequenos processadores que funcionam sincronamente para o processamento vetorial.

### 2.2.4 Área

O tipo de "instrução" praticamente define/distingue dispositivos configurados pós-fabricação. Arquiteturas convencionais utilizam um conjunto de instruções fixas, normalmente 32/64 bits, que definem o funcionamento do fluxo de dados; já arquiteturas reconfiguráveis muitas vezes são compostas por um conjunto de operadores aritméticos unários, cujo controle é feito por apenas um bit de configuração. Durante o projeto de processadores a área de silício ocupada

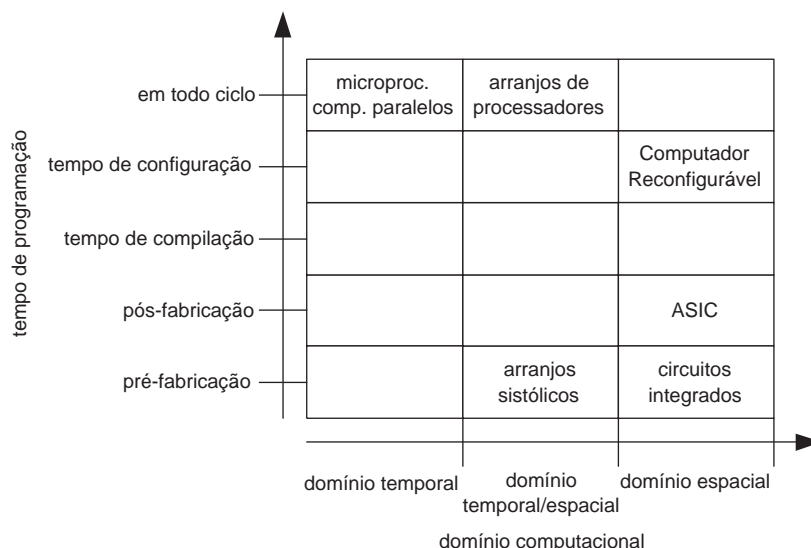


Figura 5: Comparação entre *Binding Time* e Domínio Computacional (IDE, 2003)

é basicamente definida por:

- tamanho do barramento de dados; e
- tamanho da instrução.

Nas instruções de processadores convencionais todos os bits de uma determinada instrução devem obrigatoriamente executar/participar da mesma operação, em cada ciclo, sejam dados manipulados da memória ou de registradores. Em componentes ASIC's é possível projetar um barramento de dados e tamanho de instrução coerente com o tipo de aplicação. Por outro lado, em computadores de propósito geral existe a preocupação de que o *chip* deve tanto suportar computações de granularidade grossa (mais adiante neste capítulo sera discutido a granulosidade de unidades computacionais), como pequenos cálculos aritméticos. Neste ponto, cada instrução é maior ou igual ao tamanho máximo que cada ULA suporta.

Circuitos reconfiguráveis baseados em FPGA's podem ser comparados com computadores que manipulam instruções simples de um bit, o que resulta em baixo *overhead* de processamento, pois o tamanho do circuito necessário para cada instrução é sempre ajustado para o tamanho ótimo para o cômputo. Assim a quantidade de circuitos para instruções é minimizada, aumentando a capacidade do *chip* para novos tipos de computação. Isto pode ser creditado à economia de espaço por adequar o tamanho de implementação de instruções, barramento e unidades à necessidade específica de cada aplicação, utilizando somente os bits necessários para cada instrução.



A Figura 6 ilustra um comparativo da relação do tamanho da instrução com o tamanho do barramento. Nota-se que os circuitos reconfiguráveis conseguem operar sobre um mesmo tamanho de barramento que processadores convencionais, utilizando um tamanho reduzido de instruções (DEHON; WAWRZYNEK, 1999)

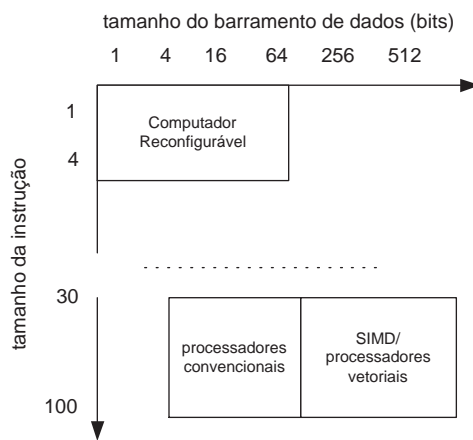


Figura 6: Comparação entre tamanho de instrução e tamanho de barramento (IDE, 2003)

## 2.2.5 Custo Financeiro

O principal componente de uma arquitetura reconfigurável, sem dúvida, é o FPGA: os elementos lógicos que o compõem são combinados de forma a se obter o desempenho desejado. Nota-se na Figura 7 (ALTERA, 2006a) que, de 1993 até os dias atuais o custo por elemento lógico, de um FPGA, vem sofrendo uma constante queda, cerca de 40% por ano e ao mesmo tempo, há um crescente aumento da quantidade de unidades lógicas por área.

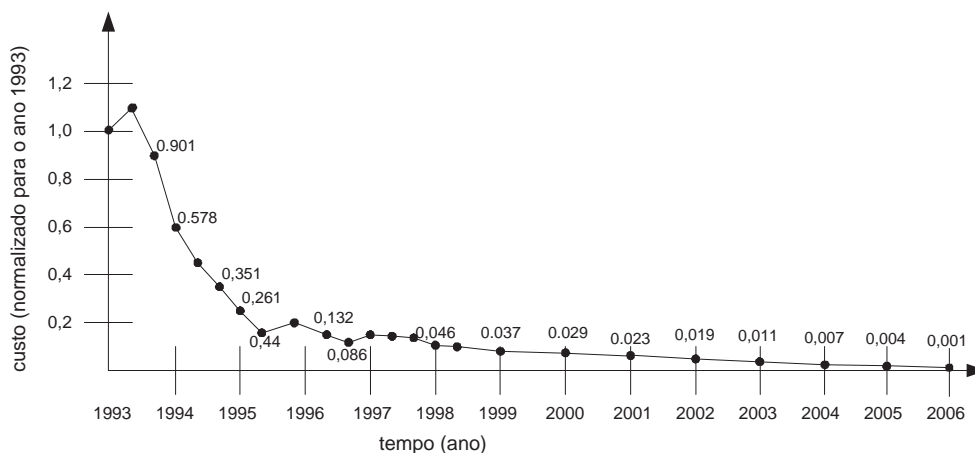


Figura 7: Diagrama de custo do elemento lógico ao longo dos últimos anos

## 2.2.6 Reconfiguração Parcial/Dinâmica

Os FPGA's mais modernos possuem a característica de reconfiguração em tempo real. Esta propriedade permite que, enquanto parte do sistema está em processamento, uma outra parte pode estar sendo reconfigurada para executar outro conjunto de instruções. Dessa forma, também é possível a reutilização de *hardware*, contribuindo para economia de recursos computacionais (MESQUITA, 2002).

## 2.3 FPGA

FPGA é um dispositivo que possui componentes lógicos programáveis (blocos lógicos) e interconexões programáveis (Switch Matrix) que podem ser configurados de acordo com as aplicações do usuário (programador). Os componentes programáveis podem ser programados de forma a se comportar como portas lógicas básicas (como E, OU, NÃO, OU-EXCLUSIVO) ou funções combinatórias mais complexas como decodificadores ou funções matemáticas simples. Na maioria dos FPGAs estes componentes programáveis podem ser elementos de memórias, que podem ser simples *flip-flops* ou blocos de memórias completos, multiplicadores ou funções mais avançadas. Os FPGAs são compostos ainda por blocos de entrada e saída (IOB - *Input Output Block*) que são responsáveis pelo interfaceamento com componentes externos. Os blocos lógicos são dispostos em forma bidimensional, e os canais de roteamento interligam as interconexões programáveis em formas de trilhas verticais e horizontais entre as linhas e as colunas dos blocos lógicos conforme mostra a Figura 8.

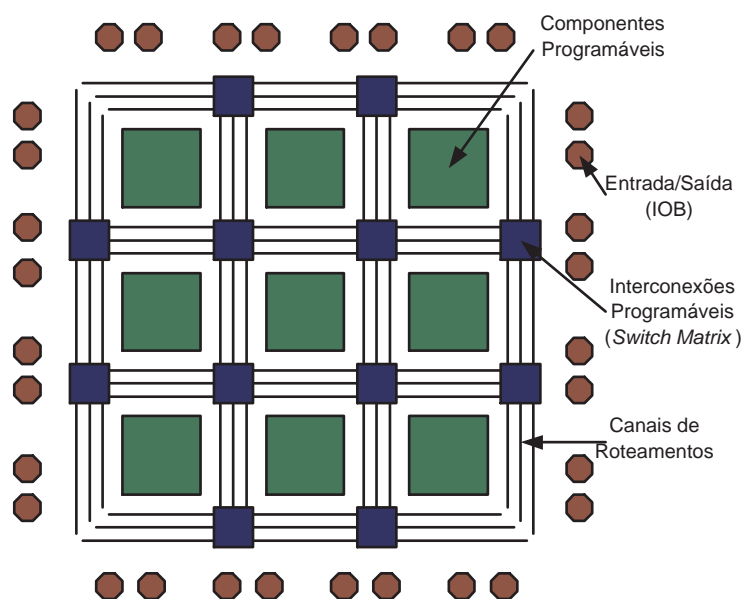


Figura 8: Estrutura interna simplificada de uma FPGA

Um bloco lógico típico de um FPGA é composto por uma *lookup table* (LUT) de 4 entradas, e um *flip-flop* tipo D como mostrado na Figura 9a. Há apenas uma saída, que pode ser registrada (fazendo uso de registrador) ou não-registrada. Cada bloco lógico tem quatro entradas para LUT e um ciclo de do relógio (clock), estando cada entrada disposta em um lado do bloco permitindo que este se conecte com seus quatro vizinhos (Figura 9b).

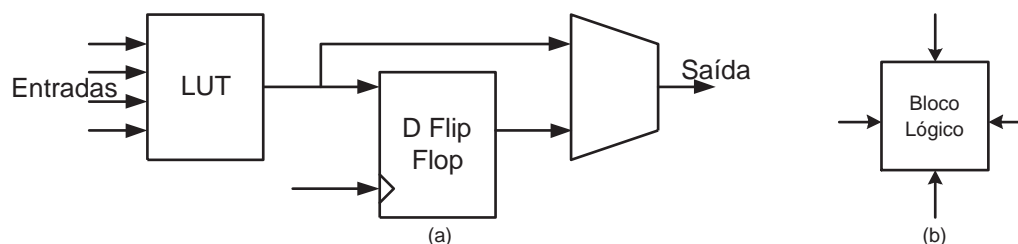


Figura 9: (a) Bloco lógico típico de uma FPGA; (b) Disposição das entradas do bloco lógico

FPGAs modernos expandiram suas capacidades e incluíram funcionalidades de alto nível implementadas diretamente em HW. Com essas funcionalidades embarcadas reduz-se a área utilizada e a complexidade de desenvolvimento das aplicações e aumenta-se o desempenho. Como exemplos pode-se citar multiplicadores, aritmética de ponto flutuante, blocos genéricos de DSP, processadores, E/S de alta velocidade e memória interna.

## 2.4 Características e definições de arquiteturas reconfiguráveis

Existem vários tipos diferentes de arquiteturas reconfiguráveis. Para classificá-los devemos considerar tanto o paradigma de *hardware* como de *software*. São apresentadas a seguir algumas características relevantes para definição e entendimento dos mais variados tipos de arquiteturas reconfiguráveis.

### 2.4.1 Acoplamento

Uma das principais diferenças entre as arquiteturas reconfiguráveis é o grau de acoplamento das unidades reconfiguráveis com o processador principal. Como certos tipos de operações não são muito eficientes em lógica programável, como por exemplo laços, saltos, desvios e E/S, os processadores reconfiguráveis estão geralmente acoplados a um microprocessador que fica responsável por executar tais tarefas. Podemos diferenciar o grau de acoplamento em quatro níveis (HAUCK, 1998) (COMPTON; HAUCK, 2000). A Figura 10 mostra os diferentes níveis de

acoplamento de unidades reconfiguráveis em destaque, quais sejam: UF (unidade funcional), coprocessador, unidade acoplada e unidade externa.

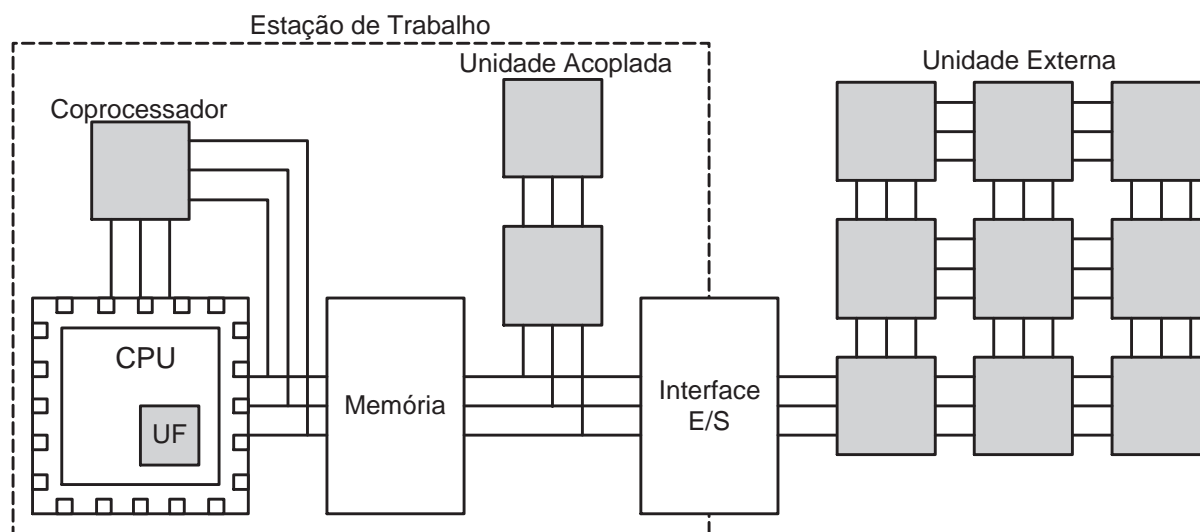


Figura 10: Formas de Acoplamento do Hardware Reconfigurável

No primeiro nível, o hardware reconfigurável é utilizado apenas como uma unidade funcional do processador principal. Assim, permite-se uma programação tradicional com a adição de instruções específicas que podem ser alteradas conforme a necessidade. Aqui, a unidade reconfigurável é executada utilizando o barramento principal do processador principal, com registradores usados para guardar operandos de entrada e saída.

No segundo nível, a unidade reconfigurável é utilizada como um coprocessador. Um coprocessador é, em geral, maior que uma unidade funcional e realiza o cômputo sem supervisão constante do processador principal. O processador principal inicializa a reconfiguração e envia os dados ou provê informação de onde os dados estão na memória.

No terceiro nível, uma unidade de reconfiguração comporta-se como um processador em um sistema multiprocessado. O *cache* de dados do processador principal não pode ser acessado diretamente pelo processador reconfigurável. Neste caso há um atraso maior na comunicação entre os processadores. Entretanto, este tipo de hardware reconfigurável permite uma ampla gama de processamento independente, enviando o resultado de uma grande quantidade de processamento de uma única vez para o processador principal.

Finalmente, o mais fracamente acoplado dos processadores reconfiguráveis são as unidades de processamento externas (*stand-alone*). Este tipo de hardware reconfigurável possui pouca comunicação com o processador principal, devido ao alto custo de comunicação. Este modelo é similar a estações de redes, onde o processamento ocorre por um longo período de tempo, sem muita comunicação.

Cada um destes tipos de acoplamento tem suas vantagens e desvantagens. Quanto mais integrado o *hardware* reconfigurável estiver com o processador principal, com mais frequência poderá ser solicitado para computações menores, devido ao baixo custo de comunicação. Entretanto, o *hardware* é incapaz de processar por longo período de tempo sem a intervenção do processador principal, pois utiliza recursos deste, e a lógica reconfigurável disponível, geralmente, é um tanto limitada. Quanto mais distante estiver do processador maior será a capacidade de execução autônoma. Entretanto, em aplicações que possuem um alto custo de comunicação, o ganho pode ser reduzido ou até mesmo ultrapassado pelo atraso na comunicação.

## 2.4.2 Granulosidade

Além do nível de acoplamento, pode-se variar o tamanho dos blocos lógicos. Cada unidade computacional, ou bloco lógico, pode variar desde uma simples porta lógica de 3 entradas a uma complexa ULA (Unidade Lógica e Aritmética). Esta diferença de tamanhos é classificada como granulosidade do sistema e pode ser fina, média ou grossa.

**Grão Fino:** Blocos lógicos de grão fino são normalmente constituídos por portas lógicas e multiplexadores. São geralmente utilizados para manipulação em nível de bit, e são frequentemente encontradas em aplicações de criptografia e processamento de imagens. Além disso, permitem um grande potencial de exploração do paralelismo da aplicação e flexibilidade. Entretanto, o *overhead* de comunicação, devido à grande quantidade de blocos, pode ser um ponto crítico.

**Grão Médio:** Intermediário entre o grão fino e grosso, é utilizado em grande parte de sistemas reconfiguráveis. É capaz de executar um conjunto de instruções por ciclo e provê estruturas computacionais mais eficientes para problemas mais complexos, como por exemplo máquinas de estados finitos. Em geral utiliza duas ou mais palavras de 4-bits.

**Grão Grosso:** São blocos lógicos grandes e complexos, otimizados para grandes cálculos, capazes de executar programas inteiros em poucos ciclos. Devido à otimização destes blocos para computações grandes, obtém melhores resultados (e utilizam menor área em chip) do que um conjunto de pequenas células conectadas formando o mesmo tipo de estrutura. Entretanto, devido à sua composição estática, são incapazes de realizar otimizações no tamanho dos operandos.

### 2.4.3 Capacidade de Reconfiguração

O sistemas reconfiguráveis podem diferir também de acordo com o momento, a quantidade de vezes e como ocorrem as configurações. Quanto à forma de reconfiguração, os sistemas podem ser divididos em 4 grupos (COMPTON; HAUCK, 2002):

**Estático:** O sistema é configurado uma única vez e nunca é modificado. O dispositivo é programado para executar uma determinada função e não é alterado durante toda a vida útil do sistema. Este grupo não tira vantagem da flexibilidade provida pela reconfiguração, utiliza-se apenas da facilidade de desenvolvimento de circuitos utilizando-se componentes reconfiguráveis.

**Estaticamente Reconfigurável:** O sistema é reconfigurado várias vezes, para várias funções diferentes, mas as reconfigurações ocorrem sempre ao fim completo de um processamento. Estes sistemas tiram proveito das características reconfiguráveis do componente para execução de diferentes tarefas em um mesmo HW.

**Dinamicamente Reconfigurável:** Estes sistemas aproveitam completamente as características dos componentes reconfiguráveis, alterando ou reconfigurando as funções em tempo de execução. Este tipo de reconfiguração acarreta um *overhead* de reconfiguração que deve ser muito bem analisado para não comprometer o desempenho final do sistema.

**Parcial e Dinamicamente Reconfigurável:** Neste tipo de sistema, apenas uma parte do componente reconfigurável é alterada em tempo de execução. Nem todos os dispositivos reconfiguráveis suportam tal tipo de reconfiguração. Este tipo de sistema apresenta, em geral, um desenvolvimento de aplicação muito complexo e necessita de um profundo conhecimento tanto do dispositivo quanto da aplicação. A reconfiguração parcial é muito utilizada para correção de falhas, sistemas evolutivos e para minimizar o *overhead* de reconfiguração dos sistemas dinamicamente reconfiguráveis.

A opção por um destes quatro tipos de modos de reconfiguração deve ser definida de acordo com as características e necessidades de desempenho, custo e disponibilidade que o sistema necessita.

## 2.5 Considerações Finais

As características e classificações apresentadas neste capítulo têm o objetivo de apresentar as arquiteturas reconfiguráveis e mostrar o quão válida é a sua utilização. A constante utilização destes componentes e os problemas por estes solucionados, motivaram o estudo e utilização

de FPGA's em implementação de computadores. Como pode ser notado, os sistemas reconfiguráveis são complexos e seu projeto é trabalhoso e custoso, pequenos detalhes podem alterar completamente as características e, até mesmo, comprometer o funcionamento de um sistema. Porém, esses sistemas são mais flexíveis para a criação de aplicações espaciais, utilizando um mesmo chip e apresentam uma característica de desenvolvimento mais simples, rápida e de custo menor em relação aos ASICs.

Das características apresentadas, destacam-se: prototipagem rápida, computação espacial, flexibilidade, baixo custo e a possibilidade de reconfiguração parcial/dinâmica.

### ***3 Abordagens para o Desenvolvimento de Aplicações Usando Arquiteturas Reconfiguráveis***

Apesar de os sistemas reconfiguráveis terem desempenho significativamente melhor em relação a sistemas convencionais e trazerem muitos benefícios, geralmente, são ignorados pelos programadores e desenvolvedores devido à complexidade de desenvolvimento de aplicações. A menos que haja uma maneira fácil de desenvolvimento dos sistemas reconfiguráveis, essa situação continua requerendo ambientes de desenvolvimento que auxiliem na criação de configurações e na inserção e interação com os sistemas convencionais. Estes ambientes podem variar de simples assistentes para projetos manuais de circuitos reconfiguráveis, a sistemas automáticos completos desses projetos.

#### **3.1 Uma visão geral**

O projeto manual de circuitos reconfiguráveis, além de moroso, requer um grande conhecimento do sistema reconfigurável para o qual será destinado, além de moroso. Por outro lado, o sistema automático é um meio rápido e fácil para criar configurações para sistemas reconfiguráveis torna tal tipo de desenvolvimento mais acessível, embora nem sempre gera a melhor solução para o problema.

Os desenvolvimentos precisam passar por diversas fases como ilustra a Figura 11, onde são apresentadas três formas de desenvolvimento de projetos de reconfiguração: (a) automático, (b) parcialmente automático e (c) manual. As etapas em cinza são realizadas manualmente pelo desenvolvedor, enquanto as etapas em branco são realizadas automaticamente. As linhas pontilhadas representam trajetos para melhorar o circuito resultante. No sistema automático o programa C é particionado em *hardware* e *software* e então, são gerados e compilados para *NetList* (lista de componentes no nível de portas lógicas), que é mapeado para o *hardware*; então, realiza-se o *Place & Route* (posicionamento e interligação dos componentes reconfig-



uráveis). No sistema parcialmente automático, o usuário realiza o particionamento em *hardware* e *software*, descreve estruturalmente o *hardware*, então realiza-se o mapeamento e o *Place & Route*. No sistema manual é realizada a descrição do *hardware* em nível de portas lógicas, que são mapeadas para o *hardware* e realiza-se então o *Place & Route*.

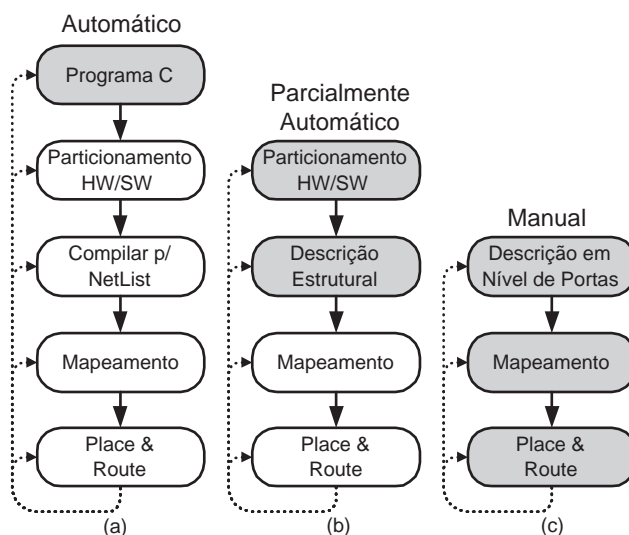


Figura 11: Três fluxos possíveis para a implementação de um circuito em um sistema reconfigurável (COMPTON; HAUCK, 2002)

A descrição do circuito é o processo que descreve as funções a serem colocadas no *hardware* reconfigurável. Isto pode ser feito de maneira simples, escrevendo um programa em linguagem de alto nível como C, que representa a funcionalidade do algoritmo a ser executado. Por outro lado, pode ser complexo, realizando a descrição das entradas, saídas e operações de cada bloco básico do sistema reconfigurável. Entre os dois extremos pode-se descrever o circuito utilizando-se componentes genéricos complexos, como somadores ou multiplicadores que serão especificados no *hardware* reconfigurável em outra etapa do processo.

Para descrições em linguagem de alto nível, como C/C++ ou Java, ou outra que utiliza blocos lógicos complexos, o código deve ser compilado em uma rede (*netlist*) de componentes em nível de portas lógicas. A implementação em linguagens de alto nível envolve a geração dos componentes para realizar as operações aritméticas e lógicas, e separadamente as estruturas de controle do programa, como laços e operações de salto. Cada descrição estrutural, gerada por uma linguagem de alto nível ou especificada manualmente pelo usuário, ou seja, cada estrutura complexa, deve ser substituída por uma rede de portas lógicas equivalentes.

Uma vez detalhado no nível de portas ou elementos lógicos, o circuito está criado. Estas estruturas devem, agora, ser convertidas aos elementos reais da lógica do *hardware* reconfigurável para o qual são desenvolvidas. Este estágio é chamado de Mapeamento, e é dependente

da arquitetura do *hardware* utilizado. Para arquiteturas baseadas em LUTs, esta etapa divide o circuito em pequenas subfunções, que possam ser mapeadas em uma única LUT (BROWN et al., 1992a) (ABOUZEID et al., 1993) (CHANG et al., 1996) (HAUCK; AGARWAL, 1996) (LIN et al., 1997) (CONG; WU, 1998) (TOGAWA et al., 1998) (CONG et al., 1999).

Depois de descrito o circuito, os blocos resultantes devem ser configurados no *hardware* reconfigurável. Cada um destes blocos deve ser colocado em uma posição específica dentro do *hardware*, sendo que esta posição deve ser a mais próxima possível aos blocos com os quais se comunica. Quanto maior for a capacidade das FPGAs, maior é o tempo gasto nesta fase de disposição (*Place & Route*). *Foorplanning* é uma técnica que pode ser usada para diminuir esse tempo. O algoritmo de *foorplanning* primeiro divide as células lógicas em conjuntos, onde as células com grande quantidade de comunicação são agrupadas. Estes conjuntos são então colocados como unidades em regiões do *hardware* reconfigurável. Uma vez realizada a disposição global, o algoritmo de posicionamento realiza o posicionamento detalhado dos blocos lógicos individuais dentro dos limites atribuídos ao conjunto (SANKAR; ROSE, 1999).

Após o *foorplanning*, os blocos lógicos individuais são colocados em células lógicas específicas. Um algoritmo que é geralmente utilizado é a técnica de *simulated annealing* (KRUPNOVA et al., 1997) (SENOUCI et al., 1998). Este método realiza uma disposição inicial (pseudo-) aleatória, e então executa uma série de "movimentos" nesta disposição. Um movimento é simplesmente a mudança de localização de uma célula lógica ou a troca de posição entre duas células. Estes movimentos são realizados um a um utilizando alvos aleatórios. Caso o movimento melhore a disposição, então a mesma é alterada. Movimentos indesejados são aceitos em uma pequena porcentagem das vezes, o que ajuda a evitar desperdício de pequenos locais dentro da disposição. Existem outros algoritmos que não se baseiam em movimentos aleatórios (GEHRING, 1996), que buscam a minimização de área utilizada, o que nem sempre gera uma solução que supre as exigências de desempenho do projeto.

Finalmente, os diferentes componentes reconfiguráveis são conectados no estágio de roteamento. Cada sinal é atribuído ao recurso de roteamento específico do *hardware* reconfigurável. Esta etapa pode tornar-se complicada caso não tenha sido realizada uma boa disposição dos blocos, pois sinais que percorrem distâncias maiores utilizam mais recursos. Um dos desafios de roteamento em FPGAs e sistemas reconfiguráveis é que os recursos de distribuição são limitados. Em geral o objetivo é minimizar o número de ligações (trilhas) utilizadas em um canal de comunicação entre unidades, mas estes canais devem ter a largura necessária para a computação. Entretanto, em sistemas reconfiguráveis, a quantidade de ligações é determinada no momento da fabricação. Assim, o roteamento de um FPGA concentra-se em minimizar o con-

gestionamento dentro das ligações disponíveis (BROWN et al., 1992b) (ALEXANDER; ROBINS, 1996) (CHAN; SCHLAG, 1997) (THAKUR et al., 1997) (NAM et al., 1999). Como o roteamento é uma etapa muito demorada, costuma-se verificar a viabilidade do mesmo antes da sua realização. Isto pode informar ao desenvolvedor se existe a necessidade de se alterar o projeto dos componentes reconfiguráveis (WOOD; RUTENBAR, 1997) (SWARTZ et al., 1998).

Cada uma das fases do projeto mencionadas anteriormente pode ser executada manualmente ou automaticamente usando ferramentas, como compiladores. Os compiladores de *hardware* para descrições de alto nível são cada vez mais utilizados para melhorar a produtividade no desenvolvimento de circuitos avançados em geral e projetos reconfiguráveis específicos.

As próximas seções apresentam métodos de projeto de alto nível sob duas perspectivas, de acordo com (TODMAN et al., 2005): projetos de propósitos específicos e projetos de propósitos gerais.

## 3.2 Projetos de Propósito Geral

Esta seção apresenta métodos e ferramentas de desenvolvimento de circuitos baseados em linguagem de propósito geral, como C, adaptadas para facilitar o desenvolvimento de *hardware*. Podemos incluir neste grupo as linguagens de descrição de *hardware* VHDL (Apêndice A) e Verilog, que são amplamente usadas em plataformas comerciais.

Um número considerável de compiladores de C para *hardware* tem sido desenvolvido. Estes variam de compiladores que apenas visam gerar o *hardware* a outros que visam o desenvolvimento completo de sistemas *hardware/software*; alguns, ainda, realizam o particionamento entre *hardware* e *software*.

Este grupo pode ser subdividido em dois tipos de abordagens: de anotações e dirigido por restrições (*annotation and constraint-driven*) e compilação direta do código fonte. A primeira abordagem preserva o programa fonte em C ou C++ tanto quanto possível e faz uso de anotações e arquivos de restrições para guiar o processo de compilação. A segunda abordagem modifica a linguagem fonte para permitir ao desenvolvedor especificar características inerentes a sistemas reconfiguráveis, como por exemplo, a quantidade de paralelismo ou tamanho das variáveis a serem utilizadas.

### 3.2.1 Abordagem de Anotações e Abordagem Dirigida por Restrições

Os sistemas mencionados a seguir empregam anotações no fonte-código e arquivos de restrições para controlar o processo de otimização. Geralmente, pequenas mudanças são suficientes para produzir um programa compilável, sem a necessidade de grandes reestruturações. Cinco métodos representativos desta abordagem são SPC (WEINHARDT; LUK, 2001), Streams-C (GOKHALE et al., 2000), Sea Cucumber (JACKSON et al., 2003), SPARK (GUPTA et al., 2003) e Catapult-C (MCCLLOUD, 2004).

#### 3.2.1.1 SPC

Este método utiliza uma abordagem para geração automática de circuitos em *pipeline* otimizados, a partir de programas em alto nível, utilizando técnicas derivadas de compiladores que realizam vetorização de software. Isso envolve essencialmente a síntese de processadores *pipeline* que executam laços internos dos programas. Para tanto, é realizada uma análise de dependência similar à vetorização de *software*, que determina se pode ser gerado um *pipeline* para cada laço. Conseqüentemente, gera circuitos que exploram o paralelismo melhor do que outras ferramentas automáticas de desenvolvimento de alto nível. Baseada no *framework* SUIF (WILSON et al., 1994), esta abordagem usa transformações de laço e tira vantagem da reconfiguração em tempo de execução (*Runtime Reconfiguration - RTR*) e otimizações de acesso à memória.

Essa abordagem inclui ainda a síntese de circuitos não *pipeline* para laços, condições e sequências não-vetorizáveis. Podem ser utilizados dois modos: modo de *hardware* e modo de *codesign*. No modo de *hardware*, um processador é gerado para o programa inteiro, convertendo descrições de um programa sequencial em alto nível em linguagem de descrição de hardware (HDL). Na opção de *codesign*, partes do programa (tais como as partes não-sintetizáveis e as altamente irregulares) permanecem em software para serem executados no microprocessador principal. Este modo resulta em sistema de *hardware* e *software* em conjunto, com a geração automática de transferência dos dados e controle entre o microprocessador principal e o *hardware* reconfigurável.

#### 3.2.1.2 Streams-C

O método Streams-C (GOKHALE et al., 2000) compila programas C para VHDL sintetizável para um ou mais FPGAs assim como o controle *multi-thread* para o processador principal. Os alvos são aplicações baseadas em *streams*, cujas características podem ser ressaltadas: altas taxas de fluxo de uma ou mais fontes de dados; computação intensiva; acesso à memória local,

guardando coeficientes e outras constantes; sincronização ocasional entre fases do processamento.

A linguagem Streams-C é composta por um pequeno conjunto de anotações e bibliotecas utilizadas dentro de um programa C convencional. As anotações são utilizadas para declarar processos, *stream*, e sinais, e para indicar os recursos do FPGA. As bibliotecas são utilizadas para comunicação de dados entre os processos. Um processo pode ser executado no processador principal ou no FPGA. Para um processo no FPGA, o corpo do processo acessa apenas a memória local e deve ser escrito utilizando um subconjunto da linguagem C suportado pelo compilador Streams-C. Além disso, possui funções intrínsecas para manipular *stream* e sinais. Todas as declarações são inicializadas no início do programa e são mantidas até a subrotina termine.

O Streams-C utiliza o modelo de programação paralela de Comunicação entre Processadores Sequenciais (*Communicating Sequential Processes*) (CSP) (HOARE, 1978).

### 3.2.1.3 Sea Cucumber

O método Sea Cucumber (SC) (JACKSON et al., 2003) gera programas para FPGAs a partir de programas em Java, utilizando um esquema similar ao Handel-C (CELOXICA, 2004) que será apresentado detalhadamente mais adiante. O SC aumenta o desempenho do sistema permitindo que o usuário particione o algoritmo comportamental em segmentos paralelos. Usando o suporte à concorrência, nativo do Java, estes segmentos paralelos são expressados como *threads*. SC gera um circuito final para cada *thread* em separado, gerando processos em hardware paralelo.

### 3.2.1.4 SPARK

SPARK (GUPTA et al., 2003) é um *framework* de síntese de alto-nível que visa processamento de imagens e multimídia. Transforma uma descrição comportamental em ANSI-C em VHDL sintetizável. Compila o código em C utilizando as seguintes etapas: (a) cria uma lista especulativa para movimentação de código e transformação de laços; (b) realiza o controle de recursos com minimização de interconexões; (c) gera a máquina de estados finitos para o código fornecido; (d) realiza a geração de código produzindo código VHDL RTL sintetizável. Deve-se então utilizar uma ferramenta de síntese para sintetizar o código gerado.

O código C de entrada não possui nenhuma construção ou estrutura especial, entretanto SPARK não suporta alguns recursos da linguagem. São eles: ponteiros; recursão; saltos irregu-

lares (*goTo*); *break* e *continue*; arranjos multidimensionais; *structs*; e instruções do tipo ( $a ? b : c$ ). SPARK utiliza técnicas de compiladores paralelos para melhorar o paralelismo em nível de instruções, incorporando idéias de operações mutuamente exclusivas, compartilhamento e custos de recursos de hardware. Entre estas técnicas pode-se ressaltar: desenrolamento de laços; fusão de laços; movimento de código; eliminação de sub-expressões comuns, eliminação de código morto; funções *in-lining*; e propagação de constantes.

### 3.2.1.5 Catapult-C

Catapult-C (MCCLLOUD, 2004) sintetiza uma descrição em C++ em nível de transferência de registradores (RTL), utilizando características da tecnologia. Usuários podem ajustar restrições para explorar melhor a utilização de espaço, controlar *pipeline* de laços e o compartilhamento de recursos.

## 3.2.2 Compilação Direta do Código Fonte

Uma abordagem diferente para adaptar a linguagem para a descrição explícita de paralelismo, comunicação e outras adaptações de recursos de *hardware*, como tamanho de variáveis. Como exemplos desse método de desenvolvimento pode-se apresentar ASC (MENCER et al., 2003), Handel-C (CELOXICA, 2004), Haydn-C (COUTINHO; LUK, 2003) e Bach-C (YAMADA et al., 1999).

### 3.2.2.1 ASC

ASC (MENCER et al., 2003) é essencialmente uma biblioteca de C++ que pode ser compilada por um compilador C++ comum. O código ASC é, basicamente, um código C++ utilizando a biblioteca ASC para compilação. Quando compilado, o código ASC gera um código executável que age como um simulador no nível de bits (RTL) ou produz um circuito na forma de *netlist* de hardware.

Para os conceitos e descrições de tempo de *hardware* utiliza tipos e operadores implementados como classes de C++. Isto permite ao usuário programar no nível desejado para cada parte da aplicação. A exploração Semi-automática de espaço permite um aumento adicional na produtividade do desenvolvimento, ao suportar processos de otimização em todos os níveis de abstração. O desenvolvimento orientado a objetos permite um eficiente reuso de código e inclui uma biblioteca integrada de geração de unidade aritmética (MENCER, 2002). Uma biblioteca

*floating-point* (LIANG et al., 2003) fornece mais de 200 unidades diferentes de ponto flutuante, cada uma com a largura de dados definida de acordo com a necessidade.

### 3.2.2.2 Handel-C

Handel-C (CELOXICA, 2004) é baseada em um subconjunto de ANSI C com extensão para o suporte a computações paralelas e especifica recursos de *hardware* como largura de variáveis, manipulação de bits e canais de comunicação. Uma característica é que o sincronismo do circuito compilado é fixo em um ciclo por instrução de C. Isto permite que os programadores de Handel-C programem recursos de *hardware* manualmente.

Outra importante característica do Handel-C é que todas as construções básicas (exceto E/S) são sintetizadas em hardware. Isso dá confiança ao desenvolvedor de que o desenvolvimento em Handel-C terá realmente o comportamento especificado.

Handel-C compila a uma máquina de estado *one-hot* usando um esquema de passagem de *token* desenvolvido por Page(PAGE; LUK, 1991). Numa máquina de estado *one-hot* cada atribuição do programa é mapeada em exatamente um flip-flop na máquina do estado. Estes flip-flops de controle capturam o fluxo do controle (representado pelo *token*) no programa: se o flip-flop de controle, que corresponde a uma atribuição particular, for ativo, o controle é então passado a essa atribuição e os circuitos compilados para essa atribuição são ativados. Quando a atribuição termina a execução, passa o *token* à atribuição seguinte no programa.

### 3.2.2.3 Haydn-C

Haydn-C (COUTINHO; LUK, 2003) inclui características de Handel-C, VHDL e linguagens orientadas a objetos visando projetos baseados em componentes. Possui semântica similar a ANSI-C. Além disso, oferece suporte à explicitação de descrição de blocos paralelos, largura de bits variável, estruturas de FIFO *pipeline* e outras estruturas que permitem explorar características de *hardware*.

A inovação principal de Haydn-C é uma estrutura (*framework*) de anotações opcionais para a descrição de restrições e de transformações tais como programação e alocação de recurso. Existem transformações automatizadas de modo que um único projeto de alto nível possa ser usado para muitas execuções diferentes.

### 3.2.2.4 Bach-C

Bach-C (YAMADA et al., 1999) é similar a Handel-C, baseado em ANSI-C com extensões que suportam especificação de paralelismo, comunicação e largura de dados. A semântica para paralelismo e comunicação é baseada em CSP e OCCAM (INMOS, 1988).

Circuitos desenvolvidos em Bach-C consistem de uma sequência hierárquica de *threads*, todas em paralelo e comunicação via canais sincronizados e variáveis compartilhadas.

A Tabela 1 apresenta os compiladores discutidos nesta seção, mostra a linguagem fonte e alvo e algumas possíveis aplicações de exemplo.

Tabela 1: Compiladores de propósito geral

Compilador	Linguagem Fonte	Linguagem Alvo	Aplicação Exemplo
Streams-C	C + Biblioteca	RTL VHDL	Alteração de Contraste em Imagens
Sea Cucumber	Java + Biblioteca	EDIF	nenhuma apresentada
SPARK	ANSI C	RTL VHDL	Preditor MPEG-1
SPC	ANSI-C	EDIF	Reconhecimento de padrão de string
ASC	C++ usando classe de biblioteca	EDIF	Encriptação
Handel-C	Extensão de C	VHDL Estrutural, Verilog, EDIF	Processamento de Imagens
Handy-C	Extensão de C	Handel-C	Filtro FIR
Bach-C	Extensão de C	VHDL Comportamental e RTL	Processamento de Imagens

## 3.3 Projetos de Propósito Específicos

Com a grande variedade de problemas em que a computação reconfigurável pode ser aplicada, existem muitos domínios de problemas que merecem atenção especial, tais como: 1) processamento de sinais utilizando MATLAB, e 2) otimização de largura de dados.

Nesta seção será apresentada uma visão geral dos métodos para processamento digital de sinais e as ferramentas que visam a implementação reconfigurável. Será descrito então o problema de otimização de largura de dados, com as soluções que prometem trazer bons benefícios. Finalmente serão apresentados outros métodos de desenvolvimento de domínios específicos.

### 3.3.1 Processamento Digital de Sinais

Processamento digital de sinais (DSP) é uma das aplicações para a qual a computação reconfigurável tem sido mais aplicada e com mais sucesso. Isso pode ser percebido com a inclusão de blocos de *hardware* específicos para suporte a DSP pelos fabricantes de FPGAs



em seus *chips*. Tanto a Xilinx quanto a Altera (principais fabricantes de FPGA do mercado) já possuem uma ampla gama de *chips* com esses blocos, como as famílias Virtex-II, Virtex-II Pro, Virtex-4 e Virtex-5 pela Xilinx e Cyclone II, Stratix e Stratix II pela Altera.

Os problemas de DSP geralmente possuem as mesmas características: o tempo de desenvolvimento é geralmente pequeno e o tempo de processamento é alto; os algoritmos tendem a ser de cálculos numéricos intensos, com estruturas de controle muito simples; é aceitável um erro numérico controlado e métricas padrões são usadas para medir a qualidade da precisão.

Uma das principais ferramentas para desenvolvimento de aplicações DSP é o ambiente gráfico de programação Mathworks MATLAB Simulink (SIMULINK, ). Simulink é uma plataforma multidomínios de simulação e baseada em modelos para sistemas dinâmicos. Ela provê uma interface gráfica e um conjunto de bibliotecas customizáveis que permite o desenvolvimento, simulação, implementação e teste de sistemas. O sistema é integrado com o MATLAB, provendo acesso a uma ampla gama de ferramentas para o desenvolvimento de aplicações, visualização e análise de dados e computações numéricas. A Xilinx possui o *System Generator for DSP* (HWANG et al., 2001) que é o componente principal do Xilinx XtremeDSP, solução que combina o estado de arte de FPGAs, ferramentas de desenvolvimento, *cores* de propriedade intelectual, além de serviços de desenvolvimento e educacionais. A Altera possui o *DSP Builder* (ALTERA, 2006b) que é um sistema de desenvolvimento que faz a interligação entre o Quartus® II, ambiente de desenvolvimento da Altera, e o MathWorks MATLAB/Simulink.

### 3.3.2 Otimização da largura de dados

Nos microprocessadores convencionais a largura em bits dos dados é definida *a priori* pela arquitetura do processador e não pode ser alterada. Na computação reconfigurável o tamanho de cada variável pode ser definido com o objetivo de obter o melhor resultado de tamanho do projeto, velocidade, consumo de energia e qualidade numérica do resultado. O uso de tamanho de dados customizado é um dos pontos fortes e principais da computação reconfigurável.

Devido a essa flexibilidade é desejável automatizar o processo de encontrar a melhor representação possível para o tamanho das variáveis. Tem sido discutido, frequentemente, que a implementação em *hardware* mais eficiente de um algoritmo é aquela que utiliza uma grande variedade de precisões para representar os diferentes tamanhos das diferentes variáveis internas (CONSTANTINIDES et al., 2001).

Em (CONSTANTINIDES; WOEINGER, 2002) foi demonstrado que o problema de otimização de tamanho de variável é NP-hard (*Non-deterministic Polynomial-time hard* - uma classe

de complexidade de problemas que podem ser resolvidos por uma máquina de Turing não determinística em um tempo polinomial), até mesmo para sistemas com propriedades matemáticas especiais que simplificam o problema por uma perspectiva prática (CONSTANTINIDES et al., 2003). Existem, entretanto, muitas abordagens publicadas para resolver o problema da otimização do tamanho de dados. Estes podem ser classificados como heurísticas que oferecem uma melhor razão área/sinal (CONSTANTINIDES et al., 2001) (KUM; SUNG, 2001) (WADEKAR; PARKER, 1998), abordagens que fazem suposições nas simplificações nas propriedades de erro (KUM; SUNG, 2001) (CANTIN et al., 2001), ou abordagens que podem ser aplicadas a algoritmos com propriedades matemáticas particulares (CONSTANTINIDES et al., 2002)

Alguns trabalhos publicados sobre o problema de otimização de tamanho de dados usam uma abordagem analítica que escalona a estimação de erro (WADEKAR; PARKER, 1998) (NAYAK et al., 2001) (STEPHENSON et al., 2000), alguns usam a simulação (KUM; SUNG, 2001) (CANTIN et al., 2001), e alguns usam um híbrido dos dois (CMAR et al., 1999). A vantagem das técnicas analíticas é que não requerem uma simulação da representação dos estímulos e podem ser mais rápidas; entretanto, elas tender a ser mais pessimísticas.

### 3.3.3 Outros métodos de desenvolvimento

Além de processamento de sinais, processamento de vídeo e imagem são outras áreas que se beneficiam muito com métodos específicos de desenvolvimento. Três exemplos seguintes mostrarão isso. Além disso, trabalhos recentes indicam que uma outra área que tem se beneficiado muito é a de redes de computadores. O exemplo de implementação de *firewall* em *hardware* ilustrará essa afirmação.

#### 3.3.3.1 CHAMPION

CHAMPION (ONG et al., 2001) é um ambiente de desenvolvimento completo que possui as ferramentas necessárias para capturar, simular e implementar aplicações para plataformas reconfiguráveis múltiplas. Ele mapeia aplicações desenvolvidas no ambiente gráfico de desenvolvimento Cantata, onde todas as aplicações são desenvolvidas usando funções ou módulos pré-desenvolvidos chamados *glyphs*. O CHAMPION então realiza a sincronização dos dados utilizando para isso *buffers* de atraso. Após a sincronização dos dados é realizado o particionamento mantendo o fluxo funcional do sistema, dividindo o sistemas em vários *netlists* que devem ser sintetizados e configurados nos componentes de *hardware* correspondentes.

### 3.3.3.2 IGOL Framework

IGOL (*Imaging and Graphics Operator Libraries*) (THOMAS; LUK, 2002) é um sistema implementado em camadas que permite um fácil desenvolvimento, teste e execução de hardware acelerador. O *framework* também suporta o desenvolvimento utilizando a linguagem Handel-C. O IGOL é baseado no Microsoft *Component Object Model* (COM) para o desenvolvimento de componentes. Isso permite que sejam desenvolvidos *plug-ins* para serem incorporados a vários aplicativos do sistema operacional Windows, como Premiere, Winamp, VirtualDub e DirectShow.

### 3.3.3.3 SA-C

SA-C (*Single Assignment C*) (NAJJAR et al., 2003) é uma variante da linguagem C criada para automatizar a compilação de algoritmos escritos em linguagem de programação em sistemas reconfiguráveis baseados em FPGAs. Uma das principais características desta linguagem é que as variáveis só podem ser atribuídas uma vez, quando da sua declaração. O SA-C possui também arranjos multidimensionais, incluindo arranjos em que o tamanho não é conhecido em tempo de compilação. Destaca-se entre esses arranjos o elemento janela (*window*), que possui funções adaptadas para ele, como o *for*.

O compilador SA-C realiza um grande conjunto de otimizações que permitem a geração de um *hardware* mais eficiente, tais como: empacotamento de constantes, redução de largura de operadores, funções *in-line*, eliminação de código morto, movimentação de código invariante, eliminação de subexpressões comuns, desenrolamento de laços e fusão de laços.

### 3.3.3.4 Framework para Firewall

Lee (LEE et al., 2003) descreve um *framework* para *firewall* capturando dados de uma descrição de alto nível baseado na especificação da linguagem Ponder (DAMIANOU et al., 1995). O fluxo de desenvolvimento do *framework* é composto de três fases: fase de desenvolvimento, fase de compilação e fase de implementação em *hardware*.

Na fase de desenvolvimento é requerida uma descrição formal do *firewall* e informações adicionais que podem auxiliar na execução da otimização. Na fase de compilação, a descrição será convertida em representações de regras para *firewall* em hardware, que passam por uma série de otimizações. Na fase de implementação, as regras serão convertidas para *hardware* específico para o desenvolvimento.

## 3.4 Outros Métodos

Nesta seção serão apresentados brevemente outros métodos de desenvolvimentos relevantes.

### 3.4.1 Customização em tempo de execução

Uma característica muito relevante dos sistemas reconfiguráveis, que não é abordada nos trabalhos descritos anteriormente é a possibilidade de reconfiguração em tempo de execução. Esta reconfiguração pode ser parcial, onde uma parte do sistema continua operacional enquanto a outra é reconfigurada, ou total. Esta possibilidade abre uma ampla gama de aplicações tais como correção de erro, aplicações de evolução genética, atualização de sistemas em execução, entre outras. Para a reconfiguração parcial as FPGAs precisam de circuitos apropriados incluídos na fabricação. Então, na compilação, um *bitstream* de configuração inicial e outros subsequentes devem ser construídos.

O Compilador DISC *Dynamic Instruction Set Computer* foi desenvolvido para uma arquitetura de propósito geral desenvolvida com o propósito principal de estudo da reconfiguração parcial e dinâmica. Ele faz uso de bibliotecas de módulos lógicos pré-compilados que podem ser carregados no *hardware* reconfigurável por um mecanismo de chamada.

### 3.4.2 Processadores Soft e Hard

Os FPGAs modernos suportam a utilização de processadores implementados via configuração, conhecidos como *soft processors* (XILINX, 2005b). Os fabricantes possuem esses processadores como bibliotecas a serem utilizadas. A Altera possui o Nios e a Xilinx o Microblaze. Os *soft processors* são programados utilizando-se linguagem de alto nível, geralmente ANSI C, de maneira similar a um microprocessador convencional. Possuem ambientes para o desenvolvimento de aplicações utilizando esses tipos de processadores configurados nos FPGAs.

Existem ainda FPGAs com núcleos de processadores implementados em hardware em tempo de fabricação, conhecidos como *hard processors*; algumas famílias de FPGAs, por exemplo chegam a ter 4 núcleos de processadores. Estes núcleos, igualmente aos *soft processors* são programados utilizando-se linguagem de alto-nível, como ANSI C, e permitem uma ampla gama de aplicações.

### 3.4.3 Hardware/software Codesign

*Hardware/software Codesign* pode ser visto como o desenvolvimento simultâneo de *hardware* e *software*. O principal objetivo do desenvolvimento simultâneo é reduzir o tempo e o custo de desenvolvimento de sistemas heterogêneos. Entretanto, o desenvolvedor necessita realizar um bom particionamento entre *hardware* e *software*, uma das etapas mais complexas deste processo. Muitos grupos de pesquisa têm estudado e desenvolvido sistemas para compilação de código C para ambos *hardware* e *software*. Entre tantos serão apresentados dois de grande relevância.

#### 3.4.3.1 Garp

A arquitetura Garp (CALLAHAN; WAWRZNEK, 1998) integra um núcleo RISC, processador MIPS, com *hardware* reconfigurável que é usado como coprocessador. O compilador utiliza a linguagem ANSI-C como entrada, sem a necessidade de nenhuma alteração ou anotação adicional ao código fonte. O sistema utiliza o compilador SUIF C para a fase inicial de compilação e otimizações básicas, sem qualquer modificação. O compilador quebra o código em blocos básicos, que são sequências de instruções sem saltos ou desvios internos ou externos. Utiliza também o conceito de *hyperblock*, que foi desenvolvido inicialmente para arquiteturas VLIW, o que otimiza o paralelismo de instruções através de vários caminhos comuns. Realiza, então, uma série de otimizações para obter o maior paralelismo em uma menor área de circuito possível. Para isso, mapeia os nós em um DFG (*Data Flow Graphic* ou Gráfico de Fluxo de Dados) a partir do qual, após otimizado, gera o arquivo de configuração.

#### 3.4.3.2 NAPA C

NAPA C (GOKHALE; STONE, 1998) permite ao programador escolher o mapeamento dos dados e computações entre um processador RISC ou FPGA. O Compilador NAPA C gera um programa C convencional que contém porções de código que são processadas no processador RISC e também o código C que controla os circuitos gerados para FPGA. Para o código ser executado no FPGA o compilador analisa os laços do código C na busca da maior quantidade possível de paralelismo e *pipeline*, gerando então um *netlist* para configuração do *hardware* reconfigurável.

### 3.5 Considerações Finais

Como foi apresentado neste capítulo, existe uma ampla gama de abordagens para o desenvolvimento de aplicações para *hardware* reconfigurável. Muitas delas são voltadas para um tipo específico de aplicação, outras de propósito geral; algumas geram as configurações para arquiteturas específicas, outras para serem programadas em qualquer FPGA. Mas nem todas, ou a minoria, oferece a interação com outros sistemas ou permitem a fácil inserção das aplicações dentro de um projeto mais abrangente. Em vista disto, este trabalho não se propõe a apresentar uma nova metodologia para o desenvolvimento da aplicação reconfigurável, mas sim um ambiente para programação e inserção de funções reconfiguráveis dentro de sistemas mais abrangentes e complexos.

## 4 *Sistemas (Hardware e Software) utilizados para o Desenvolvimento Proposto*

Neste capítulo serão apresentados os sistemas (*hardware e software*) utilizados para o desenvolvimento do trabalho. Os sistemas utilizados serão: (1) o ambiente de desenvolvimento TEV; (2) o *kernel* Virtuoso; (3) cluster de DSPs Atlas; e (3) o kit reconfigurável de desenvolvimento XUP Virtex-II Pro.

TEV (Teaching Environment for Virtuoso) (RIBEIRO et al., 1998a), (MORÓN et al., 2000), (EONIC, 2001b) é um ambiente para treinamento e desenvolvimento de sistemas paralelos de tempo-real usando o Kernel Virtuoso (Virtual Single Processor Programming System) (WIND, 2001) da EONIC SYSTEMS (EONIC, 2004). O TEV foi desenvolvido no Departamento de Computação da UFSCar (MORÓN et al., 1997) e sua primeira versão está disponível para download em <http://www.dc.ufscar.br/tev>.

Virtuoso é um sistema operacional que fornece suporte ao desenvolvimento de aplicações de tempo-real. As aplicações desenvolvidas usando o Virtuoso são divididas em tarefas, que interagem com outras tarefas através de serviços de comunicação e sincronização.

O cluster de DSPs Atlas, da Eonic Solutions GmbH (EONIC, 2001a), é um hardware onde o sistema operacional de tempo-real Virtuoso pode ser executado, para implementar e executar aplicações que necessitam de alta performance para processamento digital de sinais.

O Kit XUP (*Xilinx University Program*) Virtex-II Pro (XILINX, 2006) provê uma plataforma avançada para o desenvolvimento de aplicações reconfiguráveis, que consiste de um FPGA de alto desempenho Virtex-II Pro, com uma ampla quantidade de componentes, que podem ser usados para criação de sistemas complexos.

## 4.1 TEV

A Figura 12 apresenta um exemplo de visualização do ambiente de desenvolvimento TEV. A janela principal (maior) é o editor gráfico, com a barra de ferramentas à esquerda e com os três nós representando estruturas de dados. A janela menor é o editor de texto contendo o código de um dos nós representados no editor gráfico.

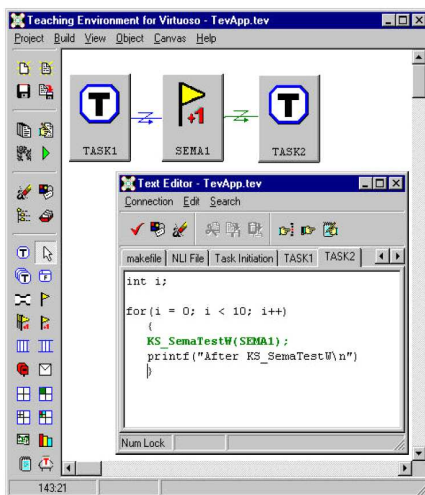


Figura 12: TEV - Ambiente visual para o desenvolvimento de programas paralelos de tempo-real

No TEV, uma aplicação paralela é representada através de gráficos, onde nós representam as estruturas de dados que compõem o programa paralelo (tarefas, semáforos, recursos e outras estruturas oferecidas pelo Kernel Virtuoso) e setas representam a comunicação e/ou sincronização de operação entre as estruturas. As informações do modelo gráfico podem ser complementadas com descrições textuais, que são segmentos de código, na linguagem C, adicionados pelo usuário. Baseado nestes componentes o código fonte da aplicação é gerado automaticamente.

Para tanto, o ambiente apresenta uma interface amigável, que ajuda o programador a dar continuidade no desenvolvimento de aplicações de tempo-real e paralelas que utilizam os métodos mais comuns (orientados a objetos ou não), tornando possível a visualização, depuração e validação dos requisitos temporais dos programas. Embora todo o ambiente visual TEV tenha sido projetado para o *Kernel Virtuoso* e para linguagem C, o mesmo pode ser adaptado para outros sistemas e linguagens de programação, como proposto neste trabalho.

Optou-se por utilizar este ambiente de desenvolvimento por ser um ambiente disponível e adequado. Nele, o desenvolvedor seleciona as tarefas a serem implementadas no *hardware* reconfigurável, que serão processadas de acordo com o método a ser apresentado no próximo capítulo.



O desenvolvimento de aplicações no TEV é descrito na Figura 13. Os retângulos denotam as ferramentas que compõem o ambiente visual, que são os seguintes: GPP (Gerador de Programas Paralelos), ATEPC (Analisador de Tempo de Execução e Pior Caso) e AE (Analisador de Escalonamento). Os arquivos gerados em cada estágio são representados por elipses, quais sejam: arquivos de projetos, arquivo *Makefile*, código fonte e arquivo executável. Os retângulos com cantos arredondados são usados para representar módulos de software com os quais o ambiente visual interage e corresponde ao compilador C e bibliotecas do Kernel Virtuoso.

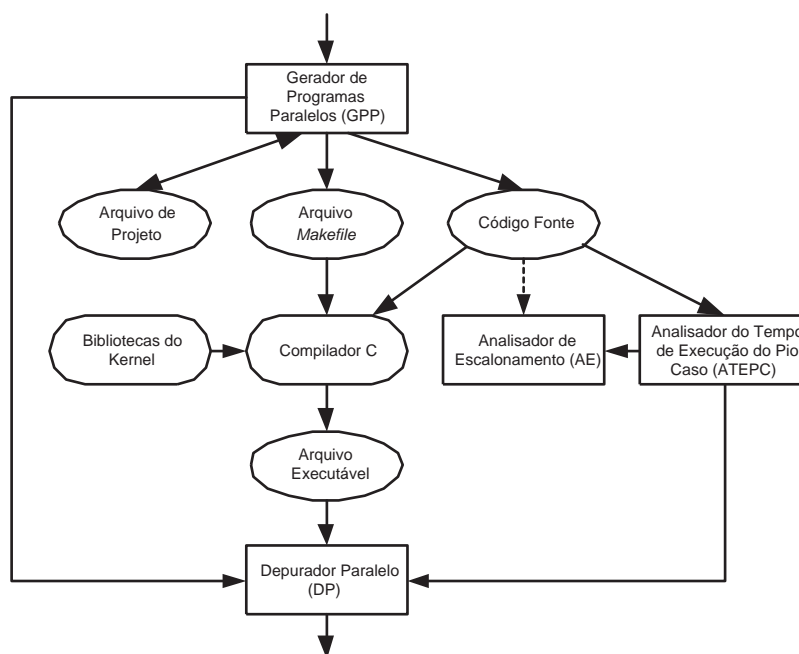


Figura 13: Componentes do desenvolvimento de aplicações no TEV (RIBEIRO et al., 1998a)

Gerador de Programas Paralelos (GPP) - O objetivo desta ferramenta é auxiliar na geração de código fonte dos programas executados na máquina paralela. No GPP, as aplicações são construídas através de um modelo gráfico, que é utilizado para integrar as demais ferramentas do ambiente visual. Este modelo é representado por um grafo, onde os nós denotam as estruturas de dados que compõem o programa paralelo (tarefas, semáforos, recursos *mailboxes* etc.) e as tarefas denotam operações de comunicação e sincronização relacionadas às estruturas. As informações do modelo gráfico podem ser complementadas com descrições textuais, ou seja, trechos de código escritos pelo usuário. As informações gráficas e textuais são armazenadas em arquivos de projeto. A partir destas informações, o GPP gera automaticamente o código fonte da aplicação e arquivos do tipo *makefile*, que são utilizados para compilar e *linkeditar* o código fonte produzido (RIBEIRO et al., 1998b).

Analisador de Tempo de Execução do Pior Caso (ATEPC) - Esta ferramenta faz o cálculo

automatizado do tempo de execução do pior caso (TEPC) das tarefas do kernel Virtuoso. O uso do TEPC é necessário para garantir o cumprimento dos requisitos temporais do sistema de tempo-real. O TEPC é calculado usando-se o caminho com o pior caso de tempo. O ATEPC também permite a visualização da estrutura das tarefas através de uma representação gráfica, onde são mostrados os principais comandos da linguagem C, primitivas do *kernel* e os TEPCs correspondentes.

Analisador de Escalonamento (AE) - esta ferramenta tem como finalidade prever se os requisitos temporais do sistema serão alcançados ou não. A partir das estimativas produzidas pelo ATEPC, o AE produz um diagrama das tarefas do sistema. Neste diagrama, é possível identificar facilmente as tarefas que não estão cumprindo os requisitos de tempo-real.

Depurador Paralelo (DP) - o objetivo desta ferramenta é permitir a depuração *on-line* dos programas desenvolvidos no ambiente visual.

As descrições seguintes correspondem aos recursos disponíveis no GPP, sendo: o editor gráfico usado para o desenvolvimento amigável e visual de estruturas de aplicações; e o editor de texto usado para detalhamento dos segmentos de códigos; e o gerador de código responsável pela geração dos arquivos contendo o código C da aplicação e o arquivo *Makefile*.

#### 4.1.1 GPP - Editor Gráfico

O editor gráfico oferece uma interface amigável para o desenvolvedor. Este editor é usado para criar a estrutura da aplicação e gerar parte do código fonte.

Usando o editor gráfico, o usuário cria diagramas que representam as características relevantes da aplicação. Neste diagrama, é utilizado um símbolo diferente para cada tipo de objeto (tarefas, semáforos, recursos, *mailboxes*, etc), e as chamadas de sistemas aparecem como ligações conectando esses objetos. Os objetos representados pelo editor gráfico podem ser divididos em duas categorias: configuráveis e executáveis. São exemplos de objetos configuráveis: tarefas, semáforos, recursos, *mailboxes*, etc; e de objetos executáveis: tarefas e funções.

Neste editor gráfico, os relacionamentos entre os objetos são representados através das conexões. Estas podem ser definidas como representações visuais das chamadas de sistema que os objetos executáveis (tarefas e funções) fazem. Uma conexão é representada no editor gráfico através de dois componentes: uma linha de conexão e um símbolo gráfico (Figura 14a). A linha de conexão é utilizada para conectar dois objetos que fazem chamadas aos serviços (tarefas ou funções) ao objeto sob o qual este serviço está sendo executado. O símbolo gráfico indica a chamada de sistema associada com esta conexão. Este símbolo é localizado, pelo

usuário, no segmento de linha que interconecta os objetos envolvidos. A Figura 14b ilustra a conexão entre duas tarefas utilizando o TEV, através de uso de duas conexões, TASK1 para SEMA1 e de SEMA1 para TASK2.

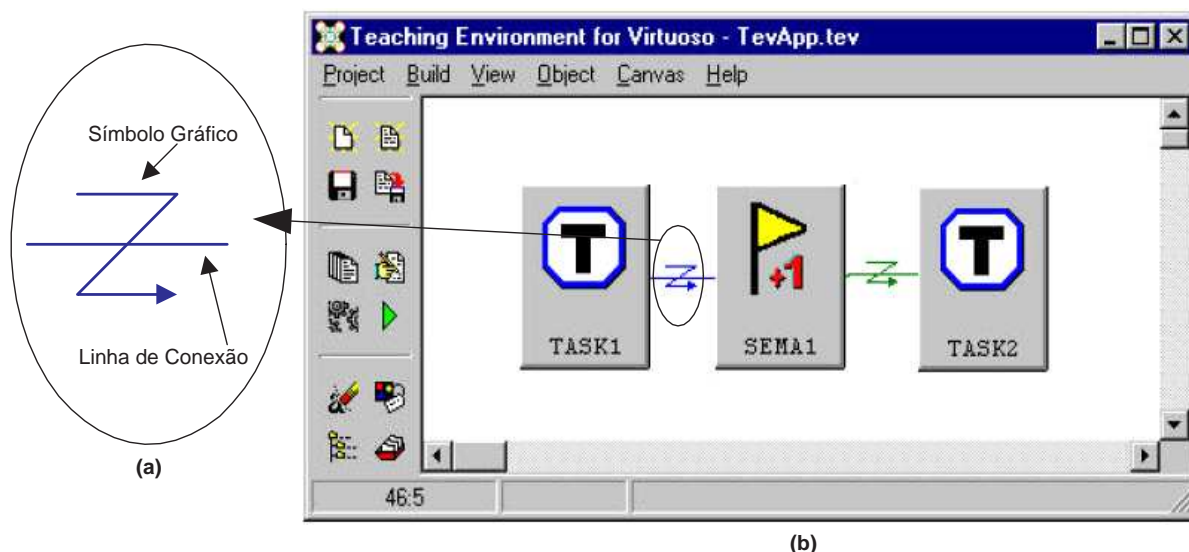


Figura 14: Conexão entre duas tarefas

#### 4.1.2 GPP - Editor de Texto

O editor gráfico permite uma representação, em alto nível, da estrutura de dados e chamadas de sistema. Entretanto, o código fonte dos objetos executáveis não contém somente chamadas de sistema. A funcionalidade destes objetos é determinada pelos segmentos de código escritos pelo usuário. Estes segmentos variam de acordo com a aplicação e é, portanto, melhor definido através de um texto do que por uma representação gráfica.

O editor de texto provê ao usuário uma facilidade de escrever, em linguagem C, o código dos objetos executáveis criados no editor gráfico. O editor de textos oferece, além das facilidades básicas encontradas de um editor de textos comum, outros recursos para integrar a parte textual à aplicação (código fonte dos objetos executáveis) com a parte gráfica (objetos executáveis e conexões).

Quando uma nova aplicação é inicializada, os esqueletos (estruturas) básicos dos arquivos especiais são automaticamente gerados pelo editor de textos. Estes *skeletons* podem ser atualizados mais tarde com informações adicionais.

### 4.1.3 GPP - Gerador de Código

O gerador de código é responsável por produzir a saída final, que são os arquivos necessários para a compilação e *linkagem* da aplicação. Dois tipos de arquivos são criados pelo gerador de código: arquivos contendo o código C da aplicação e um arquivo *makefile*.

O gerador de código produz os arquivos fonte, em linguagem C, para todos os objetos representados graficamente. Dois tipos de arquivos fontes são produzidos: Os primeiros contêm os códigos que implementam as estruturas de dados dos objetos. Os segundos são chamados de arquivos de cabeçalhos ou *header files* (arquivos .h), que podem ser inseridos no código fonte da aplicação.

Para cada objeto executável (tarefa ou função), um terceiro arquivo também é gerado. Este arquivo contém o código fonte do objeto executável.

Além disso, o gerador de código cria os arquivos *makefile*, que podem ser chamados pelo utilitário *make* para gerar os arquivos executáveis da aplicação. Os comandos *make* e *run* do menu principal podem ser utilizados para compilar, *linkeditar* e executar as aplicações, sem a necessidade de sair do ambiente visual para poder executar estes comandos.

## 4.2 Virtuoso

O Kernel Virtuoso é um sistema operacional que concentra somente os objetos e serviços necessários para o desenvolvimento de aplicações de tempo-real em sistemas multiprocessados. As aplicações desenvolvidas usando o Virtuoso são divididas em tarefas, que são módulos de programas independentes, que podem interagir com outras tarefas através de serviços de comunicação e sincronização. Em sistemas com apenas um processador, o Virtuoso usa os conceitos multitarefas simulando a execução simultânea dos processos, compartilhando o uso do processador entre os diversos processos. Em sistemas multiprocessadores, as tarefas são facilmente distribuídas entre os diferentes processadores, até os requisitos de tempo-real serem alcançados.

Cada objeto do microkernel - tarefas, semáforos, recursos e demais - tem atributos específicos e suportam um conjunto de serviços. Os objetos principais do microkernel são: tarefas, semáforos, caixas de mensagem, filas, canais, particionamento de memória, recursos e temporizadores.

### 4.2.1 Tarefa

Uma tarefa é um módulo independente, escrito em C, com sua própria *thread* de execução e um conjunto de recursos do sistema. Ela realiza um função bem definida ou um conjunto de funções e se comunica com outras tarefas usando outros objetos do sistema. O número de tarefas em um sistema é limitado apenas pela disponibilidade de memória nos nós.

Os serviços de tarefas permitem iniciar ou parar, suspender e retomar, abortar, alterar sua prioridade e ponto de entrada, e mudar seu grupo. As tarefas rodando em qualquer nó do sistema podem passar dados, ou sincronizar com qualquer outra tarefa do sistema utilizando os serviços do *kernel*. Para sincronização pode-se utilizar os eventos, semáforos ou recursos. Para troca de dados usa-se as filas e caixas de mensagem.

### 4.2.2 Semáforo

O Semáforo é um objeto do sistema que permite as tarefas sincronizarem suas atividades. A tarefa sinaliza o semáforo, que incrementa um contador. Outra tarefa testa o semáforo e se ele não estiver sinalizado aguarda até que seja sinalizado, com ou sem estouro de tempo, ou retorna.

Um semáforo é útil quando um número de sinais e de testes necessitam ser abastecidos, ou quando as tarefas que se comunicam estão localizadas em processadores diferentes. Caso contrário, deve-se utilizar um evento, que é uma forma mais simples e eficiente para sincronização local. Se diversas tarefas estiverem esperando em um semáforo simultaneamente estas são presas (bloqueadas) em uma lista por prioridade. Quando o semáforo é sinalizado a tarefa com a prioridade mais elevada será liberada.

### 4.2.3 Caixa de Mensagem

O serviço de mensagens permite que se coloque e consuma mensagens em uma fila e as leia. A caixa de mensagens pode ser usada de modo síncrono ou assíncrono. Quando é usado de modo síncrono, remetentes e destinatários são liberados apenas quando a mensagem é copiada. Quando usado assincronamente, o remetente é liberado imediatamente. As mensagens podem ser de qualquer tamanho, e podem ter prioridade. Se o remetente e o destinatário estão no mesmo nó, um ponteiro para os dados da mensagem é passado por um campo no cabeçalho da mensagem. Isto é muito mais rápido do que copiar toda a mensagem, mas menos portátil. Isto é útil quando um conjunto de tarefas utilizam os mesmos dados.

Uma mensagem de uma caixa de mensagens possui duas partes, um cabeçalho e um corpo. Somente o cabeçalho é enviado à caixa de mensagens. O cabeçalho contém informações do remetente e do destinatário da mensagem, sua localização e seu tamanho. O corpo contém os dados da mensagem.

#### 4.2.4 Fila

Fila (FIFO - *first in first out*) é um objeto do sistema apropriado para transferir quantidades pequenas de dados, tais como a informação de controle de tarefa, de maneira síncrona *bufferizada* ou ordenada no tempo. As entradas da fila devem ser de um tamanho fixo, com um tamanho máximo de dez palavras de 32-bits. Os serviços da fila permitem pôr mensagens na fila, lê-las, verificar o número de mensagens da fila, e apagá-las.

As filas no Virtuoso não possuem dono, de forma que qualquer tarefa pode ler e escrever sobre as mesmas. O acesso é normalmente protegido por um recurso. Se é necessário o conhecimento do remetente, deve-se utilizar uma fila separada para cada combinação remetente/destinatário, ou usar uma caixa de mensagens.

#### 4.2.5 Canais

Um canal consiste de uma fila de escrita e leitura e um *buffer* opcional. No caso de canal sem *buffer*, os dados de tamanhos variados fluem diretamente do escritor para o leitor. Quando se utiliza *buffer*, os dados são copiados para um *buffer*, antes de serem transferidos para o leitor. Os canais devem ser vistos como "dutos" de *software* que permitem a escrita simultaneamente à leitura. Além disso, os canais permitem a comunicação entre uma tarefa e um programa externo.

#### 4.2.6 Memória

O Virtuoso evita alguns problemas comuns de alocação de memória utilizando reservatórios (*pools*) de memória e mapas de memória. Eles podem ser usados simultaneamente na mesma aplicação. É desejável que novas aplicações sejam desenvolvidas utilizando reservatórios de memória em substituição a mapas de memória, por serem mais flexíveis.

Mapas de memória utilizam blocos de tamanho fixo e podem somente ser alocados e liberados localmente. Deve-se ter cuidado ao atribuir tarefas aos nós do projeto porque todos os mapas de memória são acessíveis pelas tarefas que as necessitam.

Os reservatórios da memória são mais versáteis do que mapas de memória. Os tamanhos

máximo e mínimo do bloco são especificados, e os blocos são feitos sob medida, dinamicamente, entre aqueles limites e alocado do reservatório de acordo com o tamanho pedido. Podem também ser compartilhados pelas tarefas que funcionam em nós diferentes. Cada *pool* é composto de diversos blocos.

#### 4.2.7 Recursos

O recurso é utilizado para controlar o acesso a dispositivos físicos que possam ser necessários a múltiplas tarefas. O objeto é bloqueado por uma tarefa, que passa a ter controle do recurso, o que previne que tarefas com prioridades maiores tomem controle antes que a tarefa termine a execução. As tarefas que aguardam por um determinado recurso são organizadas em uma fila por ordem de prioridade. Quando o recurso é liberado a tarefa com maior prioridade assume o recurso.

Quando um número limitado de objetos for protegido por uma tarefa, um semáforo pode ser usado como uma alternativa a um recurso. Por exemplo, pode-se inicializar um semáforo com uma contagem igual ao número de objetos disponíveis. Então, quando uma tarefa quer usar um dos objetos testa o semáforo e a contagem é decrescida. Quando uma tarefa termina com o objeto, sinaliza o semáforo para indicar que a entrada do objeto está disponível para ser usado por uma outra tarefa.

#### 4.2.8 Temporizador

Esta classe de chamadas permite às tarefas da aplicação utilizar temporizadores como parte de suas funções. O temporizador pode ser iniciado para gerar um evento programado em um momento especificado (disparado) ou no intervalo (cíclico). Este evento pode então sinalizar um semáforo associado. Os temporizadores são usados principalmente para regular a execução das tarefas com relação a um comportamento temporal requerido.

O Virtuoso suporta dois tipos de temporizadores: de alta e de baixa resolução. O temporizador de alta resolução corresponde ao *clock* de *hardware* do DSP. O de baixa resolução é um relógio de *software* implementado no *driver* do Virtuoso.

### 4.3 Cluster de DSPs Atlas

O sistema Atlas, da Eonic Solution GmbH (Figura 15), inclui *hardware* e *software* para implementar e executar aplicações que necessitam de alta performance. Este sistema é com-

posto por um processador principal Pentium com Windows NT, e duas placas ATLAS2-HS V1.1 (EONIC, 2001a) usadas para aplicações que requerem controles rápidos e processamento digital de sinais. Cada placa é composta por dois processadores ADSP-21160 72MHz (Hammerhead SHARC) da Analog Devices. Estes processadores DSPs são utilizados para comunicação, processamento gráfico de imagens, que combinam operadores de ponto flutuante com suporte a multiprocessamento. A Figura 16 ilustra a arquitetura do Cluster de DSPs Atlas, com o processador principal e as duas placas ATLAS.



Figura 15: Cluster de DSPs Atlas

Cada placa ATLAS2-HS, além dos dois processadores ADSP-21160, possui um controlador PCI com gerenciamento de energia, que é usado para sistemas embarcados e possui uma avançada arquitetura de escoamento de dados, que inclui dois núcleos DMA. Há duas SBSRAM (*Synchronous Burst Static RAM*) para cada DSP, o que provê 256K de 64 bits para cada DSP.

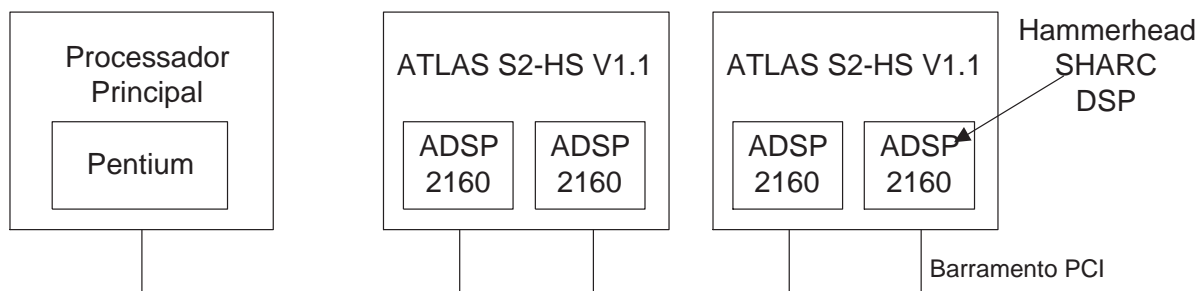


Figura 16: Sistema ATLAS - Cluster de DSPs e processador principal

#### 4.4 Kit XUP Virtex-II Pro

A placa Xilinx University Program (XUP) Virtex-II Pro (Figura 17), que provê uma avançada plataforma de hardware, é utilizada neste trabalho acoplada ao sistema paralelo Atlas, com Virtuoso e R-TEV. Esta plataforma inclui um FPGA de alto desempenho XC2VP30 com uma considerável quantidade de componentes que permitem criar sistemas complexos.



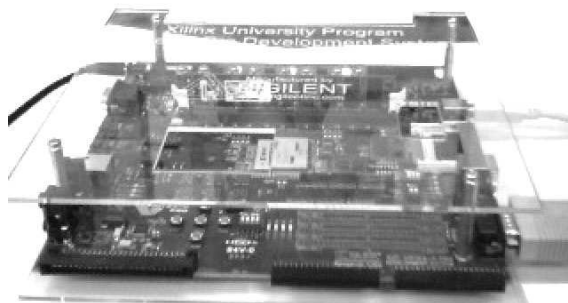


Figura 17: Kit XUP Virtex-II Pro

A Figura 18 apresenta um diagrama de blocos do Kit, mostrando seus principais componentes:

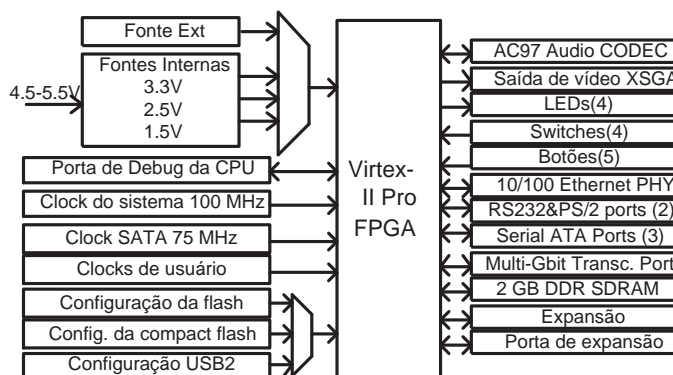


Figura 18: Diagrama de blocos do Kit XUP (XILINX, 2006)

- Virtex-II Pro FPGA com 2 núcleos PowerPCs 405;
- 512Mb de (DDR) SDRAM;
- Conector de CompactFlash para configuração da FPGA e armazenamento de dados;
- Plataforma USB para configuração da FPGA;
- Placa de rede 10/100 Ethernet PHY;
- Porta RS-232 DB9 serial;
- Duas portas seriais PS-2;
- Quatro LEDs conectados ao pinos de E/S da Virtex-II;
- Quatro *switches* conectados ao pinos de E/S da Virtex-II;
- Cinco botões conectados ao pinos de E/S da Virtex-II;

- AC-97 audio *CODEC* com amplificador de audio e saídas *speaker/headphone* e *line level*;
- Entradas de microfone e auxiliar em linha de audio;
- Saída XSGA, de até 1200 x 1600 a 70 Hz;
- Três portas Serial ATA;
- *Clock* do sistema de 100 MHz, *clock* do SATA 75 MHz;
- Fontes de energia na placa;
- Circuitos de *reset* do sistema e dos PowerPCs.

O componente principal do kit é o FPGA XC2VP30 (XILINX, 2005c) que contém dois processadores PowerPC 405D4, RAM de 2.448Kb, 136 blocos de multiplicadores e 3.492 CLBs (*slices*) de FPGA, gerenciador de clock digital (DCM) e transceptores flexíveis de alta taxa de comunicação (RocketIO) dispostos de acordo com Figura 19. A tecnologia permite o núcleo operar a 300MHz enquanto mantém baixo consumo de energia.

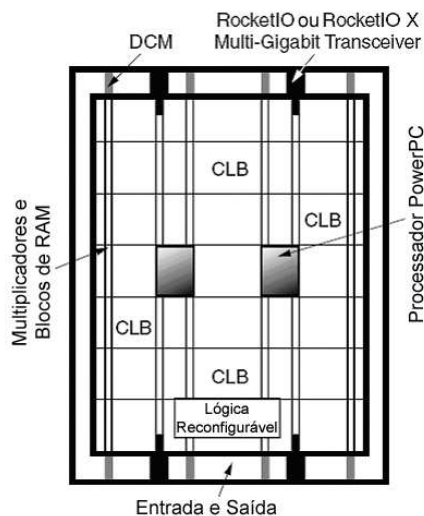


Figura 19: Diagrama do FPGA Virtex-II Pro - XC2VP30 (XILINX, 2005c)

O Kit XUP Virtex-II Pro provê vários métodos para configuração da FPGA. Os dados de configuração podem originar da plataforma Flash PROM, de um cartão de memória Compact-Flash, ou de uma configuração externa por meio da interface USB ou paralela.

O desenvolvimento de aplicações para este kit deve ser realizado utilizando-se o ambiente de desenvolvimento EDK (*Embedded Development Kit*) da Xilinx, que consiste de um conjunto de ferramentas dentre as quais destaca-se Xilinx Platform Studio (XPS), que provê um ambiente para desenvolvimento com plataformas FPGAs com *cores* PowerPCs embarcados e/ou

com *softcores* (biblioteca de software) MicroBlaze (XILINX, 2005b). Neste ambiente a programação dos *cores* processadores embarcados ou *softcore* é feita através de programas descritos utilizando a linguagem ANSI C e as bibliotecas específicas do fabricante. Deve-se destacar ainda dentro do ambiente EDK o *Base System Builder* (BSB) (XILINX, 2007), uma ferramenta de software que ajuda o usuário a construir um projeto de trabalho específico para a placa de desenvolvimento. O BSB oferece um amplo número de opções para especificação e configuração dos sistemas básicos da placa. O EDK possui ainda ferramentas que auxiliam a criação e importações de *IP cores* para o sistema e uma ferramenta para construção de programações para o *chip* a partir uma *netlist* de descrição de *hardware*, entre outras.

## 4.5 Considerações Finais

Neste capítulo foram apresentados os sistemas de *hardware* e *software* que foram utilizados no desenvolvimento deste trabalho. Estes sistemas foram integrados de forma a possibilitar que todos fossem utilizados em sua plenitude.

As características apresentadas neste capítulo fornecem uma idéia do potencial de cada um dos sistemas. O ambiente de desenvolvimento tem potencial tanto para as aplicações voltadas somente ao *Kernel Virtuoso*, quanto as aplicações voltadas à computação reconfigurável, como proposto neste trabalho. O *kernel Virtuoso* possui objetos que facilitam o desenvolvimento de aplicações paralelas de tempo real, muito interessantes para utilização em sistemas com aplicações reconfiguráveis. O cluster de DSPs Atlas possui grande poder de processamento que pode ser facilmente utilizado e aplicado em desenvolvimentos na área de processamento de imagens e sinais. O Kit de XUP possui uma FPGA com grande capacidade e funcionalidades e reúne a esta vários periféricos que auxiliam no desenvolvimento de aplicações e interação com outros sistemas.

## 5 *Trabalho Desenvolvido - Ambiente R-TEV*

A proposta principal deste trabalho é integrar a programação de *hardware* reconfigurável ao ambiente de desenvolvimento TEV. Desta maneira, faz-se uso das características do TEV que tornam o desenvolvimento de aplicações simplificado e ao alcance de qualquer programador, sem que este necessite conhecimentos profundos de programação paralela de tempo-real, ou reconfigurável. Para tanto, propõe-se a criação de uma biblioteca de funções reconfiguráveis que possa ser utilizada comunicando diretamente com as tarefas do *kernel* Virtuoso, sem alterar desta maneira o fluxo de desenvolvimento dentro do ambiente TEV. Este ambiente, TEV mais a biblioteca de funções reconfiguráveis, denominou-se R-TEV.

### 5.1 Visão Geral

Este trabalho constituiu-se das seguintes atividades:

- Criação de uma entidade reconfigurável no ambiente TEV, agora denominado R-TEV;
- Criação de uma biblioteca de funções reconfiguráveis, utilizada pela entidade criada. As funções devem ser definidas, desenvolvidas e compiladas anteriormente utilizando umas das metodologias de desenvolvimento de aplicações reconfiguráveis apresentadas no Capítulo 3;
- Definição de um formato de dados para aplicação e criação de um protocolo para a transmissão de dados ao componente ou *hardware* reconfigurável.
- Definição de um mecanismo automático para comunicação entre o módulo reconfigurável e os outros componentes do TEV;
- Inclusão de mecanismo de *upload* das funções no *hardware* reconfigurável.

O desenvolvimento de aplicações no ambiente RTEV deve seguir as seguintes etapas:

1) Após definida as funções a reconfiguráveis a serem utilizadas, deve-se verificar se tais funções encontram-se disponíveis na Biblioteca Reconfigurável . Em caso afirmativo continuar o fluxo de desenvolvimento convencional do TEV. Caso contrário é necessário desenvolvê-la ou solicitar a um especialista que o faça;

2) Caso seja necessário o desenvolvimento deve-se atentar a necessidade de inclusão do socket de comunicação no desenvolvimento da função reconfigurável.

3) Durante o desenvolvimento no RTEV a função reconfigurável deve ser adicionada utilizando o componente reconfigurável, selecionando a função desejada da Biblioteca Reconfigurável;

4) Este componente pode ser interligado aos outros componentes do RTEV utilizando os componentes disponíveis no sistema da mesma forma que se faz no TEV;

5) Ao final do desenvolvimento deve-se realizar a geração automática do código do código fonte;

6) Durante a geração, todo código do *socket* de comunicação com o HR (*hardware* reconfigurável) é adicionado de forma automática à aplicação. Além disso é adicionado, também de forma automática, o código necessário para o carregamento da configuração do HR;

7) A comunicação entre o *Kernel Virtuoso* e o HR é realizada utilizando um protocolo de comunicação definido neste trabalho. Tal protocolo será apresentado em uma próxima seção deste capítulo;

8) Após a compilação do código gerado, a sua execução é feita de forma usual.

As próximas seções deste capítulo apresentarão de forma detalhada os componentes e processos deste desenvolvimento.

## 5.2 R-TEV - Reconfigurable Teaching Environment for Virtuoso

Com o objetivo de alcançar as características desejadas do ambiente R-TEV, mantendo o mesmo fluxo de desenvolvimento do TEV, mas utilizando funções reconfiguráveis, foram necessárias várias atividades:

- Inclusão de um componente de funções reconfiguráveis dentro do ambiente TEV;

- Definição de um protocolo para troca das variáveis a serem processadas na placa, para comunicação entre o computador principal, ATLAS, e o kit reconfigurável;
- Desenvolvimento de um *socket* para comunicação entre a placa e o *kernel* Virtuoso, para troca dos dados;
- Adição, dentro do ambiente de desenvolvimento, da geração do código do *socket*, para troca de dados;
- Carregamento da função reconfigurável no *hardware* reconfigurável.

O desenvolvimento de aplicações no RTEV é descrito na Figura 20. Os objetos em destaque mostram as alterações realizadas no ambiente. Os retângulos denotam as ferramentas que compõem o ambiente visual e são os seguintes: GPPR (Gerador de Programas Paralelos e Reconfiguráveis), ATEPC (Analisador de Tempo de Execução e Pior Caso) e AE (Analisador de Escalonamento). O GPPR é complementado apresentando as subdivisões desta ferramenta, sejam elas: editor gráfico, editor de texto, gerador de código fonte e gerador de código para reconfigurável. Os arquivos gerados a cada estágio são representados por elipses, quais sejam: arquivos de projetos, arquivo *Makefile*, código fonte e arquivo executável. Os retângulos com cantos arredondados são usados para representar módulos de software com os quais o ambiente visual interage e corresponde ao compilador C e bibliotecas do Kernel Virtuoso.

O gerador de Programas Paralelos e Reconfiguráveis (GPPR) - é o principal alvo das alterações no sistema. O objetivo desta ferramenta é auxiliar na geração de código fonte dos programas executados na máquina paralela e gerar o código para execução das funções reconfiguráveis. No GPPR, as aplicações são construídas através de um modelo gráfico, que é utilizado para integrar as demais ferramentas do ambiente visual. As informações do modelo gráfico podem ser complementadas com descrições textuais, ou seja, trechos de código escritos pelo usuário. A partir destas informações, o GPPR gera automaticamente o código fonte da aplicação e arquivos do tipo *makefile*, que são utilizados para compilar e *linkeditar* o código fonte produzido.

No ambiente RTEV é adicionado ao GPP a geração de código para configuração e execução das funções reconfiguráveis, criando o GPPR. A Figura 21 ilustra essa geração realizada pelo GPPR. O gerador de código para as funções reconfiguráveis gera todo o código utilizado para troca dos dados a serem processados pela função reconfigurável, assim como o código necessário para a configuração da função no *hardware* reconfigurável. Também faz uso de informações de uma biblioteca de funções reconfiguráveis. Esta biblioteca é composta por funções

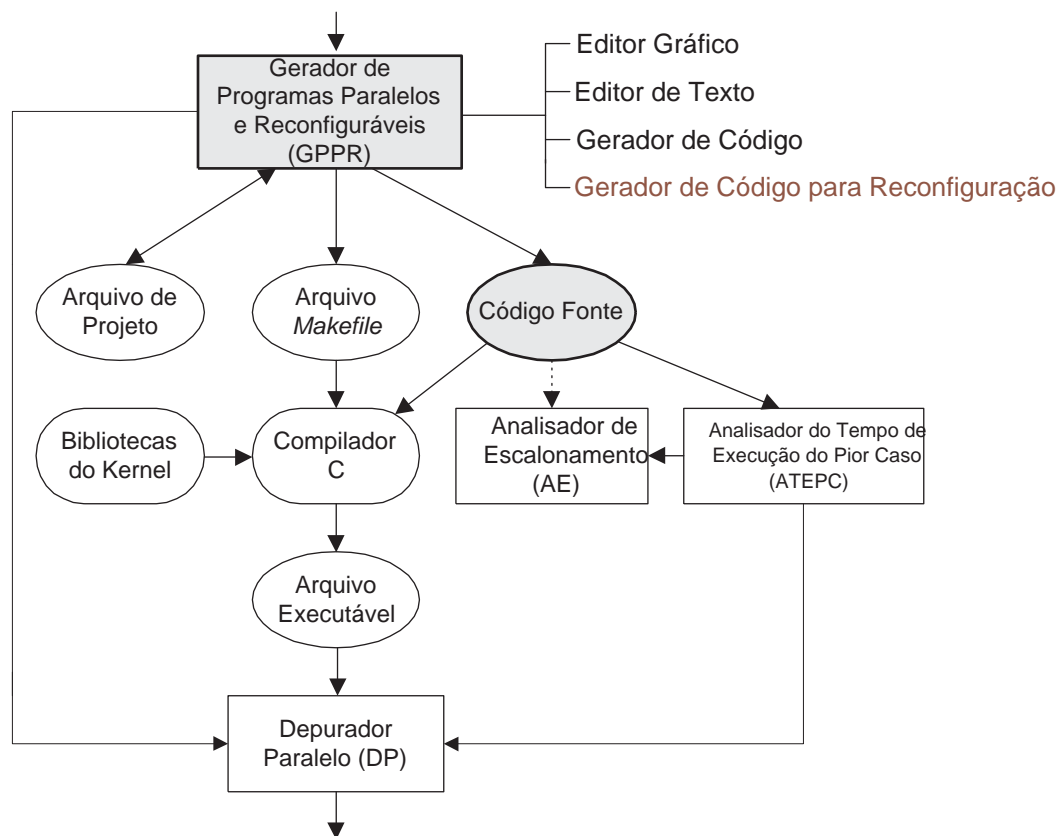


Figura 20: Componentes do desenvolvimento de aplicações no R-TEV

reconfiguráveis pré-desenvolvidas e possui as seguintes informações: nome das funções, arquivo *bitstream* a ser configurado, quantidade e tipo de parâmetros da função e variável de retorno.

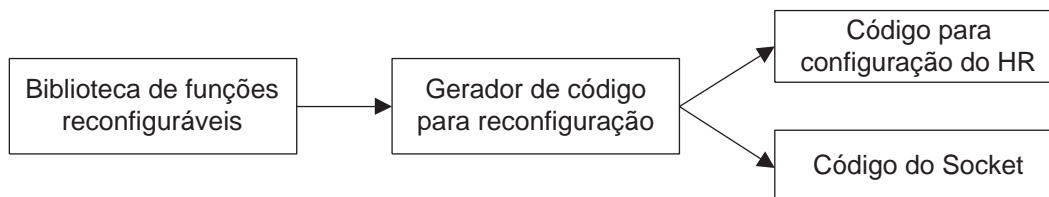


Figura 21: Geração de código reconfigurável realizada pelo GPPR

As funções a serem utilizadas na biblioteca reconfigurável devem ter sido desenvolvidas previamente. Para este desenvolvimento é sugerida a utilização de uma das metodologias apresentadas no Capítulo 3 adicionando-se à função o *socket* de comunicação. Este desenvolvimento será abordado mais adiante neste capítulo.

A arquitetura de *hardware* do sistema é apresentada na Figura 22. O sistema de *hardware* utilizado é composto pelo computador principal ATLAS e pelo kit XUP Virtex-II Pro, apresentados no capítulo anterior. Para este trabalho, os sistemas foram integrados através das interfaces

USB e *Ethernet* do Kit XUP Virtex-II Pro. A interface USB é utilizada para o carregamento da reconfiguração do *hardware* reconfigurável, PowerPCs e lógica configurável (CLBs). A interface *Ethernet* é utilizada para troca dos dados a serem processados. A troca de dados pela interface *Ethernet*, como já citado, é realizada através de um *socket* para comunicação.

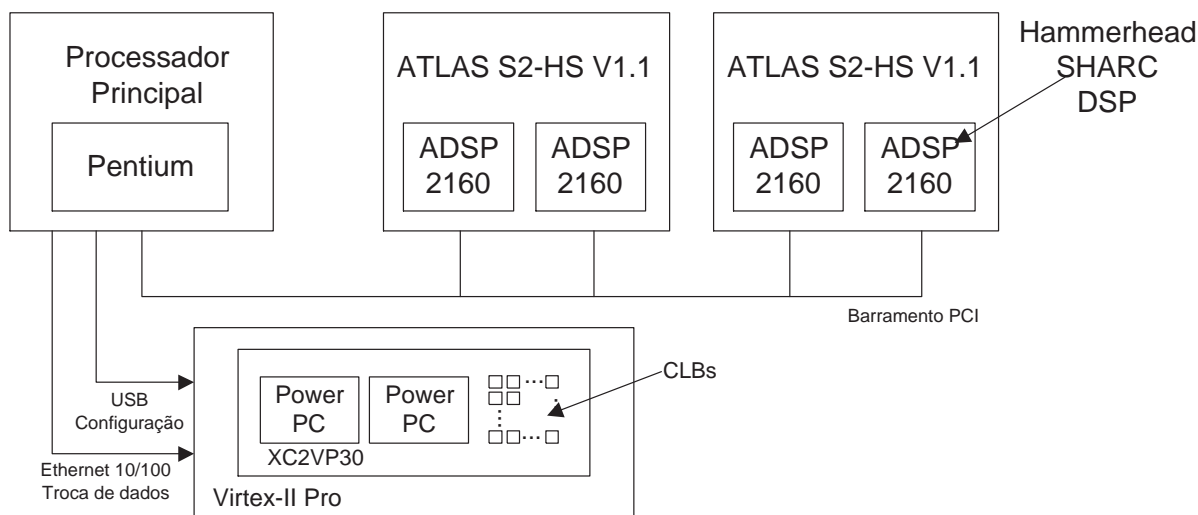


Figura 22: Sistema composto pelo ATLAS e XUP Virtex-II Pro

Neste capítulo todas estas etapas serão apresentadas e explicadas com detalhes.

### 5.3 Manipulação de Componentes Reconfiguráveis com R-TEV

Como já apresentado, o R-TEV aumenta a funcionalidade do ambiente TEV permitindo o uso de funções reconfiguráveis, desenvolvidas e incluídas dentro de uma biblioteca de funções reconfiguráveis. O sistema inclui automaticamente todo o código necessário para a comunicação com o hardware reconfigurável. Desta forma, o fluxo de desenvolvimento de aplicação com o R-TEV se mantém inalterado em relação ao TEV.

O componente reconfigurável dentro do R-TEV é muito semelhante a uma tarefa. Como apresentado na seção 4.1, aplicações no TEV são desenvolvidas utilizando modelos gráficos. Este modelo é representado por um grafo, onde nós denotam as estruturas de dados que compõem um programa paralelo, e linhas representam a comunicação e sincronização entre as estruturas. A interface para o programador incluir um componente reconfigurável é composta por uma janela com todas as funções reconfiguráveis da biblioteca, apresentando suas propriedades. A Figura 23 apresenta um exemplo de janela de funções reconfiguráveis do R-TEV. Nesta interface deve-se selecionar a função desejada e proceder o desenvolvimento, conectando o componente



reconfigurável com outros componentes.

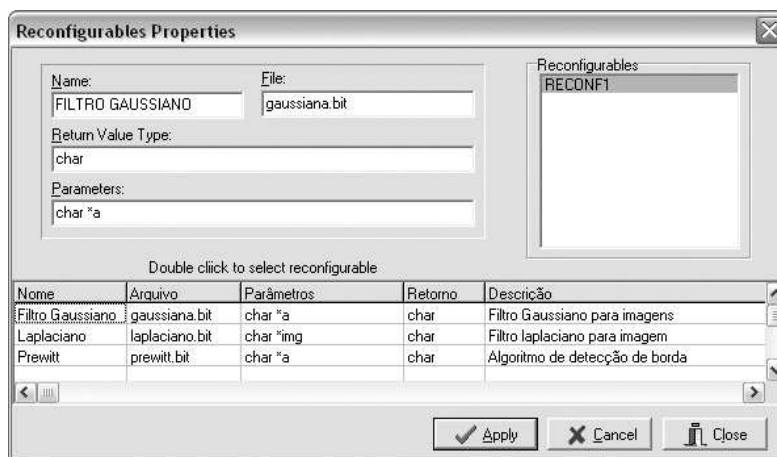


Figura 23: Janela de Funções reconfiguráveis do R-TEV

As especificações do componente reconfigurável são as seguintes:

- Nome: identifica a função;
- Descrição: descreve as características da função;
- Arquivo: aponta o arquivo *bitstream* de configuração do *hardware* para função;
- Parâmetros: especifica os parâmetros da função;
- Retorno: define o tipo da variável de retorno da função, se este existir.

Após a fase interativa do desenvolvimento de aplicações usando o ambiente R-TEV, o código fonte da aplicação é gerado automaticamente. Durante a geração do código da aplicação, é também gerado o código necessário para configuração do *hardware* reconfigurável, incluindo o *bit-stream* de reconfiguração e o *socket* do computador hospedeiro sem qualquer interação do usuário, de maneira transparente.

## 5.4 Biblioteca de Funções Reconfiguráveis

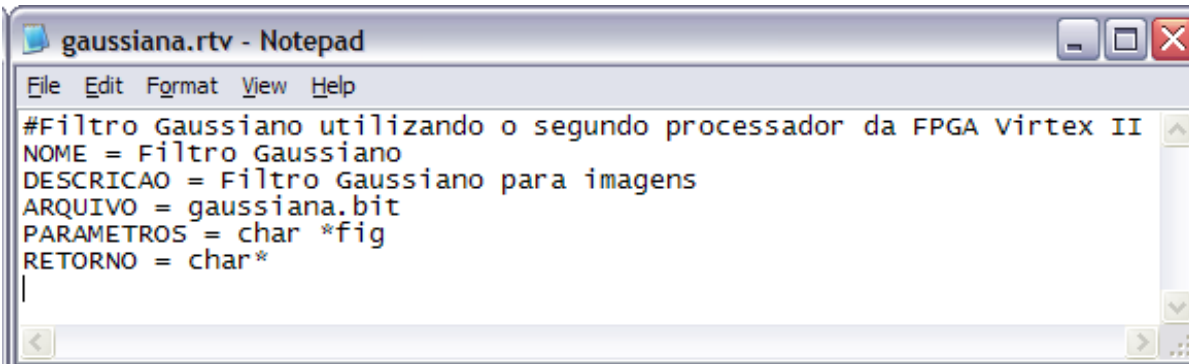
A biblioteca de funções reconfiguráveis é constituída por um conjunto de funções pré-desenvolvidas. As funções podem ser desenvolvidas utilizando-se qualquer metodologia ou ferramenta para o desenvolvimento de aplicações reconfiguráveis. Com o objetivo de auxiliar neste desenvolvimento foram apresentadas diversas metodologias no Capítulo 3. Durante o

desenvolvimento deve ser adicionado, a cada função reconfigurável, o *socket* de comunicação entre os sistemas.

Para a criação da biblioteca de funções reconfiguráveis, ou adição de funções a esta, deve-se colocar os arquivos *bitstream* em um diretório e adicionar um arquivo descritor. O arquivo descritor é um arquivo texto com o mesmo nome do arquivo *bitstream* e extensão *.rtv*. Este arquivo, mostrado na Figura 24, possui as especificações do componente reconfigurável:

- nome;
- descrição;
- nome do arquivo;
- parâmetros;
- tipo de retorno.

Cada especificação deve ser precedida de seu nome e um sinal de igual (=). Pode-se utilizar o símbolo # para adição de comentários no arquivo.



```
gaussiana.rtv - Notepad
File Edit Format View Help
#Filtro Gaussiano utilizando o segundo processador da FPGA Virtex II
NOME = Filtro Gaussiano
DESCRICAO = Filtro Gaussiano para imagens
ARQUIVO = gaussiana.bit
PARAMETROS = char *fig
RETORNO = char*
|
```

Figura 24: Arquivo descritor de função reconfiguráveis do R-TEV

## 5.5 Protocolo de Comunicação

Para que a operação de troca de dados ocorra de maneira transparente ao usuário é necessário a criação de um protocolo de comunicação entre o *hardware* reconfigurável, placa Virtex-II Pro, e o computador principal, computador ATLAS. Este protocolo objetiva ordenar a transmissão dos dados, podendo realizar a troca de uma ou mais variáveis, e a organização destes na memória RAM da placa Virtex-II Pro, doravante denominada *Hardware* Reconfigurável, ou HR, simplificada.

O funcionamento do protocolo é baseado em dois agentes, o HR e o computador principal. O seu funcionamento se inicia com a transmissão dos dados para o HR, o receptor, que os coloca de forma ordenada na memória e aguarda o processamento. Após o processamento, os dados resultantes são enviados ao computador principal.

Durante a primeira transmissão, do computador principal para o HR, no início da execução de uma aplicação reconfigurável, o *socket* de transmissão agrupa todas as variáveis de acordo com a seqüência de parâmetros da função reconfigurável. Os pacotes são então transmitidos com seu tamanho, nos quatro primeiros bytes, pois os DSPs manipulam apenas dados de 32 bits. Então o *socket* receptor (do HR) coloca todos os dados recebidos em seqüência na memória DDR, iniciando no endereço 0x13. Para controle de início e fim de recebimento e processamento é utilizado um *flag* em memória. Este *flag* está alocado no endereço 0x00 e a Tabela 2 apresenta os valores que ele pode assumir.

Tabela 2: Valores do *Flag*

Valor	Significado
0	Recebendo
1	Fim recebimento
2	Processando
3	Fim processamento

Os primeiros endereços de memória DDR do HR são utilizados como ponteiros para os dados processados. Esta organização deve ser conhecida no momento de desenvolvimento de aplicações reconfiguráveis, para que os dados possam ser manipulados corretamente. A Tabela 3 apresenta os endereços, a funcionalidade e o tamanho de cada ponteiro.

Tabela 3: Organização dos ponteiros de controle na memória DDR do HR

Endereço (hexa)	Tamanho (bytes)	Significado
0x00	1	Endereço de Flag
0x01	8	Endereço inicial dos dados processados
0x09	8	Endereço final dos dados processados
0x11	2	Quantidade de variáveis
0x13	-	Início dos dados recebidos

Após o fim do recebimento dos dados, o programa de socket do HR sinaliza o *flag* como fim de recebimento. Então, a função reconfigurável processa os dados e armazena as variáveis Var1,..., Var N, na memória DDR em seqüência, seguidas dos respectivos tamanhos Tam1, ..., Tam N. O endereço inicial é armazenado na posição 0x01, em seqüência ao *flag* e seguido pelo endereço final, pelo número de variáveis e os pelos dados recebidos. A Figura 25 mostra a organização da memória DDR.

Quando o HR termina o processamento dos dados da função reconfigurável o *flag* deve

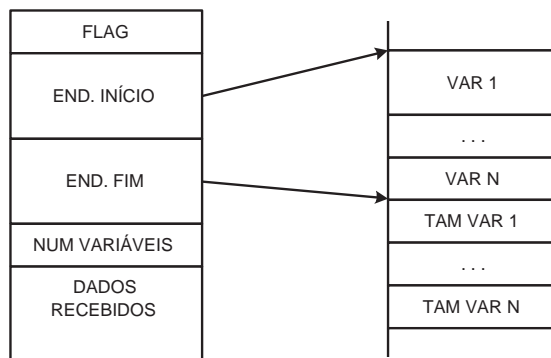


Figura 25: Organização dos dados na memória DDR do HR

ser ajustado para 3. Os dados devem então ser enviados de volta ao computador hospedeiro (que corresponde a uma operação de *write-back*). Primeiro, o HR transmite o tamanho da variável em um pacote com os quatro primeiros bytes 0, seguido de 4 bytes de tamanho. Então é transmitida a variável completa utilizando os pacotes de dados, composto por quatro primeiros bytes com o tamanho do pacote e os dados no restante do pacote. Em seguida, um pacote com os quatro primeiros valores 0 seguidos de 6 valores 1, que sinaliza o fim de variável. Se houver outra variável para transmitir, utiliza-se o mesmo algoritmo, até a última variável. O fim da transmissão é sinalizado por um pacote de 20 bytes de zeros. A Tabela 4 apresenta os pacotes que devem ser enviados, especificando o tamanho, valor dos 4 primeiros *bytes* (*bytes* de tamanho), valor do conteúdo do pacote e sua funcionalidade. A Figura 26 mostra uma representação gráfica dos 4 tipos de pacotes trocados entre o computador principal e o HR, e vice-versa.

Tabela 4: Descrição dos pacotes do *socket*

Tamanho	4 <sup>os</sup> bytes	Conteúdo	Funcionalidade
8	0	Tamanho da variável	Tamanho da variável
até 256	Tamanho Pacote	Dados	Pacotes que transmitem os dados
10	0	Tudo 1	Pacote que sinaliza fim de variável
20	0	Tudo 0	Pacote que sinaliza fim de transmissão

A Figura 27 apresenta um diagrama de estados do protocolo de comunicação. Apresenta o diagrama do computador principal 27a que é composto pela junção das variáveis (1), envio dos dados (2), até que encontre o fim dos dados, quando encontra o fim dos dados envia o pacote que sinaliza o fim de transmissão (3). Fica então aguardando o recebimento (4) até receber o tamanho da variável (5), seguido pelos dados da variável (6) e então o pacote de fim de variável (7), caso haja nova variável volta para o recebimento do pacote com o tamanho da variável (5), caso contrário recebe o pacote de fim de dados (8). No lado do HR 27b, este fica aguardando o recebimento (1), recebe então os dados (2), até receber o pacote de fim de

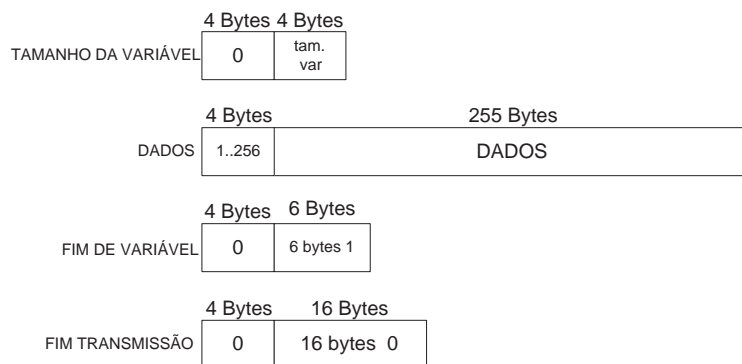


Figura 26: Apresentação gráfica dos pacotes do *socket*

transmissão (3). Após receber os dados, aguarda o processamento verificando o seu fim (4); com o fim do processamento envia o tamanho da variável (5), e então os dados da variável (6) e, ao final, o pacote de fim de variável (7), caso haja outra variável volta para envio do tamanho da variável (5), caso contrário envia pacote de fim de transmissão (8).

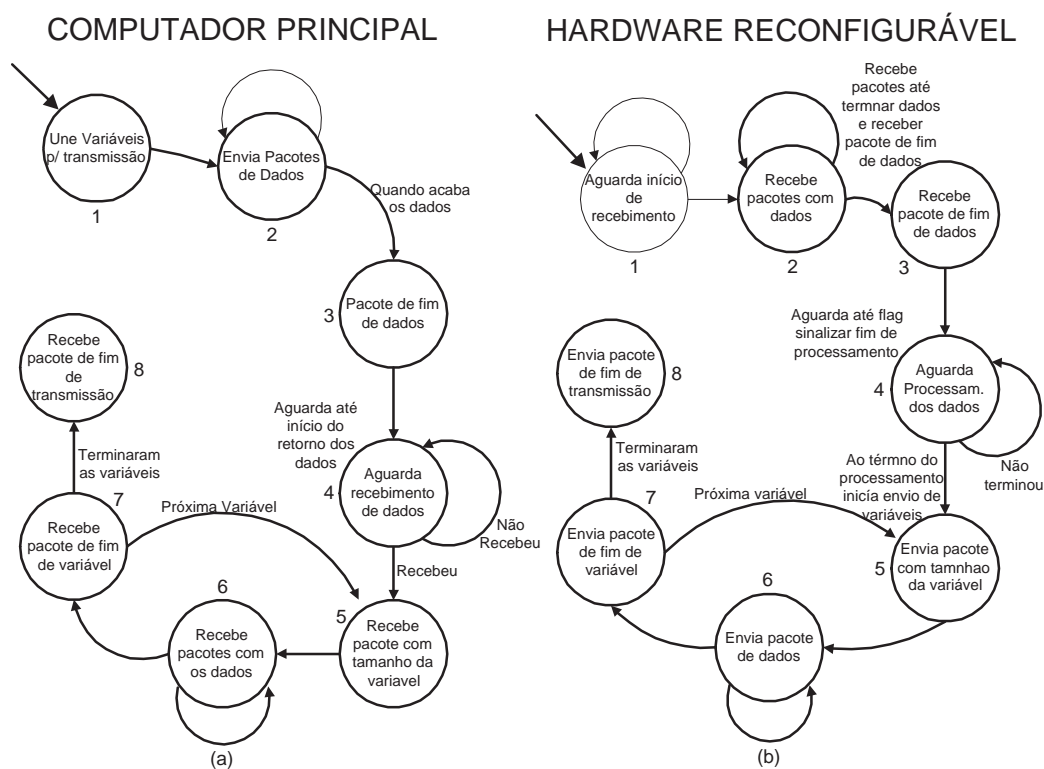


Figura 27: Diagrama de estados do protocolo de comunicação para o computador principal (a) e o HR (b)

## 5.6 Socket

Devido às características da placa Virtex-II Pro, o acoplamento da unidade reconfigurável e o computador hospedeiro é fraco. Tem-se uma unidade reconfigurável independente que comunica com o computador hospedeiro através de uma interface Ethernet 10/100 por um socket. Por outro lado, o acoplamento dos *slices* de FPGA com os processadores PowerPC internos ao XC2VP30 é forte. Felizmente, *socket* é o principal método de comunicação do kernel Virtuoso com componentes externos, então toda comunicação é realizada através de um *socket* UDP/IP.

No lado do HR, o *socket* foi escrito em linguagem ANSI C e utilizada a biblioteca de rede da Xilinx para processadores embarcados libXil Net (XILINX, 2005a), e é executado em um dos PowerPCs da FPGA. A biblioteca utilizada inclui funções de suporte a TCP/IP e UDP e uma interface de programação de alto nível. A biblioteca suporta múltiplas conexões e portanto suporta múltiplos clientes.

No sistema ATLAS, no lado do computador hospedeiro, é usado um canal que consiste de uma fila de leitura e escrita, e como opcional um canal *bufferizado* como *socket* TCP/IP, utilizando as primitivas HSSocketChannelOpen, HSChannelPutW, and HSChannelGetW, para conexão e troca de dados.

A biblioteca libXil Net, utilizada no HR, é um tanto limitada não permitindo realizar a comunicação com o ATLAS utilizando-se TCP/IP. Os pacotes não eram remontados corretamente na recepção no HR. Para solucionar o problema foi necessária a utilização de programa auxiliar no próprio sistema ATLAS, entretanto sem utilizar o /it kernel. Este programa recebe a conexão TCP/IP do *kernel* Virtuoso e retransmite os pacotes, sem qualquer alteração via UDP/IP para o HR. No retorno dos dados, do HR para o Virtuoso realiza operação oposta, recebe em UDP/IP do HR e repassa via TCP/IP para o Virtuoso. Nesta etapa de retorno dos dados ao Virtuoso é necessária a utilização de um buffer para evitar a perda de pacotes pelo *kernel* Virtuoso. Este pode ser entendido como o responsável por realizar a ligação entre o *socket* TCP/IP do *kernel* Virtuoso com o *socket* UDP/IP do HR.

O *socket* foi desenvolvido seguindo-se os passos do protocolo descrito anteriormente neste capítulo. No HR, o programa *socket* é implementado e executado em um dos PowerPCs disponíveis na FPGA. Este *socket* deve ser incluído no desenvolvimento das funções reconfiguráveis para o sistema. No lado do computador hospedeiro, o *socket* é automaticamente gerado quando um componente reconfigurável é selecionado no R-TEV. Os códigos dos *sockets* são apresentados no Apêndice.

## 5.7 Geração de Código

A geração de código dentro do ambiente R-TEV é feita de forma automática, mantendo o mesmo modo do ambiente TEV. Os códigos necessários para utilização das funções reconfiguráveis são adicionados aos códigos gerados pelo TEV.

São geradas duas porções de código. A primeira contém o código necessário ao carregamento do arquivo *bitstream* de configuração da FPGA, utilizando o aplicativo IMPACT do pacote de aplicativos da Xilinx. IMPACT é um aplicativo que realiza a configuração de componentes da Xilinx. Essa configuração pode ser realizada utilizando-se um ambiente gráfico ou em modo texto com processamento em *batch*, modo utilizado neste trabalho.

A outra porção corresponde ao código do *socket*. É incluído nesta parte todo o código necessário para realização da transmissão dos dados dos parâmetros da função reconfigurável, de acordo com o protocolo de transmissão das variáveis. A transmissão inclui o envio dos parâmetros e recebimento dos dados de retorno da função, assim como a atualização de parâmetros que por ventura tenham sido passados por referência. Portanto, não é necessário qualquer esforço adicional do desenvolvedor para utilização das funções reconfiguráveis.

## 5.8 Desenvolvimento das Funções Reconfiguráveis

Devido às características do *hardware* reconfigurável utilizado neste trabalho, o kit de desenvolvimento XUP Virtex-II Pro e à necessidade de implementação de um *socket*, baseado em um protocolo definido para a troca de dados entre o HR e o computador principal, o desenvolvimento das funções reconfiguráveis deve seguir algumas regras. Basicamente, as funções devem ser acompanhadas do *socket* de troca de dados e devem seguir a organização de memória descrita no protocolo.

O *socket*, que deve ser incluído no projeto de funções, é executado utilizando um dos processadores PowerPCs existentes no FPGA Virtex-II Pro da placa. Neste caso, como já apresentado, o desenvolvimento deve ser realizado utilizando-se o ambiente de desenvolvimento EDK (*Embedded Development Kit*) da Xilinx, que consiste de um conjunto de ferramentas dentre as quais destaca-se Xilinx Platform Studio (XPS). Mais especificamente, deve-se utilizar o *Base System Builder* (BSB), uma ferramenta de software que ajuda o usuário a construir um projeto de trabalho específico para a placa de desenvolvimento.

O BSB oferece um amplo número de opções para especificação e configuração dos sistemas básicos do HR. Estas opções incluem uma interface de depuração, configuração de *cache*, tipo

e tamanho de memória, e seleção de periféricos. Para cada opção, valores padrões são apresentados na interface gráfica, auxiliando o desenvolvimento. Ao fim do BSB, um arquivo de especificação de *hardware* (MHS) e um arquivo de especificação de *software* (MSS) são criados e carregados dentro do projeto no XPS.

Após a criação do projeto dentro do XPS, deve-se inserir em um dos processadores um novo projeto de *software*, adicionando a este projeto o arquivo contendo o código do *socket*. Além de adicionar o código do *socket*, é necessário especificar no XPS a utilização da biblioteca LibXil Net. Para isso, basta adicionar biblioteca dentro das configurações da plataforma de *software*.

A função a ser executada no hardware reconfigurável pode ser desenvolvida utilizando-se qualquer metodologia. Para isso, apresenta-se uma ampla gama de possíveis metodologias para o desenvolvimento das mais diversificadas aplicações, nas mais diversificadas áreas. O método a ser utilizado, e a ferramenta a ser utilizada, deve ser definido de acordo com as características da função e as necessidades da aplicação. Deve-se apenas tomar o cuidado de manter a organização de dados na memória, proposta no protocolo.

Após definido o método e desenvolvida a função, deve-se gerar um arquivo de configuração (*bitstream*). Este arquivo deve ser inserido dentro do projeto de hardware na ferramenta XPS como um *IP Core*. Este *IP Core* deve ser então direcionado ao *hardware* restante do FPGA, obedecendo a estrutura de barramentos do próprio FPGA.

Outra possibilidade para o desenvolvimento é utilizar o segundo processador PowerPC disponível no FPGA, como realizado no estudo de caso que será apresentado posteriormente. Neste caso, todo o desenvolvimento é realizado utilizando-se a ferramenta XPS. Para este tipo de desenvolvimento, é necessária a configuração e organização dos barramentos de acesso a periféricos e memória de acordo com o manual (BENNETT, 2005).

Em ambos os casos, deve-se ao final do desenvolvimento gerar o arquivo *bitstream* da função e adicioná-la à biblioteca de funções reconfiguráveis dentro do ambiente de desenvolvimento R-TEV.

## 5.9 Considerações Finais

Neste capítulo, foi apresentado o desenvolvimento do ambiente R-TEV. As conclusões sobre este trabalho serão melhor discutidas no capítulo seguinte de resultados obtidos e no capítulo final de conclusões e trabalhos futuros.

Neste projeto foi realizada uma extensão no ambiente de desenvolvimento permitindo-se



adicionar funções reconfiguráveis, adicionando ao ambiente todas as necessidades para correto carregamento e execução das funções no *hardware* reconfigurável, bastando ao usuário apenas selecionar as funções desejadas, utilizando-se o *kernel* Virtuoso e o computador Atlas normalmente. O ambiente de desenvolvimento TEV com a extensão para utilização de funções reconfiguráveis foi intitulado de R-TEV.

## 6 *Estudo de Caso e Resultados Obtidos*

Neste capítulo será apresentado um estudo de caso com o objetivo de validar o sistema desenvolvido. Como estudo de caso foram implementados dois algoritmos de processamento de imagens: o detector de bordas *Prewitt*; e o filtro *Gaussiano*. Ambos foram implementados utilizando-se o segundo processador PowerPC disponível no FPGA.

A seguir serão feitas as descrições de cada filtro implementado, apresentando seus devidos códigos, e finalmente serão apresentados os resultados obtidos com essa implementação.

### 6.1 Prewitt

*Prewitt* é um método de detecção de bordas que calcula a máxima resposta da convolução usando duas máscaras 3x3, horizontal e vertical (PREWITT, 1970). A magnitude do mapa dos gradientes é calculada para encontrar o ponto de maior gradiente. Este arranjo de magnitudes terá maior valor onde o gradiente da imagem for maior, mas isto não é o suficiente para encontrar as bordas. Os cumes largos no arranjo de magnitudes devem ser diluídos de modo que somente os valores nos pontos da maior mudança permaneçam. Isto é conhecido como não máxima supressão, e seus resultados são bordas afinadas.

O resultado é passado então por um *Threshold* para produzir o mapa de bordas final. Entretanto, a saída do *threshold* é extremamente sensível e não há procedimento automático para determinação do valor ideal para o *threshold* para todas imagens. A Figura 28 apresenta as máscaras horizontal (X) e vertical (Y) para o filtro.

$$\begin{array}{ccc}
 -1 & 0 & 1 \\
 -1 & 0 & 1 \\
 -1 & 0 & 1 \\
 X & & \\
 (a) & & 
 \end{array}
 \qquad
 \begin{array}{ccc}
 1 & 1 & 1 \\
 0 & 0 & 0 \\
 -1 & -1 & -1 \\
 Y & & \\
 (b) & & 
 \end{array}$$

Figura 28: Máscaras (a) horizontal e (b) vertical

A equação 6.1 apresnta o cálculo de um pixel de imagem filtrada para o algoritmo *Pre-*

witt, realizando a convolução de uma janela 3x3 da imagem (W) com as máscaras (X e Y). A operação de convolução (\*) é obtida pela somatória dos produtos dos valores correspondentes das máscaras 3x3 e da janela 3x3 da imagem. Em seguida aplica-se o operador *threshold* ao resultado equação 6.2.

$$G = \sqrt{(W * X)^2 + (W * Y)^2} \quad (6.1)$$

$$Threshold(G) = \begin{cases} 255 & \text{se } G > 127 \\ 0 & \text{caso contrário} \end{cases} \quad (6.2)$$

O código 6.1 apresenta o código do laço principal da implementação do algoritmo *Prewitt* escrito em C e executado no PowerPC do FPGA do HR. Nota-se que trata-se de um código ANSI-C sem bibliotecas para manipulação da imagem. O código completo é apresentado no ApêndiceE.

Código 6.1: Código do filtro Prewitt

```

1  for (i = 0; i <= ref; i++) { //laco para calculo
2      SUMH = SUMV = 0; // calculo da convolucao
3      SUMH = SUMH + *ln1_ptr;
4      SUMV = SUMV - *ln1_ptr;
5      SUMV = SUMV - *ln2_ptr;
6      SUMH = SUMH - *ln3_ptr;
7      SUMV = SUMV - *ln3_ptr;
8      ln1_ptr++;
9      ln2_ptr++;
10     ln3_ptr++;
11     SUMH = SUMH + *ln1_ptr;
12     SUMH = SUMH - *ln3_ptr;
13     ln1_ptr++;
14     ln2_ptr++;
15     ln3_ptr++;
16     SUMH = SUMH + *ln1_ptr;
17     SUMV = SUMV + *ln1_ptr;
18     SUMV = SUMV + *ln2_ptr;
19     SUMH = SUMH - *ln3_ptr;
20     SUMV = SUMV + *ln3_ptr;
21     G = sqrt(SUMH * SUMH + SUMV * SUMV);
22     if (G > 127) // threshold
23         *nova_imagem = 255;
24     else
25         *nova_imagem = 0;
26     nova_imagem++;
27     fd1++;
28     if ((fd1 + 2) != cols) { // atualizacao dos ponteiros
29         ln1_ptr--;
30         ln2_ptr--;
31         ln3_ptr--;
32     } else {
33         ln1_ptr++;
34         ln2_ptr++;
35         ln3_ptr++;
36         *nova_imagem = 0;
37         nova_imagem++;
38         *nova_imagem = 0;
39         nova_imagem++;
40         fd1 = 0;

```

```
41 }
42 }
```

## 6.2 Filtro Gaussiano

O filtro *Gaussiano* é uma convolução 2D usado para "borrar" imagens, remover detalhes e espalhar ruídos (BABAUD et al., 1986). A idéia de uma suavização *Gaussiana* é utilizar esta distribuição 2D como uma "função de espalhamento", e isto é conseguido por uma convolução. Sendo a imagem armazenada como um conjunto de pixels discretos é necessário produzir uma aproximação discreta para a função *Gaussiana*, que é dada pela função  $G(x,y)$ , equação 6.3 (WITKIN, 1983). A Figura 29 apresenta a máscara 3x3 utilizada nesta implementação, que faz a função  $G(x,y)$ .

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (6.3)$$

$$\begin{array}{ccc} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{array}$$

Figura 29: Máscaras Gaussiana 3x3

O código 6.2 apresenta o código do laço principal da implementação do algoritmo do filtro *Gaussiano* escrito em C e executado no PowerPC do FPGA do HR. Neta-se que trata-se de um código ANSI-C sem bibliotecas para manipulação da imagem. O código completo é apresentado no ApêndiceF.

Código 6.2: Código do filtro Gaussiano

```
43 for (i = 0; i <= rcf; i++) { //laco para calculo
44     NV = 0; // calculo da convolucao
45     NV = *ln1_ptr/16 + *ln2_ptr/8 + *ln3_ptr/16;
46     ln1_ptr++;
47     ln2_ptr++;
48     ln3_ptr++;
49     NV += *ln1_ptr/8 + *ln2_ptr/4 + *ln3_ptr/8;
50     ln1_ptr++;
51     ln2_ptr++;
52     ln3_ptr++;
53     NV += *ln1_ptr/16 + *ln2_ptr/8 + *ln3_ptr/16;
54     *nova_imagem = NV;
55     if(*nova_imagem > 255)
56         *nova_imagem = 255;
57     if(*nova_imagem < 0 )
58         *nova_imagem = 0;
59     nova_imagem++;
60     fd1++;
61     if((fd1 + 2)!= cols) {
62         ln1_ptr--;
63         ln2_ptr--;
64         ln3_ptr--;
65     } else {
66         ln1_ptr++;
67         ln2_ptr++;
68         ln3_ptr++;
69     }
70     *nova_imagem = 0;
71 }
```

```

70     nova_imagem++;
71     *nova_imagem = 0;
72     nova_imagem++;
73     fdl = 0;
74 }
75 }

```

### 6.3 O Desenvolvimento

Foi desenvolvida uma aplicação no ambiente RTEV utilizando as duas funções reconfiguráveis apresentadas. A Figura 30 ilustra a aplicação. A aplicação possui uma função responsável por carregar as imagens de/para disco (IMAGEM) e disponibilizá-las para as funções reconfiguráveis (PREWITT e GAUSSIA). As funções reconfiguráveis são aplicadas às imagens de forma independente, e são gerenciadas pela tarefa PRINCIPAL. Uma fila (CANAL) é responsável pelo canal de comunicação entre as funções.

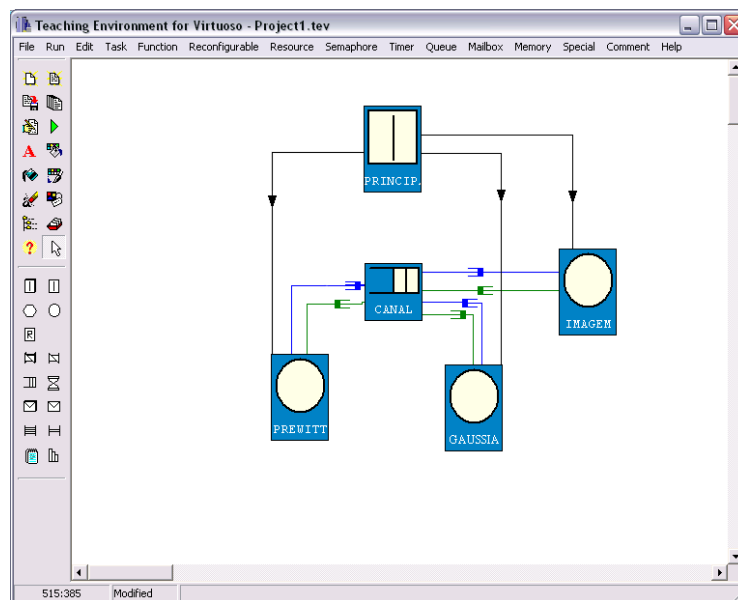


Figura 30: Aplicação com as funções reconfiguráveis *Prewitt* e filtro *Gaussiano*

A Figura 31a apresenta a imagem original que foi processada. Neste estudo de caso utilizou-se três tamanhos distintos da imagem: 800x600, 400x300, 200x150, com 8 bits por pixels. A Figura 31b apresenta a imagem processada pela função de detecção de bordas Filtro *Prewitt*. A Figura 31c apresenta a imagem processada pela função de borramento Filtro *Gaussiano*.

O desenvolvimento das funções reconfiguráveis foi realizado no ambiente de desenvolvimento da Xilinx EDK, utilizando-se a plataforma de desenvolvimento de aplicações com núcleos de processadores *Xilinx Platform Studio*. Esta plataforma permite o desenvolvimento de aplicações para os processadores do FPGA em linguagem C, criando-se um projeto com o *Base System Builder* (BSB) para a placa específica que se deseja. O BSB permite selecionar os periféricos disponíveis no Kit de desenvolvimento, e adiciona os drivers e interconecta-os para utilização no projeto. Permite também, a adição de *IP Cores*, ou núcleos de hardware, desen-

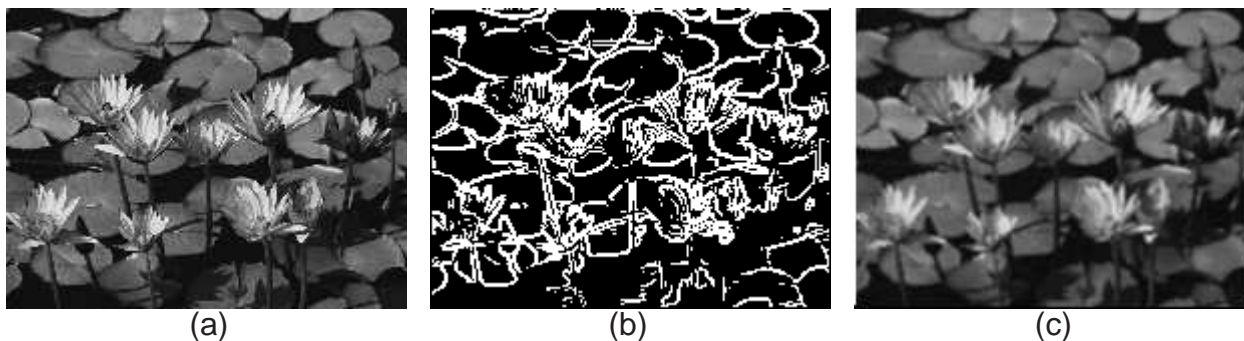


Figura 31: Imagens Processadas: (a) Imagem original; (b) Filtro Prewitt; e (c) Filtro Gaussiano

volvidos (ou adquiridos) e interconectá-los aos outros periféricos e dispositivos do HR através de seus barramentos.

Para o desenvolvimento das funções utilizou-se os dois processadores, a memória DDR, a interface *Ethernet*, a interface serial RS232, a interface de programação JTEG e o estrutura de barramentos. O primeiro processador *PowerPC* recebeu o código referente ao socket apresentado no capítulo anterior. O segundo *PowerPC* foi programado com o algoritmo da função desejada.

Para utilização dos dois núcleos processadores em conjunto, compartilhando memória e interfaces de maneira sincronizada foi necessária uma alteração na estrutura de barramentos do projeto. O procedimento completo é descrito por Bennett em (BENNETT, 2005). Estas alterações permitem acesso a recursos compartilhados por ambos processadores, e provê mecanismo para sincronização do acesso a recursos compartilhados.

## 6.4 Resultados

O objetivo principal deste estudo de caso é demonstrar a utilização do sistema desenvolvido e validá-lo como ambiente de desenvolvimento.

A Tabela 5 apresenta os resultados obtidos com execução das funções reconfiguráveis. Realizou-se experimentos com três tamanhos de imagens diferentes, 800x600, 400x300 e 200x150 pixel para as duas funções reconfiguráveis. As duas primeiras colunas apresentam a função aplicada e tamanho da imagem. A coluna seguinte apresenta o atraso devido a configuração do dispositivo reconfigurável FPGA, que é igual para todos. As três colunas seguintes apresentam o atraso de comunicação de cada experimento, sendo na sequência: atraso pelo recebimento dos dados; atraso pelo envio dos dados processados; e atraso total de comunicação. A seguir são apresentados o tempo de processamento e a quantidade de dados enviados ou recebidos e processados. A última coluna apresenta a eficiência do sistema, que é dada pela razão entre o tempo de processamento e o atraso total.

Analisando a Tabela 5 pode-se constatar que, para computações com baixo tempo de processamento, a utilização do sistema torna-se inadequada, pois existe um atraso fixo de 4s de configuração do FPGA. Desconsiderando-se o tempo de programação a eficiência do sistema melhora consideravelmente, o pior caso que possui eficiência de apenas 0,04 passaria a ter eficiência de 0,50 e o melhor caso passaria de 0,88 para 0,95. Uma alternativa a ser considerada

Tabela 5: Resultados do Estudo de Caso

Função Reconfigurável	Tamanho da Imagem	Atraso de Configuração	Atraso de Recebimento	Atraso de Envio	Atraso Total	Tempo de Processamento	Quantidade de Dados	Eficiência
Prewitt	200x150	4000ms	111ms	50ms	4161ms	3050ms	31080b	0,42
	400x300	4000ms	456ms	197ms	4653ms	12163ms	121080b	0,72
	800x600	4000ms	1787ms	770ms	6557ms	48709ms	481080b	0,88
Gaussiana	200x150	4000ms	111ms	46ms	4157ms	157ms	31080b	0,04
	400x300	4000ms	435ms	181ms	4616ms	631ms	121080b	0,11
	800x600	4000ms	1730ms	719ms	6449ms	2525ms	481080b	0,28

é a utilização da Plataforma FLASH PROM para configuração do FPGA, disponível em cartão removível no sistema Virtex II - Pro. O uso dessa plataforma para armazenamento das funções reconfiguráveis elimina o tempo de transferência dos dados de reconfiguração do computador principal para o HR.

Observando-se o tempo de troca dados, como já era esperado, constatou-se sua influência no desempenho e eficiência do sistema. Entretanto, observou-se que este não é o principal gargalo do sistema, levando-se em consideração sistemas com baixa troca de dados em relação ao tempo de processamento.

Para o Filtro *Prewitt*, que requer uma quantidade maior de processamento com o mesmo volume de dados, constatou-se eficiência maior para o processamento do que o filtro Gaussiano. Em dois dos três casos testados obteve-se eficiência superior a 0,70, o que pode ser considerada bastante relevante.

## 6.5 Implementação nos CLBs

Apesar de não ser alvo deste estudo de caso a implementação de funções nos CLBs do FPGA pode trazer um ganho significativo de desempenho. Serão apresentadas aqui duas opções para o desenvolvimento do Filtro *Prewitt*. Uma implementação da função feita diretamente em VHDL, e uma utilizando a linguagem apresentada no Capítulo 3, SA-C.

Estas implementações referem-se ao núcleo do algoritmo, somente a porção do processamento da imagem. Deve ser adicionado a estas o acesso a memória, leitura e escrita dos dados. O desenvolvimento deve ser realizado para o FPGA Virtex-II Pro xc2vp30 ff896, e deve ser então gerado um *Core* para inclusão no projeto dentro do *Xilinx Platform Studio*. Após gerado o *Core* dentro do *Xilinx Platform Studio* utiliza-se uma ferramenta que auxilia na importação e criação de *Cores* e periféricos, chamada *Create/Import Peripheral*. Esta ferramenta realiza a interconexão com os barramentos da placa permitindo a interação com os núcleos de processadores, memória e periféricos do projeto e cria os *drivers* necessários.

Outra opção para o desenvolvimento é exportar o projeto realizado dentro do *Xilinx Platform Studio* para um projeto do ISE, principal ferramenta de desenvolvimento da Xilinx. O projeto é exportado como um componente (caixa preta) a ser interligado a outros componentes possibilitando a criação de um projeto maior.

## 6.5.1 VHDL para Prewitt

O Código 6.3 apresenta o código VHDL para implementação do núcleo do Filtro *Prewitt*. Este código é apresentado em VHDL Comportamental, e será convertido em VHDL RTL sintetizável pela ferramenta de desenvolvimento da Xilinx. A porção principal do código é composta por somas, subtrações e multiplicações. O paralelismo de execução é resolvido no momento da conversão para o código VHDL RTL. Este paralelismo é guiado pelas restrições descritas no código. Pode-se ver, nas linhas 119 e 120 do código, que os parênteses guiam o paralelismo implementado posteriormente.

Código 6.3: Código do Filtro Prewitt em VHDL

```

76 library ieee;
77 use ieee.std_logic_1164.all;
78 use ieee.std_logic_unsigned.all;
79 use ieee.std_logic_arith.all;
80
81 LIBRARY LPM;
82 USE LPM.lpm_components.all;
83
84 entity Nucleo_Prewitt is
85     port (end_mem: in std_logic_vector (7 downto 0);
86           clk: in std_logic;
87           s: out std_logic_vector (7 downto 0));
88 end Nucleo_Prewitt;
89
90 architecture arch of Nucleo_Prewitt is
91
92     component read_DDR
93     port ( DDR_Adr : in std_logic_vector (7 downto 0);
94           DDR_clk, DDR_REn : in std_logic;
95           DDR_D: out std_logic_vector (7 downto 0));
96     end component;
97
98     constant K: integer := 127;           -- circuit has four stages
99
100 -- use type and subtype to define the complex signals
101 subtype bit8 is std_logic_vector(7 downto 0);
102 type kx8 is array (8 downto 0) of bit8;
103
104 -- define signal in type of arrays
105 signal ln_ptr      : kx8;
106 signal SUMH, SUMV : bit8;
107 signal G           : bit8;
108
109 begin
110     read_DDR port map(end_mem => DDR_Adr, clk => DDR_clk, '1' => DDR_REn, DDR_D => ln_ptr);
111     stages: process (rst, clk)
112     begin
113         if (rst='0') then
114             ln_ptr <= ('range => '0');
115             SUMH <= (others => '0');
116             SUMC <= (others => '0');
117             SUMG <= (others => '0');
118         elsif (clk'event and clk = '1') then
119             SUMH <= (((ln_ptr(0) - ln_ptr(2)) + (ln_ptr(4)) - ln_ptr(5))) + (ln_ptr(6) - ln_ptr(8)
120             ));
121             SUMV <= (((ln_ptr(0) - ln_ptr(1)) 2 ln_ptr(2)) + ((ln_ptr(6) + (ln_ptr(7)) - ln_ptr
122             (8))));
123             G <= SQRT( (SUMH * SUMH) + (SUMV * SUMV) );
124         if {G > K} then
125             s <= "11111111";
126         else
127             s <= "00000000";
128         end if;
129     end process;
130 end arch;

```



Este código VHDL deve ser unido manualmente a um código para acesso e gerenciamento de memória e deve ser sintetizado e gerado um *Core* como descrito anteriormente.

## 6.5.2 Prewitt SA-C

O Código 6.4 apresenta o código para o Filtro Prewitt implementado em SA-C. O laço externo é utilizado para extrair sub-arranjos 3x3 de uma imagem. O corpo deste laço aplica duas máscaras na janela extraída *W*, produzindo a magnitude. Um arranjo destas magnitudes é retornado como resultado do programa.

O compilador SA-C converte o programa para um gráfico hierárquico de dependência de dados e fluxo de controle (DDFG). Este gráfico é utilizado para muitas otimizações, como eliminação de código morto, movimentação de código invariante, deserrenlar de laços, entre outras, algumas gerais e outras específicas do SA-C e plataforma alvo. Depois de otimizado o programa é convertido em uma combinação de gráficos de fluxo (DFGs) de dados e código específico. Os DFGs são então compilados em código VHDL.

Código 6.4: Código do Filtro Prewitt em SA-C

```

129 int16 [::] main (uint8 Image[::]) {
130     int16 H[3,3] = {{-1, -1, -1}, { 0, 0, 0}, { 1, 1, 1}};
131     int16 V[3,3] = {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};
132     int16 M[::] =
133         for window W[3,3] in Image {
134             int16 dfdy, int16 dfdx =
135                 for h in H dot w in W dot v in V
136                     return (sum(h*w), sum(v*w));
137             int16 magnitude =
138                 sqrt (dfdy*dfdy+dfdx*dfdx);
139             }return (array (magnitude));
140 }return (M);

```

## 6.6 Considerações Finais

Este estudo de caso demonstra a viabilidade do ambiente desenvolvido. Permite também observar as condições ideais para seleção de aplicações a serem desenvolvidas no ambiente. De acordo com a análise feita na discussão dos resultados o programa deve ter um tempo de processamento relativamente grande, considerando-se a quantidade de dados e tempo de 4s de configuração do FPGA.

## 7 *Conclusão e Trabalhos Futuros*

### 7.1 **Conclusão**

Este trabalho constitui-se da adaptação do ambiente TEV para o desenvolvimento de aplicações paralelas de tempo-real em conjunto com funções reconfiguráveis, criando o ambiente R-TEV. Muitas das ferramentas utilizadas possuem pouca ou quase nenhuma documentação e este tipo de integração não possui precedentes nestas ferramentas. Isto tornou o trabalho muito lento e necessitou de muita pesquisa e perseverança.

O ambiente desenvolvido atingiu os objetivos esperados, facilitando a inclusão de funções reconfiguráveis em sistemas comuns. Como já afirmado diversas vezes no trabalho, o fluxo de desenvolvimento do ambiente R-TEV não foi alterado em relação ao ambiente TEV, portanto não é necessário qualquer tipo de conhecimento específico do desenvolvedor para utilizá-lo.

Este trabalho não teve como objetivo o desenvolvimento das funções reconfiguráveis, pois para tanto já existe uma ampla gama de trabalhos propostos. Muitos destes foram apresentados aqui, para os mais diferentes tipos de aplicações e abordagens de desenvolvimento. Infelizmente, este tipo de desenvolvimento ainda requer uma alta especialização do desenvolvedor e um amplo conhecimento da arquitetura alvo, e é muito difícil que seja diferente. Entretanto, muitas das metodologias apresentadas suprimem muitos detalhes e conseguem simplificar, mesmo que, em parte este desenvolvimento. Observando a evolução destas metodologias, pode-se concluir que em breve teremos métodos e ferramentas ao alcance de usuários bem menos especializados tornando cada vez mais palpável a utilização da computação reconfigurável.

Analisando o estudo de caso e as características de todo o trabalho pode-se afirmar que um ponto fraco é o nível de acoplamento da placa com o computador hospedeiro que pode criar um gargalo para o sistema, inviabilizando ou diminuindo o desempenho de aplicações que possuam alto tráfego de dados entre o HR e o computador hospedeiro. Este ponto pode ser superado utilizando interfaces de comunicação mais eficientes. Outro ponto é o atraso de reconfiguração, que depende de características do hardware da placa e mais uma vez pode ser superado com a utilização de outros *hardwares* reconfiguráveis.

Em uma análise geral, levando-se em consideração os pontos fortes e fracos levantados aqui, pode-se dizer que o objetivo do trabalho foi alcançado e um primeiro passo foi dado para a integração e facilitação para os usuários dos sistemas de desenvolvimento.

## 7.2 Trabalhos Futuros

A utilização e suporte de diferentes *hardwares* reconfiguráveis, com diferentes interfaces de comunicação e diferentes características de reconfiguração, seria muito conveniente. Isto tornaria possível superar os pontos fracos do trabalho e aumentaria a gama de possibilidade de aplicações.

Outro ponto importante para possibilitar o desenvolvimento de aplicação mais complexas e significativas é o desenvolvimento de um conjunto significativo de funções reconfiguráveis. A criação de ampla biblioteca de funções reconfiguráveis expandiria os horizontes de aplicações para ambiente R-TEV.

## *Referências*

- ABOUZEID, P.; BABBA, P.; PAULET, M. C. D.; SAUCIER, G. Input-driven partitioning methods and application to synthesis on table-lookup-based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 12, n. 7, p. 913–925, Julho 1993.
- ALEXANDER, M. J.; ROBINS, G. New performance driven FPGA routing algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 15, n. 12, p. 1505–1517, Junho 1996.
- ALTERA. *Altera Corporation* <http://www.altera.com>. 2006.
- ALTERA. *DSP Builder User Guide*. Versão 6.0. [S.l.], Abril 2006.
- BABAUD, J.; WITKIN, A. P.; BAUDIN, M.; DUDA, R. O. Unique of the gaussian kernel for scale-space filtering. *IEEE Transaction Pattern Anal. Machine Intelligence*, p. 23–33, 1986.
- BENNETT, J. K. *Shared Memory Multiprocessing Using the Virtex II PPC: Sharing Memory, Sharing a UART, and Synchronization A Guide for the XUP Development Board*. University of Colorado at Boulder, Novembro 2005.
- BROWN, S. D.; FRANCIS, R. J.; ROSE, J.; VRANESIC, Z. G. *Field-Programmable Gate Arrays*. Boston: Kluwer Academic Publishers, 1992.
- BROWN, S. D.; ROSE, J.; VRANESIC, Z. G. A detailed router for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design*, v. 11, n. 5, p. 620–628, Maio 1992.
- CALLAHAN, T.; WAWRZNEK, J. Instruction-level parallelism for reconfigurable computing. *Lecture Notes in Computer Science*, 1998.
- CANTIN, M. A.; SAVARIA, Y.; LAVOIE, P. An automatic word length determination method. *Proc. IEEE International Symposium on Circuits and Systems*, p. v55–v56, 2001.
- CELOXICA. *Handel-C Language Reference Manual for DK 3.0*. [S.l.]: Celoxica, 2004. (RM-1003-4.2).
- CHAN, P. K.; SCHLAG, M. D. F. Acceleration of an FPGA router. *5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, p. 175–181, Abril 1997.
- CHANG, S. C.; MAREK-SADOWSKA, M.; HWANG, T. T. Technology mapping for TLU FPGA's based on decomposition of binary decision diagrams. *IEEE Transactions on CAD of Integrated Circuits and Systems*, v. 15, n. 10, p. 1226–1248, Outubro 1996.
- CMAR, R.; RIJNDERS, L.; SCHAUMONT, P.; VERNALDE, S.; BOLSEN, L. A methodology and design environment for DSP ASIC fixed point refinement. *Proc. Design, Automation and Test in Europe*, 1999.

- COMPTON, K.; HAUCK, S. An introduction to reconfigurable computing. *IEEE Computer - Invited Paper*, April 2000.
- COMPTON, K.; HAUCK, S. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, v. 34, n. 2, p. 171–210, Junho 2002.
- CONG, J.; WU, C. An efficient algorithm for performance-optimal FPGA technology mapping with retiming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 17, n. 9, p. 738–748, Setembro 1998.
- CONG, J.; WU, C.; DING, Y. Cut ranking and pruning enabling a general and efficient FPGA mapping solution. *ACM/SIGDA International Symposium on FPGAs*, p. 29–35, 1999.
- CONSTANTINIDES, G. A.; CHEUNG, P. Y. K.; LUK, W. The multiple wordlength paradigm. *Proc. IEEE Symposium on Field Programmable Custom Computing Machines*, 2001.
- CONSTANTINIDES, G. A.; CHEUNG, P. Y. K.; LUK, W. Optimum wordlength allocation. *Proc. IEEE Symposium on Field Programmable Custom Computing Machines*, v. 10, p. 219–228, 2002.
- CONSTANTINIDES, G. A.; CHEUNG, P. Y. K.; LUK, W. Synthesis of saturation arithmetic architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, v. 8, p. 334–354, 2003.
- CONSTANTINIDES, G. A.; WOEGINGER, G. J. The complexity of multiple wordlength assignment. *Applied Mathematics Letters*, v. 15, p. 137–140, 2002.
- COUTINHO, J. G. F.; LUK, W. Source-directed transformations for hardware compilation. *Proceedings IEEE International Conference on Field-Programmable Technology*, p. 278–285, Dezembro 2003.
- DAMIANOU, N.; DULAY, N.; LUPU, E.; SLOMAN, M. The ponder policy specification language. *Lecture Notes in Computer Science*, p. 18–38, 1995.
- DEHON, A. *Reconfigurable Architecture for General Purpose Computing*. Tese (Doutorado) — Massachusetts Institute of Technology, 1996.
- DEHON, A. The density advantage of configurable computing. *IEEE Computer*, v. 33, n. 4, 2000.
- DEHON, A.; WAWRZYNEK, J. Reconfigurable computing: What why, and design automation requirements. *Proceedings of the 1999 Design Automation Conference*, p. 610–615, 1999.
- EONIC. *Eonic Atlas System user guide: ATLAS2-HS V1.1*. [S.l.], 2001.
- EONIC. *TEV - Teaching Environment for Vistoso*. 2001.
- EONIC. *Internet site address: <http://www.eonic.com>*. 2004. Acessado em 16/09/2004.
- GEHRING, D. The trianus system and its application to custom computing. lecture notes in computer science. *1142-Field Programmable Logic: Smart Applications, New Paradigms and Compilers*, p. 176–184, 1996. Berlin, Alemanha.

- GOKHALE, M.; STONE, J. Napa c: compiling for a hybrid RISC/FPGA architecture. *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.
- GOKHALE, M. B.; STONE, J. M.; ARNOLD, J.; KALINOWSKI, M. Stream-oriented fpga computing in the streams-c high level language. *IEEE International Symposium on FPGAs for Custom Computing Machines (FCCM 2000)*, p. 49–56, Abril 2000.
- GUPTA, S.; DUTT, N. D.; GUPTA, R. K.; NICOLAU, A. Spark : A high-level synthesis framework for applying parallelizing compiler transformations. *Proceedings of the International Conference on VLSI Design*, Janeiro 2003.
- HARTENTEIN, R. A decade of research on reconfigurable architectures - a visionary retrospective. *Proceedings of the International Conference on Design Automation and Testing in Europe.*, 2001. Exhibit and Congress Center, Munich, Germany.
- HAUCK, S. The roles of FPGAs in reprogrammable systems. *Proceedings of the IEEE*, v. 86, p. 615–638, 1998.
- HAUCK, S.; AGARWAL, A. *Software technologies for reconfigurable systems*. Northwestern University, Dept. of ECE, Technical Report 1996.
- HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM*, v. 21, n. 8, p. 666–677, 1978.
- HWANG, J.; MILNE, B.; SHIRAZI, N.; D., S. J. System level tools for DSP in FPGAs. *Lecture Notes in Computer Science*, 2001.
- IDE, A. N. *Projeto de um Computador Reconfigurável com Controle Microprogramado: CORE-M*. Dissertação (Mestrado) — Universidade Federal de São Carlos - UFSCar, Fevereiro 2003.
- INMOS, L. *occam2 reference manual*. [S.l.], 1988.
- JACKSON, P. A.; HUTCHINGS, B. L.; TRIPP, J. L. Simulation and synthesis of CSP-based interprocess communication. *IEEE International Symposium on FPGAs for Custom Computing Machines (FCCM 2000)*, p. 208–227, Abril 2003.
- KRUPNOVA, H.; RABEDAORO, C.; SAUCIER, G. Synthesis and floorplanning for large hierarchical FPGAs. *ACM/SIGDA International Symposium on FPGAs*, p. 105–111, 1997.
- KUM, K. L.; SUNG, W. Combined word-length optimization and high-level synthesis of digital processing systems. *IEEE Transactions on Computer-aided Design of Integrated Circuit and Systems*, v. 20, p. 921–930, 2001.
- LEE, T.; YUSUF, S.; LUK, W.; SLOMAN, M.; LUPU, E.; DULAY, N. Compiling policy descriptions into reconfigurable firewall processors. *Proc. IEEE Field-Programmable Custom Computing Machines*, p. 39– 48, 2003.
- LIANG, J.; TESSIER, R.; MENCER, O. Floating point unit generation and evaluation for FPGAs. *Proceedings of the 10th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

- LIN, X.; DAGLESS, E.; LU, A. Technology mapping of LUT based FPGAs for delay optimisation. *Lecture Notes in Computer Science 1304—Field-Programmable Logic and Applications*, p. 245–254, 1997. Berlin, Alemanha.
- MCCLLOUD, S. Catapult c synthesis-based design flow: speeding implementation and increasing flexibility. *White Paper on Mentor Graphics Corporation*, Outubro 2004.
- MENCER, O. PAM-Blox II: design and evaluation of C++ module generation for computing with FPGAs. *Proceedings of the 10th IEEE Symposium on Field-Programmable Custom Computing Machines*, Abril 2002.
- MENCER, O.; PEARCE, D. J.; HOWES, L. W.; LUK, W. Design space exploration with a stream compiler. *Proceedings IEEE International Conference on Field-Programmable Technology*, p. 270–277, Dezembro 2003.
- MESQUITA, D. G. *Contribuições para Reconfiguração Parcial, Remota e Dinâmica de FPGAs*. Dissertação (Mestrado), 2002.
- MIRSKY, E. A. *Coarse-Grain Reconfigurable Computing*. Tese (Doutorado) — Massachusetts Institute of Technology, 1996.
- MORÓN, C. E.; SAITO, J. H.; ABIB, S.; MUCHERONI, M. L.; FURUYA, N.; BATTAIOLA, A.; SAWANT, H. S.; ROSA, R. R.; CECATTO, J. R.; ALONSO, E. B. Parallel architecture using dsps. *Proc. 9th Brazilian Symp. on Computer Architecture and High Performance Computing, SBACPAD'97*, Brasil, p. 605–608, 1997. Brasil.
- MORÓN, C. E.; SAITO, J. H.; RIBEIRO, J. R. P.; SAWANT, H. S.; ROSA, R. R. A visual environment integrating design, implementation and debugging in parallel real-time systems. *12th Brazilian Symp. on Computer Architecture and High Performance Computing, SBAC-PAD*, Brasil, p. 313–319, Oct. 2000. Brasil.
- NAJJAR, W.; BÖHM, A. P. W.; DRAPER, B.; HAMMES, J.; RINKER, R.; BEVERIDGE, M.; CHAWATHE, M.; ROS, C. High-level language abstraction for reconfigurable computing. *IEEE Computer Society*, p. 63–69, August 2003.
- NAM, G. J.; SAKALLAH, K. A.; RUTENBAR, R. A. Satisfiability-based layout revisited: Detailed routing of complex FPGAs via search-based boolean SAT. *ACM/SIGDA International Symposium on FPGAs*, p. 167–175, 1999.
- NAYAK, A.; HALDAR, M.; CHOUDHARY, A.; BANERJEE. Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs. *Proc. Design, Automation, and Test in Europe*, p. 722–728, 2001.
- ONG, S.; KERKIZ, N.; SRIJANTO, B.; C., T.; LANGSTON, M.; NEWPORT, D.; BOULDIN, D. Automatic mapping of multiple applications to multiple adaptive computing systems. *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.
- PAGE, I.; LUK, W. Compiling ACCAM into FPGAs. *FPGAs. International Workshop on Field Programmable Logic and Applications*, p. 271–283, Setembro 1991.
- PREWITT, J. M. S. Object enhancement and extration. *Pictured Processing Psychopictorics*, 1970. Academic Press, New York.

- RIBEIRO, J. R. P.; SILVA, N. C.; MORÓN, C. E. A visual environment for the development of parallel real-time programs. *Lecture Notes in Computer Science*, p. 994–1014, 1998.
- RIBEIRO, J. R. P.; SILVA, N. C.; MORÓN, C. E.; MORÓN, G. R. Transição entre projeto e implementação de sistemas paralelos de tempo-real usando o gerador de programas paralelos. *XII Simpósio Brasileiro de Engenharia de Software*, 1998. Maringá, PR, Brasil.
- SANKAR, Y.; ROSE, J. Trading quality for compile time: Ultra-fast placement for FPGAs. *ACM/SIGDA International Symposium on FPGAs*, p. 157–166, 1999.
- SENOUCI, S. A.; AMOURA, A.; KRUPNOVA, H.; SAUCIER, G. Timing driven floorplanning on programmable hierarchical targets. *ACM/SIGDA International Symposium on FPGAs*, p. 85–92, 1998.
- SIMULINK. <http://www.mathworks.com/products/simulink/>.
- STEPHENSON, M.; BABB, J.; AMARASINGHE, S. Bitwidth analysis with application to silicon compilation. *Proc. SIGPLAN Programming Language Design and Implementation*, 2000.
- SWARTZ, J. S.; BETZ, V.; ROSE, J. A fast routability-driven router for FPGAs. *ACM/SIGDA International Symposium on FPGAs*, p. 140–149, 1998.
- THAKUR, S.; CHANG, Y. W.; WONG, D. F.; MUTHUKRISHNAN, S. Algorithms for an FPGA switch module routing problem with application to global routing. *IEEE Transactions on CAD of Integrated Circuits and Systems*, v. 16, n. 1, p. 32–46, Janeiro 1997.
- THOMAS, D.; LUK, W. A framework for development and distribution of hardware acceleration. *Proc. International Society for Optical Engineering*, v. 4867, p. 48–59, 2002.
- TODMAN, T. J.; CONSTANTINIDES, G. A.; WILTON, S. J. E.; MENCER, O.; LUK, W.; CHEUNG, P. Y. K. Reconfigurable computing: architectures and design methods. *IEEE Proceedings on Computer Digital Technologies*, v. 152, n. 2, p. 193–207, Março 2005.
- TOGAWA, N.; YANAGISAWA, M.; OHTSUKI, T. Maple-opt: A performance-oriented simultaneous technology mapping, placement, and global routing algorithm for FPGAs. *Transactions on CAD of Integrated Circuits and Systems*, v. 17, n. 9, p. 803–818, Setembro 1998.
- WADEKAR, S. A.; PARKER, A. C. Accuracy sensitive word-length selection for algorithm optimization. *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, p. 54–61, 1998.
- WEINHARDT, M.; LUK, W. Pipeline vectorization. *IEEE Transaction on Computer-Aided Design of Integrated Circuit and Systems*, v. 20, n. 2, p. 234–248, Fevereiro 2001.
- WILSON, R. P.; FRENCH, R. S.; WILSON, C. S.; AMARASINGHE, S. P.; ANDERSON, J. M.; TJIAND, S. W. K.; LIAO, S. W.; TSENG, C. W.; HALL, M. W.; LAM, M. S.; HENNESSY, J. L. Suif: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, v. 29, n. 12, p. 31–37, Dezembro 1994.
- WIND. *River Systems - VIRTUOSO User guide for version 4.2*. 2001.



WITKIN, A. P. Scale-space filtering. *Intelnational Joint Conference on Artificial Intelligence*, p. 1019–1021, 1983.

WOOD, R. G.; RUTENBAR, R. A. FPGA routing and routability estimation via boolean satisfiability. *ACM/SIGDA International Symposium on FPGAs*, p. 119–125, 1997.

XILINX. *EDK OS and Libraries Document Collection*. [S.l.], Julho 2005.

XILINX. Microblaze - the low-cost and flexible processing solution. *Xilinx, Inc* - <http://www.xilinx.com>, 2005.

XILINX. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*. [S.l.], Outubro 2005.

XILINX. *Xilinx University Program Virtex-II Pro Development System - Hardware Reference Manual*. [S.l.], Março 2006.

XILINX. Edk concepts, tools, and techniques, version 9.1. *Xilinx, Inc* - <http://www.xilinx.com>, 2007.

YAMADA, A.; NISHIDA, K.; SAKURAI, R.; KAY, A.; NOMURA, T.; KAMBE, T. Hardware synthesis with the Bach system. *IEEE Proceedings on International Symposium on Circuits and Systems ISCAS'99*, p. VI–366 – VI 369, Maio 1999.

# *APÊNDICE A – Tutorial VHDL*

## **A.1 INTRODUÇÃO**

VHDL é uma linguagem para descrever sistemas digitais utilizada universalmente.

### **A.1.1 Origem**

VHDL é proveniente de VHSIC Hardware Description Language, Linguagem para descrição de hardware, no contexto do programa americano "Very High Speed Integrated Circuits" (VHSIC), iniciado em 1980.

### **A.1.2 Vantagens**

Dentre as vantagens da linguagem VHDL citamos:

- a. facilidade de atualização dos projetos;
- b. diferentes alternativas de implementação, permitindo vários níveis de abstração;
- c. verificação do comportamento do sistema digital, através de simulação;
- d. redução do tempo e custo do projeto; e
- e. eliminação de erros de baixo nível do projeto.

### **A.1.3 Desvantagens**

Dentre as desvantagens da linguagem VHDL citamos:

- a. falta de otimização no hardware gerado; e
- b. necessidade de treinamento para lidar com a linguagem.

## A.1.4 Características

A linguagem VHDL permite particionar o sistema em diferentes níveis de abstração, quais sejam: nível de sistema, nível de transferência entre registradores (RT level), nível lógico e nível de circuito.

Permite, também, três diferentes domínios de descrição: comportamental, estrutural e físico.

Considerando-se os diferentes níveis de abstração e os diferentes domínios de descrição, temos:

## A.1.5 Níveis de Abstração

### Nível de sistema:

descrição comportamental: algoritmos

descrição estrutural: processadores e memórias

descrição física: boards e chips

### Nível RT:

descrição comportamental: transferências entre registradores

descrição estrutural: registradores, unidades funcionais e multiplexadores

descrição física: chips e módulos

### Nível Lógico:

descrição comportamental: equações booleanas

descrição estrutural: gates e flip-flop

descrição física: módulos e células

### Nível de Circuito:

descrição comportamental: funções de transferência

descrição estrutural: transistores e conexões

descrição física: células e segmentos do circuito

## A.2 COMENTÁRIOS E NOTAÇÕES NA LINGUAGEM VHDL

Os comentários em VHDL ocorrem após dois traços "--". Os caracteres maiúsculos e minúsculos não tem distinção em VHDL. Os nomes de variáveis devem iniciar-se com letras alfabéticas, sendo possível utilizar também dígitos numéricos e "\_". O caracter "\_" não pode ser usado duplicado, e nem no final de um nome.

## A.3 ESTRUTURA DE UM PROGRAMA VHDL

A Figura 32 mostra a estrutura básica de um programa em VHDL, composta de três elementos: a) library, b) entity, e c) architecture.

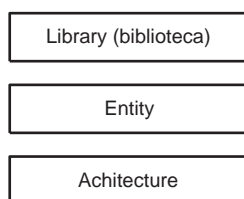


Figura 32: Estrutura de um programa em VHDL

## A.4 LIBRARY

A primeira informação contida num programa VHDL é a declaração da(as) *library(ies)* usada no projeto. Várias funções e tipos básicos são armazenados em bibliotecas. A biblioteca "IEEE" é sempre incluída.

Exemplo:

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Use IEEE.std_logic_unsigned.all;
```

Observações:

- a.a declaração Library IEEE é usada para definir a biblioteca IEEE;
- b.a declaração use IEEE.std\_logic\_1164.all é necessária para usar os dados correspondentes a lógica padrão da biblioteca; e
- c.a declaração use IEEE.std\_logic\_unsigned.all é necessária para realizar a aritmética não sinalizada.

## A.5 ENTITY

O ENTITY define a interface (port) do projeto, através dos pinos de entrada (in) e saída (out) e o tipo do sinal correspondente, no seguinte formato:

```
Entity nome_da_entity is
  port (
    Declaração dos pinos
  );
```

```
end [nome_da_entity];
```

Exemplo:

```
Entity COMPARA is
  port ( A,B: in std_logic;
        C: out std_logic);
end COMPARA;
```



Figura 33: Resultado da definição da interface através do exemplo de entity

## A.6 ARCHITECTURE

A architecture define a lógica do circuito e pode ser composta dos seguintes elementos (Figura 34 ): a) component, b) signal e c) lógica, sendo component e signal declarações de componentes e sinais intermediários opcionais.

O formato para a descrição da arquitetura é a seguinte:

```
Architecture nome_da_architecture of nome_da_entity is
  Declarações opcionais (component e signal)
begin
end [nome_da_architecture];
```

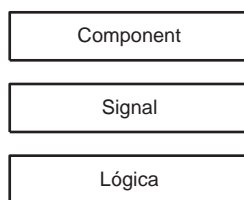


Figura 34: Elementos da descrição de architecture

## A.7 DESCRIÇÃO DO COMPONENTE

A declaração do componente é feita como segue, sendo que o componente declarado deve ser projetado através de um outro programa VHDL, ou outra forma de projeto.

```

Component nome_do_componente
port (
  Clk : in std_logic;
  Rst : in std_logic;
  Din : in std_logic;
  Dout : out std_logic
);
end component;

```

## A.8 DESCRIÇÃO DO SINAL

Podem ser declarados em entity, architecture ou em package, e servem para a comunicação entre os módulos.

sintaxe: `signal identificador (es) : tipo [restrição] [:=expressão];`

Exemplo:

```

signal cont : integer range 50 downto 1;
signal ground : bit := '0';
signal bus : bit\_vector;

```

## A.9 DESCRIÇÃO DA LÓGICA

### a) Descrição Comportamental

Uma das formas de descrever a lógica é em termos do seu comportamento, usando o comando process, cujo formato é o seguinte:

```

Process ( lista de sensibilidade ) begin
  descrição lógica
end process;

```

A lista de sensibilidade corresponde aos sinais que devem alterar a saída do circuito, e é composta de todos os sinais de entrada para os circuitos combinatórios. Para os registradores assíncronos, a lista seria composta do clock e do reset; e para os registradores síncronos, do clock.

Exemplo 1:

```

Architecture COMPORTAMENTO of COMPARA is
begin
  process (A,B)
  begin
    if(A=B) then C<='1';
    else C<='0';
    end if;
  end process;
end COMPORTAMENTO;

```

O exemplo acima descreve a lógica da arquitetura através do *process*. O comando *process* (A,B) indica que os sinais A e B formam a lista de sensibilidade. Pela descrição, a saída C será igual a 1 caso as entradas A e B sejam iguais, e C será igual a 0, caso contrário.

#### b) Descrição Estrutural

Para a descrição estrutural é feita a associação dos pinos do componente com os sinais usados no projeto, como no exemplo (Ex2) seguinte, onde U0 é um label.

Exemplo 2:

```
U0: nome_do_componente
  port map (
    Clk => clk_top;
    Rst => rst_top;
    Din => din_top;
    Dout => dou_top
  );
```

Vejamos o exemplo (Ex3), onde dois componentes XOR\_Gate e NOT\_Gate são interligados.

Exemplo 3:

```
architecture ESTRUTURA of COMPARA is
  component XOR_Gate
    port (I0, I1: in std_logic; O: out std_logic);
  end component;
  component NOT_Gate
    port (I0: in std_logic; O: out std_logic);
  end component;
  signal AUX: std_logic;
begin
  U0: XOR_Gate port map (I0=>A, I1=>B, O=>AUX);
  U1: NOT_Gate port map (I0=>AUX, O=>C);
end ESTRUTURA;
```

No exemplo Ex3, são usadas as descrições dos componentes XOR\_Gate e NOT\_Gate, Figura 35, e do sinal AUX.

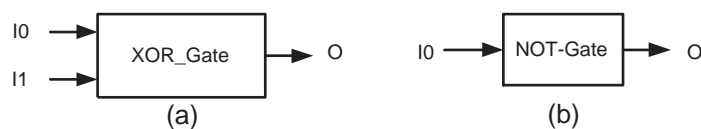


Figura 35: Componentes a) XOR\_Gate e b) NOT\_Gate

Os comandos rotulados U0 e U1 fazem a associação (*port map*) dos sinais dos componentes usados com os pinos definidos na *entity*, como mostra a Figura 36.

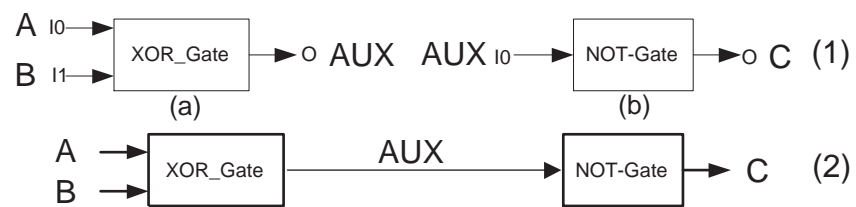


Figura 36: Resultado do projeto usando a descrição estrutural: (1) associação dos sinais (port map) sobre os componentes usados; (2) interconexão dos componentes através dos sinais associados



## APÊNDICE B – Socket do Virtuoso

```

1 K_CHANNEL CONEXAO; //channel para conexao
2 #define BUFFSIZE 0x80 //tamanho do buffer para envio
3 #define BUFFSIZER 0x100 //tamanho do buffer para recebimento
4
5 int NrOfBytesWritten; //Quantidade de dados enviados
6 int NrOfBytesRead; //Quantidade de dados enviados
7 int SendCnt; //Controle de envio de dados
8 int RecCnt; //Controle de recebimento de dados
9 int status; //Status da conexao
10 int tam_total; //tamanho total dos dados
11 int btRead; //quantidade de bytes lidos do arquivo
12 int bufSC; //tamanho do buffer menos os valores de controle
13 int fim_recebe; //Controle de fim de recebimento
14
15 unsigned char SendBuf[BUFFSIZE]; //Buffer para envio
16 unsigned char RecBuf[BUFFSIZER]; //Buffer para recebimento
17 char *BufTmp; //Ponteiro para manipulacao do arquivo para envio
18 char *BufTmpR; //Ponteiro para manipulacao do arquivo recebido
19
20 FILE *fp; //Descritor de arquivo para envio
21
22 vector<string> Referencia; //vetor para variaveis por referencia
23 int iRef = 0; //contador para variaveis passadas por referencia
24 Referencia.push_back("img"); //exemplo de variave img por referencia
25 char *Retorno; //ponteiro para retorno de dados
26
27 //conexao com o socket no próprio ATLAS
28 status = HS_SocketChannelOpen("127.0.0.1", 2303, &CONEXAO);
29 if(status==1) PRINT("Falha_na_conexao");
30 else PRINT("ãConexão_realizada_com_sucesso!");
31
32 //abertura de arquivo para transferencia
33 fp = fopen("Ninfeias2.bmp", "r+b");
34 if(fp == NULL) PRINT("—_Erro_ao_carregar_arquivo\n");
35 else PRINT("—_Abriu_Arquivo\n");
36
37 //Obtencao do tamanho do arquivo
38 fseek(fp, 0, SEEK_END);
39 tam_total = ftell(fp);
40 rewind(fp);
41
42 //aloca memoria para o arquivo
43 BufTmp = (char*) malloc(BUFFSIZE);
44 memset(BufTmp, 0, BUFFSIZE);
45
46 bufSC = BUFFSIZE-4; //carrega a quantidade de dados a ler do arquivo
47 //laco para envio do arquivo
48 while(tam_total > 0) {
49 btRead = fread(BufTmp, 1, bufSC, fp); //Le arquivo
50 SendCnt = tam_total;
51 if(SendCnt >= bufSC) { //controle para fim do arquivo
52 tam_total -= btRead;
53 SendCnt = btRead;
54 SendBuf[0] = SendCnt; //carrega tamanho do pacote
55 KS_MemCpy(NODE1, &SendBuf[1], NODE1, BufTmp, SendCnt);

```

```

56     SendCnt+=4;
57                                     //envia pacote
58     HS_ChannelPutW (CONEXAO, &SendBuf, SendCnt, &NrOfBytesWritten, _ALL_OPT);
59     SendCnt = 0;
60 } else tam_total -= SendCnt;
61 }
62
63 //se houver um ultimo pacote menor que BUFFSIZE envia agora
64 if (SendCnt != 0) {
65     SendCnt = btRead;
66     SendBuf[0] = SendCnt;
67     SendCnt+=4;
68     KS_MemCpy(NODE1, &SendBuf[1], NODE1, BufTmp, SendCnt);
69     HS_ChannelPutW (CONEXAO, &SendBuf, SendCnt, &NrOfBytesWritten, _ALL_OPT);
70     SendCnt = 0;
71 }
72
73 //envia pacote de fim de transmissao
74 memset(SendBuf, 0x00, BUFFSIZE);
75 SendCnt = BUFFSIZE;
76 HS_ChannelPutW (CONEXAO, &SendBuf, SendCnt, &NrOfBytesWritten, _ALL_OPT);
77
78 fim_recebe = 0;
79 free (BufTmp);
80
81 //recebimento
82 while (fim_recebe !=1) {
83     HS_ChannelGetW (CONEXAO, &RecBuf, BUFFSIZER, &NrOfBytesRead, _ALL_OPT);
84     if (NrOfBytesRead <= 0) //se nao recebeu continua
85         continue;
86
87     if (RecBuf[0]!=0) { //pacote de dados
88         NrOfBytesRead -= 4; //guarda em ponteiro
89         KS_MemCpy(NODE1, BufTmpR, NODE1, &RecBuf[1], NrOfBytesRead);
90         BufTmpR += NrOfBytesRead;
91     } else {
92         if (NrOfBytesRead == 8) { //pacote com tamanho da variavel
93             RecCnt = RecBuf[1]; //aloca memória para o recebimento
94             BufTmpR = (char*) realloc (BufTmpR, RecCnt);
95         }
96         if (NrOfBytesRead == 10) { //pacote de fim de variavel
97             BufTmpR -= RecCnt; //se houver variavel por referencia
98             if (iRef<Referencia.size()) { //copia os dados
99                 if (Referencia[iRef].c_str()=="img") {
100                     KS_MemCpy(NODE1, &img, NODE1, BufTmpR, RecCnt);
101                     iRef++;
102                 }
103             }
104             RecCnt = 0;
105         }
106         if (NrOfBytesRead > 10) { //pacote de fim de recebimento
107             fim_recebe = 1;
108         }
109     }
110 }
111
112 //copia dados para ponteiro de retorno
113 KS_MemCpy(NODE1, &Retorno, NODE1, BufTmpR, RecCnt);
114 free (BufTmpR);
115
116 //fecha conexao
117 HS_HostChannelClose (CONEXAO);
118
119 //retorna dados
120 return (Retorno);

```

## APÊNDICE C – Socket do Intermediário

```

1 //-----
2 #include <vcl.h>
3 #include <winsock.h>
4 #include <stdio.h>
5
6 #pragma hdrstop
7 //-----
8 #pragma argsused
9
10 #define DEFAULT_BUFLen 512 //tamanho do buffer
11 #define DEFAULT_PORTV 2303 //porta de comunicacao Virtuoso
12 #define DEFAULT_PORTHR 2302 //porta de comunicacao HR
13
14 int main(int argc , char* argv[] ) {
15
16 ///////////////////////////////////////////////////////////////////
17 //Inicializacao do pacote TCP para recebimento do Virtuoso//
18 ///////////////////////////////////////////////////////////////////
19 WORD wVersionRequestedTCP = MAKEWORD(1,1);
20 WSADATA wsaDataTCP;
21 int nRetTCP;
22
23 // Inicializa o Winsock e checa versao
24 nRetTCP = WSASStartup(wVersionRequestedTCP , &wsaDataTCP);
25 if (wsaDataTCP.wVersion != wVersionRequestedTCP) {
26     return 0; //versao errada
27 }
28
29 // cria socket TCP/IP
30 SOCKET listenSocketTCP;
31 listenSocketTCP = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
32 if (listenSocketTCP == INVALID_SOCKET) {
33     return 0;
34 }
35
36 //preenche a estrutura de endereco
37 SOCKADDR_IN saServerTCP;
38 saServerTCP.sin_family = AF_INET;
39 saServerTCP.sin_addr.s_addr = INADDR_ANY; // Winsock gerencia enderecos
40 saServerTCP.sin_port = htons(DEFAULT_PORTV); // porta para comunicacao
41
42 //bind para o socket
43 nRetTCP = bind(listenSocketTCP , (LPSOCKADDR)&saServerTCP , sizeof(struct sockaddr));
44 if (nRetTCP == SOCKET_ERROR) {
45     closesocket(listenSocketTCP);
46     return 0;
47 }
48
49 char szBufTCP[DEFAULT_BUFLen];
50
51 //coloca o socket em modo de escuta
52 nRetTCP = listen(listenSocketTCP , SOMAXCONN);
53 if (nRetTCP == SOCKET_ERROR) {
54     closesocket(listenSocketTCP);
55     return 0;

```

```

56     }
57
58     //Aguarda chegar uma requisicao
59     SOCKET     remoteSocketTCP;
60     remoteSocketTCP = accept(listenSocketTCP , NULL, NULL);
61     if (remoteSocketTCP == INVALID_SOCKET) {
62         closesocket(listenSocketTCP);
63         return 0;
64     }
65
66     ////////////////////////////////////////////////////////////////////
67     //Inicializacao do pacote UDP para envio para placa //
68     ////////////////////////////////////////////////////////////////////
69     unsigned char szBufUDP[DEFAULT_BUFLEN];
70     int nRetUDP; // Variave p/ tamanho pacote UDP
71     WORD wVersionRequestedUDP = MAKEWORD(1,1);
72     WSADATA wsaDataUDP;
73
74     // Inicializa WinSock e verifica versao
75     nRetUDP = WSASStartup(wVersionRequestedUDP, &wsaDataUDP);
76     if (wsaDataUDP.wVersion != wVersionRequestedUDP) {
77         return 0;
78     }
79     // Criar um UDP/IP datagrama socket //Socket UDP
80     SOCKET theSocketUDP;
81     theSocketUDP = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
82     if (theSocketUDP == INVALID_SOCKET) {
83         return 0;
84     }
85     //Estrutura de endereco do servidor
86     SOCKADDR_IN saServerUDP;
87     saServerUDP.sin_family = AF_INET;
88     saServerUDP.sin_addr.s_addr = inet_addr("200.18.98.152"); // IP do servidor
89     saServerUDP.sin_port = htons(DEFAULT_PORTHR); // Porta no servidor
90
91     int nFromLenUDP;
92     nFromLenUDP = sizeof(struct sockaddr);
93     int AckUDP;
94
95     ////////////////////////////////////////////////////////////////////
96     // Troca de dados do Virtuoso para o HR //
97     ////////////////////////////////////////////////////////////////////
98     do{
99         memset(szBufTCP, 0, sizeof(szBufTCP)); // zera buffer de recebimento
100         // aguarda recebimento TCP
101         nRetTCP = recv(remoteSocketTCP, szBufTCP, sizeof(szBufTCP), 0);
102         if (nRetTCP == INVALID_SOCKET) {
103             closesocket(listenSocketTCP);
104             closesocket(remoteSocketTCP);
105             return 0;
106         }
107         total += nRetTCP - 4;
108         // envia dados recebidos via UDP para placa
109         sendto(theSocketUDP, szBufTCP, nRetTCP, 0, (LPSOCKADDR)&saServerUDP, sizeof(struct
            sockaddr));
110     } while(szBufTCP[0] != 0); // ate enviar o pacote de fim de transmissao
111
112     char *bufferPTCP; // buffer para recebimento UDP
113     bufferPTCP = (char*) malloc(8); // aloca memoria para tamanho de variavel
114     int size = 0; // zera tamanho dos dados
115     int sizep = 0; // zera tamanho do ponteiro
116     int tamVet[10000]; // vetor para os tamanhos dos pacotes
117     int i = 0; // contador para o vetor
118
119     ////////////////////////////////////////////////////////////////////
120     // Troca de dados do HR para o Virtuoso //
121     ////////////////////////////////////////////////////////////////////
122     do{
123         memset(szBufUDP, 0, sizeof(szBufUDP)); // zera buffer de recebimento
124         // receba pacote UDP
125         nRetUDP = recvfrom(theSocketUDP, szBufUDP, sizeof(szBufUDP), 0, (LPSOCKADDR)&saServerUDP,
            &nFromLenUDP);
126

```

```
127     if (nRetUDP <= 0)                                     // se nao recebeu coninua
128         continue ;
129
130     size += nRetUDP;                                       // atualiza tamanho do pacote recebido
131     bufferPTCP = (char*) realloc(bufferPTCP, size);       // realoca memoria para o buffer
132     memcpy((bufferPTCP+sizep), szBufUDP, nRetUDP);
133     sizep += nRetUDP;                                       // atualiza tamanho do ponteiro
134     tamVet[i] = nRetUDP;                                    // guarda tamanho do pacote
135     i++;
136 } while((szBufUDP[0]!= 0) || (nRetUDP == 8) || (nRetUDP == 10));
137
138 int j = 0;
139 do{
140                                     //retransmite dados via TCP p/ Virtuoso
141     nRetTCP = send(remoteSocketTCP, bufferPTCP, tamVet[j], 0);
142     bufferPTCP += tamVet[j];
143     j++;
144     Sleep(5);                                       // para evitar perda de pacotes pelo Virtuoso
145 } while( j < i );
146
147
148 // finaliza os sockets existente
149 closesocket (remoteSocketTCP);
150 closesocket (listenSocketTCP);
151 closesocket (theSocketUDP);
152
153 // Termina WinSock
154 WSACleanup();
155
156 return 0;
157
158 }
```

## APÊNDICE D – Socket do Hardware Reconfigurável

```

1  #include "xparameters.h"
2  #include "xutil.h"
3  #include "lock_unlock.h"
4  /***** Include Files *****/
5  #include <net/xilnet_config.h>
6  #include <net/xilsock.h>
7  #include <xemac_l.h>
8
9  #define SERVER_PORT 2302
10 #define BUFSIZE 0x100
11 #define DDR_Flag 0x00000000 //endereço 0 – flag de controle de recebimento e processamento
12 //0 – recebendo; 1 – fim recebe; 2 – processando; 3 – fim processa
13 #define DDR_EnderecoInicial 0x00000001 //end 1 – endereço inicial dos dados
14 #define DDR_EnderecoFinal 0x00000009 //end 9 – endereço final dos dados
15 #define DDR_QuantVariaveis 0x00000011 //end 17 – quantidade de variaveis
16 #define DDR_InicioDadosRecb 0x00000013 //end 19 – inicio dados recebidos
17
18 unsigned char buffer_recebe[BUFSIZE + 10];
19 unsigned char buffer_envia[BUFSIZE + 10];
20
21 int main() {
22     volatile char *flag = (char*)DDR_Flag; //flag para controle
23     int SendCnt; //controle de envio
24     volatile struct sockaddr_in addr; //estrutura de endereço do socket
25     char bufT[4]; //buffer para ajuste de variavel
26
27     volatile char *buffer_entrada = (char*)DDR_InicioDadosRecb; //buffer de entrada
28     volatile Xuint8 *quant_var = (Xuint8*)DDR_QuantVariaveis; //ponteiro para quantidade de var
29     volatile unsigned char *inicio_dados, *fim_dados; //ponteiros para inicio e fim de dados
30     volatile int *end_envia_inicio, *end_envia_fim; //pronteiros aux para inicio e fim de dados
31     volatile int *tam_var; //ponteiro para tamanho de variavel
32
33     // Inicializa a rede
34     init_net();
35
36     // Create the socket
37     s = init_socket(&addr);
38     if (s == -1) {
39         exit(1);
40     }
41
42     int bytes_total = 0;
43     *flag = 0;
44
45     for (;;) {
46         int alen = 0;
47         // Aguarda novas conexoes
48         alen = sizeof(struct sockaddr);
49
50         int bytes = 0; //zera quantidades de bytes recebida
51         bzero(buffer_recebe, BUFSIZE); //limpa o buffer de recebimento
52         alen = BUFSIZE; //tamanho do buffer de recebimento
53         //recebe
54         bytes = xilsock_recvfrom(s, buffer_recebe, BUFSIZE, (struct sockaddr*)&addr, &alen);
55

```

```

56  if (bytes < 0)                                //se nao recebeu nada continua
57      continue ;
58
59  if (buffer_recebe[0] != 0) {                    //se existe dados no pacote recebido
60      bytes -= 4;                                //despreza o primeiro byte transmitido , tam. pacote
61      memcpy(buffer_entrada , &buffer_recebe[4], bytes); //copias dados para o buffer
62      buffer_entrada += bytes;                   //atualiza apontador dos dados de entrada
63      bytes_total += bytes;                       //atualiza contagem de bytes recebidos
64  } else {                                        //ultimo pacote recebido sem dados
65      *flag = 1;                                  //ajusta o flag
66
67      while(*flag != 3){                          //aguarda processamento
68
69          //Acerta ponteiros para os dados
70          end_envia_inicio = (char*) DDR_EnderecoInicial;
71          inicio_dados = *end_envia_inicio;
72          end_envia_fim = (char*) DDR_EnderecoFinal;
73          fim_dados = *end_envia_fim;
74          tam_var = *end_envia_fim;
75
76          bytes_total = 0;                         //zera contagem de bytes recebidos
77
78          while(*quant_var > 0) {                 //enquanto tem variaveis transmite
79              bzero(buffer_envia , BUFSIZE);     //limpa o buffer de recebimento
80              buffer_envia[0] = 0;               //zera primeiro valor do buffer
81              buffer_envia[1] = 0;
82              buffer_envia[2] = 0;
83              buffer_envia[3] = 0;
84              char *temp;
85              temp = tam_var;
86              memcpy(&buffer_envia[4], temp , 4);
87              bufT[0] = buffer_envia[7];         //inverte valor do tamanho da variavel
88              bufT[1] = buffer_envia[6];         //para corrigir diferenca de Endian
89              bufT[2] = buffer_envia[5];
90              bufT[3] = buffer_envia[4];
91              buffer_envia[4] = bufT[0];
92              buffer_envia[5] = bufT[1];
93              buffer_envia[6] = bufT[2];
94              buffer_envia[7] = bufT[3];
95              SendCnt = 8;                       //envia o tamanho da variavel
96                                                  //transmite
97              xilsock_sendto(s , buffer_envia , SendCnt , (struct sockaddr*)&addr , sizeof(addr));
98              SendCnt = *tam_var;                //verifica se ha dados > BUFSIZE na variavel
99              if (SendCnt > BUFSIZE - 4) {
100                  SendCnt = BUFSIZE - 1;
101                  *tam_var -= (SendCnt - 4);
102              }
103              while (SendCnt == (BUFSIZE - 1)) { //se houver entra no laco ate enviar todos
104                  buffer_envia[0] = SendCnt;
105                  buffer_envia[1] = 0;
106                  buffer_envia[2] = 0;
107                  buffer_envia[3] = 0;
108                  memcpy(&buffer_envia[4], inicio_dados , SendCnt-4);
109                  inicio_dados += (SendCnt - 4);
110                  xilsock_sendto(s , buffer_envia , SendCnt , (struct sockaddr*)&addr , sizeof(addr));
111                  bytes_total += (SendCnt - 4);
112                  SendCnt = *tam_var;
113                  if (SendCnt > (BUFSIZE - 4)) {
114                      SendCnt = BUFSIZE - 1;
115                      *tam_var -= (SendCnt - 4);;
116                  }
117              }
118              if (SendCnt != 0){                  //se sobrou dados < BUFSIZE envia o restante
119                  buffer_envia[0] = SendCnt + 4;
120                  buffer_envia[1] = 0;
121                  buffer_envia[2] = 0;
122                  buffer_envia[3] = 0;
123                  memcpy(&buffer_envia[4], inicio_dados , SendCnt);
124                  inicio_dados += SendCnt;
125                  xilsock_sendto(s , buffer_envia , SendCnt+4 , (struct sockaddr*)&addr , sizeof(addr));
126                  bytes_total += SendCnt;
127              }
128              buffer_envia[0] = 0;                //envia fim de variavel

```

```

129     buffer_envia[1] = 0;
130     buffer_envia[2] = 0;
131     buffer_envia[3] = 0;
132     memset(&buffer_envia[4], 1, 6);
133     SendCnt = 10;
134     xilsock_sendto(s, buffer_envia, SendCnt, (struct sockaddr*)&addr, sizeof(addr));
135     *quant_var = *quant_var - 1;
136     tam_var += 1;
137 }
138 memset(buffer_envia, 0, BUFSIZE); //envia fim de transmissao
139 SendCnt = BUFSIZE;
140 xilsock_sendto(s, buffer_envia, SendCnt, (struct sockaddr*)&addr, sizeof(addr));
141
142 sh_printf("_Dados_Enviados:_%d_\n\r", bytes_total);
143 bytes_total = 0;
144
145     *flag = 0; //ajusta flag
146     buffer_entrada = (char*) DDR_InicioDadosRecb;
147 }
148 }
149 xilsock_close(s);
150 return 0;
151 }
152
153
154 void init_net() {
155     int i;
156     unsigned char hw_addr[]="00:11:22:33:44:55 ";
157     Xuint8 ip[16]="200.18.98.152";
158
159     // Ajusta o endereço MAC do hardware no TCP/IP
160     xilnet_eth_init_hw_addr(hw_addr);
161
162     // Inicializa o hardware Ethernet
163     xilnet_eth_init_hw_addr_tbl();
164
165     // Ajusta o endereço IP
166     xilnet_ip_init(ip);
167
168     // Inicializa o endereço OPB MAC
169     xilnet_mac_init(XPAR_ETHERNET_MAC_BASEADDR);
170
171     // xemac_l.h
172     XEmac_mPhyReset(XPAR_ETHERNET_MAC_BASEADDR);
173
174     // Ajusta o MAC físico do Ethernet no Drive MAC
175     XEmac_mSetMacAddress(XPAR_ETHERNET_MAC_BASEADDR, mb_hw_addr);
176
177     // Enable MAC (xemac_l.h)
178     XEmac_mEnable(XPAR_ETHERNET_MAC_BASEADDR);
179 }
180
181
182 int init_socket(struct sockaddr_in *addr) {
183     // struct sockaddr_in addr;
184     int s;
185     int err;
186
187     // Cria um socket do tipo datagrama (UDP)
188     s = xilsock_socket(AF_INET, SOCK_DGRAM, 0);
189     if (s == -1) {
190         return -1;
191     }
192
193     // Define the address for the socket
194     addr->sin_family = AF_INET; // socket.h (IP)
195     addr->sin_addr.s_addr = INADDR_ANY;
196     addr->sin_port = SERVER_PORT;
197
198     // Bind do socket para o endereço especificado
199     err = xilsock_bind(s, (struct sockaddr *)addr, sizeof(struct sockaddr));
200     if (err == -1) {
201         print("bind()_error_\n\r");

```



```
202     return(-1);
203 }
204
205 // Socket escuchando
206 err = xilsock_listen(s, 2);
207 if (err == -1) {
208     print("listen() error\n\r");
209     return -1;
210 }
211
212 return s;
213 }
```

## APÊNDICE E – Código Fonte - Prewitt

```

1 #include "xparameters.h"
2 #include "xutil.h"
3
4 #define DDR_Flag 0x00000000 //endereco 0 - flag de controle de recebimento e processamento
5 //0 - recebendo; 1 - fim recebe; 2 - processando; 3 - fim processa
6 #define DDR_EnderecoInicial 0x00000001 //end 1 - endereco inicial dos dados
7 #define DDR_EnderecoFinal 0x00000009 //end 9 - endereco final dos dados
8 #define DDR_QuantVariaveis 0x00000011
9 #define DDR_InicioDadosRecb 0x00000013 //end 19 - inicio dados recebidos
10
11 int main() {
12     volatile int i;
13     volatile char *flag = (char*)DDR_Flag;
14
15     volatile char *inicio_dadosp = (char*) DDR_InicioDadosRecb;
16     volatile char *inicio_dados_proc;
17     volatile char *ln1_ptr, *ln2_ptr, *ln3_ptr;
18     volatile char *nova_imagem;
19     volatile int *inicio_nova_img = (int*) DDR_EnderecoInicial;
20     volatile int *fim_nova_img = (int*) DDR_EnderecoFinal;
21     volatile int *tam_varp;
22     volatile Xuint8 *quant_varp = (Xuint8*)DDR_QuantVariaveis;
23     int rows, cols;
24     int deslocamento;
25     int SUMH, SUMV;
26     char val[4];
27
28     for(;;) {
29         tam_varp = NULL;
30         while(*flag != 1){} //aguarda fim de recebimento
31         *flag = 2;
32
33         inicio_dados_proc = inicio_dadosp;
34         inicio_dadosp += 10; //pula para o incador de vamanho de cabecalho
35         val[3] = *inicio_dadosp;
36         inicio_dadosp++;
37         val[2] = *inicio_dadosp;
38         inicio_dadosp++;
39         val[1] = *inicio_dadosp;
40         inicio_dadosp++;
41         val[0] = *inicio_dadosp;
42         inicio_dadosp++;
43         memcpy(&deslocamento, val, 4); //valor de deslocamento p/ inicio de dados da imagem
44
45         inicio_dados_proc += deslocamento; //ptr de dados p/ processar aponta para dados da img
46         inicio_dadosp += 4; //pula valores nao importantes do cabecalho
47
48         val[3] = *inicio_dadosp;
49         inicio_dadosp++;
50         val[2] = *inicio_dadosp;
51         inicio_dadosp++;
52         val[1] = *inicio_dadosp;
53         inicio_dadosp++;
54         val[0] = *inicio_dadosp;
55         inicio_dadosp++;

```

```

56 mempcy(&cols, val, 4); //pega largura da imagem
57
58 val[3] = *inicio_dadosp;
59 inicio_dadosp++;
60 val[2] = *inicio_dadosp;
61 inicio_dadosp++;
62 val[1] = *inicio_dadosp;
63 inicio_dadosp++;
64 val[0] = *inicio_dadosp;
65 inicio_dadosp++;
66 mempcy(&rows, val, 4); //pega altura da imagem
67
68 ln1_ptr = inicio_dados_proc;
69 ln2_ptr = ln1_ptr + cols;
70 ln3_ptr = ln2_ptr + cols;
71
72 int rc = rows * cols; //calcula tamanho da imagem
73 float G;
74 int k;
75
76 *inicio_nova_img = (int) nova_imgem;
77 inicio_dadosp -= 26; //copia cabecalho .bmp para nova_imgem
78 mempcy(nova_imgem, inicio_dadosp, deslocamento);
79 nova_imgem += deslocamento;
80
81 *fim_nova_img = (int) nova_imgem + rc; //ajusta ponteiro de fim arquivo
82 tam_varp = (int*) *fim_nova_img;
83 *tam_varp = rc + deslocamento;
84
85 bzero(nova_imgem, cols+1);
86 nova_imgem += cols+1;
87 int fdl = 0;
88 int rcf;
89 rcf = rc - cols;
90 for (i = 0; i <= rcf; i++) { //laco para calculo
91     SUMH = SUMV = 0; //calculo da convolucao
92     SUMH = SUMH + *ln1_ptr;
93     SUMV = SUMV - *ln1_ptr;
94     SUMV = SUMV - *ln2_ptr;
95     SUMH = SUMH - *ln3_ptr;
96     SUMV = SUMV - *ln3_ptr;
97     ln1_ptr++;
98     ln2_ptr++;
99     ln3_ptr++;
100    SUMH = SUMH + *ln1_ptr;
101    SUMH = SUMH - *ln3_ptr;
102    ln1_ptr++;
103    ln2_ptr++;
104    ln3_ptr++;
105    SUMH = SUMH + *ln1_ptr;
106    SUMV = SUMV + *ln1_ptr;
107    SUMV = SUMV + *ln2_ptr;
108    SUMH = SUMH - *ln3_ptr;
109    SUMV = SUMV + *ln3_ptr;
110    G = sqrt(SUMH * SUMH + SUMV * SUMV);
111    if(G > 127) //threshold
112        *nova_imgem = 255;
113    else
114        *nova_imgem = 0;
115    nova_imgem++;
116    fdl++;
117    if((fdl + 2) != cols) { //atualizacao dos ponteiros
118        ln1_ptr--;
119        ln2_ptr--;
120        ln3_ptr--;
121    } else {
122        ln1_ptr++;
123        ln2_ptr++;
124        ln3_ptr++;
125        *nova_imgem = 0;
126        nova_imgem++;
127        *nova_imgem = 0;
128        nova_imgem++;

```

```
129     fdl = 0;
130     }
131 }
132 bzero(nova_imagem, cols - 1);
133 nova_imagem += cols - 1;
134 *tam_varp = rc + deslocamento;           //fim da imagem
135
136 *quant_varp = 1;
137 *flag = 3;
138 }
139 return 0;
140 }
```

## APÊNDICE F – Código Fonte - Filtro Gaussiano

```

1  #include "xparameters.h"
2  #include "xutil.h"
3
4  #define DDR_Flag 0x00000000 //endereço 0 – flag de controle de recebimento e processamento
5      //0 – recebendo; 1 – fim recebe; 2 – processando; 3 – fim processa
6  #define DDR_EnderecoInicial 0x00000001 //end 1 – endereço inicial dos dados
7  #define DDR_EnderecoFinal 0x00000009 //end 9 – endereço final dos dados
8  #define DDR_InicioDadosRecb 0x00000013 //end 19 – início dados recebidos
9
10 int main() {
11
12     volatile int i;
13     volatile char *flag = (char*)DDR_Flag;
14
15     volatile char *inicio_dadosp = (char*) DDR_InicioDadosRecb;
16     volatile char *inicio_dados_proc;
17     volatile char *ln1_ptr, *ln2_ptr, *ln3_ptr;
18     volatile char *nova_imagem;
19     volatile int *inicio_nova_img = (int*) DDR_EnderecoInicial;
20     volatile int *fim_nova_img = (int*) DDR_EnderecoFinal;
21     volatile int *tam_varp;
22     volatile Xuint8 *quant_varp = (Xuint8*)DDR_QuantVariaveis;
23     int rows, cols;
24     int deslocamento;
25     int NV;
26     char val[4];
27
28     for(;;) {
29         tam_varp = NULL;
30
31         while(*flag != 1){} //aguarda fim de recebimento
32         *flag = 2;
33
34         inicio_dados_proc = inicio_dadosp;
35         inicio_dadosp += 10; //pula para o incador de vamanho de cabecalho
36         val[3] = *inicio_dadosp;
37         inicio_dadosp++;
38         val[2] = *inicio_dadosp;
39         inicio_dadosp++;
40         val[1] = *inicio_dadosp;
41         inicio_dadosp++;
42         val[0] = *inicio_dadosp;
43         inicio_dadosp++;
44         memcpy(&deslocamento, val, 4); //valor de deslocamento p/ início de dados da imagem
45
46         inicio_dados_proc += deslocamento; //ptr de dados p/a processar aponta para dados da img
47         inicio_dadosp += 4; //pula valores nao importantes do cabecalho
48
49         val[3] = *inicio_dadosp;
50         inicio_dadosp++;
51         val[2] = *inicio_dadosp;
52         inicio_dadosp++;
53         val[1] = *inicio_dadosp;
54         inicio_dadosp++;
55         val[0] = *inicio_dadosp;

```

```

56     inicio_dadosp++;
57     memcpy(&cols, val, 4);           //pega largura da imagem
58
59     val[3] = *inicio_dadosp;
60     inicio_dadosp++;
61     val[2] = *inicio_dadosp;
62     inicio_dadosp++;
63     val[1] = *inicio_dadosp;
64     inicio_dadosp++;
65     val[0] = *inicio_dadosp;
66     inicio_dadosp++;
67     memcpy(&rows, val, 4);         //pega altura da imagem
68
69     ln1_ptr = inicio_dadosp_proc;
70     ln2_ptr = ln1_ptr + cols;
71     ln3_ptr = ln2_ptr + cols;
72
73     int rc = rows * cols;
74     float G;
75     int k;
76     *inicio_nova_img = (int) nova_imgem;
77
78     inicio_dadosp -= 26;           //copia cabecalho .bmp para nova_imgem
79     memcpy(nova_imgem, inicio_dadosp, deslocamento);
80     nova_imgem += deslocamento;
81
82     *fim_nova_img = (int) nova_imgem + rc;   //ajusta ponteiro de fim arquivo
83     tam_varp = (int*) *fim_nova_img;
84     *tam_varp = rc + deslocamento;
85
86     bzero(nova_imgem, cols+1);
87     nova_imgem += cols+1;
88     int fd1 = 0;
89     int rcf;
90     rcf = rc - cols;
91     for (i = 0; i <= rcf; i++) {           //laco para calculo
92         NV = 0;                           //calculo da convolucao
93         NV = *ln1_ptr/16 + *ln2_ptr/8 + *ln3_ptr/16;
94         ln1_ptr++;
95         ln2_ptr++;
96         ln3_ptr++;
97         NV += *ln1_ptr/8 + *ln2_ptr/4 + *ln3_ptr/8;
98         ln1_ptr++;
99         ln2_ptr++;
100        ln3_ptr++;
101        NV += *ln1_ptr/16 + *ln2_ptr/8 + *ln3_ptr/16;
102        *nova_imgem = NV;
103        if (*nova_imgem > 255)
104            *nova_imgem = 255;
105        if (*nova_imgem < 0 )
106            *nova_imgem = 0;
107        nova_imgem++;
108        fd1++;
109        if ((fd1 + 2) != cols) {
110            ln1_ptr--;
111            ln2_ptr--;
112            ln3_ptr--;
113        } else {
114            ln1_ptr++;
115            ln2_ptr++;
116            ln3_ptr++;
117            *nova_imgem = 0;
118            nova_imgem++;
119            *nova_imgem = 0;
120            nova_imgem++;
121            fd1 = 0;
122        }
123    }
124    bzero(nova_imgem, cols-1);
125    nova_imgem += cols - 1;
126    *tam_varp = rc + deslocamento;       //fim da imagem
127
128    *quant_varp = 1;

```

```
129     *flag = 3;
130 }
131 return 0;
132 }
```