

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**MÉTODOS DE DISTRIBUIÇÃO E CONEXÃO
DE *IP CORES* PARA DISPOSITIVOS
PROGRAMÁVEIS FPGA**

por

JOSÉ CARLOS SANT'ANNA PALMA

Dissertação de mestrado submetida como requisito parcial
à obtenção do grau de Mestre em Ciência da Computação.

Prof. Dr. Fernando Gehm Moraes
Orientador

PORTO ALEGRE

2001



Dados Internacionais de Catalogação na Publicação (CIP)

P171m Palma, José Carlos Sant'Anna
Métodos de distribuição e conexão de IP cores para dispositivos programáveis FPGA / José Carlos Sant'Anna Palma. – Porto Alegre, 2001.

110 f. : il.

Diss. (Mestrado) – Fac. de Informática, PUCRS, 2001.

1. Arquitetura de Computador. 2. Integração de Sistemas.
3. FPGA. 4. Hardware. 5. Memória de Computador I. Título.

CDD 004.22

004.5

**Ficha Catalográfica elaborada pelo
Setor de Processamento Técnico da BC-PUCRS**

RESUMO

Esta dissertação aborda a utilização de módulos de hardware pré-projetados e pré-verificados, *cores*, na implementação de sistemas computacionais integrados em um único circuito integrado – SoC (do inglês, *System-on-a-Chip*) – para dispositivos FPGAs. O estudo relativo ao estado-da-arte na utilização de *cores* apresenta: (i) características dos *cores*; (ii) integração de *cores* em SoCs; (iii) problemas relativos ao desenvolvimento e utilização de *cores*; (iv) ferramentas para parametrização de *cores*; e (v) distribuição de *cores* através da Internet. Duas ferramentas desenvolvidas são apresentadas. A primeira tem como objetivo sistematizar a organização das informações contidas nos projetos, de forma a facilitar sua distribuição através da Internet. A segunda ferramenta realiza a reconfiguração de *hard cores*, permitindo que o usuário tenha uma maior flexibilidade para adaptar *cores* já sintetizados a seus projetos. A distribuição de *cores* via Internet e a possibilidade de reconfiguração de dispositivos programáveis requer uma estrutura de comunicação que permita a estes *cores* trocarem informações. A maior contribuição deste trabalho é propor uma estrutura de barramento intra-FPGA para comunicação entre *cores*. Este barramento permite que *cores* possam ser inseridos, removidos ou substituídos sem afetar o funcionamento do resto do sistema em execução no FPGA. Desta forma, torna-se realidade o conceito de "hardware virtual", onde reside no dispositivo programável apenas o hardware que está sendo executado, sendo alocado sob a demanda de utilização, da mesma forma que hoje é utilizado o conceito de memória virtual em sistemas operacionais.

Palavras-chave: *cores*, distribuição de *cores*, prototipação, reconfiguração, reuso, FPGAs, SoC.

ABSTRACT

This work presents the use of pre-designed and pre-verified modules of hardware, cores, in the implementation of systems-on-a-chip – SoC – for FPGA programmable devices. The study concerning the state-of-art in this field presents: (i) core features; (ii) core integration; (iii) problems in the development and use of cores; (iv) tools for core customization; and (v) core distribution through the Internet. Two developed tools are presented. A core is not only a hardware description. Several design files, tools and documentation compose it. The goal of the first tool is to provide an environment to store design files, keeping them organized, and to distribute the core through the Internet. The second tool allows hard cores reconfiguration, giving higher flexibility to the designer, since the hardware can be modified without re-synthesis. Core distribution and reconfiguration of programmable devices requires a communication structure, to allow information exchange between cores. The most important contribution of this work is to propose such a structure, providing intra-FPGA communication between cores. The proposed structure allows core insertion, core removal and core substitution without stopping the fixed part of the system. This results in a “virtual hardware”, where the hardware is in the programmable device only when it is being executed, in the same way as virtual memory is used in operating systems.

Keywords: cores, core distribution, prototyping, reconfiguration, design reuse, FPGAs, SoC.

"O único lugar onde o sucesso vem antes
do trabalho é no dicionário"

Albert Einstein (1879 – 1955)

AGRADECIMENTOS

À CAPES e à PARKS, por terem viabilizado este trabalho através do suporte financeiro.

Ao PPGCC-FACIN, pelas instalações oferecidas para o desenvolvimento do trabalho.

Aos funcionários do PPGCC, Dario, Leandro e Viviane, pela disposição e amizade.

Ao corpo docente do PPGCC, pela amizade formada no decorrer do curso. Dentre estes, gostaria de citar os professores Ney Calazans, Avelino Zorzo, César De Rose, Paulo Fernandes e Lúcia Giraffa.

Aos colegas do grupo GAPH, os quais foram de grande importância na troca de conhecimento e no convívio diário. Um agradecimento especial aos amigos Aline Vieira e Leandro Möller.

Aos colegas e grandes amigos da turma 2000: Alexandre Zamberlam, Caroline Gasperin, Cristiano Sangoi, Daniel Mesquita, Daniela Ribas, Eder Mathias, Elisa Boff, Josyane Barros, Leandro Cassol, Lucio Duarte, Murilo Juchem, Rodrigo Goulart, Victor Sant'Anna e César Marcon. Amigos inseparáveis, com quem tive a felicidade de conviver durante estes dois anos de curso.

Aos colegas da turma 2001, amigos tão especiais quanto os já citados.

Ao meu orientador, Prof. Fernando Gehm Moraes, pelo companheirismo, amizade, e por estar sempre disponível quando precisei de auxílio.

À minha família, que mesmo estando distante, me incentivou e acreditou no meu sucesso, e à minha namorada, Aline Dantas, pela compreensão nos momentos de minha ausência.

À todos que, direta ou indiretamente, contribuíram para o desenvolvimento deste trabalho.

À Deus, pela minha família, pelos meus amigos, pela oportunidade de aumentar meus conhecimentos e por conhecer pessoas tão especiais.

SUMÁRIO

1	INTRODUÇÃO.....	1
2	MÓDULOS DE HARDWARE E SUA DISTRIBUIÇÃO.....	3
2.1	Definição de Cores.....	3
2.2	Sistemas Integrados em um Único Chip – SoCs.....	5
2.3	Fluxos de Projeto Usando Cores.....	6
2.3.1	Fluxo Tradicional.....	6
2.3.2	Fluxo de Projeto em Espiral.....	9
2.4	Desafios Para a Implementação de Cores.....	10
2.4.1	Integração de <i>Cores</i>	10
2.4.2	Linguagens para Descrição de Sistemas.....	17
2.4.3	Proteção da Propriedade Intelectual.....	20
2.4.4	Teste.....	25
2.5	Ferramentas para Personalização de Cores.....	26
2.5.1	Xilinx.....	26
2.5.2	Altera.....	29
2.6	Distribuição de cores na Internet.....	31
3	DISTRIBUIÇÃO DE IP CORES.....	33
3.1	Codificação VHDL para Distribuição de Cores.....	33
3.1.1	Práticas Básicas de Codificação.....	34
3.1.2	Codificação para Portabilidade.....	36
3.1.3	Codificação para Síntese.....	37
3.2	Documentação do Core.....	41
3.3	Verificação do Core.....	42
3.3.1	Plano de Verificação.....	42
3.3.2	Estratégia de Verificação.....	43
3.3.3	Projeto do <i>Testbench</i>	44
3.3.4	<i>Code Coverage</i>	47
3.4	Ambiente de Distribuição de Cores.....	50
3.4.1	Desenvolvimento da Ferramenta de Distribuição de <i>Cores</i>	50
3.4.2	Exemplo: Página da Arquitetura R6.....	54
4	RECONFIGURAÇÃO E DISTRIBUIÇÃO DE HARD CORES.....	57

4.1	Revisão da Arquitetura Virtex	58
4.1.1	CLBs e IOBs.....	59
4.1.2	Configuração do Dispositivo Virtex	60
4.2	Ferramenta Xilinx Floorplanner	61
4.2.1	Arquivos de Entrada	63
4.2.2	Arquivos de Saída.....	63
4.2.3	Fluxos de Projeto com <i>Floorplanner</i>	63
4.3	Ambiente de Reconfiguração de Hard Cores.....	66
4.3.1	Applet do Reconfigurador	68
4.3.2	Exemplo de Utilização do Reconfigurador.....	70
4.4	Reconfiguração Parcial.....	71
5	PROPOSTA DE INTERCONEXÃO ENTRE CORES EM DISPOSITIVOS VIRTEX	
	73	
5.1	Proposta de Interconexão	73
5.2	Definição do Controlador e Interfaces dos Cores	75
5.3	Validação da Proposta de Interface	79
5.4	Resultados da Simulação.....	81
5.5	Reconfiguração dos Cores Utilizando a Interface de Comunicação	82
6	CONCLUSÕES E TRABALHOS FUTUROS.....	87
7	BIBLIOGRAFIA.....	89

LISTA DE FIGURAS

Figura 1 – Objetivos do trabalho.	2
Figura 2 – Arquitetura SOC genérica.	5
Figura 3 – Fluxo de projeto tradicional.	7
Figura 4 – Fluxo de projeto para SoC.	9
Figura 5 – SoC que usa a arquitetura de barramento <i>CoreConnect</i>	11
Figura 6 – Arquitetura <i>WISHBONE</i>	12
Figura 7 – SoC, composto pelo processador Nios e pelo hardware de propósito geral.	14
Figura 8 – Fluxo de projeto com o processador Nios.	15
Figura 9 - Fluxo de projeto utilizando <i>firm core</i>	16
Figura 10 – Componente reutilizável básico.	18
Figura 11 – Projeto RTL com um componente virtual.	22
Figura 12 – Projeto em nível arquitetural com um <i>core</i> de microprocessador virtual.	23
Figura 13 – Proteção de IP de hardware/software.	24
Figura 14 – <i>CORE Generator</i> da Xilinx.	26
Figura 15 – Visão geral do ChipScope.	28
Figura 16 – Fluxo de projeto utilizando ChipScope.	29
Figura 17 – Fluxo de projeto usando o MegaCore.	30
Figura 18 – Funções <i>MegaCore</i> disponíveis no web site da <i>Altera</i>	30
Figura 19 – Latch inferido devido à ausência da condição <i>ELSE</i> na expressão <i>IF</i>	38
Figura 20 – Comportamento do projeto difere antes e depois da síntese.	38
Figura 21 – Descrição usando <i>IF-THEN-ELSE</i>	39
Figura 22 - Descrição usando <i>CASE</i>	39
Figura 23 – <i>Testbench</i> de módulo.	45
Figura 24 – Relatório da cobertura de instrução.	48
Figura 25 – Ferramenta de <i>code coverage</i> da <i>Aldec</i>	48
Figura 26 – Relatório da cobertura de caminho.	49
Figura 27 – Relatório da cobertura de expressão.	49
Figura 28 – Janela principal da ferramenta.	51
Figura 29 – Controle de versões.	51
Figura 30 – Janela de controle de arquivos.	52
Figura 31 – Permissões de arquivos.	52
Figura 32 – Sistema de controle de acessos.	53
Figura 33 – Diretório <i>public</i> do projeto <i>Processador_R6</i>	53
Figura 34 – Estrutura de um arquivo <i>.htaccess</i>	54
Figura 35 – Arquivo <i>.htpasswd</i>	54

Figura 36 – Página gerada pela ferramenta.....	55
Figura 37 – Arquitetura <i>Virtex</i>	58
Figura 38 – Estrutura do <i>bitstream</i> de configuração para o dispositivo XCV300.....	59
Figura 39 – Estrutura interna de uma CLB.....	59
Figura 40 – Estrutura interna do dispositivo <i>Virtex</i>	60
Figura 41 – Distribuição dos <i>frames</i> no circuito.....	61
Figura 42 – Endereços principais (MJAs) das colunas de CLBs do dispositivo XCV50.....	61
Figura 43 – Interface da ferramenta <i>Floorplanner</i>	62
Figura 44 – <i>Place and route</i> , depois <i>Floorplan</i>	64
Figura 45 – <i>Floorplan</i> anterior ao <i>place and route</i>	64
Figura 46 – <i>Floorplan</i> iterativo.....	65
Figura 47 - Criação de restrições a partir do posicionamento de IOBs.....	66
Figura 48 – Posições das LUTRAMs fixadas através do <i>Floorplanner</i>	67
Figura 49 - Fluxo de desenvolvimento de uma aplicação JBits.....	68
Figura 50 – Acesso às LUTs.....	69
Figura 51– Código HTML para geração da interface.....	70
Figura 52 – Exemplo de reconfigurador de <i>bitstream</i>	71
Figura 53 – Ferramenta para geração de <i>bistreams</i> parciais.....	72
Figura 54 – Barramento de comunicação de <i>cores</i>	74
Figura 55 – Duas camadas de <i>buffers</i>	75
Figura 56 – Conexões entre <i>buffers</i> e linhas horizontais dedicadas.....	75
Figura 57 – Integração de <i>cores</i> utilizando a proposta de barramento de comunicação.....	76
Figura 58 – Máquina-de-estados do árbitro do barramento.....	77
Figura 59 - Módulos Send e Receive.....	77
Figura 60 – Máquina de estados do módulo Send.....	78
Figura 61 – Máquina de estados do módulo Receive.....	78
Figura 62 – Invólucro para inserção de <i>cores</i>	79
Figura 63 – Formato da instrução.....	80
Figura 64 - Simulação da interface de comunicação.....	81
Figura 65 – <i>Bitstream</i> do controlador, com lógica extra.....	83
Figura 66 - Roteamento entre os buffers.....	84
Figura 67 – União dos <i>bistreams</i>	85

LISTA DE TABELAS

Tabela 1 - Três classificações de IP cores baseado em cinco critérios.....	4
Tabela 2 – Serviços de distribuição de cores na Internet.....	32
Tabela 3 – Codificação para operação e operandos.....	80

LISTA DE ABREVIATURAS

ASIC – *Application Specific Integrated Circuit*
CAD – *Computer Aided Design*
CPLD – *Configurable Programmable Logic Device*
CPU – *Central Processing Unit*
DCR – *Device Control Register*
DES – *Decryption/Encryption System*
DMA – *Direct Memory Access*
DSP – *Digital Signal Processor*
EDA – *Electronic Design Automation*
FPGA – *Field Programmable Gate Array*
FTP – *File Transfer Protocol*
HDL – *Hardware Description Language*
ICON – *Integrated Controller*
IEEE – *Institute of Electrical and Electronic Engineers*
ILA – *Integrated Logic Analyzer*
IP – *Intellectual Property (core)*
OPB – *On-Chip Peripheral Bus*
PCB – *Printed Circuit Board*
PCI – *Peripheral Component Interconnect*
PLB – *Processor Local Bus*
RMI – *Remote Method Invocation*
RTL – *Register Transfer Level*
SLDL – *System Level Design Language*
SoC – *System on a Chip*
TAC – *Technology Activity Commitee*
UART – *Universal Assynchronous Receiver Trasmitter*
USB – *Universal Serial Bus*
VHDL – *VHSIC Hardware Description Language*
VHSIC – *Very High Speed Integrated Circuit*
VLSI – *Very Large Scale of Integration*
VSIA – *Virtual Socket Interface Alliance*

1 Introdução

O avanço tecnológico na construção de sistemas computacionais é tal que permite hoje implementar microprocessadores com 40 milhões de transistores [SIA99]. O ritmo desses avanços da tecnologia de fabricação tem se mantido exponencial nas últimas décadas, conforme a *Lei de Moore* [SCH97]. Esta lei é devida a Gordon E. Moore, que em 1965 observou que a densidade de componentes em circuitos integrados dobrava a intervalos regulares, inferindo que este comportamento perduraria por muito tempo ainda. O intervalo medido por Moore para que a densidade média dos circuitos integrados dobrasse foi de 18 meses, o que ainda hoje permanece uma taxa estável [CAL98].

Porém, o mercado atual de semicondutores não é caracterizado apenas pela elevada complexidade dos sistemas, mas também pelo curto *time-to-market* (tempo até que um novo produto chegue ao mercado), alto desempenho e baixo consumo de potência.

É neste contexto que entram em cena os *cores*, módulos de hardware complexos pré-caracterizados, usados no desenvolvimento de sistemas computacionais integrados em um único circuito integrado (SoC – *System-on-a-chip*). Somente através do reuso destes componentes é possível atingir as exigências do mercado atual e reduzir a distância entre a quantidade de recursos disponíveis e a produtividade das equipes de projeto. Desta forma, o projetista pode concentrar-se no sistema completo sem ter que se preocupar com a funcionalidade interna e externa ou com o desempenho de componentes individuais. Conforme as estimativas da indústria de semicondutores, o percentual de reuso em CIs será de 90% em 2012 [SIA99]. Porém, a construção de SoCs através do reuso de *cores* ainda é um processo manual e bastante complicado, devido à ausência de ferramentas e padrões pré-estabelecidos pela indústria.

Projetar um *core* para ser reutilizado não é simplesmente isolar um módulo já pronto. Construir um *core* reutilizável leva de 2 a 2,5 vezes o esforço de construção de um projeto normal [HAA99]. Este esforço, porém, é recompensado quando o *core* é reutilizado, pois haverá uma redução do *time-to-market*, resultante da reutilização do *core*, e uma redução dos custos de desenvolvimento.

Além disso, para que um *core* seja reutilizado, ele deve ser codificado de maneira a facilitar o reuso, precisa ser bem documentado, e estar acompanhado de recursos de modelagem abstrata e teste (*testbenches*), para que o usuário possa validar o sistema que utiliza o *core*.

O objetivo principal deste trabalho é propor métodos para a distribuição e para a conexão de *cores* para dispositivos programáveis tipo FPGA. A idéia é permitir que usuários possam buscar *cores* na Internet e executá-los no dispositivo FPGA, comunicando-se uns com os outros, e que determinado *core* possa ser removido, quando seu uso não for mais necessário, para que outro seja inserido em seu lugar.

Para que esta idéia seja posta em prática, é necessário atender alguns requisitos. O

primeiro trata da distribuição de *cores* através da Internet. Existe a necessidade da existência de uma ferramenta que auxilie o projetista na organização e distribuição de seus projetos, mas que também proteja seus direitos autorais. Como segundo requisito, é necessário um ambiente que permita a reconfiguração parcial de FPGAs, para que seja possível a inserção e remoção de *hard cores* no sistema. Tal ambiente também deve ser usado para parametrizar os *hard cores*, de forma a adaptá-los ao projeto do usuário. O terceiro requisito diz respeito à uma interface de conexão e comunicação entre os *cores* em execução no FPGA que permita a inserção e remoção de *cores* sem afetar o funcionamento do resto do sistema.

A Figura 1 ilustra os requisitos da abordagem proposta, os quais devem convergir no objetivo principal do trabalho.

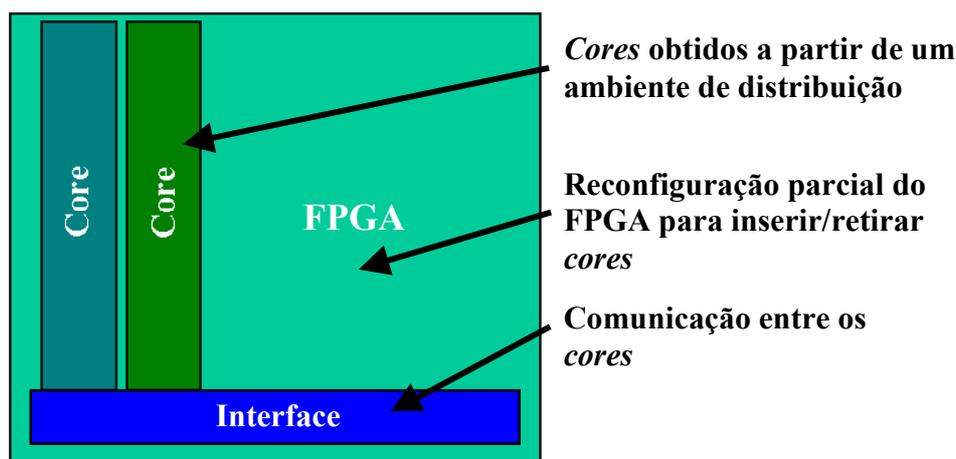


Figura 1 – Objetivos do trabalho.

Este volume está organizado da seguinte maneira. No Capítulo 2 é feita uma revisão das características de *cores*, dos sistemas integrados em um único circuito integrado e dos problemas encontrados na utilização de *cores*. Ao final, o Capítulo mostra ferramentas para parametrização e distribuição de *cores*.

O Capítulo 3 trata da codificação da descrição de *cores*, documentação, suporte à verificação e aspectos importantes que devem ser observados durante o desenvolvimento de um componente reutilizável. No final deste Capítulo é apresentada a primeira contribuição deste trabalho, uma ferramenta para organizar a distribuição de projetos através da Internet.

No Capítulo 4 é feito um estudo da arquitetura *Virtex* e da ferramenta de *Floorplanner* (editor de planta-baixa), requisitos necessários para que seja possível a reconfiguração de parte do dispositivo de forma a inserir ou retirar *cores* do sistema. Neste Capítulo também é apresentado o desenvolvimento de uma ferramenta para reconfiguração de *hard cores*.

O Capítulo 5 apresenta a proposta de um barramento de comunicação entre *cores* em funcionamento em um mesmo FPGA. A validação funcional e prototipação (parcial) deste barramento também são tratadas neste Capítulo.

Finalmente, o Capítulo 6 apresenta as conclusões deste trabalho e direções para trabalhos futuros.

2 Módulos de Hardware e sua Distribuição

Neste Capítulo, na Seção 2.1, é feita uma revisão do estado da arte em *cores* caracterizando-os (*soft*, *firm* e *hard*). A Seção 2.2 trata do projeto integrado de *cores* em um único circuito integrado (SoC), e a Seção 2.3 mostra o fluxo de projeto utilizando *cores*.

Após, na Seção 2.4, são identificados os desafios encontrados quando se implementam *cores*, tais como a integração dos *cores*, a escolha das linguagens para descrição de sistemas que serão utilizadas, a proteção da propriedade intelectual do autor e do usuário do *core* e os testes dos projetos de *cores*.

Finalmente, na Seção 2.5, são apresentadas algumas ferramentas para personalização de *cores* e, na Seção 2.6, são analisados alguns *web sites* de fornecedores de *cores* através da Internet, para que se possa avaliar a maneira como a distribuição de *cores* é feita.

2.1 Definição de Cores

Um *core* é um módulo de hardware complexo, digital ou analógico, podendo ser descrito em diferentes níveis de abstração. Estes *cores* são pré-projetados, pré-verificados (por simulação funcional e *back annotation*) e prototipados em hardware pelo menos uma vez. Os *cores* são usados para construir aplicações maiores e mais complexas em um dado circuito integrado [BER01]. Como exemplos de *cores* pode-se citar: memórias, processadores e barramento PCI.

Os principais desafios enfrentados por projetistas e usuários de *cores* são: (i) como usar *cores* projetados por terceiros com o menor esforço possível de projeto; (ii) como garantir a exatidão funcional do produto final; (iii) como garantir a testabilidade do sistema desenvolvido.

Dado o fato de que estes módulos “contêm” o conhecimento dos projetistas (que geralmente são proprietários dos *cores*), frequentemente estes componentes são submetidos à patentes e direitos de propriedade. Em outras palavras, um *core* representa uma propriedade intelectual disponibilizada pelo proprietário deste ao usuário do mesmo [GUP97].

Quanto à sua disponibilização, pode-se classificar um projeto de *core* como sendo licenciável ou proprietário. Um *core* licenciável é aquele em que o proprietário licencia um projeto e um conjunto de ferramentas conferindo a tarefa de projeto do sistema e sua fabricação ao usuário. Um *core* proprietário é aquele em que a preocupação principal do proprietário é proteger sua propriedade intelectual, resultando em baixo risco de propriedade intelectual, já que estes disponibilizam apenas a interface externa ao usuário.

Quanto à forma de disponibilização, os *cores* podem ser classificados em 3 categorias:

- *Hard cores*: são otimizados para uma tecnologia específica e não podem ser modificados

pelo projetista. Estes *cores* possuem um *layout* e planta baixa (*floorplanning*) pré-definidos. Possuem como maior vantagem a garantia dos tempos de propagação do circuito (*timing*). Sua desvantagem é a de não permitir qualquer tipo de modificação ou personalização.

- *Firm cores*: *netlist* (lista de conexões) gerado para a tecnologia empregada, que muda de fabricante para fabricante. Este tipo de *core* pode ser parcialmente parametrizado pelo projetista, permitindo que sua arquitetura seja adaptada à necessidade do projeto. Entretanto, como o *netlist* é específico para uma dada tecnologia, existe a dificuldade do uso de componentes fornecidos por fabricantes diferentes.
- *Soft cores*: descritos em linguagens de descrição de hardware (como *VHDL* ou *Verilog*), oferecendo flexibilidade e independência de tecnologia. O *core* pode ser modificado e empregado com tecnologias de diferentes fabricantes. Contudo, os *soft cores* apresentam como desvantagens: (i) a não garantia de se atingir os critérios temporais do circuito, uma vez que este ainda não foi sintetizado [RIN97]; (ii) a baixa proteção à propriedade intelectual, uma vez que o *core* é código aberto.

A Tabela 1 resume as características dos *cores*.

Tabela 1 - Três classificações de IP cores baseado em cinco critérios.

	Hard core	Firm core	Soft core
Rigidez	A organização é predefinida.	Combinação de código fonte e netlist dependente de tecnologia.	Apresenta um código fonte comportamental independente da tecnologia.
Modelagem	Modelado como um elemento de biblioteca.	Combinação de blocos sintetizáveis fixos. Permite compartilhar recursos com outros cores.	Sintetizável com diversas tecnologias.
Flexibilidade	Não pode ser modificado pelo projetista.	A personalização de funções específicas é dependente da tecnologia.	O projeto pode ser modificado e independe da tecnologia.
Previsibilidade	Temporização é garantida.	Caminhos críticos com temporizações fixas.	A temporização não é garantida, podendo não atender os requisitos do projeto.
Proteção da propriedade intelectual	Alta. A descrição normalmente corresponde a um <i>layout</i> .	Média	Muito pequena. Código fonte aberto.
Exemplo de Descrição	<ul style="list-style-type: none"> – Bitstream de FPGA – Formatos CIF ou GDS2 para layout de CI 	EDIF	VHDL, VERILOG

2.2 Sistemas Integrados em um Único Chip – SoCs

A Figura 2 [MAD97] ilustra uma arquitetura genérica de um SoC. Os *cores* são integrados através de uma rede de interconexão comercial ou adaptada, com um controlador e funções de interface com o meio externo. Caso os *cores* sejam obtidos de diferentes fontes a integração dos módulos e o teste do sistema podem ser difíceis, podendo até haver necessidade de que os *cores* sejam reprojitados, para adequá-los a um protocolo de interface comum.

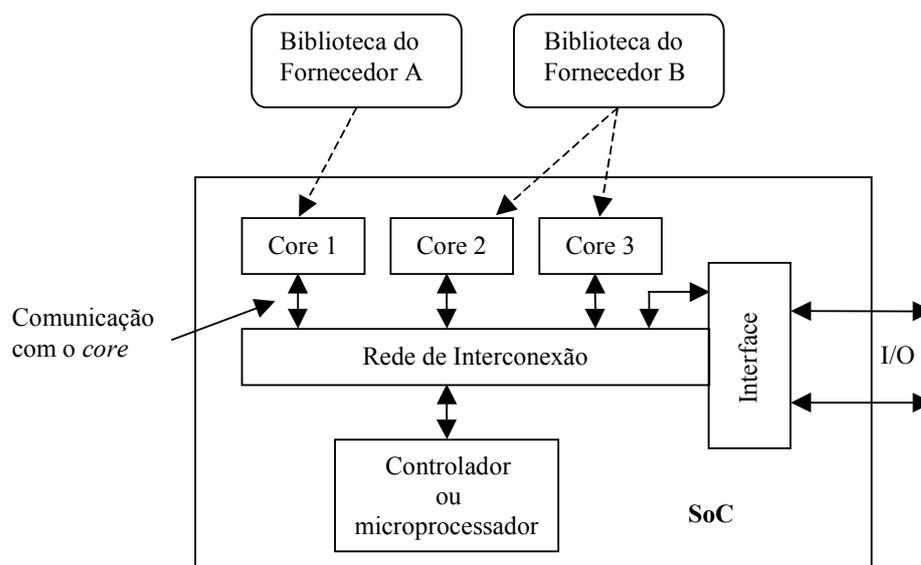


Figura 2 – Arquitetura SOC genérica.

Na prática, montar um SoC usando *cores* é ainda um processo passível de erro que exige trabalho intensivo e consome bastante tempo [MAD97]. O projeto de criar um SoC usando apenas *cores* ainda não se tornou realidade por várias razões [BER00a], entre as quais:

- Planejar o sistema é uma tarefa complexa, que requer que os projetistas respondam a questões como qual CPU deve ser usada, que funções devem ser feitas em hardware e/ou software, etc.
- A integração dos *cores* em um SoC é ainda um processo amplamente manual e passível de erro, exigindo que os projetistas entendam bem a funcionalidade, as interfaces e as características elétricas de *cores* complexos, como microprocessadores, controladores de memória, etc.
- Atender a todas as restrições de tempo é muito difícil, devido à complexidade do sistema. Em muitos casos, isto requer que os *cores* sejam adaptados, o que afeta sua reusabilidade.
- O projeto físico de sistemas complexos exige ferramentas de CAD de alto desempenho. Mesmo se a planta-baixa de cada *core* é pré-definida, colocá-los juntos com roteamento pode causar efeitos indesejados, tais como capacitância de acoplamento que degradam o desempenho.

- A verificação do sistema é um dos maiores gargalos do projeto. Mesmo se os *cores* são pré-verificados, não significa que todo o sistema funcionará quando eles forem colocados juntos. Verificação formal, no atual estado da arte, bem como técnicas de simulação de software não têm capacidade ou velocidade para manusear grandes sistemas em tempo hábil.
- A ausência de padrões de interface de *cores* pré-estabelecidos para a indústria e a ausência de ferramentas de síntese de interface eficientes tornam difícil que *cores* de diferentes fornecedores sejam integrados no mesmo SoC.
- A integração hardware-software é outro grande problema que afeta diretamente o *time-to-market* por que é, geralmente, feita mais tarde no processo de projeto, quando a parte de hardware está estável.

Depois que as principais decisões arquiteturais tenham sido tomadas, a primeira tarefa na construção de um SoC é a integração dos *cores* em um nível mais alto de hierarquia, que pode então ser simulado funcionalmente, sintetizado, ter sua planta-baixa definida e, desde cedo, usado no desenvolvimento do software do sistema. Esta tarefa de integração requer que o projetista entenda a funcionalidade de centenas de pinos em vários *cores* e determinar quais deles devem ser conectados juntos. Além do mais, *cores* são geralmente parametrizados e precisam ser configurados de acordo com o uso no SoC. O trabalho que vem sendo feito pela *VSI Alliance* [VSI00] tem como objetivo definir padrões para a integração de *cores*.

Os atuais sistemas de CAD não são orientados ao uso de *cores*, pois concentraram-se nas etapas de síntese lógica, síntese física e análise da temporização. Alguns ambientes de CAD, como Archimate [ARE01] ou Polis [BAL99] modelam os sistemas computacionais em níveis mais abstratos que linguagens de descrição de hardware, porém não são orientados ao uso de *cores*.

Ao nosso conhecimento, não existem ferramentas comerciais de software que possibilitem ao projetista construir um SoC, integrando e configurando *cores* facilmente. A complexidade dos SoCs atuais e a falta de ferramentas de alto nível apropriadas fazem do reuso uma técnica de implementação ainda distante da realidade dos projetistas.

2.3 Fluxos de Projeto Usando Cores

2.3.1 Fluxo Tradicional

Esta Seção apresenta o fluxo tradicional de projeto de sistemas baseado em *firm cores*. O fluxo de projeto consiste de três estágios: *captura*, *implementação* e *verificação*. Este fluxo de projeto é mostrado na Figura 3 [CAS97].

- **Captura do Projeto**

Durante a fase de *captura*, o projeto de aplicação do usuário é desenvolvido utilizando, preferencialmente, linguagens de descrição de hardware. É também descrito o

nível mais alto da hierarquia do projeto, o qual integra o *firm core* à aplicação do usuário. O *firm core* é tratado como uma “caixa preta”, não sendo um elemento sintetizável.

Freqüentemente, o projetista do *core* disponibiliza um modelo funcional que pode ser simulado com a descrição HDL que instancia o *core*, permitindo assim verificar a funcionalidade do sistema completo (esta etapa é realizada na *verificação do projeto*).

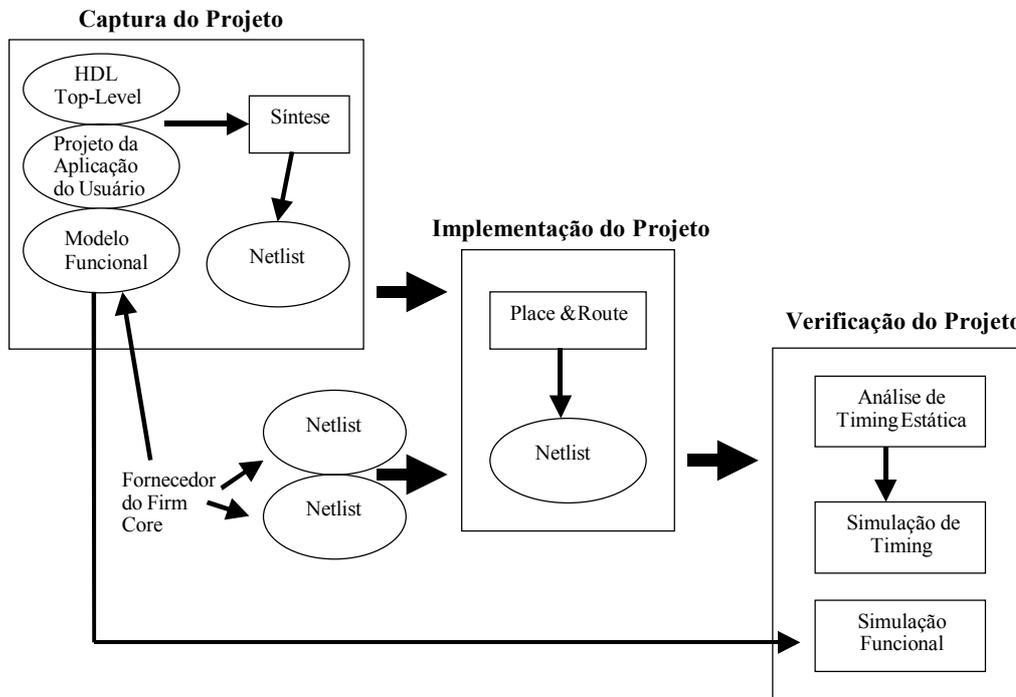


Figura 3 – Fluxo de projeto tradicional.

• Implementação do Projeto

O segundo estágio é o de *implementação*. Durante este estágio, tarefas de tradução e síntese (posicionamento e roteamento) são executadas. Como os *firm cores* são dependentes de uma dada tecnologia, eles são fornecidos juntamente com um conjunto de restrições de síntese contidas nos chamados *guide files*. Estes arquivos permitem que as ferramentas de síntese procurem atender aos requisitos temporais e de área, fixando a posição dos elementos do *core* e o roteamento dos caminhos críticos.

A integração do *core* com a aplicação do usuário é feita através de tarefas de tradução, unindo-se *netlists* oriundos da síntese do projeto do usuário e do(s) *core(s)* utilizado(s). A união do(s) *core(s)* com a aplicação do usuário pode fazer com que o sistema como um todo não atenda às restrições de área e atraso. Algumas técnicas são empregadas para garantir a temporização do circuito:

- Inserir restrições no projeto do usuário. Similar às restrições fornecidas pelo *firm core*, as restrições aplicadas ao projeto do usuário podem forçar uma determinada localização dos blocos lógicos ou impor limites para o atraso em determinados caminhos. O método recomendado para inserir restrições é realizar uma síntese preliminar do sistema sem qualquer restrição, utilizando apenas as restrições

fornecidas pelo *core*. A análise estática de temporização indicará a margem de atraso permitida ao projeto do usuário, possibilitando assim inserir-se as restrições necessárias.

- Realizar a planta baixa do sistema (*floorplaning*), posicionando manualmente blocos críticos próximos uns dos outros, minimizando desta forma atrasos causados pelo roteamento.
- Caso as técnicas acima, restrições de temporização e planta baixa não forem efetivas, deve-se modificar a aplicação do usuário. Um exemplo de modificação comum é a inserção de estágios de *pipeline*. Embora isto possa aumentar a latência do projeto, o *throughput* total aumentará.

O resultado da fase de *implementação* é um arquivo contendo a descrição física do sistema, ou seja, aplicação do usuário mais o(s) *core(s)* utilizado(s). Esta descrição física contém a disposição dos blocos lógicos, o roteamento entre eles e informações de elementos parasitas (capacitâncias de roteamento), necessários à etapa posterior, a *verificação* do projeto.

- **Verificação do Projeto**

O terceiro estágio do fluxo de projeto é a *verificação*. Este estágio consiste de duas etapas principais: *análise de timing* e *simulação funcional*. A verificação do projeto deve ser feita minuciosamente, e uma pesquisa ampla dos caminhos críticos pode indicar ao projetista onde ele pode aumentar o desempenho.

A etapa de *análise de timing* realiza a análise estática da temporização, determinando se o projeto atende ou não o desempenho desejado, através da análise dos caminhos críticos do circuito. Uma forma de guiar as ferramentas utilizadas por sistemas de CAD, como por exemplo o *Xilinx Foundation*, é inserir restrições de temporização para caminhos críticos, através dos parâmetros *TIMESPEC (timing specification)* [XIL01d]. Estas restrições são utilizadas pela síntese física, gerando-se mensagens de erro caso não seja possível sintetizar o sistema com as restrições impostas.

A *simulação funcional* verifica a temporização detalhada do sistema, podendo ser funcional ou detalhada. A *simulação funcional* é feita durante a etapa de captura do projeto, utilizando atraso zero (ou unitário). A *simulação funcional* detalhada utiliza dados provenientes da síntese física, como elementos parasitas de roteamento (*back annotation*).

Quaisquer conexões adicionais que não foram incluídas nas restrições de temporização também podem ser inspecionadas durante a análise estática, porém podem não atender necessariamente às especificações determinadas pelo projetista, uma vez que estas não são “vistas” pela síntese física. É importante restringir todos os caminhos críticos, a fim de permitir que a ferramenta de posicionamento/roteamento determine o lugar mais eficiente e mais otimizado para os componentes do projeto.

2.3.2 Fluxo de Projeto em Espiral

O modelo tradicional de fluxo de projeto para SoCs, mostrado na Figura 3, é frequentemente chamado de modelo cascata [KEA00]. Neste modelo, a transição de fase para fase nunca retorna às atividades da fase anterior. Assim, o projeto é passado de uma equipe responsável por uma fase, para outra responsável pela fase seguinte, sem que haja muita interação entre elas.

Para grandes projetos esta metodologia simplesmente não funciona. Grandes projetos têm um conteúdo de software suficiente para que haja a necessidade de se desenvolver o hardware e o software concorrentemente para assegurar a funcionalidade correta do sistema. Questões físicas devem ser consideradas no início do projeto para assegurar que o desempenho desejado seja também alcançado.

À medida que a complexidade aumenta, o modelo cascata está sendo substituído por um novo modelo, chamado de espiral. No modelo espiral, as equipes de projeto trabalham em múltiplos aspectos do projeto simultaneamente, refinando o projeto a cada etapa.

A Figura 4 mostra o fluxo de projeto em espiral.

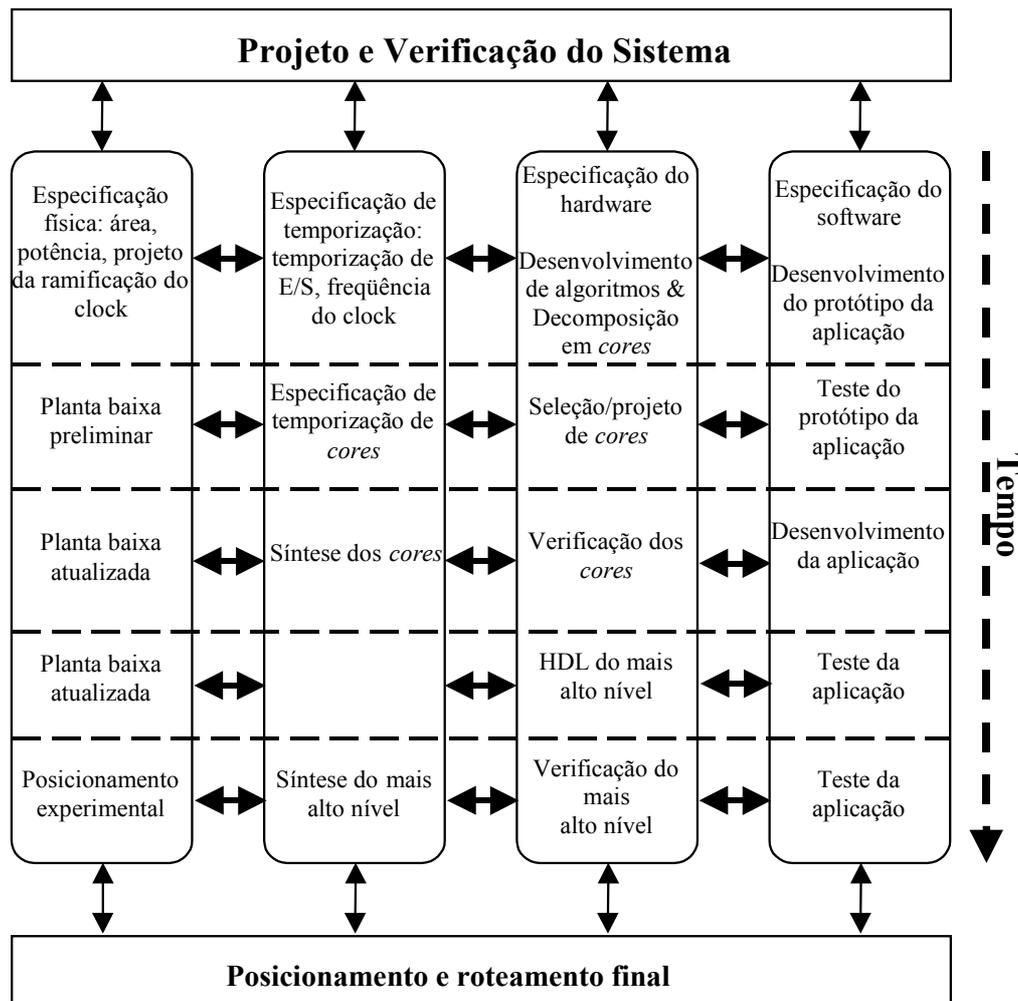


Figura 4 – Fluxo de projeto para SoC.

Este fluxo é caracterizado por:

- desenvolvimento de hardware e software concorrentemente;
- verificação e síntese dos módulos realizadas simultaneamente;
- planta baixa incluída no processo de síntese, atualizada a cada iteração;
- novos módulos são desenvolvidos somente se um *hard* ou *soft core* não está disponível para desempenhar uma tarefa;
- iterações acontecem por todo o fluxo.

Os projetistas desenvolvem simultaneamente as especificações do sistema, algoritmos para os sub-blocos críticos, verificação em nível de sistema, e as restrições de temporização para a integração final do chip. Isto significa que todos os aspectos do desenvolvimento de hardware e de software são trabalhados concorrentemente: funcionalidade, temporização, projeto físico e verificação.

2.4 Desafios Para a Implementação de Cores

Apesar das vantagens inerentes à utilização de *cores*, identificam-se quatro grandes problemas que devem ser resolvidos para que se possa construir facilmente um SoC, e que serão abordados nas Seções seguintes [PAL00]. São eles: (i) como integrar *cores*; (ii) quais linguagens para descrição de sistemas serão utilizadas; (iii) como proteger a propriedade intelectual do autor e do usuário do *core*; (iv) como testar os projetos baseados em *cores*.

2.4.1 Integração de Cores

Existem várias entidades trabalhando na padronização de *cores*, como por exemplo a IEEE e a VSIA.

O grupo IEEE P1500 [GUP97] tem por objetivos: (i) desenvolver uma linguagem de descrição de teste de *cores*, (ii) um mecanismo de controle de teste, e (iii) um mecanismo de acesso a dados do core. Estes três objetivos concentram-se na interface de teste entre um core e o restante do sistema ao seu redor. Não são padronizados os métodos de teste internos ao core e o teste de todo do sistema. A maioria dos usuários prefere usar as soluções de sua propriedade para estes testes.

A VSIA – Virtual Socket Interface Alliance [VSI00] – seleciona padrões abertos para a interface entre cores, tornando possível o projeto de SoCs. Quando um padrão proprietário é necessário, a VSIA tratará com o proprietário deste para disponibilizar seu uso sem pagamento de direitos autorais, como aconteceu por exemplo, com a linguagem Verilog. A VSIA denomina cores como componentes virtuais. O objetivo principal da padronização VSIA é permitir que projetistas sejam capazes de conectar cores a *sockets* (pinos) virtuais em circuitos complexos, como eles conectariam CIs discretos em placas de circuito impresso.

- Exemplos de metodologias de interconexão de *cores*

Exemplo 1: Barramentos Padrão

Nos estágios iniciais do projeto de um SoC, os *cores* são projetados com muitas e diferentes interfaces e protocolos de comunicação. Para contornar o problema de integração entre *cores*, foram criados padrões para estruturas internas ao circuito integrado. Atualmente, existem poucas arquiteturas de barramento publicamente disponíveis para guiar os fabricantes, tais como a *CoreConnect* [IBM99] da *IBM*, *AMBA* [ARM00] da *ARM* e *WISHBONE* [SIL01] da *Silicore*. Estas arquiteturas de barramento são geralmente vinculadas à arquitetura de um processador, tal como o *PowerPC* ou o *ARM* [BER00a].

A Figura 5 [BER00a] ilustra um SOC baseado na arquitetura de barramento *CoreConnect*.

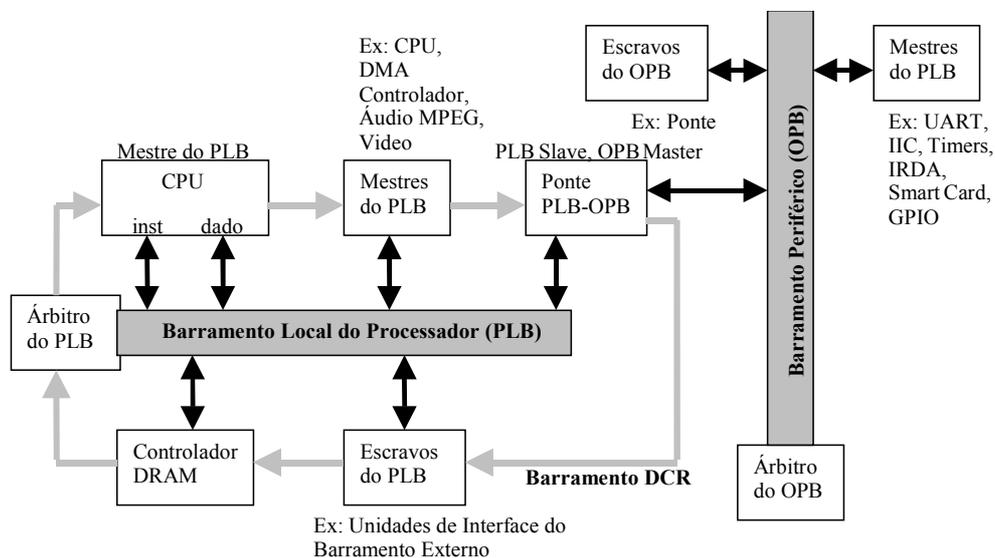


Figura 5 – SoC que usa a arquitetura de barramento *CoreConnect*.

A arquitetura *CoreConnect* da *IBM* fornece três barramentos para interconectar *cores* e lógica personalizável:

- Barramento Local do Processador (*Processor Local Bus - PLB*): usado para interconectar *cores* com alto desempenho, grande largura de banda, tais como o *PowerPC*, controladores DMA e interfaces de memória externa.
- Barramento Periférico (*On-Chip Peripheral Bus - OPB*): usado para interconectar periféricos que trabalham com baixas taxas de dados, tais como portas seriais, portas paralelas, UARTs (Universal Asynchronous Receiver Transmitter), e outros *cores* com pequena taxa de transferência de E/S.
- Barramento de Registradores de Controle de Dispositivos (*Device Control Register Bus - DCR*): caminho de baixa velocidade, usado para transmitir configuração e informações de estado entre o *core* processador e outros *cores*.

Se comparado com a arquitetura SoC genérica mostrada na Figura 2, o PLB seria equivalente ao barramento de interconexão, enquanto que o OPB é responsável pela interface I/O.

Outra arquitetura que deve ser citada é a *WISHBONE*. Esta especificação pode ser utilizada para *soft*, *firm* e *hard cores*, já que *firm* e *hard cores* são geralmente concebidos a partir de *soft cores*. A especificação não requer o uso de ferramentas de desenvolvimento ou dispositivos-alvo (hardware) específicos [SIL01].

Os desenvolvedores do *WISHBONE* foram influenciados por três fatores principais. Primeiro, havia a necessidade de uma solução boa e confiável para a integração de *cores* em SoCs. Segundo, havia a necessidade de uma especificação de interface comum para facilitar as metodologias de projeto estruturadas para grandes equipes de projeto. Terceiro, eles foram influenciados pelas soluções de integração de sistemas tradicionais, fornecidos por barramentos de microcomputador como o PCI, por exemplo.

De fato, a arquitetura do *WISHBONE* é análoga a um barramento de microcomputador, sendo que: (i) oferece uma solução flexível para integração que pode ser facilmente adaptada à uma aplicação específica; (ii) oferece uma variedade de ciclos de acesso ao barramento e de larguras de caminhos de dados para atender a diferentes sistemas; e (iii) permite que os *cores* sejam projetados por vários fornecedores.

Os projetistas do *WISHBONE* criaram uma especificação robusta o suficiente para assegurar a compatibilidade entre IP *cores* e, ao mesmo tempo, sem restringir a criatividade do projetista e do usuário final. Em Janeiro de 2001, a organização *OpenCores* adotou o *WISHBONE* como um padrão de conectividade entre seus *cores*. Ele foi escolhido por ser o único a atender os requisitos adotados por esta instituição. É um barramento flexível, simples, e o único completamente aberto atualmente, pois sua criadora, a empresa *Silicore Corporation*, tornou-o aberto ao domínio público [OPE01].

Ao contrário de arquiteturas como a *CoreConnect*, o *WISHBONE* tem uma estrutura simples, composta de um único barramento, como mostra a Figura 6. Um sistema com muitos componentes pode incluir duas interfaces *WISHBONE*: uma para blocos que exigem alto desempenho e outra para periféricos de baixo desempenho.

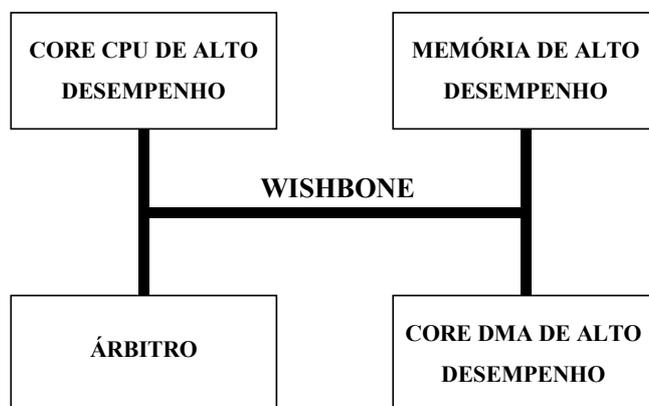


Figura 6 – Arquitetura *WISHBONE*.

Mesmo os *cores* tendo sido projetados para ter uma interface com um barramento padrão, o projetista deve executar as seguintes tarefas para a criação correta da descrição do SOC:

- Definir os *cores* necessários para implementar a funcionalidade desejada. Este processo é uma combinação de identificar *cores* pré-projetados para serem usados com ou sem modificação e identificar novos *cores* para serem projetados. Esta identificação tem favorecido a criação de “portais” de *cores*, como por exemplo o *Design and Reuse* [DES00].
- Verificar se os *cores* selecionados são compatíveis com a frequência de operação, largura de barramento, área requerida, etc.
- Entender a funcionalidade de todos os pinos de entrada/saída em todos os *cores* e determinar quais devem ser conectados juntos. Esta tarefa é sensivelmente simplificada pela adoção de um barramento padrão.
- Definir as prioridades de acesso ao barramento (política de arbitragem do barramento) e definir prioridades de requisição de interrupção do processador.
- Interconectar os pinos de acordo com suas prioridades, deixando espaço para mudanças de última hora (por exemplo, pode-se decidir que pinos de interrupção devem mudar sua prioridade, ou uma nova interrupção ser adicionada).
- Definir quais *cores* podem acessar a memória através de um controlador DMA e executar a tarefa de acordo com a prioridade do dispositivo requerente.
- Definir mapas de endereço para todos os *cores* e passar os valores como parâmetros para cada *core*, garantindo que não haja conflito de endereços entre *cores*.
- Definir quais são os clocks do sistema e conectar os clocks corretos a cada *core*, bem como a lógica de controle de clock apropriada.
- Definir as entradas e saídas do circuito integrado, incluindo nestes os pinos necessários para a metodologia de teste que será empregada.
- Inserir lógica de cola entre *cores*, caso necessário.

Estas tarefas são feitas manualmente nas metodologias e ferramentas atuais.

Exemplo 2: Projeto com Processador Nios

O Nios é um *firm core* de um processador RISC de propósito geral, configurável (de forma que possa atender às necessidades dos projetos), e que pode ser combinado com a lógica do usuário em um dispositivo lógico programável da Altera. A integração do processador Nios com o hardware de propósito geral no mesmo FPGA resulta em um SoC (*System-on-a-Chip*), como observado na Figura 7.

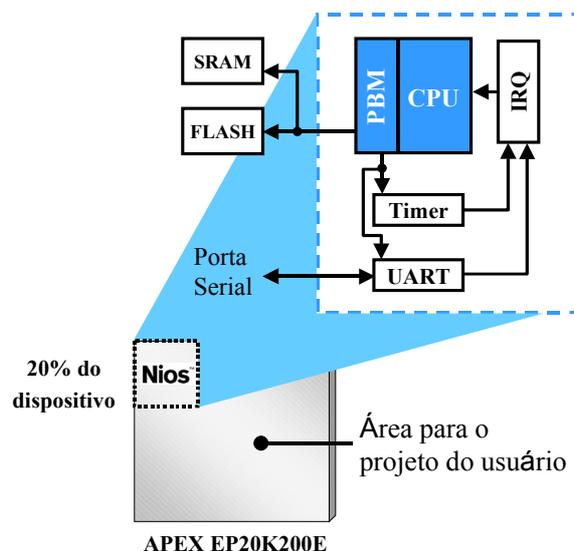


Figura 7 – SoC, composto pelo processador Nios e pelo hardware de propósito geral.

Algumas características do processador Nios são: (i) conjunto de instruções de 16 bits; (ii) largura de dados com 16 ou 32 bits; (iii) execução de uma instrução por ciclo de clock; (iv) suporte para memória *on-chip* e *off-chip*; (v) desempenho superior à 50 MIPS.

O Nios possui um conjunto de periféricos, tais como UARTs (*Universal Asynchronous Receiver/Transmitter*), PIOs (*Parallel Input/Output*), temporizadores e interfaces com memórias SRAM e FLASH. A comunicação com os periféricos é feita através do PBM (*Peripheral Bus Module*). O PBM é criado automaticamente pela ferramenta construtora do SoC, de acordo com os periféricos requisitados pelo usuário. O PBM inclui: (i) decodificador de endereços, o qual define o endereço de cada periférico e gera os sinais de “chip select”; (ii) multiplexador de entrada de dados, que fornece um barramento de conexão para transferência de dados entre cada periférico e o processador; (iii) controle de interrupções, que permite que o usuário determine uma interrupção de hardware para cada periférico; (iv) gerador de estado de espera, que determina o número de estados de espera necessários a cada periférico; (v) adaptador de barramento, módulo opcional que fornece interfaces com periféricos de 8, 16 e 32 bits.

O fluxo de projeto com o processador Nios é mostrado na Figura 8. Primeiro, o usuário configura o processador de acordo com as necessidades do projeto e seleciona os periféricos a serem utilizados. Após, de acordo com os periféricos selecionados, é configurado o barramento de comunicação PBM. Só então o processador Nios é gerado, juntamente com os periféricos. Na geração são criados, no lado do hardware, os arquivos HDL, *scripts* de síntese e *testbenches*; no lado do software, funções e drivers para os periféricos.

A configuração do processador e dos periféricos é feita conforme mostrado no Exemplo 1 desta mesma Seção, ou seja, seleciona-se as prioridades de interrupção, o mapa de memória, a largura do barramento de dados/endereço, etc.

A parte hardware é sintetizada com as ferramentas *Leonardo* e *Quartus*. Estas ferramentas têm por função gerar o projeto (hardware) do usuário, conectar o processador Nios e eventualmente conectar *cores* de terceiros. O resultado da síntese é um arquivo binário

(formato *SOF*) que é carregado no FPGA. A ferramenta *SignalTap* permite capturar sinais diretamente do dispositivo FPGA, atuando como um analisador lógico. A parte software é feita compilando-se a aplicação do usuário, juntamente com os *drivers* e bibliotecas do periféricos, através do uso do *GCC*.

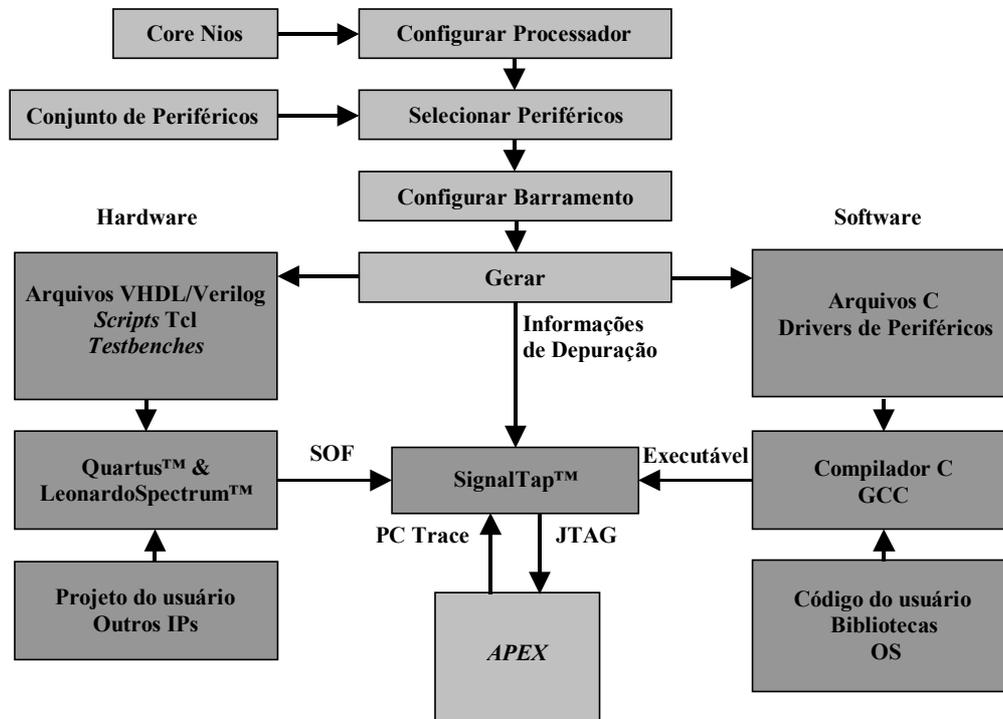


Figura 8 – Fluxo de projeto com o processador Nios.

Este fluxo de projeto já permite o desenvolvimento de aplicações completas em um único circuito integrado (SoC). Muitas das tarefas são feitas manualmente, exigindo um alto grau de conhecimento da arquitetura do processador e periféricos por parte do projetista. Falta, contudo, um sistema de validação que permita simular conjuntamente o processador, o barramento e lógica do usuário (*co-simulação*).

Exemplo 3: *Firm Core* PCI desenvolvido para FPGAs Xilinx

Como terceiro exemplo de integração de *cores* ilustra-se o fluxo de projeto para a utilização de *firm cores* da Xilinx. Dada à disponibilidade do *firm core* PCI, utiliza-se este como exemplo. A Figura 9 apresenta o fluxo de projeto de *hardware* e *software* da aplicação do usuário [CAP01].

O primeiro passo para implementar um sistema composto por partes *hardware* e *software* é a criação da descrição da parte *hardware*. Para isso há no sistema de desenvolvimento um projeto padrão, que é utilizado como modelo. Este projeto padrão contém a interface externa que a parte *hardware* do sistema deve possuir. Esta descrição, em linguagem VHDL, não deve ter a síntese física realizada. A única etapa realizada é a síntese lógica. O resultado desta síntese é um *netlist* em formato EDIF, que é posteriormente utilizado, juntamente com o *core* PCI para a síntese física.

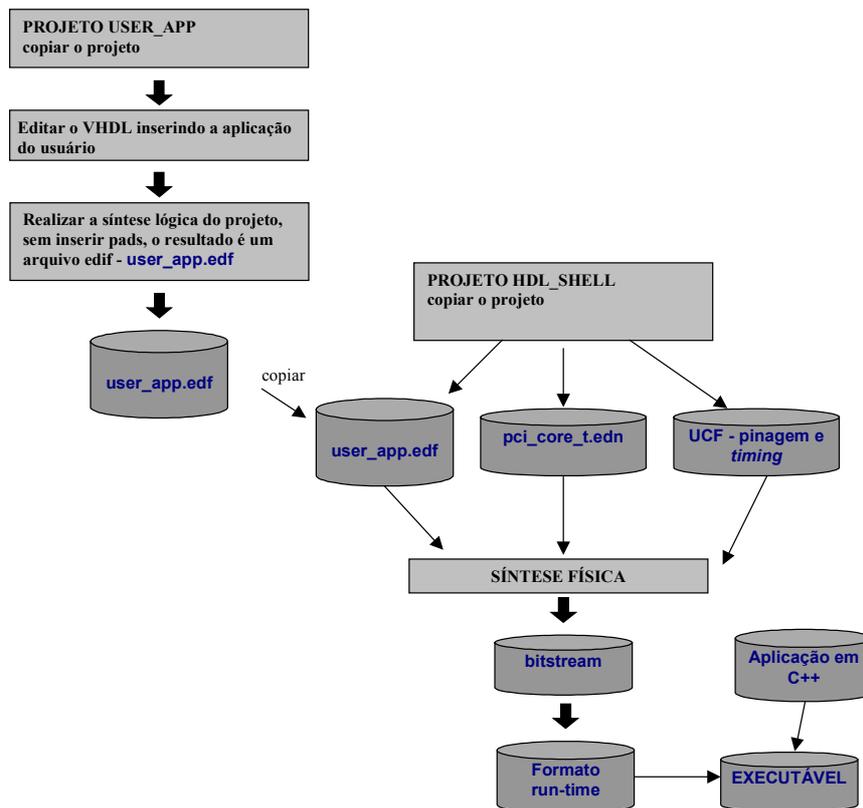


Figura 9 - Fluxo de projeto utilizando *firm core*.

Este fluxo de projeto não dispõe de um modelo funcional do *core* para simulação, o que dificulta a validação das aplicações desenvolvidas. Tipicamente, o usuário descreve sua aplicação sem preocupar-se com o *core* PCI, validando-a por simulação funcional. Uma vez sua aplicação desenvolvida, é feita a integração aos pinos do *core*.

Uma vez obtido o *netlist* EDIF, deve-se copiar este *netlist* para o local onde se encontra o *netlist* do *core*, o esquemático que une os *netlists* (usuário e *core*) e o arquivo contendo as restrições de temporização e posicionamento do *core* no FPGA. Observa-se que não há um ambiente integrado de desenvolvimento, sendo necessária a execução de diversas etapas manuais. Também não há a possibilidade de desenvolvimento puramente a partir de VHDL. Deve-se ter, no nível mais alto de hierarquia, um esquemático.

Uma vez obtido um diretório de projeto contendo os dois *netlists* e o arquivo com as restrições (*guide files*), procede-se à última etapa no fluxo da síntese de *hardware*, que é a síntese física. É importante comentar que o arquivo de restrições contém três partes principais, (i) localização dos pinos de entrada e saída na periferia do FPGA, (ii) restrições de temporização e (iii) posicionamento dos elementos do *core* no interior do FPGA. Este posicionamento é importante para que o circuito alcance o desempenho esperado, que é a frequência de operação em 33 MHz.

O arquivo resultante da síntese física, *bitstream*, é utilizado com a parte software da aplicação do usuário.

2.4.2 Linguagens para Descrição de Sistemas

O uso de sistemas compostos por parte hardware e parte software, com desenvolvimento concorrente (*codesign*), tem aumentado em uma variedade de domínios de projetos, devido ao fato de fornecerem alto desempenho e flexibilidade [HEL97]. Componentes de hardware fornecem um desempenho maior do que o que pode ser atingido pelo software para subsistemas com temporização crítica. O hardware também fornece interfaces para sensores e atuadores que interagem com o ambiente físico. Por outro lado, o software permite ao projetista especificar o sistema em altos níveis de abstração em um ambiente flexível onde erros, mesmo nos estágios finais do projeto, podem ser rapidamente corrigidos. O software também contribui para a diminuição do *time-to-market* e do custo do sistema.

Linguagens de especificação utilizadas por projetistas de sistemas computacionais podem ser divididas em linguagens de programação de software e linguagens de descrição de hardware. Linguagens de software como C ou C++ são tradicionalmente baseadas em um modelo de execução seqüencial derivado de semânticas de execução de processadores de propósito geral. Entretanto, linguagens de software geralmente não têm suporte para modelar concorrência e temporização. Estas deficiências podem ser superadas fornecendo-se ao projetista pacotes de bibliotecas que simulam as características ausentes. Linguagens de descrição de hardware, tais como Verilog [THO91] e VHDL [MAZ92] são concebidas para especificar hardware, tendo como características comuns hierarquia (descrição estrutural com utilização de componentes), concorrência entre processos (todos os módulos em hardware operam concorrentemente) e temporização.

É desejável para o processo de *codesign* usar uma única linguagem de especificação para a captura de projeto, pois especificações que usam linguagens diferentes para software e hardware combinam modelos de execução diferentes. Isto torna estas especificações difíceis de simular e analisar. Alguns projetistas utilizam uma linguagem de programação, geralmente C++, e estendem esta linguagem com construções para suportar características de hardware. Exemplos são as linguagens Scenic [LIA97], V++ [MCG97], SystemC [SYN01a] e SpecC [CEC99]. Descreve-se abaixo características de algumas linguagens utilizadas para modelar sistemas computacionais. As linguagens citadas são Java, VHDL, Objective VHDL, SpecC e Esterel.

- **Java**

O projetista modela o sistema completo em um algoritmo ou modelo comportamental. Uma vez que a especificação está completa, um processo de compilação automático é usado para analisar a especificação, identificando concorrência entre os processos. O particionamento e os passos de síntese usam a informação de concorrência obtida na etapa de compilação para criar um sistema de hardware-software otimizado. Como exemplo de sistemas que utilizam Java para modelar sistemas computacionais, citamos JavaCAD [DAL00a], que é um ambiente distribuído utilizado para simulação remota e proteção de

propriedade intelectual.

- **VHDL**

A linguagem VHDL [MAZ92] é utilizada para descrever a parte hardware dos sistemas computacionais, em diversos níveis de abstração: algorítmico (ou comportamental), transferência entre registradores, nível lógico com atraso unitário ou nível lógico com atraso detalhado. A linguagem permite explorar, em um nível mais alto de abstração, diferentes alternativas de implementação. O fato de a linguagem VHDL ser um padrão [IEE88], torna as ferramentas de CAD compatíveis entre si, permitindo também um maior reuso dos *soft cores*.

A síntese do VHDL para o nível físico é feita automaticamente por ferramentas de CAD, havendo pouca interação com usuário, tipicamente através da inserção de restrições de área e temporização. Isto evita a ocorrência de erros no nível físico, diminuindo o *time-to-market*.

- **Objective VHDL**

A idéia de reuso de IPs concentra-se na integração de componentes em um conjunto funcional, tão perto quanto possível da especificação. As tentativas de executar a reutilização no domínio de software conduzem à orientação a objetos e a paradigmas de componentes de software. A linguagem *Objective VHDL*, ou simplesmente OVHDL, permite definir e manipular objetos reutilizáveis, através da extensão da linguagem VHDL original. O módulo de hardware reutilizável é descrito em VHDL, enquanto seu invólucro é escrito em Objective VHDL, como mostra a Figura 10 [BAR99]. Dessa forma, a reutilização é possível para descrições já existentes, e ainda tira proveito das características da orientação à objetos, como herança, polimorfismo e sobrecarga de operadores, que são suportados por esta extensão do VHDL.

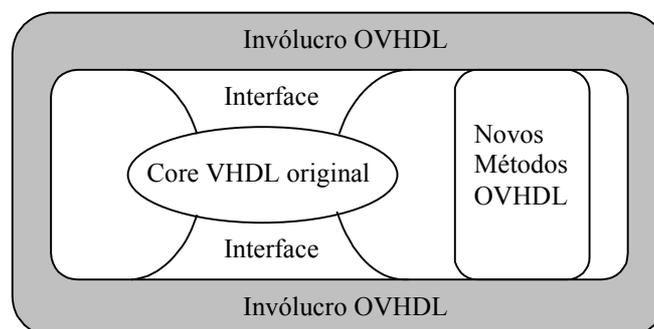


Figura 10 – Componente reutilizável básico.

- **SpecC**

A síntese de SoCs feita diretamente da especificação em nível de sistema é uma

abordagem mais flexível, pois o projetista se preocupa com o sistema como um todo, não desenvolvendo separadamente as partes hardware e software. Entretanto, é necessário o desenvolvimento de uma tecnologia de CAD para fornecer técnicas e ferramentas para a síntese de diferentes componentes (*cores*) [GAJ00].

A linguagem SpecC é uma extensão da linguagem C que inclui suporte para três modelos computacionais: processos sequenciais concorrentes (para software), máquinas de estado finitas com bloco de dados (para hardware), e eventos discretos (para protocolos) [CEC99].

A linguagem SpecC foi desenvolvida por diferentes razões:

- A fim de especificar um processo de projeto para construir SoCs, é necessária uma linguagem para especificação e modelagem de sistemas nos diferentes estágios do processo de projeto. Por este motivo, a linguagem SpecC é usada como uma linguagem de especificação de SoCs executável.
- Foi desenvolvida como uma SLDL (linguagem de descrição em nível de sistema) [BAR98]. Assim, é possível estudar e experimentar modelos e ferramentas, e desenvolver uma metodologia para SoCs clara, que incluía *cores*.
- Inicialmente, pode servir como uma SLDL padrão, para ser estendida mais tarde, incluindo partes mecânicas, analógicas, ou outras partes do sistema; assim, ela pode tornar-se um padrão universal para sistemas.
- **Esterel**

A linguagem Esterel é uma linguagem síncrona desenvolvida a partir de 1982 conjuntamente por dois laboratórios franceses INRIA e ENSMP. A necessidade de atender simultaneamente concorrência e determinismo é a base do modelo de programação síncrono da linguagem Esterel [FAR00]. Os princípios básicos deste modelo são os seguintes:

- *Reatividade*: O modelo é reativo uma vez que se aplica a sistemas que entram em ação reagindo à presença de estímulos vindos do ambiente em instantes discretos. Cada reação, portanto, está associada a um instante preciso; o conjunto destes instantes caracteriza a vida do sistema reativo.
- *Sincronismo*: As reações são instantâneas, sendo que as entradas e saídas se apresentam sincronizadas. As reações são atômicas, e o modelo síncrono não permite uma nova ativação do sistema enquanto o mesmo estiver reagindo ao estímulo atual. Portanto, não há concorrência entre as reações, eliminando assim uma fonte de não determinismo que corresponderia ao entrelaçamento (*interleaving*) de execuções concorrentes.
- *Difusão Instantânea*: A comunicação entre componentes é sempre realizada por um mecanismo de difusão instantânea (*broadcasting*). Um sinal emitido é visto no mesmo instante da sua emissão por todos seus receptores. A difusão é limitada aos instantes de

reação: um sinal emitido num instante é visto como presente em todos os receptores neste mesmo instante; pode haver, entretanto, várias emissões e recepções de sinal em seqüência num mesmo instante.

- *Determinismo*: Contrariamente às abordagens assíncronas, onde a concorrência leva ao não determinismo, o modelo faz conviver concorrência e determinismo. O modelo síncrono é determinista, o que tem como resultado a simplificação das programações reativas e a capacidade de reproduzir seus comportamentos, simplificando testes e verificações destas.

Na abordagem síncrona, o tempo é considerado como uma entidade multiforme, sendo visto como um evento externo entre outros, de características diversas do tempo físico; a noção de tempo físico é na verdade substituída pela noção de ordem e de simultaneidade entre eventos. Essa visão de tempo multiforme e a não ocorrência de entrelaçamento facilitam em muito o entendimento e a análise dos sistemas de tempo real.

2.4.3 Proteção da Propriedade Intelectual

Companhias que desenvolvem *cores* (também denominados como IP *cores*, do inglês *Intellectual Property*), ferramentas para integração de *cores* e organizações como a VSI Alliance geraram grandes expectativas com relação ao valor e à reusabilidade dos *cores*. Apesar de tudo, um obstáculo conhecido para as metodologias baseadas no reuso é a falta de mecanismos para proteger os direitos dos projetistas e proprietários dos *cores*.

Para que se tenha sucesso, o projeto de um determinado *core* deve:

- fornecer técnicas para avaliar a correção e a qualidade (área, velocidade, potência, testabilidade) dos sistemas que virão a utilizar o *core*;
- garantir proteção da propriedade intelectual tanto do projetista do *core* quanto do usuário do *core*;
- fornecer mecanismos de *test drive* aos projetistas que desejam adquirir o *core*.

O maior desafio para as ferramentas de projetos baseados no reuso de IPs é a validação do projeto sem comprometer a propriedade intelectual [DAL00a].

Uma técnica utilizada para a proteção de propriedade intelectual é a *watermarking*. Ela protege o projetista do *core*, embutindo uma assinatura digital em componentes IP. Se a assinatura digital não puder ser removida, em caso de litígio, o fornecedor pode provar que o componente, ilegalmente empregado pelo usuário, carrega sua assinatura. A principal limitação da *watermarking* é que ela protege o projetista do *core* só de uma instanciação ilegal de seu componente, mas não esconde sua propriedade intelectual [KAH98].

Uma *watermarking* é uma marca que é:

- Embutida em um objeto (texto, imagem, audio, vídeo) ou parte de propriedade intelectual

(hardware, software, algoritmo, organização de dados);

- Projetada para identificar o autor, a fonte, as ferramentas e as técnicas utilizadas e/ou o receptor da propriedade intelectual.
- Difícil de detectar e remover.

Uma outra técnica para proteção é baseada em criptografia [DAL00a]. Neste caso o fornecedor libera ao usuário um modelo de simulação preciso, e protege seu *core* criptografando-o. Este modelo pode ser instanciado e simulado dentro do projeto do usuário. O modelo criptografado é fornecido ao usuário como um objeto pré-compilado para ser conectado ao seu sistema no momento da síntese. Usar um modelo criptografado e pré-compilado traz à tona dois problemas:

- O modelo criptografado não é portátil. O projetista do core precisa preparar uma versão diferente do modelo criptografado para cada arquitetura alvo.
- O usuário do *core* deve confiar que o modelo criptografado será fiel ao modelo funcional.

Uma terceira técnica para a proteção de IP é empregada no ambiente JavaCAD [DAL00a]. JavaCAD é um ambiente distribuído, baseado em uma arquitetura cliente-servidor onde os clientes são os usuários de *cores* e os servidores são os fornecedores de *cores*.

O objetivo de JavaCAD é fornecer uma plataforma que suporta o uso de componentes IP distribuídos através da Internet [DAL00b]. Antes de comprar um componente, o usuário do IP quer obter estimativas de custo em termos de área, desempenho e potência da instanciação do componente. Durante as pesquisas no início do projeto, estimativas aproximadas de custo podem ser suficientes, e podem ser obtidas de um *datasheet* padrão. À medida que o projeto for sendo refinado, o usuário do IP pode precisar de informações mais precisas. Primeiro, ele pode querer um modelo funcional abstrato para testar a funcionalidade do componente dentro do seu projeto. Após, ele pode necessitar estimativas de custo precisas que não podem ser obtidas sem o conhecimento (parcial) da implementação e do ambiente do componente. No lado do fornecedor do IP, a questão principal é a proteção da propriedade intelectual: o fornecedor quer comunicar-se com o usuário com a maior quantidade de informações possível para facilitar a compra, mas ele não quer revelar a sua propriedade intelectual ao usuário do IP.

A definição de formatos padrão para descrições de projetos, como o VHDL e o EDIF, foram os primeiros passos importantes para que fosse possível fornecer infra-estrutura às ferramentas para reuso no projeto de hardware. Com o crescimento da computação por rede, surgiram novos paradigmas para a organização do acesso à informações específicas ao projeto. A idéia básica é que as informações críticas de projeto, dispersas geograficamente, podem ser automaticamente coletadas e visualizadas através da Web. Virtualmente, todas as maiores companhias de hardware têm servidores Web que fornecem informações (*data-sheets*, *application notes*, documentação técnica) aos projetistas.

A Internet pode ser aproveitada não só para compartilhar informações, mas também

para executar tarefas críticas de projeto, tais como validação e otimização. Nesta abordagem, um projetista é um cliente em uma arquitetura cliente-servidor, e ele pode solicitar serviços de uma variedade de servidores. O uso de ferramentas de CAD é um dos tipos de serviços, como também o é a distribuição de propriedade intelectual.

No JavaCAD, o usuário do IP cria um projeto instanciando e conectando módulos. Alguns módulos são disponíveis localmente, outros são remotos. O JavaCAD tem várias características únicas:

- Não especifica quais métodos em um módulo remoto devem ser remotos. Cada fornecedor pode decidir o grau de proteção que ele considera necessário.
- Protege também o usuário, pois um módulo não tem visibilidade alguma da estrutura do projeto além de seu limite.
- É completamente distribuído. Fornecedores e usuários podem seguramente comunicar-se pela Internet.

Todas as técnicas de proteção de IP enfocam, principalmente, o projeto de hardware. Entretanto, a maioria dos sistemas em larga escala contém hardware e software. A maioria dos componentes IP bem sucedidos no mercado, hoje, são *cores* que incluem um ou mais processadores.

Enquanto a proteção do *core* do hardware é ainda uma necessidade fundamental, o software forma frequentemente a maioria dos IP valiosos para os projetistas de sistemas [DAL00a]. Portanto, é preciso garantir proteção do software desenvolvido pelo usuário, ou pelo fornecedor.

A Figura 11 [DAL00a] mostra um projeto genérico, utilizando um *core* (em cinza). A proteção de IP é representada como uma barreira (linha pontilhada) entre o *core*, que é uma propriedade intelectual do fornecedor, e o resto do sistema, que pertence ao projetista. Ambos, o fornecedor e o projetista, não podem “enxergar” através da barreira, pois, se pudessem, violariam a propriedade intelectual da outra parte.

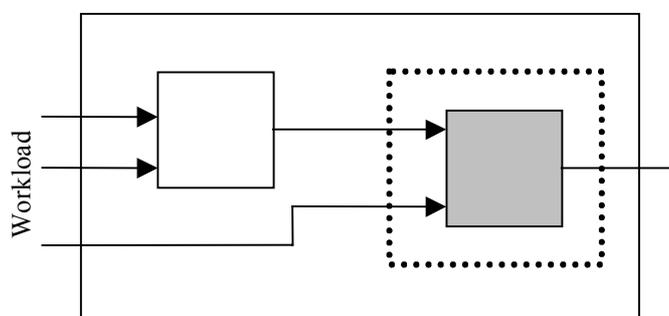


Figura 11 – Projeto RTL com um componente virtual.

Em um ambiente de simulação distribuída, o projetista inicia a sessão de simulação aplicando um conjunto de vetores de teste (*workload*) nas entradas primárias do projeto. Caso esteja sendo realizada uma simulação funcional, é muito provável que a instância do *core*

contenha localmente o modelo funcional. Entretanto, caso o projetista deseje informações mais apuradas, a simulação do *core* ocorre remotamente, nas máquinas do fornecedor do *core*. A premissa básica desta simulação distribuída, ou virtual, é que eventos nas fronteiras não carregam nenhuma propriedade intelectual. Infelizmente, em geral, essa premissa não é verdadeira no caso de componentes virtuais programáveis.

A Figura 12 [DAL00a] ilustra um projeto que contém um *core* de microprocessador (μ P), memória (M) e alguns dispositivos de entrada/saída (I/O) conectados por um barramento. A funcionalidade do projeto é determinada pelo algoritmo executado no *core*. Antes da execução, o algoritmo precisa ser compilado e carregado na memória.

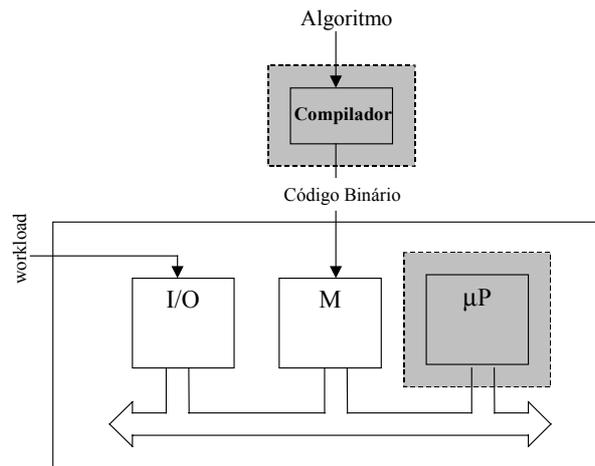


Figura 12 – Projeto em nível arquitetural com um *core* de microprocessador virtual.

O *workload* externo é aplicado através dos dispositivos de entrada/saída. Mesmo se for assumido que o *workload* não é crítico, novas questões surgem com a presença do *core* do microprocessador: (i) o algoritmo é uma propriedade intelectual do projetista; (ii) o *core* do microprocessador é uma propriedade intelectual do fornecedor; (iii) o compilador pertence ao fornecedor e pode conter informações críticas sobre a arquitetura do *core* microprocessador; (iv) o código executável representa o algoritmo, contendo informações críticas do projetista.

Logo, ocorrem dois problemas: (i) o compilador precisa do algoritmo do projetista para gerar o código binário, desprotegendo desta forma a propriedade intelectual do projetista do sistema; (ii) a simulação do sistema necessita do código executável que é executado no *core* do microprocessador, desprotegendo a propriedade intelectual do fornecedor do *core*. Para este propósito, foi proposta uma nova abordagem baseada no encapsulamento e criptografia, vista na Figura 13 [DAL00a].

O módulo executável, que carrega a propriedade intelectual do fornecedor é representado pelo bloco F. Ele pode ser o compilador ou o microprocessador da Figura 12. Em um ambiente desprotegido, o módulo seria diretamente executado pelo projetista para processar os dados de entrada (in) e fornecer os resultados (out). No ambiente proposto, o módulo é fechado em uma caixa que fornece proteção de IP, inibindo a execução direta do módulo, enquanto exibe a mesma interface externa e o mesmo comportamento de I/O. O bloco cinza da Figura 9 é chamado de módulo empacotado (*wrapped module*).

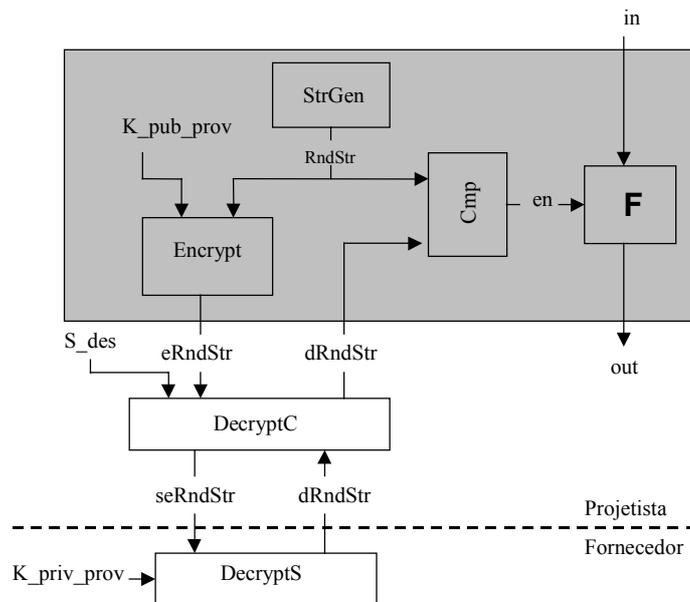


Figura 13 – Proteção de IP de hardware/software.

O projetista instancia e usa módulos empacotados no ambiente do JavaCAD como se eles fossem abertos (desempacotados). Entretanto, a qualquer momento em que ele tente executar o módulo para processar novos dados de entrada, o invólucro (*wrapper*) verifica se a execução é ou não autorizada pelo fornecedor do *core*.

O mecanismo de autorização funciona da seguinte forma:

1. Uma *string* (**RndStr**) é randomicamente gerada pelo bloco **StrGen** no invólucro do IP.
2. A *string* é criptografada usando a chave pública do fornecedor (**K_pub_prov**).
3. A *string* criptografada (**eRndStr**) é passada para um servidor de autorização (**DecryptS**) rodando na máquina do fornecedor.
4. O servidor de autorização decriptografa a *string* usando a chave privada do fornecedor (**K_priv_prov**) e decide se o módulo com proteção de IP pode ou não ser executado na máquina do projetista. A autorização (se concedida) é notificada através do envio da *string* decriptografada (**dRndStr**), de volta para o projetista.
5. A *string* decriptografada é recebida pelo invólucro do IP e comparada com a *string* original (**RndStr**). Se elas forem iguais, o módulo interno **F** é executado e os resultados são disponibilizados ao projetista.

Como mostrado na Figura 13, o módulo empacotado não tem comunicação direta com o servidor de autorização. Preferencialmente, um cliente de decriptografia (**DecryptC**) externo ao invólucro do IP é usado para este propósito. Existem duas razões para que seja colocado um módulo adicional entre o invólucro do IP e o servidor de autorização. Primeiro, a decriptografia cliente adiciona a assinatura do projetista à *string* criptografada, assim, permite ao fornecedor identificar o projetista e, possivelmente, permitir que ele execute o módulo IP. Segundo, a decriptografia cliente verifica que nada além da *string* é passado ao fornecedor,

através da monitoração do tamanho dos dados transmitidos.

O módulo IP empacotado é executado na máquina do projetista. Logo, dados de entrada e de saída (geralmente carregando IP do projetista) não são enviados através da Internet nem desprotegem o fornecedor. Conforme [DAL00a] este protocolo de autorização não impõe penalidades de desempenho significativas.

2.4.4 Teste

Na maioria dos casos, exceto para *soft cores*, os usuários de *cores* possuem pouco conhecimento do conteúdo interno dos mesmos, tratando-os como caixa preta. A estratégia do teste dos *cores* deve ser definida pelos projetistas dos *cores*. O usuário tem apenas acesso à fronteira externa dos módulos de hardware instanciados.

Do ponto de vista do projetista do *core*, o método de teste deve ser definido sem o conhecimento de onde o *core* virá a ser utilizado. Como resultado, o projetista pode ou não fornecer um teste com a qualidade adequada. Se a cobertura é muito fraca, a qualidade do sistema está em risco. Se é muito alta, o custo de teste pode aumentar (tempo).

Conforme mencionado na Seção 2.4.1, o grupo de trabalho *IEEE P1500* trabalha na definição de padrões para a realização do teste de *cores* embutidos em SoCs.

Técnicas de BIST (*built-in-self-test*) são utilizadas para o teste de *cores* [ZOR99]. Uma outra técnica de teste é baseada na serialização dos padrões de testes. Uma varredura ao redor do *core* permite acesso indireto, mas completo, a todas as entradas e saídas do *core*. Esta técnica é chamada de mecanismo *ring-access*. Ela pode ser interna ou externa ao *core* e pode ser implementada pelo projetista ou pelo integrador do *core*. A complexidade e os requisitos de hardware desta abordagem são aceitáveis para os atuais circuitos integrados, já que permanece independente das limitações de pinos [GUP97].

Em adição aos modos de teste normal e internos em *cores*, um sistema baseado em *cores* frequentemente requer vários outros modos de testes. Um deles é o teste externo ao *core* para detectar falhas de interconexão (estática ou dinâmica) entre *cores* e para testar a lógica definida pelo usuário, ao redor do *core*.

O maior desafio na concretização de um sistema baseado em *core* é a integração e coordenação de testes e capacidade de diagnóstico no circuito integrado. Isto engloba o teste interno de cada *core*, da lógica definida pelo usuário, e suas interconexões.

Para resumir, os testes de projetos baseados em *cores* enfrentam três problemas principais:

- Tornar o teste de *core* portátil;
- Fornecer acessibilidade aos nós internos do *core*;
- Compor um teste integrado e seu mecanismo de controle para todo o sistema.

2.5 Ferramentas para Personalização de Cores

Nesta Seção, serão mostradas algumas ferramentas disponíveis atualmente no mercado e que são utilizadas para parametrizar e distribuir *cores*. A ênfase é dada para dispositivos programáveis tipo FPGA, objeto de estudo deste trabalho.

2.5.1 Xilinx

A empresa Xilinx possui diversas ferramentas que permitem projetar sistemas digitais utilizando *cores*. Dentre estas ferramentas, pode-se citar o *Core Generator*, que serve para parametrizar *cores*, sendo utilizado em conjunto com o *IP Internet Capture*, que fornece um método automatizado para identificar, capturar, e documentar *cores*. Outra ferramenta, chamada *Chipscope*, gera *cores* que se comunicam com os módulos do sistema, permitindo a análise dos sinais internos.

- **Xilinx Core Generator System**

O *Xilinx CORE Generator System* [XIL01a] gera *cores* parametrizados e otimizados para os FPGAs da Xilinx. O usuário seleciona um *core*, identifica os parâmetros de entrada e é gerado o *core* otimizado. Ele é compatível com descrições VHDL, Verilog e com fluxos de projeto baseados em esquemáticos.

Os *cores* são entregues com o projeto lógico e com a planta baixa otimizada ou o layout deste. O fabricante garante que o desempenho é independente do tamanho do dispositivo FPGA e que ele permanece constante à medida em que mais *cores* são acrescentados. Para cada *core* há a geração do seu *data sheet* e de seu modelo VHDL. A ferramenta permite que se gere *cores* simples gratuitamente, como por exemplo, um multiplexador (mostrado na Figura 14), somadores, subtratores ou memórias.

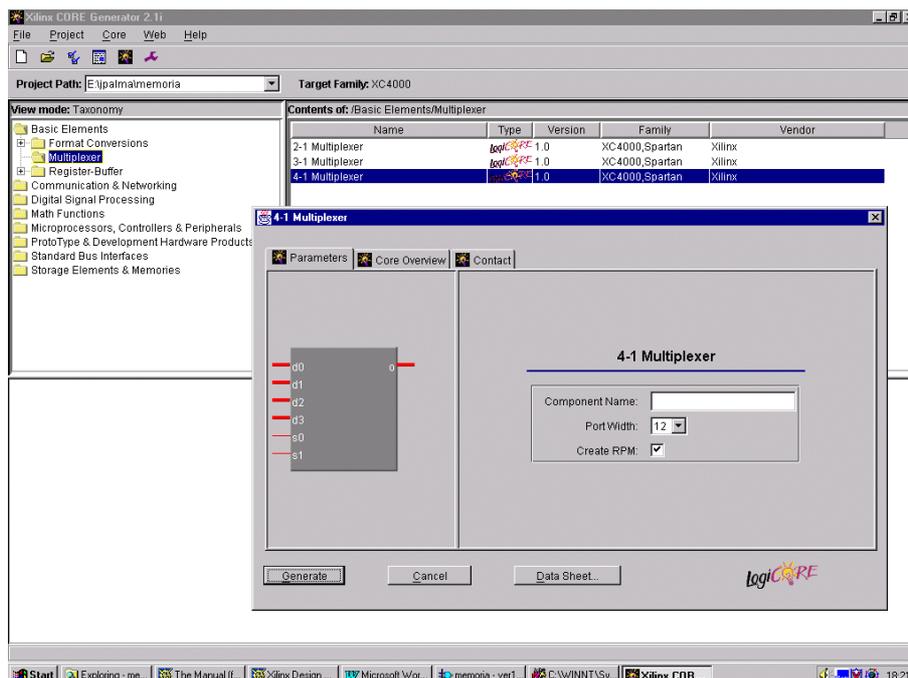


Figura 14 – CORE Generator da Xilinx.

Para *cores* mais complexos, como DSPs, processadores e *cores* para telecomunicações, que precisam ser comprados, a ferramenta disponibiliza links para que o usuário possa entrar em contato com o fornecedor.

- **Xilinx IP Internet Capture**

A ferramenta *IP Internet Capture* [XIL01b] opera em conjunto com o sistema *Xilinx CORE Generator*, catalogando *cores* desenvolvidos. O *IP Internet Capture* empacota módulos de projetos criados por projetistas individuais, disponibilizando-os a outros que utilizam o *Xilinx CORE Generator*.

A ferramenta *IP Internet Capture* não fornece apenas a capacidade de fazer a busca de IPs. Ela também permite que o projetista inclua modelos de simulação comportamental, *testbenches*, ou outros arquivos com vetores de simulação.

O *IP Internet Capture* requer que os projetistas forneçam links para a documentação de seus IPs. Isto assegura que todo IP catalogado no sistema *Xilinx CORE Generator* forneça um ponto de partida útil para que os projetistas avaliem se o IP está de acordo com as suas necessidades. Esta documentação é apresentada na forma de arquivos PDF ou páginas HTML.

Esta ferramenta fornece aos projetistas um método automatizado para identificar, capturar e documentar um *core*. Este *core* pode estar na forma de código VHDL ou *Verilog*, ou em forma de uma *netlist* de função fixa.

O *core* pode ser compartilhado através da rede do cliente ou em um web site. Uma vez que o novo módulo é capturado e enviado, outros projetistas podem utilizar *browsers* padrão da Internet para fazer download do IP e instalá-lo em sua cópia do sistema *Xilinx CORE Generator*.

- **ChipScope**

O aumento da densidade dos dispositivos FPGAs torna cada vez mais difícil a tarefa de utilizar equipamentos de teste para examinar estes dispositivos em funcionamento. O *ChipScope* [XIL01c] é uma ferramenta para geração de *cores*, que permite a análise lógica dos sinais internos de sistemas desenvolvidos para dispositivos *Virtex*, *Virtex-E* e *Spartan-II*. Os *cores* disponibilizados pela ferramenta *ChipScope* comunicam-se com os módulos do sistema em desenvolvimento, permitindo ao projetista visualizar o funcionamento do hardware da mesma forma que faria com um analisador lógico, sem a necessidade da utilização de equipamentos de teste.

As ferramentas disponibilizadas pelo *ChipScope* são:

- *Core Generator*: fornece *netlists* e modelos de instanciação para o *core* ICON (*Integrated Controller*) e para o *core* ILA (*Integrated Logic Analyzer*);

- *Core Inserter*: insere automaticamente os dois tipos de *cores* no projeto sintetizado do usuário.
- *Analyzer*: permite configurar e exibir os sinais dos *cores* ILA. Os *cores* ILA permitem que se possa escolher os sinais de sincronismo (*trigger*) e capturar sinais. O *core* ICON comunica-se com os pinos *Boundary Scan* da placa de prototipação.

A Figura 15 ilustra a estrutura geral de um sistema utilizando os *cores* inseridos pelo *ChipScope*.

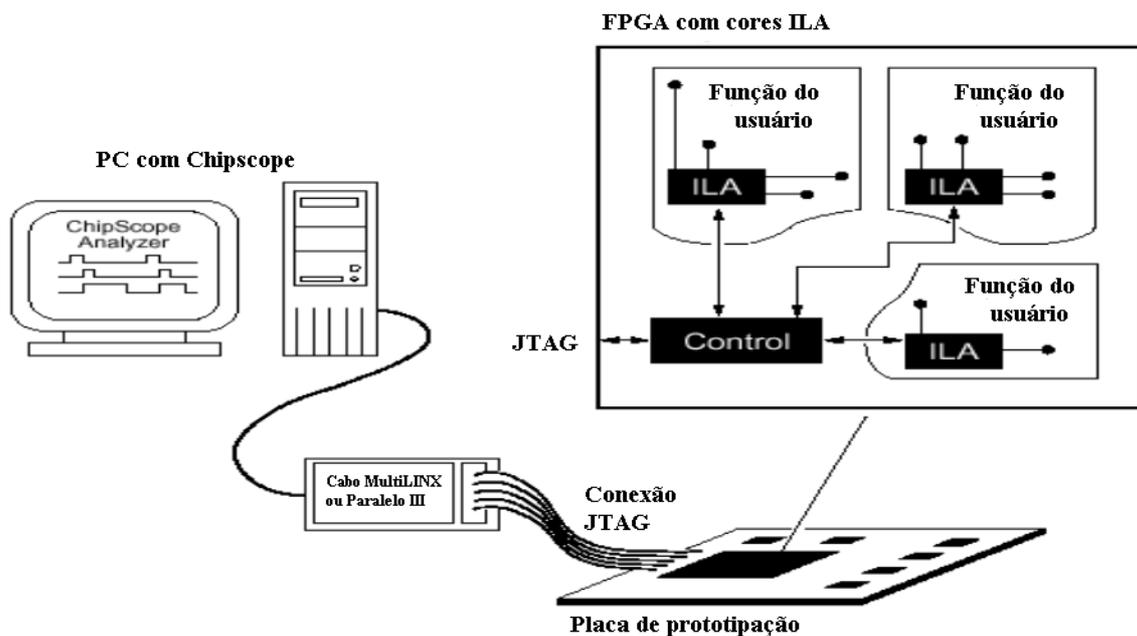


Figura 15 – Visão geral do ChipScope.

Os *cores* ILA (para leitura dos sinais internos) e ICON (de controle) podem ser inseridos no projeto do usuário de duas formas:

- Gerando os *cores* com o *Core Generator* e instanciando-os no código-fonte HDL;
- Unindo-se os *cores* após a síntese lógica dos módulos do sistema em desenvolvimento, usando o *Core Inserter*.

A Figura 16 ilustra o fluxo de projeto utilizando os *cores* gerados pelas ferramentas do *ChipScope*.

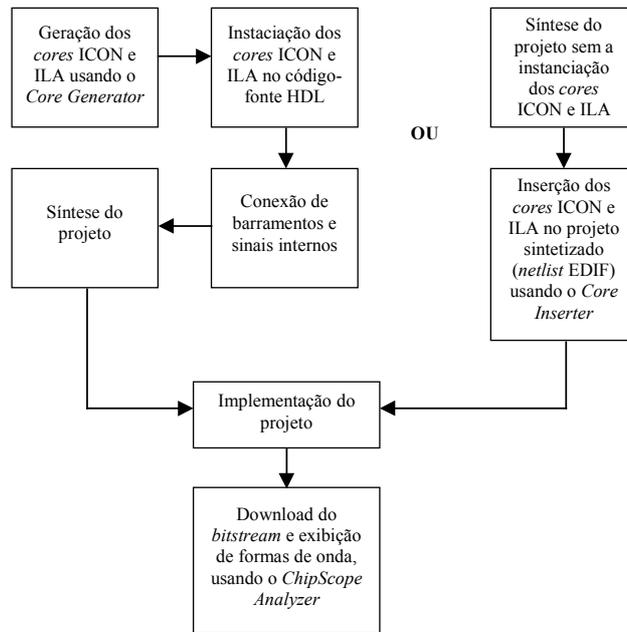


Figura 16 – Fluxo de projeto utilizando ChipScope.

2.5.2 Altera

A Altera disponibiliza *cores* em seu web site (<http://www.altera.com>), denominando-os megafunções [ALT01a]. As megafunções são IPs, parametrizados e pré-testados, que são otimizados para fazer uso eficiente da arquitetura do dispositivo programável ao qual o projeto é direcionado (somente para dispositivos Altera). As megafunções são disponíveis de duas fontes:

- **Altera Megafunction Partners Program (AMPP).** O AMPP é uma aliança entre a Altera e desenvolvedores de propriedade intelectual, fornecendo aos usuários de dispositivos programáveis da Altera um grande conjunto de megafunções. Para obter-se as megafunções do AMPP, é preciso conectar-se diretamente com os fornecedores. O web site da Altera fornece uma extensa lista de fornecedores que compõem o AMPP.
- **MegaCore Functions da Altera.** As funções *MegaCore* são arquivos descritos em HDL, sendo muitas vezes utilizada a linguagem AHDL, proprietária da Altera [ALT01b]. Estas megafunções são desenvolvidas, pré-testadas, documentadas e licenciadas pela Altera.

Estas funcionalidades são semelhantes aos sistemas *Xilinx Core Generator* e *Xilinx IP Internet Capture*.

A Altera também oferece um sistema de *test drive* para que se possa avaliar uma megafunção antes de comprá-la. Este sistema permite que o projetista instancie, compile e simule a função para verificar seu tamanho e desempenho, antes de decidir adquirir a licença da mesma. O fluxo de projeto utilizando o *MegaCore* é mostrado na Figura 17.

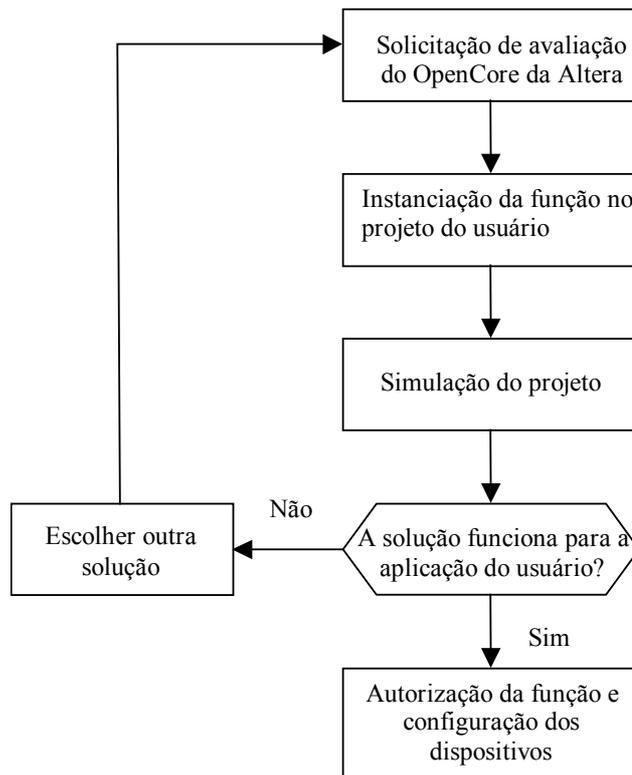


Figura 17 – Fluxo de projeto usando o MegaCore.

O web site da Altera permite realizar o download das megafunções, como ilustrado na Figura 18.



Figura 18 – Funções MegaCore disponíveis no web site da Altera.

2.6 Distribuição de cores na Internet

Existem na Internet dezenas de web sites de fornecedores de *cores* abertos ou com propriedade intelectual protegida. A maioria dos web sites que distribuem *cores* na Internet são de empresas comerciais, que desenvolvem seus próprios módulos e os vendem através da rede. Estes módulos são geralmente *soft cores* sintetizáveis ou *hard cores* já sintetizados que não se encontram disponíveis para download. Apenas os documentos contendo as características funcionais dos *cores* podem ser acessados. Os *cores* somente são enviados ao cliente após este entrar em contato com o fornecedor e pagar por eles.

Atualmente, algumas destas empresas estão investindo em sistemas de *test drive* dos *cores*, ou seja, o cliente pode simulá-los através de simuladores HDL padrão antes de comprá-los, conforme mencionado na Seção 2.5.2. Para isto, o cliente seleciona o *core* desejado entre os disponíveis na lista do fornecedor. Logo após ele preenche uma ficha de registro, contendo os dados do cliente. Geralmente o pedido é atendido em poucas horas, e o cliente recebe um *link* para FTP via e-mail e pode fazer o download do *soft core* criptografado, de um modelo de simulação também criptografado, de padrões de teste e de uma documentação abreviada do produto.

Outra categoria de distribuidores de *cores* encontrada na Internet é a de entidades que promovem a distribuição de módulos com propriedade intelectual livre, também chamados de *cores* abertos. Devido ao fato dos *cores* pagos serem bastante caros, a quantidade dessas entidades vem crescendo cada vez mais. Estas entidades geralmente disponibilizam *soft cores* sintetizáveis já simulados, depurados e testados, que podem ser capturados em seus web sites. Juntamente com os *cores*, pode-se fazer o download das documentações completas, contendo informações sobre a tecnologia para a qual foi desenvolvido, interface com o mundo externo e informações para teste. Em alguns desses web sites também pode-se fazer o download de ferramentas para entrada de dados, ferramentas para simulação de código em *Verilog* ou *VHDL*, e também ferramentas de síntese tanto para ASICs como para FPGAs.

Os projetistas podem contribuir com estes sites de distribuição de IPs gratuitos, inserindo novos *cores* às bibliotecas e ajudando na identificação ou resolução de *bugs*.

Além de web sites onde é possível comprar *cores* e web sites onde é possível encontrar gratuitamente *soft cores*, há também uma classe de web sites que servem como “portais” de *cores*, ou seja, têm como objetivo principal reunir várias outras entidades que disponibilizam *cores* e ferramentas. Estes “portais” organizam e classificam estes serviços e facilitam o acesso à essas entidades, direcionando o usuário à seus web sites.

A maior parte dos *cores* disponíveis, tanto os abertos quanto os com IP protegida, são descritos em linguagem *VHDL*, enquanto alguns são descritos em *Verilog*. Já as aplicações mais encontradas são *cores* para telecomunicação, processadores, memórias, DES, USB e Ethernet.

A Tabela 2 mostra alguns fornecedores de *cores*, a linguagem de descrição utilizada, as principais aplicações disponíveis e o endereço de seus respectivos web sites.

Tabela 2 – Serviços de distribuição de cores na Internet.

Fornecedor	Descrição	Principais aplicações	Endereço	IP
Free-IP	VHDL	RAMs parametrizáveis, RISC8, DES	http://www.free-ip.com/	Aberto
Opencollector	VHDL e Verilog	Sistemas Embarcados, Processadores	http://www.opencollector.org	Aberto
In-Silicon	Verilog	Core PCI 33MHz, Controlador USB, MAC Ethernet	http://www.in-silicon.com	Teste/ Pago
Vautomation	VHDL	Microprocessadores, USB, Ethernet, Ferramentas de Projeto	http://www.vautomation.com	Pago
CMOSexod	VHDL e Verilog	Processador CISC, DSP 12-bits, Controlador SDRAM	http://www.cmosexod.com	Aberto
FMF	VHDL e Verilog	Bibliotecas de FIFOS, de SRAMs e de Processadores	http://www.vhdl.org/fmf	Aberto
Design-Reuse	VHDL e Verilog	Microcontrolador de 8-bit, Controladores de SDRAM, Ferramentas de Projeto	http://www.design-reuse.com	Teste/ Pago
Opencores	VHDL e Verilog	Microprocessadores, Memórias, DSPs, Controladores de Video	http://www.opencores.org	Aberto
Arasan Chip Systems	VHDL e Verilog	Core USB Genérico, Controlador de Hub USB 2.0	http://www.arasan.com/	Pago
Amphion Semiconductor	Netlist	Processamento de Imagem, Processamento de voz, Codificação de Canais, DSP	http://www.amphion.com	Pago
Eureka Technology	VHDL e Verilog	Controlador de sistema, Barramento PCI	http://www.eurekatech.com	Pago
Altera	VHDL e Verilog	Controladores de Memória, Comunicações, PCI, DSP	http://www.altera.com/products/ip/ipm-index.html	Teste/ Pago
Innovative Semiconductors	-	Codificadores/Decodificadores de Vídeo, USB	http://www.isi96.com	Pago

3 Distribuição de IP Cores

O Capítulo anterior tratou dos *cores* e mostrou as dificuldades em sua utilização. Este Capítulo trata de aspectos importantes para que componentes possam ser reutilizados adequadamente, como por exemplo: técnicas de codificação de componentes para reuso, documentação e suporte à verificação.

Na Seção 3.1 são apresentadas algumas práticas de codificação visando facilitar a reutilização de módulos de hardware em novos projetos. A Seção 3.2 e 3.3 tratam, respectivamente, da documentação e dos testes que devem acompanhar o *core*.

Ao final do Capítulo, na Seção 3.4, apresenta-se a primeira contribuição deste trabalho, um ambiente desenvolvido para a distribuição de *cores*, independentemente de seu formato, via Internet.

3.1 Codificação VHDL para Distribuição de Cores

Cada vez mais as equipes de projeto utilizam IP *cores* para construir SoCs. Assim, está surgindo um consenso comum entre os projetistas sobre os aspectos-chave de um projeto baseado em reuso, ou seja, um modelo padrão para reuso de projetos [KEA00].

Neste modelo padrão, a proposta fundamental é que *cores* bem projetados são a chave para o sucesso do projeto de um SoC. Não importa o quão bom é o fluxo de integração do SoC, se os blocos que estão sendo utilizados não forem bem projetados o processo de integração será longo e complicado.

Um circuito projetado para ser reutilizado deve ser descrito da maneira mais simples possível e mesmo assim atingir o desempenho desejado. Quando se projeta pensando em reutilização, algumas recomendações importantes devem ser observadas:

- Suporte a tratamento rápido de interrupções (com um banco de registradores dedicado).
- Usar construções simples, tipos básicos (para VHDL) e esquemas simples para clock.
- Usar estilo de código, nomenclatura, e estruturas consistentes para os processos e máquinas de estados.
- Usar um esquema de particionamento regular, com todas as saídas dos módulos registradas e com os módulos aproximadamente do mesmo tamanho.
- Fazer o código RTL fácil de entender, usando comentários, nomes significativos, e constantes ou parâmetros, em vez de números.

Tomando estes cuidados, o projetista será capaz de produzir um código que convirja rapidamente ao desempenho desejado, em termos de funcionalidade, temporização, área e potência.

3.1.1 Práticas Básicas de Codificação

Estruturas simples e regulares são mais fáceis de integrar, verificar e sintetizar do que estruturas complexas. Para se desenvolver estruturas simples, deve-se utilizar algumas convenções e práticas de codificação. Isto facilita a identificação e compreensão de sinais e estruturas que compõem a descrição. A seguir serão descritas algumas das convenções mais importantes, com exemplos de codificação em VHDL.

A) Convenção geral para nomes

- Usar letras minúsculas para os nomes de sinais, nomes de variáveis e nomes de portas (entradas/saídas do circuito). Ex.: SIGNAL data_out : STD_LOGIC.
- Usar letras maiúsculas para nomes de constantes e para tipos definidos pelo usuário. Ex.: CONSTANT MAX_COUNT : INTEGER = 64;
- Usar nomes significativos para sinais, portas, funções e parâmetros. Ex.: não utilizar *ra* como nome de um barramento de endereço de RAM. Para este componente, o nome *ram_addr* seria mais sugestivo.
- Usar o nome *clk* para sinais de *clock*. Se existe mais de um *clock* no projeto, usar *clk* como prefixo para todos os *clocks*, como por exemplo *clk_1* ou *clk_interface*.
- Para sinais ativos em '0' (zero), utilizar *_n* no final do nome.
- Usar o prefixo *rst* nos nomes de sinais de reset.
- Na descrição de barramentos, usar uma ordem consistente de bits (*x DOWNTO y*), a fim de estabelecer um padrão entre os projetos. Por exemplo, *bus1 : STD_LOGIC_VECTOR (7 DOWNTO 0)*.
- Quando possível, usar o mesmo nome, ou um similar, para portas e sinais interconectados.
- Não utilizar palavras reservadas de outra linguagem (VHDL ou Verilog) para nomes de elementos no código-fonte. Isto permite uma tradução mais fácil de VHDL para Verilog (ou vice-versa).

B) Nomes de arquiteturas

- Para as arquiteturas, aconselha-se utilizar nomes sugestivos como:
 - ARCHITECTURE *rtl* OF *my_model* IS identificando que se trata do modelo RTL ou
 - ARCHITECTURE *tb* OF *my_test_bench* IS identificando que se trata do testbench.

C) Cabeçalhos em arquivos-fonte

- Incluir um cabeçalho no início de cada arquivo-fonte. O cabeçalho deve incluir: (i) aviso de direitos autorais, (ii) nome do arquivo, (iii) autor, (iv) data de criação, (v) versão, (vi) Contato, (vii) descrição da função do módulo e (viii) histórico das modificações, incluindo datas, nome do projetista e descrição das mudanças.

Exemplo de cabeçalho:

```
-----  
-- This confidential proprietary software may be used only as  
-- authorized by a licensing agreement from GAPH.  
-- File      :    pci_core.vhd  
-- Author    :    Ewerton Capellatti  
-- Date      :    06/07/01  
-- Version   :    1.0  
-- Abstract  :    This file has the entity, architecture and  
--                configuration of the PCI 2.1 MacroCell core  
--                module.  
-- Modification History:  
-- Date      By          Version      Change Description  
-----  
-- 06/07/01  EC          0.1         Original  
-- 06/26/01  JC          Changes in ism_ad_em_ffd_n  
--  
-----
```

D) Comentários

- Usar comentários para explicar cada processo, função e declaração de tipos e subtipos.
- Usar comentários para explicar portas, sinais e variáveis, bem com grupos de sinais e variáveis.
- Os comentários devem ser inseridos na linha ou linhas anteriores às do código ao qual se refere.
- Devem ser breves, concisos e explicativos. Explicações de funções óbvias devem ser evitadas.

Exemplo de bom comentário:

```
-- Create a subtype INTEGER_256 for built-in error checking of  
-- legal values  
SUBTYPE INTEGER_256 IS TYPE INTEGER RANGE 0 TO 255;
```

E) Layout

- Não usar mais de uma expressão (*statement*) por linha, facilitando a visualização dos mesmos e a inserção de comentários.
- Usar no mínimo 3 espaços para cada indentação e no máximo 6. Indentações estreitas dificultam a identificação de escopos aninhados. Indentações largas fazem com que o código rapidamente alcance a margem direita. Exemplo:

```

IF (bit_width(m+1) >= 2) THEN
  FOR i IN 1 TO m LOOP
    Matrix(i) := j;
  END LOOP;
END IF;

```

- Evitar o uso de TABs. Diferenças entre editores podem modificar a posição das tabulações, comprometendo a indentação do código.
- O comprimento da linha não deve exceder 72 caracteres. Linhas com mais de 72 caracteres são difíceis de ler tanto no monitor do PC quanto em impressões. Se a linha for maior que 72 caracteres, deve-se dividi-la e indentar a próxima linha, indicando que se trata da continuação da linha anterior. Por exemplo:

```

hp_req <= (x0_hp_req or t0_hp_req or x1_hp_req or t1_hp_req or
          s0_hp_req or s1_hp_req or t2_hp_req);

```

- Usar um layout tabular, com uma declaração por linha. Isto facilita a inserção e remoção de declarações. Por exemplo:

```

SIGNAL count1      : INTEGER;
SIGNAL count2      : INTEGER;
SIGNAL c           : INTEGER;

```

F) Declarações de portas e mapeamentos

- Declarar as portas em uma ordem lógica e manter essa ordem consistente em todo o projeto.
- Declarar as portas, uma por linha, com comentários para cada uma delas, de preferência na mesma linha.
- Declarar primeiro as portas de entrada e depois as de saída. Declarar as portas na ordem: *clocks*, *resets*, *enables*, outros sinais de controle e por último sinais de dados e de endereços.
- Sempre usar mapeamento explícito para portas e genéricos, usando associação por nome e não associação posicional.
- Deixar uma linha em branco entre mapeamentos de portas de entrada e de saída.

3.1.2 Codificação para Portabilidade

Outra questão importante quando se projeta pensando no reuso de componentes, diz respeito à portabilidade. Para que se desenvolva um componente portátil é preciso que se crie um código independente de tecnologia, compatível com várias ferramentas de simulação e facilmente traduzível de VHDL para Verilog (ou vice-versa).

A) Usar somente tipos do padrão IEEE

- Pode-se criar tipos e subtipos, mas todos devem ser baseados nos tipos do padrão IEEE.

- Usar de preferência *STD_LOGIC* ao invés de *STD_ULOGIC* e *STD_LOGIC_VECTOR* ao invés de *STD_ULOGIC_VECTOR*. O tipo *LOGIC* fornece funções de resolução necessárias em barramentos *tristate*. O tipo *ULOGIC* não as fornece. Além disso, o tipo *LOGIC* é melhor suportado pelas ferramentas de CAD.
- Padronizar a utilização de *LOGIC* ou *ULOGIC* é ainda mais importante do que a escolha do tipo a ser utilizado.

B) Não usar apenas valores numéricos em declarações de vetores. Por exemplo, em vez de usar:

```
SIGNAL data_reg : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

Usar uma constante:

```
SIGNAL data_reg : STD_LOGIC_VECTOR (bus_size DOWNTO 0);
```

Isto facilita a adaptação do circuito a diferentes larguras de barramentos.

C) Usar bibliotecas independentes de tecnologia, como a *DesignWare Foundation Library* [SYN01b], que contêm arquiteturas otimizadas para componentes aritméticos como somadores, multiplicadores, comparadores e incrementadores/decrementadores.

D) Não usar expressões *GENERATE* ou *BLOCK*, de modo a facilitar a tradução da descrição VHDL para Verilog. Não existem construções similares em Verilog.

3.1.3 Codificação para Síntese

Existem algumas regras de codificação para a questão de síntese. Pode-se criar um código que atinja melhores tempos de compilação e melhores resultados na síntese, incluindo: (i) testabilidade, (ii) desempenho, (iii) simplificação da análise de temporização estática e (iv) comportamente em nível de portas-lógicas que corresponde ao código RTL original.

A) Inferir Registradores – Registradores (flip-flops) são os mecanismos mais utilizados em lógica sequencial. Para manter a consistência e assegurar a síntese correta, deve-se utilizar alguns padrões, de forma a inferir registradores independentes de tecnologia.

- Usar o sinal de *reset* do sistema para inicializar sinais sincronizados por registradores. Exemplo:

```
--process with asynchronous reset
EXA_PROC: PROCESS(clk, reset)
BEGIN
    IF rst = '1' THEN
        ...
    ELSIF (clk`EVENT AND clk = '1') THEN
        ...
    END IF;
END PROCESS EXA_PROC
```

B) Evitar *Latches* – O problema do uso de *latches* deve-se ao fato da necessidade de criar uma estrutura de armazenamento de uma variável que será utilizada posteriormente,

porém sem explicitar qual o sinal de controle para que este armazenamento ocorra. Logo, a ferramenta infere o sinal de controle, podendo este não ser o desejado pelo projetista.

Exemplo de trecho de código VHDL que infere um latch devido à ausência da condição *ELSE* na expressão *IF*:

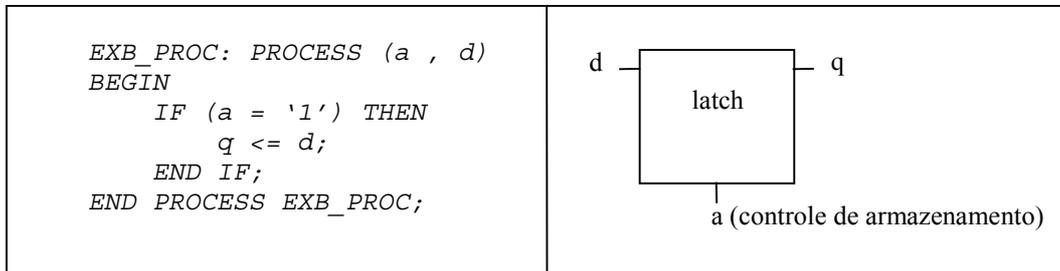


Figura 19 – Latch inferido devido à ausência da condição *ELSE* na expressão *IF*.

Exemplo de trecho de código VHDL que infere um latch devido ao fato de que a saída 'z' não está associada à condição *WHEN OTHERS*:

```

EXC_PROC: PROCESS (c)
BEGIN
  CASE c IS
    WHEN '0' => q <= '1'; z <= '0';
    WHEN others => q <= '0';
  END CASE;
END PROCESS EXB_PROC;

```

C) Especificar a Lista Completa de Sensitividade

- Incluir uma lista de sensibilidade completa em cada processo. Se esta lista não for utilizada, o comportamento do projeto antes da síntese pode diferir do comportamento após a síntese (Figura 20). Alguns compiladores detectam a ausência de sinais na lista, emitindo avisos de erro.

Exemplo:

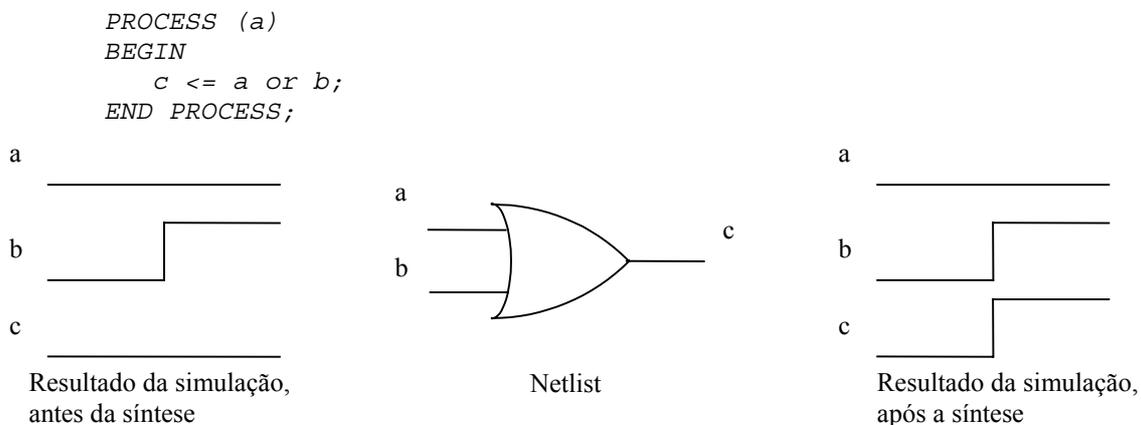


Figura 20 – Comportamento do projeto difere antes e depois da síntese.

D) *SIGNAL* VS. *VARIABLE* - Na simulação VHDL, os sinais recebem valores no próximo ciclo

de execução, enquanto que as variáveis têm efeito imediato.

- Na escrita de um código sintetizável, aconselha-se a utilização de sinais ao invés de variáveis, a fim de assegurar que o comportamento da simulação do projeto antes da síntese será equivalente ao da *netlist* após a síntese.

E) Expressões *CASE* vs. expressões *IF-THEN-ELSE* – Em VHDL, uma expressão *CASE* infere um nível simples de multiplexação, enquanto que uma expressão *IF-THEN-ELSE* infere uma combinação de multiplexadores em cascata, codificados com prioridade. As Figuras 17 e 18 mostram, respectivamente, um circuito descrito com expressões *IF-THEN-ELSE* e outro descrito com a expressão *CASE*.

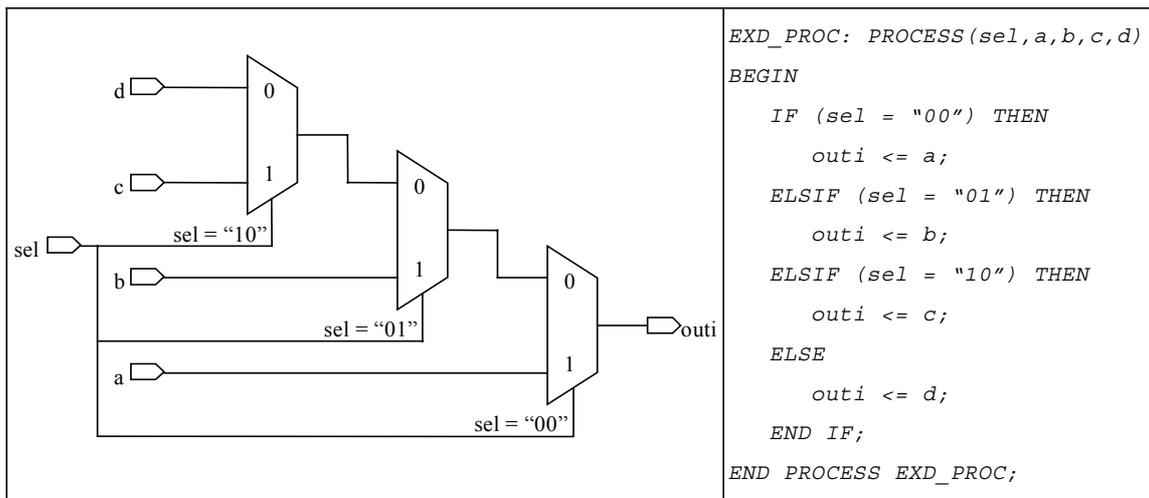


Figura 21 – Descrição usando *IF-THEN-ELSE*.

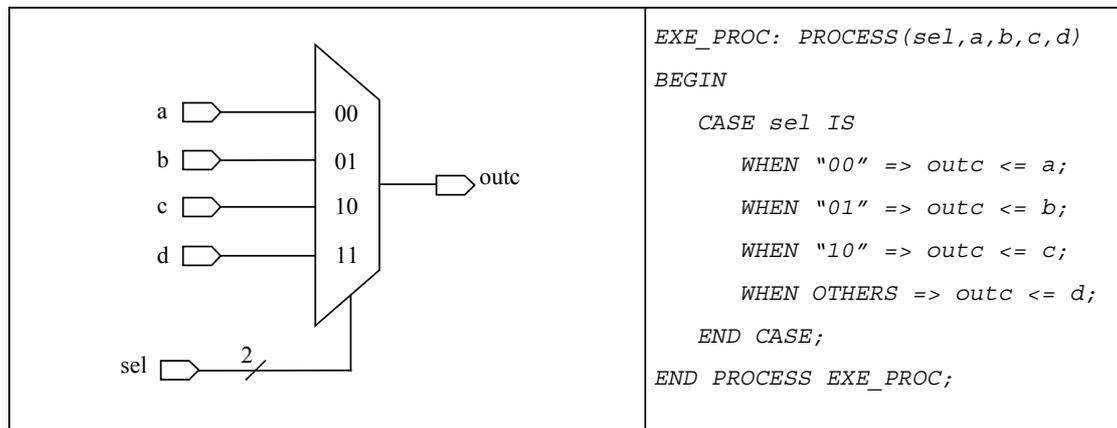


Figura 22 - Descrição usando *CASE*.

- Se uma estrutura codificada com prioridade não for necessária, recomenda-se o uso da expressão *CASE* ao invés de expressões *IF-THEN-ELSE*.
- Uma atribuição condicional também pode ser usada para inferir um multiplexador. Em grandes multiplexadores, na maioria dos simuladores a expressão *CASE* executa mais rápido do que uma atribuição condicional.

Exemplo de atribuição condicional para inferir um multiplexador:

```

z1 <= a WHEN sel_a = '1' ELSE
      b WHEN sel_b = '1' ELSE
      c;

```

F) Máquinas de Estado

- Separar a descrição da máquina de estados em dois processos, um para a lógica combinacional e outro para a lógica seqüencial.
- Criar um tipo enumerado para o vetor de estados.
- Atribuir um estado *default* para a máquina de estados, de forma a garantir que a máquina partirá do estado ocioso, caso nenhuma condição ocorra para colocá-la em um estado específico. Em VHDL, esta atribuição é feita pela condição *WHEN OTHERS*.

Exemplo:

```

ENTITY fsm IS
  PORT(
    rst      : IN STD_LOGIC;
    clock    : IN STD_LOGIC;
    x        : IN STD_LOGIC;
    z        : OUT STD_LOGIC );
END fsm;

ARCHITECTURE rtl OF fsm IS
  TYPE state IS ( STATE_0, STATE_1, STATE_2 );
  SIGNAL current_state, next_state : state;
BEGIN

  -- combinational process calculates next state
  COMB_PROC: PROCESS ( x, current_state )
  BEGIN
    CASE ( current_state ) IS
      WHEN STATE_0 =>
        z <= '0';
        IF x = '0' THEN
          next_state <= STATE_0;
        ELSE
          next_state <= STATE_1;
        WHEN STATE_1 =>
          z <= '0';
          IF x = '0' THEN
            next_state <= STATE_1;
          ELSE
            next_state <= STATE_2;
        WHEN STATE_2 =>
          z <= '0';
          IF x = '0' THEN
            next_state <= STATE_2;
          ELSE
            next_state <= STATE_0;
        WHEN OTHERS =>
          next_state <= STATE_0;
        END CASE;
    END PROCESS COMB_PROC;

  --synchronous process updates current state

```

```

SYNCH_PROC: PROCESS ( rst, clock )
BEGIN
  IF ( rst = '1' ) THEN
    current_state <= STATE_0;
  ELSIF ( clock'EVENT AND clock = '1' ) THEN
    current_state <= next_state;
  END IF;
END PROCESS SYNCH_PROC;
END rtl;

```

3.2 Documentação do Core

Uma das questões mais importantes quando se trata de reuso de *cores* é ter uma boa documentação. Esta documentação deve conter todas as informações que o usuário precisa para manusear o *core* e integrá-lo ao seu sistema. Primeiramente, deve-se definir um conjunto de documentos com formato uniforme. Dois tipos diferentes de informação podem ser diferenciados: (1) parte funcional, descrevendo funcionalidade, arquitetura e interfaces, e (2) parte relativa à implementação, descrevendo a estrutura de dados, a maneira como o *core* foi projetado e como ele pode ser verificado e adaptado ao sistema do usuário [HAA99]. Assim, a documentação pode conter as seguintes informações:

1. Parte funcional:
 - 1.1. Visão Geral das Características e da Funcionalidade – descrevendo as características principais e mostrando uma brevemente a funcionalidade.
 - 1.2. Descrição da Interface – lista todos os sinais e registradores acessíveis e descreve suas funções e possíveis configurações.
 - 1.3. Guia de Programação – descreve os modos de operação e as funções do *core*, e também como programá-lo e configurá-lo.
 - 1.4. Estrutura de dados – fornece informações sobre a base de dados do *core*. Isto inclui todas as informações necessárias para manipular a base de dados (estrutura de diretórios, estrutura do projeto, configuração, etc.).
 - 1.5. Modelo Comportamental – contém todas as informações necessárias para entender, compilar e utilizar o modelo.
2. Parte relativa à implementação:
 - 2.1. Informações da Implementação – fornece informações sobre a metodologia de projeto e síntese. Inclui, por exemplo, informações sobre a distribuição de clock, RAMs internas e partes assíncronas do projeto. Inclui informações sobre como o *core* pode ser mapeado para a tecnologia alvo e também informa características do projeto em termos de frequência, tamanho e potência.
 - 2.2. Informações da Verificação – descreve a metodologia utilizada para verificar o projeto. Fornece informações sobre os casos de teste e como estes casos podem ser criados.

A reutilização de componentes não é somente a transferência do código reutilizável, mas sim a transferência do conhecimento para manusear a funcionalidade dele. A qualidade da documentação é o ponto-chave para essa transferência de conhecimento.

3.3 Verificação do Core

O objetivo da verificação do *core* é assegurar que o mesmo está correto tanto na funcionalidade quanto na temporização. O comportamento do *core* deve estar de acordo com a funcionalidade e temporização descritas na especificação funcional. A verificação é um dos maiores desafios no desenvolvimento de um projeto, principalmente quando se projeta um *core* para ser reutilizado.

A verificação deve garantir que o *core* não apresenta nenhum defeito, pois o mesmo pode ser usado para construir desde um simples video-game até uma aplicação crítica e complexa, como em uma missão aeroespacial. A ausência de defeitos deve ser garantida para todo tipo de configuração do *core*, com todos os valores possíveis aceitos por seus parâmetros. Além disso, a equipe de integração deve ser capaz de reutilizar os *testbenches* no nível do *core*, pois o *core* deve ser verificado tanto como um projeto isolado quanto no contexto da aplicação final.

3.3.1 Plano de Verificação

Devido à complexidade e escopo da verificação funcional, é essencial que um plano global de verificação funcional seja criado e seguido pela equipe de projeto. Definindo o plano de verificação logo no início do projeto, a equipe pode desenvolver o ambiente de verificação, incluindo *testbenches* e conjuntos de testes de verificação, também no início do ciclo de projeto. Ter uma definição clara dos critérios que o *core* deve obedecer ajuda a concentrar o esforço de verificação e ter uma noção mais exata de quando o *core* está pronto para ser utilizado.

Os benefícios específicos do desenvolvimento do plano de verificação no início do projeto incluem: (i) a criação deste plano faz com que os projetistas analisem as tarefas que consomem mais tempo antes de executá-las; (ii) a equipe pode concentrar os esforços nas áreas que a verificação é mais necessária; (iii) o esforço redundante pode ser minimizado; (iv) os projetistas podem compartilhar sua experiência e conhecimento acumulado com o resto da equipe; (v) o plano fornece um mecanismo formal para correlacionar as necessidades do projeto à testes específicos, garantindo a integridade da cobertura dos testes; (vi) as informações contidas no plano permitem que uma equipe de suporte à verificação desenvolva o ambiente de verificação em paralelo às tarefas de captura do projeto, executadas pela equipe principal de projetistas. Isto pode reduzir significativamente o tempo de projeto.

O ambiente de verificação é formado por um conjunto de componentes de *testbench*, como modelos funcionais de barramentos, monitores de barramento, modelos de memória e a interconexão entre estes componentes e o projeto a ser testado.

O plano de verificação deve ser completamente descrito, ou na especificação funcional do *core*, ou em um documento separado. Este documento será mudado à medida em que novas questões forem surgindo durante o projeto e que as estratégias forem refinadas. O plano deve incluir: (i) uma descrição da estratégia de teste, tanto no nível de sub-bloco quanto no nível mais elevado (*top level*); (ii) uma descrição detalhada do ambiente de simulação, incluindo o diagrama de blocos; (iii) a lista dos componentes do *testbench*, com suas características, e indicando se cada um deles pode ser comprado de terceiros ou precisam ser desenvolvidos; (iv) a lista de todas as ferramentas de verificação necessárias, incluindo simuladores e ferramentas de criação de *testbenches*; (v) uma lista de testes específicos, com os objetivos e cobertura e com o custo de projeto estimado para cada um; (vi) a análise dos pontos principais da especificação do *core*, identificando quais testes verificam cada um deles; (vii) a especificação da funcionalidade que será testada em nível de sub-bloco e da que será testada no nível do *core*; (viii) a especificação dos critérios que serão utilizados para determinar quando o processo de verificação estará completo.

3.3.2 Estratégia de Verificação

A verificação de um *core* consiste de três fases principais: (i) verificação de módulos individuais; (ii) verificação do *core* e (iii) prototipação.

O objetivo da primeira fase é atingir um nível bastante alto de cobertura dos testes no nível de módulos e, depois, concentrar a verificação em nível de *core*, testando suas interfaces e sua funcionalidade. Esta abordagem de verificação *bottom-up* (começando do nível mais baixo até o mais alto) é baseada no princípio de localidade. É mais fácil detectar e consertar erros em módulos pequenos do que em módulos grandes.

A abordagem tradicionalmente utilizada por projetistas difere da anterior pelo fato de não haver um teste exaustivo (ou o mais completo possível) dos módulos. Na abordagem tradicional determinados módulos servem de *testbench* para outros blocos, simplificando a etapa de geração de *testbenches*. O problema é que podem haver falhas em determinados módulos, acarretando geração de estímulos com falhas, o que invalida a verificação dos módulos em teste.

Outras vantagens de se utilizar o teste no nível de módulos são:

- A observação e o controle de nós internos torna-se mais difícil quanto maior é o circuito. Atingir uma cobertura completa é mais fácil com blocos menores.
- Fazer a depuração no nível do *core* pode ser muito mais difícil e consome mais tempo do que depurar em nível de módulos.

A abordagem de verificação *bottom-up*, como um modelo de desenvolvimento em cascata, não é verdadeiramente eficiente. Na prática, uma abordagem em espiral envolvendo iterações é a que realmente funciona. A equipe de projeto faz testes exaustivos no sub-bloco, atinge cerca de cem por cento de cobertura, integra o sub-bloco no *core*, e então faz a verificação do *core*. Inevitavelmente surgem erros, geralmente envolvendo as interfaces e interações entre os módulos. São então feitas as modificações necessárias nos projetos dos

módulos, seguidas de uma nova verificação e integração dos módulos, para que seja feita a verificação do *core* completo. O processo composto por várias iterações e refinamentos, até que se atinja o nível desejado de confiabilidade do circuito é chamado de “construção por correção” [KEA00].

Na realidade, esta abordagem garante uma segurança bastante alta, mas não de cem por cento de correção no funcionamento. A construção de um protótipo rápido do *core* é o que permite que a equipe de projeto possa testar o *core* em aplicações reais, aumentando sua confiabilidade e robustez. Esta necessidade de um protótipo do *core* é atendida pelo uso do dispositivos programáveis de alta densidade, como os FPGAs.

Em cada fase do projeto, a equipe deve decidir que tipo de testes serão utilizados e quais as ferramentas de verificação que serão necessárias para isso. Os tipos básicos de testes de verificação incluem:

Teste de adequação – Estes testes verificam se o projeto está de acordo com a especificação. Para um padrão da indústria, como uma interface PCI ou uma interface IEEE 1394, estes testes também verificam a adequação com a especificação publicada. Em todos os casos, a adequação do projeto com a especificação funcional é verificada da maneira mais completa possível.

Teste de casos extremos – Estes testes tentam encontrar situações complexas e casos extremos, em que o projeto provavelmente apresente falhas.

Teste randômico – Para muitos projetos, como um processador ou interfaces de barramento complexas, os testes randômicos podem ser úteis como complementos aos testes de adequação e de casos extremos. Estes são limitados à situação que os projetistas previram. Testes randômicos podem criar situações que os projetistas não previram e descobrir a maioria dos erros mais difíceis de serem detectados no projeto.

Teste de código real – Uma das etapas mais importantes na verificação de um projeto é testá-lo em uma aplicação real, com código real. Sempre existe a possibilidade de que a equipe de projeto entenda mal uma especificação, e acabe projetando e testando seu código com uma especificação errada.

Teste de regressão – À medida em que os testes são desenvolvidos, eles devem ser adicionados ao conjunto de testes de regressão. Um dos problemas mais típicos encontrados durante a verificação é que, quando se conserta um erro, outro pode ser descuidadamente introduzido. O conjunto de testes de regressão ajuda a verificar que em determinado ponto de referência, a funcionalidade continua sendo mantida à medida em que novas características são adicionadas, e que todos os erros, até aquele ponto, foram corrigidos.

3.3.3 Projeto do *Testbench*

O termo *testbench*, em VHDL ou Verilog, geralmente se refere ao código utilizado para criar uma seqüência predeterminada de entradas para um circuito e, opcionalmente, observar suas saídas. É comumente implementado, através das próprias linguagens VHDL ou

Verilog, podendo incluir arquivos externos ou rotinas C [BER00b].

O projeto do *testbench* difere dependendo da função do *core*. Por exemplo, o *testbench* em nível mais alto para um *core* de um microprocessador deve, tipicamente, executar programas de teste, enquanto que o *testbench* de um *core* de uma interface de barramento deve usar modelos funcionais de barramentos e monitores de barramentos para aplicar estímulos e analisar os resultados. Existem diferenças significativas entre o projeto do *testbench* de sub-bloco e o projeto do *testbench* no nível do *core*. Em ambos os casos, o mais importante é garantir que a cobertura de testes é adequada.

A) *Testbench* de Sub-bloco

Devido ao fato de que estes módulos quase nunca possuem interfaces bidirecionais, pode-se desenvolver um único *testbench* que gere um conjunto de entradas para as portas do sub-bloco e verifique as portas de saída do mesmo. Na maioria dos sistemas digitais, estas entradas não são aleatórias, mas sim um conjunto de transações que devem ocorrer em uma determinada porta (Figura 23).

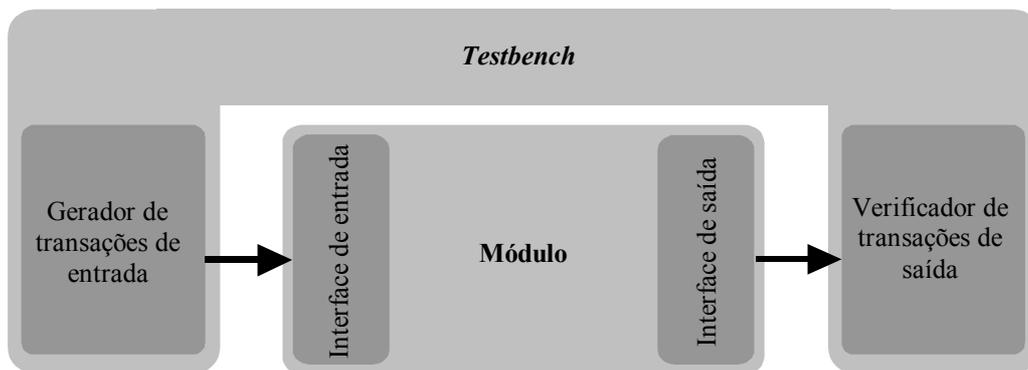


Figura 23 – *Testbench* de módulo.

Geração de Estímulos

Quando se projeta um sub-bloco, pode-se especificar os tipos de transações permitidas em determinada porta do sub-bloco. Por exemplo, a escrita em um registrador consiste em uma seqüência específica de dado, endereço e mudança de pinos de controle. Qualquer outra seqüência de ações é ilegal. No projeto do restante do *core*, é necessário garantir que nenhum outro sub-bloco que envie sinais para esta porta gere transações ilegais.

Uma vez definido o conjunto de transações legais para as portas de entrada, é necessário gerar seqüências destas transações com valores apropriados para os dados e endereços, para testar o sub-bloco. Analisa-se a funcionalidade do sub-bloco para determinar seqüências úteis, que servirão para verificar se o comportamento do sub-bloco está de acordo com a especificação, e então são feitos os testes de casos extremos. Após serem feitos todos os testes, deve-se executar uma ferramenta de cobertura de código, que fornece uma boa indicação da integridade do conjunto de testes.

Verificação de Saídas

A geração dos casos de teste é apenas a primeira parte da verificação. É necessário analisar as reações do projeto para verificar se ele está funcionando corretamente. Esta análise pode ser feita manualmente, através da monitoração das saídas em um visualizador de formas de onda. Porém, este processo tende a muitos erros, sendo necessário um verificador de saídas automático para o *testbench*. Este verificador deve ser exclusivo para o sub-bloco a ser testado, mas existem alguns aspectos comuns à maioria deles: (i) é possível verificar que somente transações legais são geradas pelas portas de saída do projeto; (ii) é possível verificar que transações específicas são respostas corretas às transações de entrada geradas.

B) *Testbench* do *Core*

É possível estender os conceitos utilizados em *testbenches* de módulos para *testbenches* usados para testar *cores*. Estando os módulos integrados dentro do *core*, constrói-se um *testbench* que, novamente, gera automaticamente as transações para as entradas do *core* e testa as transações das portas de saída. Existem algumas razões pelas quais deve-se desenvolver um *testbench* mais poderoso e bem documentado neste nível: (i) o projeto está mais complexo, exigindo cenários de teste também mais complexos, (ii) mais projetistas (talvez toda a equipe que desenvolveu os módulos) estarão trabalhando na verificação do *core* e (iii) o *testbench* será fornecido ao usuário juntamente com o *core*, para que ele possa testar o *core*.

C) Exemplo de *Testbench* para teste randômico

O trecho de código abaixo exemplifica um *testbench* randômico que obtém uma semente em um arquivo e, a partir dela, gera entradas aleatórias para o circuito a ser testado. Posteriormente, são calculados os valores esperados para as saídas do circuito, e estes são comparados aos resultados reais obtidos a partir do circuito.

```
...
use work.SeedFile_pack; -- function to read file and get seeds to random number generation
use work.RNG;          -- pseudo random number generation
...
-- processo para gerar e enviar N padrões de teste aleatório para o
-- circuito em teste
test: process
  variable found_error : boolean := false;
  variable seed        : natural;
  variable RndData     : RNG.Uniform;
  variable sumAux      : std_logic_vector(N downto 0);
  variable sum_expect  : std_logic_vector(N-1 downto 0);
  variable cout_expect : std_logic;

begin

  -- obtêm semente para geração de números aleatórios
  seed := SeedFile_Pack.GetSeed;
  RndData := RNG.InitUniform(seed, 0.0, real(2**N-1));

  for i in 1 to NOperations loop

    -- gera de forma aleatória as entradas do circuito
    RNG.Genrnd(RndData);
    a <= CONV_STD_LOGIC_VECTOR(integer(RndData.rnd), a'length);
    RNG.Genrnd(RndData);
    b <= CONV_STD_LOGIC_VECTOR(integer(RndData.rnd), b'length);
    RNG.Genrnd(RndData);
    cin <= SeedFile_Pack.Prob0(RndData.rnd, 70.0);
```

```

-- tempo para enviar o próximo padrão de teste
wait for 100 ns;

-- calcula os resultados esperados
sumAux      := ('0' & a) + ('0' & b) + cin;
cout_expect := sumAux(sumAux'left);
sum_expect  := sumAux(sumAux'left-1 downto 0);

-- verifica os resultados obtidos contra os esperados com msg de erro
if (sum /= sum_expect) then
    assert false
        report "Sum is " & to_string(sum)
            & ". Expected " & to_string(sum_expect);
    found_error := true;
end if;

if (cout /= cout_expect) then
    assert false
        report "Cout is " & to_char(cout) & ". "
            & "Expected value is " & to_char(cout_expect);
    found_error := true;
end if;

end loop;

-- ao final verifica se houveram ou não erros
assert not found_error
    report "There were ERRORS in the test."
    severity note;
assert found_error
    report "Test completed with no errors."
    severity note;
wait;
end process;

```

3.3.4 Code Coverage

Code coverage é uma técnica utilizada para exercitar todas as linhas do código e verificar o comportamento de todas as funções ou combinações possíveis destas [BER00b].

Uma ferramenta de *code coverage* funciona da seguinte maneira:

- Primeiro acrescenta-se *checkpoints* (pontos de verificação) em locais estratégicos do código-fonte para verificar se uma dada construção foi exercitada. O código dos *testbenches* não precisa receber pontos de verificação.
- Posteriormente, o *core* é simulado normalmente, usando todos os *testbenches* disponíveis. Os dados de todas as simulações são então coletados e armazenados em uma base de dados. A partir desta base de dados, são gerados os relatórios que determinam as medidas de cobertura do conjunto de verificação utilizado sobre o projeto.

Os relatórios mais comuns são sobre cobertura de instrução, caminho e expressão. A cobertura de instrução é também chamada cobertura de bloco, onde um bloco é uma seqüência de instruções que são executadas se uma única condição for verdadeira.

A) Cobertura de instrução

A cobertura de instrução (ou bloco) avalia quantas do total de linhas do código foram executadas pelo conjunto de verificação. Algumas ferramentas de *code coverage* fornecem

interfaces gráficas, permitindo que o usuário identifique facilmente as instruções que não foram executadas. A Figura 24 mostra um exemplo de relatório de cobertura de instrução, de forma gráfica, para um pequeno trecho de código. Este tipo de relatório varia de ferramenta para ferramenta.

```

 if (parity == ODD || parity == EVEN) begin
     tx <= compute_parity (data, parity);
     #(tx_time);
 end
 tx <= 1`b0;
 #(tx_time);
 if (stop_bits == 2) begin
     tx <= 1`b0;
     #(tx_time);
 end

```

Figura 24 – Relatório da cobertura de instrução.

O exemplo acima mostra que duas das oito linhas (25% do código) não foram executadas. Para que se obtenha uma cobertura de 100%, é preciso que se descubra quais condições são necessárias para que o resto do código seja executado e incluí-las no *testbench*. No caso apresentado, a paridade deve ser definida como ímpar (ODD) ou par (EVEN). Uma vez determinadas as condições, é preciso, primeiro, entender por que elas nunca aconteceram. Tratando-se de uma condição que não pode ocorrer nunca, o código em questão está efetivamente “morto”: nunca será executado. Deve, portanto, ser removido.

Como exemplo de ferramenta de *code coverage* disponível no mercado, pode-se citar a *Aldec Code Coverage* [ALD01a], que acompanha o ambiente de simulação *Active-HDL*. A Figura 25 mostra a janela de relatório da cobertura de código feita pela ferramenta.

↙ Numeração das linhas do código
 ↘ Número de vezes que cada linha é executada

Line	Count	Source
522	616	when S7 => if count<39 then state <= S7;
523	8	else state <= S8;
524	312	end if;
525	694	when S8 => if request(1)='1' then state <= S9;
526	690	else state <= S12;
527	692	end if;
528	2	when S9 => state <= S10;
529	2	when S10 => state <= S11;
530	154	when S11 => if count<39 then state <= S11;
531	2	else state <= S12;
532	78	end if;
533	692	when S12 => if request(0)='1' then state <= S13;
534	692	else state <= S0;
535	692	end if;
536	0	when S13 => state <= S14;
537	0	when S14 => state <= S15;
538	0	when S15 => if count<39 then state <= S15;
539	0	else state <= S0;
540	0	end if;
541	4000	end case;
542	8003	end if;
543	8003	end process;

Figura 25 – Ferramenta de *code coverage* da *Aldec*.

B) Cobertura de caminho

A cobertura de caminho avalia todos os caminhos possíveis pelos quais podem ser executados uma seqüência de instruções. O código mostrado na Figura 26 possui quatro caminhos possíveis para a execução, pois tanto o primeiro *IF* quanto o segundo podem ser *TRUE* ou *FALSE*. Para verificar todos os caminhos possíveis neste trecho de código, é necessário executá-lo sob todas combinações possíveis para as instruções *IF*: *FALSE-FALSE*, *FALSE-TRUE*, *TRUE-FALSE* e *TRUE-TRUE*.

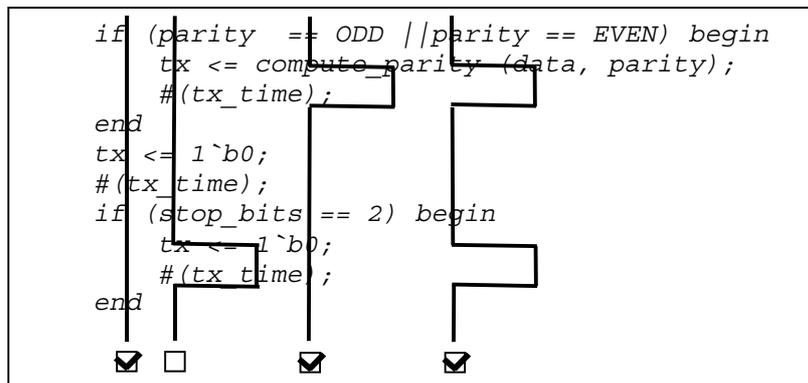


Figura 26 – Relatório da cobertura de caminho.

No exemplo acima, a cobertura de caminho foi de 75%, pois um dos caminhos não foi exercitado. Novamente, é necessário determinar as condições para que todos os caminhos sejam executados. Neste caso, para que este caminho seja executado, a paridade não deve ser definida nem como par, nem como ímpar, e o número de *stop bits* deve ser dois. Como no caso anterior, é importante questionar-se sobre a possibilidade de a condição não acontecer nunca.

C) Cobertura de expressões

No código da Figura 27, pode-se observar que existem duas condições independentes que podem causar a execução da condição *THEN* na primeira instrução *IF*: a paridade sendo par ou sendo ímpar. A cobertura de expressão avalia todos os caminhos pelos quais uma condição de controle pode ser *TRUE*.

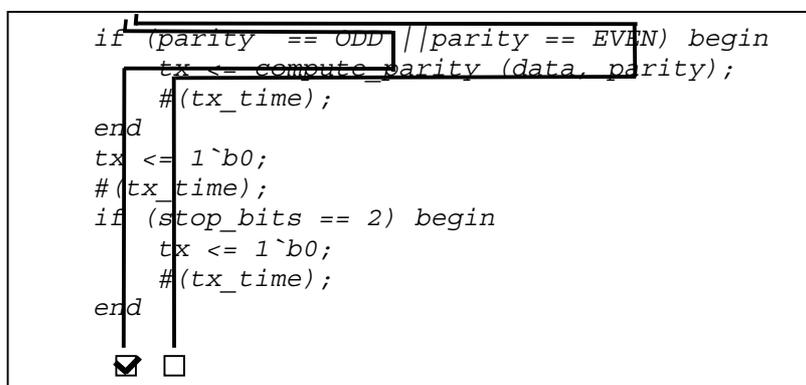


Figura 27 – Relatório da cobertura de expressão.

Como nos casos anteriores, é preciso entender por que um termo de controle de uma expressão não é exercitado. No exemplo anterior, nenhum *testbench* definiu a paridade como sendo par. Deve-se, então avaliar se esta condição nunca ocorrerá, e se o termo foi inserido no código equivocadamente.

3.4 Ambiente de Distribuição de Cores

A padronização da codificação VHDL (Seção 3.1) é necessária para que projetistas possam trabalhar de forma cooperativa. O mesmo refere-se à documentação e à metodologia de teste empregada.

Quando se desenvolve um *core*, gera-se uma grande quantidade de informações, incluindo o próprio *core*, sua documentação e seu *testbench*. Visando contribuir com técnicas de distribuição de *cores*, foi desenvolvido neste trabalho um ambiente de distribuição através da Internet, independente do formato do *core*. O objetivo em se criar tal ambiente é o de prover aos projetistas uma forma de distribuir seus projetos e de sistematizar a organização das informações contidas nos projetos.

Para que se manuseie adequadamente os projetos, é necessário ter uma estrutura de dados bem definida. Isto é ainda mais importante durante o desenvolvimento e a distribuição do *core*. A base de dados consiste em uma árvore de diretórios, onde ficam localizadas todas as informações dos projetos. A árvore de diretórios e os arquivos devem ser gerenciados por um sistema de controle de versões. Isto assegura um processo de desenvolvimento seguro, com consistência de dados.

3.4.1 Desenvolvimento da Ferramenta de Distribuição de Cores

Para atender ao objetivo de auxiliar na distribuição de cores, desenvolveu-se uma aplicação que cria uma *home page* de um dado projeto, permitindo: (i) organizar os diversos arquivos pertencentes ao projeto; (ii) controlar as versões desenvolvidas; (iii) distribuir sua propriedade intelectual; (iv) proteger a informação através de controle de acesso. Tal aplicação foi desenvolvida em linguagem Java.

A janela principal da ferramenta é mostrada na Figura 28. Nela, o projetista pode entrar com as informações do projeto, tais como nome do projeto, responsável, e-mail para contato, descrição do projeto e pinagem. A pinagem pode ser definida como de domínio público ou restrita à usuários autorizados.

Nesta mesma janela, encontram-se os botões “Docs”, “Fontes”, “Testbench”, “Arq. Auxiliares” e “Fer. Auxiliares” e “Permissões”. Os cinco primeiros botões abrem janelas secundárias, onde é feita a inserção dos arquivos pertencentes ao projeto (documentação, código-fonte, *testbenches*, arquivos e ferramentas auxiliares). O objetivo é permitir ao usuário organizar a informação pertencente ao projeto, assim como controlar as diferentes versões desenvolvidas ao longo do desenvolvimento do *core*.

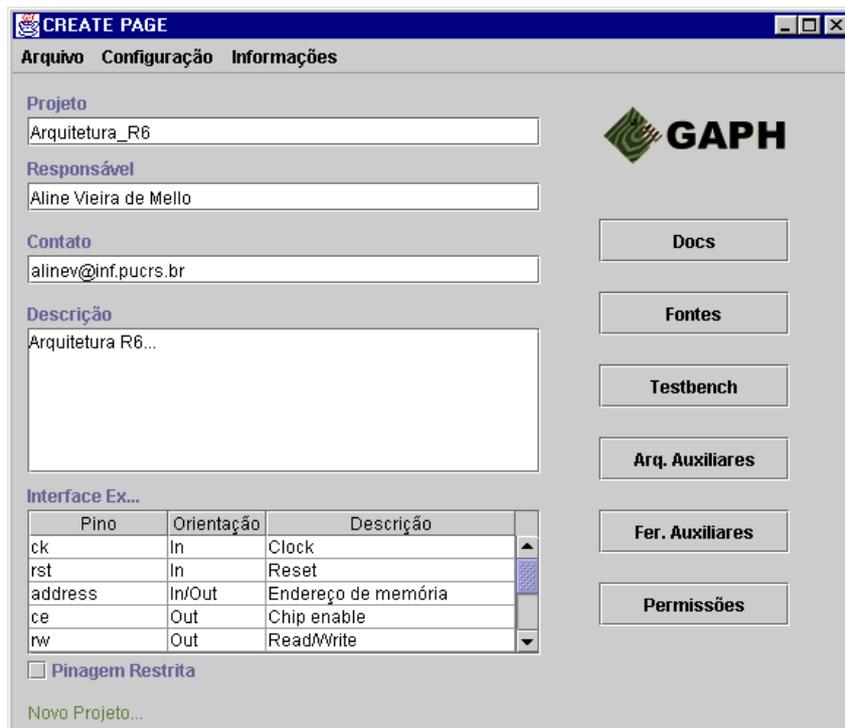


Figura 28 – Janela principal da ferramenta.

Para a entrada de arquivos-fonte e de ferramentas auxiliares é utilizado um controle de versões, pois deve haver a possibilidade de que se tenha várias versões do código-fonte e das ferramentas. A janela de controle de versões (mostrada na Figura 29) permite que o usuário possa controlar as versões existentes, bem como fornece acesso à janela de controle de arquivos (Figura 30).

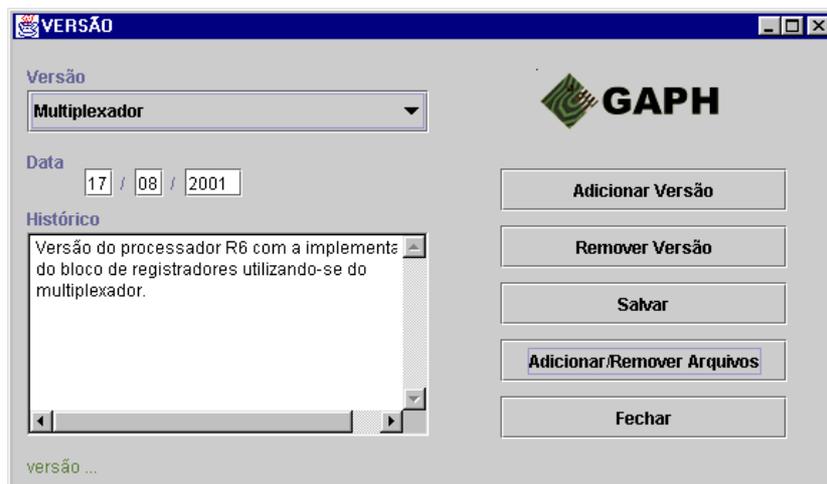


Figura 29 – Controle de versões.

Nas janelas de controle de arquivos, além de entrar com os nomes dos arquivos a serem inseridos na base de dados, deve-se fazer uma breve descrição dos mesmos, de maneira a facilitar a consulta. Também é possível remover arquivos de projetos e abrir arquivos para visualizar seu conteúdo. A interface também permite definir um arquivo como sendo restrito. Os arquivos restritos somente podem ser acessados por usuários autorizados. A autenticação de usuários é feita através do nome do usuário e senha.

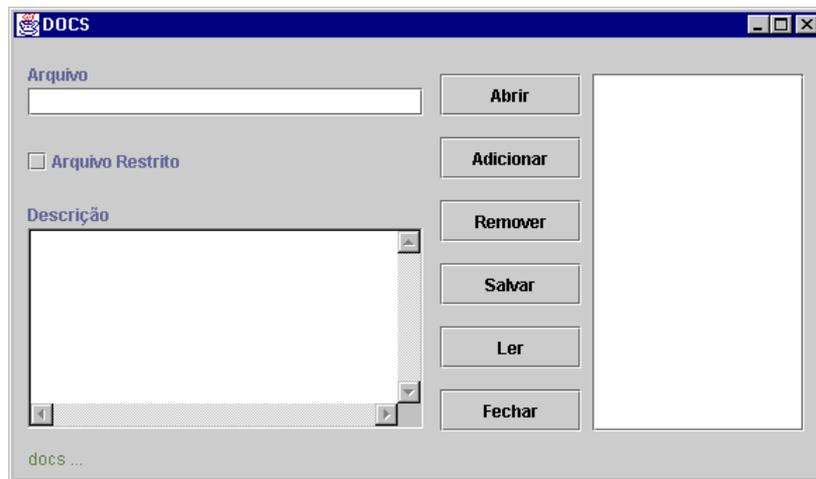


Figura 30 – Janela de controle de arquivos.

O sexto e último botão da janela principal (“Permissões”, na Figura 28) abre a interface do sistema de controle permissões de arquivos (Figura 31). Nela podemos visualizar duas listas. Na lista da esquerda encontram-se os arquivos de domínio público, enquanto que na lista da direita estão os arquivos restritos.

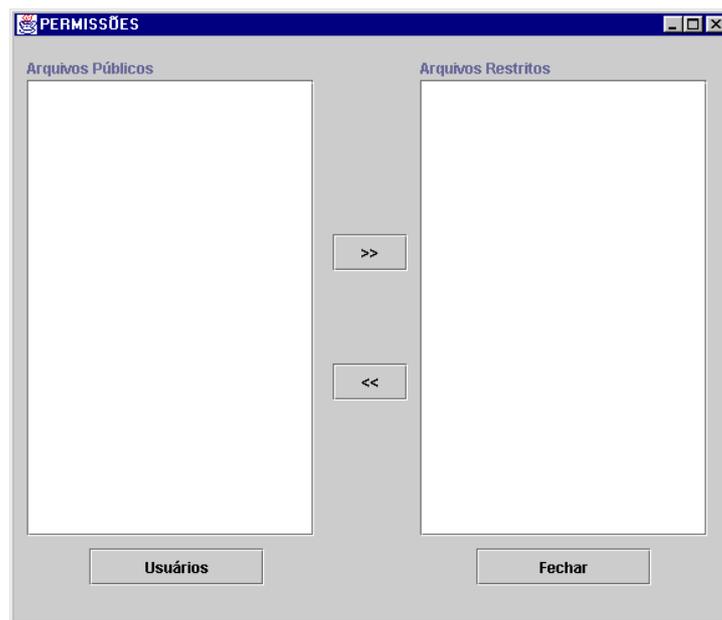


Figura 31 – Permissões de arquivos.

Nesta tela, além de definir um arquivo como sendo público ou restrito, têm-se acesso a uma outra tela, a de controle de usuários (Figura 32). Na esquerda há uma lista com todos os usuários cadastrados, e na direita somente os usuários com acesso aos arquivos restritos. O funcionamento do controle de acesso será discutido com maiores detalhes posteriormente.

Para cada projeto é criado um diretório raiz com o nome do projeto, no caminho configurado pelo usuário. Dentro deste diretório é armazenada a página HTML do projeto, juntamente com os subdiretórios *public* e *restrict* que contêm os arquivos de domínio público e de acesso restrito, respectivamente.

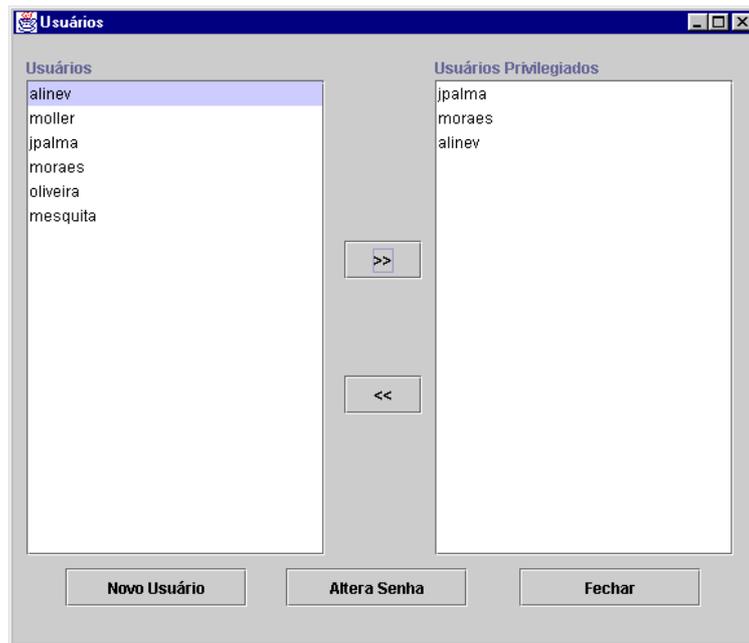


Figura 32 – Sistema de controle de acessos.

Dentro dos subdiretórios *public* e *restrict* também são criados outros subdiretórios que correspondem às versões dos arquivos-fonte e das ferramentas auxiliares. Por exemplo, a Figura 33 mostra o diretório *public* que contém os subdiretórios das ferramentas auxiliares “Montador” e “Simulador”, e das versões “Multiplexador” e “Tri-State” do projeto *Processador_R6*, que possuem arquivos de acesso público.

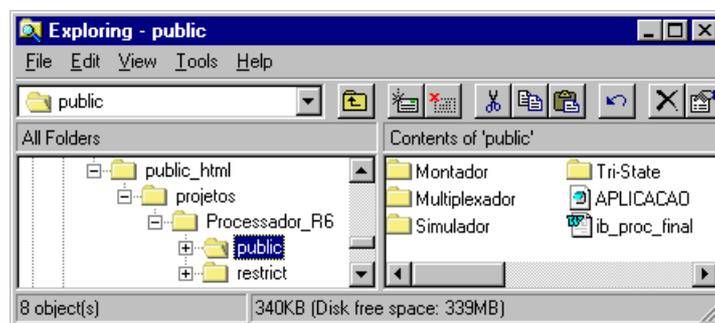


Figura 33 – Diretório *public* do projeto *Processador_R6*.

Se as ferramentas auxiliares ou versões possuírem arquivos com acesso público e também arquivos com acesso restrito, dois subdiretórios com o nome da ferramenta e/ou versão são criados, um no diretório *public* e outro no diretório *restrict*.

O controle de acesso é feito através de dois arquivos: **.htaccess** e **.htpasswd** [APA01]. O arquivo **.htaccess** deve ser inserido no diretório que contém os arquivos restritos. Ele é constituído por um apontador para o arquivo **.htpasswd** e por uma lista contendo os *usernames* dos usuários autorizados a acessar este diretório, como mostra a Figura 34. Cada diretório restrito tem seu próprio arquivo **.htaccess**, porém, todos devem apontar para um único arquivo **.htpasswd**, que possui a lista com os *usernames* de todos os usuários cadastrados e suas respectivas senhas criptografadas. A Figura 35 exemplifica a estrutura de um arquivo **.htpasswd**.

```
AuthUserFile /home3/gaph/alinev/public_html/.htpasswd
AuthName Processador_R6
AuthType Basic
<Limit GET>
require user moraes
require user laugusto
require user alinev
</Limit>
```

Figura 34 – Estrutura de um arquivo **.htaccess**.

No exemplo acima, o arquivo **.htaccess** aponta para o arquivo **.htpasswd**, que encontra-se no diretório `/home3/gaph/alinev/public_html/`, o nome do projeto é `Processador_R6` e os usuários com acesso ao diretório são: `moraes`, `laugusto` e `alinev`.

```
aline:$apr1$ty1.....$rAGk4dR8BCsQ530qlFCp1
w:$apr1$GB2.....$Qhj838FuCD6ibbVq/RrAK/
wi:$apr1$sK2.....$ysJ6vBM1HSY11.3mpRAUE0
```

Figura 35 – Arquivo **.htpasswd**.

O arquivo **.htpasswd** é gerado pelo programa executável `Htpasswd` pertencente ao servidor Apache. Esse programa é utilizado pelo sistema quando a senha de um usuário é alterada ou quando um novo usuário é incluído.

Para alterar a senha de um usuário ou incluir um novo usuário a seguinte linha de comando é executada automaticamente pelo sistema:

```
Htpasswd -b .htpasswd usuário senha
```

Os arquivos, **.htaccess** e **.htpasswd**, são considerados ocultos pelo servidor Web. Logo, eles não são vistos via FTP ou via HTTP.

O controle de acesso funciona da seguinte maneira: quando o navegador tenta acessar um arquivo, o servidor Apache verifica se existe, no diretório onde tal arquivo se encontra, um arquivo **.htaccess**. Caso não exista, o acesso é imediato. Caso exista, significa que aquele diretório é restrito, e então o servidor abre uma janela de autenticação de usuário. Após o usuário entrar com seu nome, o sistema verifica se o mesmo se encontra na lista de usuários autorizados do arquivo **.htaccess** deste diretório. Se o nome estiver na lista, o sistema procura pelo arquivo **.htpasswd** indicado pelo arquivo **.htaccess** e compara a senha digitada com a que consta neste arquivo. Só então este usuário poderá acessar os dados restritos.

3.4.2 Exemplo: Página da Arquitetura R6

Esta Seção ilustra o resultado da criação da página de um projeto, feita através da ferramenta apresentada na Seção anterior. Como exemplo, foi utilizada a página do Processador R6, um processador RISC desenvolvido pelo GAPH. A Figura 36 mostra a estrutura da página gerada pela ferramenta. No topo encontra-se o nome do projeto, seguido dos links para as seções “Descrição”, “Interface Externa”, “Download” e “Contato”, que encontram-se logo abaixo na página.

Arquitetura_R6

- [Descrição](#)
- [Interface Externa](#)
- [Download](#)
- [Contato](#)

Descrição

A arquitetura, denominada R6, é uma organização Von Neumann, Load/Store, com CPI igual a 2, barramento de dados e endereços de 16 bits. Esta arquitetura é praticamente uma máquina RISC, faltando contudo algumas características gerais de máquina RISC, tal como pipeline e módulos de entrada/saída, como tratamento de interrupções. Estas deficiências devem-se ao fato desta arquitetura ter sido originalmente desenvolvida visando o ensino de Organização de Computadores em cursos de graduação

Interface Externa

Pino	Orientação	Descrição
ck	In	Clock
rst	In	Reset
data	In/Out	Barramento de dados
address	Out	Barramento de endereço
ce	Out	Chip enable
rw	Out	Read/Write

Download

Versões

Versão	Acesso	Arquivo	Descrição
Multiplexador - 23/11/2001 Versão da arquitetura R6 com a implementação do bloco de registradores utilizando multiplexadores.	Public	r6mux.vhd	implementação da arquitetura R6 com o bloco de registradores implementado na forma de multiplexador.
Tri-States - 23/11/2001 Versão da arquitetura R6 com a implementação do bloco de registradores utilizando tri-states.	Public	R6tri_states.vhd	Implementação da arquitetura R6 com o bloco de registradores utilizando tri-states.
Mista - 23/11/2001 Versão da arquitetura R6 com a implementação do bloco de registradores utilizando multiplexadores e tri-states.	Public	r6misto.vhd	Implementação da arquitetura R6 com o bloco de registradores utilizando multiplexadores e tri-states
Array - 23/11/2001 Versão da arquitetura R6 com a implementação do bloco de registradores utilizando arrays.	Public	r6array.vhd	Implementação da arquitetura R6 com o bloco de registradores utilizando arrays.

Arquivos Complementares

Tipo	Descrição	Acesso	Arquivo
Documento	Arquivo texto que descreve a arquitetura.	Publico	arq_r6.doc
Testbench	Testbench que realiza a simulação da arquitetura R6.	Restrito	R6_tb.vhd
Auxiliar	Programa para simulação VHDL da arquitetura R6.	Publico	SUBRT.TXT

Contato

Responsável: Aline Vieira de Mello
E-mail: alinev@inf.pucrs.br

Figura 36 – Página gerada pela ferramenta.

As seções “Descrição” e “Interface Externa” refletem as entradas feitas na janela principal da ferramenta, mostrada na Figura 28. A área de “Download” é formada por duas tabelas. A primeira tabela contém as versões do projeto, descrição de cada versão, arquivos VHDL que compõem cada versão, com suas descrições, indicação de acesso público ou restrito, e *links* para download dos arquivos. A segunda tabela (com arquivos complementares), contém a indicação do tipo do arquivo (documentação, *testbench*, ferramentas e arquivos auxiliares), descrição, indicação de acesso e link para *download*.

Como já mencionado no início da Seção anterior, esta ferramenta auxilia o projetista de hardware na organização dos dados pertencentes a seus projeto, como por exemplo, arquivos de descrição de hardware, documentação e ferramentas auxiliares.

Além da organização de dados e controle de versões, a ferramenta também cria uma página HTML para cada projeto, permitindo a distribuição de projetos com proteção de propriedade intelectual, caso seja necessário.

4 Reconfiguração e Distribuição de *Hard Cores*

A segunda contribuição deste trabalho é a reconfiguração e a distribuição de *hard cores* na forma de arquivos binários de configuração de dispositivos FPGAs (*bitstreams*).

A motivação para que se distribua (gratuitamente ou não) *hard cores* é permitir que outros projetistas possam utilizar módulos de hardware já sintetizados, prontos para serem integrados a novos projetos. Não só outros projetistas podem ser beneficiados por esta abordagem, mas também usuários comuns, mesmo que estes não tenham conhecimento algum sobre FPGAs ou qualquer outro tipo de hardware programável. Digamos que um usuário tenha em seu computador uma placa aceleradora com um FPGA, e que em um determinado momento ele precise de um hardware para acelerar um processo gráfico ou qualquer outra função crítica em tempo de processamento. Este usuário poderia conectar-se à Internet, fazer o download do módulo de hardware que ele deseja, e carregar o módulo no FPGA da sua placa, tendo assim o acelerador que necessita.

Entretanto, a distribuição de propriedade intelectual na forma de módulos já sintetizados (*hard cores*) para FPGAs somente é útil se houver a possibilidade de se "encaixar" novos *cores* em um sistema sendo executado em um FPGA sem interromper ou afetar o funcionamento do mesmo. Para que esta nova abordagem seja factível de ser implementada, é necessário preencher uma série de requisitos:

- (i) FPGAs com arquitetura regular e disponibilidade de reconfiguração parcial. Alguns FPGAs, como por exemplo os da família *Virtex*, possuem arquitetura interna baseada em colunas, com endereçamento individual. Este fato permite que o dispositivo possa ter sua configuração modificada dinâmica e parcialmente.
- (ii) Existência de ferramentas que permitam fixar a forma e a posição relativa dos *cores* no dispositivo programável. Exemplo de ferramenta que permite fixar a posição dos circuitos no interior do FPGA é o *Floorplanner* (*Foundation – Xilinx*).
- (iii) Existência de ferramentas que permitam a geração de *bitstreams* parciais. O conjunto de classes JBits [ROX00] permite teoricamente gerar um arquivo parcial, porém foi verificado pelos autores deste trabalho que o *bitstream* gerado é incorreto. Desta forma, não há ao conhecimento dos autores, ferramentas para geração e manipulação de arquivos parciais para dispositivos *Virtex*.
- (iv) Existência de ferramentas que permitam o download de *bitstreams* parciais.
- (v) Estrutura de conexão entre o *core* inserido e os demais *cores* já em operação no FPGA.
- (vi) “Virtualização” dos pinos de entrada e saída.

Os itens (iii), (iv), (v) e (vi) carecem hoje de ferramentas. Não se encontra na literatura referência a ferramentas que manipulam arquivos parciais de configuração.

O presente Capítulo tem por objetivo mostrar o desenvolvimento de uma ferramenta que permite a reconfiguração de parâmetros de um *hard core*. Desta forma, o usuário pode

buscar no ambiente de distribuição previamente descrito o *core* já sintetizado, não havendo a necessidade do uso de ferramentas de CAD para síntese, e configurá-lo conforme o seu ambiente de utilização. A geração do arquivo binário para reconfiguração pode ser total ou parcial, em função das ferramentas de download disponíveis.

Este Capítulo é organizado da seguinte forma. A Seção 4.1 revisa a arquitetura dos dispositivos programáveis *Virtex* que, como comentado anteriormente, é o dispositivo que permite reconfiguração parcial. A Seção 4.2 apresenta a ferramenta que permite fixar a área de um determinado *core* dentro do FPGA. A Seção 4.3 apresenta o desenvolvimento da ferramenta que permite reconfigurar parâmetros de um *hard core*, mediante o emprego das classes *JBits*. Finalmente, a Seção 4.4 estende o conceito de reconfiguração para reconfiguração parcial.

4.1 Revisão da Arquitetura Virtex

Existem atualmente no mercado duas famílias de FPGAs que suportam reconfiguração parcial: a família At40k [ATM00], desenvolvida pela empresa *Atmel*, e a família *Virtex* [XIL00], desenvolvida pela empresa *Xilinx*.

O presente trabalho utiliza a família *Virtex* por esta (i) estar disponível no ambiente de trabalho do autor, (ii) ter maior uso no mercado (iii) dispor de ferramentas de CAD para o projeto e (iv) permitir a implementação de circuitos complexos devido ao elevado número de portas lógicas.

Os FPGAs da família *Virtex* são compostos por uma matriz de blocos lógicos configuráveis (*Configurable Logic Blocks* – CLBs) cercados por blocos de entrada/saída (*Input/Output Blocks* – IOBs). Nas bordas da direita e esquerda estão localizadas as *Block RAMs* (BRAMs), como mostra a Figura 37.

As CLBs são módulos básicos que contém elementos para implementar portas lógicas, flip-flops e roteamento personalizáveis. Os IOBs permitem a comunicação dos sinais internos com dispositivos externos. As BRAMs permitem o armazenamento de dados com densidade mais alta que CLBs. Cada BRAM é um bloco configurável de 4096 bits. Com cada CLB também é possível implementar RAMs de 32-bit síncronas ou assíncronas.

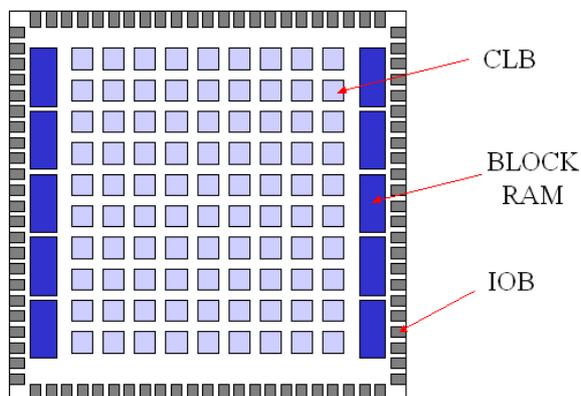


Figura 37 – Arquitetura *Virtex*.

Estas funções lógicas são configuradas através de um *bitstream* de configuração. Um *bitstream* é uma seqüência de *bits* de configuração, formado por comandos e dados de configuração. A estrutura do *bitstream* é mostrada na Figura 38. O número abaixo de cada campo indica a quantidade de palavras de 32 bits para configurar o dispositivo XCV300. A quantidade total de palavras é 54.744 , sendo o número de bits de configuração 1.751.808.

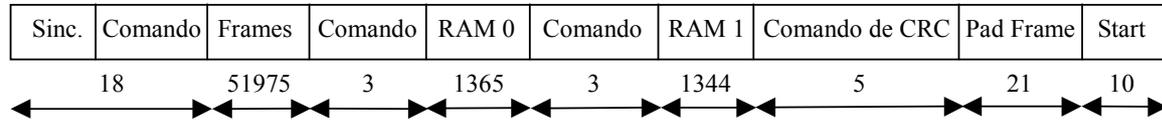


Figura 38 – Estrutura do *bitstream* de configuração para o dispositivo XCV300.

4.1.1 CLBs e IOBs

Cada CLB contém dois *slices*, como mostra a Figura 39. Cada *slice* é formado por duas LUTs (*Look-Up-Tables*) de quatro entradas, dois flip-flops D e lógica de propagação rápida de “vai-um” aritméticos.

Os elementos básicos dos *slices* são a LUT F, a LUT G, o flip-flop X e o flip-flop Y. As LUTs podem ser usadas para implementar portas lógicas ou pequenas memórias. Os flip-flops podem ser usados para criar lógica sequencial. Os *slices* também possuem multiplexadores internos para controlar a conectividade dos recursos internos (não mostrados na Figura). Também em cada *slice* existe a lógica para implementar propagação rápida de “vai-um” em operadores aritméticos.

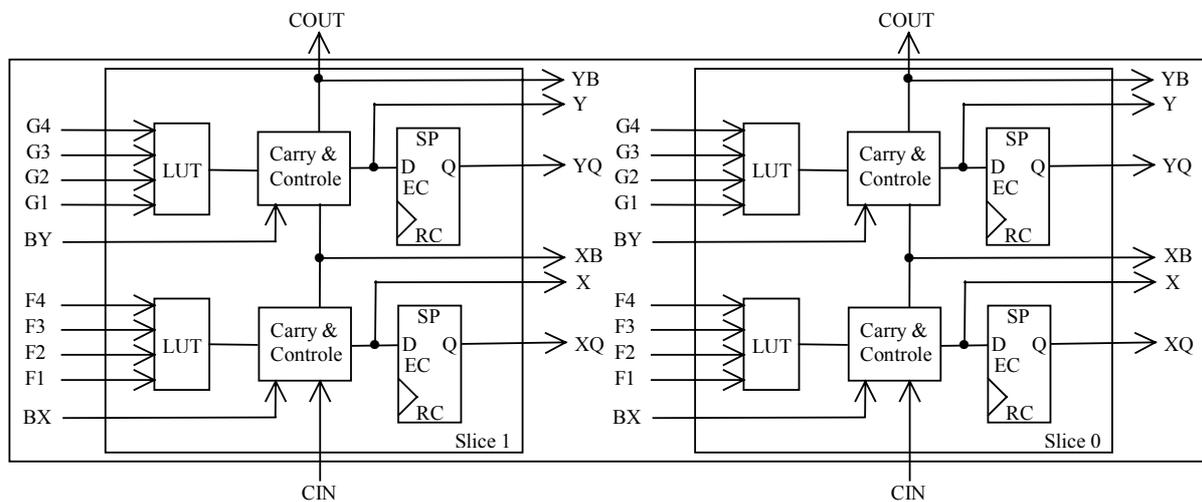


Figura 39 – Estrutura interna de uma CLB.

A Figura 40 ilustra a estrutura interna do dispositivo *Virtex*. A maior parte da área é utilizada por recursos de roteamento (linhas horizontais e verticais). Estes recursos de roteamento são conectados aos blocos lógicos por meio das *switch boxes*. As *switch boxes* por sua vez estão conectadas aos multiplexadores de entrada e saída, que se ligam às entradas e saídas dos *slices*. A razão de se utilizar estes multiplexadores é reduzir o número de conexões entre os *slices* e as *switch box*, pois caso não os utilizássemos, seria necessário inserir para

cada entrada/saída um número de chaves proporcional ao número de linhas de roteamento conectadas pela *switch box*. Ainda nesta Figura estão ilustrados os dois buffers *tri-state* da cada CLB.

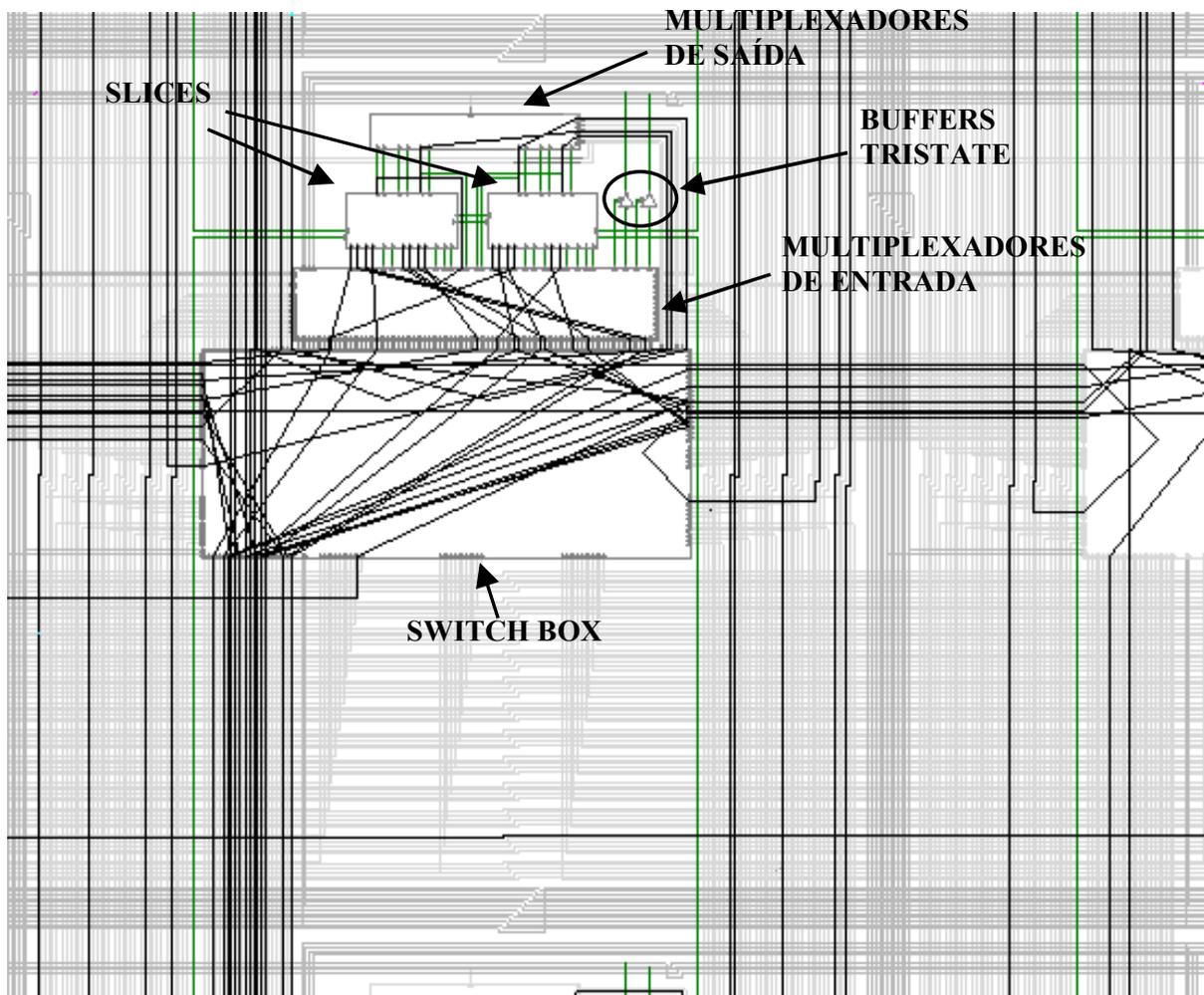


Figura 40 – Estrutura interna do dispositivo Virtex.

Cada coluna de CLBs tem dois IOBs acima e dois IOBs abaixo dela. Nas bordas direita e esquerda existem três IOBs por linha de CLBs.

4.1.2 Configuração do Dispositivo Virtex

A estrutura em colunas da arquitetura de FPGAs *Virtex* é mostrada na Figura 41. A unidade mínima de configuração é um *frame*, o qual corresponde à uma coluna de configuração, direcionada do topo à base do FPGA.

Uma coluna de CLBs contém 48 *frames* paralelos. Cada coluna de IOB, uma na esquerda e outra na direita possui 54 *frames* de dados de configuração. Cada *Block SelectRAM* tem 64 *frames* e os blocos de Interconexão de *SelectRAM* têm 27 *frames*. O bloco central é dedicado à distribuição de *clock*, contendo 8 *frames*.

A Figura 41 ilustra a distribuição dos *frames* sobre os recursos dos circuitos da família

O *Floorplanner* (gerador de planta-baixa) [XIL01f] é uma ferramenta gráfica de localização que permite ao projetista controlar a disposição do seu projeto em um dispositivo FPGA usando um paradigma de “arrastar e soltar” usando o mouse. A ferramenta mostra uma representação hierárquica do projeto na janela *Design Hierarchy*, usando estruturas e cores para distinguir os diferentes níveis. A janela *Design Nets* mostra todas as redes de interconexão pertencentes ao projeto. A janela *Placement* mostra a disposição atual da lógica no FPGA, e a janela *Floorplan* é onde o usuário (projetista) define a nova disposição do projeto no dispositivo. A Figura 43 mostra as janelas que compõem a interface da ferramenta *Floorplanner*.

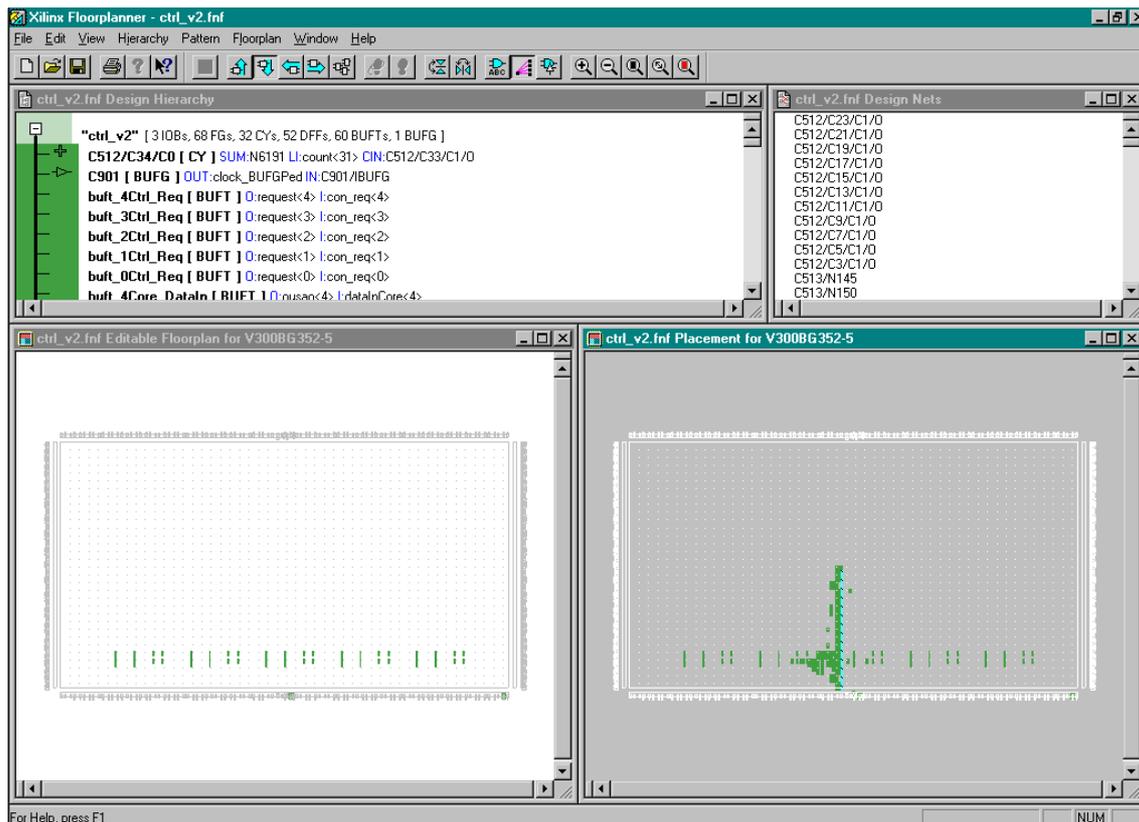


Figura 43 – Interface da ferramenta *Floorplanner*.

O usuário pode selecionar a parte da lógica que deseja na janela *Design Hierarchy* e colocá-la onde deseja no FPGA, representado na janela *Floorplan*.

Floorplanning é uma metodologia opcional utilizada para aumentar o desempenho e a densidade de um projeto que foi posicionado e interconectado automaticamente. Ela é especificamente direcionada para auxiliar os usuários que necessitam que certos detalhes de seu projeto sejam feitos à mão. Para isto, o usuário precisa conhecer os detalhes da arquitetura do dispositivo alvo. Para que se obtenha um *floorplanning* bem sucedido, é preciso um processo iterativo que pode levar muito tempo até que se possa superar o processo automático da ferramenta de síntese. O *Floorplanner* suporta todas as arquiteturas das famílias de FPGAs Spartan /-II, *Virtex* / -E / -II, e XC4000 da Xilinx.

4.2.1 Arquivos de Entrada

Para a compreensão dos fluxos de projeto, descritos posteriormente, é necessário apresentar os arquivos que são manipulados pela ferramenta de *floorplan*.

- **NCD** – Arquivo gerado pelo MAP (*Mapping*) ou pelo PAR (*Place and Route*). É usado pelo Floorplanner para gerar um projeto físico para o arquivo FNF, na hora de criar um novo floorplan.
- **NGD** – Arquivo gerado pelo NGDBuild. É usado pelo Floorplanner na hora de criar um novo floorplan, para correlacionar o projeto físico ao projeto lógico, quando gerar o arquivo FNF.
- **FNF** (*Floorplanner Netlist File*) – É a base de dados do Floorplanner. Nele são gravadas todas as restrições físicas mostradas na janela de *Floorplan*. Se o arquivo FNF é gerado usando um arquivo NCD já posicionado, as informações de posicionamento também são gravadas no arquivo FNF para serem usadas pela janela de posicionamento.
- **UCF** (*User Constraint File*) – O arquivo UCF especifica as restrições lógicas do projeto. Estas restrições afetam a maneira como a lógica será implementada no dispositivo alvo. A versão 3.1i do *Floorplanner* permite que se obtenha um arquivo UCF como uma das entradas para *Floorplanning*.

4.2.2 Arquivos de Saída

- **FNF** (Floorplanner Netlist File) – Mencionado anteriormente.
- **MFP** (Mapper Floorplan) – É gerado quando o arquivo FNF é salvo. O arquivo MFP é usado como entrada para o MAP, para transferir as restrições físicas impostas no *floorplanning* para a ferramenta de implementação automática.
- **UCF** – Após criar-se mais restrições graficamente no *Floorplanner*, é possível que se grave estas restrições no arquivo UCF.

4.2.3 Fluxos de Projeto com *Floorplanner*

O *Floorplanner* pode ser utilizado em cinco fluxos de projeto diferentes.

O fluxo mais comumente utilizado é executar um posicionamento e roteamento inicial (ferramentas MAP e PAR), e posteriormente utilizar a ferramenta *floorplan* para refinar a solução obtida. O usuário define o projeto usando uma ferramenta de captura de esquemático ou HDL. Após, é executada a síntese lógica e física, gerando-se um arquivo em formato *ncd*. Posteriormente, utiliza-se o *floorplanner* para restringir caminhos críticos ou ajustar o posicionamento automático. O arquivo resultante do *floorplanner*, *mfp*, é utilizado para a geração final do arquivo de configuração. Este fluxo pode ser visualizado na Figura 44.

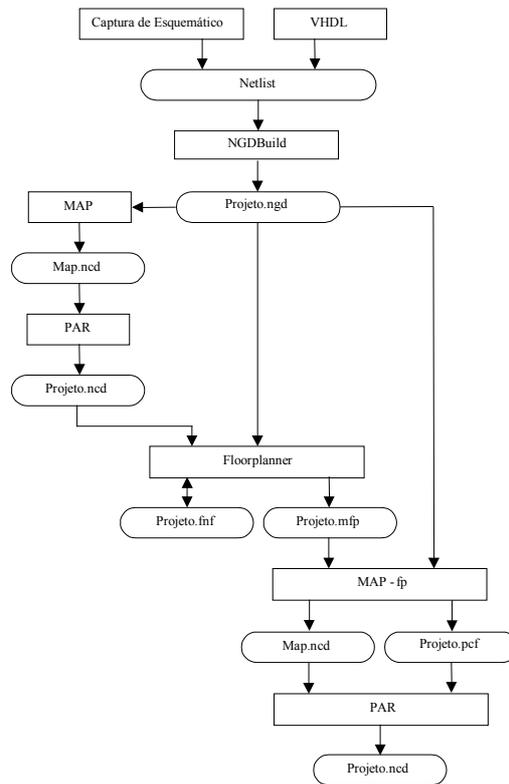


Figura 44 – Place and route, depois Floorplan.

O segundo fluxo de projeto é semelhante ao anterior, porém não há uma síntese física (posicionamento e roteamento) prévia. O programa MAP cria o arquivo de projeto físico (NCD). O *Floorplanner* é então usado para definir manualmente restrições de posicionamento. Feito isso, o MAP é novamente executado, desta vez com o PAR, para acomodar o projeto no dispositivo FPGA alvo, usando as restrições do *floorplanning*. A Figura 45 ilustra este fluxo. Este fluxo de projeto é recomendado para *cores*, pois evita que a primeira síntese física disperse os elementos pelo FPGA. O resultado é uma síntese física já guiada pelas restrições de área do *core* a ser sintetizado.

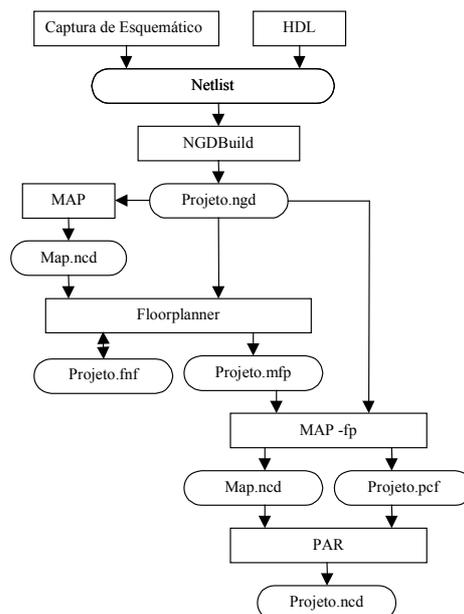


Figura 45 – Floorplan anterior ao place and route.

O terceiro fluxo de projeto é denominado de "*floorplanning* iterativo". É muito semelhante ao primeiro (Place and Route, depois Floorplanning). A diferença é que o usuário pode utilizar o Floorplanner em várias iterações, fazendo pequenas modificações no posicionamento feito pelo PAR. O laço Floorplanner-MAP-PAR é repetido até que se atinja o desempenho esperado (Figura 46).

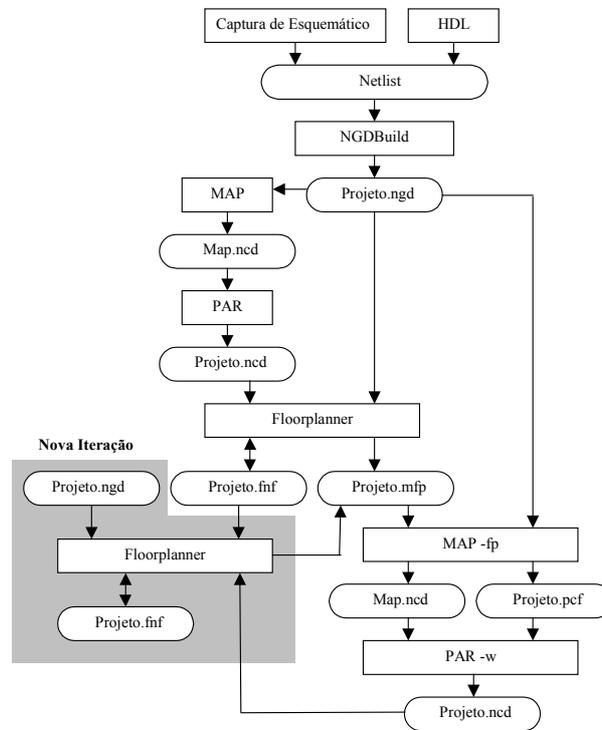


Figura 46 – Floorplan iterativo.

O quarto fluxo de projeto é denominado de "ampliação de projeto". É utilizado quando se deseja alterar um projeto previamente implementado em um FPGA. Este projeto pode ter sido implementado com ou sem a utilização do *Floorplanner*. O *Floorplanner* é utilizado para correlacionar a lógica do projeto e para ajustar as informações de restrições de acordo com as novas mudanças.

Finalmente, o quinto fluxo de projeto (Figura 47) permite a criação de restrições a partir do posicionamento de IOBs. Pode-se inserir restrições no arquivo UCF através do *Floorplanner* e implementar o projeto iterativamente, até atingir o posicionamento desejado. Para isso, é necessário apenas o arquivo NGD gerado previamente. No *Floorplanner*, o usuário manualmente cria restrições que serão automaticamente escritas no arquivo UCF. As novas restrições criadas pelo *Floorplanner* têm precedência sobre as já existentes no arquivo UCF. Posteriormente, o NGDBuild (ferramenta que converte uma descrição obtida da síntese lógica (formato EDIF ou XFF) em formato de síntese física (NGD)) é novamente executado, recebendo as restrições inseridas, seguido do MAP e do PAR.

Neste Capítulo utilizaremos esta ferramenta para fixar alguns elementos internos (LUT RAM) e pinos de entrada/saída (IOBs). Estes elementos também podem ter suas posições fixadas mediante a utilização do arquivo de restrições do usuário. No Capítulo 5 utilizaremos esta ferramenta para definir a forma e a posição dos *cores*.

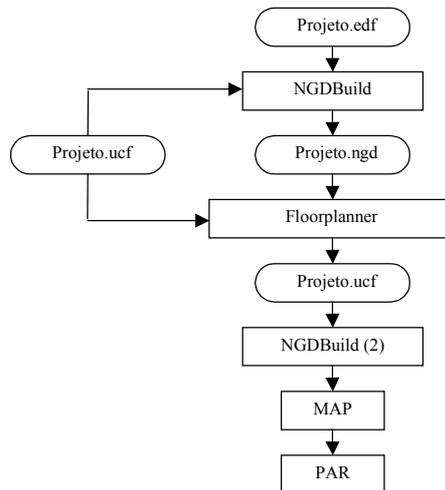


Figura 47 - Criação de restrições a partir do posicionamento de IOBs.

4.3 Ambiente de Reconfiguração de Hard Cores

Como estudo de caso, é apresentado aqui um reconfigurador total de *bitstreams*, que utiliza JBits [XIL01e]. JBits é constituído por um conjunto de classes Java que fornecem uma API (*Application Program Interface*) que permite manipular o *bitstream* da família de FPGAs *Virtex* [GUC00]. Com base neste conjunto de classes, desenvolveu-se um ambiente (em forma de *Applet*) para ser utilizado através da Internet, onde o usuário pode entrar com novos dados e alterar remotamente o conteúdo de um *bitstream*. Para isto, é necessário que o dispositivo FPGA, o *bitstream*, a *Applet* e a página HTML que instancia a *Applet* estejam localizados em uma máquina executando um programa servidor.

Um *bitstream (hard core)* é tido como uma caixa preta, cujo conteúdo não é visível nem possível de ser alterado. Porém, com o uso das classes do JBits, surge a possibilidade de reconfiguração de *hard cores*, mudando este conceito. Convencionalmente, seria necessário que se alterasse o código-fonte do *soft core* do circuito. Logo após, o mesmo precisaria ser novamente sintetizado (lógica e fisicamente). Logo, percebe-se a vantagem em termos de economia de tempo e de custos que se tem utilizando esta abordagem. A ferramenta de configuração de *bitstreams* também permite “esconder” do usuário final a arquitetura do sistema.

O processo de geração de um *hard core* reutilizável envolve duas etapas: geração do *hard core* e criação da página do reconfigurador.

Na primeira etapa, o *hard core* é gerado de forma a receber os parâmetros que caracterizam o funcionamento do circuito através de blocos de RAM internos ao FPGA (por exemplo LUTRAM).

Normalmente, a modificação destes é feita através de uma interface com microprocessador externo ou chaves. Porém, a abordagem que utiliza blocos de RAM é mais eficiente e mais flexível. Desta forma, ao gerarmos o *hard core*, todos os parâmetros deste são lidos de blocos de RAM. Estes blocos de RAM têm a posição física fixada dentro do FPGA

através de restrições impostas no momento da síntese (Figura 48). O *core* sintetizado não tem nenhum parâmetro definido até que seja feita a reconfiguração do *bitstream*, pois os blocos de RAM foram inicializados em zero.

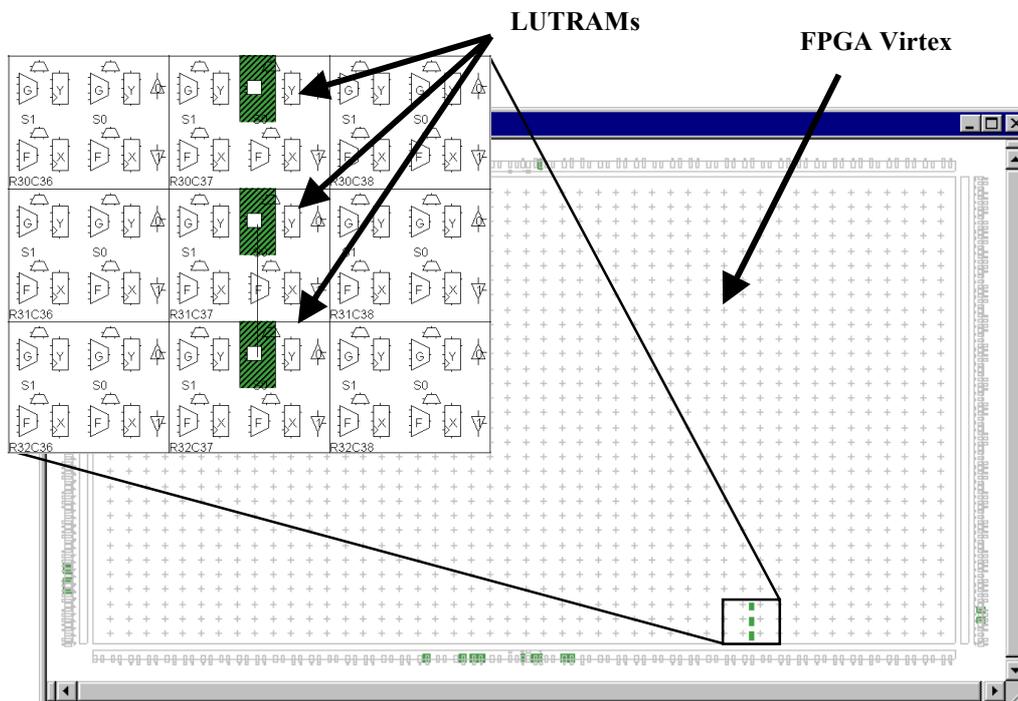


Figura 48 – Posições das LUTRAMs fixadas através do Floorplanner.

A segunda etapa do trabalho consiste em gerar a página HTML que define a interface do reconfigurador, para que o usuário tenha acesso aos parâmetros que deseja alterar.

Esta interface é criada através de parâmetros inseridos na descrição HTML da página. Os parâmetros são passados para a *Applet* no momento em que a página é carregada. Os campos de visualização e inserção de valores são instanciados através desses parâmetros. Cada campo corresponde à uma LUT (ou parte dela). Isto permite que se desenvolvam vários reconfiguradores diferentes apenas alterando tais parâmetros, sem a necessidade de ter que alterar a *Applet*.

Uma vez gerado o *bitstream* e a página do reconfigurador, procede-se à alteração do *bitstream* para configurá-lo conforme a necessidade do usuário. Após reconfigurar o *bitstream*, o usuário pode realizar o download do circuito para o dispositivo de prototipação através do mesmo ambiente.

É importante que se perceba o aspecto inovador desta abordagem. Uma vez gerado o circuito, pode-se modificá-lo sem a necessidade de re-síntese. Torna-se também extremamente simples modificar o comportamento do circuito, uma vez que toda a manipulação é feita em software. O software é baseado em um modelo cliente-servidor, executando em um *browser* qualquer. Assim pode-se remotamente reconfigurar o hardware. Por exemplo, uma dada empresa pode reconfigurar o hardware de todos os seus clientes remotamente, sem necessidade de ir até o cliente, economizando assim custos de manutenção e atualização.

4.3.1 Applet do Reconfigurador

O diagrama da Figura 49 ilustra os passos básicos envolvidos no desenvolvimento de uma aplicação que utiliza o JBits.

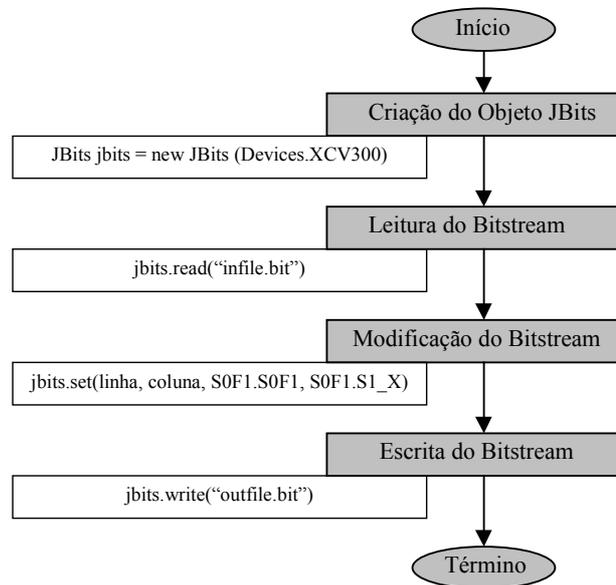


Figura 49 - Fluxo de desenvolvimento de uma aplicação JBits.

Método construtor: Um objeto JBits deve ser construído antes de qualquer procedimento. O construtor recebe o tipo de dispositivo a ser utilizado como parâmetro. A lista de dispositivos pode ser encontrada na classe *Devices*. Este método constrói um modelo para o dispositivo selecionado e executa várias inicializações. O modelo para o construtor é *JBits (int deviceType)*.

Exemplo:

```
JBits jbits = new Jbits (Devices.XCV300);
```

Leitura do *Bitstream*: Este método recebe um único parâmetro contendo o nome do arquivo do *bitstream* a ser lido. Ele carrega um *bitstream* no objeto JBits construído e mapeia os dados do *bitstream* para o modelo do dispositivo. Uma vez que o *bitstream* tenha sido carregado, os dados de configuração podem ser lidos e escritos.

Exemplo:

```
Jbits.read("infile.bit");
```

Alteração dos bits do *bitstream*: O método *set()* escreve os dados de configuração de um dado recurso no elemento configurável. O recurso é identificado por uma linha e uma coluna de CLBs. O recurso na CLB selecionada é identificado por uma constante. Estas constantes são definidas nas classes Java que contém os objetos configuráveis. Por exemplo, o ajuste da configuração do recurso **SLICE0 F1** é feito usando-se a constante **S0F1** na classe **S0F1**, ou seja **S0F1.S0F1**. Um vetor de inteiros fornecendo os bits de configuração é passado para o método *set*. Para ajustar a entrada do **S0F1.S0F1** para o valor da saída do **SLICE1 X**, por

exemplo, a constante **S0F1.S1_X** é utilizada. O modelo para o método *set* é *void JBits.set(int linha, int coluna, int[][]recurso, int[]bits);*

Exemplo:

```
JBits.set(clbLinha, clbColuna, S0F1.S0F1, S0F1.S1_X);
```

Escrita do *Bitstream*: Similar ao método de leitura do *bitstream*, o método de escrita recebe um único parâmetro, uma string contendo o nome do arquivo do *bitstream* a ser escrito. Este método escreve em um arquivo o *bitstream* do objeto JBits contruído. O modelo para o método *write* é *int JBits.write(String outFileName);*

Exemplo:

```
JBits.write("outfile.bit");
```

Lendo a configuração: O método *get()* é usado para ler a configuração de um determinado recurso de um elemento configurável. O recurso é identificado usando a mesma convenção mencionada no método *set()*. A maioria dos dados obtidos através deste método podem ser interpretados e usados por outras partes da aplicação JBits. O modelo para o método *get()* é *int[]JBits.get(int linha, int coluna, int[][]recurso);*

Exemplo:

```
int[]theValue = jbits.get(clbLinha, clbColuna, S0F1.S0F1);
```

Acessando as LUTs

As LUTs nos CLBs implementam as funções lógicas. No JBits, as LUTs das CLBs da *Virtex* são definidas na classe LUT. Esta classe define quatro arrays bidimensionais de inteiros, SLICE0_F, SLICE0_G, SLICE1_F e SLICE1_G.

Os dados de configuração podem ser lidos ou escritos em uma LUT, através dos métodos *get()* e *set()*, respectivamente. A Figura 50 mostra como os métodos *get()* e *set()* acessam as LUTs. O parâmetro **valor** é usado para especificar a lógica a ser implementada pela LUT. O **valor** é um array de inteiros (somente “0”s e “1”s) representando os 16 bits na LUT.

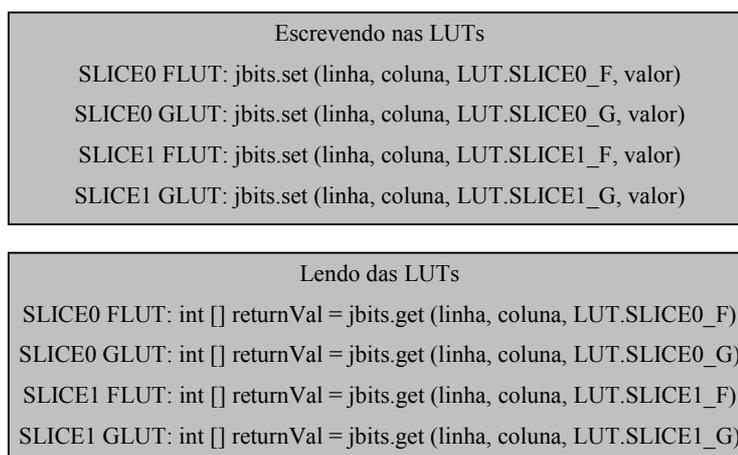


Figura 50 – Acesso às LUTs.

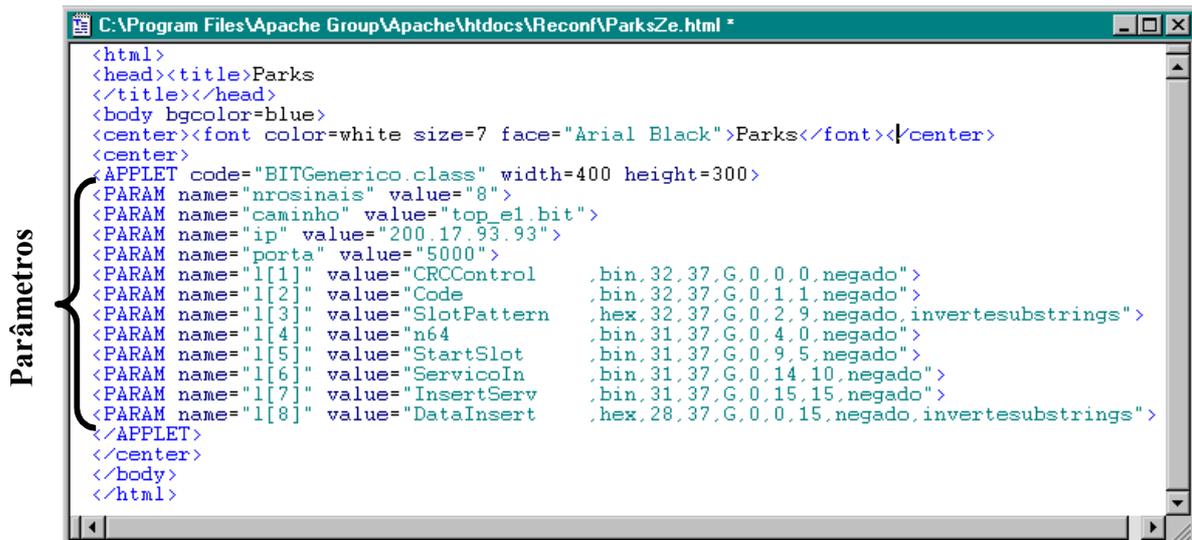
4.3.2 Exemplo de Utilização do Reconfigurador

Nesta Seção é apresentado um exemplo de página HTML que gera a interface para o reconfigurador de um determinado *bitstream*, a partir das funções de manipulação de *bitstreams*, encapsuladas na forma de uma Applet (Seção 4.3.1).

A Figura 51 mostra o código HTML que cria a página do reconfigurador. Dentro da *tag Applet* são definidos: (i) a quantidade de parâmetros a ser transmitidos ("*nrosinais* = 8"), e que correspondem à quantidade de campos de visualização e inserção de valores; (ii) o arquivo binário (*bitstream*) a ser utilizado ("*caminho* = *top_e1.bit*"); (iii) o endereço IP do servidor ("*ip* = 200.17.93.93"); (iv) a porta de comunicação ("*porta* = 5000"); e (v) um vetor ("*l[1..8]*") contendo os valores dos parâmetros do circuito a ser configurado (Ex: "*n64* (nome do sinal), *bin* (exibido como binário ou hexadecimal), *31* (linha da CLB), *37* (coluna da CLB), *G* (LUT F ou G), *o* (slice 0 ou 1), *4* (bit inicial), *o* (bit final)").

Os bits inicial e final indicam a área dentro da LUT RAM (de 16 bits) ocupada pelo sinal (ou vetor). Existem também dois parâmetros opcionais, "negado" e "invertesubstrings". O parâmetro "negado" é utilizado para que os bits sejam visualizados pelo usuário com lógica negada em relação ao que foi lido da LUTRAM. O motivo disto é por que as LUTRAMs armazenam os dados com lógica negada. Por exemplo, se o usuário gravar a palavra "1010111100001111" em uma LUTRAM, na realidade, o valor armazenado será "0101000011110000".

O parâmetro "invertesubstrings" é utilizado quando deseja-se visualizar um vetor no sentido contrário ao qual foi declarado no projeto. Por exemplo, se um sinal foi declarado em VHDL como *std_logic_vector(0 TO 15)* e recebeu o valor "1010111100001111", ele pode ser visualizado como sendo (*15 DOWNTO 0*), apresentando o valor "1111000011110101".



```
<html>
<head><title>Parks
</title></head>
<body bgcolor=blue>
<center><font color=white size=7 face="Arial Black">Parks</font></center>
<center>
<APPLET code="BITGenerico.class" width=400 height=300>
<PARAM name="nrosinais" value="8">
<PARAM name="caminho" value="top_e1.bit">
<PARAM name="ip" value="200.17.93.93">
<PARAM name="porta" value="5000">
<PARAM name="l[1]" value="CRCControl .bin,32,37,G,0,0,0,negado">
<PARAM name="l[2]" value="Code .bin,32,37,G,0,1,1,negado">
<PARAM name="l[3]" value="SlotPattern .hex,32,37,G,0,2,9,negado,invertesubstrings">
<PARAM name="l[4]" value="n64 .bin,31,37,G,0,4,0,negado">
<PARAM name="l[5]" value="StartSlot .bin,31,37,G,0,9,5,negado">
<PARAM name="l[6]" value="ServicoIn .bin,31,37,G,0,14,10,negado">
<PARAM name="l[7]" value="InsertServ .bin,31,37,G,0,15,15,negado">
<PARAM name="l[8]" value="DataInsert .hex,28,37,G,0,0,15,negado,invertesubstrings">
</APPLET>
</center>
</body>
</html>
```

Figura 51– Código HTML para geração da interface.

A Figura 52 mostra a interface do reconfigurador, criada através da página HTML. Nela, pode-se visualizar: (i) os campos contendo os valores armazenados nas LUTRAMs, e

que podem ser alterados, (ii) o botão “Salvar”, para gravar os valores nas LUTRAMs, e (iii) o botão “Download”, que carrega o arquivo de configuração na placa de prototipação.

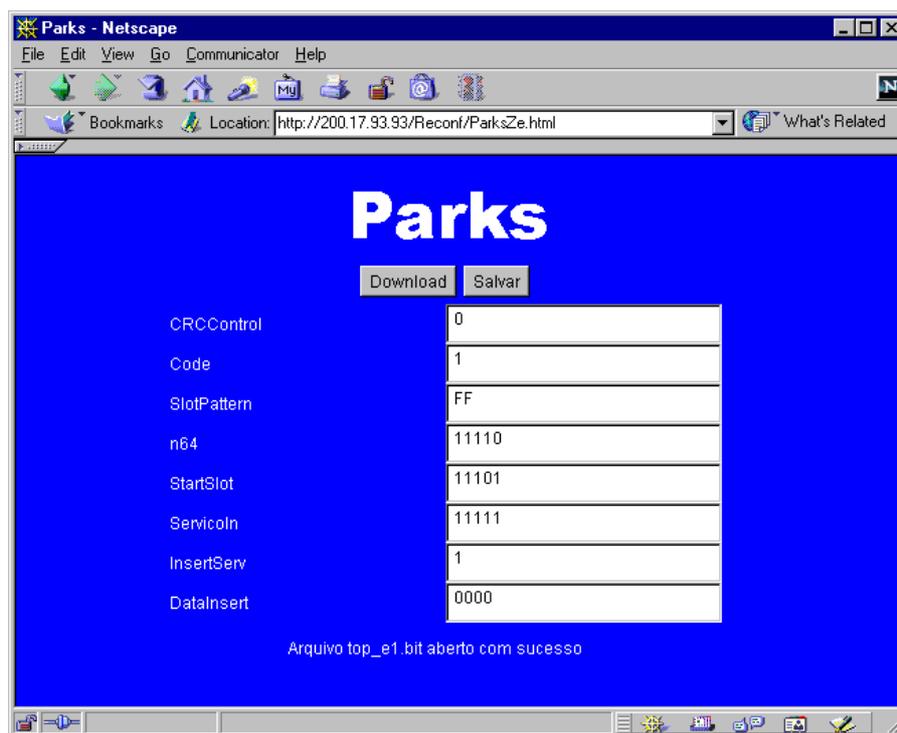


Figura 52 – Exemplo de reconfigurador de *bitstream*.

A aplicação para reconfiguração total e remota de *cores* é inserida também no ambiente de distribuição apresentado no Capítulo 3. É importante ressaltar que um *bitstream* completo só é utilizável no ambiente (placa) para o qual ele foi sintetizado, devido às restrições de pinos de entrada e saída. Para que um *hard core* seja distribuído e utilizado em diferentes ambientes, é necessária a reconfiguração parcial e a “virtualização” dos pinos de entrada e saída (conexão a um barramento padrão interno e não mais aos pinos de entrada e saída do FPGA). Esta “virtualização” é equivalente à inserção de um barramento interno, como o *WISHBONE*, dentro do FPGA.

4.4 Reconfiguração Parcial

Encontra-se em fase de desenvolvimento uma ferramenta para geração de *bitstreams* parciais para FPGAs da família *Virtex*. Esta aplicação está escrita em Java, sem a utilização das classes JBits.

Baseada em um estudo minucioso da arquitetura interna de FPGAs da família *Virtex* e de seu arquivo de configuração (*bitstream*), em [MES01], essa ferramenta realiza as mesmas operações do reconfigurador apresentado na Seção anterior, além de gerar um *bitstream* parcial.

A Figura 53 mostra a interface dessa ferramenta. Ela possui como entradas um arquivo de configuração completo e um arquivo-texto contendo os parâmetros do protocolo para geração de um *bitstream* parcial. Na janela maior, o usuário visualiza o conteúdo de um

arquivo de configuração. Nas janelas menores (parte inferior direita da Figura) encontram-se as opções para localização de elementos dentro do *bitstream*. É possível realizar a busca por um bit específico informando-se a linha, coluna, *Slice* e LUT desejados. O bit correspondente é, então, destacada na janela maior (à esquerda), onde pode ser alterado. Depois da alteração, no momento de “salvar” o arquivo, o CRC é recalculado e gravado no registrador apropriado. Caso a opção “Salvar Parcial” seja selecionada, um *bitstream* parcial é gerado, conforme os parâmetros informados no início do processo.

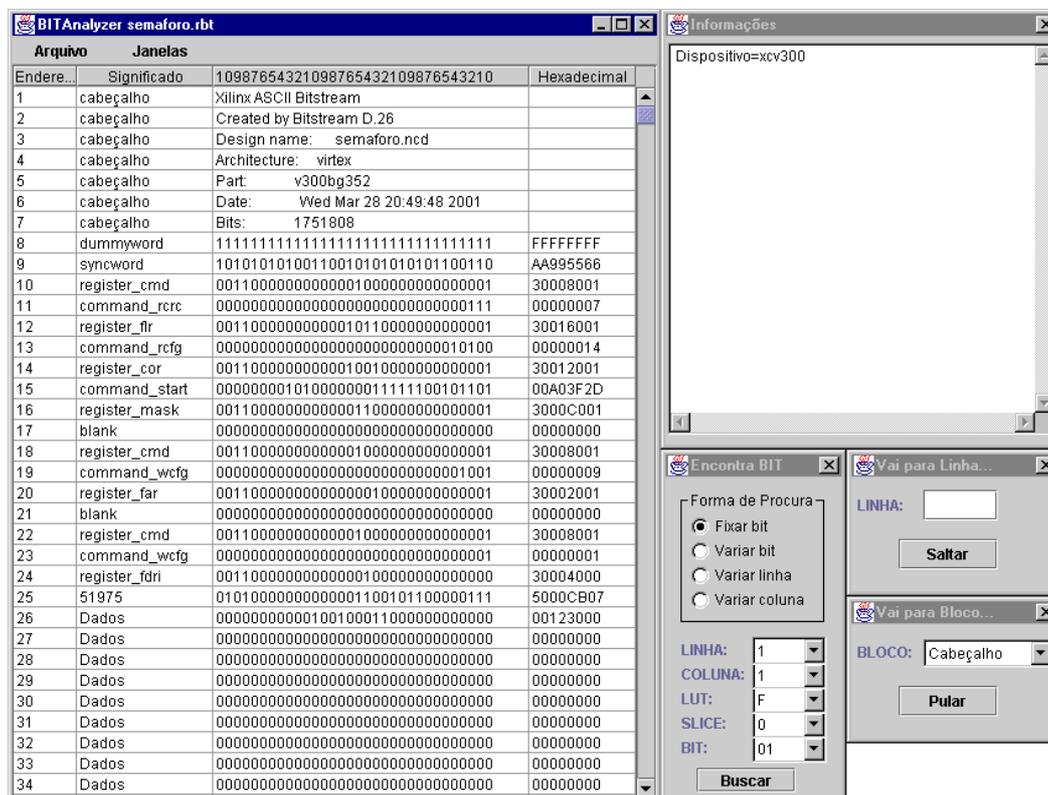


Figura 53 – Ferramenta para geração de *bitstreams* parciais.

Devido ao fato de a ferramenta de download disponível não permitir o download de arquivos de configuração parciais, o *bitstream* parcial gerado foi pré-validado através do download no FPGA sem erro de inicialização. Apesar do download correto, a ferramenta de download apaga todos dados de configuração do dispositivo após o download do *bitstream* parcial, impedindo que a reconfiguração parcial seja de fato realizada. Encontra-se em desenvolvimento uma solução para este problema.

5 Proposta de Interconexão entre Cores em Dispositivos Virtex

Como dito anteriormente, é possível alterar parcialmente a funcionalidade do hardware sem interromper o funcionamento do sistema. Para que este tipo de aplicação seja uma realidade em projetos de hardware é necessário definir uma estratégia de reconfiguração parcial para os dispositivos FPGA. A idéia é que haja no FPGA uma estrutura análoga a uma interface PCI em um PC, onde se conectam dispositivos sem a modificação do sistema. A estrutura também deve ser responsável pela “virtualização” dos pinos de entrada/saída. Esta “virtualização” é necessária para que o *core* seja portátil, pois os pinos de entrada/saída diferem de um dispositivo para outro. Esta estrutura está relacionada aos itens (v) e (vi) dos requisitos citados no início do Capítulo 4.

Até o presente momento não foi encontrada na literatura referência sobre uma proposta de interface intra-FPGA para que dispositivos reconfiguráveis possam receber *cores* de forma modular. Essa modularidade consiste em conectar e remover *cores* sem que haja necessidade de maiores alterações na lógica pré-existente no FPGA. Há técnicas semelhantes, voltadas para o projeto de *cores* em ASICs. Como exemplo podem ser citados os padrões de barramento *Amba*, *CoreConnect* e *WISHBONE*, já mencionados anteriormente.

A Seção 5.1 apresenta a proposta do barramento de comunicação entre *cores*, explicando a idéia e a forma de conexão entre os *cores*. Na Seção 5.2 são definidas as estruturas internas do controlador do barramento e dos módulos de envio e recepção de dados. A Seção 5.3 e 5.4 apresentam, respectivamente, a aplicação utilizada para validar o barramento de comunicação e os resultados da simulação desta aplicação. A Seção 5.5 apresenta o início da prototipação da interface de comunicação, visando a reconfiguração dinâmica dos *cores*.

5.1 Proposta de Interconexão

O ponto de partida para a definição da interface de comunicação foi estabelecer uma porção de hardware fixo (estático) no FPGA, chamado de controlador. O controlador é responsável pela comunicação com o mundo externo (pinos de entrada/saída do dispositivo) e pela comunicação com os *cores*. Isto significa que os módulos de hardware a serem conectados, os *cores*, somente comunicar-se-ão com o mundo externo através desta interface. Estamos, desta forma, “virtualizando” os pinos de entrada e saída.

O FPGA deve ser inicialmente carregado apenas com o controlador. Os *cores* são carregados em tempo de execução, quando necessários. Desta forma ter-se-á um conceito de hardware semelhante ao conceito de memória virtual, ou seja, em nosso hardware ter-se-á apenas os módulos que precisam ser executados naquele momento. Os demais estão armazenados externamente, para uso posterior.

Em [VIL97] é apresentado um sistema de reconfiguração utilizando como estudo de

caso um sistema para processamento de imagens. Este sistema é reconfigurado quatro vezes para cada conjunto de dados (imagem) a ser processado. O FPGA inicialmente armazena o sinal de vídeo em uma memória, aplicando em seguida duas transformações diferentes sobre as imagens, e finalmente torna-se um modem e transmite o resultado do processamento. Este exemplo pode ser implementado utilizando a proposta de interface de conexão de *cores*, sendo cada uma das quatro configurações um *core* comunicando-se com a *interface*.

A comunicação entre o controlador e os demais *cores* é feita através de pinos virtuais. Estes pinos virtuais são implementados através de *buffers tristate*, presentes na arquitetura de FPGAs da família *Virtex*, como mencionado na Seção 4.1.

A idéia inicial era utilizar *buffers* na fronteira entre o controlador e o *core*, servindo de conector entre os mesmos. Estes *buffers* fariam parte tanto da descrição do controlador quanto da descrição do *core*. Haveria, assim, uma sobreposição destes *buffers* no momento da reconfiguração parcial do FPGA, como motra a Figura 54.

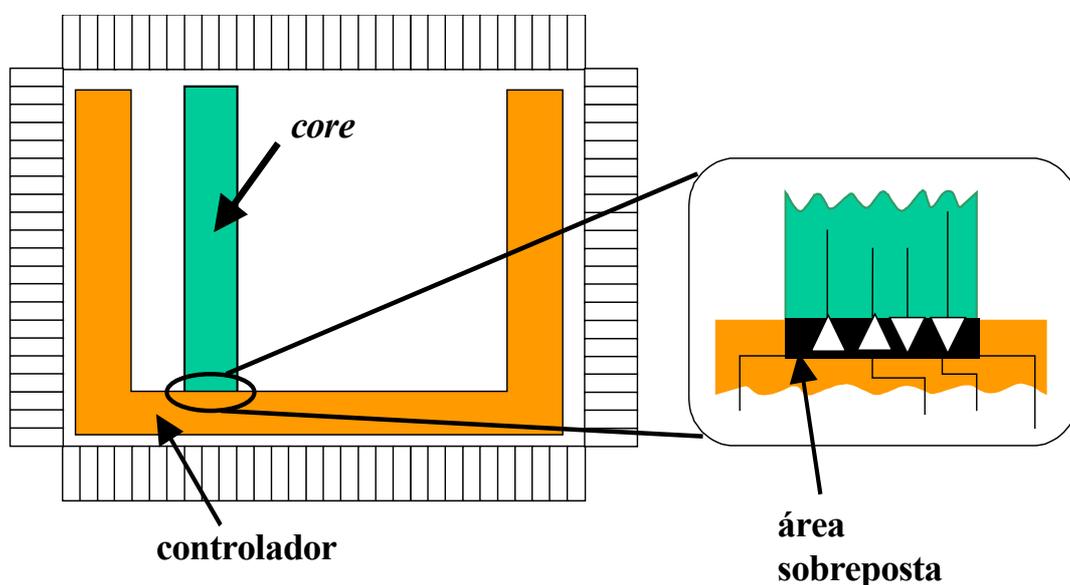


Figura 54 – Barramento de comunicação de *cores*.

Esta proposta não é factível. Isto deve-se a fato da CLB compartilhar LUTs, *Flip-flops* e roteamento. Logo, uma dada CLB do controlador, que faria a interface com o *core*, conteria também recursos adicionais de lógica, os quais seriam destruídos no momento da inserção do *core*. As ferramentas de síntese não possuem recursos para proibir o uso de roteamento em uma área.

Esta abordagem, portanto, não resolveria o problema. A solução encontrada foi utilizar duas camadas de *buffers*, uma camada pertencendo ao controlador e outra pertencendo ao *core*, como mostra a Figura 55. A conexão dos *cores* ao controlador é feita por uma linha comum de roteamento.

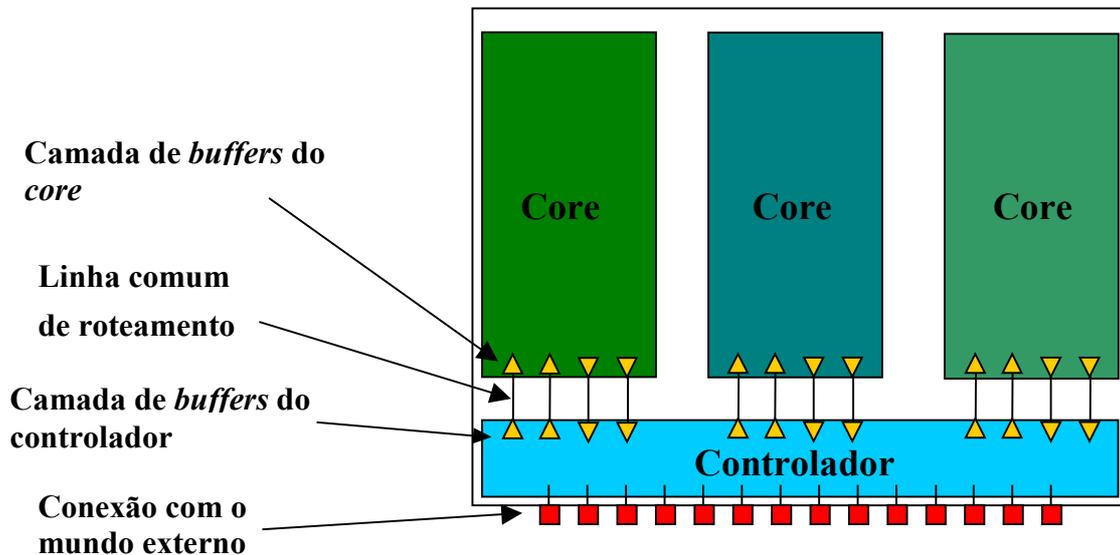


Figura 55 – Duas camadas de buffers.

5.2 Definição do Controlador e Interfaces dos Cores

A Figura 56 mostra os recursos de roteamento nas saídas dos *buffers tristate*. Existem apenas quatro fios paralelos para cada quatro colunas de CLBs (um conjunto de oito *buffers*). Cada fio está ligado à saída de um dos *buffers* de cada CLB, identificados por círculos na Figura. Quando um destes *buffers* é utilizado, os outros, ligados ao mesmo fio, ficam sem recursos disponíveis para sua saída.

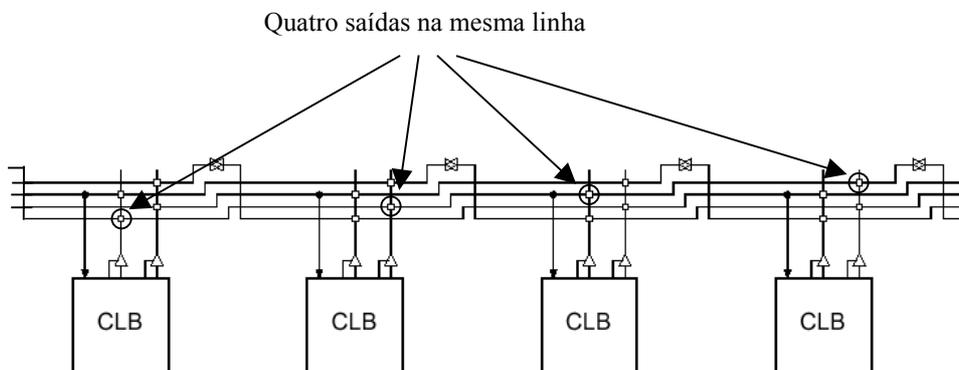


Figura 56 – Conexões entre buffers e linhas horizontais dedicadas.

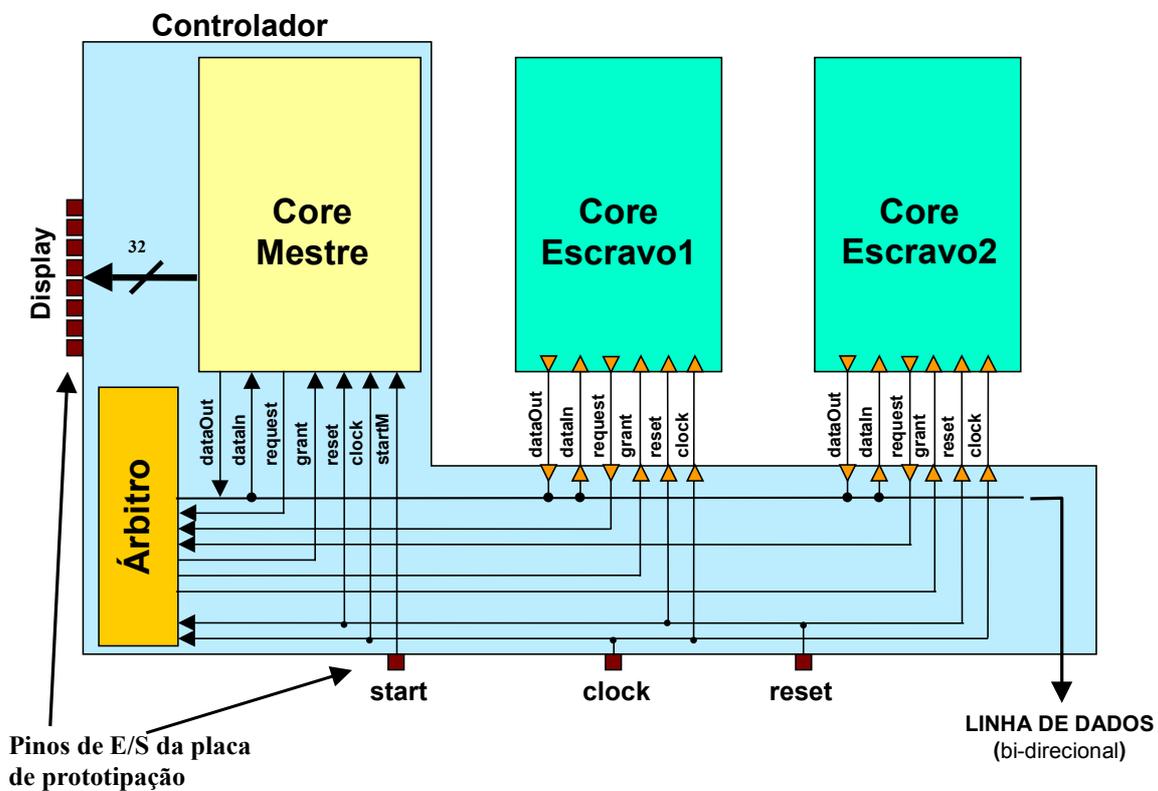
Estas limitações na arquitetura de roteamento dos *buffers tristate* impossibilitam o uso de barramentos com grande número de bits (largura da palavra com 16 bits, por exemplo). Experimentos realizados com barramentos de largura de 8 bits não tiveram sucesso na etapa de roteamento. Decidiu-se, então, utilizar um fio apenas para a transmissão de dados entre os *cores*.

O controlador da interface deve ser o árbitro deste barramento, gerenciando o aceite ou a rejeição de conexões, bem como o controle de conflitos relativos ao acesso aos pinos de entrada/saída do FPGA.

Ainda no controlador é definido um *core* mestre, também estático, que controla a interface com o mundo externo. O *core* mestre pode ser modificado, de acordo com a aplicação utilizada.

A Figura 57 ilustra a utilização da interface de comunicação, visando sua validação. Esta aplicação é composta por 3 módulos: (1) controlador, composto pelo mestre, árbitro e *buffers* de comunicação; (2) *core* escravo número 1; (3) *core* escravo número 2. A interface com o mundo externo é feita na parte fixa do FPGA, ou seja, pelo controlador. Esta interface é composta por um barramento de 32 bits (*display*), e os sinais *clock*, *start*, *reset*.

A comunicação entre o controlador e cada *core* é feita por 6 sinais: (1-2) *data in* e *data out*, responsáveis pelo tráfego de dados; (3-4) *request* e *grant*, responsáveis pela solicitação de envio de dados e permissão para enviá-los; (6-7) *clock* e *reset*.



OBSERVAÇÃO: Por clareza do diagrama não são representados os comandos de abertura dos *buffers tri-state*. Excetuando-se o *buffer* responsável por escrever na linha de dados, sinal *dataOut*, o qual é controlado pelo árbitro, todos os demais estão sempre conduzindo. Esta simplificação é adotada em todos os diagramas deste Capítulo.

Figura 57 – Integração de cores utilizando a proposta de barramento de comunicação.

A máquina de estados do árbitro é mostrada na Figura 58. Este tipo de controle permite que haja *cores* com prioridade mais alta que outros e, ao mesmo tempo, garante que nenhum *core* com alta prioridade seja atendido mais de uma vez, enquanto um de baixa prioridade fica em estado de espera.

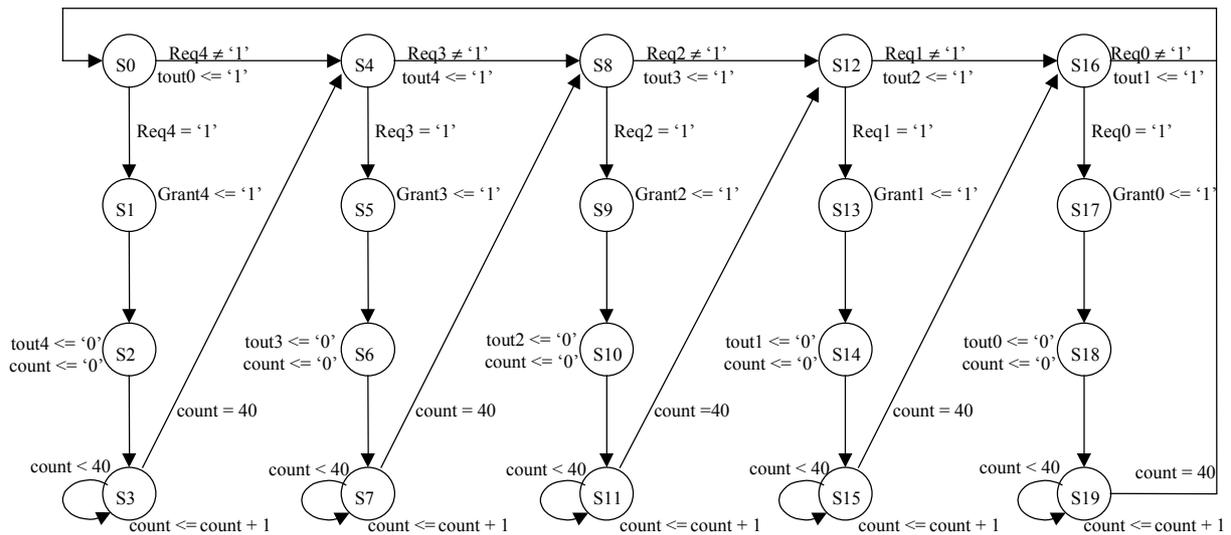


Figura 58 – Máquina-de-estados do árbitro do barramento.

Quando um *core* deseja enviar dados para outro módulo do sistema, o sinal “req” (request) é enviado pelo *core*, e este fica aguardando o sinal “grant”. O árbitro recebe os sinais de request dos *cores* e atende aos pedidos de acesso ao barramento (um de cada vez) enviando o sinal “grant” para o *core* que tem permissão de acesso naquele momento. Após o envio do sinal “grant” a um determinado *core*, o sinal que habilita o *buffer tristate* de escrita na linha de dados (“tout”) é ativado (em zero) apenas para aquele *core*. Cada transmissão dura 40 ciclos de clock, pois a unidade mínima de transmissão é um pacote de 40 bits (8 para endereço de destino e 32 para dados). Após cada transmissão, o árbitro volta a verificar a existência de outro sinal “req” ativo.

Para que os *cores* sejam compatíveis com a interface de comunicação mostrada na Figura 57, foram criados os módulos “Send” e “Receive” (Figura 59), utilizados para a transmissão e para a recepção de dados em cada *core*.

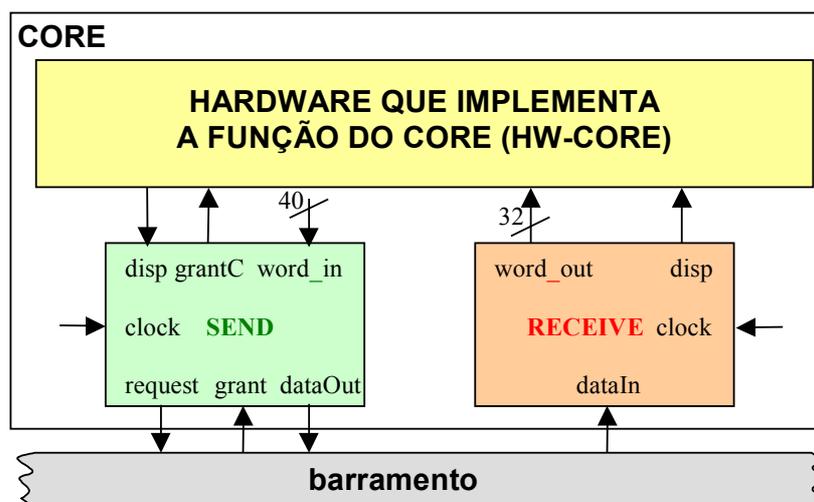


Figura 59 - Módulos Send e Receive.

A máquina de estados do módulo “Send” é mostrada na Figura 60.

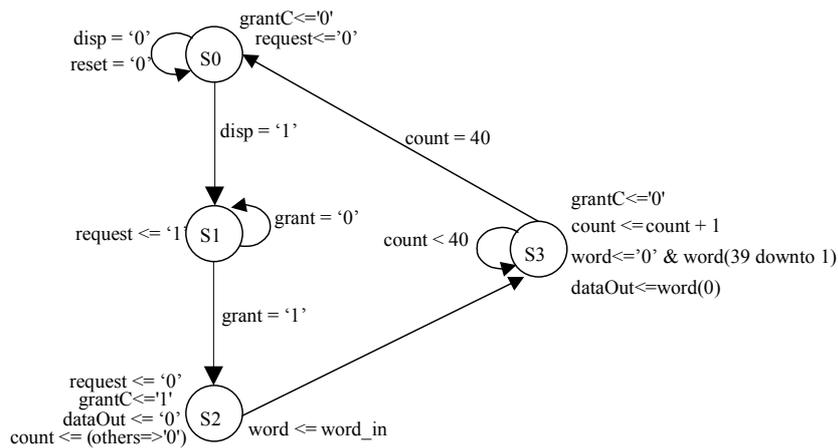


Figura 60 – Máquina de estados do módulo Send.

A máquina de estados do módulo “Send” fica aguardando a solicitação de envio de dados, no estado “S0”. Quando um dado está disponível para ser enviado ao barramento, o módulo “Send” recebe o sinal “disp” do *hw-core*. Isto faz com que este módulo envie um sinal “request” para o árbitro e fique esperando pelo sinal “grant” (em “S1”). Ao receber o sinal “grant”, o módulo recebe o pacote de 40 bits do *hw-core*, através do sinal “word_in” (em “S2”). Neste momento, o sinal “grantC” é enviado ao *hw-core* para avisar que o dado já está no módulo “Send”, e inicia-se a transferência. Primeiro, é enviado (através do sinal “dataOut”) o “start bit” (bit ‘0’) (ainda em “S2”), seguido da transferência serial do pacote para o barramento de comunicação, durante os 40 ciclos de clock (em “S3”).

A máquina de estados do módulo “receive” é mostrada na Figura 61.

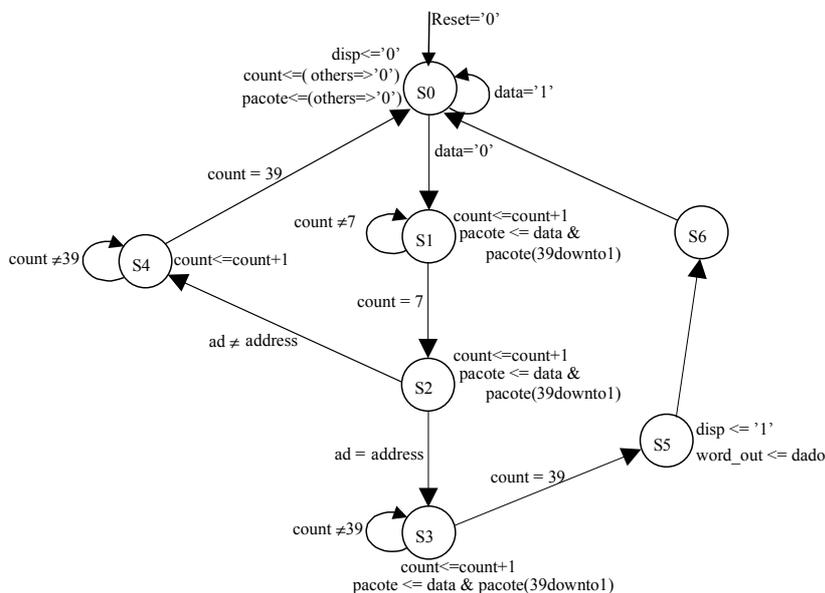


Figura 61 – Máquina de estados do módulo Receive.

O módulo “Receive” monitora constantemente a linha de dados (“S0”). O estado normal da linha é em ‘1’ lógico. Quando recebe um bit ‘0’, o módulo “Receive” percebe que uma transmissão está iniciando, e passa a armazenar os dados da linha, através do sinal dataIn

("S1"). Após armazenar os 8 primeiros bits, é feita a verificação do endereço de destino ("S2"). Se o endereço de destino corresponde ao endereço do *core* que está recebendo o dado, a recepção dos 32 bits restantes continua ("S3"). Do contrário, o restante do pacote é ignorado ("S4"). Mesmo ignorando o pacote, a máquina volta a monitorar a linha somente após os 40 ciclos da transmissão em andamento. Após receber todo o pacote, o módulo avisa ao *hw-core*, através do sinal "disp" ("S5"), e disponibiliza este pacote através do sinal "word_out". O estado "S6" é utilizado para que o sinal "disp" fique ativo durante 2 ciclos de clock.

Para ser utilizado junto ao barramento proposto, o *core* deve ser inserido em um invólucro (*wrapper*), contendo os módulos "Send" e "Receive", além dos *buffers* de interface, como mostra a Figura 62.

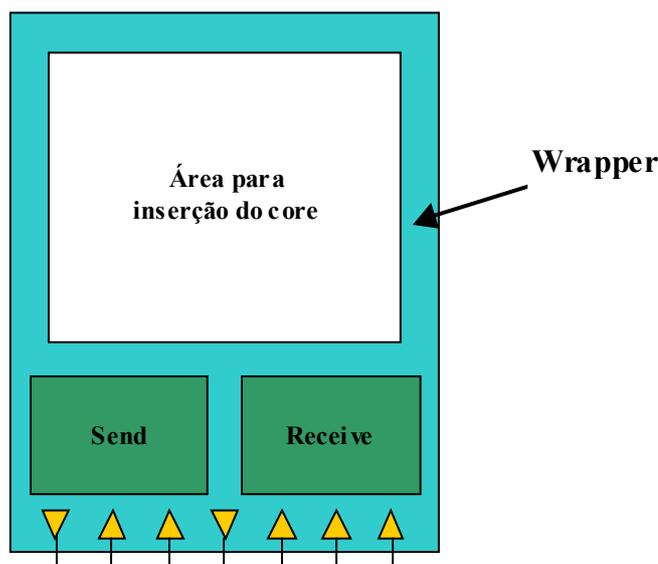


Figura 62 – Invólucro para inserção de *cores*.

5.3 Validação da Proposta de Interface

Para que se pudesse por em funcionamento o barramento de interconexão, foi desenvolvida uma aplicação onde podem ser utilizados até três *cores* (escravos), cada um realizando uma operação aritmética diferente com dois operandos.

O *core* mestre (integrado ao controlador), além de determinar quando e qual *core* deve efetuar a operação, também fornece os operandos. Nesta aplicação foram utilizados três operandos ("A", "B" e "C") de 32 bits e duas operações (adição e subtração). Tanto os operandos, quanto a seqüência de instruções são armazenadas em LUTRAMs, dentro do *core* mestre. Cada operando utiliza 2 LUTRAMs, enquanto que a seqüência de instruções utiliza 4 LUTRAMs, podendo armazenar até 8 instruções de 8 bits.

Da mesma maneira que no exemplo da Seção 4.3, estas LUTRAMs foram fixadas em locais pré-determinados, através da ferramenta *Floorplanner*. Após, os parâmetros contidos nas LUTRAMs foram inicializados através da ferramenta apresentada também na Seção 4.3.

Ao receber o sinal "StartM", proveniente do mundo externo, o *core* mestre busca as

instruções nas LUTRAMs e as carrega ordenadamente em um vetor de 64 bits (oito instruções). Simultaneamente, acontece a busca dos operandos, que são carregados em três registradores de 32 bits.

Cada instrução para esta aplicação é formada como mostra a Figura 63:

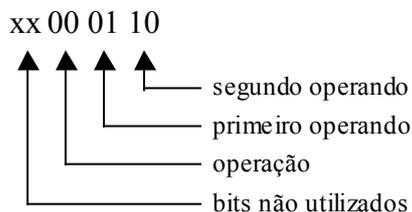


Figura 63 – Formato da instrução.

O *core* mestre possui ainda um registrador “R” que tem por função armazenar os resultados parciais de cada operação. Este registrador também pode ser utilizado como um operando, como mostra a Tabela 3. Por exemplo, se tivermos como instrução a palavra “00010011”, é realizada a operação A – R.

Tabela 3 – Codificação para operação e operandos.

Cod.	Operação	Operando
00	Adição	A
01	Subtração	B
10	-	C
11	Halt	R

Após a busca de operandos e instruções nas LUTRAMs, o mestre começa as etapas de decodificação e execução de cada instrução. Para cada instrução, o mestre envia dois pacotes para o mesmo destino, cada um contendo um dos dois operandos.

Primeiramente, é analisada a operação a ser executada. Isto determina o destino dos dois operandos, ou seja, trata-se do endereço do *core* que deve efetuar a operação. Neste momento são definidos os 8 primeiros bits do primeiro pacote (contendo este endereço). Nesta aplicação, foram utilizados apenas 2 bits para codificar a operação, limitando o número de *cores* em três (além da operação “Halt”). Posteriormente é buscado o primeiro operando, cujo conteúdo é inserido nos 32 bits restantes do pacote e, logo em seguida, é feito o mesmo procedimento com o segundo operando.

O mestre fica, então, em estado de espera, até que o *core* retorne o resultado. O *core* executa a operação para a qual foi projetado e retorna um pacote endereçado ao mestre, contendo o resultado da operação. O mestre armazena o resultado no registrador “R” e só então busca a próxima instrução. Os resultados parciais são mostrados no *display* da placa de

prototipação. Após executar as oito instruções, ou ao encontrar a instrução “Halt”, o sistema termina a execução, mostrando o resultado final, e fica esperando novamente pelo sinal “StartM”.

5.4 Resultados da Simulação

Antes de ser prototipado em hardware, o sistema foi simulado utilizando o ambiente Active-HDL, da Aldec [ALD01b]. A Figura 64 ilustra um trecho da simulação.

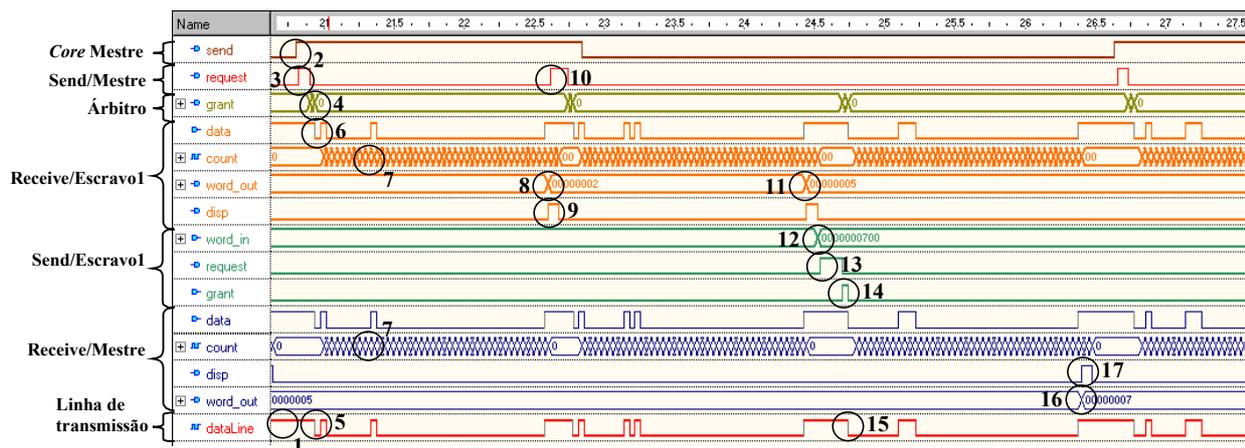


Figura 64 - Simulação da interface de comunicação.

1. Inicialmente, a linha de transmissão está recebendo '1' lógico (estado *default*).
2. O core mestre envia o sinal "send" ao módulo "Send", avisando que deseja transmitir um pacote.
3. O módulo "Send", então, envia o sinal "request" ao árbitro, e fica esperando pelo sinal "grant".
4. O sinal "grant" é enviado ao módulo "Send" do mestre.
5. A transmissão inicia com o "start bit".
6. O módulo "Receive" do core escravo1 detecta o "start bit" e passa a armazenar os dados. Neste momento, todos os módulos "Receive" de todos os cores estão recebendo o pacote.
7. Após receber os oito primeiros bits (endereço do destino), estes módulos verificam se este endereço corresponde ao seu. O que corresponder continua a recepção. Os demais esperam 32 ciclos (o restante do pacote) para começar a monitorar a linha novamente.
8. Após 40 ciclos de clock, o pacote contendo o primeiro operando foi recebido pelo seu destino, o módulo "Receive" do core escravo1.
9. O dado é disponibilizado ao core.

10. Terminada a primeira transmissão, o core mestre permanece com o sinal "send" ativo, pois deseja transmitir o segundo pacote, com um segundo operando. O módulo "Send" do mestre envia novamente o sinal "request" e o processo se repete.
11. O segundo operando é recebido pelo módulo "Receive" do core escravo.
12. O core escravo executa a operação para a qual foi desenvolvido (neste exemplo, soma) e disponibiliza o resultado ao seu módulo "Send".
13. Este módulo requisita a linha de transmissão.
14. O sinal "grant" é enviado pelo árbitro, para que se inicie a transmissão.
15. O módulo "Receive" do core mestre detecta um "start bit" na linha de transmissão e começa a receber o pacote, contendo o resultado da operação.
16. O pacote é recebido e o dado fica disponível ao core mestre.
17. O core mestre recebe o aviso do módulo "Receive".

A aplicação apresentada na Seção 5.3 foi prototipada na placa VW300 (*Virtual Workbench*) [VCC01]. A forma de depuração utilizada na placa de prototipação foi exibir os valores dos operandos e dos resultados em um display alfa-numérico.

Nas primeiras tentativas de prototipação, surgiram problemas que impediram o funcionamento correto do barramento. Um dos problemas aconteceu devido à falta de inicialização de algumas máquinas de estado. Uma segunda causa de erro encontrada foi ocasionada pela existência de sinais flutuantes, que afetaram o funcionamento do árbitro. A causa da existência destes sinais foi o fato de que o grupo de *buffers tri-state* de conexão entre o controlador e o terceiro *core* não era utilizado. A solução foi criar uma lógica extra, apenas para que estes *buffers* recebessem '0' lógico, evitando assim sinais flutuantes. Um terceiro fato importante, diz respeito à linha de dados. Quando nenhuma transmissão está acontecendo, ela deve ficar recebendo '1' lógico. Isto evita que a linha fique em alta impedância, ocasionando novamente sinais flutuantes.

Após a resolução destes problemas, o funcionamento no FPGA foi correto, o que demonstrou que a estrutura de barramento com *buffers tri-state* pode ser utilizada em dispositivos *Virtex*.

5.5 Reconfiguração dos Cores Utilizando a Interface de Comunicação

Uma vez a interface de comunicação validada por simulação e prototipação, procede-se à geração dos módulos em separado.

Para o exemplo da Figura 57 são gerados 3 *bitstreams*: um para o controlador e um para cada *core* escravo. Para a validação foram criados apenas 2 *bitstreams*, um para o controlador e outro para um dos *cores*. A posição física dos elementos de cada *bitstream* é definida pela ferramenta de *floorplaning* (planta-baixa).

Primeiramente, foi gerado um *bitstream* completo contendo o controlador, com seus *buffers tristate* de conexão ligados à outra camada de *buffers*, externa ao controlador, correspondentes aos *buffers* do *core* a ser inserido. Esta outra camada será sobrescrita pela área do segundo *bitstream* a ser inserida, no momento da reconfiguração parcial.

Os *buffers* desta camada não podem ficar desconectados, ou seja, não ter nenhuma lógica associada a ele no lado da conexão com o *core*. Isto por que a ferramenta de síntese exclui todos os *buffers* desconectados, pois estes são tidos como desnecessários. Foi então utilizada uma lógica extra (representada por uma porta lógica *and*), que serviu para garantir a permanência dos *buffers* no momento da síntese. Esta lógica não precisa ser removida, pois é sobrescrita no momento da reconfiguração. A Figura 65 ilustra o *bitstream* gerado para o controlador, com a segunda camada de *buffers* e com a lógica extra. Como nunca pode ocorrer uma situação em que todos os sinais (“clock”, “grant”, “dataIn” e “reset”) de entrada recebam ‘1’ lógico simultaneamente, os *buffers* de saída sempre recebem ‘0’, não interferindo com o funcionamento do árbitro.

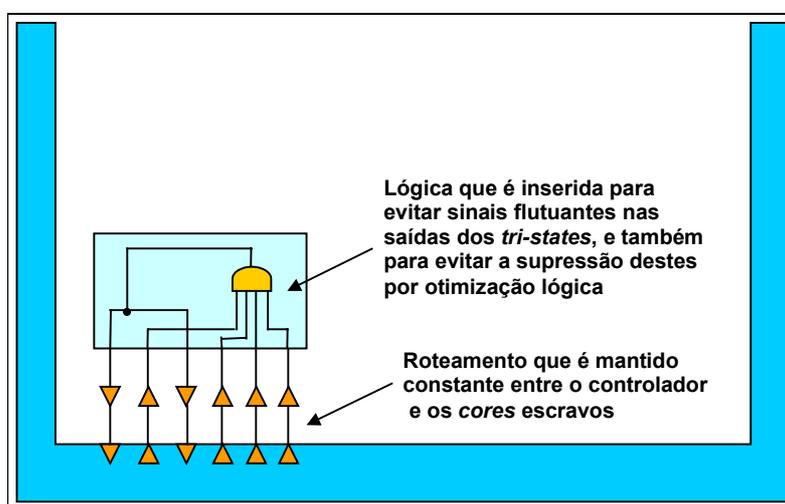


Figura 65 –*Bitstream* do controlador, com lógica extra.

Posteriormente, foi gerado o segundo *bitstream*, contendo um *core* e as duas camadas de *buffers* de comunicação, uma interna ao *core*, e outra correspondendo aos *buffers* do controlador. Da mesma forma que para o controlador, os *buffers* externos ao *core* também devem ser utilizados, para que não sejam suprimidos no momento da síntese. Além disso, a ferramenta de síntese não aceita como entrada um projeto que não contenha pinos de saída. A forma de resolver estes dois problemas foi ligar os *buffers* externos ao *core* (correspondentes aos do controlador) a pinos quaisquer de entrada/saída do FPGA.

Algumas das maiores dificuldade encontradas na etapa de síntese do controlador e do *core* separadamente, foram:

- (i) Fixar os elementos de comunicação – *buffers tri-state* – em locais que permitissem a reconfiguração parcial do *bitstream* sem afetar o resto do sistema, e que os locais fossem os mesmos para o controlador e para o *core*. A ferramenta de *floorplanning* muitas vezes impediu a fixação em determinadas áreas, devido a limitações do dispositivo.

- (ii) Garantir que o roteamento da lógica do controlador não utilizasse as CLBs a serem ocupadas pelo *core*, e vice-versa. A ferramenta de *floorplanning* permite restringir apenas a lógica do sistema, não o roteamento. Para isto foi preciso adaptar a disposição da lógica, forçando a ferramenta de roteamento a escolher um caminho que não interferisse na área a ser ocupada pelo outro *core*.
- (iii) Garantir que as linhas de roteamento entre as camadas de *buffers* fossem exatamente iguais para ambos os *bitstreams*, pois estes fios serão sobrepostos parcialmente no momento da inserção do *core*.
- (iv) Evitar a supressão dos *buffers* ligados a pinos de saída do FPGA. Quando encontra um *buffer* deste tipo, a ferramenta o exclui, e passa a utilizar o *buffer* do pino de saída. Para contornar este problema, é necessário restringir os pinos de saída, no momento da síntese lógica, como apenas “output pin”, pois o estado *default* é “tri-state pin”.
- (v) Evitar que o *core* utilizasse o *clock* global do FPGA. A linha de clock global passa verticalmente pelo centro do FPGA, ramificando-se por todas as linhas de CLBs. Isto faz com que o roteamento do *core* utilize áreas que serão ocupadas, posteriormente, por outros *cores* e até mesmo pelo controlador. Para que o clock global não seja utilizado, também é necessário que se faça restrições no momento da síntese lógica do projeto, indicando “don’t use” para o parâmetro clock global.

Estas dificuldades mostram o quão difícil é implementar a reconfiguração de *hard cores* em FPGAs comerciais e a carência de ferramentas para isto. É importante mencionar que a maior dificuldade foi garantir o mesmo roteamento entre as camadas de *buffers*. Foi necessário manualmente remover o roteamento do *core* e do controlador, refazer manualmente o roteamento das linhas de conexão entre as camadas de *buffers*, e depois utilizar o roteamento automático para rotear o restante do circuito para cada um dos dois *bitstreams*.

A Figura 66 mostra os 6 fios de roteamento entre as camadas de *buffers* e o limite entre o controlador e do *core*.



Figura 66 - Roteamento entre os buffers.

Após diversas iterações manuais obteve-se dois *bitstreams* com o mesmo roteamento entre as camadas de *buffers*. A etapa seguinte é a união destes *bitstreams*.

Utilizando uma ferramenta desenvolvida localmente ao grupo de pesquisa, estes *bitstreams* são agrupados em um único arquivo de configuração. Esta ferramenta permite a geração de um *bitstream* completo, contendo todo o circuito. Neste caso, a reconfiguração não é parcial, mas sim total, ou seja, um novo arquivo de configuração completo será carregado no FPGA.

A Figura 67 ilustra, na parte superior, os *bitstreams* gerados separadamente e, na parte inferior o *bitstream* final, contendo a lógica do *core* inserida no primeiro *bitstream*. A área pontilhada no segundo *bitstream* indica a área exata a ser inserida no *bitstream* do controlador.

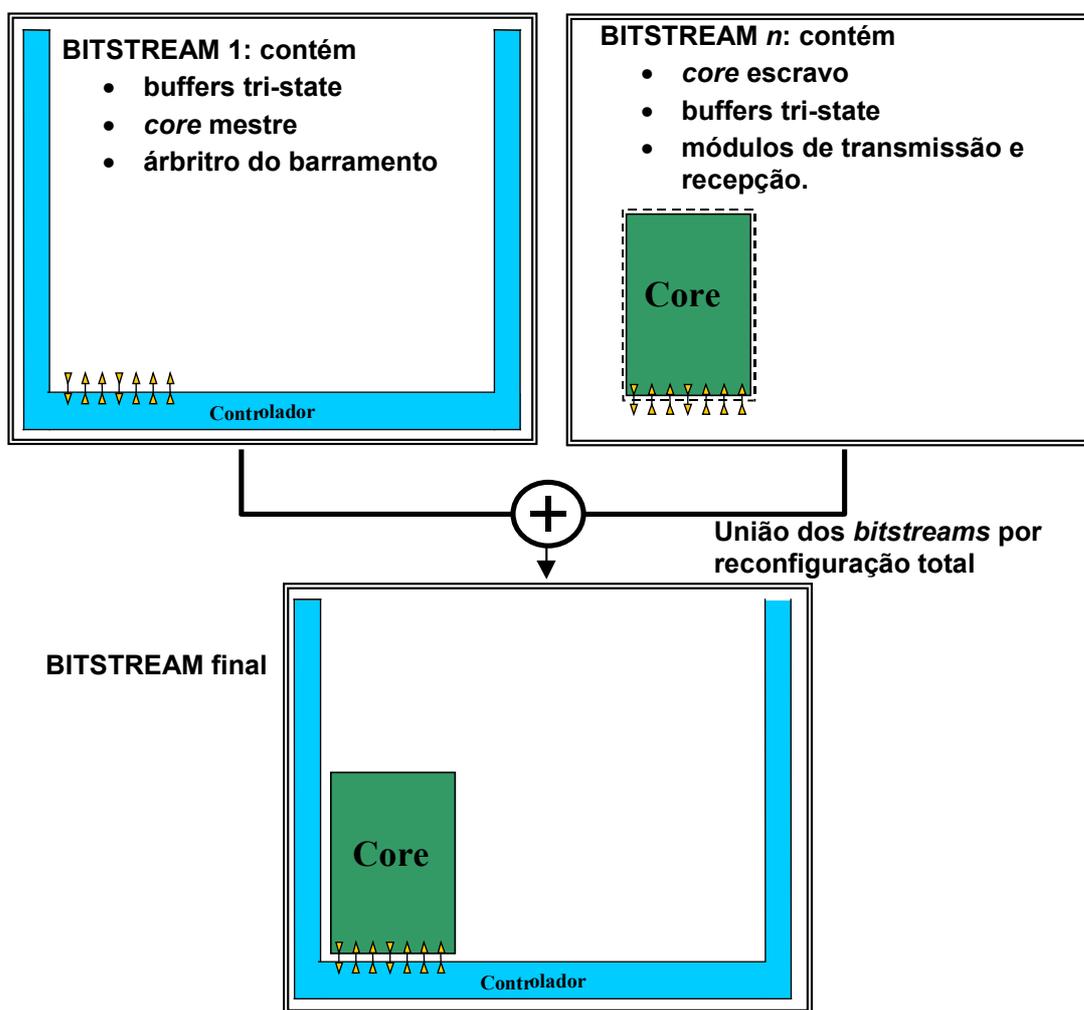


Figura 67 – União dos *bitstreams*.

No presente momento estamos validando (por prototipação) este procedimento, visando provar que é possível a união de diferentes *bitstreams* em um único, mantendo o funcionamento do sistema.

Uma vez o sistema validado por união de *bitstreams*, e configuração completa do FPGA, procede-se à última etapa do trabalho, ou seja, a geração do arquivo parcial de configuração, contendo o *core* a ser inserido no *bitstream* do controlador.

6 Conclusões e Trabalhos Futuros

O presente trabalho apresentou: (i) o estado da arte em *cores*; (ii) a distribuição de *cores* através da Internet; (iii) a reconfiguração de *hard cores*; e (iv) a proposta de um barramento intra-FPGA de comunicação entre *cores*.

No estudo do estado da arte em *cores*, foram apresentados diversos problemas inerentes à utilização de *cores* na construção de SoCs, tais como: (i) a interconexão de *cores* fornecidos por diferentes fabricantes; (ii) a proteção de propriedade intelectual do projetista do *core*; (iii) a escolha da linguagem utilizada para desenvolver o *core* e (iv) os testes de *cores*.

O desenvolvimento da ferramenta para distribuição de *cores* através da Internet foi motivado pela necessidade de organização de dados de projetos, cujo grau de dificuldade é diretamente proporcional ao tamanho do projeto. Tal ferramenta auxilia ao projetista na organização de seus projetos, permitindo também a proteção da sua propriedade intelectual.

A ferramenta de reconfiguração de *hard cores* permite a reconfiguração completa (local ou remota) de *bitstreams*, sendo utilizada para alterar os parâmetros que definem a funcionalidade do circuito (armazenados em LUTRAMs), através de uma página HTML. Isto reduz bastante o hardware de controle do circuito, diminuindo também o custo. Esta abordagem também reduz o tempo de projeto, pois o circuito não precisa ser novamente sintetizado. A reconfiguração remota é factível de ser implementada e tem aplicação industrial, porém, ainda não está sendo explorada amplamente.

A reconfiguração parcial de FPGAs *Virtex* ainda não é explorada, embora ela seja possível, devido à inexistência de aplicações comerciais que a utilizem e, principalmente, devido à ausência de ferramentas de CAD para suporte. Além do mais, ela só é realmente útil se conectar *cores* diretamente em um barramento interno ao FPGA. Tal barramento ainda não existe e seu desenvolvimento é problemático, devido ao número reduzido de recursos internos ao FPGA, tais como *buffers tri-state* e roteamento entre eles. Conclui-se desta forma que a reconfiguração parcial em FPGAs comerciais é praticamente **inviável**, dada a falta de suporte arquitetural e de ferramentas. Todo o trabalho de geração de *cores* para reconfiguração teve que ser realizado manualmente para que os *cores* fossem posicionados e tivessem seu roteamento em locais adequados.

A contribuição científica deste trabalho pode ser resumida em 3 aspectos: (1) desenvolvimento de ferramenta para distribuição de *cores*; (2) ferramenta para reconfiguração de *hard cores*; (3) proposta de uma interface de comunicação de *cores*, já validada funcionalmente, e em fase de prototipação em FPGAs.

Como trabalhos futuros, cita-se: (i) concluir a prototipação do barramento, utilizando reconfiguração parcial do dispositivo FPGA; (ii) sistematizar o processo de interconexão entre *cores*, através de uma ferramenta de CAD; (iii) propor um barramento mais genérico, tendo, por exemplo, maior largura de banda, múltiplos árbitros, tolerância a falhas; (iv) estudar e

propor modelos para dispositivos reconfiguráveis mais adequados à utilização de barramentos internos e reconfiguração parcial; e (v) incluir outros mecanismos de proteção de propriedade intelectual no ambiente de distribuição de *cores*, como simulação distribuída.

7 Bibliografia

- [ALD01a] ALDEC, INC. **Code Coverage for Active-HDL**. Disponível por WWW em <http://www.aldec.com/activehdl/codecoverage.htm>, 2001.
- [ALD01b] ALDEC, INC. **Active-HDL**. Disponível por WWW em <http://www.aldec.com/ActiveHDL/default.htm>, 2001.
- [ALT01a] ALTERA, CO. **IP MegaStore**. Disponível por WWW em <http://www.altera.com/products/ip/ipm-index.html>, 2001.
- [ALT01b] ALTERA, CO. **MAX+PLUS II AHDL, Version 6.0**. User's Manual, 2001.
- [APA01] APACHE SOFTWARE FOUNDATION. **Apache HTTP Server Project**. Disponível por WWW em <http://www.apache.org>, 2001.
- [ARE01] AREXSYS, INC. **Archimate V2.08, User Manual**. Disponível por WWW em <http://www.arexsys.com>, 2001.
- [ARM00] ARM, Ltda. **AMBA Specification Overview**. Disponível por WWW em <http://www.arm.com/Pro+Peripherals/AMBA>, 2000.
- [ATM00] ATMEL. **At40k Series Configuration**. Disponível por WWW em <http://www.atmel.com/atmel/postscript/doc1009.ps.zip>, 2000.
- [BAL99] BALARIN, F. **POLIS A design environment for control-dominated embedded systems - version 0.4**. User's Manual, Berkeley, USA, 1999.
- [BAR99] BARNA, C.; ROSENSTIEL, W. **Object-oriented reuse methodology for VHDL**. Design, Automation and Test in Europe, 1999, pp. 689 -693.
- [BAR98] BARTON, D. **Systems Level Design Language Committee**. Disponível por WWW em <http://www.inmet.com/SLDL/sysroad/index.html>, 1998.
- [BER00a] BERGAMASCHI, R. A.; LEE, W. R. **Designing systems-on--chip using cores**. 37th Design Automation Conference, 2000, pp. 420-425.
- [BER00b] BERGERON, J. **Writing testbenches: functional verification of HDL models**. Kluwer Academic Publishers, Norwell, MA, 2000, 384 p.
- [BER01] BERGAMASCHI, R. A.; BHATTACHARYA, S.; WAGNER, R.; FELLEZ, C.; MUHLADA, M.; WHITE, F.; DAVEAU, J. M.; LEE, W. R. **Automating the design of SOCs using cores**. IEEE Design & Test of Computers, Volume: 18(5), Sept-Oct 2001, pp. 32 -45.
- [CAL98] CALAZANS, N. L. V. **Projeto Lógico Automatizado de Sistemas Digitais Seqüenciais**. Imprinta Gráfica e Editora Ltda. 11^a Escola de Computação,

Universidade Federal do Rio de Janeiro - UFRJ, Rio de Janeiro, 20-24 Jul., 1998, 318 p.

- [CAP01] CAPPELATTI, E. A. **Implementação do Padrão de Barramento PCI para Interação Hardware/Software em Dispositivos Reconfiguráveis**. Dissertação de Mestrado. Programa de Pós-Graduação em Ciência da Computação. PUCRS, 2001, 81 p.
- [CAS97] CASE, J., GUPTA, N., MITTAL, J., RIDGEWAY, D.: **Design Methodologies for Core-Based FPGA Designs**. Xilinx Application Notes, Abril, 1997.
- [CEC99] CECS - Center for Embedded Computer Systems, Dept. of Information and Computer Science, University of California, Irvine. **SpecC System**. Disponível por WWW em <http://www.ics.uci.edu/~specc/index.html>, 1999.
- [DAL00a] DALPASSO, M.; BOGLIOLO, A.; BENINI, L. **Hardware/software IP protection**. 37th Design Automation Conference, 2000, pp. 593 –596.
- [DAL00b] DALPASSO, M.; BOGLIOLO, A.; BENINI, L.; FAVALLI, M. **Virtual fault simulation of distributed IP-based designs**. Design, Automation and Test in Europe, 2000, pp. 99 –103.
- [DES00] DESIGN-REUSE. Disponível por WWW em <http://www.design-reuse.com>, 2000.
- [FAR00] FARINES, J., FRAGA, J., OLIVEIRA, R. **Sistemas de Tempo Real**. Escola de Computação 2000, USP, São Paulo, Brasil, 24-28 Jul., 2000.
- [GAJ00] GAJSKI, D., ZHU, J., DÖMER, R., GERSTLAUER, A., ZHAO, S. **SpecC: Specification Language and Methodology**. Kluwer Academic Publishers, Norwell, MA, 2000, 336 p.
- [GUC00] GUCCIONE, S. A., LEVI, D., SUNDARARAJAN, P. **JBits: A Java-based Interface for Reconfigurable Computing**. 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD), 2000.
- [GUP97] GUPTA, R. K., ZORIAN, Y. **Introducing Core-Based System Design**. IEEE Design & Test of Computers, Volume: 14(4), Out-Dez 1997, pp. 15-25.
- [HAA99] HAASE, J., OBERTHÜR, T., OBERWESTBERG, M. **Design Methodology for IP Providers**. Kluwer Academic Publishers, Norwell, MA, 1999.
- [HEL97] HELAINEN, R., OLUKOTUN, K. **Java as a Specification Language for Hardware-Software Systems**. International Conference on Computer Aided Design, 1999, pp. 690-697.
- [IBM99] IBM. **The CoreConnect™ Bus Architecture**. Disponível por WWW em http://www.chips.ibm.com/products/coreconnect/docs/crcon_wp.pdf, 2000.

- [IEE88] IEEE. **IEEE Standard VHDL Language: Reference Manual**. IEEE, New York, NY, 1988.
- [KAH98] KAHNG, A. B.; LACH, J.; MANGIONE-SMITH, W. H.; MANTIK, S.; MARKOV, I. L.; POTKONJAK, M.; TUCKER, P.; WANG, H.; WOLFE, G. **Watermarking techniques for intellectual property protection**. 35th Design Automation Conference, 1998, pp. 776 -781.
- [KEA00] KEATING, M. **Reuse methodology manual for system-on-a-chip designs**. Second Edition. Kluwer Academic Publishers, Norwell, MA, 2000.
- [LIA97] LIAO, S.; TJIANG, S.; GUPTA, R. **An Efficient Implementation Of Reactivity For Modeling Hardware In The Scenic Design Environment**. 34th Design Automation Conference, 1997. pp. 70 -75.
- [MAD97] MADISSETTI, V. K., SHEN L. **Interface Design for Core-Based Systems**. IEEE Design & Test of Computers, Volume: 14(4), Out-Dez 1997, pp. 45-51.
- [MAZ92] MAZZOR, S., LANGSTRAAT, P. **A Guide to VHDL**. Kluwer Academic Publishers, Norwell, MA, 1992.
- [MCG97] McGEER, R. **The V++ Systems Design Language**. A talk sponsored by the Stanford CAD group, May, 1997.
- [MES01] MESQUITA, D. **Contribuições para Reconfiguração Parcial, Remota e Dinâmica de FPGAs**. Dissertação de Mestrado. Programa de Pós-Graduação em Ciência da Computação. PUCRS, 2001, 103 p.
- [OPE01] OPENCORES.ORG. **Wishbone SoC Interconnection**. Disponível por WWW em http://www.opencores.com/press/pr_8jan2001.shtml, 2001.
- [PAL00] PALMA, J. C. S. **Técnicas para Implementação de Sistemas Digitais Reutilizáveis**. Trabalho Individual I. Programa de Pós-Graduação em Ciência da Computação. Mestrado, PUCRS. Set., 2000, 54 p.
- [RIN97] RINCON, A. M. et. al. **Core Design and System-on-a-chip Integration**. IEEE Design & Test of Computers, Volume: 14(4), Out-Dez 1997, pp. 26-35.
- [ROX00] JAMES-ROXBY, P.; GUCCIONE, S. A. **Automated extraction of run-time parameterisable cores from programmable device configurations**. IEEE Symposium on Field-Programmable Custom Computing Machines, 2000, pp. 153 -161.
- [SCH97] SCHALLER, R. R. **Moore's Law: Past, Present and Future**. IEEE Spectrum, Volume: 34(6), pp. 52-59, Jun., 1997.
- [SHA98] SHAH, N. **The Challenges of Doing a PCI Design in FPGAs**. Xilinx Application Notes. Abril, 1998.

- [SIA99] SIA - **Semiconductor Industry Association**, 1999. Disponível por WWW em http://notes.sematech.org/1999_SIA_Roadmap/Home.htm, 1999.
- [SIL01] SILICORE Co., **WISHBONE**. Disponível por WWW em <http://www.silicore.net/pdfiles/wishbone.pdf>, 2001.
- [SYN01a] SYNOPSYS Inc. **Functional Specification for SystemC 2.0, version 2.0-N**. Disponível por WWW em <http://www.SystemC.org>, 2001.
- [SYN01b] SYNOPSYS Inc., **DesignWare Foundation Library**. Disponível por WWW em http://www.synopsis.com/products/designware/dw_lib.html, 2001.
- [THO91] THOMAS, D., MOORBY, P. **The Verilog Hardware Description Language**. Kluwer Academic Publishers, 1991.
- [VCC01] VCC – VIRTUAL COMPUTER CORPORATION. **The Virtual Workbench**. Disponível por WWW em <http://www.vcc.com/Papers/vw300.pdf>, 2001.
- [VIL97] VILLASENOR, J., W. H. Mangione-Smith. **Configurable Computing**. Scientific American, p. 54-59, Jun., 1997.
- [VSI00] VSI - *Virtual Socket Interface Alliance*. **Architecture Documentation** Disponível por WWW <http://www.vsi.org/library.htm>, 2000.
- [XIL00] XILINX INC. **Virtex Series Configuration Architecture User Guide**. Disponível por WWW em <http://www.xilinx.com/xapp/xapp151.pdf>, 2000.
- [XIL01a] XILINX, INC. **Xilinx CORE Generator**. Disponível por WWW em <http://www.xilinx.com/products/logicore/coregen>, 2001.
- [XIL01b] XILINX, INC. **Xilinx IP Center**. Disponível por WWW em <http://www.xilinx.com.br/ipcenter>, 2001.
- [XIL01c] XILINX, INC. **Chipscope Software and ILA Cores User Manual**. Disponível por WWW em http://support.xilinx.com/products/software/chipscope/chipscope_ila_um.pdf, 2001.
- [XIL01d] XILINX, INC. **Libraries Guide**. Disponível por WWW em http://support.xilinx.com/support/sw_manuals/3_1i/download/libguide.zip, 2001.
- [XIL01e] XILINX INC. **The JBits 2.8 SDK for Virtex**. Disponível por WWW em ftp://customer:xilinx@ftp.xilinx.com/download/JBits2_8.exe, 2001.
- [XIL01f] XILINX INC. **Floorplanner Guide, v3.1i**. 2001.
- [ZOR99] ZORIAN, Y.; MARINISSEN, E.J.; DEY, S. **Testing embedded-core-based system chips**. IEEE Computer , Volume: 32(6), pp. 52-60, Jun., 1999.