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

Reconfiguração Parcial e Dinâmica para Núcleos de Propriedade Intelectual

por

Eduardo Wenzel Brião

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

Orientador: Prof. Dr. Ney Laert Vilar Calazans

Porto Alegre, janeiro de 2004.

Sumário

LISTA DE FIGURAS v

LISTA DE TABELAS ix

LISTA DE SÍMBOLOS E ABREVIATURAS xi

RESUMO xix

ABSTRACT xxi

Capítulo 1: Introdução 1

1.1 Motivação . . . 4
1.2 Objetivos . . . 5
1.3 Organização do Volume . . . 5

Capítulo 2: Estado da Arte em SDRs 7

2.1 Definições Básicas de Reconfiguração . . . 7
2.2 Propostas de SDRs . . . 8
2.2.1 Reconfiguração parcial em nível de circuito integrado . . . 9
2.2.2 Reconfiguração parcial em nível de placa . . . 14
2.3 Formas de Reconfiguração . . . 17
2.3.1 Reconfiguração baseada em alterações incrementais . . . 17
2.3.2 Reconfiguração baseada na inserção/remoção de núcleos . . . 18
2.4 Requisitos para Habilitar SDRs . . . 18
2.5 Suporte de Software para Reconfiguração Parcial . . . 20
2.5.1 Conjunto de classes JBits . . . 20
2.5.2 Ferramenta JRTR (*Java Run-Time Reconfiguration*) . . . 22
2.5.3 Ferramenta PARBit . . . 22
2.5.4 Gerador de arquivo de configuração - JPG . . . 24
2.5.5 Ferramentas alternativas desenvolvidas por Möller . . . 26

Capítulo 3: Reconfiguração Baseada em Núcleos e Interconexão de Núcleos **31**
3.1 Métodos Habilitadores 31
3.1.1 Barramento proposto por Palma 31
3.1.2 Reconfiguração parcial via Projeto Modular 33
3.2 Métricas para Avaliação de SDRs 34
3.3 Padronização da Comunicação Intra-chip 36
3.3.1 Abordagem centrada no meio de comunicação 36
3.3.2 Abordagem centrada na interface de comunicação 37
3.3.3 Protocolo OCP 38
3.4 Contexto do Trabalho 44

Capítulo 4: Fluxo de Projeto Modular **47**
4.1 Preparação dos Módulos e Fases do Projeto Modular 48
4.1.1 Preparação da entrada do fluxo 48
4.1.2 Ferramental para a execução do fluxo 53
4.1.3 Fase Orçamento Inicial 55
4.1.4 Fase Implementação do Módulo Ativo 56
4.1.5 Fase Montagem Final 57
4.2 Contribuições 58
4.3 Crítica do Processo 60
4.4 Proposta de Ferramental 61

Capítulo 5: Validação do Fluxo Modular de Projeto **65**
5.1 Descrição da Plataforma de Reconfiguração Alvo 65
5.2 Estudos de Caso 66
5.2.1 Calculadora reconfigurável 67
5.2.2 Contador reconfigurável 68
5.2.3 Contador reconfigurável com interface OCP 69
5.2.4 Contador com duas áreas reconfiguráveis 70
5.2.5 Controlador de memória SRAM 71
5.3 Análise do tempo de reconfiguração 71
5.3.1 Tempos de reconfiguração 71

Capítulo 6: Processador Reconfigurável **77**
6.1 O Processador R8 77
6.2 Características Adicionadas ao Processador R8 - Processador R8R 80

6.3 Validação por Simulação 83
6.4 Prototipação 89
6.5 Análise dos Resultados na Execução do Processador R8R 93
6.5.1 Coprocessador de multiplicação 93
6.5.2 Coprocessador de divisão 94
6.5.3 Coprocessador de raiz quadrada 94

Capítulo 7: Conclusão e Trabalhos Futuros 97

REFERÊNCIAS BIBLIOGRÁFICAS 101

Apêndice A: Códigos-fonte para o Processador R8R 109
A.1 Software para teste de todos os coprocessadores reconfiguráveis 109
A.2 Multiplicação em software 112
A.3 Divisão em software 113
A.4 Raiz quadrada em software 115

Apêndice B: Considerações quanto a Erros de Roteamento 117
B.1 Impossibilidade da Geração de Sinais de Lógica Global 117
B.2 Comportamento Indesejável do Projeto: Escorregamento de Relógio 120

Lista de Figuras

1.1 Arquitetura genérica de um SoC. 3

2.1 Estrutura de interconexão das células no interior do FPGA Xilinx XC6200. . 10
2.2 FPGA AT40K da ATMEL, onde núcleos IP armazenados em memória configuram o FPGA em tempos diferentes. 11
2.3 Esquema de um CLB do FPGA XCV300. 11
2.4 Disposição em colunas dos elementos do FPGA Virtex XCV300. 12
2.5 Arquitetura de um dispositivo da Família VirtexII. 12
2.6 Arquitetura do SPLASH 2 15
2.7 Arquitetura CHESS constituída por um arranjo matricial de ULAs conectadas entre si. 16
2.8 Arquitetura do sistema reconfigurável *Amalgam*. 16
2.9 Arquitetura do *Cluster* Reconfigurável. 17
2.10 Núcleos IP reconfiguráveis conectados a uma interface de comunicação. 18
2.11 Fluxo de projeto do JBits para prover controle e dados de configuração. . . . 21
2.12 Aplicação escrita em JAVA desenvolvida pelo usuário. Percebe-se que há diferentes níveis de abstração para a geração de arquivos de configuração parciais, ocultando detalhes do usuário. 21
2.13 Fluxo de projeto do JRTR. 23
2.14 FPGA com uma área hachurada, a qual contem um módulo que será gerado no arquivo reconfigurável pela ferramenta PARBIT. 23
2.15 Fluxo de projeto e execução da ferramenta PARBIT. 25
2.16 Fluxo de projeto do gerador de arquivos de configuração JPG. 26
2.17 Interface gráfica do *CoreUnifier*. 28

3.1 Estrutura de um SoC usando o método proposto por Palma [PAL02]. Neste, duas camadas de *buffers* que possibilitam a implementação do controlador no FPGA. 32
3.2 Estrutura do invólucro para inserção de núcleos IP no método proposto por Palma [PAL02]. 32

3.3 Visão geral do fluxo de reconfiguração dinâmica e parcial baseada no Projeto Modular. 33
3.4 Exemplo da desvantagem principal de comunicação baseada na abordagem centrada no meio de comunicação. Neste modelo [OCP02], um núcleo IP denominado A com protocolo de comunicação adaptado a um barramento X pode ser conectado diretamente ao barramento X. No entanto, para conectar o mesmo núcleo a um barramento Y, (protocolo de comunicação diferente do barramento X) são necessárias alterações no protocolo deste núcleo para adaptar este ao barramento Y. 36
3.5 Exemplo de uma arquitetura baseada na abordagem *core-centric*. 37
3.6 Sistema usando um barramento e núcleos empacotados com instâncias OCP. . 39
3.7 Diagrama do protocolo de comunicação de uma transferência de escrita e leitura de dados. 42
3.8 Ferramenta *CoreCreator* para validação e certificação do protocolo OCP aplicado em núcleos IP ou sistemas. A ferramenta valida os módulos com OCP através da geração de tráfego de dados. 43
3.9 Fluxo de desenvolvimento e implementação de sistemas dinamicamente reconfiguráveis de acordo com o arcabouço PADREH proposto. A área hachurada representa o agrupamento das tarefas propostas neste trabalho. 45

4.1 Visão geral das fases do Projeto Modular. 48
4.2 Esquemático de uma bus macro, formada por 8 tristates. Os sinais LT e RT são responsáveis pelo controle de acesso aos fios compartilhados pelos módulos, denominados O[3:0]. Um dos conjuntos de sinais LI e RI pode ser usado a cada instante como fonte de informação a enviar para o módulo do outro lado da bus macro. Cada *bus macro* provê um barramento de comunicação de 4 bits entre os módulos. 49
4.3 Comunicação de dois módulos (fixos/reconfigurável e reconfigurável) através de uma *bus macro*. 50
4.4 Situações particulares para implantação da bus macro usando pinos E/S. . . . 51
4.5 Conexão do pinos de E/S no módulo A através de bus macro. 51
4.6 Situações particulares para implantação da bus macro na comunicação entre módulos. 52
4.7 Conexão entre os módulos B/A, A/C e B/C através de bus macro 52
4.8 Estrutura de diretórios recomendada pela Xilinx para a execução do Fluxo de Projeto Modular. 53
4.9 Fluxo de execução da fase de Orçamento Inicial do Projeto Modular. 56
4.10 Ferramenta floorplanner. Usada para edição de restrições para roteamento, posicionamento de elementos lógicos e definição da planta-baixa do FPGA. . . 57

4.11 Fluxo de execução da fase Implementação do Módulo Ativo do Projeto Modular. 58
4.12 Fluxo de execução da fase Montagem Final do Projeto Modular. 59
4.13 Interface gráfica da ferramenta *MDLaucher*. 62

5.1 Diagrama de blocos da plataforma V2MB1000. 66
5.2 Bloco lógico de uma calculadora reconfigurável de duas operações. 67
5.3 Diagrama de blocos de um contador reconfigurável. 69
5.4 Diagramas de tempos do contador OCP. 70
5.5 Diagrama de blocos de um controlador de memória SRAM conectada a um módulo cliente. 72
5.6 Gráfico das medidas dos tempos de reconfiguração do modo de configuração mais rápida suportada pelo FPGA Virtex II (tempo de reconfiguração versus freqüência para transmissão de dados). 72
5.7 Gráfico da estimativa dos tempos de reconfiguração para o modo de configuração mais rápido suportado pelo FPGA Virtex II XC2V1000 (tempo de reconfiguração em microssegundos versus tamanho dos bitstreams em kilobytes). 73
5.8 Amostragem do sinal *CCLK* durante uma configuração de FPGA usando o cabo Multilinx. 74
5.9 Ganho obtido no tempo de reconfiguração utilizando o emulador desenvolvido por Carvalho em relação ao modo *slave selectMAP*. Nota-se que houve uma redução, em média, de 94.4% do tempo de reconfiguração em relação ao modo *slave selectMAP*. 76

6.1 Diagrama de blocos da relação entre o processador R8 e a memória. 79
6.2 Uma visão geral do sistema R8R. 81
6.3 Simulação funcional do coprocessador *4-avg*. 84
6.4 Diagrama de tempos do coprocessador *2-avg*. 85
6.5 Diagrama de tempos da execução da instrução SELR. 87
6.6 Diagrama de tempos da execução da do módulo *sqrt*. 87
6.7 Diagrama de tempos da execução da do módulo *multi*. 88
6.8 Diagrama de tempos da execução da do módulo *div*. 89
6.9 Telas de aplicação do software que envia dados oriundos de um PC hospedeiro para o processador R8R. 90
6.10 Gráfico comparativo entre os tempos de reconfiguração e execução do coprocessador de multiplicação com o tempo de execução do software ambos em relação ao número de operações. 94
6.11 Gráfico comparativo entre os tempos de reconfiguração e execução do coprocessador de divisão e o tempo de execução do software ambos em relação ao número de operações. 95

6.12 Gráfico comparativo entre os tempos de reconfiguração e execução do coprocessador de raiz quadrada e o tempo de execução do software ambos em relação ao número de operações. 95

B.1 Esquemático esquivalente do código VHDL apresentado para a ferramenta de posicionamento e roteamento possa realizar as conexões entre os componentes. 121

B.2 Um projeto hipotético onde o módulo fixo está “espremido” à esquerda do FPGA. Os sinais que conectam os componentes DCM e BUFGMUX atravessam a o limite entre a área reconfigurável e a área fixa. 122

B.3 Esquemático mostrando como devem ser fixadas as bus macros no FPGA para permitir que os sinais possam interconectar os componentes DCM e buffer sem comportamentos indesejáveis na prototipação. 123

Lista de Tabelas

2.1 Características das ferramentas de suporte a reconfiguração parcial e dinâmica. 29

3.1 Sinais básicos do OCP. 40
3.2 Comandos de codificação do sinal *MCmd*. 41
3.3 Comandos de codificação do sinal *SResp*. 41

5.1 Tempos de reconfiguração de bitstreams de tamanhos diferentes usando cabo USB. 74

6.1 Instruções adicionadas para executar funções associadas ao(s) coprocessador(es) reconfigurável(is). 82
6.2 Identificação dos coprocessadores reconfiguráveis através de endereços específicos. 84
6.3 Exemplo de seqüência de instruções reconfiguráveis utilizado no software de teste (Apêndice A.1). 91
6.4 Conteúdo da memória do R8R logo após feita a reconfiguração parcial e execução do software. 93

Lista de Símbolos e Abreviaturas

ACU *Array Control Unit*

AMBA *Advanced Microcontroller Bus Architecture*

API *Application Programming Interface*

ASIC *Application Specific Integrated Circuit*

BRAM *Block Random Access Memory*

CAD *Computer Aided Design*

CLB *Configurable Logic Block*

CPLD *Complex Programmable Logic Device*

CRC *Cyclic Redundancy Check*

DCM *Digital Clock Manager*

DCR *Device Control Register*

DHP *Dynamic Hardware Plugin*

DISC *Dynamic Instruction Set Computer*

DSP *Digital Signal Processor*

FPGA *Field Programmable Gate Array*

GAPH *Grupo de Apoio ao Projeto de Hardware*

GPP *General Purpose Processor*

HDL *Hardware Description Language*

ICAP *Internal Configuration Access Port*

IP *Intellectual Property*

IR *Instruction Register*

ISE *Integrated Software Environment*

JRTR *Java Run-Time Reconfiguration*

LUT *Look-Up Table*

NCD *Xilinx Native Circuit Description*

NGD *Xilinx Netlist Generic Database*

NIC *Network Interface Card*

NMC *Netlist Macros*

NRE *Non-Recurring Engineering*

OCP *Open Core Protocol*

OPB *On-chip Peripheral Bus*

PADREH *Partial and Dynamic Reconfiguration of Hardware Project*

PAR *Place and Route*

PC *Program Counter*

PCF *Physical Constraints File*

PIM	*Physically Implemented Modules*
PLB	*Processor Local Bus*
R8R	*R8 Reconfigurável*
RAM	*Random Access Memory*
SDR	*Sistema Digital Reconfigurável*
SIMD	*Single Instruction Multiple Data*
SoC	*System-on-Chip*
SoPC	*System-on-Programmable-Chip*
SP	*Stack Pointer*
SRAM	*Static Random Access Memory*
SRS	*Semiconductor Reuse Sector*
STL	*Sonics Transaction Language*
TRCE	*Timing Reporter and Circuit Evaluator*
UART	*Universal Asynchronous Receiver/Transmitter*
UCF	*User Constraints File*
UCP	*Unidade Central de Processamento*
ULA	Unidade Lógica Aritmética
USB	*Universal Serial Bus*
VLSI	*Very Large Scale Integration*

VSIA *Virtual Socket Interface Alliance*

XHWIF *Xilinx HardWare InterFace*

"A receita para a eterna ignorância é muito simples e efetiva: esteja satisfeito com suas opiniões e contente com o seu conhecimento".
Elbert Hubbard (1856-1915)

Agradecimentos

Em primeiro lugar, a Deus pelas coisas boas da vida, por minha saúde, pela minha gana e esforço no trabalho e na luta constante pela sobrevivência nos dias atuais. Agradeço a Deus por tudo que me concebeste, desde o ar que respiro até os maiores de todos os tesouros: o amor, a fraternidade e a humildade.

Aos meus pais, Lea Beatriz e Paulo Renato pelo incentivo, carinho, confiança e amizade, que mesmo estando distante, batalharam e rezaram pelo meu sucesso.

À minha avó, Suely Ceiglinski pela amizade, fraternidade, companheirismo nos cafés da tarde e apoio para o meu sucesso profissional e acadêmico.

À minha namorada, Carla Dorini pelo apoio, compreensão nos momentos de minha ausência e por seu imensurável amor e amizade nos bons e maus momentos deste período que estive morando em Porto Alegre.

Ao meu orientador, Ney Laert Vilar Calazans pelo empenho, amizade e acima de tudo, por mostrar-me o caminho da pesquisa acadêmica para que esta seja usada de maneira direta ou indireta para favorecer a sociedade como um todo.

Ao professor Fernando Gehm Moraes, pela orientação adicional, pela amizade, companheirismo e por estar disponível, quando precisei de auxílio.

Aos colegas Edson Moreno, Ewerson Carvalho e Luciano Ost pelo coleguismo e amizade no decorrer do curso. Pelos jogos de futebol, pelas conversas, pelo dia-a-dia proporcionado por estes três caras. Foram importantes no meu processo de adaptação em Porto Alegre.

Agradecimento especial para Leandro Möller, Aline Vieira, Daniel Camozzato, Luis Ries, Thiago e Rodrigo Wertonge pelas eventuais ajudas extremamente eficazes no desenvolvimento de trabalhos no grupo GAPH.

Agradeço a Kamal Patel, engenheiro da Xilinx, por me proporcionar informações valiosas para o desenvolvimento deste trabalho.

A todos que, de maneira direta ou indireta, contribuíram para o desenvolvimento deste trabalho.

À PARKS e ao CNPq por terem viabilizado este trabalho através do suporte financeiro e terem garantido a minha alimentação e manutenção em Porto Alegre.

Aos demais colegas que conheci na PUCRS e outras instituições em Porto Alegre ou fora dela, pela oportunidade de conhecer pessoas tão especiais.

Aos meus colegas de quarto que me incentivaram e trocaram conhecimentos e idéias, sempre buscando o bem comum.

Ao Estado do Rio Grande do Sul pela sua existência, pela sua cultura, pelo seu povo e por nossa identidade gaúcha, digna e cheia de glórias! Obrigado, meus pais, por proverem meu nescimento em terras gaúchas. Obrigado Deus, por ser simplesmente gaúcho!

Resumo

A reconfiguração de hardware apresenta-se como uma tecnologia promissora para aumentar a flexibilidade e o poder computacional de sistemas digitais complexos. Já existem no mercado dispositivos comerciais de alta complexidade que habilitam a reconfiguração de hardware de forma dinâmica e parcial, ou seja, dispositivos VLSI cujo hardware pode ser parcialmente alterado em tempo de execução, enquanto o restante do dispositivo continua a operar normalmente. Contudo, existe uma série de carências, sobretudo em ferramentas e fluxos de projeto, que inviabilizam hoje a utilização de reconfiguração parcial e dinâmica de dispositivos de hardware em larga escala. A alteração de dispositivos reconfiguráveis pode ser dividida em duas classes principais: alterações incrementais, onde a lógica, as interfaces de entrada/saída ou o roteamento de uma pequena porção do dispositivo é mudada, e a inserção e/ou remoção dinâmica de módulos complexos, onde blocos inteiros de lógica e roteamento são alterados ou susbtituídos. A primeira classe de alterações é considerada tecnologia dominada, sendo útil em aplicações restritas. A segunda é mais complexa, de aplicação mais ampla e carece de suporte adequado. A principal contribuição deste trabalho é a proposta de parte de uma infra-estrutura de suporte para o projeto e implementação de sistemas digitais reconfiguráveis complexos sobre dispositivos comerciais. Especificamente, propõe-se e implementa-se um método de geração de arquivos de configuração parciais que correspondem à implementação física de núcleos de propriedade intelectual arbitrariamente complexos. A geração é realizada de tal forma a habilitar que estes núcleos possam ser inseridos ou removidos de um dispositivo reconfigurável em tempo de execução, através de procedimentos mecânicos de reconfiguração parcial e dinâmica. O método foi desenvolvido a partir de adaptação e extensão de técnicas propostas por um fornecedor de dispositivos reconfiguráveis. Como contribuição adicional, foi proposta e implementada uma ferramenta de software para automatizar parcialmente o complexo processo de aplicação do método, aumentando o nível de abstração em que um projetista de um sistema reconfigurável atua. Um conjunto de estudos de caso de implementação de sistemas reconfiguráveis foi empregado para validar o método e seu emprego.
Palavras-chave: Reconfiguração parcial e dinâmica, Projeto Modular, Infra-estrutura de reconfiguração, interfaces de comunicação padronizadas, sistemas digitais reconfiguráveis, núcleos de propriedade intelectual, reuso.

Abstract

Hardware reconfiguration stands as a promising new technology to enable the increase of the flexibility and computational power of VLSI digital systems. Complex reconfigurable devices are already available in the market. Some of these support partial and dynamic reconfiguration, which means that part of the device can be changed while the rest of it remains operational. However, there are several features lacking in the design support of reconfigurable systems, which justifies why this technology has not yet become mainstream. Changing the hardware of reconfigurable devices can be achieved by means of two classes of techniques: incremental changes, where a small piece of logic, input/output characteristics and/or routing is altered, and IP core insertion/removal, where arbitrarily large blocks of the integrated circuit are replaced. The first is well known, but has limited scope of application. The second is more complex, has a larger spectrum of applicability and lacks adequate support. The main contribution of this work is the proposition of part of an infrastructure to enhance the design and implementation support of reconfigurable digital systems in commercial devices. Specifically, a method to generate partial and dynamic device reconfiguration files is proposed. Each of these files correspond to the implementation of an arbitrarily complex IP core. The generation process enables that these IP cores be inserted in a reconfigurable device at execution time through the use of standard partial and dynamic device reconfiguration techniques. The method has been developed by adapting and extending techniques proposed by a reconfigurable device vendor. As an additional contribution, a software tool was proposed and implemented to partially automate the complex process of applying the method. This increases the abstraction level in which the reconfigurable system designer interacts with the implementation process. A set of reconfigurable system case studies has been employed to validate the method and its use.

Keywords: Partial and dynamic reconfiguration, Modular Design, Reconfiguration framework, standard communication interfaces, reconfigurable digital system, intellectual property cores, reuse.

Capítulo 1

Introdução

Durante o desenvolvimento e a execução de funções computacionais por um determinado sistema, deseja-se sempre alcançar o máximo desempenho das mesmas ao menor custo final. Porém, a tarefa de atingir tais objetivos é complexa. O custo final depende de características tais como: (i) área de silício do projeto, (ii) potência dissipada pelo sistema e (iii) tempo de projeto (custos de engenharia não recorrentes, em inglês NRE costs - *Non-Recurring Engineering costs*). A diminuição de todas estas figuras de mérito conflita diretamente com o desempenho do sistema. Para diminuir o custo final de um sistema, o mais importante atualmente, é diminuir o tempo de projeto, embora outras figuras contribuam em maior ou menor grau, de acordo com o sistema e/ou área de aplicação específica. A dificuldade de reduzir tempo de projeto pode ser atenuada através de técnicas de reuso de projeto. Para a obtenção de reuso, módulos devem ser genéricos. Porém a generalidade seguidas vezes implica perda de desempenho para realizar tarefas específicas.

Duas formas de se obter generalidade são através de sistemas *programáveis* e sistemas *reconfiguráveis*. Os primeiros possuem a propriedade de executar *software* (características dos GPPs (*General Purpose Processors*), e os últimos alcançam generalidade ao permitir que o *hardware* seja alterado de forma dinâmica, apresentando a propriedade de *reconfigurabilidade*. Um sistema que não possua nenhuma destas propriedades pode apresentar o máximo desempenho na execução de uma tarefa como no caso de ASICs (*Application-Specific Integrated Circuits*). Por outro lado, é possível conceber sistemas mistos, ao mesmo tempo programáveis e reconfiguráveis. Naturalmente estes levam a um grau de flexibilidade máximo. Exemplo de um tal sistema é descrito no Capítulo 6.

Alguns pesquisadores sugerem que sistemas reconfiguráveis podem apresentar uma série de vantagens sobre sistemas com estas características. Segundo Hauck [HAU98], a computação reconfigurável possui potencial para se tornar um paradigma de propósito geral para o desenvolvimento de sistemas computacionais. Um exemplo de tal paradigma hoje é aquele em que se emprega um microprocessador ou microcontrolador associado a software. Hauck também afirma que a flexibilidade de poder inserir novas funções não apenas pela reprogramação de

software, mas também pela reconfiguração do hardware pode resultar em maiores graus de flexibilidade para sistemas. Shirazi et. al. [SHI98] estimam que o uso da reconfigurabilidade pode constituir-se em um paradigma intermediário entre ASICs e sistemas microprocessados, apresentando mais flexibilidade que os primeiros e mais desempenho que os últimos. Segundo Vissers [VIS03], a análise e a exploração de técnicas de reconfiguração dinâmica devem prover um grande passo no sentido de viabilizar plataformas de computação reconfigurável. Ainda segundo este autor, esta viabilização se dá porque tais plataformas permitem atender a compromissos de espaço e tempo sem a necessidade de reescrever e/ou ressintetizar algoritmos.

Um exemplo de dispositivos configuráveis/reconfiguráveis são os FPGAs. FPGAs (*Field-Programmable Gate Arrays*) são circuitos integrados compostos fundamentalmente por uma matriz de elementos lógicos configuráveis, interconexão configurável entre estes e blocos de entrada e saída configuráveis. FPGAs podem ter parte de seu hardware especializado para executar funções específicas com alto desempenho (característica de ASICs). O comportamento dos módulos funcionais em um FPGA pode também ser alterado em diferentes momentos, dependendo da necessidade (característica de GPPs).

Memórias, FPGAs e microprocessadores são os três principais tipos de circuitos integrados de alto volume de produção usados como geradores de demanda de avanços tecnológicos em microeletrônica. O crescente avanço da tecnologia de implementação dos circuitos integrados (CIs) permite o desenvolvimento de dispositivos cada vez mais densos [CAL98]. Esta tecnologia de implementação viabiliza a construção de sistemas computacionais complexos integrados em um único CI o que é conhecido como SoC (*System-on-Chip*) [MAR01]. SoCs podem ser desenvolvidos combinando módulos tais como FPGAs, ASICs e GPPs em um único CI. Um SoC é composto por um ou mais processadores, memórias, módulos dedicados de hardware para realização de funções específicas, módulos de software e mesmo sub-sistemas micro-eletromecânicos, sub-sistemas ópticos e outros [JUN01]. Diversos setores da indústria aproveitam o potencial da combinação de tais tecnologias heterogêneas no mesmo CI [IBM02, TEN03, ALT04]. Além da heterogeneidade, outros fatores justificam o interesse por SoCs. Entre estes, pode-se citar a redução do tamanho final do produto, a redução de potência dissipada, o aumento de desempenho potencial e a segurança de projeto [MAR01]. A Figura 1.1, baseada na proposta de Madisetti e Shen em [MAD97], ilustra a arquitetura genérica de um SoC, conforme considerado neste trabalho.

O modelo de SoC usado aqui pressupõe estes compostos basicamente por núcleos de propriedade intelectual (denominados aqui *núcleos* IP, IPs, ou simplesmente núcleos). Núcleos IP são módulos de hardware complexos pré-caracterizados e pré-validados [BER00]. Estes devem ser reaproveitáveis, tornando viável desenvolver SoCs em tempo reduzido, gerando produtos que podem levar menos tempo para chegar ao mercado [BER00, JAC01].

SoCs podem receber denominações alternativas quando implementados sobre FPGAs. Termos como SoPC (*System-on-Programmable Chip*), SoRC (*System-on-Reconfigurable Chip*) e RSoC (*Reconfigurable System-on-Chip*) aparecem em abundância em publicações técnicas.

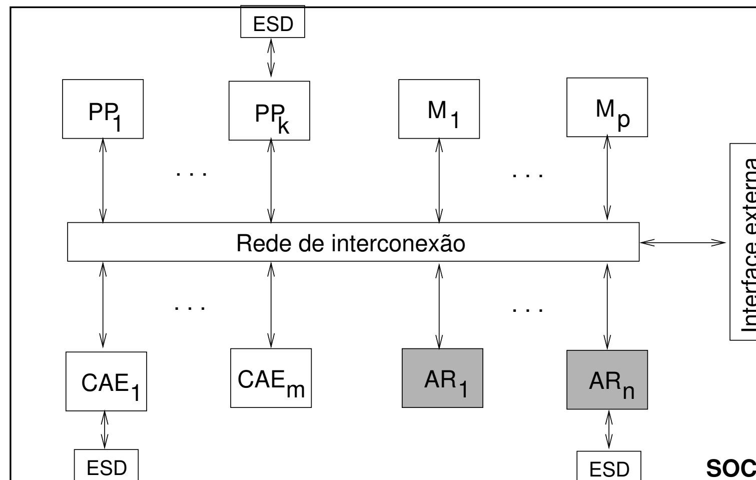

Figura 1.1: Arquitetura genérica de um SoC considerada neste trabalho. Um SoC pode conter vários processadores programáveis, blocos de memória, circuitos de aplicação específica, áreas reconfiguráveis e módulos de entrada e saída dedicados.

Um dos principais problemas enfrentados durante o projeto de SoCs é a definição de como seus IPs componentes se comunicam. Projetistas têm tradicionalmente adotado a abordagem de interconectar núcleos IP através de arquiteturas de barramento padronizadas [BER00, RIN99]. Porém, o uso destas pode limitar o reuso de IPs, pois estes devem possuir interface específica para a arquitetura de barramento escolhida. Se núcleos IP são construídos com interface específica, dificilmente podem ser empregados no contexto de outras arquiteturas de barramento sem modificações, ou mesmo em ambientes onde a comunicação não emprega barramentos. Outras limitações impostas por barramentos são a baixa escalabilidade e a seqüencialidade de acesso ao recurso de comunicação [BEN02]. Uma abordagem alternativa de padronizar a comunicação é desenvolver núcleos com uma interface padrão, que não dependam do meio de interconexão escolhido. Este pode ser uma arquitetura de barramento, padronizada ou não, ou mesmo uma interconexão mais complexa, tal como uma rede intra-chip [ZEF03, MOR03a]. Isto auxilia o projetista no desenvolvimento do seu núcleo, já que o mesmo pode concentrar-se no projeto do IP, e não na forma como este interage com o restante do sistema. Neste caso, não é necessário dominar o protocolo de comunicação específico utilizado, nem os sinais que compõem a interface de cada núcleo que venha a ser utilizado no projeto [OST02]. Tal abordagem é viabilizada, por exemplo, pela utilização de padrões tais como OCP (*Open Core Protocol*) [OCP03].

Quanto à adaptabilidade, quanto mais um núcleo é flexível e parametrizável, mais este é reusável, porém menos otimizada cada uma de suas instâncias específicas o será. Existe um compromisso entre reusabilidade e otimalidade de núcleos, segundo critérios tais como área, consumo de potência e velocidade. Um núcleo pode tornar-se um produto de sucesso se alcançar um bom compromisso entre flexibilidade e desempenho.

Quando se emprega reconfiguração, a dificuldade de se obter o compromisso ótimo entre otimalidade e reusabilidade pode ser reduzida. Isto ocorre, pois um núcleo IP ao invés de ser parametrizável, pode ser gerado em diversas versões. Para cada utilização deste IP se escolhe a versão mais adequada. Ou seja, no contexto de SoCs, reconfiguração insere um grau de liberdade a mais no projeto e no uso deste. Isto tende a melhorar a adaptabilidade do hardware ao cenário de uso, que pode assim, mesmo para uma aplicação específica, mudar dinamicamente, fazendo o SoC adaptar-se a este novo cenário.

1.1 Motivação

Conforme colocado anteriormente, reconfiguração dinâmica e parcial de hardware é uma tecnologia com potencial para se tornar amplamente utilizada. Esta em si é uma das principais motivações deste trabalho. Para poder contribuir neste sentido, é necessário dominar a tecnologia de modelagem, projeto, validação e implementação de SDRs o que se apresenta como a segunda motivação do trabalho.

Um dos motivos pelos quais SDRs podem se tornar mais amplamente utilizados é o fato de introduzirem graus de liberdade adicionais no projeto de sistemas computacionais. Estes graus surgem a partir da possibilidade de desenvolver sistemas cujo comportamento do hardware pode ser alterado dinamicamente, de forma similar ao que ocorre com software em sistemas programáveis. A reconfigurabilidade pode contribuir para a economia de recursos. Quando uma dada tarefa pode ser quebrada em várias fases, uma configuração diferente pode ser carregada para cada fase seqüencialmente [DEH00] operando de forma análoga à memória virtual em sistemas operacionais. Dessa forma, o tamanho do sistema pode ser menor que o necessário para implementar uma funcionalidade total, o que implica redução de custos e redução de área do dispositivo. Assim, uma outra motivação para o presente trabalho é disponibilizar dados quantitativos sobre compromissos espaço-temporais na implementação de sistemas reconfiguráveis e sistemas programáveis.

Um exemplo de emprego de reconfigurabilidade ocorre em aplicações espaciais. Alterações indesejadas na funcionalidade dispositivos eletrônicos no ambiente hostil do espaço provenientes de radiação podem acarretar erros severos na funcionalidade de tais dispositivos. Reconfigurabilidade pode ser usada, por exemplo, para corrigir erros no circuito e torná-lo tolerante a falhas [BEZ01, BEZ00, CAR00]. Existem casos em que circuitos implementados em áreas de silício danificadas por radiação são substituídos por equivalentes em outra área de um dispositivo. Pode-se elaborar módulos de teste que verificam determinados circuitos e, se estes

estão danificados, reconfiguram o FPGA [BEZ01]. Outras motivações são a economia de área em uma aplicação espacial e atualizações remotas feitas no hardware que implementa uma aplicação espacial realizadas pela reconfiguração. Fabricantes como a Xilinx e a Actel provêem FPGAs que podem ser usados em aplicações espaciais [XIL01, ACT04]. Aos compromissos espaço-temporais citados antes, pode-se acrescentar como motivação a capacidade de SDRs em particular melhorarem a característica de tolerância a falhas de sistemas computacionais e, em geral, aumentarem a flexibilidade dos mesmos.

1.2 Objetivos

O objetivo principal deste trabalho é propôr parte de uma infra-estrutura de reconfiguração para desenvolvimento de sistemas dinamicamente e parcialmente reconfiguráveis usando interfaces de comunicação padronizadas. Em particular buscou-se produzir um método completo de geração de arquivos de configuração parciais e demonstrar seu efetivo funcionamento em aplicações práticas. Esta parte da infra-estrutura de reconfiguração habilita hoje o grupo local de pesquisa (GAPH - Grupo de Apoio ao Projeto de Hardware) que sediou este trabalho a implementar sistemas parcial e dinamicamente reconfiguráveis.

Como objetivo secundário, é proposta e implementada uma ferramenta para incrementar a automatização do fluxo de projeto para desenvolvimento de sistemas dinamicamente reconfiguráveis.

1.3 Organização do Volume

O restante do presente volume está dividido em 6 capítulos.

O Capítulo 2 apresenta o estado da arte em SDRs, incluindo a introdução de uma proposta de terminologia, uma avaliação sucinta de trabalhos anteriores em SDRs, uma discussão das formas de reconfiguração habilitadas por sistemas atuais e uma revisão de software de suporte proposto para reconfiguração parcial de sistemas.

O Capítulo 3 discute conceitos associados à reconfiguração baseada em núcleos IP, e ao uso de interfaces padronizadas de comunicação intra-chip, concluindo com a apresentação do contexto e da proposta de desenvolvimento contemplada por este trabalho.

O Capítulo 4 detalha a proposta de método de projeto para geração de configurações parciais, baseado no fluxo de projeto modular da Xilinx, bem como a proposta de ferramenta de automatização parcial do método, através da ferramenta MDLauncher.

O Capítulo 5 discute os estudos de casos simples usados na validação do método proposto e apresenta resultados obtidos com estes.

O Capítulo 6 mostra um estudo de caso realista de um processador com conjunto de instruções extensível através do uso de co-processadores dinamicamente reconfiguráveis. Uma comparação inicial de desempenho entre implementações puramente em software e usando

hardware reconfigurável é apresentada, onde o tempo de execução em software é comparado com a soma de tempos de execução e de reconfiguração, visando estabelecer alguns compromissos entre as diferentes implementações.

Finalmente, o Capítulo 7 apresenta algumas conclusões e direções para trabalhos futuros.

Capítulo 2

Estado da Arte em SDRs

Este Capítulo tem como objetivo situar o leitor no que se refere ao estado-da-arte em SDRs, requisitos para habilitar o projeto e a implementação de SDRs, e ferramentas para geração de arquivos de configuração para o desenvolvimento de SDRs.

2.1 Definições Básicas de Reconfiguração

Sistemas reconfiguráveis combinam o desempenho de hardware dedicado a graus de flexibilidade similares a componentes de software. Sistemas programáveis estão limitados à arquitetura do microprocessador utilizado. O uso de hardware reconfigurável permite adaptar arquiteturas às aplicações [VIL97]. Para isto, funções de hardware podem ter suas características modificadas. Para tanto, é necessário realizar *configuração* ou *reconfiguração*, total ou parcial.

Uma *configuração* em um dispositivo ou sistema de hardware configurável é um conjunto de bits que deve ser carregado em posições de uma memória de controle para determinar as funções e a estrutura de hardware que se quer construir [MES02]. O termo configuração também pode ser usado para definir o processo descrito na última frase. A configurabilidade pode ser vantajosa, pois o hardware pode ser alterado sem a necessidade de desenvolver um outro dispositivo que atende estas novas características. *Reconfiguração* é o processo de alterar uma dada configuração de forma total ou parcial, mudando assim as funções desempenhadas pela estrutura do hardware. Dessa forma, a *reconfiguração total* é uma configuração onde a memória de controle do dispositivo reconfigurável é inteiramente sobrescrita. *Reconfiguração parcial* como o próprio nome diz, é o processo de configuração onde a memória de controle do dispositivo é alterada apenas parcialmente.

Segundo Sanchez [SAN99], reconfigurações podem ser dinâmicas ou estáticas. Se o sistema não necessita ter seu processamento interrompido (sem disrupção) para que uma reconfiguração seja realizada então ele é dito *dinâmico*, caso contrário, é dito *estático*.

Pode-se ainda classificar dispositivos reconfiguráveis de acordo com o *tamanho do grão*

configurável. Entende-se por *grão* a menor unidade configurável de um dispositivo. Modernamente, se as configurações se dão no nível de portas lógicas ou funções simples booleanas de poucas variáveis diz-se que o dispositivo é de *grão pequeno*. Se estas se dão sobre unidades funcionais maiores, tais como ULAs, diz-se que o dispositivo possui *grão médio*. Quando estas se dão em unidades de porte considerável, tais como um microprocessador, diz-se que o dispositivo é de *grão grande*.

2.2 Propostas de SDRs

Estrin, em trabalho publicado nos anos 60 [EST63], propôs conceitos hoje considerados como precursores de sistemas de hardware reconfigurável.

O sistema de Estrin chamado Sistema Computador Reestruturável (*Restructurable Computer System*) baseia-se em um repertório de funções armazenadas em hardware. Se o programa a executar necessita uma determinada função, e esta se encontra no repertório, então a função é executada em hardware, aumentando o desempenho desta em relação à mesma implementada em instruções nativas.

O sistema de Estrin é dividido basicamente em três partes:

- **Processador Central** (*Central Processor*): Este é um processador de propósito geral, o qual executa programas convencionais. A implementação realizada usou o computador IBM7090;

- **Unidade Supervisora de Controle** (*Supervisory Control Unit*): esta unidade faz análise de instruções que estão sendo executadas no processador central. Se um determinado conjunto de instruções resulta num algoritmo "conhecido" pela unidade de controle supervisora, então este conjunto é executado pelo Inventário de Estrutura Variável.

- **Inventário de Estrutura Variável** (*Variable Structure Inventory*). Este é um repertório de funções implementadas em hardware que se encontra comumente em algoritmos para resolução de diversos problemas matemáticos que consomem muito tempo de execução. As seqüências de instruções equivalentes executadas no Inventário de Estrutura Variável têm melhor desempenho do que estas instruções executadas no processador central. Quando uma instrução é executada nesta unidade, o processador central pode estar executando o seu fluxo normal em paralelo, obtendo ganhos de desempenho no que diz respeito ao processamento de dados.

Para modificar uma determinada função no Inventário de Estrutura Variável, é necessário habilitar um conjunto de transistores ou mudanças físicas na localização dos módulos (funções implementadas em hardware) e suas interconexões. Para a construção de uma função razoavelmente complexa, era necessário um esforço considerável para implementação da mesma [EST63].

Este foi um trabalho isolado. Somente cerca de vinte anos mais tarde, surgiram os primeiros dispositivos configuráveis comerciais.

SDRs podem ser classificados, de acordo com o número de dispositivos que os compõe, em duas grandes classes: sistemas reconfiguráveis em nível de circuito integrado e sistemas reconfiguráveis em nível de placa. Estes sistemas são discutidos nas próximas seções.

No escopo deste trabalho aborda-se apenas a reconfiguração em nível de circuito integrado usando como base o FPGA Virtex II da Xilinx [XIL01a].

2.2.1 Reconfiguração parcial em nível de circuito integrado

Este tipo de reconfiguração parcial acontece sobre áreas de silício de um CI. A unidade fundamental para a reconfiguração parcial neste nível pode ser constituída por LUTs (alguns FPGAs são constituídos por esta unidade de reconfiguração). FPGAs são por excelência sistemas deste tipo. Entre estes, pode-se citar as famílias Xilinx XC6200 [?], ATMEL AT40K [ATM04] e Xilinx Virtex [XIL00].

Um exemplo de FPGAs que permite a técnica de reconfiguração parcial é o FPGA XC6200 da *Xilinx*. Consiste em uma matriz de 64 x 64 células rodeada por portas de entrada e saída. Todas as células lógicas podem implementar qualquer função lógica combinacional de duas entradas. Cada célula pode implementar um flip-flop do tipo D com o objetivo de implementar a função combinacional da célula. As células têm um esquema de hierarquia de barramento. Células são organizadas dentro de blocos de 4 x 4, 16 x 16, etc. Um conjunto de barramentos rápidos estão associados ao tamanho do bloco. Todos os registradores que estão contidos dentro de algum bloco podem ser acessados por uma interface de usuário. Registradores são endereçados através de colunas via registrador de mapeamento. A reconfiguração é parcial e dinâmica realizada através de controle de armazenamento SRAM de seis transistores estáveis. Esta memória SRAM pode ser mapeada dentro de um espaço de endereçamento de um processador hospedeiro e suporte lógico adicional é provido para permitir reconfiguração para todas as partes do dispositivo. O FPGA XC6200 suporta hardware virtual nas quais os circuitos em execução podem ser armazenados. Isto permite que recursos do FPGA podem ser utilizados por tarefas diferentes, e então os circuitos são restaurados num dado momento, com o mesmo estado interno dos seus registradores. A Figura 2.1 mostra uma estrutura de interconexão entre as células que compõe o FPGA.

Os FPGAs da família AT40K da ATMEL foram projetados para suportarem reconfiguração parcial e dinâmica. Porém esta família de FPGAs da ATMEL suportam um máximo de 50 mil portas lógicas para implementação de projetos reconfiguráveis, ou seja, o tamanho do FPGA é pequeno comparado com o estado da arte em FPGAs de alta densidade, que excedem este valor de pelo menos duas ordens de grandeza. Este FPGA utiliza uma memória de configuração para armazenamento do contexto. Se novas funções são necessárias para a execução da aplicação, as porções antigas são sobrescritas conforme mostrado na Figura 2.2.

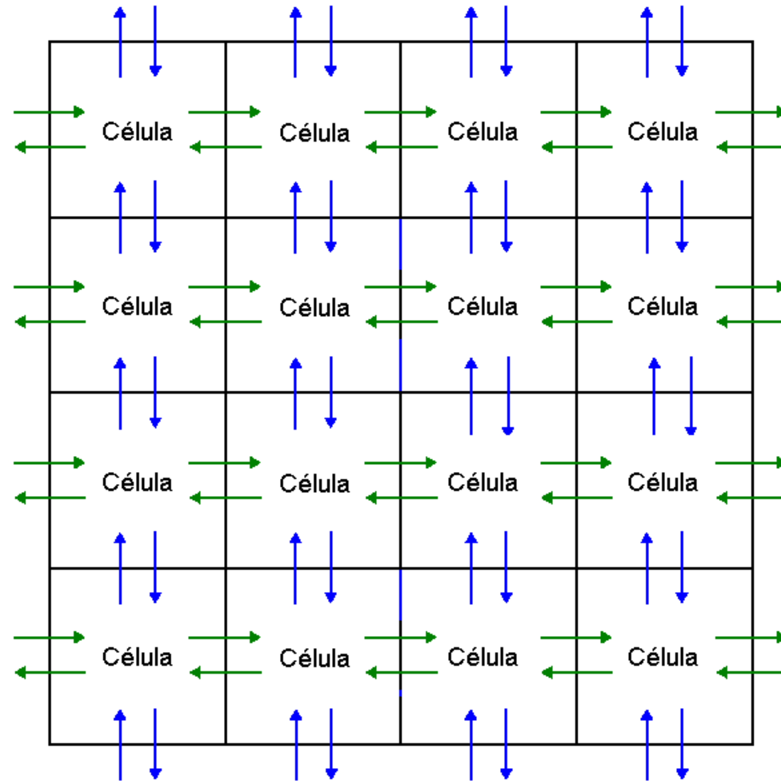

Figura 2.1: Estrutura de interconexão das células no interior do FPGA Xilinx XC6200.

Mais recentemente, a Xilinx desenvolveu os FPGAs da família *Virtex*, *Virtex* II e *Virtex* II *Pro*. A memória de configuração da Virtex pode ser vista como uma matriz bidimensional de bits. Estes bits são agrupados em quadros verticais de 1 bit de largura, e se estendem do topo à base do dispositivo. Um quadro é a unidade mínima de configuração, ou seja, é a menor porção de memória de configuração que pode ser lida ou escrita. Quadros são lidos e escritos seqüencialmente, com endereços crescentes para cada operação. Como os quadros podem ser lidos e escritos individualmente, é possível reconfigurar parcialmente esses dispositivos através da modificação desses quadros no arquivo de configuração. Além disso, a disposição regular de elementos configuráveis permite, em teoria, relocação e desfragmentação de módulos, que têm importância no que se refere no assunto de reconfiguração parcial e dinâmica [COM02]. Os elementos configuráveis são CLBs (*Configurable Logic Block*), BRAMs, roteamento e multiplicadores. A Figura 2.3 mostra um CLB do FPGA *Virtex* XCV300.

Nesta Figura, nota-se que cada CLB contém duas fatias (*slices*), que por sua vez contém duas LUTs cada, além dos recursos de *Carry* e dois *flip-flops*.

Os quadros podem ser vistos em relação a uma coluna através da Figura 2.4. Cada coluna CLB é cortada verticalmente por 48 quadros sucessivos. Como a disposição das CLBs é em colunas, a modificação de uma CLB implica na alteração de todas as CLBs da coluna a

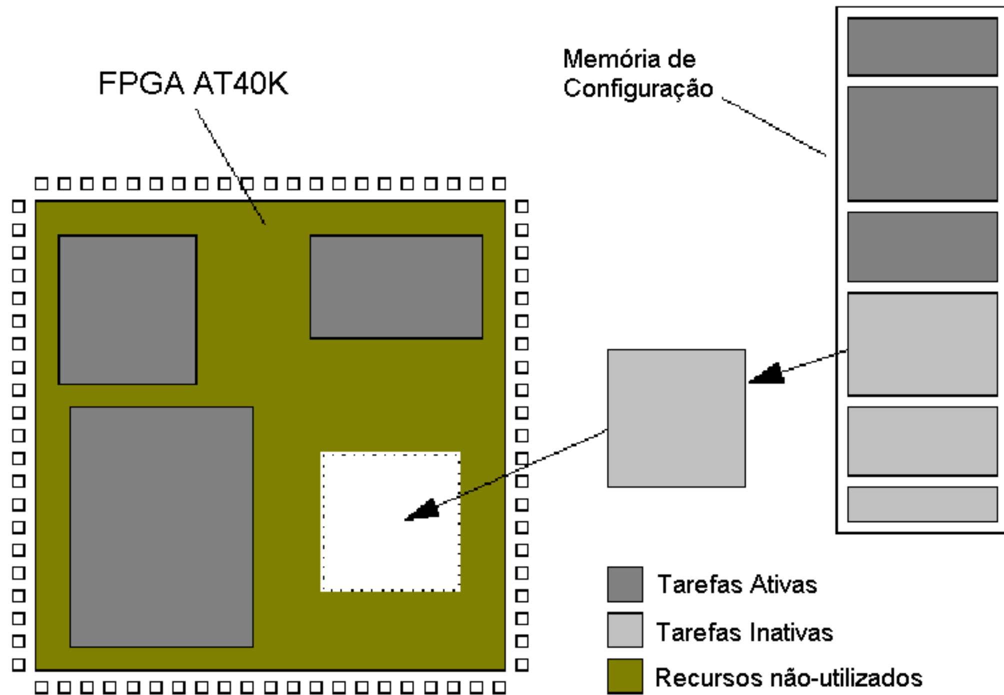

Figura 2.2: FPGA AT40K da ATMEL, onde núcleos IP armazenados em memória configuram o FPGA em tempos diferentes.

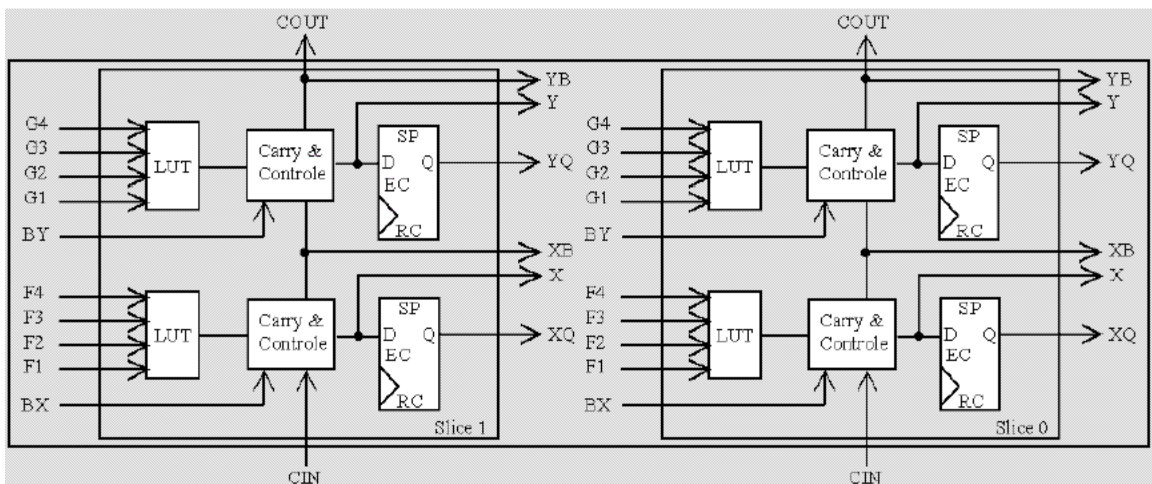

Figura 2.3: Esquema de um CLB do FPGA XCV300.

que pertence. Nota-se ainda que as colunas são numeradas a partir do 0 (zero), atribuído à coluna central. As demais colunas são numeradas em ordem crescente, com valores pares à esquerda da coluna central, e ímpares a sua direita. A numeração é importante para o

endereçamento dos elementos. Para maiores informações sobre detalhes da arquitetura da Virtex, veja [XIL00].

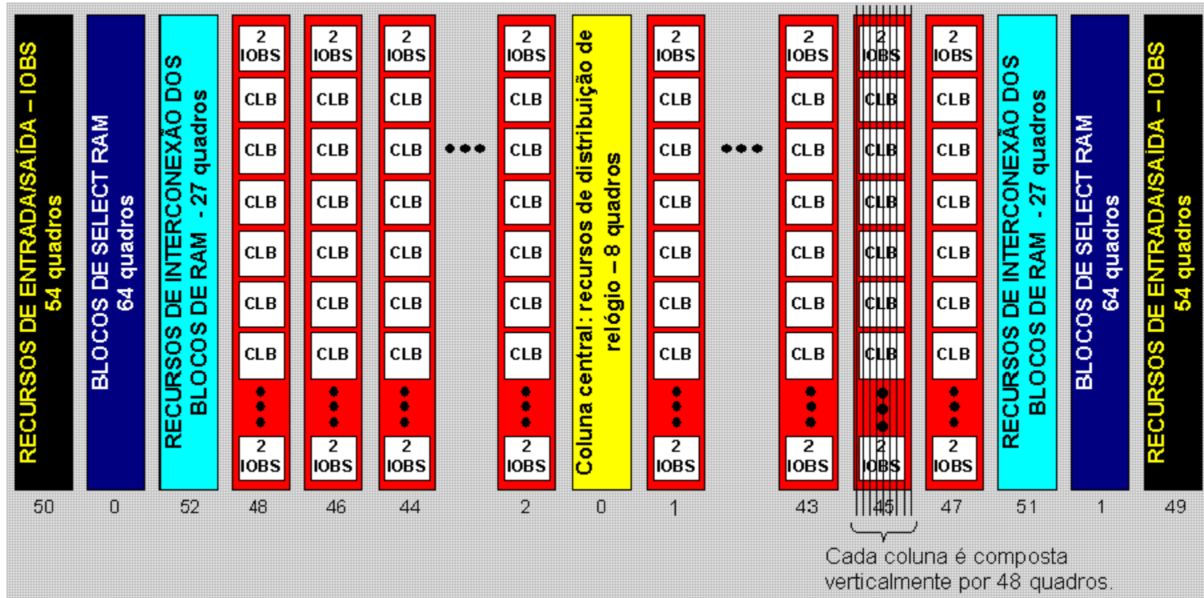

Figura 2.4: Disposição em colunas dos elementos do FPGA Virtex XCV300.

Uma das famílias mais recentes dos FPGAs da *Xilinx* é a *Virtex* II. Os FPGAs desta família permitem desenvolver circuitos de 500×10^3 a 10000×10^3 portas lógicas.

A arquitetura geral de um dispositivo da família *Virtex* II [XIL02] está apresentado na Figura 2.5.

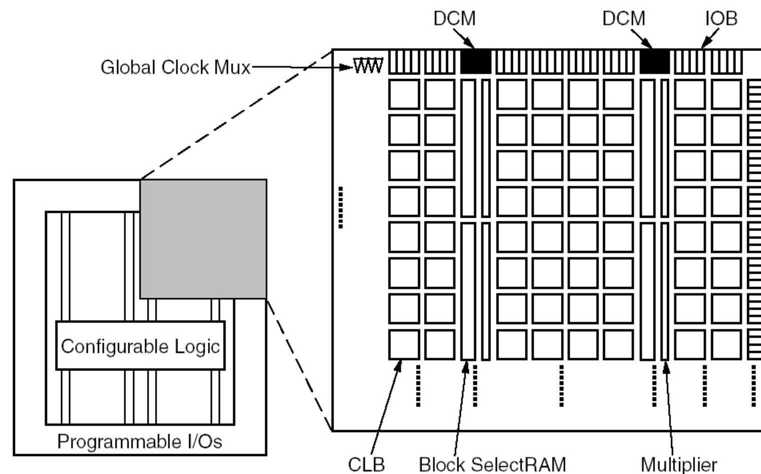

Figura 2.5: Arquitetura de um dispositivo da Família VirtexII.

Na Figura 2.5 pode-se ver que os IOBs programáveis fornecem uma interface entre os pinos

externos e a lógica interna configurável. Essa lógica interna inclui quatro elementos principais, os quais são organizados em uma matriz regular:

- Os CLBs fornecem elementos funcionais para lógicas combinacionais e síncronas, incluindo elementos básicos de armazenamento. *Buffers* tri-states (TBUFs) são associados a cada CLB e dedicados diretamente a recursos de roteamento horizontal que podem ser segmentados.

- Módulos de memória, chamados *Block SelectRAM*, fornecem elementos de armazenamento de 18 Kbits com verdadeiras RAMs de porta dupla.

- Blocos Multiplicadores, compostos por multiplicadores de 18-bits x 18-bits.

- Gerenciador de Clock Digital (DCM), módulos responsáveis pelo sincronismo de relógio, com possibilidade de multiplicação e divisão de freqüência do relógio, e mudança de fase.

Todos os elementos programáveis, incluindo os recursos de roteamento, são controlados por valores armazenados em células de memória estática. Esses valores são carregados em células de memórias durante a configuração e podem ser recarregados a qualquer momento para modificar a funcionalidade do elemento programável.

Como já foi citado, os IOBs são programáveis. Eles podem ser vistos como um bloco de entrada com um registrador opcional, um bloco de saída com um registrador, um buffer tri-state, ou um bloco bidirecional com qualquer combinação de configurações de entradas e saídas. Os IOBs são fornecidos no perímetro de cada dispositivo em grupos de dois ou de quatro.

Os blocos lógicos configuráveis (CLBs) da família *Virtex* II são compostos por quatro *slices* (ou fatias) e dois buffers tri-states. Cada um dos slices são equivalentes e são compostos por dois geradores de funções (denominados de F e G) de quatro entradas, dois elementos de armazenamento e multiplexadores. Tais geradores podem ser configurados como *Look-Up-Tables* (LUTs) de quatro entradas capazes de implementar qualquer função combinacional de quatro entradas, registradores de deslocamento de 16 bits ou memória SelectRAM de 16 bits.

Os multiplexadores que fazem parte de um slice e os geradores de funções na *Virtex* II podem ser implementados como: multiplexador 4:1 em um *slice*, multiplexador 8:1 em dois slices, multiplexador 16:1 em um CLB (4 slices) e multiplexador 32:1 em CLBs.

Cada CLB contém dois dispositivos tri-state, cada um com seu próprio pino de controle e seu próprio pino de entrada. Cada um dos quatro slices tem acesso a dois tri-states através de conexões diretas. A saída do tri-state é dirigida para recursos de roteamento horizontais usados para implementar barramentos tri-state.

Cada dispositivo *Virtex* II incorpora uma quantidade de blocos de memória SelectRAM de 18 Kbits, as quais são programáveis desde 16Kx1 bit até 512x36 bits, em várias larguras

e profundidades. As SelectRAMs são de porta dupla, possuindo duas portas com clocks e controles síncronos independentes, as quais acessam uma área de armazenamento comum.

Os FPGAS da família Virtex II-Pro [XIL03] constituem uma extensão da família Virtex II. A novidade desta família é conter processadores IBM PowerPC 405 embarcados. Segundo os fabricantes, pela primeira vez os projetistas podem particionar e reparticionar seus sistemas entre hardware e software durante o ciclo de desenvolvimento de uma maneira mais flexível, ou seja, não apenas no início do projeto. O maior dispositivo desta família possui quatro processadores PowerPC embarcados.

2.2.2 Reconfiguração parcial em nível de placa

Este tipo de reconfiguração é realizada em sistemas reconfiguráveis implementados sobre uma ou mais placas de circuito impresso contendo vários componentes como processadores, memórias, UARTs, barramentos e até mesmo vários FPGAs. Tais sistemas podem ser caracterizados como grão fino, médio e grosso, inclusive pode haver casos em que estes sistemas reconfiguráveis utilizam FPGAs como unidades fundamentais de reconfiguração.

A seguir, serão mostrados alguns exemplos de arquiteturas reconfiguráveis que têm como característica a reconfiguração parcial em nível de placa.

Splash 2

O Splash 2 é uma arquitetura reconfigurável baseada em FPGAs. Essa é constituída por 17 FPGAs 4010 da Xilinx, cada um acoplado a blocos de 512Kb de memória e um hospedeiro SparcStation. Dezesseis desses FPGAs são conectados em um vetor, e também a uma rede crossbar que introduz uma flexibilidade maior que se fosse tratado como uma matriz linear [ARN92, ARN93]. O décimo sétimo FPGA é conectado a rede *crossbar* e se comporta como um hardware de distribuição dos dados que recebe do barramento SIMD (*Single Instruction Multiple Data*). Existe também uma interface de conexão ao hospedeiro (*Interface Board*) que contém um relógio de sistema programável e provê acesso ao DMA para a memória do hospedeiro através de dois FPGAs (XL e XR), os quais são usados para enviar e receber dados do hospedeiro ao Splash 2. Um sistema de interconexão é usado para prover o acesso aos Splash 2 a interface de conexão ao hospedeiro. A placa Splash2 é ligada ao resto do sistema por cinco barramentos. Além do barramento SIMD, usado para transferir dados do computador hospedeiro à placa, existem dois barramentos de extensão, um barramento da dados de saída e um barramento de configuração. A arquitetura do SPLASH 2, bem como sua organização são mostradas na Figura 2.6.

O sistema Splash2 apresenta ferramentas de software baseadas em VHDL. Um simulador de Splash2 compreende um conjunto de descrições VHDL para cada um dos componentes do sistema, e pode simular o comportamento do sistema como um todo. Uma outra parte da plataforma de software é um compilador que compila código VHDL em uma netlist Xilinx, e a

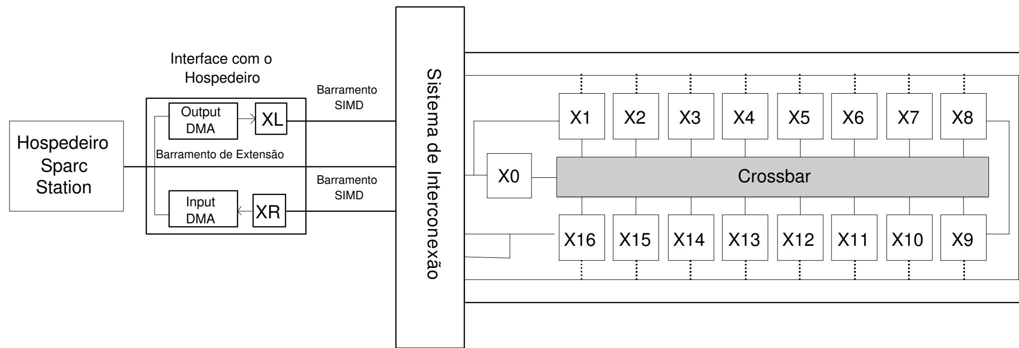

Figura 2.6: Arquitetura do SPLASH 2.

otimiza para o sistema Splash2. Softwares da Xilinx fazem o roteamento e a posicionamento. Splash2 foi desenvolvido para incrementar certos aspectos do sistema Splash: escalabilidade, largura de banda para E/S e programabilidade. Tem conexões mais flexíveis que seu antecessor, apresenta rede crossbar. Assim como seu antecessor, Splash2 sofre pela baixa velocidade de integração, mas tem a vantagem da reconfiguração parcial. Esta propriedade é devida a possibilidade de reconfiguração da rede de interconexão.

CHESS

CHESS [MAR99] é uma arquitetura reconfigurável para aplicações multimídia, desenvolvidas no *Hewlett Packard Laboratories* em 1999. A arquitetura do CHESS é constituída basicamente por um conjunto de ULAs (Unidade Lógica Aritmética) de 4 bits dispostas matricialmente e interconectadas (Figura 2.7). Cada ULA utiliza como entrada dados provenientes da saída de uma outra ULA. Bancos de memória externa podem ser conectados à arquitetura. Um arranjo de 512 ULAs pode ser configurado num tempo de 40 ns. CHESS oferece flexibilidade no roteamento, escalabilidade e reconfiguração parcial/dinâmica.

Amalgam

A arquitetura *Amalgam* [WAL02] é um exemplo de arquitetura que utiliza FPGA como unidade mínima reconfigurável. Esta arquitetura foi desenvolvida na Universidade de Illinois em 2002. Componentes como unidades de processamento programáveis e reconfiguráveis chamados *clusters* são integrados à arquitetura conectados a um barramento com um sistema de memória compartilhada. A arquitetura *Amalgam* é constituída de quatro *clusters* programáveis (*PClust*) e quatro *clusters* reconfiguráveis (*RClust*) como mostrado na Figura 2.8.

O *RClust* é constituído por um arranjo de blocos lógicos (32x32) de 32 bits, particionados em quatro segmentos e organizados em anel (Figura 2.9). Neste *cluster*, existe um controlador programável chamado de ACU (*Array Control Unit*), responsável pelo gerenciamento da

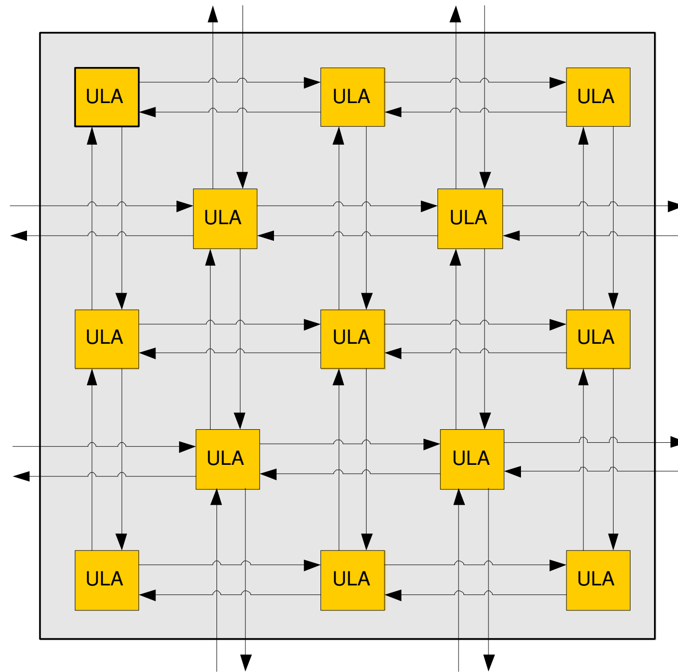

Figura 2.7: Arquitetura CHESS constituída por um arranjo matricial de ULAs conectadas entre si.

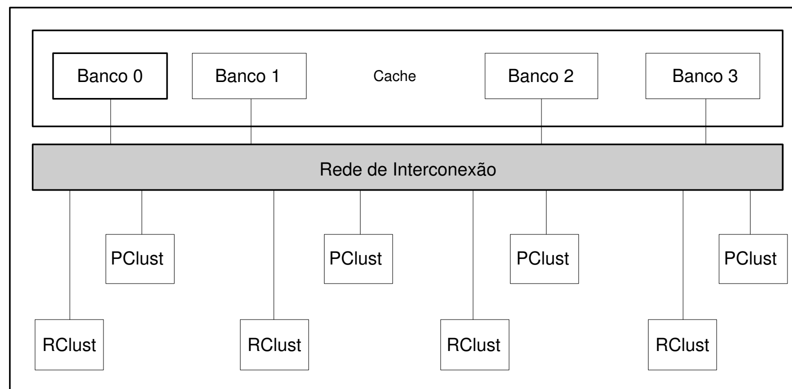

Figura 2.8: Arquitetura do sistema reconfigurável *Amalgam*.

computação e controle do fluxo de dados com a interface de rede (NIC - *Network Interface Card*). *RClust* é conectado a um banco de registradores, cada qual com portas de leitura e escrita, integrando os blocos lógicos, transparecendo ser um único sistema. De acordo com Walstrom et. al. [WAL02], cada *RClust* atua como um coprocessador para cada *PClust*.

O *PClust* contém um banco de dados de registradores, *cache* de instruções (*I-Cache*) e duas ULA's que executam um conjunto de instruções baseado no MIPS ISA.

A arquitetura Amalgam é uma nova proposta de sistemas embarcados que integra unidades programáveis e reconfiguráveis. *RClust* tem suporte a reconfiguração dinâmica, Porém apenas

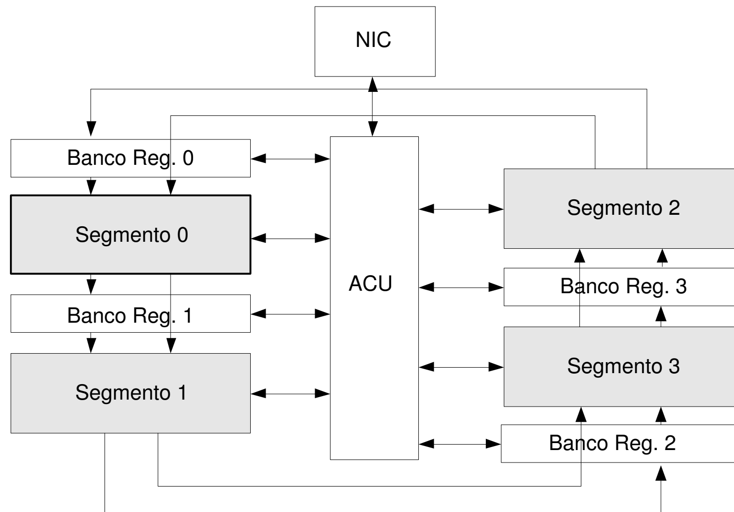

Figura 2.9: Arquitetura do *Cluster* Reconfigurável.

um único contexto pode ser armazenado em cada ciclo. As configurações uma vez utilizadas são armazenadas em memória *cache*.

2.3 Formas de Reconfiguração

Atualmente, existe uma carência no que diz respeito ao ferramental para desenvolvimento de sistemas dinamicamente reconfiguráveis [ZHA00]. Segundo McMillan [MCM99], existem duas iniciativas que se destacam na tentativa de solucionar este problema. A primeira delas é provida por pesquisas acadêmicas, que propôe um conjunto de ferramentas de CAD baseada em JAVA que habilita o projeto e a implementação de sistemas dinamicamente reconfiguráveis, e ferramentas para modificação de *bitstreams* para implementar módulos de hardware reconfiguráveis [HOR02, HOR01].

A segunda iniciativa é provida pelo setor industrial. A equipe de desenvolvimento da empresa Xilinx (fabricante de FPGAs) disponibiliza técnicas baseadas em fluxos de execução de ferramentas de síntese e implementação. Além disto, a equipe oferece classes em Java que provêem APIs (*Application Programming Interface*) para manipulação de arquivos de configuração (*bitstream*) do FPGA, de modo a torná-los parcialmente reconfiguráveis.

2.3.1 Reconfiguração baseada em alterações incrementais

Este método de reconfiguração baseado na técnica *Small Bit Manipulation*, descrita através da XAPP290 [LIM03], consiste em modificar apenas alguns quadros que constituem o FPGA. A idéia central é gerar bitstreams parciais usando a diferença de um bitstream total e modificações feitas num arquivo que posteriormente é lido pela ferramenta geradora de

bitstreams denominada *BitGen*. A ferramenta de edição do FPGA (FPGA *Editor*) permite a modificação das funções contidas em uma ou mais LUTs. Dados de BRAMs (*Block RAM*) e pinos de I/O também podem ser manipulados em nível de bits e modificados dinamicamente. Isto é útil para aplicações reconfiguráveis que necessitam de pequenas alterações pontuais. Porém esta técnica é extremamente penosa para desenvolvimento de arquivos de configuração que armazenam módulos de hardware. Para a geração de bitstreams parciais que representam módulos inteiros de um dado projeto, a Xilinx recomenda o uso do fluxo do Projeto Modular [XIL01b] apresentado no Capítulo 4.

2.3.2 Reconfiguração baseada na inserção/remoção de núcleos

Este método de reconfiguração consiste em reconfigurar parcialmente núcleos IP que podem ocupar os mais variados tamanhos no FPGA. A Figura 2.10 mostra alguns núcleos IP conectados a uma interface de comunicação. O objetivo é possibilitar que os núcleos IP possam ser inseridos e removidos (*plug-and-play*) dinamicamente sem afetar o resto do sistema. A entrada para a execução deste método de reconfiguração é um sistema descrito em HDL, e a execução desse método gera arquivos de configurações totais e parciais, habilitando então a reconfiguração parcial e dinâmica nesse sistema.

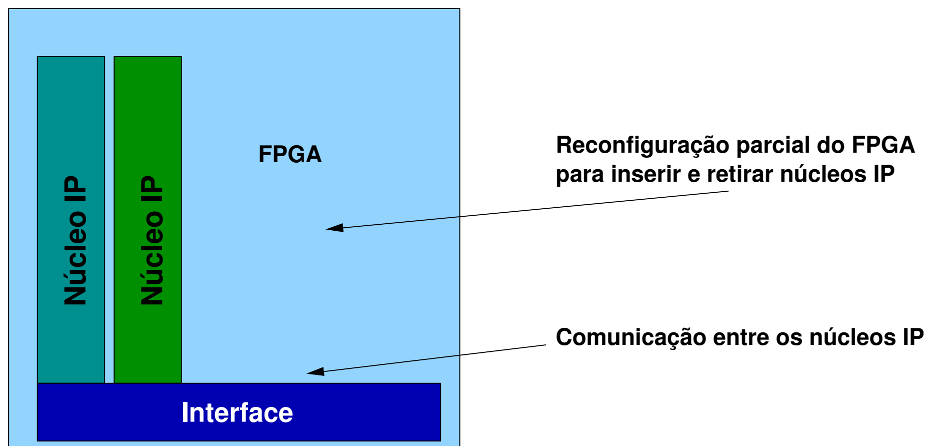

Figura 2.10: Núcleos IP reconfiguráveis conectados a uma interface de comunicação.

Porém existe uma carência de ferramentas para a execução de tal método. Na bibliografia, existem poucos fluxos e ferramentas para desenvolvimento de projetos dinamicamente reconfiguráveis. Esses fluxos para habilitação da reconfiguração baseada em núcleos IP são descritos mais detalhadamente na Seção 3.1. Como inconvenientes. o uso de recursos adicionais do FPGA para prover comunicação entre os núcleos IP e problemas quanto a erros eventuais de roteamento do sistema, cujo usuário deve fazer o roteamento manualmente.

2.4 Requisitos para Habilitar SDRs

O presente trabalho enfoca a reconfiguração parcial em nível de CI apenas (Seção 2.2.1). Além disto, pressupõe-se que o processo de reconfiguração é baseado em inserção/remoção de IPs (Seção 2.3.2).

Para a realização de reconfiguração parcial em dispositivos reconfiguráveis, é possível identificar seis requisitos:

- *FPGAs que suportam reconfiguração parcial e dinâmica*: atualmente dois fabricantes de FPGAs oferecem suporte para reconfiguração parcial e dinâmica: Atmel e Xilinx. Atmel produz os FPGAs das famílias AT40K e AT6000. Estes FPGAs são inadequados para o desenvolvimento de SoCs, por causa do seu tamanho reduzido em relação ao número de unidades fundamentais de reconfiguração (LUTs). Por outro lado, os FPGAs da Xilinx disponibilizam as famílias Virtex, Virtex-II e Virtex-II Pro. A unidade atômica de reconfiguração se chama *quadro*. Este compreende em uma seção vertical que se extende por todo o FPGA. Um núcleo IP é fisicamente implementado nestes dispositivos da Xilinx como um conjunto de quadros consecutivos;

- *Ferramentas para determinar a forma e a posição do núcleo IP*: restrições de posicionamento devem ser inseridas durante o projeto de SDRs para a definição do tamanho, formato e posição dos núcleos IP reconfiguráveis. ex: Floorplanner da Xilinx;

- *Suporte para geração de arquivos de reconfiguração parciais e configuração destes no FPGA*: protocolos de reconfiguração parcial e total são distintos. Características especiais devem estar presentes nas ferramentas e dispositivos usados para reconfigurar parcialmente os FPGAs;

- *Interface de Comunicação entre Núcleos IPs*: em um projeto de SoC convencional, arquiteturas de barramento intra-chip padronizadas, tais como AMBA e *CoreConnect* podem ser usadas para prover comunicação entre os núcleos IPs. Entretanto, estas arquiteturas não apresentarem características necessárias em SDRs, como isolamento de núcleos IPs durante a reconfiguração. Para habilitar a inserção e remoção dinâmica de núcleos IPs dentro do FPGA em operação, uma interface de comunicação para conexão e isolamento de núcleos em momentos distintos deve ser usada. Esta interface deve prover funções diversas tais como arbitragem, comunicação entre módulos, virtualização dos pinos de entrada e saída e controle de configuração;

- *Virtualização dos pinos de entrada e saída*: este conceito permite troca de dados entre o núcleo IP reconfigurável e o meio externo sem alteração das características de pinagem do CI;

- *Controle de reconfiguração dinâmica*: Métodos para controle ou execução do processo de reconfiguração dinâmica são um requisito imprescindível. Estes métodos podem ser implementados em hardware, software ou ambos [CAR04].

2.5 Suporte de Software para Reconfiguração Parcial

A geração de arquivos de configuração parciais apresenta-se como uma tarefa crucial para o desenvolvimento de SDRs. Os fabricantes fornecem junto com seus dispositivos ambientes de CAD para desenvolvimento de sistemas que contém, muitas vezes, diversas ferramentas. Dentre estas, algumas se dedicam à geração de arquivos de configuração parciais.

O fluxo de geração de arquivos parciais com ferramentas fornecidas pelos fabricantes deve ser realizado, ainda hoje de forma manual. Por isso, trata-se de um processo demorado e complexo, passível de erros, para projetos que necessitam de diversas configurações parciais. Necessita-se, portanto, de ferramentas mais automatizadas para o projeto de sistemas reconfiguráveis.

Nesta Seção, apresentam-se algumas ferramentas que tem por objetivo auxiliar projetistas na tarefa de gerar arquivos de configuração parciais.

2.5.1 Conjunto de classes JBits

JBits é um conjunto de classes Java que fornecem uma API que permite manipular o arquivo de configuração das famílias de FPGAs Virtex e Virtex-E da Xilinx [GUC99]. Esta interface opera tanto em arquivos de configuração gerados pelas ferramentas de projeto da Xilinx quanto em arquivos de configuração lidos do hardware (*Readback*). Para que este conjunto de classes possa dar suporte à reconfiguração parcial e dinâmica, 3a ferramenta deve ser rápida e prover informações físicas detalhadas a respeito da arquitetura do hardware, bem como elementos arquiteturais reconfiguráveis do dispositivo a ser configurado. Isto permite que os circuito seja reconfigurado sem a necessidade de se refazer o posicionamento e o roteamento com ferramentas convencionais oferecidas pelos fabricantes de FPGAs.

A Figura 2.11 ilustra o fluxo de projeto do JBits. Neste fluxo, um compilador de Java lê o programa descrito pelo usuário com as classes para acesso ao FPGA. o compilador também lê dados da biblioteca do conjunto de classes JBits. O compilador gera um código executável que fornece o controle de configuração e dados para a lógica de reconfiguração do dispositivo. Neste fluxo de projeto, o arquivo executável gerado pelo compilador Java é somente um componente arbitrário do código Java compilado. Este executável permite desenvolver aplicações em software para acessar a aplicação reconfigurável no dispositivo. A informação de controle ou de dados pode ser compartilhada entre o computador hospedeiro e o FPGA.

A Figura 2.12-a apresenta uma aplicação escrita em JAVA desenvolvida pelo usuário. O diagrama da Figura 2.12-b é dividido em três níveis de abstração. O nível de abstração

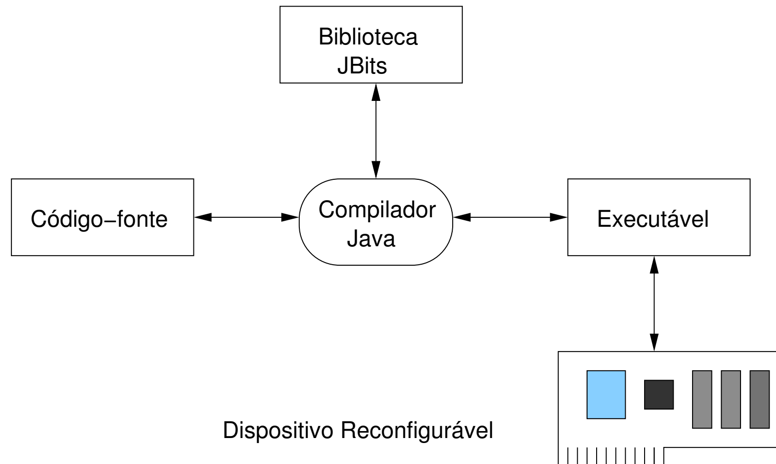

Figura 2.11: Fluxo de projeto do JBits para prover controle e dados de configuração.

mais baixo, apresentado na Figura 2.12-b.3, representa uma classe denominada *Bitstream* que gerencia o arquivo de configuração do dispositivo e disponibiliza suporte para leitura e escrita das configurações dos arquivos. Em seguida, o nível de abstração intermediário, apresentado na Figura 2.12-b.2, representa a classe *Interface em Nível de Bit*. Este nível é responsável por ocultar características específicas relativas a dispositivos diferentes de uma mesma família. Também, neste nível, é possível modificar até um único bit no arquivo de configuração. O nível de abstração mais alta (JBits) provê uma abstração que permite a modificação de conjunto de bits no arquivo de configuração através de uma ou mais chamadas de métodos da classe *Interface em Nível de Bit*. A API JBits utiliza o software XHWIF (*Xilinx HardWare InterFace*) para configurar o dispositivo e realizar leituras através deste. (Figura 2.12-d).

Finalmente, o módulo mostrado na Figura 2.12-e, utiliza uma biblioteca de núcleos IP (conjunto de classes em JAVA) que podem ser adicionados ou removidos do dispositivo.

As limitações desta API devem-se ao fato que todos os recursos e manipulação destes devem ser explicitados manualmente no código-fonte. Devido à necessidade desta especificação detalhada de todos os recursos do FPGA no código-fonte do software, o conjunto de classes *JBits* pode favorecer circuitos mais estruturados. Circuitos não-estruturados (lógica aleatória, por exemplo) não são bem adaptados para uso com o *Jbits*. Outra limitação é que o usuário deve estar bem familiarizado com a arquitetura-alvo, pois sem o conhecimento devido dos detalhes arquiteturais do dispositivo, pode danificar o FPGA [GUC99].

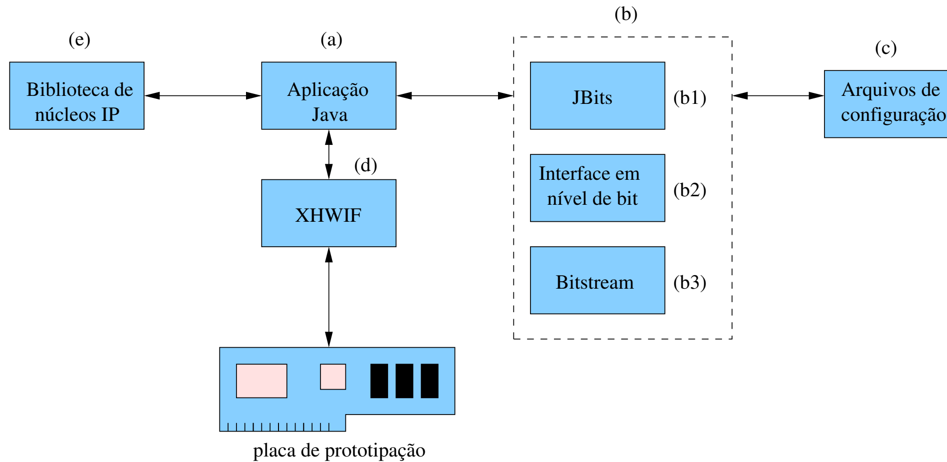

Figura 2.12: Aplicação escrita em JAVA desenvolvida pelo usuário. Percebe-se que há diferentes níveis de abstração para a geração de arquivos de configuração parciais, ocultando detalhes do usuário.

2.5.2 Ferramenta JRTR (*Java Run-Time Reconfiguration*)

JRTR é uma extensão da API JBits. Esta interface provê um modelo de cache onde as modificações dos dados referentes à configuração são ajustadas, e somente os dados realmente necessários são escritos no FPGA ou lidos dele [MCM99]. Esta extensão resultou no suporte direto à reconfiguração parcial. Este suporte utiliza uma combinação de técnicas de hardware e programas que permitem pequenas modificações no arquivo de configuração das Virtex de maneira direta, rápida e sem interrupção de operação. A interface JBits é ainda utilizada para leitura e escrita de arquivos de configuração. O JRTR ainda tem um *parser* para análise do arquivo de configuração e além disso, mantém a imagem dos dados e informações de acesso. A API atual provê controle simples, porém completo da cache de configurações. O usuário pode gerar configurações parciais em qualquer instante, e então carregá-las no hardware. O suporte de dispositivos desta técnica é dependente com os dispositivos que o conjunto das classes *JBits* suporta. A Figura 2.13 ilustra o fluxo de projeto do JRTR. A aplicação em Java utiliza a classe JBits para ler e escrever arquivos de configuração parcial. O *parser*, como foi mencionado anteriormente, é usado para analisar o arquivo de configurações (comandos usados para geração de arquivos de configuração parcial e total). A aplicação em Java ainda utiliza o software XHWIF para realizar configurações e leituras no dispositivo reconfigurável.

2.5.3 Ferramenta PARBit

Esta ferramenta foi desenvolvida em uma parceria entre a Washington University e a empresa Xilinx. Segundo Horta et. al. [HOR02, HOR01], PARBit foi desenvolvido para implementar módulos de hardware configuráveis, utilizando arquivos de configuração totais como entrada para a implementação dos mesmos. A ferramenta utiliza um arquivo de configuração

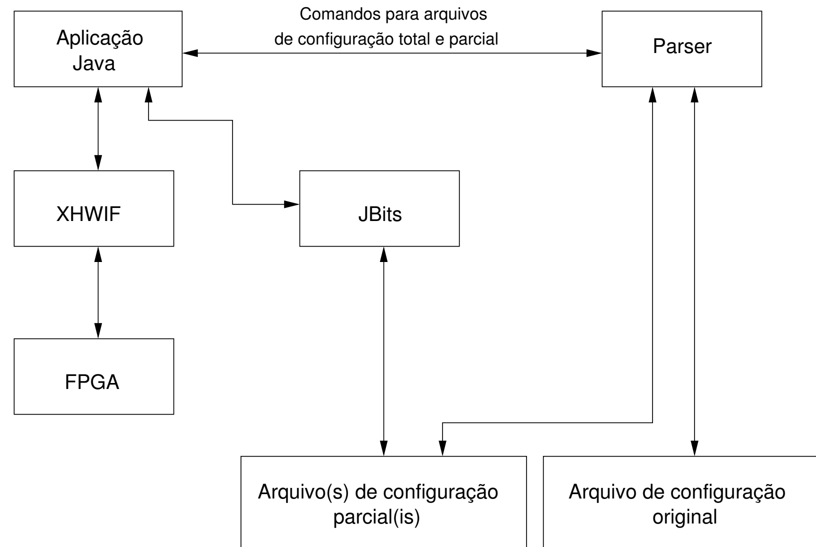

Figura 2.13: Fluxo de projeto do JRTR.

original, um arquivo de configuração a ser gravado, e parâmetros que devem ser inseridos para restringir a área do arquivo de configuração original que será copiado. Os parâmetros são inseridos sob forma de coordenadas cartesianas no plano bidimensional do FPGA. A ferramenta, através destes parâmetros e com o arquivo de configuração total, gera então um arquivo de configuração parcial já roteado e posicionado. A Figura 2.14 mostra um diagrama arquitetural de um FPGA, com um bloco marcado através das coordenadas *Coluna_Inicial*, *Linha_Inicial*, *Coluna_Final* e *Linha_Final*. A área hachurada representa o módulo que será armazenado no arquivo de configuração parcial.

Há dois modos de operação definidos pelos parâmetros do usuário. Estes são descritos a seguir:

- **Slice**: neste modo, o usuário especifica um slice contendo uma ou mais colunas de CLBs (quadros). A ferramenta gera o arquivo de configuração parcial com estes quadros na mesma posição que estes estavam situados do arquivo de configuração original;

- **Bloco**: neste modo, é possível definir uma área (bloco) abrangendo os quadros que constituem uma ou mais CLBs. O usuário define em que posição do arquivo de configuração alvo será gravado este bloco. A ferramenta gera arquivo de configuração parcial, a partir do arquivo de configuração original, contendo a área selecionada pelo usuário (bloco). Este arquivo de configuração original é então usado para reconfigurar o dispositivo alvo;

A Figura 2.15 apresenta o fluxo de projeto e execução da ferramenta PARBIT. A entrada desta ferramenta são os parâmetros que corresponde o arquivo de configuração total, as

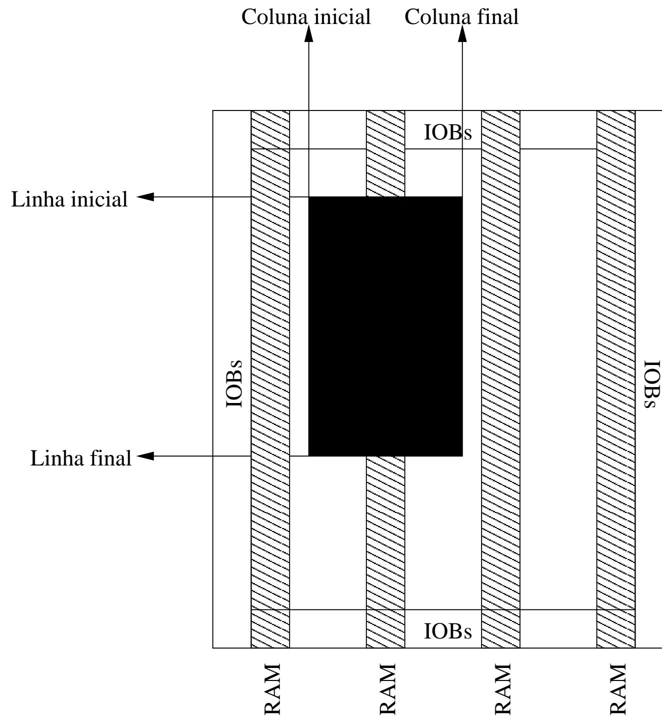

Figura 2.14: FPGA com uma área hachurada, a qual contem um módulo que será gerado no arquivo reconfigurável pela ferramenta PARBIT.

coordenadas e o nome do arquivo de configuração parcial alvo. Se os parâmetros estiverem certos, então o cabeçalho do arquivo de configuração é escrito no arquivo de configuração alvo. Logo a seguir, colunas de CLBs denominadas *slices* do arquivo de configuração original são copiadas para o arquivo de configuração parcial alvo. Se os parâmetros definem o modo de operação como o modo *bloco* então é feita a cópia do bloco no arquivo de configuração parcial alvo. Depois disto, então é gerado o fim do arquivo onde algumas operações, entre elas, o cálculo do CRC (*Cyclic Redundancy Check*), são realizadas.

A mesma equipe de pesquisadores que desenvolveram o PARBIT propôs o conceito de *Dynamic Hardware Plugins* (DHP) [HOR02, HOR01]. DHPs permitem que múltiplas aplicações de hardware, ou *plugins*, sejam dinamicamente carregadas em um único dispositivo e executem em paralelo. Em suma, esta técnica pode ser utilzada para o desenvolvimento e implementação de SDRs.

Um problema enfrentado na geração de arquivos de configuração parciais pelo PARBIT é controlar o processo de roteamento. Por isso, é importante notar, em [HOR02], a existência de uma colaboração direta com a empresa Xilinx, que adotou uma ferramenta de roteamento para que esta aceite a imposição de restrições que viabilizem a reconfiguração parcial de maneira automatizada.

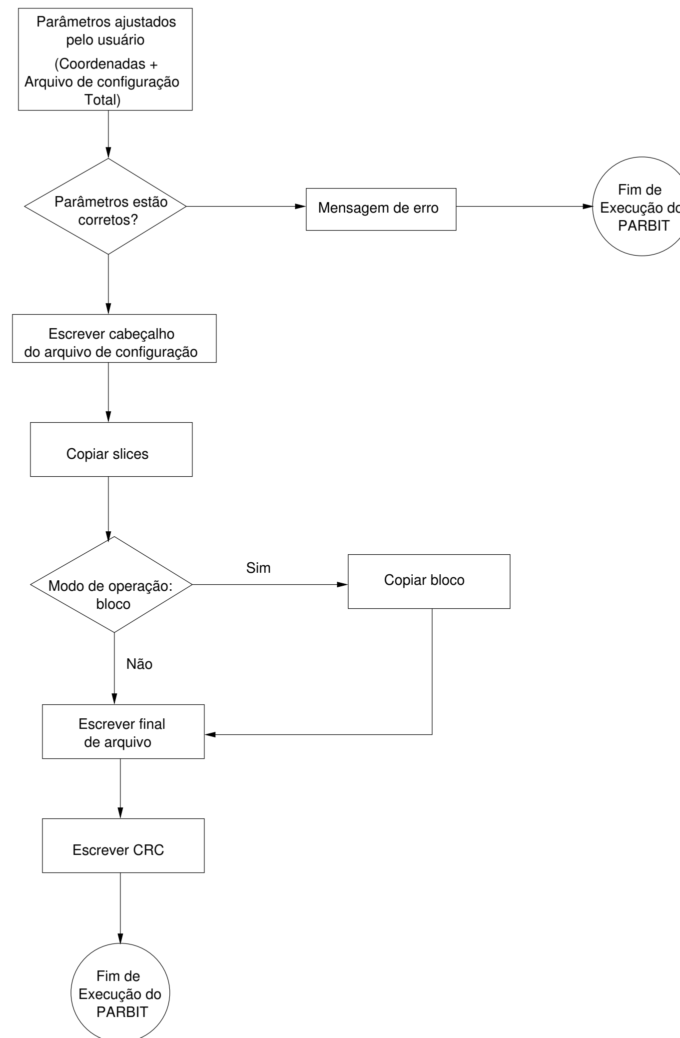

Figura 2.15: Fluxo de projeto e execução da ferramenta PARBIT.

2.5.4 Gerador de arquivo de configuração - JPG

Segundo Raghavan [RAG02], JPG é uma ferramenta utilizada dentro do fluxo de projeto da Xilinx necessária para gerar arquivos de configuração parciais. JPG foi desenvolvida em Java, baseada na API JBits [GUC99]. Para a geração dos mesmos, deve-se ter pelo menos um projeto base (arquivo de configuração total).

A Figura 2.16 ilustra o fluxo de projeto do JPG.

A primeira fase do processo de projeto é a partição do projeto base em módulos menores. Uma planta-baixa inicial é criada para restringir o posicionamento de módulos individuais em regiões específicas do FPGA, onde módulos serão reconfigurados nestas posições. Os módulos são sintetizados, mapeados, posicionados e roteados usando ferramentas disponibilizadas

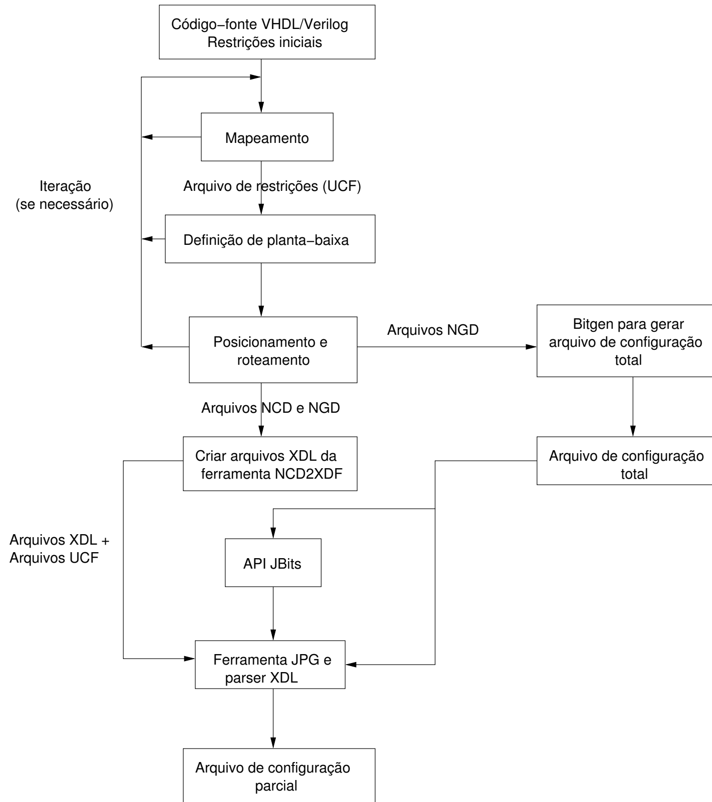

Figura 2.16: Fluxo de projeto do gerador de arquivos de configuração JPG.

pela Xilinx, gerando assim, um arquivo de configuração completo para ser o projeto base do processo.

A segunda fase do projeto objetiva criar versões diferentes dos módulos que serão usados para a reconfiguração parcial. São geradas plantas-baixas para cada um dos módulos parciais para fixação destes no instante da reconfiguração do dispositivo. Ferramentas são necessárias para o mapeamento, posicionamento e roteamento dos módulos utilizando arquivos de restrições. Neste passo, podem vir a ser necessárias várias iterações para se obter os resultados desejados. A seguir, arquivos mapeados com formato interno da Xilinx são traduzidos para um arquivo ASCII o qual contém informações de recursos que implementam o projeto. Então a ferramenta JPG é usada neste ponto para a geração de arquivos de configuração parciais. Esta técnica de geração de arquivos de configuração parciais estabelece uma ligação direta entre o sistema de desenvolvimento da Xilinx e JBits, facilitando a tarefa de geração de arquivos parciais, dependendo, portanto, da API fornecida pela Xilinx.

De acordo com Raghavan [RAG02] estabelece uma ligação entre o fluxo de projeto da Xilinx e a ferramenta JBits [GUC99]. Esta afirmação permite que os projetistas usem o fluxo convencional de projeto da Xilinx obtendo vantagem em algumas características de baixo

nível oferecidas pelo JBits. Não é necessário a programação em Java para o uso de JBits e o conhecimento de detalhes arquiteturais do dispositivo alvo.

A principal desvantagem desta ferramenta é que o sistema de CAD (*Computer Aided Design*) da Xilinx não automatiza totalmente o fluxo, resultando num maior esforço de desenvolvimento de projetos reconfiguráveis para o projetista. Outra desvantagem é que o suporte de dispositivos desta técnica depende dos dispositivos que o conjunto das classes *JBits* suporta.

2.5.5 Ferramentas alternativas desenvolvidas por Möller

As ferramentas aqui documentadas têm por objetivo trabalhar com o arquivo final do projeto de hardware (*bitstream*). Segundo Möller [MÖLL03], existem dois formatos para o arquivo de configuração do dispositivo reconfigurável: ASCII e binário. O formato ASCII (também denominado *Rawbits*) é voltado para o melhor entendimento sobre as especificidades arquiteturais e informações de configuração. Por outro lado, o formato binário tem a vantagem de ser cerca de oito vezes menor que o arquivo de configuração em formado ASCII.

Algumas ferramentas elaboradas por Möller são baseadas no pacote de APIs JBits [GUC99] e são descritas a seguir.

BitProgrammer

Consiste em um pacote de ferramentas desenvolvida para acessar bitstreams a partir de uma máquina local ou remota. Estes bitstreams devem estar no formato binário e podem ser parciais ou totais. As ferramentas permitem visualizar valores que dizem respeito a especificidades do bitstream, procurar valores diferentes dos padrões, modificar valores internos das LUTs, executar a carga remota de um bitstream e localizar LUTs com uma lógica específica. *BitProgrammer* foi desenvolvido fazendo o uso das classes oferecidas pelo JBits.

CircuitCustomizer

É um cliente adicional que usa o pacote de ferramentas *BitProgrammer*. A sua principal vantagem é executar funcionalidades em maior nível de abstração às ferramentas contidas no pacote mencionado. Deste modo, a ferramenta *CircuitCustomizer* permite especificar a localização dos sinais de um projeto de hardware para a posterior visualização e alteração dos valores a partir de uma página HTML. Desta forma o usuário final não precisa saber LUTs específicas, CLBs ou posições onde os parâmetros foram fixados para alterar parâmetros do sistema.

BitAnalyzer

É uma ferramenta para análise de arquivos de configuração totais e parciais em ASCII. O arquivo de configuração é mostrado detalhadamente pela ferramenta. Neste programa, é

possível escolher uma parte do arquivo de configuração total e gerar um arquivo de configuração parcial partindo de um protocolo de geração de arquivos parciais. Pode-se localizar um ou mais bits das LUTs do arquivo de configuração e alterar seus respectivos valores.

CoreUnifier

Partindo do endereçamento de elementos em FPGAs da família *Virtex*, em conjunto com o conhecimento adquirido no desenvolvimento das ferramentas descritas previamente, definiu-se a criação de um software para manipulação de núcleos IPs. A Figura 2.17 apresenta uma tela da interface gráfica da ferramenta em questão.

Esta ferramenta possui as seguintes características:

- Permite a leitura de informações de configurações diretamente do *bitstream*;
- Possibilita a união entre áreas de dois bitstreams diferentes (desde que não sejam sobrepostas), para geração de um terceiro arquivo de configuração;
- A ferramenta possui uma interface gráfica para seleção de áreas específicas do FPGA;

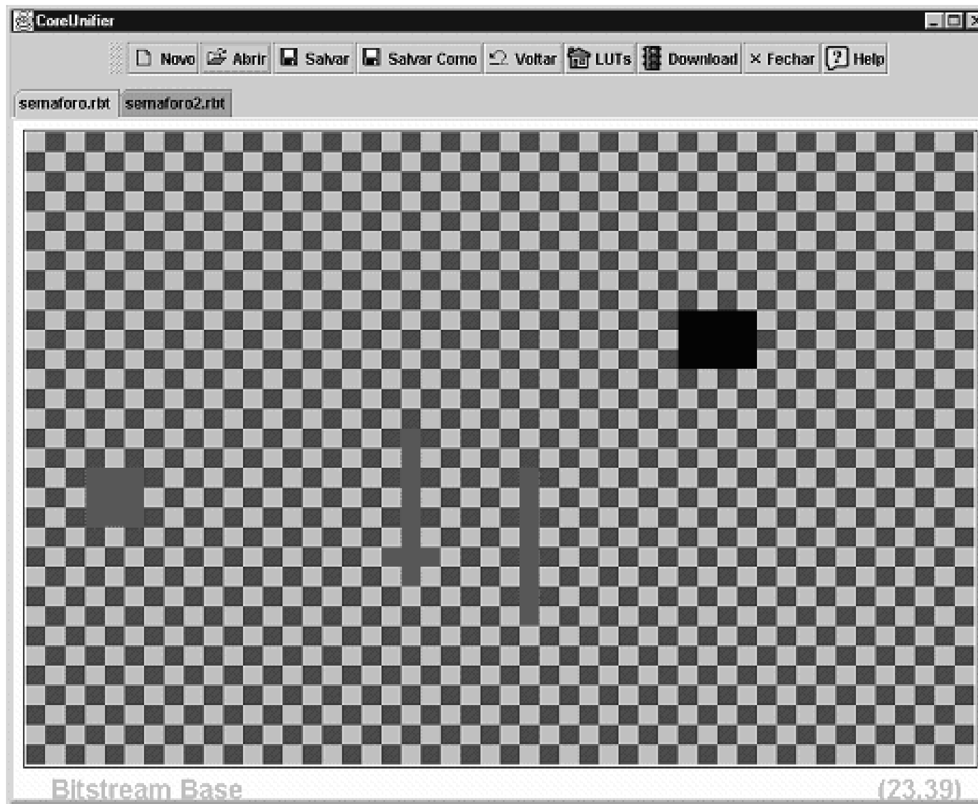

Figura 2.17: Interface gráfica do *CoreUnifier*.

Essas, entre outras ferramentas desenvolvidas no grupo, habilitam a manipulação de arquivos de configuração totais e parciais. O conjunto desenvolvido por Möller manipula apenas arquivos de configuração gerados para dispositivos da família Virtex da Xilinx.

Algumas ferramentas de suporte para reconfiguração parcial e dinâmica foram apresentadas nesta Seção. A Tabela 2.1 apresenta um resumo das características presentes em cada uma das ferramentas de suporte a reconfiguração parcial e dinâmica. A maioria destas ferramentas foram desenvolvidas usando como base o conjunto de classes JBits, pois este habilita a manipulação de arquivos de configuração das famílias de FPGAs Virtex e Virtex-E conforme mostrado na Subsseção 2.5.1. Porém, com o advento de novas famílias de FPGAs da Xilinx, o conjunto de classes JBits deve ser modificado para que este possa acessar especificidades arquiteturais destas novas famílias de dispositivos. Ou seja, tais ferramentas são amplamente dependentes do conjunto de classes JBits.

Tabela 2.1: Características das ferramentas de suporte a reconfiguração parcial e dinâmica.

Ferramentas	**Famílias (Xilinx)**	**Dependende de JBits?**	**Usa ferramentas da Xilinx?**	**Arquivos de entrada**	**Nível de abstração**	**Ano**
JBits [GUC99]	Virtex/ Virtex-E	–	Não	Bitstream	Baixo	1999
JRTR [MCM99]	Virtex/ Virtex-E	Sim	Não	Bitstream	Baixo	2000
JPG [RAG02]	Virtex/ Virtex-E	Sim	Sim	VHDL/ Verilog	Alto	2002
PARBIT [HOR01]	Viretx/ Virtex-E	Não	Não	Bitstream	Alto	2001
BitProgrammer [MÖLL03]	Virtex	Sim	Não	Rawbits	Baixo	2001
BitAnalyser [MÖLL03]	Virtex	Não	Não	Rawbits	Baixo	2001
CoreUnifier [MÖLL03]	Virtex	Não	Não	Rawbits	Médio	2001

Por outro lado, as ferramentas que não fazem o uso de JBits, como por exemplo as ferramentas PARBIT, *CoreUnifier*, *BitAnalyser* e outras, devem ser modificadas, ou até mesmo reescritas para poderem ter acesso e alteração do conteúdo de componentes arquiteturais de uma nova família de FPGAs da Xilinx, fator que limita a portabilidade destas ferramentas para outras tecnologias.

No entanto, existe um método para reconfiguração parcial baseada em núcleos, como o fluxo do Projeto Modular [LIM03], que provê acesso a todas as famílias de FPGAs da Xilinx. Esta característica permite que uma ferramenta desenvolvida usando tal fluxo não precise ser reescrita para cada família de FPGAs, e também não é necessário o conhecimento de toda a arquitetura do FPGA, pois este fluxo manipula arquivos de configuração total e parcial num maior nível de abstração. O Capítulo 4 detalha o fluxo do Projeto Modular.

Capítulo 3

Reconfiguração Baseada em Núcleos e Interconexão de Núcleos

3.1 Métodos Habilitadores

3.1.1 Barramento proposto por Palma

Palma [PAL02] propôs um barramento para interconectar núcleos IP reconfiguráveis possibilitando o desenvolvimento de sistemas dinamicamente reconfiguráveis. Este barramento faz parte de um módulo denominado *controlador*, e é uma porção de hardware fixo no FPGA. O controlador é responsável pela comunicação com o mundo externo (pinos de entrada/saída do dispositivo) e pela comunicação com os núcleos IP.

O dispositivo reconfigurável deve ser inicialmente carregado apenas com o controlador localizado em uma posição fixa do dispositivo reconfigurável. Os núcleos de propriedade intelectual são carregados sob demanda. A comunicação entre o controlador e os demais núcleos IP é feita através de pinos virtuais. Estes pinos são implementados através de *buffers tristate*.

A idéia inicial era utilizar *buffers* na fronteira entre o controlador e o núcleo IP, responsáveis pela conexão entre os mesmos. Estes *buffers* fariam parte tanto da descrição do controlador quando da descrição do núcleo IP. Palma mostrou que esta abordagem não seria factível de implementação. No momento da inserção do núcleo IP, recursos adicionais de lógica de um dado CLB que faria interface com o mesmo seria destruído. Então foi criada uma abordagem que utiliza duas camadas de *buffers*, uma camada pertencendo ao controlador e outra pertencendo ao núcleo IP, como mostra a Figura 3.1. A conexão de núcleos IP ao controlador é feita por uma linha comum de roteamento.

Junto ao controlador, o núcleo IP deve ser inserido em um invólucro (*wrapper*), contendo módulos denominados de *Send* e *Receive*, além dos *buffers* de interface conforme mostrada na Figura 3.2. Os módulos *Send* e *Receive* têm o objetivo de transmitir e receber dados para

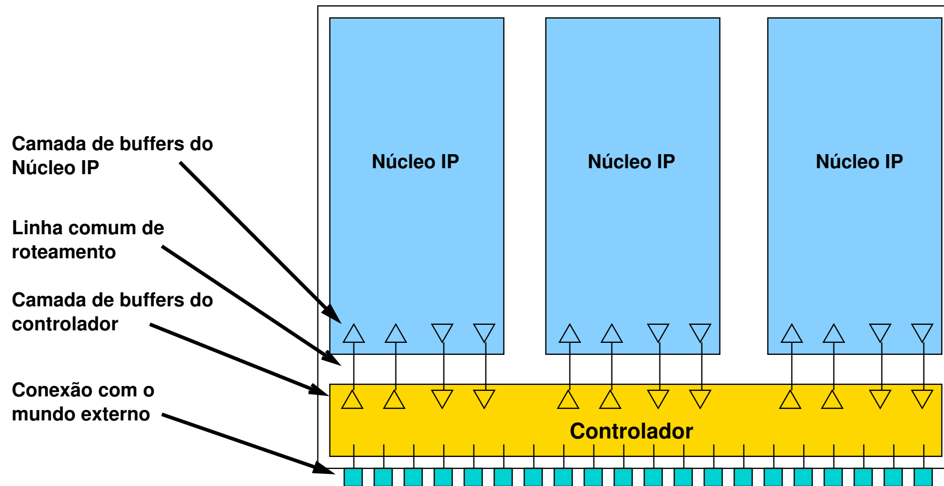

Figura 3.1: Estrutura de um SoC usando o método proposto por Palma [PAL02]. Neste, duas camadas de *buffers* que possibilitam a implementação do controlador no FPGA.

o/do barramento dependendo de solicitações externas.

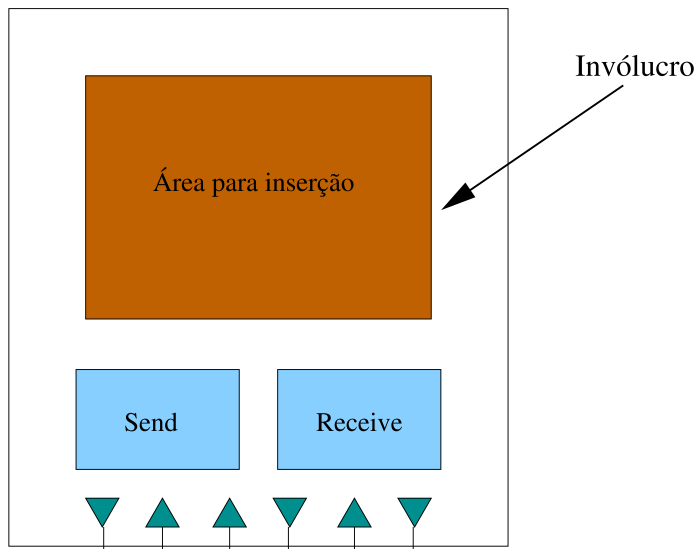

Figura 3.2: Estrutura do invólucro para inserção de núcleos IP no método proposto por Palma [PAL02].

O fluxo de desenvolvimento proposto sofre com o problema relacionado à falta de suporte para restringir o roteamento. Através de um software para visualização de elementos roteados e posicionados dentro do FPGA, o roteamento é feito manualmente tornando pouco automatizado o fluxo proposto por Palma e, dependendo a complexidade do núcleo IP a ser conectado ao controlador, pode ser inviável desenvolver sistemas reconfiguráveis mais complexos. Estas limitações na arquitetura de roteamento dos *buffers tristate* impossibilitam o uso

de barramentos com um número razoável de bits (largura da palavra com 16 bits, por exemplo). Segundo Palma, apenas um experimento usando barramento serial para transmissão transmissão de dados entre os núcleos IP foi factível de implementação.

3.1.2 Reconfiguração parcial via Projeto Modular

Projeto Modular, originalmente, é utilizado para desenvolver projetos de forma distribuída. Um projetista é responsável pela manutenção e atualização do arquivo de nível hierárquico mais alto e os projetistas restantes são responsáveis pelos módulos que compõem o projeto [XIL01b]. Estes são sintetizados separadamente. No final do fluxo do Projeto Modular, todos os módulos são reunidos em um único projeto junto ao módulo de maior nível hierárquico, resultando no projeto final pronto para ser configurado no FPGA.

Este fluxo utiliza um conjunto de ferramentas do pacote ISE (*Integrated Software Environment*) [XIL01b], usados para mapear, posicionar, rotear e gerar arquivos de configuração parciais e totais. O Projeto Modular permite que um módulo seja modificado de forma independente, não afetando os demais módulos que compõe o projeto. Os módulos e o arquivo de maior nível hierárquico são sintetizados separadamente.

De forma alternativa, é possível gerar módulos reconfiguráveis armazenados dentro de arquivos de configuração [LIM03]. A partir desta premissa, é possível configurar o FPGA com um arquivo de configuração total inicial e, logo após, configurar sucessivos arquivos de configuração parciais no FPGA, para a execução dinamicamente reconfigurável. A Figura 3.3 mostra uma visão geral das fases do fluxo do Projeto Modular. Estas serão apresentadas de maneira mais aprofundada no Capítulo 4.

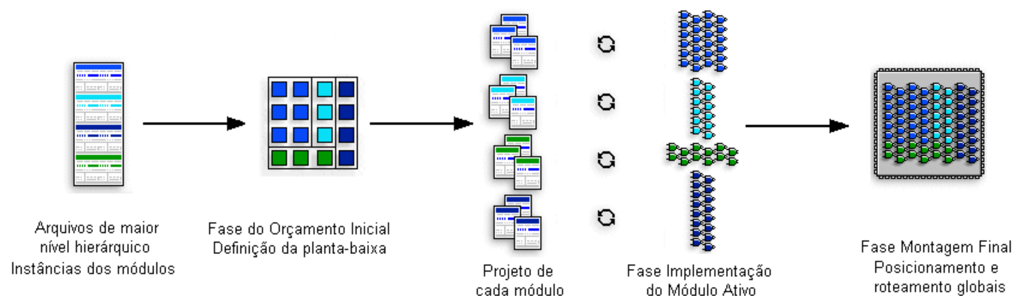

Figura 3.3: Visão geral do fluxo de reconfiguração dinâmica e parcial baseada no Projeto Modular.

Com o uso do fluxo de Projeto Modular, o usuário não precisa se preocupar com vários detalhes inerentes à implementação do projeto. O nível de abstração do fluxo esconde especificidades arquiteturais do usuário, permitindo que este se detenha mais no desenvolvimento do projeto no nível HDL e arquivos de restrições. No entanto, existem diversas fases da execução do Projeto Modular efetuadas manualmente. A codificação HDL também deve-se adequar em certas medidas às características impostas pelo fluxo do Projeto Modular.

Ao contrário do fluxo do barramento proposto por Palma, os núcleos reconfiguráveis são conectados ponto-a-ponto pelas bus macros. Não é necessário lógicas adicionais para controle do barramento.

Dentro do escopo deste trabalho, adotou-se este fluxo do Projeto Modular para prover a reconfiguração parcial e dinâmica.

3.2 Métricas para Avaliação de SDRs

O incremento na eficiência provido por SDRs não é obtido sem custo [WIR98]. Para transferência do arquivo de configuração do circuito de uma unidade de armazenamento que esteja situada fora do FPGA para dentro da memória de configuração, tempo e largura de banda da memória adicionais são necessários. Em alguns casos, este tempo extra pode reduzir e até mesmo eliminar o número de benefícios oferecidos de uma especialização de tempo de execução. No entanto, a relação entre custo de reconfiguração e ganho em desempenho e economia de recursos pode compensar em muito o tempo de configuração. Este é o principal objetivo de ser atingido durante o projeto de SDRs.

Um método que calcula o equilíbrio entre eficiência e custo de configuração foi proposto por Wirthlin e Hutchings para avaliar a utilização de SDRs. Este método é baseado na *métrica de densidade funcional* [WIR98].

A métrica de densidade funcional é definida nos termos de custo de implementação de uma lógica em hardware. O custo (C) é medido como o produto entre a área total requerida para implementar a lógica em hardware (A) e o tempo total de operação (T):

$$C = AT \tag{3.1}$$

A variável T inclui o tempo necessário para execução, controle, inicialização e transferência de dados. Densidade Funcional (D) mede a vazão computacional (número de operações por segundo) de uma medida de hardware, e é definida como o inverso de C:

$$D = \frac{1}{C} = \frac{1}{AT} \tag{3.2}$$

A métrica densidade funcional em 3.2 será utilizada para comparar circuitos modificados estaticamente contra as alternativas em reconfiguração dinâmica.

O incremento (I) na densidade funcional de alguns SDRs sobre as alternativas equivalentes estaticamente reconfiguráveis (D_s) é computada como a diferença normalizada entre D_r e D_s, como segue:

$$I = \frac{\Delta D}{D_s} = \frac{D_r - D_s}{D_s} = \frac{D_r}{D_s} - 1 \tag{3.3}$$

A porcentagem de incremento é medida multiplicando a Equação 3.3 por 100.

A maior diferença entre SDRs e seus equivalentes estáticos é o custo adicional de reconfiguração. Isto força a adição do tempo de configuração ao tempo total de operação de um SDR. Para estes sistemas, o tempo total de operação T inclui o tempo total de execução (T_e) e o tempo total de configuração (T_c):

$$T = T_c + T_e \tag{3.4}$$

Substituindo T em 3.2 é evidente que o tempo de configuração reduz a densidade funcional.

Contudo, apesar do tempo de configuração absoluto ser um importante parâmetro, a relação entre o tempo de configuração e o tempo de execução é muito mais informativo. A taxa de configuração:

$$f = \frac{T_c}{T_e} \tag{3.5}$$

define este parâmetro importante. O tempo total de operação pode ser expresso como:

$$T = T_e(1 + f) \tag{3.6}$$

Substituindo este tempo em 3.2 obtém-se a métrica de densidade funcional:

$$D_r = \frac{1}{AT_e(1 + f)} \tag{3.7}$$

Como mostrado em 3.7, tempos longos de configuração podem ser tolerados se houver um tempo longo de execução correspondente (f pequeno). No limite, se f tende a zero, a sobrecarga imposta pela configuração é irrelevante. Tais sistemas aproximam-se da máxima densidade funcional provida por SDRs. Este valor máximo (D_{max}) é calculado ignorando-se o tempo de configuração:

$$D_{max} = \lim_{f \to 0} D_r = \frac{1}{AT_e} \tag{3.8}$$

Usando-se D_{max}, o limite superior do incremento (I_{max}) sobre um sistema estático pode ser encontrado. Este parâmetro sugere o benefício máximo de um sistema SDR, e é uma boa indicação da relativa utilização de SDR para uma dada aplicação:

$$I_{max} = \frac{D_{max}}{D_s} - 1 \tag{3.9}$$

Wirthlin e Hutchings [WIR98] ao propôr essas métricas, sugerem que ao projetar SDRs, procura-se aumentar ao máximo o tempo de execução do sistema de cada configuração, para que se justifique o uso de SDR.

3.3 Padronização da Comunicação Intra-chip

As interfaces de comunicação padrão podem ser classificadas em *abordagem centrada no meio de comunicação* (*bus-centric*) e *abordagem centrada na interface de comunicação* (*core-centric*).

3.3.1 Abordagem centrada no meio de comunicação

Núcleos devem seguir o protocolo do meio de comunicação ou serem adaptados para poderem acessar o meio de comunicação. A maioria das interfaces padrão centradas no meio de comunicação existentes são baseadas em arquiteturas de barramentos. A Figura 3.4 mostra a desvantagem de modelos baseados nesta abordagem.

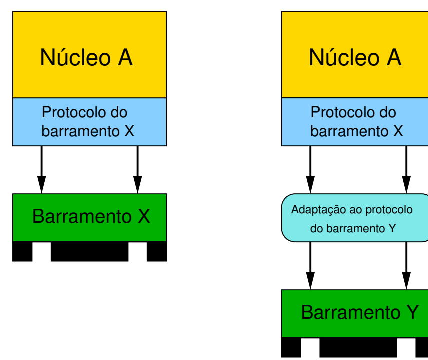

Figura 3.4: Exemplo da desvantagem principal de comunicação baseada na abordagem centrada no meio de comunicação. Neste modelo [OCP02], um núcleo IP denominado A com protocolo de comunicação adaptado a um barramento X pode ser conectado diretamente ao barramento X. No entanto, para conectar o mesmo núcleo a um barramento Y, (protocolo de comunicação diferente do barramento X) são necessárias alterações no protocolo deste núcleo para adaptar este ao barramento Y.

Existem vários sistemas de interfaces baseados em barramentos. Alguns são descritos a seguir:

CoreConnect [IBM02a] é uma interface de comunicação baseada na arquitetura de barramento desenvolvida pela IBM que provê integração e reutilização de núcleos IP em SoCs.

A IBM disponibiliza, além das especificações do *CoreConnect*, um conjunto de ferramentas, modelos HDL comportamentais dos elementos da arquitetura.

Wishbone [SIL01, SIL03] é uma interface baseada na arquitetura de barramento desenvolvida pela SILICORE, que utiliza uma arquitetura mestre/escravo. Isto significa que os módulos com interfaces mestre iniciam transações de dados, e interfaces escravo respondem a essas transações. Este barramento foi desenvolvido pela necessidade de uma solução boa e confiável para integração dos núcleos IP em SoCs, pela necessidade de uma especificação de interface comum para facilitar as metodologias de projeto estruturadas para grandes equipes de projeto.

A arquitetura do *Wishbone* é análoga a um barramento de microcomputador, sendo que esta provê uma solução flexível para integração que pode ser facilmente adaptada a uma aplicação específica, provê uma variedade de ciclos de acesso ao barramento e de larguras de caminhos de dados para atender diferentes sistemas. *Wishbone* também permite que os núcleos IP sejam projetados por vários fornecedores.

AMBA *Advanced Microcontroller Bus Architecture* [ARM04] é um padrão aberto desenvolvido pela ARM Corp, ou seja, uma especificação de barramento que define uma estratégia para interconexão e gerenciamento de blocos funcionais para o desenvolvimento de SoCs.

Finalmente, Avalon [ALT02] é uma arquitetura de barramento para integração de módulos no desenvolvimento de SoCs desenvolvida para FPGAs. Também é uma interface que especifica conexões de entre componentes mestres e escravos, e temporização na comunicação entre os módulos. Avalon suporta características como barramento para múltiplos mestres, entre outros. Esta última característica provê flexibilidade na construção de SoCs e alta largura de banda para comunicação com os periféricos. Vários componentes mestre podem realizar transações simultaneamente (não com o mesmo componente escravo).

3.3.2 Abordagem centrada na interface de comunicação

Esta abordagem tem sido proposta para facilitar o desenvolvimento de núcleos IP, já que o projetista destes irá se concentrar somente no desenvolvimento do núcleo, e não em como este núcleo interage com o ambiente. Assim, é possível integrar um núcleo a qualquer sistema com o mesmo protocolo sem a necessidade de alteração em termos de largura de banda, freqüência e sinais de interface deste núcleo. A Figura 3.5 ilustra a abordagem centrada na interface de comunicação através de um exemplo de arquitetura.

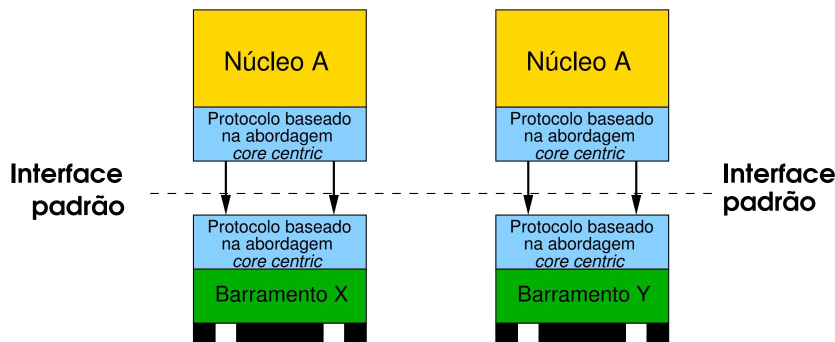

Figura 3.5: Exemplo de uma arquitetura baseada na abordagem centrada na interface de comunicação [OCP02]. Um núcleo IP denominado A projetado com protocolo de comunicação baseado no modelo core-centric pode ser conectado ao barramento X ou Y sem a necessidade de adaptação do protocolo do núcleo ao protocolo de comunicação de nenhum dos dois barramentos, desde que estes sejam "encapsulados" com o protocolo de comunicação baseado na abordagem centrada na interface de comunicação. Alterações de protocolos de comunicação tornam-se desnecessário.

Existem alguns protocolos de comunicação baseados no modelo disponíveis tais como:

O *Open Core Protocol* (OCP) [OCP02] é um protocolo de comunicação, que provê interface de comunicação entre núcleos no desenvolvimento de SoCs. OCP pode ser aplicado à qualquer tipo núcleo IP, inclusive barramentos. Os núcleos IP são empacotados pelo protocolo. Desta forma, este protocolo é independente do sistema de interconexão adotado. O uso deste protocolo de comunicação minimiza problemas de conectividade entre núcleos IP, possibilitando o reuso de um núcleo IP de um determinado projeto. Desta forma, o uso do OCP torna mais simples e rentável a integração deste núcleo, tanto para o provedor quanto para o integrador do núcleo.

VSIA *Virtual Socket Interface Alliance* é um consórcio internacional de companhias dedicadas a acelerar a mudança de paradigmas de projeto de ASICs para SoCs [VSI97]. Esta desenvolve padrões abertos para fazer a interface entre núcleos IP, tornando possível o projeto de SoCs.

Dentro do escopo deste trabalho, abordou-se o padrão de comunicação OCP pela sua disponibilidade no grupo de estudo local e por ser um padrão com a documentação gratuita.

3.3.3 Protocolo OCP

O OCP é um protocolo de comunicação que possui uma série de regras bem definidas , as quais provêem a integração de núcleos IP à um SoC, independente do sistema de interconexão adotado. O uso deste protocolo de comunicação pode minimizar problemas de interconexão de núcleos IP, possibilitar a reutilização de um núcleo de um dado projeto, tornando mais simples e rentável a integração deste núcleo, tanto para o provedor quanto para o integrador do núcleo.

OCP define uma interface ponto-a-ponto entre duas entidades como núcleos IPs e módulos de interface de barramento (*bus wrappers*). As entidades que usam o protocolo OCP agem como mestre e escravos, ou sejam iniciando requisições e respondê-la a elas, respectivamente.

A Figura 3.6 ilustra um sistema contendo um barramento empacotado com interfaces OCP para o mundo externo e três núcleos IP: um atuando como mestre, outro apenas como escravo, e um terceiro atuando com características mestre e escravo. Nota-se que a comunicação no modelo OCP é sempre ponto-a-ponto, exigindo sempre pares mestre-escravo. Percebe-se na Figura 3.6, que o módulo OCP escravo interno ao barramento, atua de fato como mestre do barramento.

As características do núcleo IP determinam se o núcleo necessita de um empacotamento mestre ou escravo, ou ambos. A transferência de dados através deste sistema acontece da seguinte forma: o iniciador do sistema (núcleo IP com módulo OCP mestre) envia um comando e disponibiliza dados ao módulo escravo do barramento. Este módulo escravo atua como mestre do barramento, disparando um sinal de *request* dentro do barramento do sistema. O módulo OCP mestre recebe do módulo OCP escravo (mestre do barramento) o comando e os dados, enviando estes para o módulo OCP escravo do núcleo IP destino.

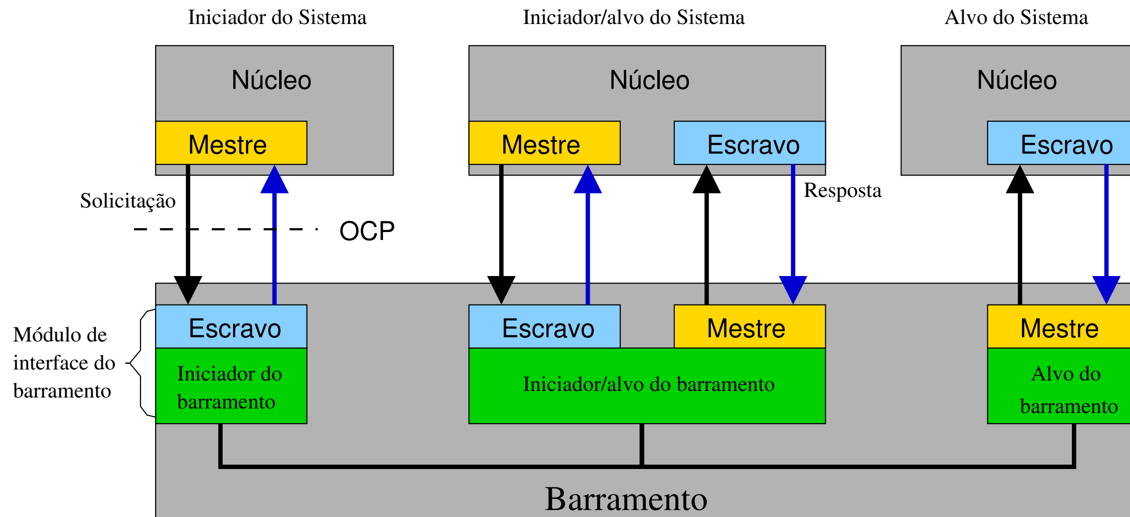

Figura 3.6: Sistema usando um barramento e núcleos empacotados com instâncias OCP.

Além do suporte a modelos de transferência de dados simples baseado em requisição e resposta, o OCP suporta modelos de transferência de dados de alto-desempenho baseados em *multi-threads* e *pipeline* [OCP02].

Conjunto de sinais do OCP

Os sinais do OCP são divididos em três grupos: sinais de fluxo de dados, sinais opcionais de controle e sinais de teste. Os sinais de fluxo de dados são responsáveis pela troca de dados entre os núcleos (mestre e escravo). Os sinais opcionais de controle têm o objetivo de transmitir informações de controle tais como: inicialização, interrupção, erro e identificação de núcleos; sinais para mudanças de controle e informação de estado entre um núcleo e o resto do sistema. Finalmente, os sinais de teste servem para varredura e teste da interface de um núcleo IP.

Neste trabalho, o enfoque será dado apenas para o conjunto de sinais básicos do OCP. Para maiores informações, veja [OCP02].

Todos os sinais do OCP são amostrados na borda de subida do relógio (*clock*). Alguns sinais do OCP são configurados a partir de parâmetros, dentre estes citam-se o *addr_ wdth* e o *data_ wdth*. Estes e outros parâmetros são descritos no decorrer desta Seção. Dentre todos os sinais suportados pelo protocolo, existe um conjunto de sinais denominados de sinais básicos. Estes devem compor a interface de qualquer núcleo IP baseado no OCP.

Sinais de fluxo de dados

Sinais básicos

Nesta Seção são descritos os sinais básicos que devem compor a interface de um núcleo IP.

A Tabela 3.1 apresenta os sinais básicos do padrão OCP e suas respectivas funcionalidades. Estes sinais são:

- *Clk*: a borda de subida deste sinal sincroniza todos os sinais da interface;
- *MDAddr*: especifica o endereço em função do núcleo com comportamento escravo desejado. A largura deste sinal pode ser configurada a partir do parâmetro *addr_wdth*;
- *MCmd*: indica o tipo de transferência que deve ser efetuada (leitura, escrita, entre outras). Os comandos de codificação deste sinal são descritos na Tabela 3.2;
- *MData*: este sinal transporta os dados do mestre para o escravo. A largura deste sinal é configurada a partir do parâmetro *data_wdth*;
- *SCmdAccept*: quando o sinal está no nível lógico alto, indica que o núcleo IP escravo aceita a requisição de transferência efetuada pelo núcleo mestre;
- *SData*: este sinal transporta os dados do escravo para o mestre. A largura deste sinal é configurada a partir do parâmetro *data_wdth*;
- *SResp*: sinal de resposta do núcleo escravo para uma requisição de transferência do núcleo mestre. A codificação de resposta é apresentada na Tabela 3.3;

Tabela 3.1: Sinais básicos do OCP.

Nome	Largura	Parâmetro de configuração de largura	Controle	Função do sinal
Clk	1	Fixo	Variável	Relógio OCP
Maddr	1-32	*Addr_wdth*	Mestre	Endereço de Transferência
MCmd	3	Fixo	Mestre	Endereço de Comando
MData	8/16/32/64/128	*Data_wdth*	Mestre	Escrita de Dados
SCmdAccept	1	Fixo	Escravo	Aceitação de Transferência
SData	8/16/32/64/128	*Data_wdth*	Escravo	Leitura de Dados
SResp	2	Fixo	Escravo	Resposta de Transferência

Sinais de extensão

Os sinais de extensão simples permitem a adição de espaço de endereçamento, suporte ao modo rajada, *datahandshake* (sinais de estabelecimento de comunicação entre dispositivos utilizados para assegurar a transferência de dados) e controle de fluxo de resposta. Também melhoram a capacidade de transferência básica, introduzindo o conceito de *threads*. Porém estes sinais não serão apresentados neste Capítulo.

Tabela 3.2: Comandos de codificação do sinal *MCmd*.

MCmd[2:0]	Tipo de Transação	Mnemônico
0 0 0	Ociosa	IDLE
0 0 1	Escrita	WR
0 1 0	Leitura	RD
0 1 1	ReadEx	RDEX
1 0 0	Reservada	-
1 1 0	Reservada	-
1 1 1	Reservada	-

Tabela 3.3: Comandos de codificação do sinal *SResp*.

SResp[2:0]	Tipo de Transação	Mnemônico
0 0	Sem Resposta	NULL
0 1	Dado Válido/Aceita	DVA
1 0	Reservado	-
1 1	Erro de Resposta	ERR

Protocolo de comunicação

No protocolo OCP, existem vários modos de transferência. Porém neste Capítulo, deu-se o enfoque principal para o protocolo de uma transferência simples de escrita e leitura de dados.

A Figura 3.7 apresenta um diagrama do protocolo de comunicação de uma transferência simples de escrita e leitura de dados.

A seqüência para a transferência segue os seguintes passos:

A) O mestre começa a fase de requisição quando ocorre a transição do sinal *Mcmd* do estado IDLE para WR. No mesmo instante é apresentado um endereço válido (A_1) no campo *Maddr*, e dado válido no campo *Mdata*. Estes três sinais devem ser apresentados juntos, na mesma borda de subida do relógio (borda 1). O escravo, por sua vez, ativa o sinal de *ScmdAccept* no mesmo ciclo, permitindo a transferência sem latência;

B) Nesse instante, o escravo captura os valores referentes ao endereço, aos dados e utiliza estes internamente para efetuar a escrita. Como o *ScmdAccept* está ativo, isto indica término da fase de requisição;

C) O mestre inicia a fase de requisição de leitura (RD_2) atribuindo o comando de leitura RD, no sinal *Mcmd*. Neste momento o mestre apresenta um endereço (A_2) no campo $Maddr$;

D) O escravo captura o valor do *Maddr* e utiliza este internamente para determinar que dado deve ser retornado. A fase de resposta começa quando o comando *Sresp* passa de NULL para DVA. No mesmo instante o escravo apresenta o dado do campo *Sdata*. Como o sinal *ScmdAccept* está ativo, isto indica término da fase de requisição;

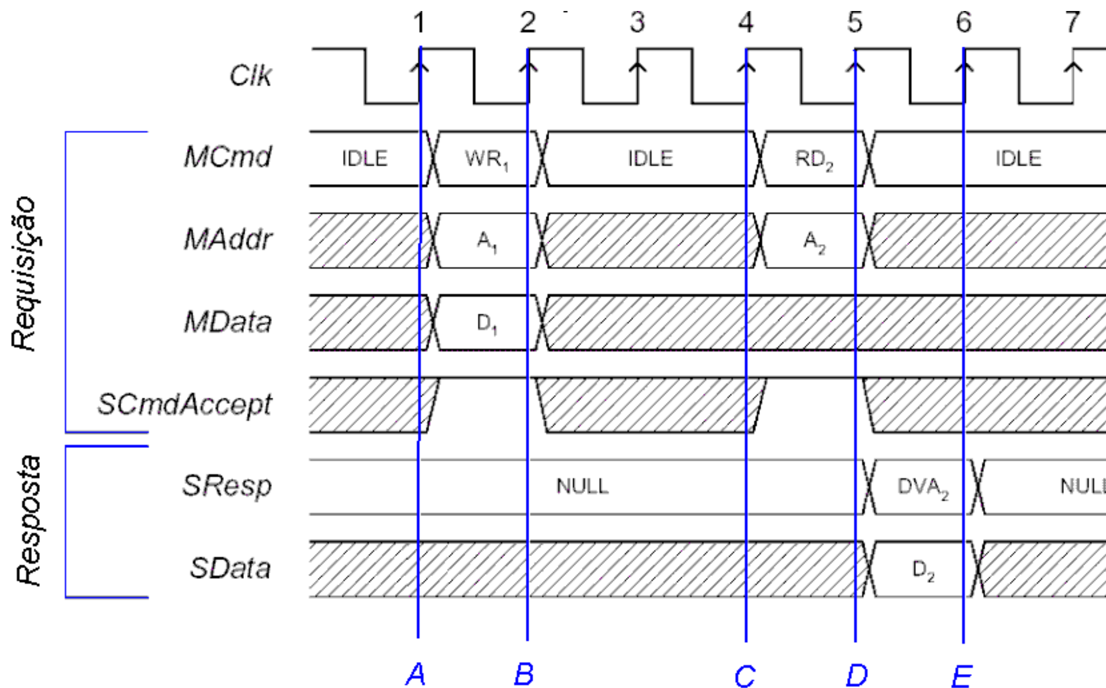

Figura 3.7: Diagrama do protocolo de comunicação de uma transferência de escrita e leitura de dados.

E) Ao perceber que o campo *Sresp* indica um dado válido, o mestre captura o dado de leitura do sinal *Sdata*, completando a fase de resposta. A latência gerada por essa fase é igual a 1 (um) ciclo de relógio;

Para validação do protocolo OCP, deve-se utilizar uma ferramenta chamada *CoreCreator* (Figura 3.8). Esta ferramenta valida funcionalmente o projeto utilizando OCP através de simulação e da geração de tráfego de dados na entrada e saída do núcleo ou sistema a ser validado. O núcleo pode gerar ou receber tráfego, dependendo de sua característica inicializador/alvo (mestre/escravo).

A Figura 3.8 mostra um exemplo de núcleo sendo validado pela ferramenta *CoreCreator*. O bloco denominado de *Qmaster* simula o comportamento de um núcleo mestre, inicializando o sistema simulado e gerando tráfego para o núcleo escravo referenciado por *W_ram* para validação do mesmo. Entre o núcleo a ser validado e o núcleo inicializador do sistema, existe um monitor que verifica e armazena informações referente a transferência de dados entre os núcleos. Na ferramenta *CoreCreator* existe comandos para a execução de cada fase de certificação. Estas são descritas abaixo:

1. *Create Netlist*: nesta fase são criados arquivos de configuração de núcleos para serem usados na simulação e síntese; São descrições de hardware (*testbenches*) os quais tem a função de simular o comportamento de cada módulo a ser testado;

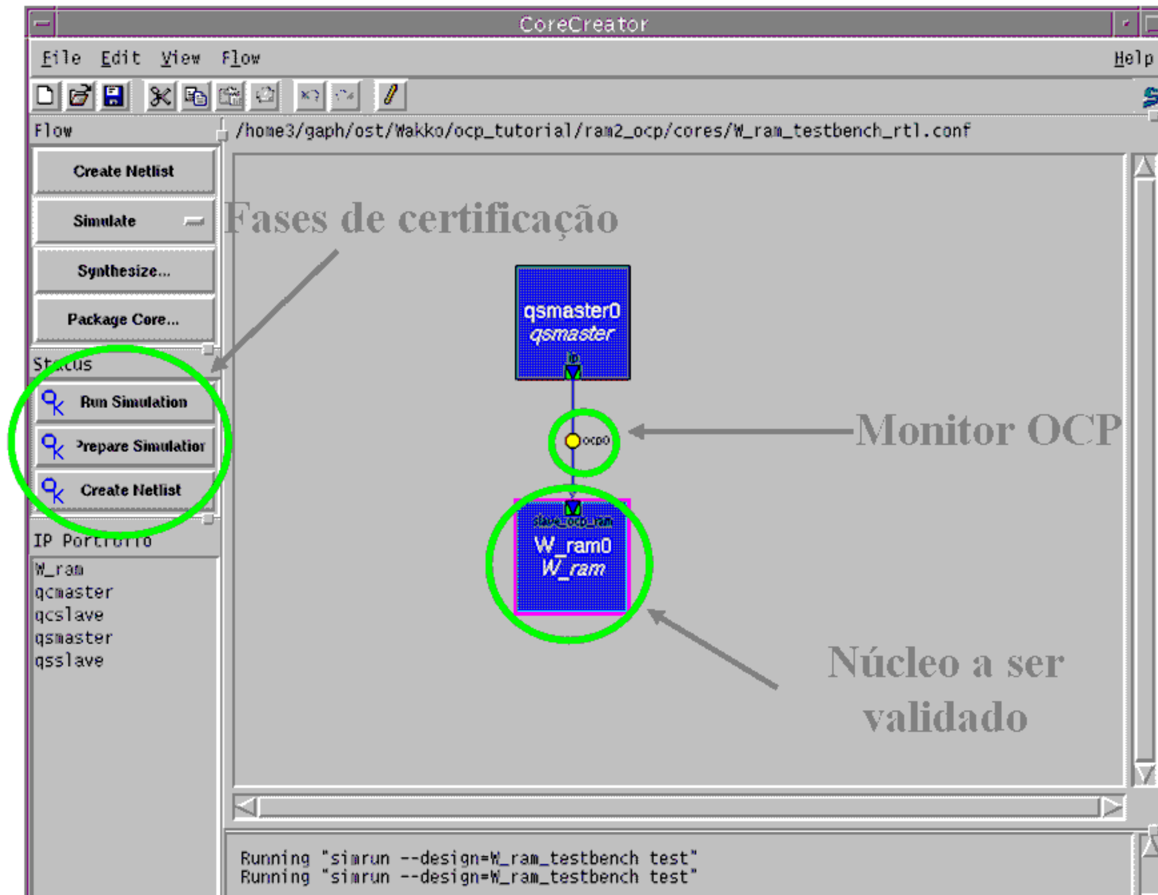

Figura 3.8: Ferramenta *CoreCreator* para validação e certificação do protocolo OCP aplicado em núcleos IP ou sistemas. A ferramenta valida os módulos com OCP através da geração de tráfego de dados.

2. *Prepare Simulation*: nesta fase, arquivos de teste e arquivos STL (*Sonics Transaction Language*) são lidos e produzem um modelo comportamental mestre ou escravo do núcleo gerador de tráfego. Este comportamento é descrito num arquivo que será usado na simulação;

3. *Run Simulation*: nesta fase, os arquivos gerados pela fase anterior são usados na simulação. A simulação gera resultados para análise e verificação do núcleo. Relatórios são criados aqui com o intuito de verificação e eventuais análises comportamentais.

4. *Analyze Results*: o processamento dos resultados gerados na fase anterior é realizado. Neste ponto, a ferramenta pode informar se o protocolo descrito em HDL é compatível ou não com o protocolo OCP.

3.4 Contexto do Trabalho

O presente trabalho é parte de um ambiente proposto para desenvolvimento e implementação de sistemas dinamicamente reconfiguráveis denominado PADREH (*Partial and Dynamic Reconfiguration of Hardware*) que é apresentado na Figura 3.9, cuja finalidade é permitir a geração de sistemas complexos, conceitualmente maiores que o dispositivo alvo. O sistema PADREH subdivide-se em três partes:

- Módulo de captura e validação funcional de projeto [MOR03b];
- Módulo de particionamento e escalonamento [MAR02];
- Módulo de síntese física e parte da infra-estrutura de reconfiguração [BRI03];

De acordo com a Figura 3.9, o módulo de *captura e validação funcional do projeto* responsabiliza-se por descrever, refinar e validar o sistema no nível transacional (TLM - *Transaction Level Modeling*). A descrição no nível RTL (*Register Transfer-level*) é a entrada do módulo de *particionamento e escalonamento*. Este módulo particiona o sistema de acordo com informações de comportamento do sistema e de características do SoC alvo da implementação. O módulo de *particionamento* e *escalonamento* gera arquivos que descrevem o comportamento do sistema na forma de HDLs (*Hardware Description Language*) para o módulo de *síntese física e parte da infra-estrutura de reconfiguração*. O módulo de *síntese física e parte da infra-estrutura de reconfiguração* executa as seguintes tarefas abaixo:

- gera arquivos de configurações totais e parciais de acordo com o particionamento;
- adiciona módulo controlador de configurações [CAR04] parametrizado de acordo com as características do sistema;
- gera a implementação física de uma rede de interconexão entre os núcleos do sistema fornecido pelo módulo de *particionamento* e *escalonamento*;
- gera arquivos de configuração totais e parciais;

O desenvolvimento do arcabouço PADREH proposto implica abordar temas tais como (i) o particionamento de hardware; (ii) o escalonamento de configurações para execução; (iii) a geração de arquivos de configuração parciais e totais; (iv) a configuração de dispositivos parcialmente reconfiguráveis; (v) a representação de informações a respeito do cronograma de escalonamento e (vi) o gerenciamento da operação de configurações.

O presente trabalho propõe uma parte da infra-estrutura de reconfiguração que faz parte do módulo *síntese física e parte da infra-estrutura de reconfiguração*. Sua finalidade é a geração de arquivos de configuração parciais e totais do projeto, utilizando métodos de interconexão de núcleos usando OCP.

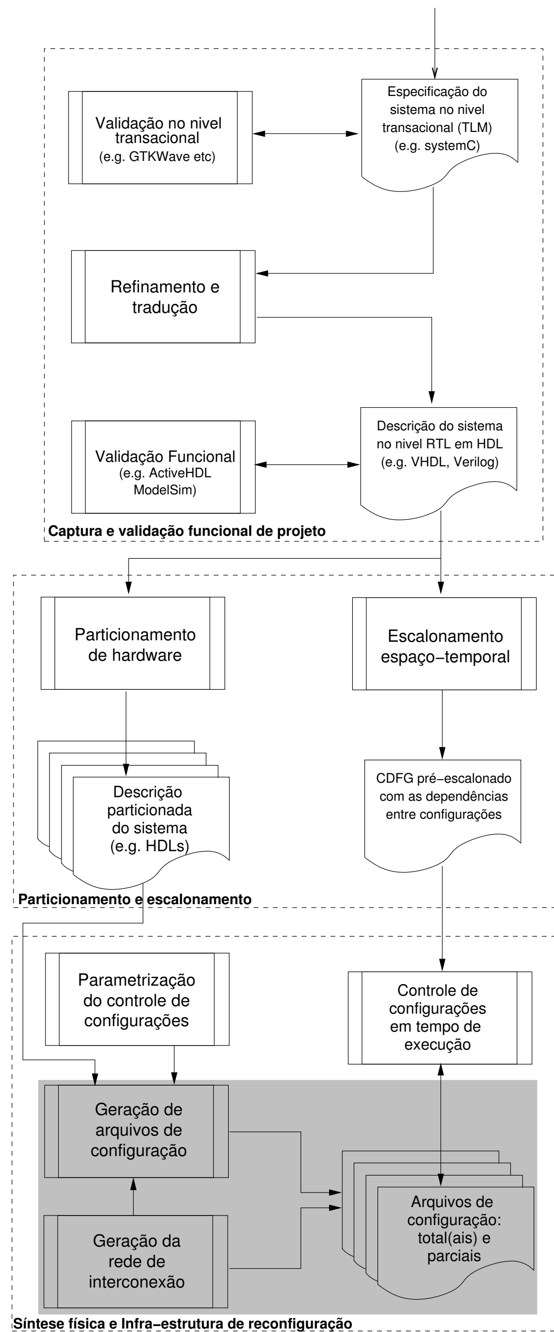

Figura 3.9: Fluxo de desenvolvimento e implementação de sistemas dinamicamente reconfiguráveis de acordo com o arcabouço PADREH proposto. A área hachurada representa o agrupamento das tarefas propostas neste trabalho.

Capítulo 4

Fluxo de Projeto Modular

Usando o Fluxo de Projeto Modular descrito em [LIM03], é possível desenvolver projetos com reconfiguração parcial e dinâmica. Projeto Modular, em seu objetivo inicial, é uma técnica utilizado para desenvolver projetos complexos de forma distribuída. Um projetista é responsável pela manutenção e atualização do arquivo de nível hierárquico mais alto e os projetistas restantes são responsáveis pelos módulos que compõem o projeto [XIL01b]. Projeto Modular permite que um módulo seja modificado de forma independente, não afetando os demais módulos que compõe o projeto. Os módulos são sintetizados separadamente. No final do Fluxo de Projeto Modular, todos os módulos são reunidos em um único projeto junto ao módulo de maior nível hierárquico, resultando no projeto final pronto para ser prototipado. Executando o Projeto Modular de forma alternativa, é possível desenvolver módulos reconfiguráveis sob forma de bitstreams parciais. Deste modo, é possível inicializar o FPGA com um bitstream e então fazer os sucessivos downloads de bitstreams parciais no FPGA.

Este fluxo foi usado para o desenvolvimento de sistemas dinamicamente reconfiguráveis no trabalho a partir da disponibilidade de FPGAs Virtex II da Xilinx, uma das famílias de dispositivos que permite o uso dos bitstreams gerados pelo fluxo. Outra característica deste fluxo é a geração de bitstreams parciais dos módulos reconfiguráveis sem roteamento manual, caso algum sinal não seja roteado pela ferramenta de síntese física. O Projeto Modular é dividido em três fases:

1. *Fase do Orçamento Inicial*: nesta fase, determina-se qual é a estrutura do arquivo de nível hierárquico mais alto do projeto. Atribuições de restrições de área e temporização também são realizadas nesta fase;

2. *Implementação dos Módulos Ativos*: cada módulo é sintetizado e implementado separadamente. Esta fase gerará bitstreams parciais de cada módulo usado na reconfiguração;

3. *Montagem Final*: a terceira e última fase do Projeto Modular onde é(são) gerado(s) o(s) módulo(s) de maior nível hierárquico, une-se todos os outros módulos àquele e sintetiza-os em um único projeto;

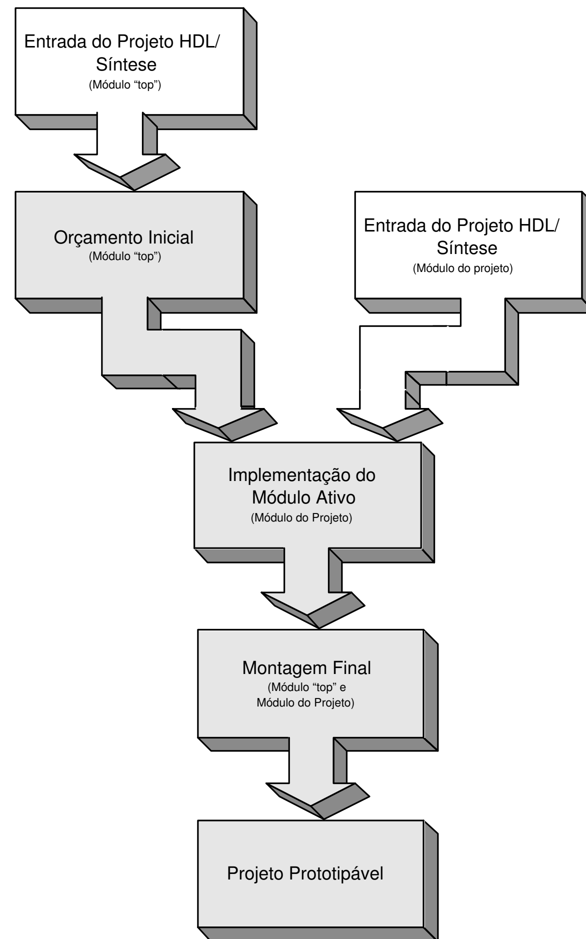

Figura 4.1: Visão geral das fases do Projeto Modular (em cinza) [XIL01b]. Caixas brancas representam a definição dos arquivos de nível hierárquico mais alto e os módulos de nível hierárquico inferior em HDLs. A síntese lógica de todos os módulos é realizada individualmente e ocorre antes da execução da Fase do Orçamento Inicial do Fluxo de Projeto Modular.

4.1 Preparação dos Módulos e Fases do Projeto Modular

4.1.1 Preparação da entrada do fluxo

Antes da execução do Fluxo de Projeto Modular, é necessário definir os módulos *top* e os módulos que constituem o projeto. O módulo *top* (módulo de maior nível hierárquico) é apenas uma camada que une os módulos hierarquicamente inferiores. Os módulos são apenas instanciados no módulo *top* e conectados diretamente ou usando *bus macros*.

Bus macro é um componente (macro) pré-definido e pré-roteado, fornecido pelo fabricante, constituído por oito *buffers tristate* para assegurar a comunicação de 4 bits entre módulos fixos/reconfiguráveis e reconfiguráveis/reconfiguráveis possuindo a estrutura descrita na Figura 4.2. A comunicação que a bus macro provê é ponto-a-ponto. O principal objetivo da

bus macro é prover uma forma de controlar a interface de comunicação entre dois módulos, a partir de qualquer um dos módulos.

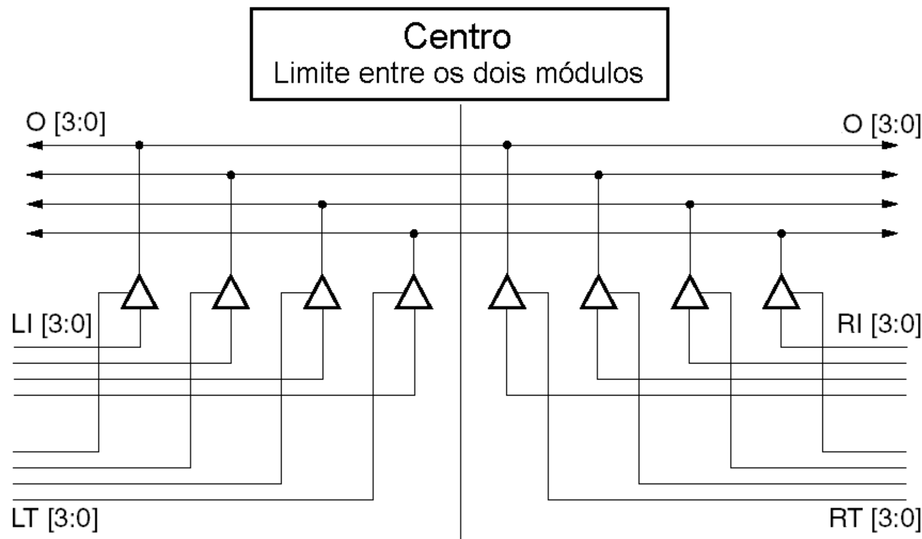

Figura 4.2: Esquemático de uma bus macro, formada por 8 tristates. Os sinais LT e RT são responsáveis pelo controle de acesso aos fios compartilhados pelos módulos, denominados O[3:0]. Um dos conjuntos de sinais LI e RI pode ser usado a cada instante como fonte de informação a enviar para o módulo do outro lado da bus macro. Cada *bus macro* provê um barramento de comunicação de 4 bits entre os módulos.

De acordo com a Figura 4.2, uma *bus macro* tem 16 sinais de entrada e 4 sinais de saída. Oito dos sinais de entrada controlam o componente. Os demais são sinais que transportam informação entre os módulos. A *bus macro* é dividida em dois grupos de sinais: sinais do lado esquerdo e do lado direito. Quatro pinos de controle acionam quatro *tristates* em uma *bus macro*. Todos os *tristates* são ativados em nível lógico '0'. Se um *tristate* do lado esquerdo da *bus macro* for acionado, obrigatoriamente o *tristate* do lado direito que está conectado na mesma linha de saída deverá ser desativado. Desta forma, evitam-se conflitos.

A Figura 4.3 ilustra uma conexão de dois módulos através de uma *bus macro*. O módulo situado à esquerda da *bus macro* possui dois sinais, usados na comunicação com o módulo da direita, denominados *Sinal1* (recebe dados de 1 bit do módulo da direita) e *Sinal2* (envia 1 bit para o módulo da direita). O *Sinal2* (comunicação no sentido esquerda-direita) do módulo da esquerda é conectado a um sinal chamado *Sinal2_vindo_modulo*. Este sinal é conectado a uma das entradas da *bus macro* situada à esquerda (LI). Nota-se que a entrada de controle (LT) referente ao sinal de entrada da *bus macro* recebe um valor de nível lógico baixo (GND) para habilitar incondicionalmente o *tristate* e possibilitar a passagem dos dados de um lado da *bus macro* para outro. Este sinal propaga-se pela saída (O) que é conectada ao sinal *Sinal2_vindo_BM*. Este por sua vez é ligado no módulo da direita pelo *Sinal2*. Por outro lado, o *Sinal1* (comunicação direita-esquerda) do módulo da direita é conectado ao sinal denominado *Sinal2_vindo_modulo*. Este sinal é conectado a uma das entradas da direita da

bus macro (RI). O sinal RT referente à entrada (RI) é posto no nível lógico zero (GND). Então, o sinal de entrada da *bus macro* é propagado para a saída da mesma (O) que é conectada ao sinal *Sinal1_vindo_BM*. Este sinal, por sua vez, é ligado no módulo da direita pelo *Sinal1*.

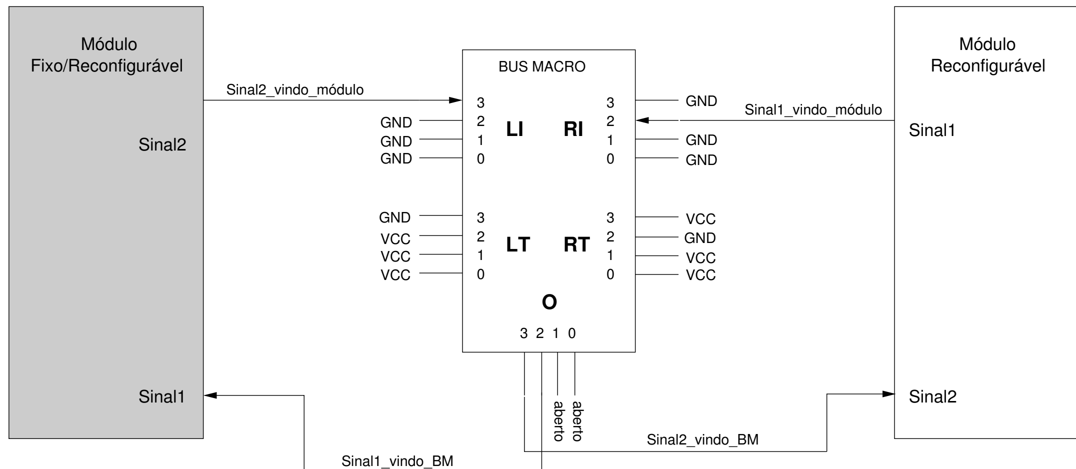

Figura 4.3: Comunicação de dois módulos (fixos/reconfigurável e reconfigurável) através de uma *bus macro*.

As instâncias das *bus macros* devem ser declaradas no módulo *top* para conexão dos módulos que compõe o projeto. Também deve-se usar atributos específicos da ferramenta de síntese usada para preservar sinais não utilizados das *bus macros* após a síntese. *Buffers* e circuitos de sincronismo de relógio (DCM) devem ser instanciados no módulo *top* para redução do escorregamento de relógio, e também para que nos próximos passos do Projeto Modular a ferramenta de roteamento consiga fixar tal sinal no FPGA. Finalmente, artifícios para criar sinais VCC e GND dentro do FPGA através de LUTs instanciadas para fornecimento de energia para as *bus macros* são necessários, pois não deve haver desperdício dos pinos de entrada e saída do FPGA. Recomenda-se que os pinos de entrada e saída do FPGA não sejam utilizados para alimentação das *bus macros* [LIM03].

De acordo com a XAPP290 [LIM03], se um módulo está localizado em uma área onde não existe um determinado recurso do FPGA (pinos E/S, reset, etc), e se este necessita conectar-se aos recursos localizados em outra área definida para fixar outro módulo, deve-se utilizar bus macros. As Figuras 4.4 e 4.5 mostram uma situação que exige bus macro para comunicação de pinos E/S com os módulos. As Figuras 4.7 e 4.6 ilustram uma situação que exige bus macro para comunicação de pinos de módulos internos [BRI03].

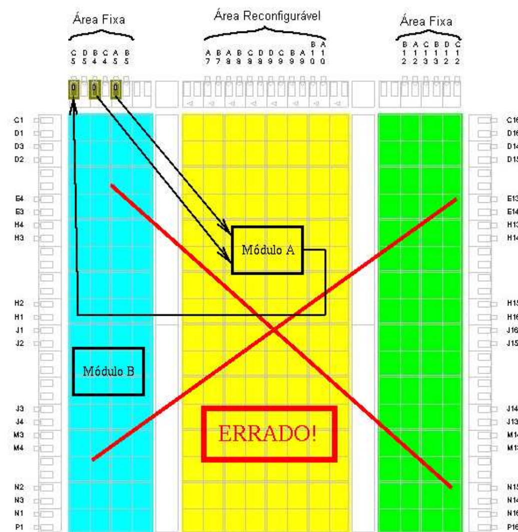

Figura 4.4: Situação em que não é possível conectar pinos de E/S no módulo A (reconfigurável). Os pinos de E/S estão na área esquerda do FPGA. O módulo A se encontra à direita. Não é possível conectar diretamente os pinos de E/S do módulo A, pois as ferramentas do Projeto Modular não permitem qualquer conexão de uma área para outra diretamente, a menos que estas sejam áreas fixas.

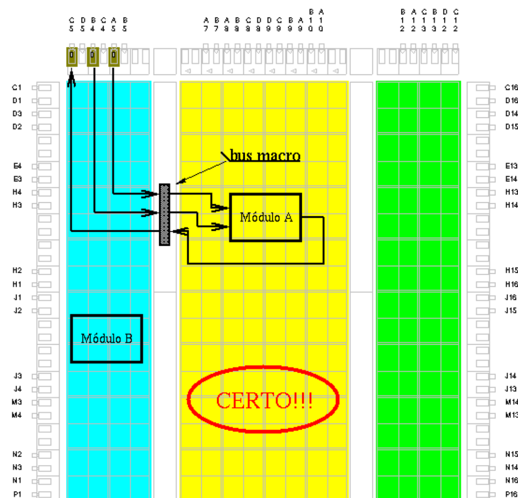

Figura 4.5: Conexão do pinos de E/S no módulo A através de bus macro. Todas as entradas e saídas do módulo são conectadas à bus macro. Esta se conecta aos pinos de E/S, provendo a comunicação.

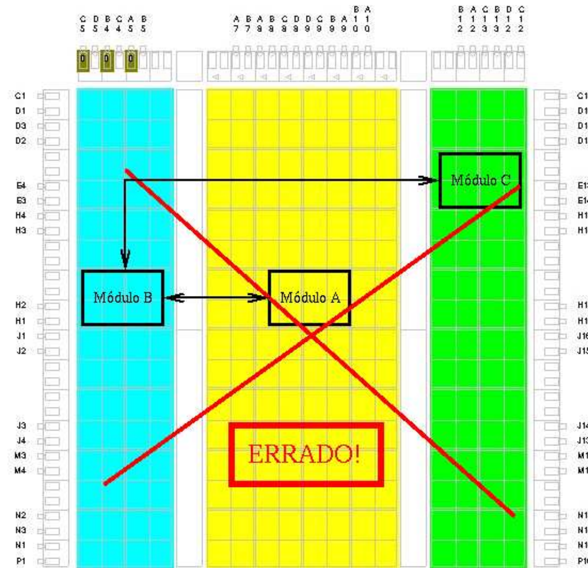

Figura 4.6: Situações particulares para implantação da bus macro na comunicação entre módulos Nesta situação, não é possível conectar os sinais do módulo A,B e C diretamente, pois os sinais atravessam a fronteira entre módulos fixos e reconfiguráveis. É necessário bus macro para interligar sinais de ambos os módulos (ligação B/A e B/C).

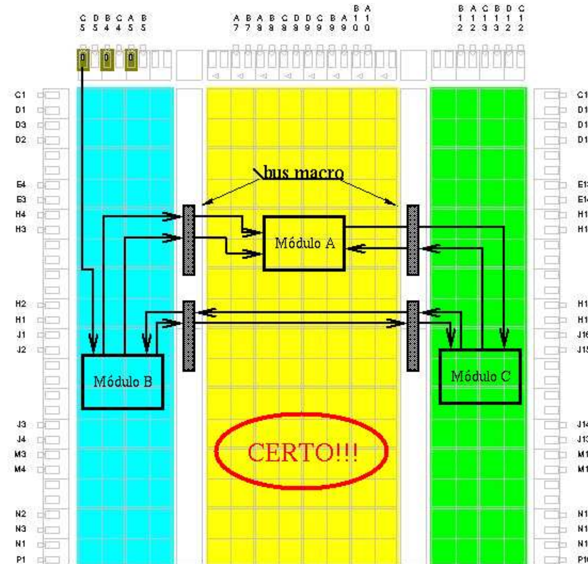

Figura 4.7: Conexão entre os módulos B/A, A/C e B/C através de bus macro.

Para posicionar elementos arquiteturais do FPGA, os arquivos de restrições são editados, bem como as áreas onde estarão localizados os módulos. No desenvolvimento de SDRs usando este fluxo, deve-se posicionar LUTs, DCMs e bus macros. As últimas devem ser sempre posicionadas nos slices que estão localizados nas coordenadas horizontais múltiplas de 4 no dispositivo reconfigurável [LIM03]. O desrrespeito a esta restrição resulta em erros na

ferramenta de posicionamento e roteamento.

Depois da síntese dos módulos, deve-se organizar os arquivos de síntese gerados em diretórios, conforme recomendados na Figura 4.8 [LIM03] para a execução do Fluxo de Projeto Modular.Isto deve ser realizado para que as ferramentas não sobreescrevam dados entre fases de processo, o que pode causar problemas na execução do fluxo como um todo.

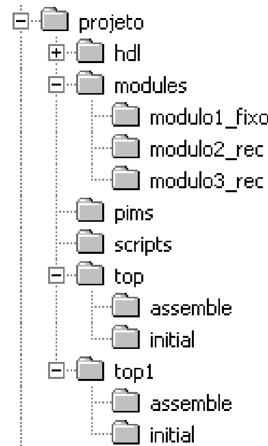

Figura 4.8: Estrutura de diretórios recomendada pela Xilinx para a execução do Fluxo de Projeto Modular.

Abaixo, é descrito cada diretório da Figura 4.8 recomendada pela Xilinx.

- *projeto*: diretório que mantém todos os arquivos necessário para a execução do fluxo;
- *hdl*: diretório onde se localizam os códigos fonte descritos em HDLs;
- *modules*: diretório onde estão contidos todos os módulos (fixos e reconfiguráveis) usado na fase da Implementação do Módulo Ativo;
- *top* e *top1*: diretórios que armazenam arquivos auxiliares do Fluxo de Projeto Modular durante as fases Orçamento Inicial(sub-diretório *initial*) e Montagem Final;
- *pims*: diretório usado para armazenar arquivos auxiliares provenientes da fase da Implementação do Módulo Ativo;
- *scripts*: diretório onde estão arquivos para execução de comandos para a ferramenta de síntese usada;

Depois que todos os módulos são sintetizados e organizados na estrutura de diretórios recomendada, então , é possível executar o Fluxo de Projeto Modular;

4.1.2 Ferramental para a execução do fluxo

A execução do Fluxo de Projeto Modular requer diversas ferramentas executadas de forma seqüencialmente ou paralelamente.

Entre as ferramentas usadas na execução do Projeto Modular disponibilizadas pela Xilinx [XIL01b], as principais são:

- *NgdBuild*:Esta ferramenta aceita como entrada arquivos EDIF, e como saída, gera arquivos de extensão NGO (arquivo intermediário que contém descrição lógica do projeto, preservando componentes originais e sua hierarquia interna), NGD (*Xilinx Netlist Generic Database*) e UCFs(*User Constraint File*). Opcionalmente, aceita como entrada arquivos de componentes pré-roteados (arquivos NMC (*Netlist Macros*)). Esta ferramenta converte um arquivo *netlist* de acordo com o modelo e fabricante do dispositivo reconfigurável, reduzindo todos os componentes em primitivas internas no formato da Xilinx, incluindo macros. A ferramenta acessa também bibliotecas apropriadas para cada componente e impondo restrições de acordo com o arquivo UCF. São feitos também testes lógicos na estrutura dos componentes neste formato, e finalmente o arquivo NGD é escrito. Esta ferramenta é executada em diferentes formas em cada etapa do Projeto Modular;

- *Floorplanner*: Ferramenta que aceita como entrada um arquivo NGD para definição da planta-baixa, agrupando módulos por área e fixando em um determinado local a partir de comandos do usuário os componentes primitivos da Xilinx;

- *Map*: programa que mapeia uma descrição lógica, tais como blocos lógicos e componentes. Remove também lógica não usada, associa os PADs nos seus blocos lógicos de entrada e saída (I/O) associados, mapeia a lógica dentro dos componentes (CLB, IOBs, etc). Esta ferramenta aceita como entrada arquivos NMC e NGD e como saída gera arquivo de restrições físicas de extensão PCF (*Physical Constraints File*) e também gera um arquivo NCD (*Native Circuit Description*). Este arquivo é uma representação física do projeto mapeado em termos de componentes fornecidos pelas bibliotecas da Xilinx. É gerado também um arquivo de extensão NGM, o qual contém informação do projeto físico produzido pelo MAP;

- *PAR (Place and Route)*: o arquivo NCD criado pelo MAP é a entrada do PAR, junto com as restrições físicas (PCF). PAR realiza o roteamento e o posicionamento das linhas de conexão utilizando dois critérios básicos. O primeiro deles é baseado no *custo*. São utilizadas várias tabelas de pesos e custos para estimar comprimento de conexões, restrições impostas pelo PCF e recursos de roteamento disponíveis. Outro critério é a temporização. O roteamento e posicionamento são feitos seguindo a análise de restrições temporais. O arquivo gerado pelo PAR é um NCD roteado e posicionado, pronto para ser utilizado pelo gerador de bitstreams;

- *FPGA Editor*: Ferramenta opcional utilizada para visualizar os componentes e o roteamento dentro do FPGA. Esta ferramenta permite também editar conteúdo dentro de LUTs e modificar o roteamento;

- *TRCE (Timing Reporter and Circuit Evaluator)*: Gera relatórios utilizando como entrada o arquivo de restrições físicas e um NCD roteado e posicionado. Estes relatórios contêm dados sobre verificação de temporização, usando restrições de tempo, e finalmente verificação de violações de posicionamento;

- *PIMCreate (*Physically Implemented Modules Creator*)*: PIMs são projetos já roteados e posicionados. *PIMCreator* automaticamente copia os arquivos NGO, NGM e NGD para diretórios apropriados para a execução do Projeto Modular, na preparação na fase de Montagem Final. O utilitário também renomeia arquivos de acordo com necessidades do Projeto Modular;

- *BitGen*: Produz Bitstreams para configuração de um dispositivo de configuração da Xilinx. A entrada desta ferramenta é um arquivo NCD roteado e posicionado gerado pelo PAR. Este tipo de arquivo contém todas as informações de configuração, que definem toda a lógica interna e interconexões, num dispositivo configurável específico. O bitstream pode, através de uma ferramenta de download, ser enviado para o FPGA para configuração. bitstreams gerados podem ser totais ou parciais;

4.1.3 Fase Orçamento Inicial

Como já foi mencionado anteriormente, determina-se nesta fase o arquivo *top* do projeto, bem como atribuições de restrições de área e temporização para cada módulo. Cada módulo reconfigurável do projeto será encapsulado por um módulo *top*. A fase do Orçamento Inicial é aplicada em todos os módulos *top*. Nesta fase são feitos: (a) definição de pinagem e restrições de tempo; (b) posicionamento de elementos para redução de atraso de clock, *bus macros*, e fontes de alimentação das mesmas; (c) posicionamento dos módulos no FPGA (definição da planta-baixa). A Figura 4.9 apresenta o fluxo de execução da fase de Orçamento Inicial.

A ferramenta NGDBuild é executada para gerar os arquivos necessários para as demais fases do Projeto Modular, posicionando toda a lógica do módulo *top* e agregando blocos que ainda não foram expandidos e que representam módulos. A entrada desta ferramenta são os arquivos EDIF já sintetizados, o arquivo que contém a *bus macro* pré-roteada (NMC) e o arquivo de restrições que é gerado inicialmente pelo usuário. Este arquivo é depois editado alternativamente pela ferramenta *Floorplanner*. Arquivos NGD e NGO são gerados, porém apenas o último será usado como entrada da próxima fase. A ferramenta *Floorplanner* é executada nesta fase para definição da planta-baixa, fixando os locais onde os módulos serão roteados e posicionados nas próximas fases do Projeto Modular. Os demais componentes (LUTs, *bus macros*, etc) também são fixados através desta ferramenta.

A Figura 4.10 apresenta a interface gráfica da ferramenta *Floorplanner* para edição e definição dos locais no FPGA onde serão fixado os módulos. Esta ferramenta poderá ser reexecutada tantas vezes quantas forem necessárias para obtenção de melhores resultados, o

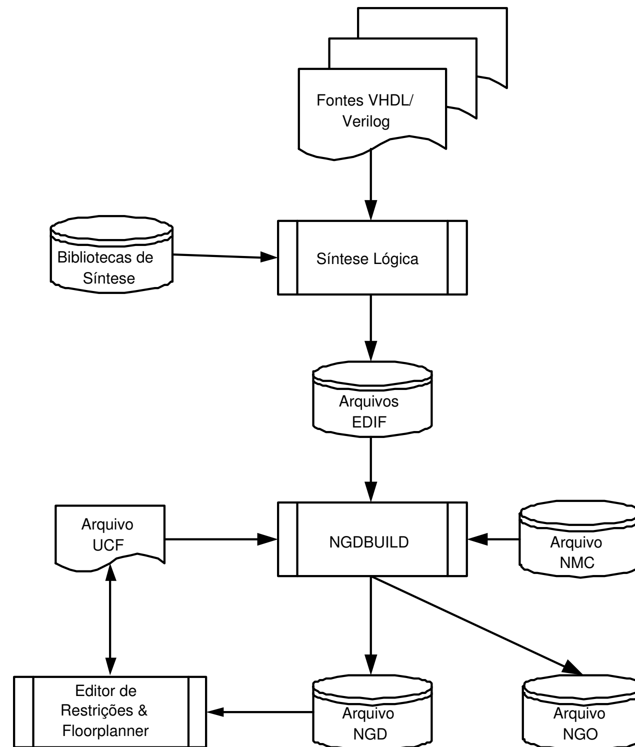

Figura 4.9: Fluxo de execução da fase de Orçamento Inicial do Projeto Modular.

que é ilustrado na Figura 4.9 pelo ciclo entre NGDBuild e Floorplanner.

4.1.4 Fase Implementação do Módulo Ativo

Na fase da Implementação do Módulo Ativo, como a própria nomenclatura define, ocorre o posicionamento, mapeamento, roteamento e a geração dos bitstreams parciais que representam cada módulo (fixo ou reconfigurável) do projeto. Os bitstreams parciais gerados a partir de cada módulo reconfigurável podem ser configurados individualmente no FPGA depois da execução do Fluxo de Projeto Modular.

O fluxo desta fase é apresentada na Figura 4.11. As entradas deste fluxo são as mesmas utilizadas pela fase do Orçamento Inicial. Entretanto, usa-se mais um arquivo de entrada gerado na fase do Orçamento Inicial com extensão NGO. NGDBuild gera um arquivo que será usado como entrada na ferramenta de mapeamento (MAP). Esta gera restrições físicas e um arquivo NCD não roteado, que serão usados na ferramenta de posicionamento e roteamento (PAR). Finalmente, são gerados os bitstreams parciais e uma estrutura de diretórios chamada PIM (*Physically Implemented Modules*), onde serão armazenados arquivos necessários para a execução da terceira e última fase do Projeto Modular.

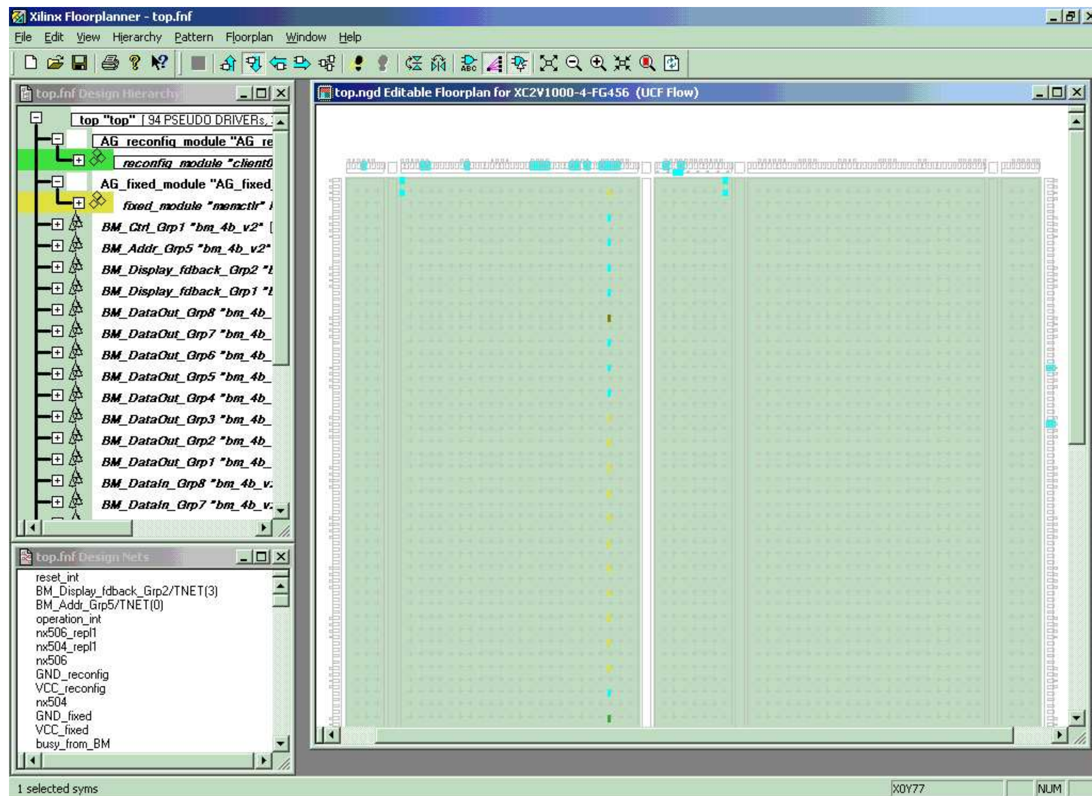

Figura 4.10: Ferramenta floorplanner. Usada para edição de restrições para roteamento, posicionamento de elementos lógicos e definição da planta-baixa do FPGA.

4.1.5 Fase Montagem Final

Finalmente, na fase Montagem Final, todos os módulos implementados fisicamente que fazem parte do módulo *top* são agregados ao mesmo. O posicionamento, mapeamento e roteamento são feitos de maneira unificada, anexando todos os módulos em um único projeto. O bitstream total do projeto final é criado nesta fase, pronto para ser prototipado. O fluxo de execução desta fase é ilustrada na Figura 4.12.

De acordo com a Figura 4.12, a ferramenta NGDBuild lê o arquivo EDIF do módulo *top*, o arquivo de restrições, o arquivo onde estão contidas informações para implementar fisicamente a *bus macro* e todo o conjunto de arquivos PIM gerados na fase anterior. O restante deste fluxo da fase Montagem Final é semelhante ao fluxo da fase anterior, exceto pelo fato da ausência da execução da ferramenta PIMCreator para criar a estrutura de diretórios PIM. O bitstream total gerado nesta fase é o produto final é já pode ser configurado no FPGA.

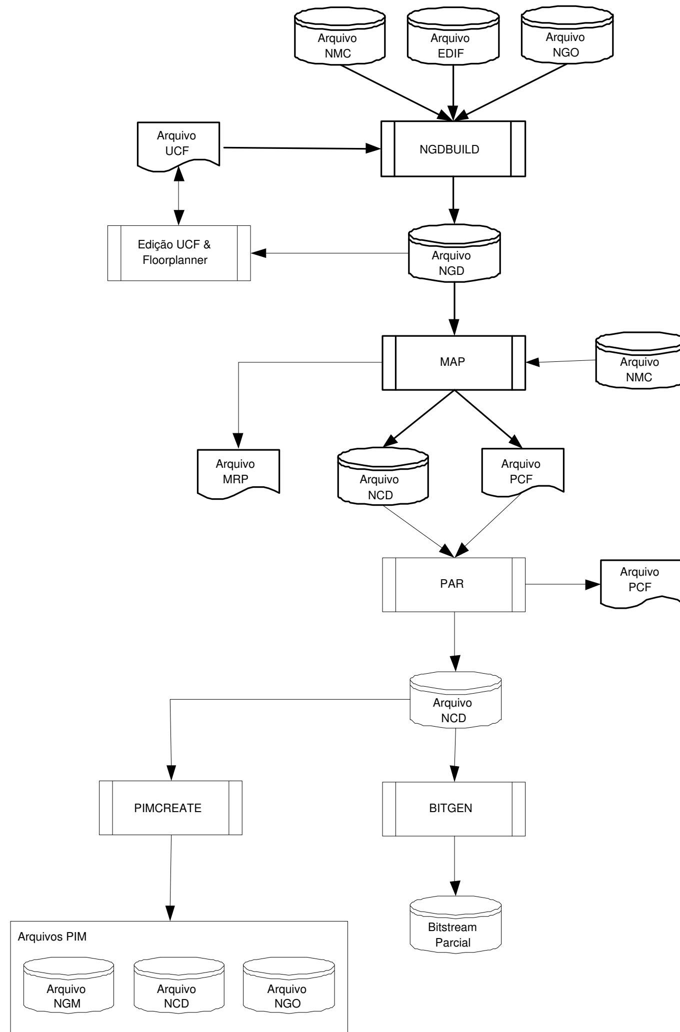

Figura 4.11: Fluxo de execução da fase Implementação do Módulo Ativo do Projeto Modular.

4.2 Contribuições

Nos documentos providos pela Xilinx ([LIM03] e *Development System Reference Guide*) [XIL01b], existem passos que são omitidos. A síntese é incompletamente descrita na documentação. Os arquivos de restrições de usuário (UCF) são fornecidos com pouquíssimas informações sobre como restringir o projeto. A ferramenta *Floorplanner*, essencial para particionamento dos módulos no FPGA é pouco detalhada na documentação disponível. A documentação não explica como são aplicadas e posicionadas as bus macros no projeto. Muitos procedimentos foram criados a partir de extensa pesquisa, interação com o setor de suporte da Xilinx e procedimentos exaustivos de tentativa e erro.

A partir destas precariedades sobre a falta de informação referente à execução do fluxo nos manuais da Xilinx, foram adicionados passos, que são parte da contribuição deste trabalho:

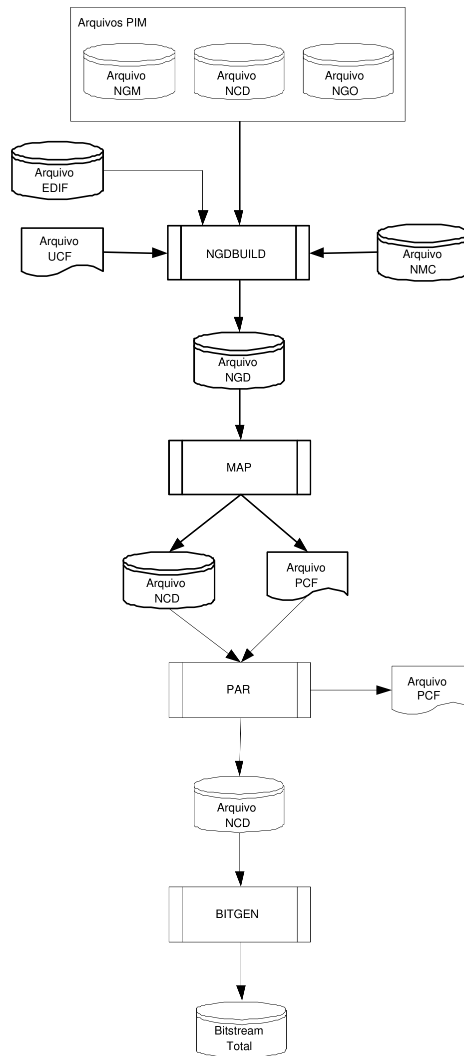

Figura 4.12: Fluxo de execução da fase Montagem Final do Projeto Modular.

- Síntese Lógica: foram adicionados alguns atributos de síntese lógica como restrições de sinais que não são usados (a síntese lógica removia tais sinais) e a não remoção de certos componentes que não eram usados, mas devem fazer parte do projeto. Foi realizada também a síntese lógica do arquivo *top* definindo os sub-módulos que compõe este arquivo como caixas-preta. Esta definição permite a geração de um arquivo *netlist* do *top* apenas com as conectividades com os sub-módulos, sem a lógica dos mesmos anexada ao arquivo. Esta é uma restrição imposta pela execução do fluxo precariamente detalhada nos manuais;

- Inclusão de *tags* no arquivo de restrição para permitir a reconfiguração parcial usando a ferramenta *Floorplanner*;

- Adição de Componentes para Geração de VCC e GND: usados para alimentação das bus macros. Ao invés de usar recursos externos do FPGA como os pinos de I/O para alimentação das bus macros, foram usadas LUTs para a alimentação das mesmas. Aqui foram adicionados comandos na síntese lógica para instância das LUTs e também foram incluídas no arquivo de restrições *tags* para posicionamento das LUTs no FPGA;

- Inserção de variáveis de ambiente para solucionar erros na ferramenta de mapeamento: foram usados variáveis de ambiente para ajustar algumas opções na ferramenta para permissão do uso de certas características no projeto reconfigurável;

- Adição de DCMs: Adição de componentes de gerenciamento de relógio para diminuir o escorregamento de relógio e permitir o funcionamento de circuitos seqüenciais reconfiguráveis;

- Adição de LUTs para Alimentação do DCM: foram usadas LUT para alimentação do DCM para evitar erros de roteamento. A Subsseção 4.3 abrange este assunto;

- Posicionamento de componentes em locais estratégicos do dispositivo reconfigurável que são conectados ao DCM para evitar erros de sincronização;

- A execução das ferramentas mostradas nos manuais da Xilinx não funcionavam em certos estudos de caso. O fluxo foi ajustado para executar qualquer estudo de caso, independente de suas características inerentes;

A informação contida neste Capítulo é um resumo de documentação bem mais extensa contida em relatório de pesquisa publicado [BRI03].

4.3 Crítica do Processo

Apesar do Fluxo de Projeto Modular poder ser usado para conceber módulos reconfiguráveis sem a manipulação detalhada de informações físicas do projeto, o fluxo mostra-se pouco automatizado na execução de suas fases. A falta de automatização pode resultar em inúmeros erros cometidos ao longo da execução do fluxo como um todo.

Além disso, o número de sinais que atravessa as bus macros é determinado pelo número de tristates em uma coluna inteira do FPGA. O número de bits dos sinais que atravessam as bus macros deve ser menor ou igual ao número de tristates numa dada coluna do FPGA. Este fator é limitante para a concepção de projetos integrados que são desenvolvidos através deste fluxo. Também, hardware adicional deve ser inserido no projeto, como por exemplo DCMs (*Digital Clock Manager*) para diminuir escorregamento do relógio.

Na fase de Montagem Final, encontrou-se problemas quanto ao roteamento e posicionamento de sinais gerados pela síntese (*Global Logic*) oriundos de componentes geradores de

alimentação (VCCs) que interconectam pinos do DCM. A ferramenta de síntese também infere outros componentes que devem ser conectados no DCM. Porém, esta conexão nem sempre foi bem-sucedida usando a ferramenta de posicionamento e roteamento. O roteamento não é determinístico e quando o fluxo foi executado, ora a ferramenta de posicionamento e roteamento conseguia rotear estes sinais, ora não conseguia, gerando uma mensagem de erro. A solução encontrada foi instanciar explicitamente no código todos os componentes e conectá-los através da descrição no código HDL. Instanciou-se e fixou-se LUTs geradoras de constantes para conectá-las aos pinos do DCM. O Apêndice B.1 detalha este procedimento. Desta maneira, a ferramenta de posicionamento e roteamento consegue ser executado sem erros de roteamento e geração de *Global Logic*.

Outro problema que surgiu foram erros na geração da rede de relógio. O projeto prototipado na placa podia ou não funcionar de acordo com o não determinismo da ferramenta de posicionamento e roteamento. Isto aconteceu quando um módulo fixo foi agrupado no canto esquerdo do dispositivo. Os DCMs usam *buffers* para realimentação para que seja possível a diminuição do escorregamento de relógio. O DCM que foi conectado ao módulo fixo estava localizado no canto esquerdo do FPGA. Porém tais *buffers* ficam posicionados no centro do dispositivo onde se encontrava a área reconfigurável escolhida. O sinal que ligava os dois componentes passava entre uma área fixa e uma reconfigurável, e erros aconteciam no momento da reconfiguração parcial. Então para solucionar este problema, foi instanciada uma bus macro para prover comunicação entre a área fixa e reconfigurável, como mostrado no Apêndice B.2. Desta forma o projeto funcionou corretamente no FPGA.

Finalmente, quando se desenvolve um projeto utilizando mais de uma área reconfigurável, a complexidade do projeto aumenta quanto à aplicação de bus macros e uso de LUTs para alimentar as bus macros e pinos do DCM. Uma solução para o desenvolvimento de projetos com esta característica seria posicionar vários componentes do projeto em posições estratégicas no FPGA de acordo com a complexidade do mesmo e também do número de áreas reconfiguráveis. Isto minimiza a quantidade de bus macros e LUTs. Porém ainda não é o suficiente para ter uma redução de complexidade do projeto dentro do FPGA de maneira significativa.

Quanto a projetos de duas ou mais áreas reconfiguráveis, devem-se utilizar artifícios (multiplexadores) para interligar fios que passam em suas áreas reconfiguráveis, pois sem estes artifícios, problemas de condução do sinal aconteciam. Isto acarretava em problemas no sistema reconfigurável implementado em duas ou mais áreas reconfiguráveis.

4.4 Proposta de Ferramental

Como foi mencionado anteriormente, o Projeto Modular permite o desenvolvimento de SDRs e de módulos parcialmente reconfiguráveis de maneira independente e simultânea com outros módulos [XIL01b]. No entanto, o desenvolvimento de tais sistemas, usando o Fluxo de Projeto Modular exige grande esforço, desde a codificação do projeto até a fase de

implementação, o que torna o projeto propenso a erros.

Observando o padrão do Fluxo de Projeto Modular, suas entradas e estruturas de diretório, foi desenvolvida a ferramenta apresentada na Figura 4.13, para automatizar o fluxo, da codificação até a geração dos bitstreams parciais e totais. A ferramenta denominada *MDLauncher* (*Modular Design Launcher*), é capaz de executar praticamente todos os passos do fluxo. O usuário apenas informa quais módulos são reconfiguráveis, e disponibiliza os arquivos de restrição para o usuário (UCF) com a planta-baixa definida. *MDLauncher* gera *scripts* necessários para a criação de estrutura de diretórios, síntese e implementação que serão executados pela mesma. Dentro da ferramenta, pode-se escolher a ferramenta de síntese lógica (*Leonardo Spectrum*, XST, FPGA *Express*, etc). *MDLauncher* foi desenvolvida usando linguagem de programação JAVA.

Esta ferramenta é importante para o desenvolvimento de SDRs pelo fato de automatizar e eliminar o número de erros ocorridos na execução do fluxo.

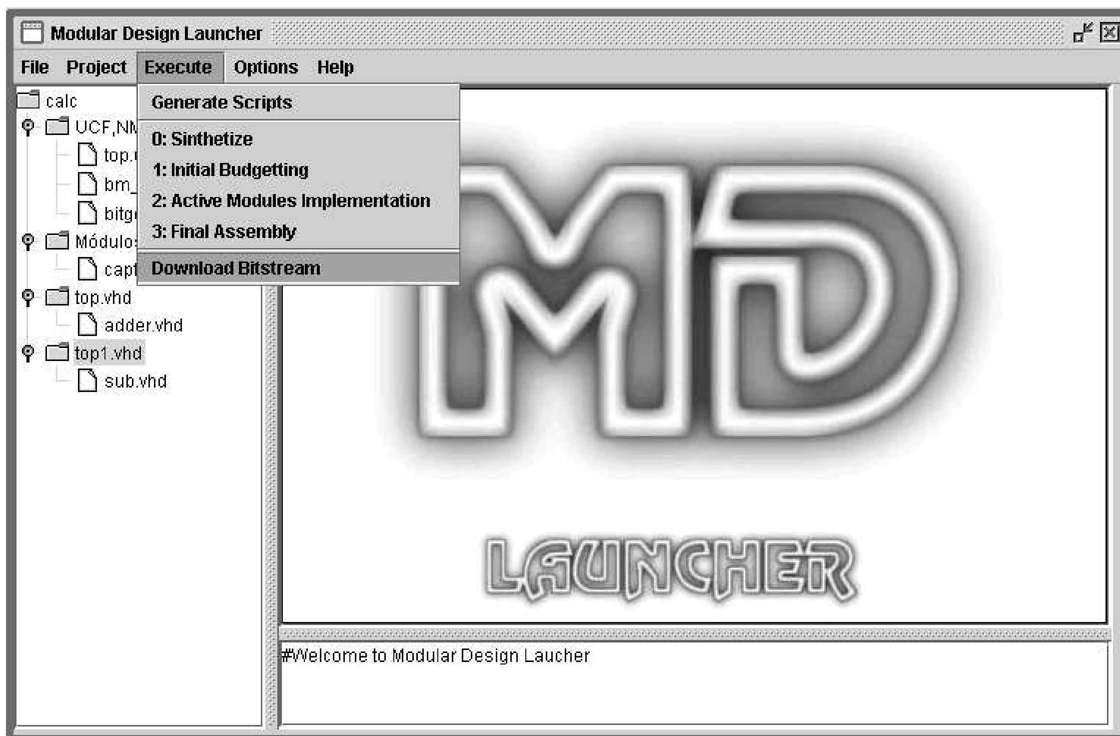

Figura 4.13: Interface gráfica da ferramenta *MDLaucher*. Esta ferramenta automatiza parcialmente a execução do Fluxo de Projeto Modular. O usuário disponibiliza os fontes em linguagem de descrição de hardware, arquivos de restrições e parâmetros armazenados em um arquivo texto para geração dos arquivos de configuração parciais e totais.

Nesta ferramenta, é possível executar o fluxo para qualquer modelo da família Virtex II. Esta ferramenta foi desenvolvida para possibilitar a adição de outras ferramentas de síntese lógica como mencionado anteriormente. Porém o nível de dependência tecnológica é alta, ou seja, esta ferramenta só pode ter utilidade para FPGAs da família Virtex II.

Existem passos que devem ser seguidos para a execução da ferramenta. Estes são apresentados a seguir:

1. *Definição dos arquivos de configuração*: primeiramente cria-se um projeto novo, e definem-se quais os arquivos de configuração a serem inseridos no projeto, tais como arquivo de bus macro, arquivo UCF com todas as restrições de posicionamento e área definidas e um arquivo de configuração para geração do bitstream total ou parcial;

2. *Definição dos arquivos-fonte*: o usuário deverá inserir os arquivos de maior nível hierárquico (*top*, e os arquivos que implementam cada módulo fixo e reconfigurável do sistema). Estes arquivos são apresentados no software como uma árvore de diretórios. A pasta *módulos fixos* armazenam os módulos de hardware comum a todos os arquivos *top*; A pasta *top.vhd* armazena o arquivo *top* que contém um módulo de hardware específico que está contido nesse. A pasta *top1.vhd* armazena um arquivo *top* que contém um outro módulo de hardware específico. Estes módulos específicos originarão no final da execução do fluxo os bitstreams parciais. No final deste processo, automaticamente é gerada a estrutura de diretórios recomendada pela Xilinx [LIM03] semelhante àquela mostrada na Figura 4.8;

3. *Generate Scripts - geração de scripts e árvore de diretório*: neste momento, os scripts indispensáveis serão criados para a execução da síntese lógica;

4. *Sinthetize - Síntese*: todos os arquivos fonte são sintetizados logicamente pela ferramenta de síntese escolhida. Atualmente, o programa suporta apenas a ferramenta *Leonardo Spectrum*. Nesta fase é gerado um arquivo EDIF para cada arquivo-fonte (módulos fixos, módulos reconfiguráveis e arquivos *top*);

5. *Initial Budgeting - Orçamento Inicial*: execução da primeira fase do Projeto Modular, como detalhado neste Capítulo. Gera arquivos intermediários com toda a estrutura do arquivo *top* descritas num formato da Xilinx com atribuições e restrições de temporização e área;

6. *Active Modules Implementation - Implementação do Módulo Ativo*: geração dos bitstreams parciais;

7. *Final Assembly - Montagem Final*: geração dos bitstreams totais e completos, pronto para serem prototipados;

Nesta ferramenta, há opções que podem ser configuradas como modelo do FPGA, diretório onde estão localizadas as ferramentas de síntese física, implementação e geração de bitstreams, diretório onde está localizada a ferramenta de síntese lógica e diretório onde serão armazenados todos os arquivos necessários para a execução do fluxo.

A ferramenta *MDLauncher* visa automatizar o fluxo de Projeto Modular para a geração automática de SDRs.

Nesta ferramenta foram validados todos os estudos de caso mostrados nos Capítulos 6 e 5. Gerou-se os bitstreams totais e parciais para estes estudos de caso e estes funcionaram perfeitamente na plataforma de prototipação. Porém se houver a re-síntese lógica e física do projeto para outra arquitetura alvo, deve-se reposicionar os módulos fixo e reconfigurável usando a ferramenta *Floorplanner* e deve-se reposicionar, quando for o caso, as bus macros, LUTs para fornecimento de energia das mesmas e DCMs.

Capítulo 5

Validação do Fluxo Modular de Projeto

Neste Capítulo, serão apresentados estudos de caso que validam o método proposto no Capítulo 4.

5.1 Descrição da Plataforma de Reconfiguração Alvo

A plataforma V2MB1000 da empresa Memec-Insight utiliza um FPGA de um milhão de portas lógicas (XC2V1000-4fg456C [XIL02a]) em um CI de 456 pinos. Além disso, a plataforma oferece também uma memória DDR (*Double Data Rate*) de 16M x 16 bits, uma PROM XC18V04 de 4MBits, dois geradores de relógio (um de 24MHz e outro de 100MHz), uma porta RS-232 e conector de 160 pinos para permitir acrescentar módulos especiais. Junto à plataforma, existe uma interface LVDS (*Low-Voltage Differential Signaling*) que provê transmissão de dados de 16 bits em alta velocidade. A Figura 5.1 mostra um diagrama de blocos de alto nível da plataforma V2MB1000.

A plataforma dispõe de dois módulos externos acopláveis ao conector de 160 pinos, um módulo para desenvolvimento de projetos de comunicação e um extensor de pinagem. A plataforma ainda dispõe de suporte para configuração e teste via cabo JTAG e MultiLINX e programação dos modos de configuração via chaves de programação na placa (*jumpers*). Para finalizar, um microprocessador *soft-core* é integrado à plataforma para desenvolvimento de projetos relativamente complexos.

Todos os estudos de caso descritos ao longo deste Capítulo foram prototipados na plataforma V2MB1000.

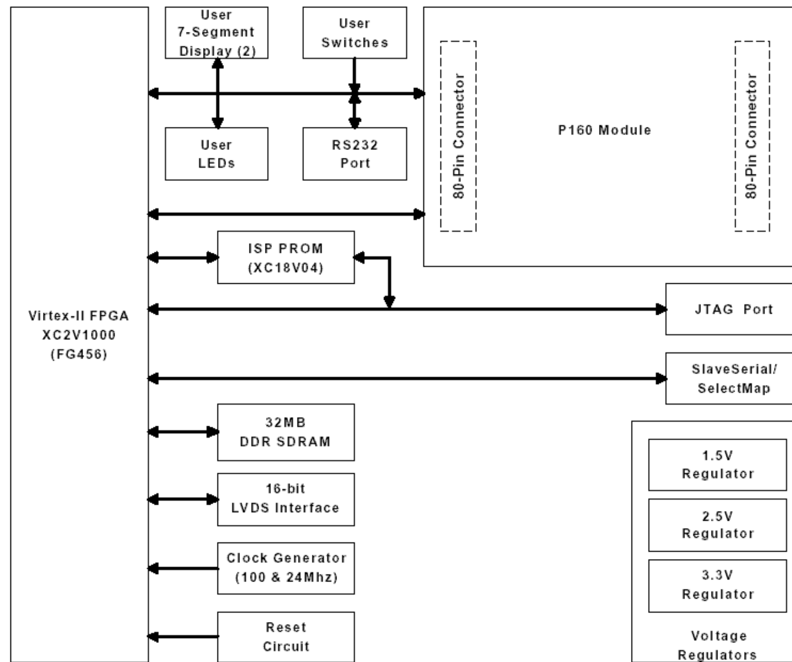

Figura 5.1: Diagrama de blocos da plataforma V2MB1000.

5.2 Estudos de Caso

Para validação do fluxo do Projeto Modular no contexto da reconfiguração parcial e dinâmica, foram desenvolvidos os seguintes estudos de caso abaixo:

- **Calculadora reconfigurável**: como o próprio nome diz, esta é uma calculadora de duas operações (subtração/adição), onde cada operação é representada por um módulo (bitstream) reconfigurável;

- **Contador reconfigurável**: este é um contador que pode assumir duas configurações: um contador de 1 Hz e um contador de 4 Hz de freqüência de operação;

- **Contador OCP reconfigurável**: foi implementado o mesmo contador descrito acima, porém utilizando OCP como protocolo de comunicação entre os módulos;

- **Contador com duas áreas reconfiguráveis**: foram implementados dois contadores: um contando em ordem crescente e outro em ordem decrescente. Adicionalmente, estes podem ser configurados em duas áreas reconfiguráveis;

- **Controlador de Memória SRAM**: permite a mais de uma configuração de um módulo cliente enviar dados para memória SRAM (*Static Ram*) da plataforma V2MB1000;

- **Sistema R8 Reconfigurável**: é um estudo de caso maior, dividido em dois módulos: processador R8 e coprocessador reconfigurável;

Neste Capítulo serão descritos e analisados cada um dos estudos de caso acima, com excessão do último, que é discutido no Capítulo 6.

5.2.1 Calculadora reconfigurável

Este estudo de caso consiste em uma calculadora de uma operação que pode ser configurada em duas funcionalidades: adição e subtração.

Este projeto consiste em dois módulos: módulo de captura dos dados (*capture*) situado na área fixa (área que não ocorre a reconfiguração) do dispositivo e um módulo que fica situado na área reconfigurável (módulos de adição e subtração denominados de *adder* e *sub* respectivamente) conforme a Figura 5.2.

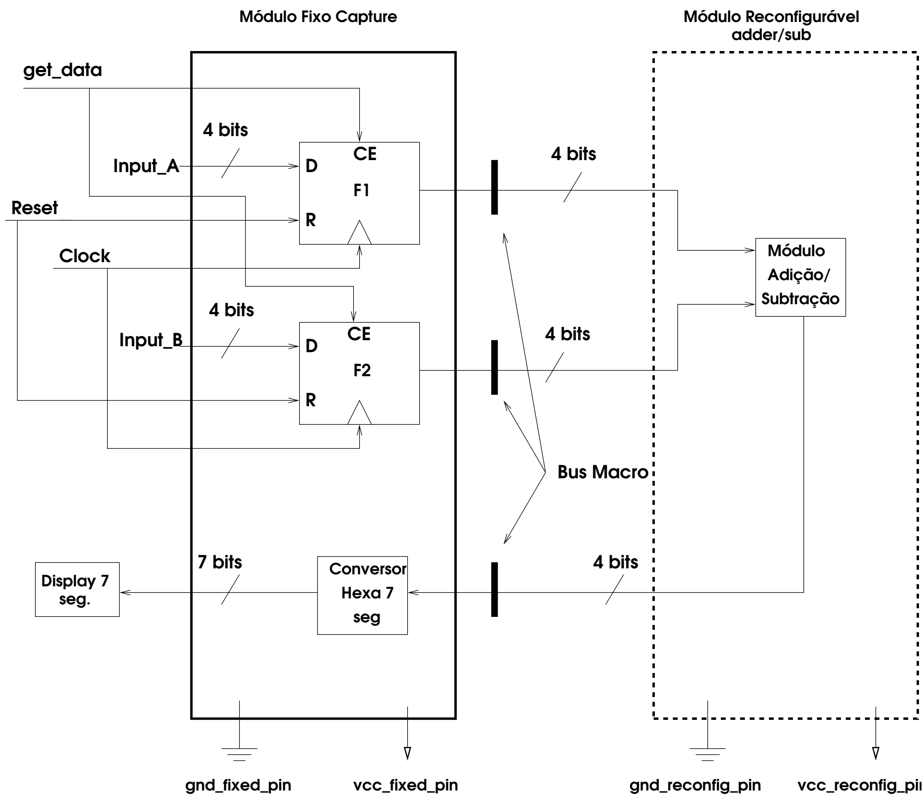

Figura 5.2: Bloco lógico de uma calculadora reconfigurável de duas operações.

O módulo de captura de dados é responsável por adquirir dados representados nas chaves *dip-switch* da plataforma e armazenar estes dados nos registradores denominados $F1$ e $F2$. A entrada *Input_ A* é definida pelas 4 primeiras chaves do FPGA e a entrada *Input_ B* é definida pelas outras 4 entradas restantes. A entrada *get_ data* é responsável pela aquisição dos dados das chaves para os registradores $F1$ e $F2$.

A área fixa do FPGA, onde está situado o módulo de captura, se interconecta através de três bus macros com a área reconfigurável, onde pode estar situado o módulo de adição ou de

subtração. Duas bus macros são responsáveis pela comunicação dos sinais de entrada que são a saída dos registradores. Os barramentos de entrada, ao todo formam 8 bits (2 bus macros). O módulo *adder/sub* recebe então os valores oriundos das saídas dos registradores $F1$ e $F2$ e opera sobre estes. O resultado então é transmitido por um barramento de saída de 4 bits (1 bus macro) conectado a um display da plataforma.

A calculadora é composta por dois módulos reconfiguráveis combinacionais (*adder* e *sub*). Esta calculadora foi prototipada no FPGA e foram feitos dois testes: primeiramente carrega-se o dispositivo reconfigurável com um *bitstream* inicial de soma e logo após carrega-se o dispositivo com um módulo reconfigurável de subtração. Depois foi feito outro teste utilizando um *bitstream* inicial de subtração e então carregou-se um módulo reconfigurável de soma. Quando uma determinada configuração das chaves foi fixada e quando o sinal *get_ data* ficou em nível lógico 0 pelo disparo manual de um botão da plataforma associado ao *get_ data*, o display mostrou o resultado da operação corretamente. Quando se reconfigurou o FPGA com outro bitstream parcial, o display foi modificado indicando o resultado correto referente a esta nova funcionalidade reconfigurada.

Para o posicionamento das bus macros, usou-se pinos de E/S para alimentar as mesmas. Pinos de alimentação da bus macro devem estar associados a uma área (fixa ou reconfigurável). Para cada fronteira entre módulos reconfiguráveis e fixos/reconfiguráveis, devem ser reservados dois pinos para alimentação das bus macros.

Neste projeto foi observado que qualquer sinal, inclusive sinais que comunicam pinos de E/S que atravessem a fronteira entre as áreas fixa e reconfigurável precisam obrigatoriamente de bus macros. Todos os pinos que são utilizados pelo módulo fixo foram incluídos na área fixas definida pela ferramenta *Floorplanner*.

Finalmente foi feito um experimento utilizando como *bitstream* inicial os módulos *capture* e *adder*. Depois de carregar o dispositivo reconfigurável com este *bitstream*, foi feita nova carga do dispositivo com um *bitstream* parcial que armazena o módulo *adder*. Observou-se o comportamento dos displays da plataforma de prototipação e notou-se que não houve absolutamente nenhuma mudança comportamental no circuito. Inicialmente foi feito a soma dos valores 6 e 4. O display da plataforma mostrou o valor 10. Quando foi feita reconfiguração do módulo de adição novamente, o valor mostrado no display não se alterou. Mesmo experimento foi feito utilizando um *bitstream* total com o módulo *sub*. Depois de carregá-lo no FPGA, foi feita a carga do *bitstream* parcial que armazena o módulo *sub*. Observando o comportamento dos displays, novamente não houve nenhum distúrbio de comportamento do circuito, nem antes, nem durante, nem depois da reconfiguração.

5.2.2 Contador reconfigurável

O circuito contador realiza uma contagem de 0 a 9 em uma determinada freqüência. Existe um módulo fixo que captura os dados de entrada que são passo (Δ) de contagem (contagem

pode ser $t_1 = 0 + \Delta, t_2 = t_1 + \Delta, t_3 = t_2 + \Delta$) e o módulo reconfigurável é responsável pela implementação do contador. A funcionalidade do módulo fixo é muito semelhante ao módulo utilizado pela calculadora reconfigurável. No entanto, foram criados dois módulos reconfiguráveis: um contador de freqüência de 1 Hertz e outro contador de freqüência de 4 Hertz. Foram feitas sucessivas configurações no FPGA e os resultados foram corretos. Alternou-se os dois módulos reconfiguráveis e os dois tiveram o comportamento desejado. Este estudo de caso utilizam dois módulos reconfiguráveis seqüenciais simples.

Para evitar erros no momento da reconfiguração parcial em circuitos síncronos, deve-se utilizar DCMs, como já comentado no Capítulo 4. O circuito não funciona mais depois da reconfiguração parcial em versões preliminares do projeto.

Na Figura 5.3, é mostrado um esquemático do circuito reconfigurável. Usa-se um registrador para armazenar o valor de entrada definido pelo pino *get_ data*, exatamente da mesma forma que no exemplo do circuito combinacional mostrado na Figura 5.2.

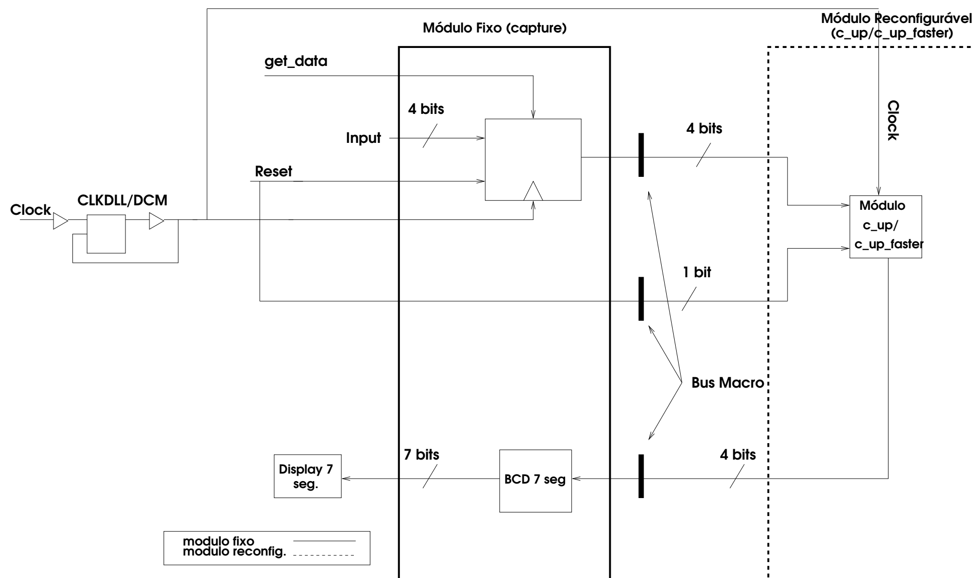

Figura 5.3: Diagrama de blocos de um contador reconfigurável.

5.2.3 Contador reconfigurável com interface OCP

O contador mencionado na Seção anterior foi implementado com o encapsulamento de protocolo de comunicação OCP entre o módulo fixo e o módulo reconfigurável. O módulo *capture*, por ser o inicializador do sistema, e por requisitar serviços para os módulos de contagem (*c_ up/c_ up_ faster*), foi encapsulado com um invólucro OCP mestre. Os módulos de contagem foram encapsulados com um invólucro OCP escravo. O protocolo foi validado através da ferramenta *CoreCreator*. Foi usado o protocolo de comunicação de sinais básicos do OCP como mostrado na Figura 3.7.

A Figura 5.4 mostra um diagrama de temporização do contador OCP.

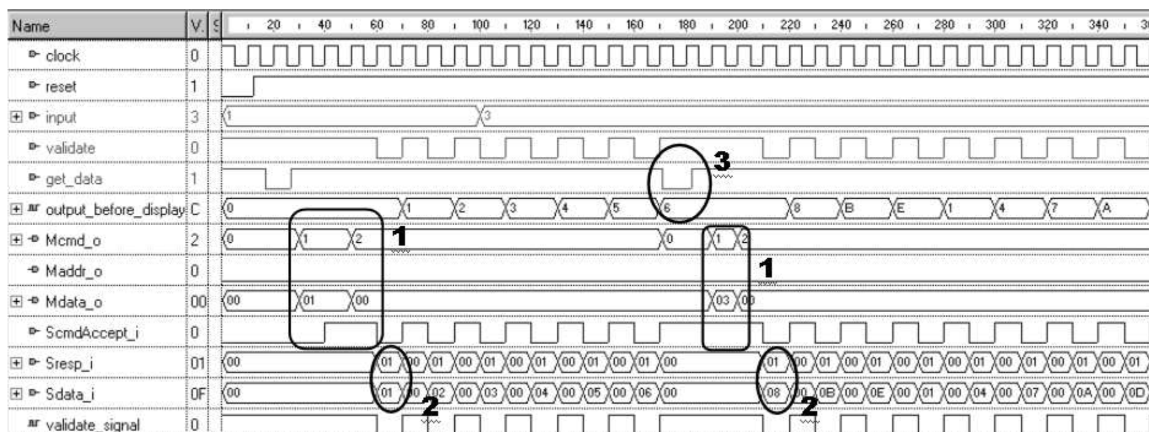

Figura 5.4: Diagramas de tempos do contador OCP.

De acordo com o protocolo de comunicação mostrado na Figura 3.7:

1. o módulo de captura (mestre) faz uma requisição ao módulo de contagem (escravo) apresentando os sinais *Mcmd* = WR (valor 01), *Maddr* e *Mdata*. Este último sinal representa o passo de contagem do sistema aqui apresentado. Um ciclo de relógio após, o escravo responde com o sinal *ScmdAccept* em nível lógico 1. O mestre então requisita uma leitura gerando o sinal *Mcmd* = RD (valor 02);

2. o módulo escravo passa a responder para o mestre enviando um sinal de resposta *Sresp* = DVA (valor 01) e o dado (*Sdata*) que é a saída do contador, a ser mostrado no display do FPGA;

3. quando as chaves são modificadas pelo usuário, o passo de contagem é atualizado quando o usuário dispara o sinal *get_ data* ativando um botão na plataforma de prototipação;

O circuito foi prototipado e o comportamento dos dois núcleos IP foi conforme o comportamento esperado. Foi feita a reconfiguração parcial e dinâmica usando núcleos IP com interfaces de comunicação padronizadas, usando protocolo OCP.

5.2.4 Contador com duas áreas reconfiguráveis

Neste circuito, existe um módulo de captura dos dados chamado de *capture* e foram implementados 2 módulos reconfiguráveis: um módulo denominado *contador_ up* que conta em ordem crescente e outro módulo chamado de *contador_ down* que realiza a contagem em ordem decrescente. Na plataforma existem um display de 7 segmentos de dois dígitos. Neste estudo de caso, cada dígito é associado a uma área reconfigurável. Através do fluxo do Projeto Modular, foram gerados 4 bitstreams parciais: *contador_ up* e *contador_ down*

para uma área reconfigurável e estes mesmos circuitos de contagem desenvolvidos para a outra área reconfigurável. Inicialmente os dois dígitos do display contam em ordem crescente. Quando faz a reconfiguração carregando o dispositivo com um *bitstream* parcial, dependendo da área a ser reconfigurada, o dígito do display associado a esta área mostra uma contagem decrescente. A mesma reconfiguração foi feita na outra área reconfigurável obtendo-se assim o comportamento desejado dos quatro bitstreams parciais. No desenvolvimento deste circuito, foi observada uma dificuldade no que diz respeito à delimitação das áreas do circuito. A ferramenta de mapeamento gerou erros quanto a estas delimitações. Estes erros foram sanados a partir de testes modificando a delimitação das áreas através da ferramenta *Floorplanner*. Este foi o primeiro estudo de caso usando duas áreas reconfiguráveis.

5.2.5 Controlador de memória SRAM

A partir da especificação da memória localizada no módulo de comunicação da plataforma V2MB1000, foi desenvolvido um controlador de memória dividido em dois módulos: um módulo reconfigurável chamado cliente, o qual recebe e envia dados de/para memória e o outro que implementa o protocolo de comunicação entre o cliente e a memória SRAM conforme ilustrado na Figura 5.5. Um módulo reconfigurável cliente escreve um determinado valor na memória. O outro módulo reconfigurável cliente escreve um outro dado na memória. Em nível de simulação e prototipação deste circuito estaticamente, os resultados foram obtidos com sucesso, validando funcionalmente o circuito. Porém com a inclusão das bus macros para a comunicação entre os módulos e depois da execução do fluxo do Projeto Modular, os bitstream totais e parciais gerados por este não funcionaram adequadamente no dispositivo reconfigurável. Por alguma razão indeterminada até o momento, as bus macros estão alterando a temporização do protocolo de comunicação entre os dois módulos (fixo e reconfigurável), tornando instável o acesso à memória SRAM. Foram realizados vários ajustes no fluxo sem sucesso. Contactou-se o suporte da Xilinx e espera-se resposta para a resolução de eventuais erros de temporização causadas pelas bus macros.

5.3 Análise do tempo de reconfiguração

5.3.1 Tempos de reconfiguração

De acordo com [BRI02], o tempo de reconfiguração total do FPGA *Virtex II* de 1 milhão de portas equivalentes da plataforma V2MB1000, no modo de configuração mais rápido medido (*Master SelectMAP*) leva em torno de 7.5 ms a uma freqüência de configuração de 60 MHz. Este arquivo foi inicialmente armazenado na PROM das plataforma e transferido para o FPGA via modo autônomo de configuração onde o FPGA comanda a sua configuração. O tempo de reconfiguração foi o tempo de transferência dos dados de configuração da PROM para o FPGA. A Figura 5.6 mostra os tempos de reconfiguração no modo de configuração

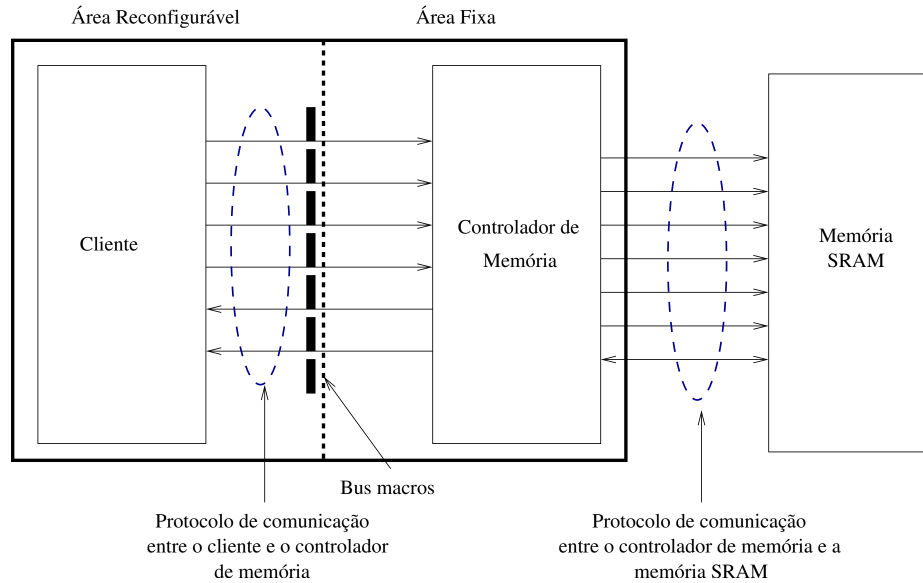

Figura 5.5: Diagrama de blocos de um controlador de memória SRAM conectada a um módulo cliente.

mais rápido do FPGA em função da freqüência de operação para transmitir os dados de configuração. Detalhes quanto à extração destes tempos de configuração são apresentadas em [BRI02].

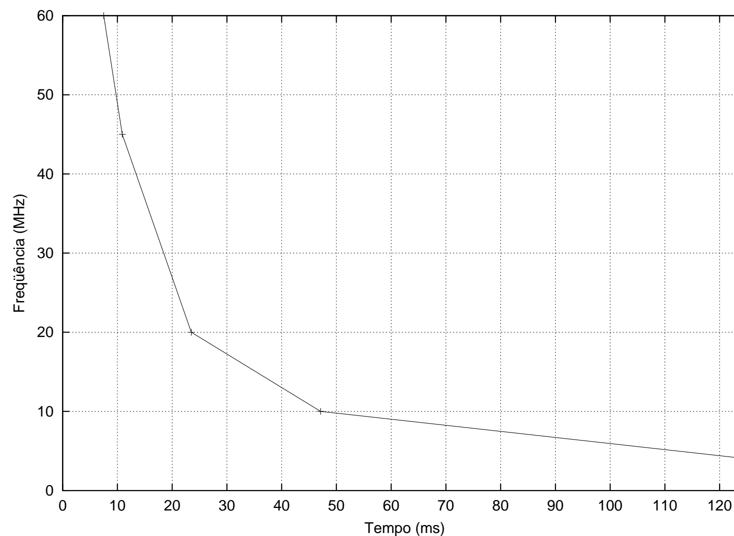

Figura 5.6: Gráfico das medidas dos tempos de reconfiguração do modo de configuração mais rápida suportada pelo FPGA Virtex II (tempo de reconfiguração versus freqüência para transmissão de dados).

O modo de configuração *Master SelectMAP* não pode ser usado para reconfiguração parcial e dinâmica. Porém, o tempo de reconfiguração neste modo de configuração tem a mesma ordem de grandeza que a configuração realizada usando o módulo ICAP (*Internal Confi-*

guration Access Port) do FPGA [ECK04]. Este componente permite o acesso à porta de configuração do dispositivo. Desenvolvendo um hardware para acessar esta porta de configuração, é possível realizar a configuração a uma freqüência máxima de 33MHz. Porém este componente não foi usado corretamente segundo razões descritas em [CAR04]. Através dos tempos na freqüência máxima de transmissão que foram extraídos para reconfiguração total para o modo de configuração *Master SelectMAP* (Figura 5.6), é possível fazer estimativas de tempo de reconfiguração parcial, sabendo que o tempo da mesma é proporcional ao tamanho do bitstream. A Figura 5.7 mostra a estimativa dos tempos de reconfiguração em função do tamanho do *bitstream*.

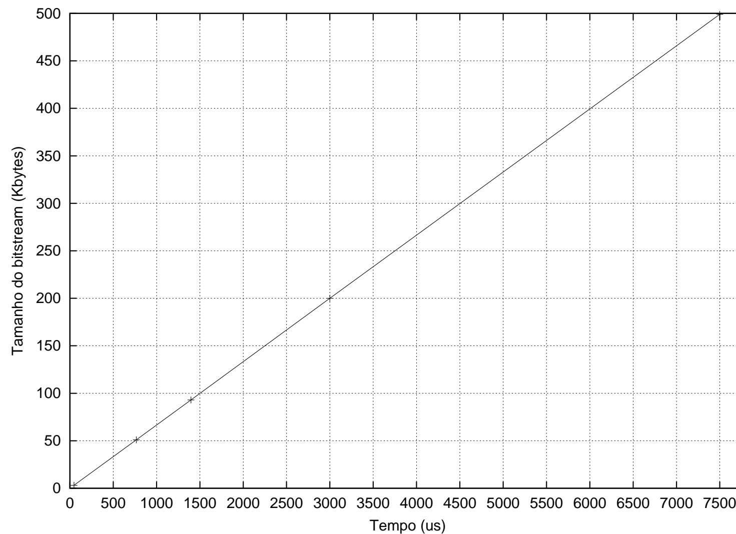

Figura 5.7: Gráfico da estimativa dos tempos de reconfiguração para o modo de configuração mais rápido suportado pelo FPGA Virtex II XC2V1000 (tempo de reconfiguração em microssegundos versus tamanho dos bitstreams em kilobytes).

Segundo [XIL02], existem apenas dois modos de configuração que permitem reconfigurar parcialmente o FPGA. Estes modos são denominados *Boundary-Scan* e *Slave SelectMAP*. Todos os estudos de caso foram reconfigurados parcialmente usando o modo *Slave SelectMAP* através do cabo USB (*Universal Serial Bus*), pois este é o mais rápido modo de configuração via cabo.

Através da implementação e execução dos estudos de caso descritos ao longo deste Capítulo, foram extraídos alguns tempos de reconfiguração usando cabo USB. Estes dados são mostrados na Tabela 5.1.

Segundo Carvalho [CAR04], o componente ICAP é documentado de forma muito precária pelo fabricante de FPGAs Xilinx. As poucas informações sobre ele definem sua interface igual ao modo *Slave SelectMAP*. Desta forma, o componente não pode ser utilizado no projeto desenvolvido por Carvalho. Para contornar o problema, foi desenvolvido um emulador de *SelectMAP*. Trata-se de um módulo que configura o dispositivo através da interface externa

Tabela 5.1: Tempos de reconfiguração de bitstreams de tamanhos diferentes usando cabo USB.

Tamanho do bitstream (Kb)	Área em núm. de CLBs	Área em núm. de LUTs	Tempo de reconfiguração (ms)
28	80	640	360
51	160	1280	404
93	320	2560	520
200	640	5120	740
499	1280	10240	1510

de configuração.

Para desenvolvimento do emulador de *SelectMAP*, na plataforma V2MB1000, foram estudados os sinais que são usados no modo *SelectMAP*. Entre este sinais, existe um sinal de relógio que controla a configuração do FPGA. Este sinal é denominado *CCLK*. De acordo com este sinal mostrado na Figura 5.8, a diferença entre o tempo de amostragem da borda de subida da primeira oscilação do sinal e o tempo de amostragem da borda de descida da última oscilação do mesmo resulta no tempo de reconfiguração do FPGA (Δt).

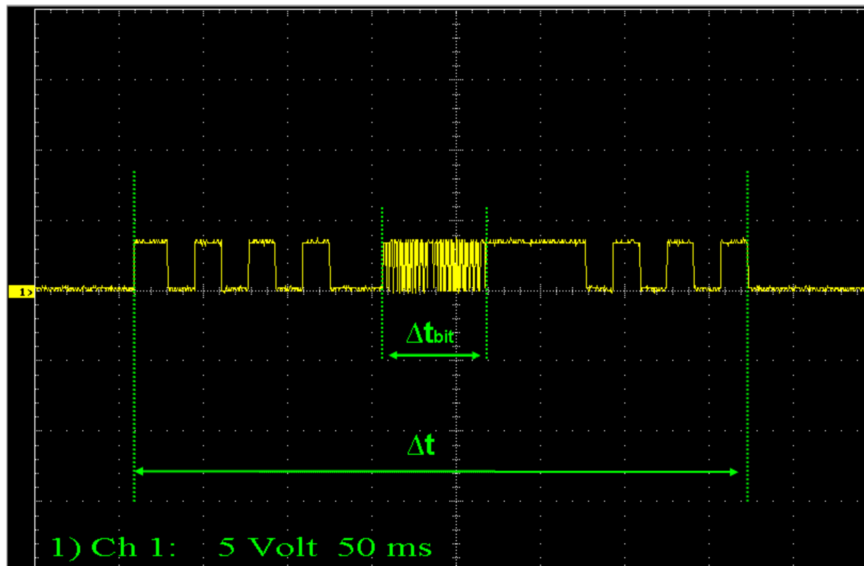

Figura 5.8: Amostragem do sinal *CCLK* durante uma configuração de FPGA usando o cabo Multilinx.

O tempo em que leva para os dados do bitstream serem carregados para o FPGA é determinado pela variável Δt_{bit} mostrada na Figura 5.8. A variável Δt_{rest} (intervalo de tempo de dados para sincronização) é determinada através da equação abaixo:

$$\Delta t_{rest} = \Delta t - \Delta t_{bit} \tag{5.1}$$

Carvalho [CAR04] fez experimentos para possibilitar a redução do tempo de reconfigu-

ração. Neste trabalho, foram analisados todos os sinais da interface *selectMAP* durante o processo de configuração. A partir daí, foi desenvolvido um circuito que emula o modo *slave selectMAP* da plataforma descrita nesta Seção. Carvalho observou que era possível realizar melhorias no emulador através de alguns eventos percebidos:

- Identificou-se que o tempo total de configuração poderia ser reduzido a 1/3 do tempo original na configuração via software. Isso foi possível porque os primeiros quatro pulsos de relógio de configuração, mostrados na Figura 5.8, possuíam uma freqüência desnecessariamente baixa em relação aos demais que compunham ao processo de configuração em si;

- Identificou-se também que os pulsos de relógio após o último pulso que envia dados de configuração (Δt_{bit} da Figura 5.8) eram desnecessários. Dessa forma, pode-se remover estes ciclos;

- Gradativamente aumentou-se a freqüência de operação do emulador até 12 MHz;

Carvalho diminuiu empiricamente o período dos ciclos de sincronização na ordem de poucos microssegundos, onde $\Delta t_{rest} << \Delta t_{bit}$, obtendo então $\Delta t \approx \Delta t_{bit}$. Carvalho ainda conseguiu reduzir o tempo representado por Δt_{bit} para uma ordem de grandeza através do aumento da freqüência de operação do emulador.

Para certificar o correto funcionamento desse emulador, foi implementado um sistema em duas placas de prototipação. A primeira delas contém a lógica reconfigurável. A segunda contém o emulador *SelectMAP*. As duas plataformas foram conectadas, ligando os dados gerados pelo emulador à interface *SelectMAP* de outra plataforma através de fios externos. Assim, a segunda plataforma reconfigura parcialmente o dispositivo da primeira plataforma, transformando a lógica original desta.

A Figura 5.9 apresenta uma comparação dos tempos de reconfiguração extraídos no modo *slave selectMAP* e usando o circuito emulador desenvolvido por Carvalho [CAR04].

Analisando os resultados, houve uma redução com o uso do emulador, em média, de 94.4% do tempo de reconfiguração usando USB. Isto significa dizer que um bitstream que obteve 740 ms de tempo de reconfiguração usando cabo USB, este bitstream obteve 42 ms de tempo de reconfiguração com o uso do emulador em relação à freqüência de operação do cabo usado. Note-se que este último valor é ainda maior que os 7.5 ms de configuração obtidos no modo *Master SelectMAP*. Esta diferença ocorre porque o gargalo de configuração é a freqüência máxima de operação para acesso à memória SRAM externa da plataforma V2MB1000 (24MHz para o controle usado). É possível realizar reconfigurações parciais de forma eficiente com esta proposta de emulador.

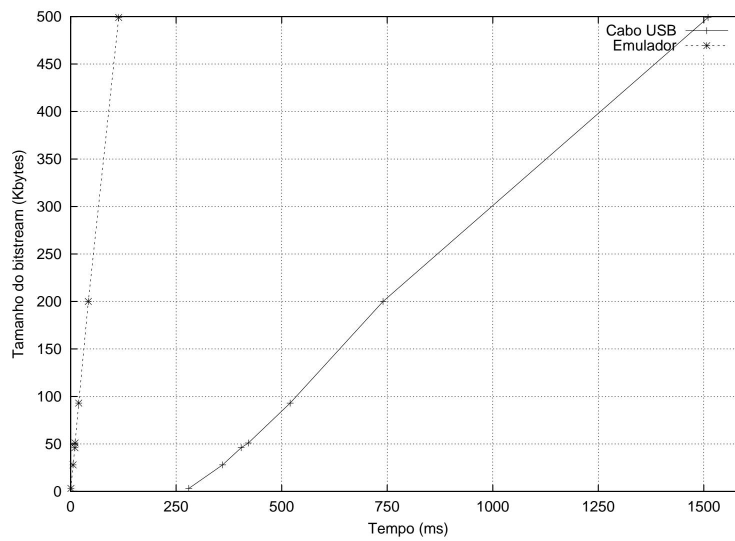

Figura 5.9: Ganho obtido no tempo de reconfiguração utilizando o emulador desenvolvido por Carvalho em relação ao modo *slave selectMAP*. Nota-se que houve uma redução, em média, de 94.4% do tempo de reconfiguração em relação ao modo *slave selectMAP*.

Capítulo 6

Processador Reconfigurável

Para validação de um estudo de caso maior usando o fluxo do Projeto Modular, adicionou-se módulos reconfiguráveis e instruções de acesso aos módulos em um processador existente desenvolvido pelo grupo local de trabalho. O sistema R8 Reconfigurável (R8R) é composto por dois núcleos IP: (i) um processador que contém instruções adicionais para acesso a um núcleo reconfigurável (R8R); (ii) núcleos reconfiguráveis chamados de coprocessadores reconfiguráveis que podem ter suas funcionalidades alteradas de acordo com a execução do programa no processador R8R.

6.1 O Processador R8

Neste estudo de caso usa-se um processador *load-store* de 16 bits chamado de R8 [MOR03]. O processador R8 faz parte de uma família de processadores, concebida com a finalidade de dar suporte ao ensino de conceitos de arquitetura e organização de computadores em nível de graduação e pós-graduação. Esse processador é uma máquina Von Neuman, com memória de dados e instruções conjunta. Este apresenta um formato regular de instrução: todas as instruções tem exatamente o mesmo tamanho, ocupando uma palavra de 16 bits. A instrução contém código de operação e a especificação dos operandos, no caso que estes existam. Este processador é uma máquina RISC, porém ainda faltam algumas características tão comuns na maioria dos processadores RISC tais como *pipeline*. As principais características organizacionais específicas neste processador multi-ciclo são:

- Dados e endereços de 16 bits;
- Banco de registradores com 16 registradores de propósito geral;
- 4 flags de status: negativo, zero, *carry* e *overflow*;
- A execução das instruções requer 2 a 4 ciclos de relógio. Os ciclos são assim denominados: (i) ciclo 1 - busca de instruções; (ii) ciclo 2 - leitura de registradores; (iii) ciclo 3 -

operação com a ULA e (iv) ciclo 4 - escrita dos resultados;

O conjunto de instruções do processador realiza as seguintes operações:

- Operações lógicas e aritméticas binárias (com 3 operandos): soma, subtração, E, OU, OU exclusivo.
- Operações lógicas e aritméticas com constantes curtas: soma, subtração;
- Operações unárias (com 1 operando): deslocamento para direita ou para a esquerda e inversão (NOT);
- Carga de metade de um registrador com uma constante (LDL e LDH);
- Inicialização do apontador de pilha (LDSP) e retorno de subrotina (RTS);
- NOP (no operation): operação nula;
- HALT: suspende a execução de instruções;
- Load: leitura de posição de memória para um registrador (LD);
- Store: armazenamento de dado de um registrador em uma posição de memória (ST);
- Saltos e chamada de subrotina com endereçamento relativo com deslocamento curto ou longo (contido em um registrador) e endereçamento absoluto (a registrador);
- Inserção e remoção de valores no/do topo da pilha (PUSH e POP);

O processador R8 conta com o seguinte conjunto de registradores de 16 bits:

- IR (*Instruction Register*): armazena o código de operação (*opcode*) da instrução atual e o(s) operando(s) desta;
- PC (*Program Counter*): é o contador de programas do processador R8;
- SP (*Stack Pointer*): armazena o endereço do topo da pilha, controla a chamada e retorno de subrotinas. Deve ser inicializado a cada programa com a instrução LDSP (carrega endereço do topo da pilha);
- 16 registradores de propósito geral rotulados de R0 a R15. O banco de registradores tem uma porta de escrita e duas de leitura. Isto significa que é possível escrever em apenas um registrador por vez, é possivelmente realizar duas leituras simultâneas;
- quatro bits de estado, denominados n (negativo), z (zero), c (*carry*) e v (*overflow*);

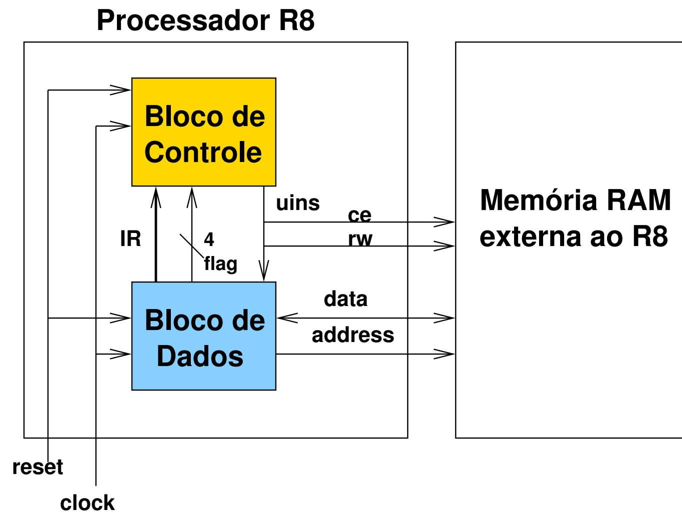

Figura 6.1: Diagrama de blocos da relação entre o processador R8 e a memória.

A Figura 6.1 mostra um diagrama de blocos processador-memória. O processador recebe do mundo externo dois sinais de controle: *clock*, que sincroniza os eventos internos ao processador; e *reset*, que inicializa o processador para iniciar a execução de instruções a partir da posição zero da memória.

O bloco de controle gera a microinstrução (*uins*) para execução das instruções. A microinstrução é responsável por comandar as ações que serão executadas no bloco de dados, como habilitação de escrita em registradores, controle de multiplexadores no bloco de dados, operação que a ULA executará e acesso à memória externa.

O bloco de dados envia para o bloco de controle a instrução corrente (conteúdo do IR) e os qualificadores de estado (flags). Os sinais para a troca de informações com a memória são: *data* (barramento bidirecional de 16 bits para os dados) e *address* (barramento de 16 bits com os endereços de memória). O controle de acesso à memória é feito pelo bloco de controle, através dos sinais *ce* e *rw*. O sinal *ce* indica se está em curso uma operação com a memória e o sinal *rw* indica se esta operação é de escrita ou de leitura.

É importante ressaltar que os blocos de dados e de controle operam em fases distintas do sinal *clock*. Em uma borda do *clock* (por exemplo, subida) o bloco de controle gera a micro-instrução, e na borda seguinte (descida) o bloco de dados modifica os registradores. Com isto sempre se tem dados estáveis nas transições de relógio em cada um dos blocos.

6.2 Características Adicionadas ao Processador R8 - Processador R8R

O desenvolvimento do processador R8R é um projeto desenvolvido como uma das contribuições deste trabalho. Este emprega conceitos de processadores que contém um conjunto de instruções dinâmico, denominados *Dynamic Instruction Set Computers* ou DISCs, tratados por Wirthlin e Hutchings [WIR98]. Tal projeto propõe o desenvolvimento de um processador dinâmico e parcialmente reconfigurável (processador R8R), baseado na estrutura do processador R8.

O processador R8R permite a execução de vários coprocessadores em uma mesma área do FPGA através da reconfiguração parcial e dinâmica. O sistema R8R prototipado consiste em ter um módulo fixo denominado de *processador R8R* (sistema de um processador R8R e uma memória) e um módulo reconfigurável chamado *coprocessador* (coprocessador reconfigurável).

O módulo fixo é responsável por agregar cinco módulos:

- *processador R8R*: este módulo é o processador R8 modificado para prover controle dos coprocessadores reconfiguráveis. O processador executa funções em software e também requisita ao coprocessador que este execute determinada função;

- *memória RAM*: responsável conectar os módulos serial e processador R8 em memórias *blockRAMs* do FPGA;

- *serial*: este módulo permite a comunicação do sistema R8R com um PC hospedeiro. O software do lado do hospedeiro é um software elaborado pelo grupo local para enviar e receber dados pela interface serial do computador. Este envia dados para o sistema R8R incluindo carga da memória com o código-objeto que o processador R8R vai executar. O sistema hospedeiro também pode receber dados do sistema R8R através de leitura da memória do mesmo;

- *barramento*: módulo responsável pelo tráfego de dados que ocorrem entre os módulos serial, processador R8R e memória;

- *árbitro*: módulo responsável para definir qual é o módulo que vai ter acesso ao barramento em um determinado instante;

Para que o processador pudesse interagir com módulos, estes foram interconectados através de um barramento compartilhado. Neste, foi implementado uma política de acesso pelos coprocessadores e processador. Foi implementado um árbitro de barramento responsável por iniciar e monitorar todas as requisições feitas ao barramento [PAT98].

Como mostrado na Figura 6.2, os módulos *processador R8R*, *serial* e *memória* são conectados ao barramento. O coprocessador que está localizado na área reconfigurável está

conectado diretamente com o processador R8R através de sinais de dados e controle. O módulo controlador de configurações é usado para reconfigurar coprocessadores por demanda gerada pelo processador R8R (via software). Seu detalhamento está fora do escopo deste trabalho e encontra-se em [CAR04].

Ainda na Figura 6.2 mostra os sinais que fazem a interconexão entre o processador R8R, coprocessadores reconfiguráveis e o controlador de configurações.

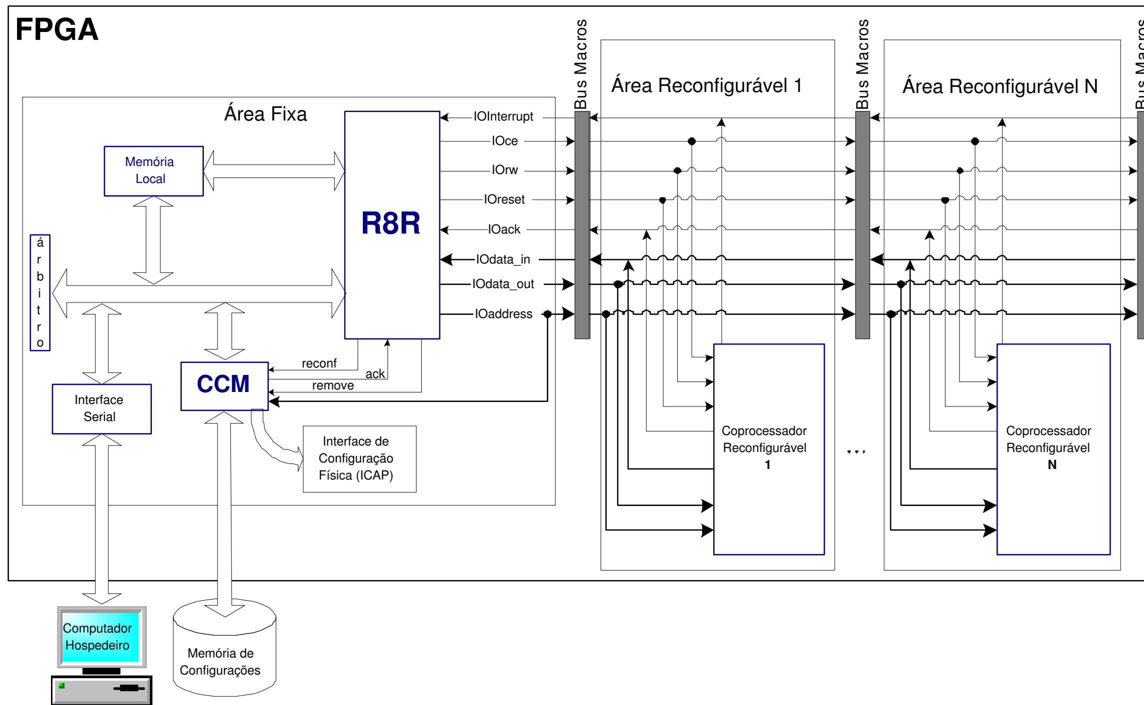

Figura 6.2: Uma visão geral do sistema R8R.

Para obter a funcionalidade desejada do processador R8R, as seguintes características foram adicionadas ao processador R8 original:

- suporte para conexão de um ou mais coprocessadores reconfiguráveis, de acordo com o protocolo de comunicação entre o R8R e o coprocessador;

- interface de 16 bits de dados com o coprocessador reconfigurável;

- interface de 11 bits com o controlador de configurações, responsável por envia arquivos de configurações sob demanda no dispositivo reconfigurável;

- adição de 5 novas instruções para seleção (SELR), inicialização (INITR), leitura (RDR), escrita (WRR) e disabilitação (DISR) de coprocessadores reconfiguráveis conforme detalhado na Tabela 6.1;

Tabela 6.1: Instruções adicionadas para executar funções associadas ao(s) coprocessador(es) reconfigurável(is).

Instrução reconfigurável	**Código Objeto**	**Descrição**
selr	B *ender_8_bits* B	Seleciona para operação o coprocessador identificado por *ender_8_bits*. Caso o coprocessador não esteja atualmente configurado, o controlador de configurações o carrega, avisando o processador do término do processo.
initr	B *ender_8_bits* C	Inicializa um determinado coprocessador identificado por *ender_8_bits*.
disr	B *ender_8_bits*	Informa para o controlador de configurações que o coprocessador identificado por *ender_8_bits* foi removido logicamente.
wrr	B reg_fonte_1 reg_fonte_2 D	Envia os conteúdos de 2 registradores reg_fonte_1 e reg_fonte_2 para o coprocessador.
rdr	B reg_fonte_1 reg_fonte_2 E	Envia o conteúdo de reg_fonte_1 para o coprocessador selecionado e lê em seguida um dado que é colocado em reg_fonte_2

Um protocolo de comunicação é utilizado entre o processador R8R e o coprocessador formado por 8 sinais:

- *IOce*: sinal ativo em '1', que permite ao processador acessar o coprocessador identificado pelo sinal *IOaddress*;

- *IOInterrupt*: sinal ativo em '1'. Este sinal indica o término de execução do coprocessador. Se este sinal estiver desabilitado e se for executada uma operação de leitura (instrução RDR) no coprocessador, a execução da instrução ficará esperando até que o sinal *IOInterrupt* seja habilitado.

- *IOaddress*: barramento de oito bits que contém o endereço para identificação do coprocessador. Este é selecionado quando seu identificador é idêntico ao valor transmitido por este barramento;

- *IOrw*: sinal para modos de operação leitura/escrita que o processador requisita ao coprocessador. Este sinal corresponde à escrita de dados do processador para o coprocessador, em '1' corresponde à operação de leitura;

- *IOreset*: sinal ativo em baixo, que inicializa um coprocessador identificado por *IOaddress*;
- *IOack*: sinal ativo em '1', utilizado para um coprocessador sinalizar que recebeu determinado dado ou realizou uma operação com sucesso;
- *IOdata_in*: barramento de 16 bits utilizado para permitir que o processador possa ler dados do coprocessador identificado por *IOaddress*;
- *IOdata_out*: barramento de 16 bits utilizado para permitir que o processador possa escrever dados no coprocessador identificado por *IOaddress*;

Três sinais foram adicionados ao processador R8R para prover comunicação deste com o controlador de configurações:

- *IOaddress*: este é utilizado também para a comunicação entre o coprocessador e o processador; Este sinal é utilizado para identificação do coprocessador que será reconfigurado e acoplado no processador;
- *req_reconf*: sinal usado para requisitar uma dada configuração identificada por *IOaddress* ao controlador de configurações;
- *ack_reconf*: sinal para informar ao processador que a reconfiguração do coprocessador foi terminada. A partir daí, o processador continua sua execução normal;
- *req_remove*: sinal para informar ao controlador de configurações que o coprocessador identificado pelo sinal *IOAddress* foi removido logicamente;

As instruções SELR e DISR avisam o controlador de configurações para remover ou inserir módulos identificados no barramento *IOAddress*. Estas instruções têm caráter bloqueante.

Na implementação dos coprocessadores, é comum estabelecer que sempre seja enviado uma palavra de 16 bits de comando e a seguir, uma palavra de dados de 16 bits do processador para o coprocessador.

A partir das simulações do comportamento geral do processador R8R, será explicado detalhadamente o protocolo de comunicação entre o processador e o coprocessador.

6.3 Validação por Simulação

Depois de descrever o sistema R8R em VHDL, usou-se um software (*Modelsim* e *Active HDL*) para simulação funcional do comportamento do circuito. Foi escrito um software em linguagem de montagem para que fosse possível ao processador R8R acessar e obter os resultados da execução do coprocessador reconfigurável.

Foram desenvolvidos inicialmente cinco coprocessadores reconfiguráveis para validar o sistema R8R:

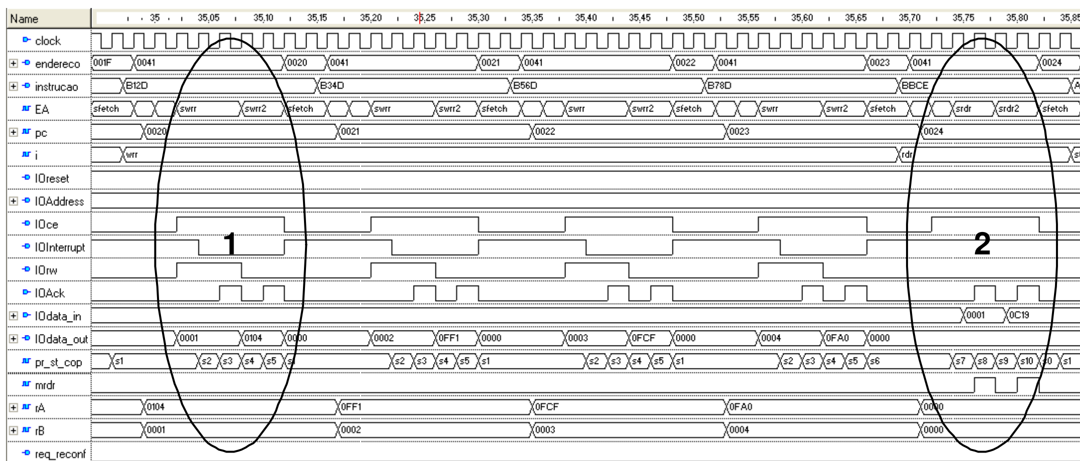

Figura 6.3: Simulação funcional do coprocessador *4-avg*.

- *2-avg*: coprocessador que executa a média de dois valores. Simulação na Figura 6.4;
- *4-avg*: coprocessador que executa a média de quatro valores. Simulação na Figura 6.3;
- *sqrt*: coprocessador que executa a raiz quadrada de um número de 32 bits e o resultado é um valor de 16 bits. Simulação na Figura 6.6);
- *multi*: coprocessador que executa a multiplicação de dois valores. Simulação na Figura 6.7;
- *div*: coprocessador que executa a divisão inteira de dois valores gerando o quociente e o resto da operação. Simulação na Figura 6.8;

Todos os coprocessadores são identificados por um endereço. A Tabela 6.2 apresenta os coprocessadores e seus endereços respectivamente.

Tabela 6.2: Identificação dos coprocessadores reconfiguráveis através de endereços específicos.

Coprocessador reconfigurável	Endereço (Hex)
4-avg	01h
2-avg	02h
multi	03h
div	04h
sqrt	05h

Na Figura 6.3 é mostrado um diagrama de tempos extraído da simulação de hardware do coprocessador *4-avg*.

Os passos da simulação do coprocessador *4-avg* são descritos a seguir:

1. a instrução B12D (corresponde ao mnemônico WRR r1,r2) é carregada. Então o procedimento de escrita dos conteúdos dos registradores 1 e 2 é realizado. Primeiramente,

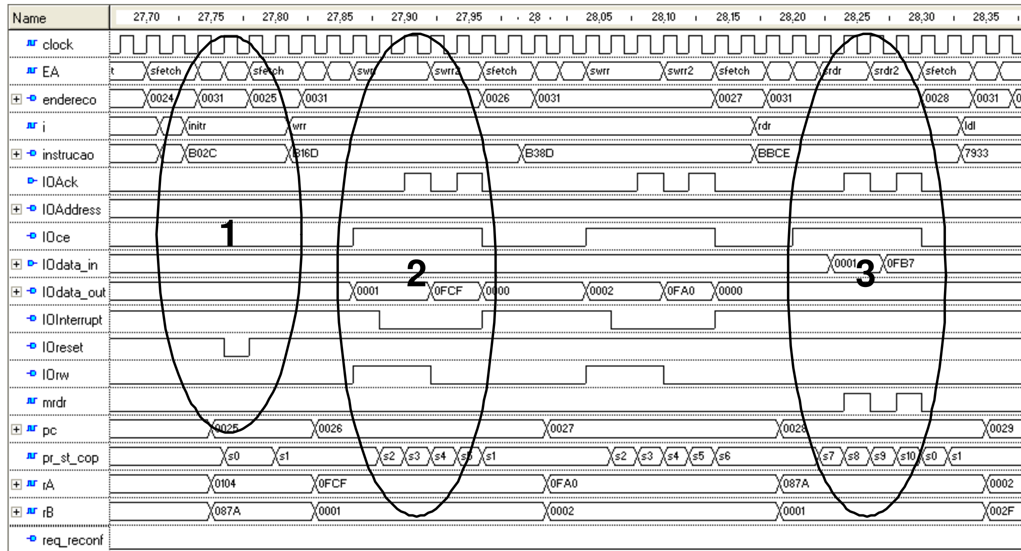

Figura 6.4: Diagrama de tempos do coprocessador *2-avg*.

os sinais *IOce* e *IOrw* são ativados para acesso ao processador no modo de escrita. A seguir, *IOdata_ out* recebe o valor do registrador r1. Ao receber este dado, o coprocessador responde ativando *IOack*. Então, o valor do registrador r2 é escrito no barramento *IOdata_ out* e mais uma vez, o coprocessador ativa *IOack* para que o processador continue executando a instrução. O identificador de estados denominado *pr_ st_ cop* mostra o estado atual da máquina de estados que implementa a lógica do coprocessador. As instruções B34D, B56D, e B78D também são executadas da mesma forma. Estas quatro últimas instruções escrevem respectivamente um comando e um dado. O comando faz com que a máquina de estados do coprocessador avance e o dado é utilizando como um operando da média de 4 valores. Os dados escritos em seqüência são 0104h, 0FF1h, 0FCFh e 0FA0h;

2. a leitura dos dados oriundos do coprocessador se deve através destes procedimentos: (i) a instrução BBCE (mnemônico RDR r11,r12) é executada e significa que o coprocessador vai escrever dados nos registradores r11 e r12 do processador R8R; (ii) o sinal de *IOce* deve estar ativo e o sinal de *IOrw* deve estar inativo para informar ao coprocessador que o processador requisita a leitura dos dados. (iii) a leitura só será feita se o sinal *IOInterrupt* estiver habilitado, indicando término da execução do coprocessador. Então o coprocessador envia no registrador r11 um dado informando o *status* e no registrador r12 a resposta da média dos quatro números previamente escritos no coprocessador (valor C19h);

A Figura 6.4 apresenta a simulação do coprocessador que realiza a média de 2 operandos.

Os passos da simulação do coprocessador *2-avg* são descritos a seguir, da mesma forma como ocorre na Figura 6.3:

1. a instrução B02C (INITR #02) é carregada e executada objetivando inicializar o coprocessador identificado pelo endereço 02 no barramento *IOaddress*. *IOreset* é ativado para inicializar o coprocessador;

2. o comando B16D (WRR r1,r6) escreve o conteúdo dos registradores r1 e r6 no coprocessador. Os sinais *IOce* e *IOrw* são ativados e os dados são escritos no barramento *IOdata_out*. Os dados 0001 (neste caso, este dado significa um comando que o coprocessador reconhece) e 0FCFh (dado para cálculo da média) são enviados serialmente para o coprocessador. Para cada dado enviado, o coprocessador ativa *IOack* avisando que o processador R8R pode continuar a execução da instrução atual. Da mesma forma acontece na instrução B38D (WRR r3,r8) para o envio do comando 0002 e do dado FA0h;

3. para leitura dos dados armazenados no coprocessador, então executa-se a instrução BBCE (RDR r11,r12) através dos procedimentos apresentados na Figura 6.3 para leitura para requisição e leitura dos dados do coprocessador. Desta forma, o coprocessador envia no registrador r11 um dado informando o *status* e no registrador r12 a resposta da média dos dois números previamente escritos no coprocessador (valor FB7h).

A Figura 6.5 apresenta a simulação da instrução SELR utilizada para a seleção de um coprocessador reconfigurável.

Os passos da simulação da instrução SELR são descritos a seguir:

1. a instrução B02B (SELR #02) é carregada e executada;

2. o sinal *req_reconf* é ativado informando ao controlador de configurações para que seja feita a reconfiguração de um novo coprocessador;

3. o coprocessador identificado por *IOAddress* vai ser reconfigurado pelo controlador de configurações;

4. a R8R permanece no estado *sselr* no seu estado atual (*EA*) até que o controlador de configurações ative o sinal *ack_reconf* informando que a reconfiguração parcial do novo coprocessador foi realizada com sucesso;

A Figura 6.6 apresenta o diagrama de tempos da execução do módulo *sqrt*. Os diagramas de tempos dos coprocessadores *sqrt*, *multi* e *divide* utilizam uma variação dos protocolos de comunicação dos módulos *2-avg* e *4-avg*.

Os items abaixo mostram cada passo da simulação do coprocessador *sqrt*. Para definir o protocolo de comunicação deste módulo, deve-se seguir as regras de seleção, inicialização e uso de sinais de acesso ao coprocessador como foi mostrado nas Figuras 6.3 e 6.4.:

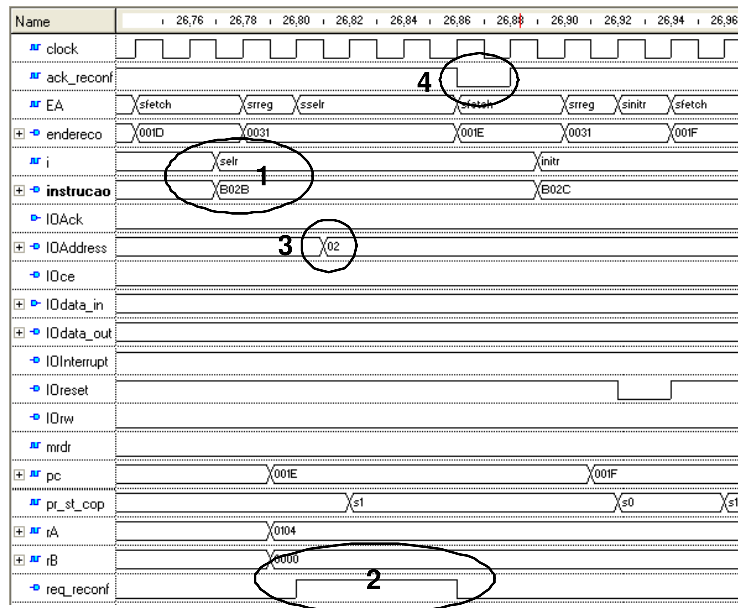

Figura 6.5: Diagrama de tempos da execução da instrução SELR.

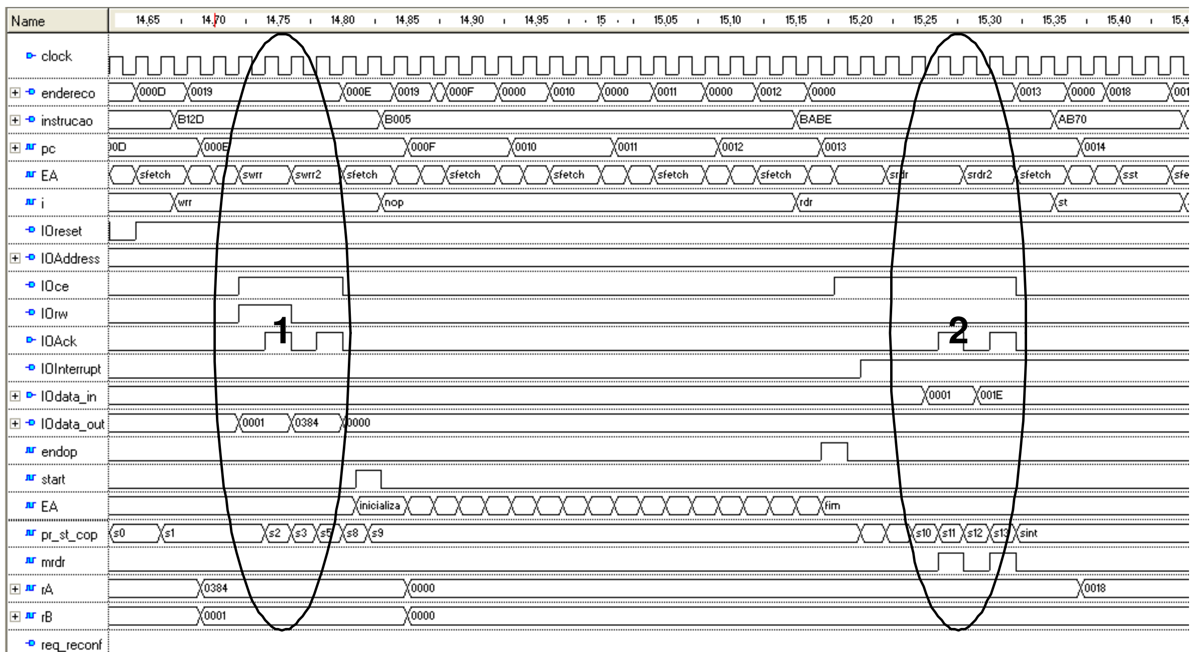

Figura 6.6: Diagrama de tempos da execução da do módulo *sqrt*.

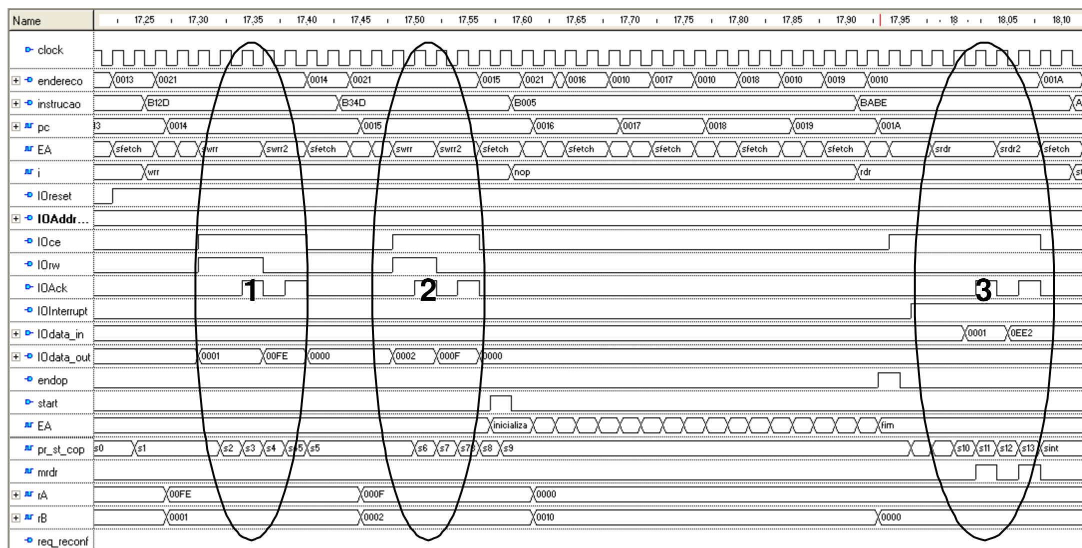

Figura 6.7: Diagrama de tempos da execução da do módulo *multi*.

1. o processador envia um valor através do sinal *IOdata_ out* para o coprocessador (valor 0384h) através de uma instrução "wrr". Desta forma, é iniciado o processo de execução do coprocessador *sqrt*;

2. quando o sinal *IOInterrupt* estiver habilitado, o coprocessador terminou sua execução. Este sinal é habilitado depois de 26 ciclos de relógio depois do início da instrução "wrr". Então a instrução "rdr" é executada. O sinal *IOdata_ in* recebe o resultado da raiz quadrada (valor 001Eh).

Nesta simulação, logo após a execução da instrução "wrr", o processador executa outras instruções em paralelo com a execução do coprocessador. O sinal rdr é executado, neste caso, depois da execução do coprocessador. A simulação do coprocessador *mult* também apresenta esta característica.

A Figura 6.7 apresenta o diagrama de tempos da execução do módulo *multi*. Este módulo é responsável pela execução da multiplicação de dois valores.

Os items abaixo mostram cada passo da simulação do coprocessador *multi*:

1. o processador envia um valor através do sinal *IOdata_ out* para o coprocessador (valor 00FEh) através de uma instrução "wrr";

2. executando uma outra instrução "wrr", *IOdata_ out* envia um outro valor (000Fh). A partir daí, começa a execução do coprocessador *multi*;

3. depois de 19 ciclos de relógio, o sinal *IOInterrupt* é habilitado, e então a instrução "rdr" é executada totalmente. O sinal *IOdata_ in* recebe o resultado da multiplicação (valor 0EE2h);

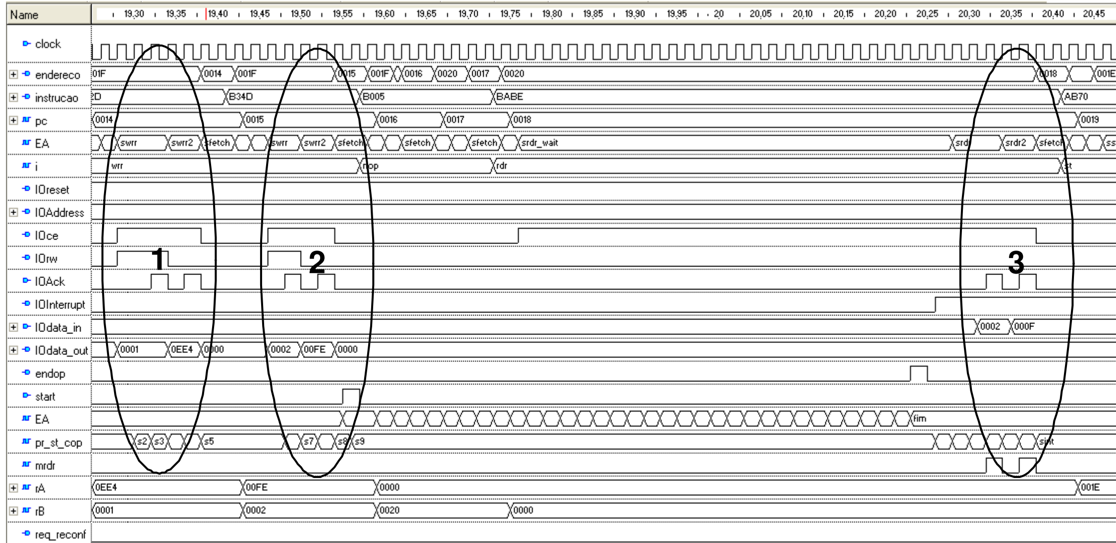

Figura 6.8: Diagrama de tempos da execução da do módulo *div*.

A Figura 6.8 apresenta a simulação do módulo *div*. Este coprocessador executa a divisão inteira entre dois valores resultando no quociente e resto.

Os items abaixo mostram cada passo da simulação do coprocessador *div*:

1. o processador envia um valor através do sinal *IOdata_out* para o coprocessador o valor do dividendo (valor 0EE4h) através de uma instrução "wrr";

2. executando uma outra instrução "wrr", *IOdata_out* envia o valor do divisor (valor 00FEh). O processo de execução do coprocessador é iniciado;

3. depois de 32 ciclos de relógio, o sinal *IOInterrupt* é habilitado, e então a instrução "rdr" é executada totalmente. O sinal *IOdata_in* recebe o resultado do resto e da divisão (valores 0002h e 000Fh respectivamente);

Nesta simulação, o processador executa a instrução de leitura (instrução "rdr") antes do término de execução do coprocessador. Então o processador bloqueia a sua execução e espera o término da execução do coprocessador. Quando o coprocessador termina sua execução, o processador é liberado e executa a instrução de leitura. conforme a Figura 6.8.

6.4 Prototipação

Depois da execução da simulação do sistema como um todo com os cinco coprocessadores (não no mesmo sistema), prototipou-se a R8R usando o fluxo do Projeto Modular.

No sistema, foram inseridos DCMs e bus macros para o provimento de compensação de propagação do sinal de relógio e comunicação entre as áreas fixa e reconfigurável respectivamente. Foram inseridos 11 bus macros instanciadas no código HDL e posicionadas nos arquivos de restrições para a comunicação dos sinais entre o processador e o coprocessador. Depois da inserção de todos os componentes e arquivos necessários para a execução do fluxo, deve-se sintetizar logicamente cada arquivo fonte (arquivos de maiores níveis hierárquicos e os módulos coprocessadores). Em seguida, executar o fluxo usando como entrada os arquivos gerados pela síntese lógica e, arquivos de restrição do usuário e arquivos de macros provida pelo fabricante.

Depois da execução do fluxo do Projeto Modular e ao término da geração dos bitstreams totais e parciais, para a validação em hardware vários passos são necessários. Primeiro foi feito a configuração do bitstream total do sistema R8R com o coprocessador *4-avg* acoplado. Foi previamente desenvolvido software que primeiramente executa instruções para acessar o coprocessador *4-avg* e depois executa instruções para acessar os coprocessadores *2-avg*, *sqrt*, *multi* e *div*.

O primeiro software encontra-se no Apêndice A.1, faz a média de 4 valores. Na Tabela 6.3 é mostrado a seqûencia das instruções reconfiguráveis executadas neste software.

O software é transferido para a memória da R8R através de um software em JAVA desenvolvido pelo grupo local. A Figura 6.9 mostra uma tela desta aplicação.

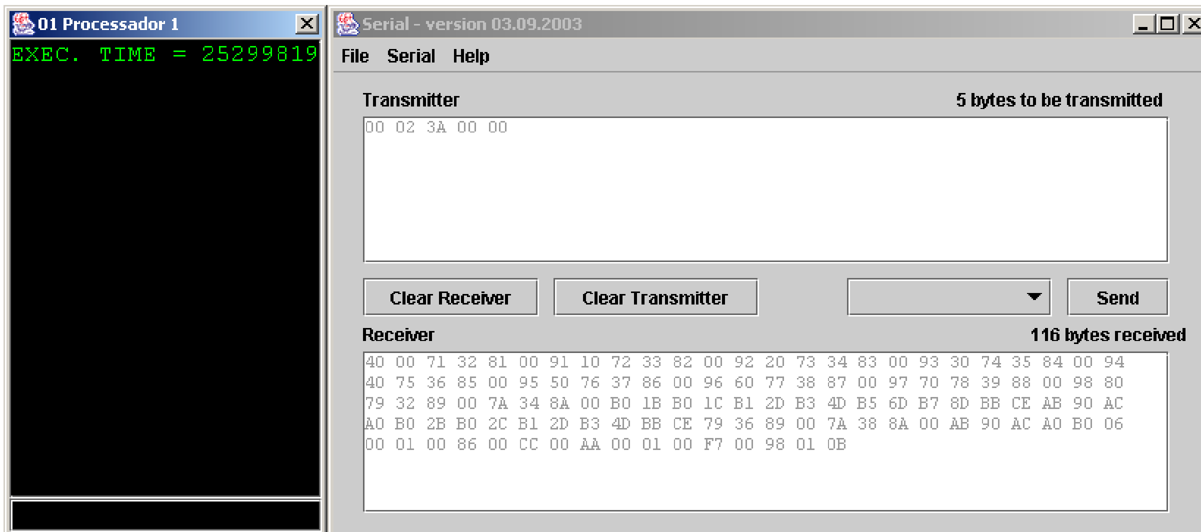

Figura 6.9: Telas de aplicação do software que conduz dados oriundos de um PC hospedeiro para o processador R8R. A janela superior mostra dados que serão transmitidos para a R8R. A janela inferior direita são apresentados os dados de leitura da memória do R8R, bem como envio de comandos do R8R para o PC hospedeiro. A janela situada à esquerda mostra resultados de tempos de execução e informações a respeito do término da execução do software executado pela R8R.

Depois de enviar o software para a R8R, deve-se então disparar o processamento da mes-

Tabela 6.3: Exemplo de seqüência de instruções reconfiguráveis utilizado no software de teste (Apêndice A.1).

Instrução reconfigurável	**Descrição**
initr #1	inicializa coprocessador de endereço 1 (cop. *4-avg*)
wrr r1,r2	envia o conteúdo de R1 (0001h) e R2 (0086h) serialmente para o coprocessador 1
wrr R3,R4	envia o conteúdo de R3 (0002h) e R4 (00AAh) serialmente para o coprocessador 1
wrr R5,R6	envia o conteúdo de R5 (0003h) e R6 (00F7h) serialmente para o coprocessador 1
wrr R7,R8	envia o conteúdo de R7 (0004h) e R8 (010Bh) serialmente para o coprocessador 1
rdr R11,R12	recebe o s dados do coprocessador 1 serialmente (o *status* se destina ao R11 (0001h) e o resultado se destina ao R12 (00CCh)
selr #2	requisita a reconfiguração do coprocessador de endereço 2 (cop.*2-avg*)
initr #2	inicializa coprocessador de endereço 2 (cop. *2-avg*)
wrr R1,R2	envia o conteúdo de R1 (0001h) e R2 (0086h) serialmente para o coprocessador 2
wrr R3,R4	envia o conteúdo de R3 (0002h) e R4 (00AAh) serialmente para o coprocessador 2
rdr R11,R12	recebe os dados do coprocessador 2 serialmente (o *status* se destina ao R11 (0001h) e o resultado se destina ao R12 (0098h).
selr #3	requisita a reconfiguração de endereço 2 (cop. *multi*)
wrr R1,R2	envia o conteúdo de R1 (0001h) e R2 (00FEh) serialmente para o coprocessador 3
wrr R3,R4	envia o conteúdo de R3 (0002h) e R4 (000Fh) serialmente para o coprocessador 3
rdr R11,R12	recebe os dados do coprocessador 3 serialmente (o *status* se destina ao R11 (0001h) e o resultado se destina ao R12 (0EE2h).
selr #4	requisita a reconfiguração de endereço 4 (cop. *div*)
wrr R1,R2	envia o conteúdo de R1 (0001h) e R2 (0EE4h) serialmente para o coprocessador 4
wrr R3,R4	envia o conteúdo de R3 (0002h) e R4 (00FEh) serialmente para o coprocessador 4
rdr R11,R12	recebe os dados do coprocessador 4 serialmente (o resto da divisão se destina ao R11 (0002h) e o resultado se destina ao R12 (000Fh).
selr #5	requisita a reconfiguração de endereço 5 (cop. *sqrt*)
wrr R1,R2	envia o conteúdo de R1 (0001h) e R2 (0384h) serialmente para o coprocessador 5
rdr R11,R12	recebe os dados do coprocessador 5 serialmente (o *status* se destina ao R11 (0001h) e o resultado se destina ao R12 (001Eh).

ma. Utiliza-se então um comando especial no software da serial para iniciar a execução do processador.

Como o controlador de configurações estava em desenvolvimento durante a fase de implementação do sistema R8R, foi elaborado um meio de prototipar o sistema sem o uso do controlador, conforme descrito a seguir, emulando este usando chaves e displays da plataforma. Para que fosse feita a a reconfiguração parcial, inicialmente alguns sinais de saída do processador R8 foi mapeado em *leds*, botões da plataforma, além de se empregar um analisador lógico para observação de pinos selecionados do FPGA. Abaixo são apresentados os sinais

que foram mapeados:

- *req_reconf*: este sinal foi mapeado em um *led* da plataforma. Quando o processador requisita uma dada configuração, o *led* acende;
- *IOaddress*: este barramento foi mapeado num analisador lógico identificando qual é o coprocessador que será reconfigurado;
- *ack_reconf*: sinal associado por um botão da plataforma servindo para avisar que a reconfiguração do coprocessador identificado por *IOaddress* já foi terminada e também informar que o processador já pode continuar a sua execução;

Quando a luz do *led* acender, então deve-se configurar o FPGA com o arquivo de configuração parcial identificado no analisador lógico (02), ou seja, deve-se configurar o FPGA com o coprocessador *2-avg* através de um software e hardware de configuração. Depois que a reconfiguração parcial for realizada, então deve-se pressionar o botão do FPGA que ativa o sinal *ack_reconf* para que o processador possa continuar sua execução. Depois de realizar sucessivas reconfigurações parciais de acordo com o identificador amostrado no analisador lógico, o software termina de executar, enviando um aviso para a serial. Para capturar os resultados, deve-se então realizar um *dump*[1] na memória. A Tabela 6.4 mostra o conteúdo da memória (exceto instruções) depois da execução do software. Os resultados dos registradores r11 e r12, no software, foram armazenados na área de dados do programa definido nas coordenadas D1 a I2 da Tabela 6.4. O dado B006 definido nas posições B1 a C1 é a última instrução do programa (HALT). Os valores do coprocessador *4-avg* armazenados em R11 e R12 foram armazenados nas posições de memória apontados por D1 a E1 (valor do status 0001h) e H1 a I1 (valor da resposta da média 00CCh). Os valores do coprocessador *2-avg* armazenados em R11 e R12 foram armazenados respectivamente nas posições de memória B2 a C2 (status: 0001h) e F2 a G2 (resposta da média de dois valores: 0098h). As respostas da execução dos coprocessadores *multi*, *div* e *sqrt* estão armazenados respectivamente nas posições B4 a C4, D5 a E5 e B6 a C6. Através destes resultados valida-se funcionalmente a reconfiguração parcial e dinâmica nos coprocessadores da R8R.

Na execução, foram encontrados erros de sincronismo de relógio e corrigidos. O problema foi causado por um sinal conectado ao DCM e que este sinal atravessou a fronteira entre a área fixa e reconfigurável. Solucionou-se este problema com a inserção de uma bus macro neste sinal.

No que diz respeito a todos os estudos de caso apresentados no Capítulo 5 e neste Capítulo, foi feito o seguinte experimento: carregou-se o dispositivo reconfigurável com um *bitstream* total inicial e a seguir, carregou-se novamente o dispositivo, em tempo de execução, com um *bitstream* parcial contendo o mesmo módulo de hardware que estava armazenado no

[1] Cópia dos dados da memória para um arquivo especificado. Esta cópia pode ser de toda a memória ou apenas cópia dos dados apontados dentro de certos limite

Tabela 6.4: Conteúdo da memória do R8R logo após feita a reconfiguração parcial e execução do software.

	A	B	C	D	E	F	G	H	I	J
linha 1	...	B0	06	00	01	00	86	00	CC	00
linha 2	AA	00	01	00	F7	00	98	01	0B	00
linha 3	01	00	FE	00	02	00	0F	00	01	00
linha 4	01	0E	E2	00	01	0E	E4	00	02	00
linha 5	FE	00	02	00	0F	00	01	03	84	00
linha 6	01	00	1E	...						

bitstream total inicial. Em **todos** os estudos de caso apresentados, o comportamento do circuito permaneceu inalterado no instante da reconfiguração parcial e depois desta a partir da observação de leds e displays da plataforma de prototipação e através de medidas feitas pelo analisador lógico..

6.5 Análise dos Resultados na Execução do Processador R8R

No sistema R8R, foram desenvolvidos vários coprocessadores reconfiguráveis, segundo o que consta neste Capítulo. Os *bitstreams* parciais que implementam coprocessadores que executam operações matemáticas tais como raiz quadrada, divisão e multiplicação têm em média um tamanho de 46 Kb. Segundo as medidas extraídas do gráfico da Figura 5.9, o tempo de reconfiguração feito via cabo USB foi cerca de 400 ms. Porém com o uso do emulador, é possível realizar uma reconfiguração em um tempo de 10 ms [CAR04]. Para avaliar o desempenho quanto a velocidade de cada coprocessador, admitiu-se aqui o uso do tempo de reconfiguração de 10 ms.

Desenvolveu-se versões em softwares com a mesma funcionalidade para cada um dos coprocessadores reconfiguráveis. Deste modo, pode ser feita uma análise comparativa entre os tempos reconfiguração e execução dos módulos reconfiguráveis versus o tempo de execução em software. A seguir, serão discutidos os resultados iniciais de avaliação do tempo de execução de cada um dos coprocessadores reconfiguráveis.

6.5.1 Coprocessador de multiplicação

No Apêndice A.2 é apresentado o software que foi executado na R8R. Este software não acessa nenhum coprocessador, apenas executa as instruções nativas do processador R8 [MOR03]. A Figura 6.10 mostra um gráfico comparativo entre os tempos de execução do software em relação ao número de operações e o tempo de reconfiguração e execução do coprocessador em relação também ao número de operações, supondo que versões de software e hardware sejam iteradas de forma ininterrupta.

A partir de um certo número de operações, as retas se cruzam. Isto significa que a partir de 750 operações, a reconfiguração parcial do coprocessador de multiplicação é mais eficiente

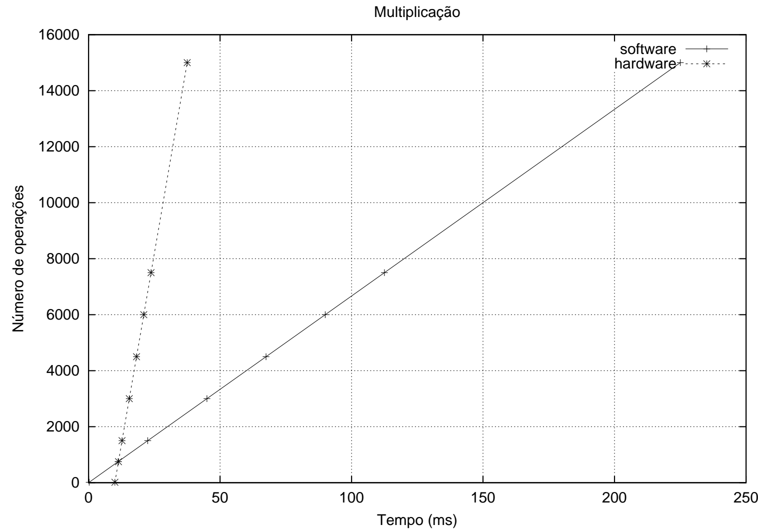

Figura 6.10: Gráfico comparativo entre os tempos de reconfiguração e execução do coprocessador de multiplicação com o tempo de execução do software ambos em relação ao número de operações.

em termos de tempo de execução que uma implementação em software. Uma observação importante é que assume-se o software começando a ser executado ao mesmo tempo que o coprocessador a ser reconfigurado. A execução do coprocessador só é inicializada depois da reconfiguração (a partir do tempo de 10 ms).

Os tempos de execução do software foram extraídos através da contagem do número de ciclos de relógio do software assumindo uma freqüência de relógio de 24 MHz para o processador. Os tempos de execução do hardware foram extraídos através da contagem do número de ciclos de relógio da execução do coprocessador a uma freqüência de 24 MHz.

6.5.2 Coprocessador de divisão

No Apêndice A.3 é apresentado o software de divisão que foi executado na R8R. A Figura 6.11 mostra um gráfico comparativo entre os tempos de execução do software em relação ao número de operações e o tempo de reconfiguração e execução do coprocessador em relação também ao número de operações. As condições de cálculo dos tempos são idênticas às do estudo de caso anterior.

No ponto de cruzamento das duas retas, nota-se que há ganho o uso do coprocessador reconfigurável ao invés da execução do software a partir da execução de 300 operações de divisão, antes de uma nova configuração.

6.5.3 Coprocessador de raiz quadrada

A Figura 6.12 apresenta um gráfico comparativo semelhante aos gráficos mostrados nas Figuras 6.10 e 6.11 para o coprocessador e o software de raiz quadrada. No Apêndice A.4 é

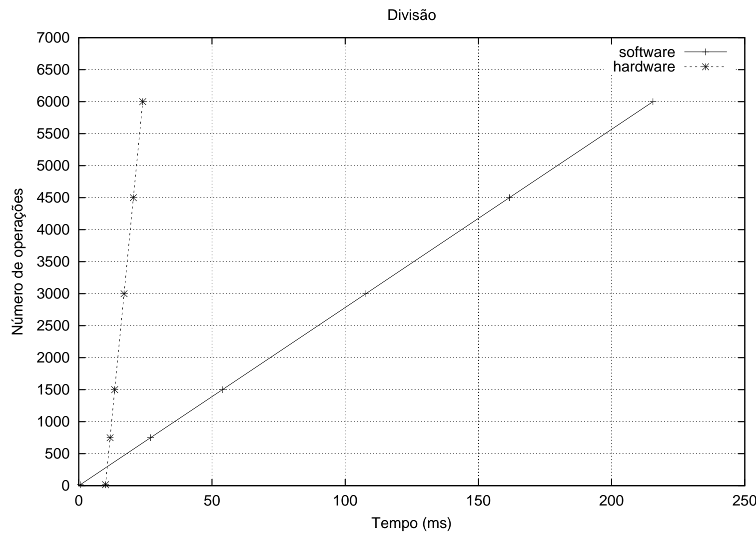

Figura 6.11: Gráfico comparativo entre os tempos de reconfiguração e execução do coprocessador de divisão e o tempo de execução do software ambos em relação ao número de operações.

listado o código-fonte do software que implementa a função raiz quadrada na R8R.

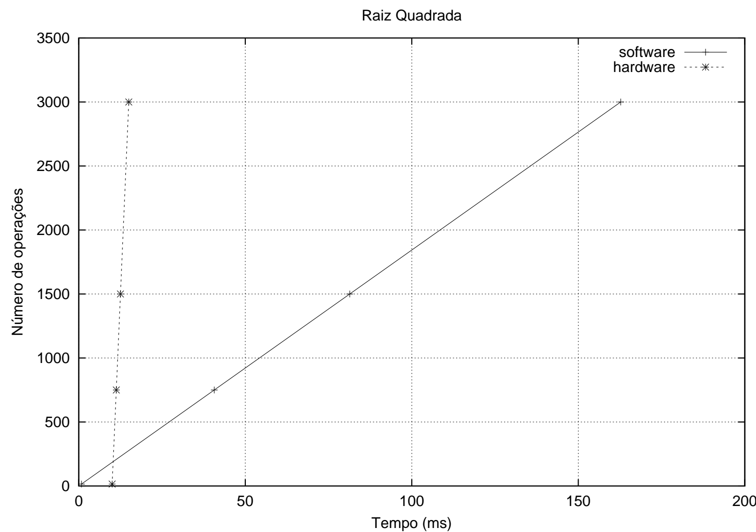

Figura 6.12: Gráfico comparativo entre os tempos de reconfiguração e execução do coprocessador de raiz quadrada e o tempo de execução do software ambos em relação ao número de operações.

Como é observado na Figura 6.12, a partir de 191 operações de raiz quadrada, o uso do coprocessador reconfigurável passa a ser mais interessante que o software equivalente.

As curvas referentes a operação dos coprocessadores implementados em hardware representam uma abordagem do pior caso. Isso porque os experimentos foram desenvolvidos usando apenas uma área reconfigurável. Se o sistema possuir diversas áreas reconfiguráveis, o tempo

de reconfiguração pode ser escondido, assim obtendo um ganho de desempenho global do sistema.

Para melhor entendimento da importância do uso dos coprocessadores em contraproposta às rotinas em software pode-se imaginar um sistema de software que aplica uma série de filtros em imagens. Se esses filtros aplicam multiplicações, divisões, médias de valores e raízes quadradas constantemente, muito tempo poderá ser ganho utilizando os coprocessadores de alto desempenho, ainda mais se esses filtros forem aplicados a diversas imagens ou se as mesmas forem de grande tamanho.

Duas das mais importantes vantagens do processador implementado utilizando técnicas de reconfiguração dinâmica e parcial são o aumento da densidade funcional e da flexibilidade do sistema, conforme mencionado no Capítulo 1. Tal flexibilidade permite a modificação do sistema em tempo de execução, com razoável facilidade, através da adição de outros coprocessadores.

Capítulo 7

Conclusão e Trabalhos Futuros

O presente trabalho apresentou: (i) uma proposta de método de geração de bitstreams parciais baseada no fluxo de Projeto Modular; (ii) uma ferramenta para automatizar a aplicação do fluxo de Projeto Modular e, (iii) diversos estudos de caso, inclusive um estudo de caso de complexidade maior: processador R8R, gerados pelo fluxo do Projeto Modular.

Pode-se compreender, neste trabalho, o cenário atual do desenvolvimento de SDRs. Ficou evidenciada a falta de ferramentas para o projeto e suporte à implementação de sistemas desta natureza. Embora diversos estudos sejam realizados enfatizando o uso de reconfiguração parcial e dinâmica, os fabricantes ainda não disponibilizam dispositivos que habilitam a sua reconfiguração dinâmica e parcial de uma forma mais facilmente integrável ao fluxo de projeto tradicional de sistemas computacionais. O fluxo de projeto convencional não se adequa corretamente ao projeto de SDRs e, na grande maioria dos casos é necessário utilizar estratégias alternativas para desenvolver SDRs, tal como o fluxo de Projeto Modular, proposto pela Xilinx e detalhado no Capítulo 4.

O uso do fluxo do Projeto Modular para desenvolvimento de SDRs foi motivado pelo desenvolvimento de circuitos reconfiguráveis em um nível de abstração maior que os níveis de abstração providos pela maioria das ferramentas usadas hoje para atingir este objetivo. A documentação provida pela Xilinx sobre a execução do fluxo mostrou-se incompleta. Muitos passos da execução do fluxo foram omitidos nesta documentação. Muitos procedimentos adicionais foram criados neste trabalho a partir de extensa pesquisa. Para automatizar a execução do fluxo, foi desenvolvida uma ferramenta que gera automaticamente *bitstreams* totais e parciais de um dado sistema reconfigurável.

No desenvolvimento de circuitos, quando se emprega a reconfiguração, um grau de liberdade a mais é inserido no projeto e no emprego de SoC, pois núcleos IPs podem ser desenvolvidos em diversas versões, ao invés de serem parametrizados. Porém, na maioria das aplicações embarcadas o interesse é custo final baixo, o que não é razoável de conseguir com o uso de hardware reconfigurável.

Neste trabalho, foram apresentados estudos de caso de circuitos reconfiguráveis, mostrando

dificuldades encontradas na execução do fluxo, resolução de tais dificuldades e análise dos resultados. Com a implementação de estudos de caso desenvolvidos usando o fluxo do Projeto Modular, foi possível tornar factível a reconfiguração parcial e dinâmica, possibilitando o desenvolvimento de produtos comerciais.

Desenvolveu-se um estudo de caso maior (processador R8R) que apresenta resultados satisfatórios que viabilizam o uso da reconfigurabilidade. Os coprocessadores reconfiguráveis conectados ao processador R8R apresentam maior desempenho que implementações em software, desde que o coprocessador seja usado um certo número de vezes antes de empreender uma nova reconfiguração da área do coprocessador. Também foi observado que há um compromisso entre o tamanho do *bitstream* parcial e o tempo de reconfiguração. Quanto menor a área que ocupa o *bitstream* parcial, menor o tempo de reconfiguração.

A reconfiguração parcial e dinâmica aplicada nos estudos de caso foi executada através de cabo USB, tornando o processo de reconfiguração lento. Teoricamente, o modo mais rápido de reconfiguração seria a utilização do módulo ICAP, o qual permitiria o acesso a porta de configuração do dispositivo reconfigurável. Porém, segundo Carvalho [CAR04], a documentação precária e a falta de suporte da empresa de FPGAs Xilinx impossibilitou o uso do componente ICAP.

É possível, para um conjunto de aplicações restrito, que se possa encontrar um ponto viável nesta comparação entre o hardware reconfigurável e estático. Para desenvolvimento de SDRs para aplicações com potencial comercial, a redução dos tempos de reconfiguração e o uso de mais de uma área reconfigurável, para que seja possível paralelizar várias reconfigurações, diminuindo o tempo total de reconfiguração, são imprescindíveis. Exemplos de tais aplicações comerciais são filtros de imagem, os quais podem realizar operações que podem estar implementados em módulos reconfiguráveis.

A contribuição científica deste trabalho pode ser resumida em quatro aspectos: (i) domínio do fluxo do projeto proposto pela XAPP290 [LIM03]; (ii) proposta da parte de uma infra-estrutura de reconfiguração para desenvolvimento de SDRs para mostrar a real possibilidade de se utilizar estes sistemas desenvolvidos por essa infra-estrutura para estudos de casos reais; (iii) ferramenta para automatização do fluxo para geração de SDRs; e (iv) tutorial extenso [BRI03] que complementa a documentação da Xilinx XAPP290 [LIM03], resumidamente descrito no Capítulo 4.

Como trabalhos futuros, citam-se:

1. Conclusão da prototipação da R8R com duas áreas reconfiguráveis e realizar análises de desempenho;

2. Desenvolvimento de mais coprocessadores reconfiguráveis para o processador R8R. Estes devem executar processamento exaustivo para comprovar o uso de reconfiguração em relação a circuitos estáticos;

3. Acréscimo de um módulo que insira automaticamente bus macros na ferramenta *MDLauncher* sem que o projetista precise explicitá-las no código-fonte;

4. Desenvolvimento de um mecanismo para posicionar estrategicamente os componentes no FPGA tais como LUTs, DCMs e buffers objetivando evitar erros de roteamento explicados na Seção 4.3;

5. Adaptação do fluxo do Projeto Modular para o desenvolvimento de SDRs com duas ou mais áreas reconfiguráveis;

6. Implementação de estudos de casos mais complexos, como por exemplo: vários processadores reconfiguráveis acoplados a um sistema de interconexão (barramentos, NoCs (*Network-on-Chip*)).

O trabalho realizado contribui também de forma estratégica, para o desenvolvimento e a implementação de SDRs. Através dele, pôde-se melhor entender o processo de criação de arquivos de configuração parciais, cruciais para o desenvolvimento de SDRs.

Referências Bibliográficas

[ACT04] ACTEL CORP. Aerospace & HiRel. 2004. (Capturado em: http://www.actel.com/products/aero/index.html, janeiro 2004).

[ALT02] ALTERA CORP. Avalon Bus Specification. 2002. (Reference Manual. Document Version 1.2).

[ALT04] ALTERA CORP. Nios Embedded Processor System Development. 2004. (Capturado em: http://www.altera.com/products/ip/processors/nios/nio-index.html, janeiro 2004).

[ARM04] ARM CORP. AMBA 2.0 Specification. 2004. (Capturado em: http://www.arm.com/products/solutions/AMBA_Spec.html, janeiro 2004).

[ARN92] ARNOLD, J.; BUELL, D.; DAVIS, E. Splash 2. In: 4th Annual ACM Symposium on Parallel Algorithms and Architectures, 1992. **Anais...** 1992. p.316–322.

[ARN93] ARNOLD, J.; BUELL, D.; HOANG, D.; PRYOR, D.; SHIRAZI, N.; THISTLE, M. The Splash 2 Processor and Applications. In: IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'93), 1993. **Anais...** 1993. p.482–485.

[ATM04] ATMEL CORP. AT40K Series Configuration. 2004. (Application Note. 41p. Capturado em: http://www.atmel.com/dyn/resources/prod_documents/DOC1009.PDF, janeiro 2004).

[BEN02] BENINI, L.; DE MICHELI, G. Networks on Chips: A New SoC Paradigm. **Computer**, v.35, n.1, p.70–78, 2002.

[BER00] BERGAMASCHI, R.; LEE, W. Designing System-on-Chip Using Cores. In: 37th Design Automation Conference (DAC), 2000. **Anais...** 2000. p.420–425.

[BEZ00] BEZERRA, E.; VARGAS, F.; GOUGH, M. Merging BIST and Configurable Computing Technology to Improve Availability in Space Applications. In: IEEE Latin American Test Workshop (LATW'00), 2000. **Anais...** 2000. p.146–151.

[BEZ01] BEZERRA, E.; VARGAS, F.; GOUGH, M. Improving Reconfigurable Systems Reliability by Combining Periodical Test and Redundancy Techniques: a case study. **Journal of Electronic Testing: Theory and Applications - JETTA**, v.17, n.3, p.701–711, 2001.

[BRI02] BRIÃO, E. Configuração, Reconfiguração Total e Reconfiguração Parcial de Hardware sobre a plataforma Memec-Insight V2MB1000. 2002. (Trabalho Individual II - Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN - PUCRS, Porto Alegre, RS, Brasil. 74p.).

[BRI03] BRIÃO, E. Tutoriais sobre Reconfiguração Parcial e Dinâmica Usando o Fluxo do Projeto Modular sobre a Plataforma Insight V2MB1000. 2003. (Relatório Técnico Num. TR033 - Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN - PUCRS, Porto Alegre, RS, Brasil. 93p. Capturado em: http://www.inf.pucrs.br/tr/tr033.pdf, dezembro 2003).

[CAL98] CALAZANS, N. **Projeto Lógico Automatizado de Sistemas Digitais Seqüenciais.** XI Escola de Computação. Universidade Federal do Rio de Janeiro - UFRJ, 1998. (342 p.).

[CAR00] CARMICHAEL, C.; CAFFREY, M.; SALAZAR, A. Correcting Single-Event Upsets Through Virtex Partial Configuration. 2000. (Application Note XAPP216. 12p. Capturado em: http://www.xilinx.com/bvdocs/appnotes/xapp216.pdf, julho 2000).

[CAR04] CARVALHO, E. RSCM - Controlador de Configurações para Sistemas de Hardware Reconfigurável. 2004. (Dissertação de Mestrado - Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN - PUCRS, Porto Alegre, RS, Brasil. 152p.).

[COM02] COMPTON, K.; LI, Z.; COOLEY, J.; KNOL, S.; HAUCK, S. Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing. **ACM Computing Surveys (CSUR)**, v.34, n.2, p.171–210, 2002.

[DEH00] DEHON, A.; CASPI, E.; CHU, M.; HUANG, R.; YEH, J.; MARKOVSKY, Y.; WAWRZYNEK, J. Stream Computations Organized for Reconfigurable Execution (SCORE): extended abstract. In: 10th International Conference on Field Programmable Logic and Applications (FPL'00), 2000. **Anais...** 2000. p.605–614.

[ECK04] ECK, V.; KAIRA, P.; LEBIANC, R.; MCMANUS, J. In-Circuit Partial Reconfiguration of RocketIO Attributes. 2004. (Application Note XAPP662. 29p. Capturado em: http://www.xilinx.com/bvdocs/appnotes/xapp662.pdf, janeiro 2004).

[EST63] ESTRIN, G. Parallel Processing in a Restructurable Computer System. **IEEE Transactions on Electronic Computers**, v.EC–12, n.5, p.747–755, 1963.

[GUC99] GUCCIONE, S; LEVI, D.; SUNDARARAJAN, P. JBits: A Java-based Interface for Reconfigurable Computing. In: Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD'99), 1999. **Anais...** 1999.

[HAU98] HAUCK, S. The Future of Reconfigurable Systems. In: 5th Canadian Conference on Field Programmable Devices (FPD'98), 1998. **Anais...** 1998.

[HOR02] HORTA, E.; LOCKWOOD, J.; TAYLOR, D.; PARLOUR, D. Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration. In: 39th Design Automation Conference (DAC'02), 2002. **Anais...** 2002. p.343–348.

[HOR01] HORTA, E.; LOCKWOOD, J. PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). 2001. (Relatório Técnico WUCS-01-13. University of Washington. 39p. Capturado em: http://www.arl.wustl.edu/arl/projects/fpx/references/wucs-01-13.pdf, julho 2002).

[IBM02] IBM CORP. The IBM PowerPC 440GP System-on-Chip. 2002. (Capturado em: http://www-306.ibm.com/chips/micronews/vol6_no4/MN_vol6_no4_fnl.pdf, abril 2002).

[IBM02a] IBM CORP. The CoreConnect Bus Architecture. 2002. (Capturado em: http://www.chips.ibm.com/products/coreconnect, novembro 2002).

[JAC01] JACOME M.; PEIXOTO, H. A Survey of Digital Design Reuse. **IEEE, Design & Test of Computers**, v.18, n.3, p.98–107, 2001.

[JUN01] JUNEIDI, Z.; TORKI, K.; NICOLESCU, G.; COURTOIS, B.; JERRAYA, A. Global Modeling and Simulation of System-on-Chip embedding MEMS devices. In: 4th International Conference on ASIC (ASICON'01), 2001. **Anais...** 2001. p.666–669.

[LIM03] LIM, D.; PEATTIE, M. Two Flows for Partial Reconfigurations:module based or small bit manipulations. 2003. (Application Note XAPP290. 28p. Capturado em: http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf, dezembro 2003).

[MAD97] MADISETTI, K.; SHEN, L. Interface Design for Core-Based Systems. **IEEE Design and Test of Computers**, v.14, n.4, p.45–51, 1997.

[MAR02] MARCON, C. Análise do Particionamento e Escalonamento de Recursos para o Projeto Integrado de Sistemas de Hardware/Software. White Paper. 2002.

[MAR99] MARSHALL, A.; STANSFIELD, T.; KOSTARNOV, I.; VUILLEMIN, J.; HUTCHINGS, B. A Reconfigurable Arithmetic Array for Multimedia Applications. In: 1999 ACM/SIGDA 17th International Symposium on Field Programmable Gate Arrays (FPGA'99), 1999. **Anais...** 1999. p.135–143.

[MAR01] MARTIN, G.; CHANG, H. Tutorial - System on Chip Design. In: 9th International Symposium on Integrated Circuits, Devices & Systems (ISIC'01), Tutorial 2, 2001. **Anais...** 2001.

[MCM99] MCMILLAN, S.; GUCCIONE, S. Partial Run-time Reconfiguration Using JRTR. In: 9th International Conference on Field-Programmable Logic and Applications (FPL'99), 1999. **Anais...** 1999. p.352–360.

[MES02] MESQUITA, D. Contribuições para Reconfiguração Parcial, Remota e Dinâmica de FPGAs. 2002. (Dissertação de Mestrado - Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN - PUCRS, Porto Alegre, RS, Brasil. 121p.).

[MÖLL03] MÖLLER, L. Ferramentas de Reconfiguração Parcial, Remota e Dinâmica de FPGAs Virtex. 2003. (Relatório Técnico Num. TR035 - Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN - PUCRS, Porto Alegre, RS, Brasil. 29p. Capturado em http://www.inf.pucrs.br/tr/tr035.pdf, novembro 2003).

[MOR03] MORAES, F.; CALAZANS, N. R8 Processor - Architecture and Organization Specification and Design Guidelines. 2003. (Capturado em: http://www.inf.pucrs.br/~gaph/Projects/R8/public/R8_arq_spec_eng.pdf, agosto 2003).

[MOR03a] MORAES, F.; MELLO, A.; MöLLER, L.; OST, L.; CALAZANS, N. A Low Area Overhead Packet-switched Network on Chip: Architecture and Prototyping. In: IFIP International Conference on Very Large Scale Integration (VLSI-SoC 2003), 2003. **Anais...** 2003. p.318–323.

[MOR03b] MORENO, E. Modelando, Descrevendo e Validando NoCs para SoCs em Diferentes Níveis de Abstração. 2003. (Seminário de Andamento - Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN - Porto Alegre, RS, Brasil).

[OCP02] OCP International Partnership. The Importance of Socket in SOC Design. White Paper. 2002. (10p. Capturado em: http://www.ocpip.org/data/sockets_socdesign.pdf, novembro 2002).

[OCP03] OCP International Partnership. Open Core Protocol Specification. Release 2.0. 2003. (210p. Capturado em: http://www.ocpip.org/, novembro 2003).

[OST02] OST, L. Desenvolvimento de Interfaces para Núcleos IP Baseado em OCP. 2002. (Trabalho Individual II - Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN - PUCRS, Porto Alegre, RS, Brasil. 51p.).

[PAL02] PALMA, J. Métodos para desenvolvimento e distribuição de IP-cores. 2002. (Dissertação de Mestrado - Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN - PUCRS, Porto Alegre, RS, Brasil. 111p.).

[PAT98] PATTERSON D.; HENNESSY, J. **Computer Organization and Design**. San Francisco, CA Morgan Kaufmann, 1998. (551p.).

[RAG02] RAGHAVAN, A.; SUTTON, P. JPG - A Partial Bitstream Generation Tool to Support Partial Reconfiguration in Virtex FPGAs. In: Parallel and Distributed Processing Symposium (IPDPS'02), 2002. **Anais...** 2002. p.155–160.

[RIN99] RINCON, A.; LEE, W.; SLATTERY, M. The Changing Landscape of System-on-a-Chip Design. In: 1999 IEEE Custom Integrated Circuits Conference (CICC'99). Invited Paper, 1999. **Anais...** 1999.

[SAN99] SANCHEZ, E.; SIPPER, M.; HAENNI, J.; BEUCHAT, J.; STAUFFER, A.; PEREZ-URIBE, A. Static and Dynamic Configurable Systems. **IEEE Transactions on Computers**, v.48, p.556–564, 1999.

[SHI98] SHIRAZI, N.; LUK, W.; CHEUNG, P. Run-Time Management of Dynamically Reconfigurable Designs. In: 8th International Conference on Field-Programmable Logic and Applications (FPL'98), 1998. **Anais...** 1998.

[SIL01] SILICORE CORP. WISHBONE - System-on-chip: Interconnection Architecture for Portable IP Cores. 2001. (Revision B.2).

[SIL03] SILICORE CORP. Wishbone Frequently Asked Questions. 2003. (Capturado em: http://www.silicore.net/wishfaq.htm, janeiro 2003).

[TEN03] TENSILICA INC. True Application-Specific Embedded Processors Now a Reality For System-on-Chip IC Designs. 2003. (Capturado em: http://www.tensilica.com/html/pr_1999_02_15.html, janeiro 2003).

[VIL97] VILLASENOR, J.; MANGIONE-SMITH, W. Configurable Computing. **Scientific American**, v.19, p.66–71, 1997.

[VIS03] VISSERS, K. Parallel Processing Architectures for Reconfigurable Systems. In: 6th Design, Automation and Test in Europe Conference and Exhibition (DATE'03), 2003. **Anais...** 2003. p.396–397.

[VSI97] VSI ALLIANCE. Architecture Document. 1997. (Version 1.0. 69p. Capturado em: http://www.vsi.org/, janeiro 2002).

[WAL02] WALSTROM, J.; COOK, J.; GOTTLIEB, D.; FERRERA, S.; WANG, C.; CARTER, N. The Design of the Amalgam Reconfigurable Cluster. In: 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02), 2002. **Anais...** 2002. p.309–310.

[WIR98] WIRTHLIN, M.; HUTCHINGS, B. Improving Functional Density Through Run-Time Constant Propagation. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v.6, n.2, p.247–256, 1998.

[XIL00] XILINX, INC. Virtex Series Configuration Architeture User Guide. 2000. (Application Note XAPP151. 45p. Capturado em: http://www.xilinx.com/xapp/xapp151.pdf, janeiro 2000).

[XIL01] XILINX, Inc. Aerospace and Defense Products. 2001. (Capturado em: http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=aero_overview, março 2003).

[XIL01a] XILINX, INC. Virtex-II Platform FPGA Handbook. 2001. (Version 1.3. 490p. Capturado em: http://direct.xilinx.com/bvdocs/userguides/ug002.pdf, dezembro 2001).

[XIL01b] XILINX, INC. ISE 4.0 - Development System Reference Guide. 2001. (536p. Capturado em: http://toolbox.xilinx.com/docsan/xilinx4/pdf/docs/dev/dev.pdf, abril 2003).

[XIL02] XILINX, INC. Virtex-II 1.5V: Field-Programmable Gate Array. Advance Product Specification. 2002.

[XIL02a] XILINX, INC. Virtex-II User Guide Manual. 2002.

[XIL02] XILINX, INC. Virtex-II Platform FPGAs: Introduction and Overview. 2002. (Advance Product Specification DS031-1 (v1.9). Capturado em: http://www.xilinx.com/partinfo/ds031.pdf, outubro 2003).

[XIL03] XILINX, Inc. Virtex-II Pro^{TM} Platform FPGAs: Introduction and Overview. 2003. (Document DS110-1 (v2.4.1). Capturado em: http://direct.xilinx.com/bvdocs/publications/ds110-1.pdf, outubro 2003).

[ZEF03] ZEFERINO, C.; SUSIN, A. SoCIN: A Parametric and Scalable Network-on-Chip. In: 16th Symposium on Integrated Circuits and Systems Design (SBCCI'03), 2003. **Anais...** 2003. p.169–174.

[ZHA00] ZHANG, X.; NG, K. A review of high-level synthesis for dynamically reconfigurable FPGAs. **Microprocessors and Microsystems**, v.24, p.199–211, 2000.

Apêndice A

Códigos-fonte para o Processador R8R

A.1 Software para teste de todos os coprocessadores reconfiguráveis

Nesta Seção, é apresentado o código-fonte do software usado para testar todos os coprocessadores reconfiguráveis no processador R8R.

```
1   .CODE
2           xor r0,r0,r0
3           ldl r1, #01h
4           ldh r1, #00h
5           ldl r2, #86h
6           ldh r2, #00h
7           ldl r3, #02h
8           ldh r3, #00h
9           ldl r4, #AAh
10          ldh r4, #00h
11          ldl r5, #03h
12          ldh r5, #00h
13          ldl r6, #F7h
14          ldh r6, #00h
15          ldl r7, #04h
16          ldh r7, #00h
17          ldl r8, #0Bh
18          ldh r8, #01h
19
20          ldl r15,#4avg
21          ldh r15,#4avg
22
23          initr #1
24          wrr r1,r2
25          wrr r3,r4
26          wrr r5,r6
27          wrr r7,r8
28          rdr r11,r12
29          st r15,r12,r0
30
31          ldl r15, #2avg
```

```
32          ldh r15, #2avg
33          selr #2
34          initr #2
35          wrr r1,r2
36          wrr r3,r4
37          rdr r11,r12
38          st r15,r12,r0
39
40          ldl r1, #FEh
41          ldh r1, #00h
42
43          ldl r2, #0Fh
44          ldh r2, #00h
45
46          ldl r3, #E4h
47          ldh r3, #0Eh
48
49          ldl r4, #84h
50          ldh r4, #03h
51
52          ldl r10, #produto
53          ldh r10, #produto
54
55          ldl r11, #divisao
56          ldh r11, #divisao
57
58          ldl r12, #resto
59          ldh r12, #resto
60
61          ldl r13, #raizq
62          ldh r13, #raizq
63
64          ldl r15, #01h
65          ldh r15, #00h
66
67          selr #3
68          initr #3
69          wrr r1,r2
70          rdr r5,r6  ; r6 é o produto
71
72          st r6,r10,r0 ; pmem(r10+r0) <- r6
73
74          selr #4
75          initr #4
76          wrr r3,r2   ;  EE4h div FEh
77          rdr r5,r6   ; r5 = resto  r6 = quociente
78
79          st r5, r12, r0
80          st r6, r11, r0
81
82          selr #5
83          initr #5
84          wrr r15,r4
85          rdr r5,r6
86
```

```
87          st r6,r13,r0
88
89          halt
90
91  .ORG #024H
92
93  .DATA
94  avg4:     DB #0000H
95  avg2:     DB #0000H
96  produto:  DB #0000H
97  divisao:  DB #0000H
98  resto:  DB #0000H
99  raizq:  DB #0000H
100 .ENDDATA
```

A.2 Multiplicação em software

Aqui é apresentado o código-fonte do software que realiza operações de multiplicação.

```
;By Eduardo Wenzel Brião
.CODE

      xor r14,r14,r14
      ldl r1,#02h ; load multiplicand into r1
      ldh r1,#00h

      ldl r2,#02h ; load multiplier into r2
      ldh r2,#00h

      ldl r15,#01h ; load constant "1" into r15
      ldh r15,#00h ;

      ldl r3,#00h ;
      ldh r3,#00h ;

loop:
      and r0,r2,r15   ; r0 <- r2 and r15 (1)
      or r7,r2,r14    ; if r2 = 0?:
      jmpzd #finish   ; then go to #finish
      or r7,r0,r14    ; r0 <- r0 or r14 (0)
      jmpzd #loopend  ; if r0 = 0 then go to #loopend
      add r6,r3,r14   ; r6 <- r3
      add r6,r1,r6    ; r6 <- r1 + r6
      add r3,r6,r14   ; r3 <- r6


loopend:
      sr0 r2,r2     ;  r2 << 1
      sl0 r1,r1     ;  r1 >> 1
      jmpd #loop

finish:
      halt


.ORG #015H
.DATA
 product:  DB #0000H
.ENDDATA
```

A.3 Divisão em software

Aqui é apresentado o código-fonte do software que realiza operações de divisão.

```
1  ;By Daniel Camozzato and
2  ;   Eduardo Wenzel Brião
3  .CODE
4
5
6          xor R0, R0, R0 ; R0 <- 0
7          ldl R1, #8Fh ; Quociente <- 9
8          ldl R2, #0Fh ; Div <- 4
9          ldl R4, #0Fh ; R4 <- 15
10         ldh R7, #80h ; R7 <- 1000000
11
12         and R6, R7, R1 ; R6 <- (R1 and R7)
13         jsrd #flag ; Vai para subrotina "flag"
14         sl0 R1, R1 ; Quociente <- SHL(Resto) &0
15         jsrd #carry ; Vai para subrotina "carry"
16
17
18 loop:        sub R3, R3, R2 ;  Resto <- Resto - Div
19         jmpnd #neg ; if (negativo) goto "neg"
20
21
22 pos:        and R6, R7, R1 ; R6 <- (R1 and R7)
23         jsrd #flag ; Vai para subrotina "flag"
24         sl1 R1, R1 ; SHL(quociente) &1
25         jsrd #carry ; Vai para subrotina "carry"
26         jmpd #testafim ; Vai para "testafim"
27
28
29 neg:        add R3, R3, R2 ; Resto <- Resto + Div
30         and R6, R7, R1 ; R6 <- (R1 and R7)
31         jsrd #flag ; Vai para subrotina "flag"
32         sl0 R1, R1 ; SHL(Quociente) &0
33         jsrd #carry ; Vai para subrotina "carry"
34         jmpd #testafim ; Vai para "testafim"
35
36 carry:  subi R5, #01h ; flag <- flag - 1
37         jmpzd #C ; if (zero) goto C
38
39         ; (else)
40 noC:        sl0 R3, R3 ; Resto <- ssl(Resto) &0
41         rts ; retorna
42
43 C:        sl1 R3, R3 ; Resto <- ssl(Resto) &1
44         rts ; retorna
45
46 flag:        jmpnd #setflag ; se R6 = negativo, goto "setflag"
47         xor R5, R5, R5 ; R5 <- 0
48         rts ; retorna
49
50 setflag:        xor R5, R5, R5 ; R5 <- 0
51                 ldl R5, #01h ; flag <- 1
```

```
52                  rts ; retorna
53
54  testafim:        subi R4, #01 ; R4 <- R4 - 1
55                  JMPND #fim ; if (negativo) goto "fim"
56                  JMPD #loop ; else goto "loop"
57
58  fim:        sr0 R3, R3
59          halt
60
61  .ENDCODE
```

A.4 Raiz quadrada em software

Nesta Seção, é apresentado o código-fonte do software que efetua operações de raiz quadrada.

```
; By Eduardo Wenzel Brião
.CODE
        xor r0,r0,r0
        ldl r1,#FFh
        ldh r1,#0Fh

        addi r1,#01h
        sr0 r1,r1

        ldl r2,#FFh
        ldh r2,#FFh

ciclo1:
        addi r2,#01h
        sub r1,r1,r2
        jmpnd #fim1
        jmpzd #fim1
        jmpd #ciclo1

fim1:

        ldl r7,#c1
        ldh r7,#c1
        st r2,r7,r0 ; pmem(r7+r0) <- r2 (dado do cop)
         halt

.ORG #10H
.DATA
C1:  DB #FFFFH
.ENDDATA
```

Apêndice B

Considerações quanto a Erros de Roteamento

Nas ferramentas de síntese, sinais que não são criados pelo usuário e sim gerados pelas mesmas são chamados de *Global Logic Signals* ou Sinais de Lógica Global. Porém, a ferramenta de posicionamento e roteamento não é determinística e sempre que esta ferramenta é executada, a mesma faz um roteamento baseado em tabelas de custos e heurísticas para a realização do roteamento de um determinado sinal. Em projetos de complexidade razoável, determinados sinais de *global logic* que fazem a interconexão de componentes tais como buffers e DCMs podem não ser completamente roteados, resultando em erros na ferramenta de posicionamento e roteamento, ou no comportamento na prototipação. A seguir, serão mostrados possíveis problemas que possam ocorrer utilizando DCM e a possível solução para as mesmas.

B.1 Impossibilidade da Geração de Sinais de Lógica Global

Este problema é muito comum na fase da Montagem Final do Projeto Modular. A ferramenta PAR não consegue rotear sinais de lógica global para interconectar componentes referentes a gerência do relógio, gerando a monstruosa mensagem de erro abaixo:

```
FATAL_ERROR:Guide:basgitaskphyspr.c:369:1.17.4.3:137 -
Guide encountered a Logic0 or Logic1 signal GLOBAL_LOGIC1_179
that does not have a driver or load within the module boundary.
This problem may be caused by having a constant driving the
input from outside the module boundary or because a driver or
load comp did not meet the par-guiding criteria.  The design
will not be completely placed and routed by Par-Guide. Process
will terminate.  To resolve this error, please consult the
Answers Database and other online resources at
```

```
http://support.xilinx.com. If you need further assistance,
please open a Webcase by clicking on the "WebCase" link at
http://support.xilinx.com
```

Para a resolução deste problema, deve-se instanciar um DCM, instaciar um componente chamado BUFGMUX ao invés de um BUFG, interconectar estes componentes via código, e finalmente conectar pinos não-usados do DCM com LUTs que armazenam constante '0'. Abaixo é mostrado o código-fonte do top com as medidas tomadas para a resolução deste problema:

Declaração dos componentes envolvidos:

```
component LUT1 is
generic(INIT: bit_vector:= X"3");
  port(O: out std_ulogic;
       I0: in std_ulogic);
end component;

component IBUFG
 is port (
     I:in std_logic;
     O: out std_logic);
 end component;

 component BUFGMUX is
 port (O : out STD_ULOGIC;
      I0 : in STD_ULOGIC;
      I1 : in STD_ULOGIC;
      S : in STD_ULOGIC);
 end component;

 component DCM is
 port (CLK0 : out STD_ULOGIC;
       CLKFB : in STD_ULOGIC;
       CLKIN : in STD_ULOGIC;
       DSSEN : in STD_ULOGIC;
       PSCLK : in STD_ULOGIC;
       PSEN : in STD_ULOGIC;
       PSINCDEC: in STD_ULOGIC;
```

```
      RST : in STD_ULOGIC);
 end component;
```

Declaração dos sinais:

```
signal dll_clk_in : std_logic;
signal dll_clk_out : std_logic;
signal reset_dll : std_logic;
signal clk_top:     std_logic;
signal clk_top_in: std_logic;

signal ps_sig: std_logic;



  reset_dll <= not reset; --reset é ativado em '1';


  -- buffer de entrada
  ibuf_dll:  IBUFG port map (I => clock, O => dll_clk_in);


  --LUT para gerar constante 0
  DCM_LUT: LUT1 generic map (INIT => b"00")
   port map (O => ps_sig, I0 => ps_sig);

  -- Digital Clock Manager
  dll_1: DCM port map (CLKIN => dll_clk_in,
       CLKFB => clk_top_in,
       CLK0 => dll_clk_out,
      RST => reset_dll,
      PSCLK => PS_sig,
      PSEN => PS_sig,
      DSSEN => PS_sig,
      PSINCDEC => PS_sig );
```

```
  -- BUFGGMUX ao inves de BUFG.
  global_clk : BUFGMUX port map (O => clk_top_in,
         I0 => dll_clk_out,
         I1 => dll_clk_out,
         S => ps_sig);

clk_top <= clk_top_in;
```

O código apresentado é análogo no esquemático da Figura B.1. A ferramenta de posicionamento e roteamento tenta rotear sinais *global logic* que serão conectados nos pinos PSCLK, PSEN, DSSEN e PSINCDEC no DCM. Então neste ponto é que ocorre o erro. Então foi instanciada uma LUT que armazena a constante '0' conectada ao DCM pelo sinal *ps_ signal* aos pinos mostrados anteriormente do DCM. Desta maneira a ferramenta de síntese não necessita criar sinais globais, o que possam causar problemas no roteamento.

No arquivo UCF é recomendável inserir as linhas abaixo:

```
1. INST "global_clk" LOC = "BUFGMUX7P";
2. INST "DCM_LUT" LOC = "SLICE_X27Y79";
3. INST "DCM_LUT" LOCK_PINS;
4. INST "dll_1" LOC = "DCM_X0Y1";
```

A linha 1 do UCF posiciona o buffer-multiplexador que é utilizado para a auto-alimentação do DCM. No caso este foi posicionado na parte de cima do FPGA.

As linhas 2 e 3 posicionam a LUT que gerará a constante '0' para os sinais do DCM. Recomenda-se que esta LUT esteja mais próxima possível do DCM. Nota: esta LUT deve estar em um SLICE sem que nenhuma outra LUT esteja sendo usada, pois a diretiva LOCK_PINS não terá efeito e uma mensagem de erro referente será lançado pela ferramenta MAP ou PAR.

Finalmente, a linha 4 posiciona o DCM.

B.2 Comportamento Indesejável do Projeto: Escorregamento de Relógio

Quando o fluxo do Projeto Modular gera todos os bitstreams necessários para a execução do projeto reconfiguirável, nem sempre pode-se garantir que tal projeto vai executar corre-

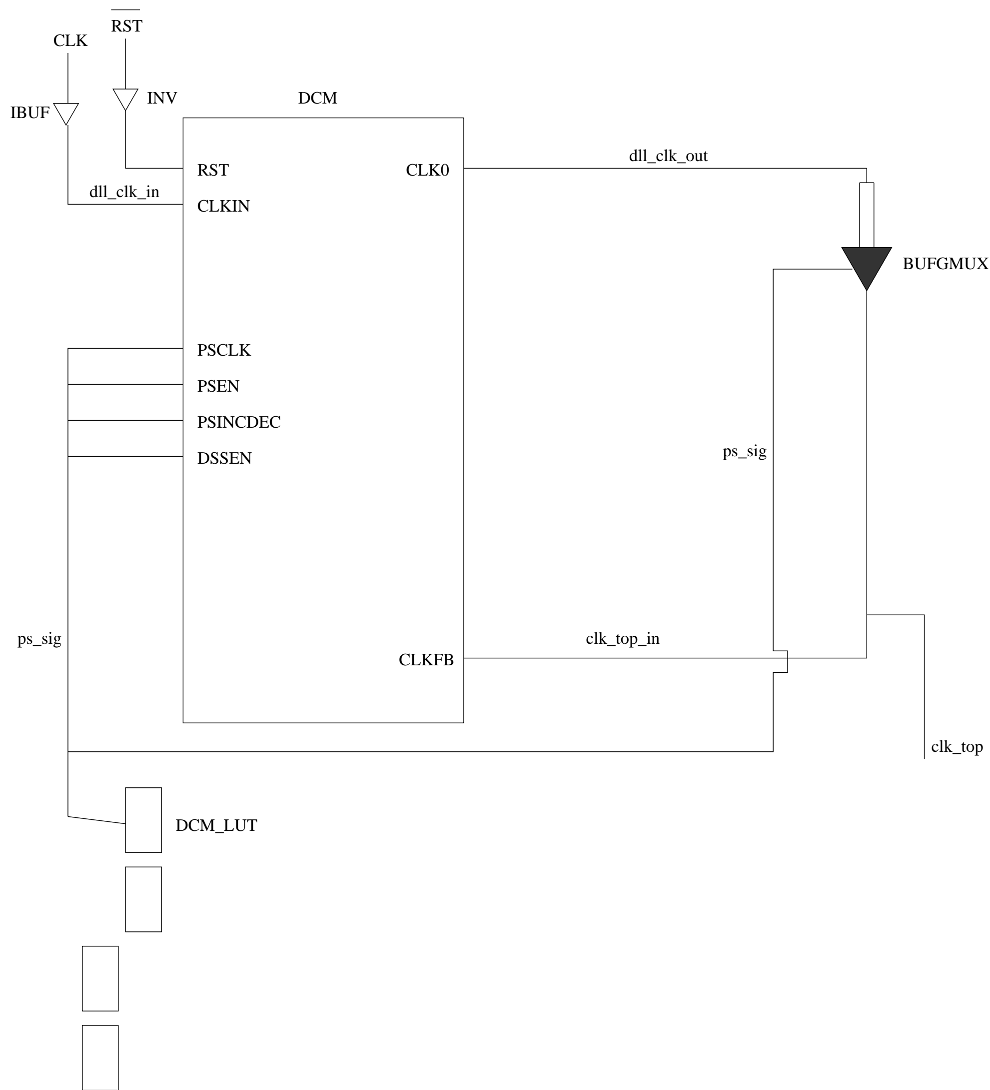

Figura B.1: Esquemático esquivalente do código VHDL apresentado para a ferramenta de posicionamento e roteamento possa realizar as conexões entre os componentes.

tamente. A Figura B.2 mostra um hipotético projeto onde o módulo fixo está agrupado à esquerda do FPGA.

Percebe-se que o DCM está situado à esquerda do FPGA junto ao móduilo fixo. Porém nota-se também que os BUFGs ou BUFGMUX (buffer e multiplexador) estão concentrados no meio do FPGA. O sinal que faz a auto-alimentação do DCM atravessa a fronteira entre o módulo fixo e reconfigurável. Por tanto, recomenda-se fortemente conectar uma bus macro para a passagem deste sinal como mostrado na Figura B.3.

Recomenda-se também que o DCM seja instanciado da mesma maneira mostrada na Seção B.1. Utiliza-se esta configuação mostrada em tal Seção e então adiciona-se a bus macro para conexão do sinal que atravessa os dois módulos (fixo e reconfigurável).

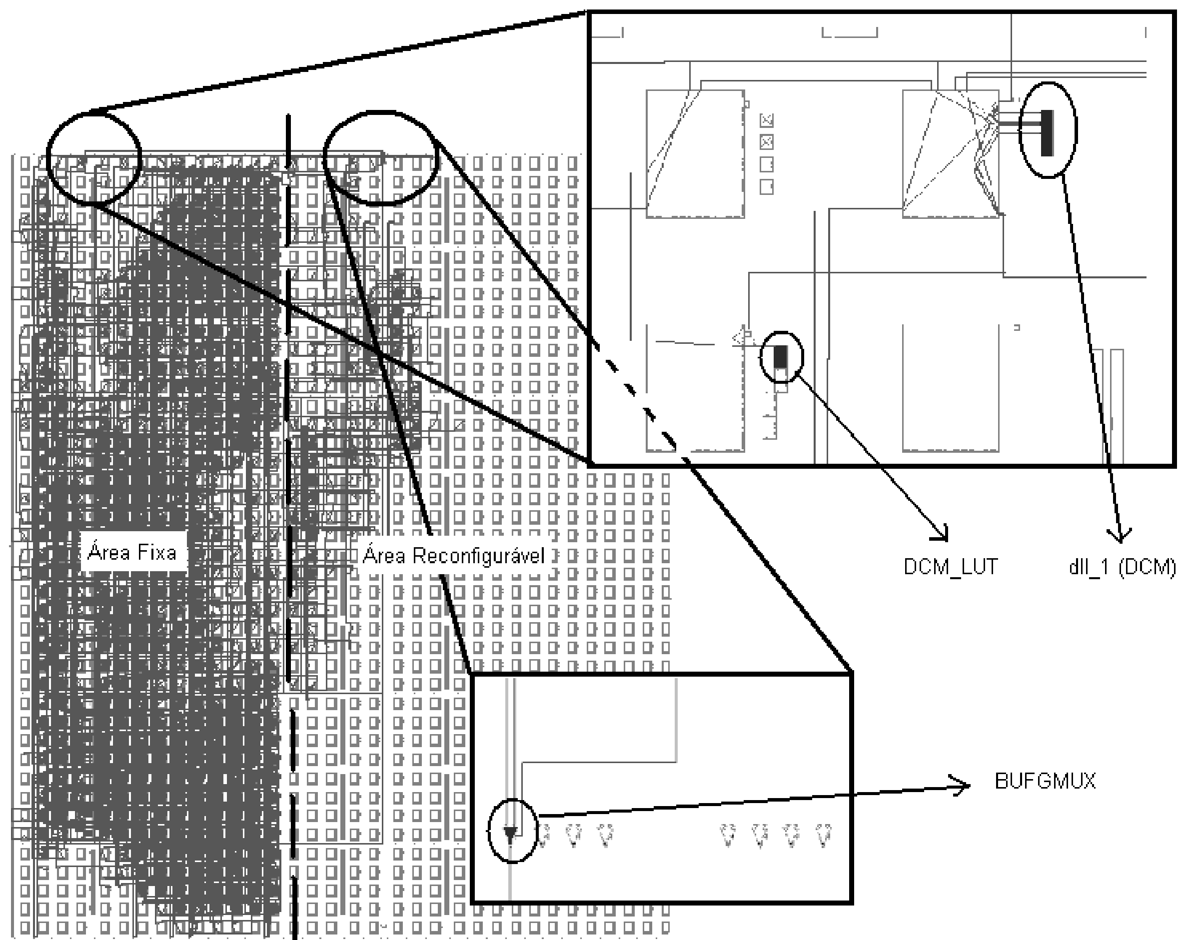

Figura B.2: Um projeto hipotético onde o módulo fixo está "espremido" à esquerda do FPGA. Os sinais que conectam os componentes DCM e BUFGMUX atravessam a o limite entre a área reconfigurável e a área fixa.

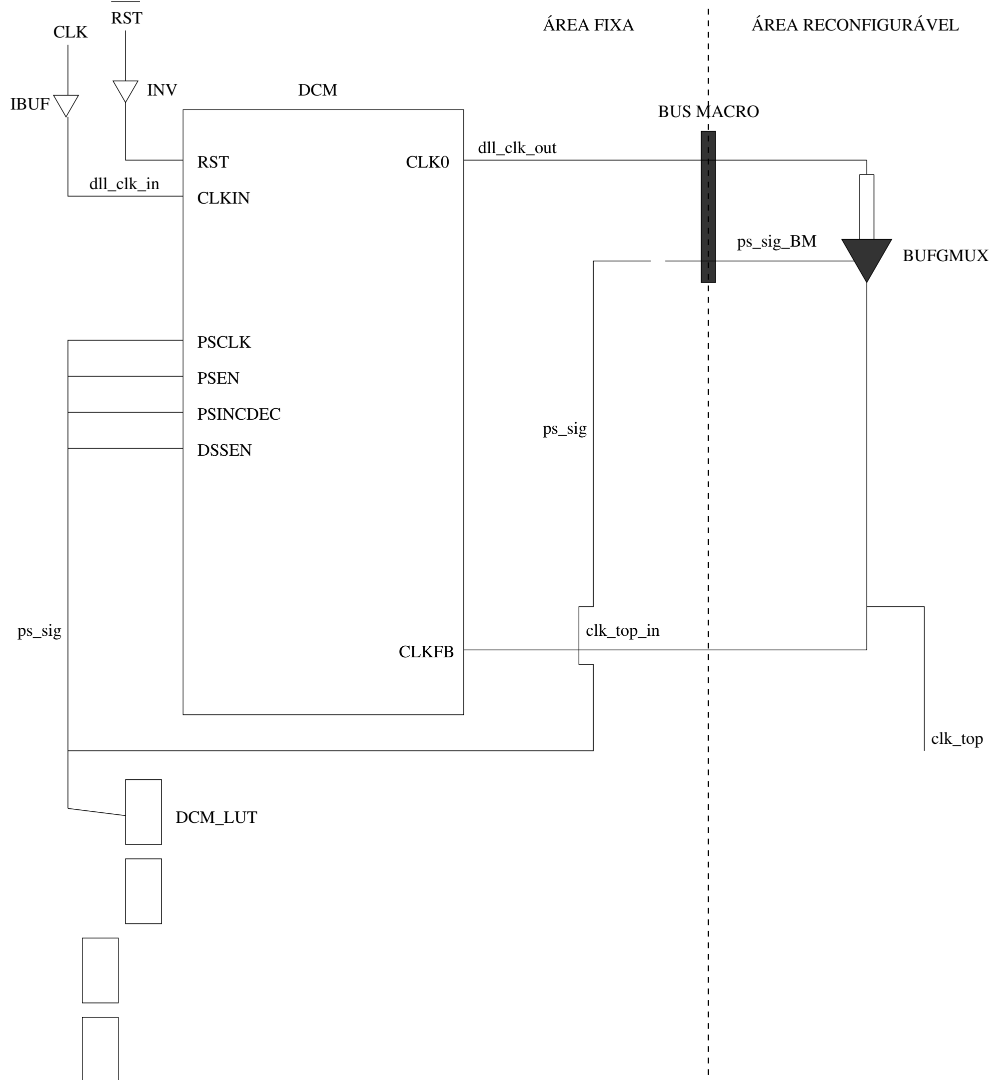

Figura B.3: Esquemático mostrando como devem ser fixadas as bus macros no FPGA para permitir que os sinais possam interconectar os componentes DCM e buffer sem comportamentos indesejáveis na prototipação.