PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FLUXOS DE PROJETO PARA RECONFIGURAÇÃO PARCIAL

Trabalho Individual II

Mestrando: Leandro Heleno Möller

Orientador: Prof. Fernando Gehm Moraes

Porto Alegre, Novembro de 2004

Sumário

Lista de Figuras

Listas de Abreviaturas

BRAM	*Block Select RAM*
CAD	*Computer Aided Design*
CLB	*Configurable Logic Block*
CPLD	*Complex Programmable Logic Device*
DCM	*Digital Clock Manager*
EPROM	*Erasable Programmable Read-Only Memory*
FPGA	*Field Programmable Gate Array*
ICAP	*Internal Configuration Access Port*
IOB	*Input Ouput Block*
LUT	*Look-Up Table*
NOC	*Network On Chip*
PLA	*Programmable Logic Array*
RAM	*Random Access Memory*
RHOS	*Reconfigurable Hardware Operating Systems*
SDR	Sistemas Dinamicamente Reconfiguráveis
SRAM	*Static Random Access Memory*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*
VLSI	*Very Large Scale Integrat*

1 INTRODUÇÃO

Dispositivos programáveis são utilizados tanto na prototipação de sistemas digitais, quanto em produtos finais. A vantagem da utilização destes dispositivos em equipamentos eletrônicos, em relação a outras tecnologias, é a capacidade de modificações do hardware, permitindo a atualização do produto e por conseqüência um tempo de vida maior do mesmo. Outra vantagem associada à utilização de dispositivos programáveis em produtos finais é a redução do tempo de chegada ao mercado, pois a fase de fabricação do circuito integrado não esta incluída no ciclo de fabricação do produto, já que o dispositivo programável é comprado pronto, necessitando apenas a configuração do dispositivo com a lógica exclusiva do produto. O ponto negativo de dispositivos programáveis, em relação a outras tecnologias disponíveis no mercado, é um maior consumo de área de silício e potência, devido ao fato de possuírem componentes adicionais que permitem a modificação do hardware.

O processo de modificar o hardware de dispositivos programáveis é denominado *reconfiguração*. Chama-se de *reconfiguração total* a modificação completa do hardware, e *reconfiguração parcial* a alteração de uma parte do dispositivo programável. Esta reconfiguração pode ser *estática*, necessitando que o sistema pare de operar para que ocorra a modificação do hardware, ou *dinâmica*, permitindo que partes do sistema continuem executando enquanto as alterações são efetuadas. *Sistemas reconfiguráveis* são sistemas eletrônicos, digitais ou analógicos, que utilizam em sua implementação dispositivos reconfiguráveis, onde a reconfiguração é uma característica importante no projeto.

As principais justificativas para a utilização de *sistemas reconfiguráveis* são o aumento do tempo de vida do produto, a miniaturização do hardware e a possibilidade de especialização de funções em produtos genéricos. O tempo de vida do produto pode ser estendido com a reconfiguração do hardware para adicionar novas funcionalidades, novos protocolos de comunicação ou corrigir defeitos do equipamento. A miniaturização do produto é obtida com uma administração inteligente do dispositivo programável, mantendo configurado apenas o hardware que está sendo atualmente utilizado. O fato de não necessitar que todo o sistema esteja em hardware no mesmo instante de tempo permite economizar energia, sendo esta uma questão importante na durabilidade e tamanho das baterias de equipamentos portáteis. A especialização de funções em produtos genéricos pode ser facilmente exemplificada. Pense em um produto genérico como sendo um computador de mão e como produto específico uma máquina fotográfica digital. Existem computadores de mão que tiram fotografias, mas a sua qualidade não se compara a uma máquina fotográfica digital, pois o computador de mão faz todo o processamento em software. Com a utilização de dispositivos programáveis o computador de mão (sistema genérico) pode ser configurado para possuir o hardware necessário de uma máquina fotográfica digital (sistema específico), assim obtendo melhor desempenho que a alternativa somente software.

Exemplos de dispositivos programáveis utilizados na indústria são EPROMs, PLAs, CPLDs e FPGAs. Entre estes se destaca o FPGA, pela sua alta densidade de elementos lógicos e rápida reprogramabilidade, quando utilizada a tecnologia SRAM para configuração. Dos FPGAs disponíveis no mercado optou-se pelos dispositivos Xilinx [XIL04a], pela sua disponibilidade de documentação e trabalhos relacionados no assunto tanto no meio acadêmico [HUB04a][DYE02a][MÖL04] quanto na indústria, ferramentas que permitem interagir nas fases intermediárias da descrição de hardware

à prototipação em FPGA, à diversidade de plataformas e à alta densidade em termos de portas lógicas dos dispositivos.

O objetivo deste trabalho é estudar fluxos de projeto que habilitem a implementação de *sistemas dinamicamente reconfiguráveis* (SDRs). A literatura diferencia SDRs em dois tipos básicos: baseado em módulos e baseado em pequenas alterações [XIL04c]. Este trabalho versa sobre a reconfiguração de módulos, buscando adicionar/substituir funcionalidades ao projeto. Ferramentas do autor deste trabalho, que permitem pequenas alterações, também sendo chamada de reconfiguração de parâmetros, podem ser encontradas em [MES03][MOR03b][MOR03b].

Para auxiliar no entendimento do processo de reconfiguração em dispositivos Virtex-II, faz-se necessário uma breve apresentação de sua arquitetura interna, conteúdo abordado na Seção 1.1. A Seção 1.2 apresenta o fluxo básico de um projeto VLSI em dispositivos FPGAs. Os passos básicos do fluxo para a implementação de sistemas reconfiguráveis é abordado na Seção 1.3. A Seção 1.4 apresenta a organização do restante do documento.

1.1 Arquitetura interna da Virtex-II

Esta Seção apresenta os componentes internos à arquitetura Virtex-II, que devem ser conhecidos para que seja possível posteriormente estudar como estes são configurados. Todas as informações contidas nesta Seção foram retiradas de [XIL04b]. A Figura 1 apresenta os principais componentes dos dispositivos Virtex-II.

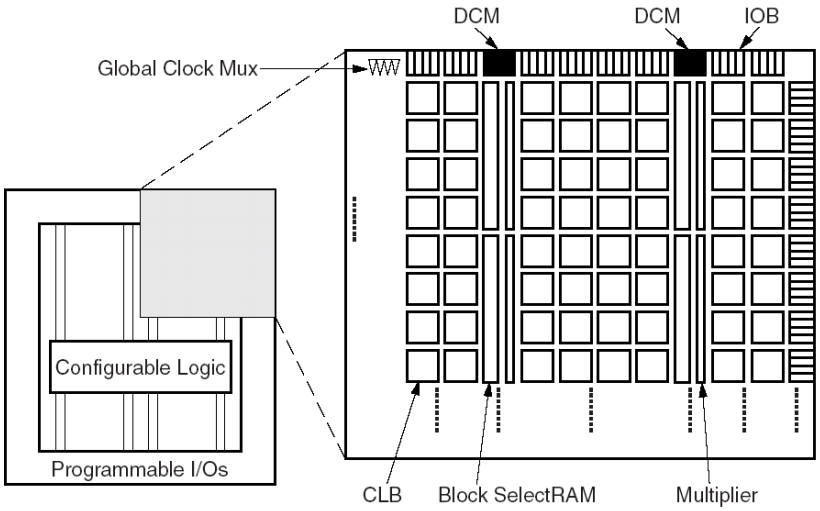


Figura 1 – Arquitetura interna do dispositivo Virtex-II.

A Virtex-II é composta basicamente por blocos lógicos configuráveis (CLBs – *Configurable Logic Blocks*), blocos de entrada e saída (IOBs – *Input/Output Blocks*), blocos de memória (BRAMs - *Block Select RAMs*), gerenciadores de escorregamento de relógios (DCMs – *Digital Clock Manager*), multiplicadores e recursos de roteamento que interconectam todos estes componentes. As CLBs provêm elementos funcionais para a implementação de lógica síncrona e combinacional. Cada CLB possui quatro slices e dois *tri-states*. Cada slice possui dois geradores de função de quatro entradas, 2 flip-flops para armazenamento de um bit cada e outros recursos, apresentados na Figura 2, como propagação rápida de *carry*. Cada BRAM possui 18Kb de memória dupla porta, programável em diversas larguras de dados e endereços. A Tabela 1 apresenta a quantidade desses e outros

recursos existentes em cada dispositivo da família Virtex-II.

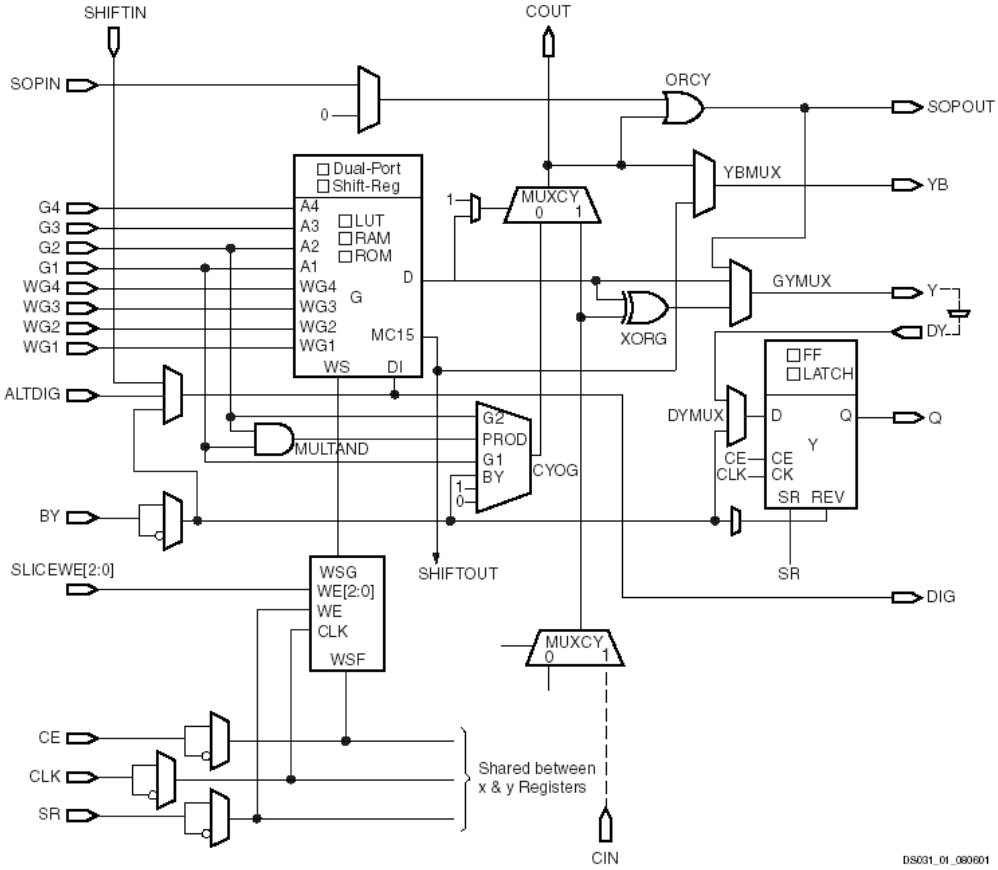


Figura 2 – Metade superior de um slice da Virtex-II.

Tabela 1 – Número de recursos internos de cada dispositivo da família Virtex-II.

Dispositivo	Quantidade de Portas Lógicas Equivalentes	1 CLB = 4 slices = Máximo 128 bits			Blocos Multiplicadores	BRAMs		DCMs	Número Máximo de pinos de I/O
		Matriz linha x coluna	Slices	Máximo de RAM distribuído (Kbits)		Blocos de 18 Kbits	Máximo de RAM (Kbits)		
XC2V40	40K	8 x 8	256	8	4	4	72	4	88
XC2V80	80K	16 x 8	512	16	8	8	144	4	120
XC2V250	250K	24 x 16	1.536	48	24	24	432	8	200
XC2V500	500K	32 x 24	3.072	96	32	32	576	8	264
XC2V1000	1M	40 x 32	5.120	160	40	40	720	8	432
XC2V1500	1,5M	48 x 40	7.680	240	48	48	864	8	528
XC2V2000	2M	56 x 48	10.752	336	56	56	1.008	8	624
XC2V3000	3M	64 x 56	14.336	448	96	96	1.728	12	720
XC2V4000	4M	80 x 72	23.040	720	120	120	2.160	12	912
XC2V6000	6M	96 x 88	33.792	1.056	144	144	2.592	12	1.104
XC2V8000	8M	112 x 104	46.592	1.456	168	168	3.024	12	1.108

Os dispositivos Virtex são organizados como uma matriz bidimensional de bits. No entanto, o dispositivo só pode ser configurado unidimensionalmente em colunas, onde cada coluna é composta por frames. O frame é a menor unidade que pode ser lida ou escrita do FPGA. Quanto menor o número de linhas de CLBs do FPGA, menor é o tamanho do frame e conseqüentemente mais rápido é o tempo de reconfiguração de um frame. A Tabela 2 apresenta o número de bits de um frame dos dispositivos apresentados na Tabela 1, assim como o tempo de reconfiguração de um bitstream total para diversos modos de configuração.

Tabela 2 – Informações para a reconfiguração de dispositivos Virtex-II.

Dispositivo	Número de Frames	Tamanho do Frame em bits	Bits de Configuração	Total de bits (incluindo cabeçalho)	Tempo de download SelectMap (50 MHz) em ms	Tempo de download Serial (50 MHz) em ms	Tempo de download JTAG (33 MHz) em ms
XC2V40	404	832	360.096	339.040	0.84	6.72	10.19
XC2V80	404	1472	635.296	598.880	1.49	11.89	18.02
XC2V250	752	2112	1.697.184	1.593.696	3.97	31.76	48.13
XC2V500	928	2752	2.761.888	2.560.608	6.38	51.08	77.39
XC2V1000	1104	3392	4.082.592	3.752.800	9.36	74.90	113.48
XC2V1500	1280	4032	5.659.296	5.170.272	12.90	103.22	156.39
XC2V2000	1456	4672	7.492.000	6.813.024	17.01	136.05	206.13
XC2V3000	1804	5312	10.494.368	9.594.720	23.96	191.66	290.39
XC2V4000	2156	6592	15.659.936	14.226.784	35.53	284.25	430.68
XC2V6000	2508	7872	21.849.504	19.759.968	49.36	394.86	598.27
XC2V8000	2860	9152	29.063.072	26.194.272	65.44	523.49	793.17

Tabela 3 – Significado dos pinos de entrada e saída do ICAP.

Porta ICAP	Descrição
I [0:7]	Barramento de entrada de dados por onde é feito uma configuração. O pino I0 é o bit mais significativo.
O [0:7]	Barramento de saída de dados por onde é feito readback e por onde saem informações de estado durante uma configuração. O pino O0 é o bit mais significativo do byte.
BUSY	Indica quando o FPGA pode aceitar outro byte. Quando BUSY está em nível lógico baixo, o FPGA lê o barramento de dados na próxima borda de subida de CCLK quando ambos CE e WRITE estiverem ativados em nível lógico baixo. Se BUSY estiver em nível lógico alto, o byte atual é ignorado e precisa ser mantido no barramento até a próxima borda de descida de CCLK onde BUSY estiver em nível lógico baixo.
CE	Habilita o ICAP. Este pino é ativo em nível lógico baixo, embora possa ser invertido no FPGA Editor.
WRITE	Indica se está sendo feito uma configuração ou um readback do FPGA. É ativado em nível lógico baixo, podendo ser invertido no FPGA Editor assim como o sinal CE. Quando em nível lógico baixo, WRITE indica que um byte está sendo escrito pelo barramento I [0:7]. Quando em nível lógico alto, WRITE indica que um byte está sendo lido pelo barramento O [0:7].
CCLK	Sincroniza as leituras e escritas dos barramentos de configuração e readback.

Os dispositivos da família Virtex-II possuem uma porta interna de acesso de configuração denominada ICAP (*Internal Configuration Access Port*). Este componente está localizado na parte inferior direita do FPGA e permite que o dispositivo se auto reconfigure. O ICAP pode funcionar a uma velocidade máxima de 66 MHz [BLO03] e a interface de comunicação é semelhante à interface SelectMAP, não possuindo os pinos utilizados para reconfiguração total (DONE, INIT e PROGRAM) e os pinos de modo de configuração (M0, M1 e M2). A Tabela 3 apresenta o significado dos pinos de entrada e saída do ICAP, enquanto a Figura 3 ilustra cada um dos pinos.

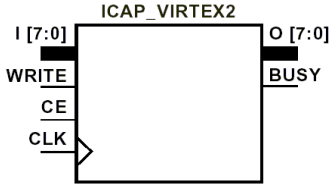


Figura 3 – Interfaces de entrada e saída do ICAP.

1.2 Fluxo básico de projetos VLSI em dispositivos FPGAs

Esta Seção apresenta uma introdução ao fluxo básico utilizado na implementação de projetos VLSI em FPGAs e respectivas alternativas de ferramentas para cada um dos passos. Este fluxo inicia com um projeto em linguagem de descrição de hardware e é finalizado na geração do bitstream. É importante conhecer cada um desses passos e as respectivas ferramentas, pois algumas destas farão uso de parâmetros avançados e específicos quando aplicados a projetos reconfiguráveis.

O primeiro passo, conforme o fluxo apresentado na Figura 4, é fazer a simulação funcional do projeto descrito em linguagem de descrição de hardware (VHDL, Verilog ou System-C RTL). Nesta etapa é possível verificar o correto funcionamento lógico do projeto a partir da criação de vetores de teste ligados à entrada do mesmo. O passo seguinte é a síntese lógica dos arquivos de descrição de hardware. Algumas ferramentas de síntese lógica são o XST da Xilinx, o Leonardo Spectrum da Mentor e o Synplify da Synplicity. Estas ferramentas suportam uma série de otimizações do usuário, como maior esforço em área, velocidade ou potência. Estas ferramentas geram uma descrição lógica do circuito em formato EDIF (*Electronic Digital Interchange Format*). A ferramenta de síntese lógica também pode gerar um arquivo SDF (*Standard Delay Format*), que contém os atrasos das portas lógicas. Isto permite fazer uma simulação mais precisa, possibilitando encontrar erros de temporização, não encontrados na simulação funcional. O arquivo EDIF, gerado pela ferramenta de síntese lógica, acompanhado do arquivo de restrições de recursos do usuário (UCF – *User Constraints File*), específico do FPGA alvo, são as entradas para a fase de síntese física. A síntese física é responsável pelo mapeamento dos dispositivos lógicos contidos no arquivo EDIF para os recursos disponíveis no FPGA, pelo posicionamento e pelo roteamento destes recursos. As ferramentas da Xilinx que executam estes passos são NGDBuild, MAP e PAR. Após o término da execução destas ferramentas, que pode demorar dependendo da complexidade do projeto, um bitstream total do projeto e uma série de relatórios são gerados. Entre os principais relatórios analisados estão o de velocidade e área, que informam os atrasos entre componentes do projeto e a ocupação deste no FPGA. O bitstream total do projeto é gerado pela ferramenta BitGen da Xilinx, podendo este ser configurado no FPGA a partir da ferramenta Impact.

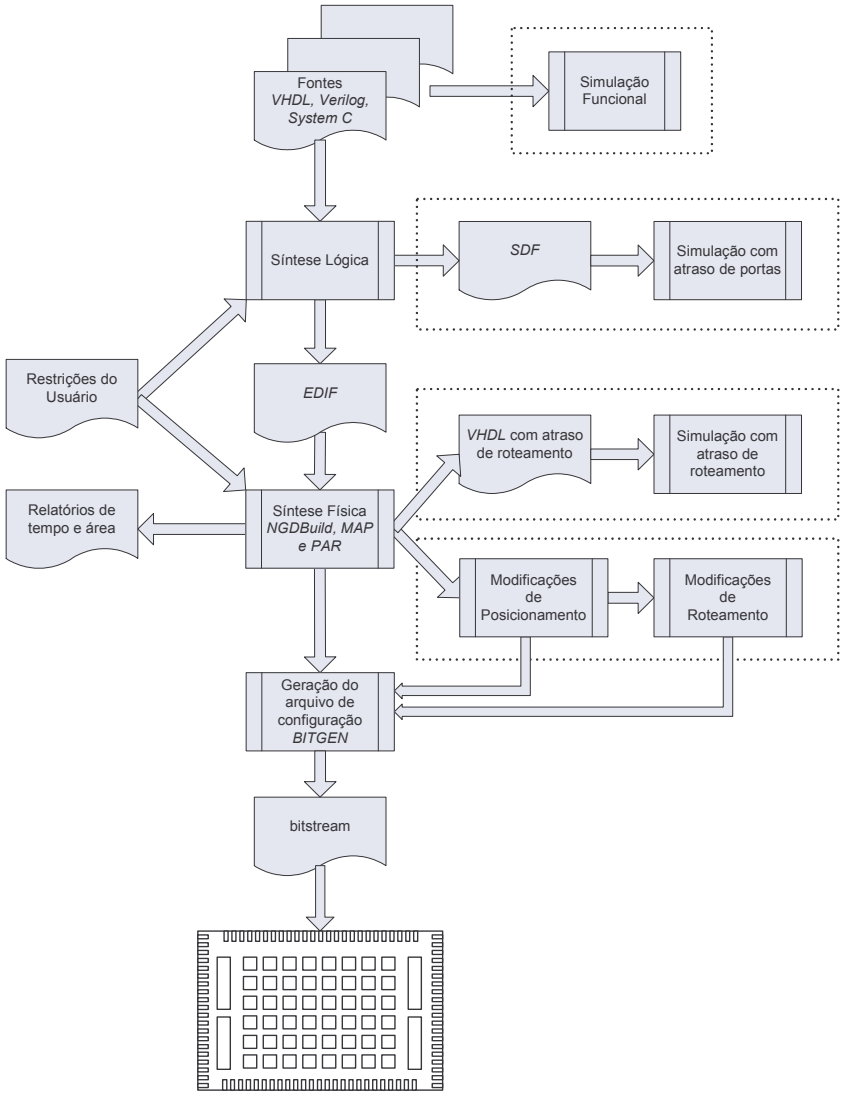


Figura 4 – Fluxo básico que inicia com o projeto em linguagem de descrição de hardware e termina com o bitstream gerado.

1.3 Fluxo básico para implementação de SDRs

Uma primeira motivação para a pesquisa em SDRs é a falta de um consenso sobre quais métodos e fluxos de projeto são mais indicados para a implementação destes sistemas. Os maiores desafios em implementar SDRs consistem em definir qual a porção do sistema será fixa (não reconfigurável), quais serão reconfiguráveis, como restringir módulos a uma posição específica do dispositivo, como será feita a comunicação entre os módulos, como os bitstreams parciais serão gerados e quem controlará o processo de reconfiguração.

O primeiro passo do fluxo de projeto de SDRs é definir um sistema inicial, escolhendo a quantidade de áreas reconfiguráveis que farão parte do sistema e quais módulos estarão inicialmente configurados. É aconselhável prototipar este sistema, de forma a garantir que as restrições temporais deste projeto sejam atendidas e que o mesmo funcione em hardware. O motivo para esta prototipação é que serão posteriormente adicionados novos elementos ao projeto, agregando complexidade ao mesmo.

O passo seguinte é contabilizar o número de módulos que serão configurados nas áreas previ-

amente estabelecidas. Uma forma organizada de gerar todos os módulos reconfiguráveis para todas áreas reconfiguráveis do sistema é criar um projeto por módulo, onde o mesmo módulo é instanciado em todas áreas reconfiguráveis. Desta forma o número de projetos diferentes a serem criados é igual ao número de módulos reconfiguráveis existentes no sistema. O número de bitstreams parciais será igual ao número de módulos reconfiguráveis multiplicado pelo número de áreas reconfiguráveis. A Figura 5 apresenta três módulos reconfiguráveis e os projetos propostos para a geração de um menor número de combinações necessárias para obter todos os módulos para todas as áreas reconfiguráveis do sistema.

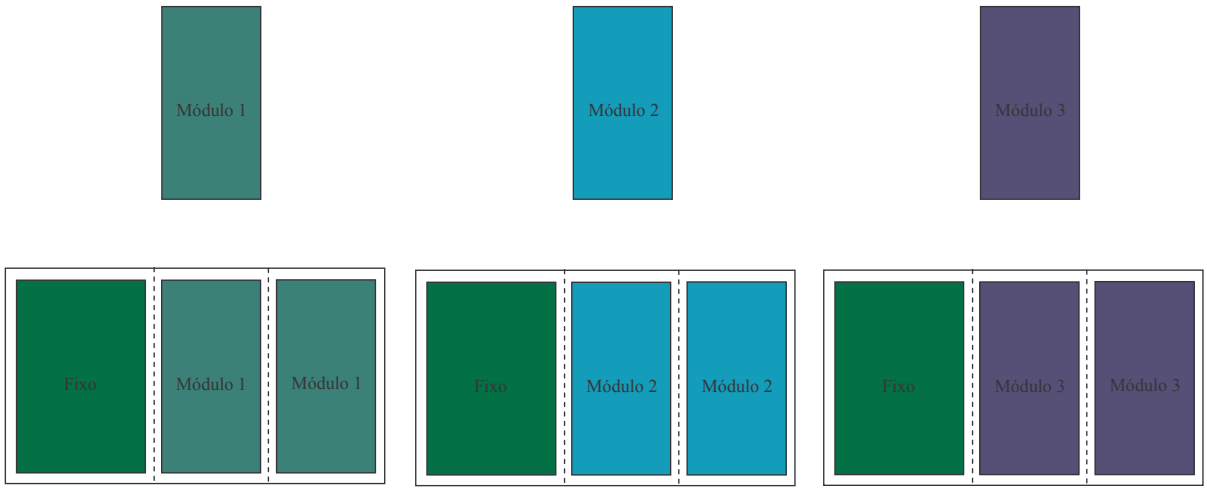


Figura 5 - Três módulos reconfiguráveis e os respectivos três projetos que permitem criar todos os módulos reconfiguráveis para todas as áreas reconfiguráveis de um SDR.

O terceiro passo é a adição de componentes *isoladores* entre os módulos reconfiguráveis e o restante do sistema em cada um dos projetos. Este componente isolador possui a função de fixar as conexões para que cada um dos módulos reconfiguráveis conecte os fios utilizando a mesma posição física, evitando falhas de roteamento durante a reconfiguração. Outra função dos componentes isoladores é impedir a passagem de possíveis dados transitórios no momento da reconfiguração. O componente indicado pela Xilinx para esta tarefa é a *bus macro*, que é formada por oito *tri-states* e disponibiliza quatro fios de um bit para a comunicação do sistema com o módulo reconfigurável ou vice-versa. Devido ao número reduzido de *tri-states* no FPGA, outras soluções foram propostas por Huebner [HUB04b] e Dyer [DYE02a], que serão estudadas no Capítulo 2.

O passo seguinte no fluxo genérico de implementação de SDRs é a *restrição da área física* do módulo reconfigurável. Nesta fase é importante fixar, em cada um dos projetos definidos no passo 3, os módulos reconfiguráveis dentro das áreas reconfiguráveis. Deve-se tomar cuidado em restringir tanto os recursos lógicos que foram alocados para cada módulo quanto os fios internos aos módulos, caso contrário um ou mais fios poderão ser incorretamente conectados no momento da reconfiguração, podendo causar um simples não funcionamento ou até um dano físico no dispositivo. A ferramenta Floorplanner da Xilinx auxilia no processo de restrição do posicionamento dos recursos lógicos dos módulos. Infelizmente, ao restringir o módulo a uma determinada posição, não obrigatoriamente os fios que interconectam os componentes internos do módulo estarão posicionados na mesma área, acontecendo com freqüência eles invadirem outros módulos. Parâmetros devem ser passados para as ferramentas de síntese física para forçar os fios a ficarem em uma área definida. A ferramenta FPGA Editor da Xilinx permite visualizar os fios do sistema e rotear manualmente ou automaticamente aqueles que saíram da sua área reservada.

O quinto passo para a geração de SDRs é a geração dos *bitstreams parciais* de cada um dos módulos reconfiguráveis do sistema. A ferramenta de geração de bitstreams, tanto parciais quanto totais, da Xilinx é o BitGen. Outra alternativa disponibilizada pela Xilinx é o JBits, uma API da linguagem Java que acessa o bitstream, para a modificação e criação de um bitstream parcial a partir de um bitstream já existente. A ferramenta CoreUnifier, apresentada no Capítulo 3, é uma ferramenta construída pelo autor deste trabalho para gerar bitstreams parciais com uma abordagem diferente dos métodos adotados pelo fabricante.

Existe ainda um passo opcional para a execução de SDRs que é a modificação do posicionamento dos módulos reconfiguráveis do sistema, que neste trabalho é chamado de realocação. A realocação pode ser útil quando a sugestão do passo 2 de gerar um bitstream para cada módulo em determinada área reconfigurável não for seguida. Alguns motivos do não cumprimento desta sugestão podem ser o elevado número de combinações a serem sintetizadas ou a dificuldade de restringir um determinado módulo em uma determinada área reconfigurável. Neste caso existe a alternativa de se utilizar uma ferramenta de realocação de módulos. Até o presente momento não se conhece uma ferramenta que permita realocar módulos para o dispositivo Virtex-II. O Capítulo 4 apresenta a ferramenta PARBIT [HOR04] que permite realocar módulos em dispositivos Virtex.

1.4 Organização do documento

O Capítulo 2 apresenta alternativas de componentes que podem ser utilizados para interligar áreas reconfiguráveis e o restante do sistema de forma a manter o roteamento fixo. O Capítulo 3 apresenta métodos e ferramentas para a geração de bitstreams parciais. A realocação de módulos internos ao sistema é tratada no Capítulo 4. O Capítulo 5 apresenta dois fluxos para a implementação de SDRs. Por fim o Capítulo 6 apresenta as conclusões e os trabalhos futuros.

2 INTERCONEXÃO ENTRE MÓDULOS RECONFIGURÁVEIS

A interconexão entre módulos reconfiguráveis e o restante do sistema deve ser feita por algum componente intermediário, denominado na Seção 1.3 de *componente isolador*. Este componente permite ter um ponto em comum conhecido entre cada um dos módulos instanciados em diferentes projetos. Portanto, o componente intermediário deve ser fixo no sistema e os módulos devem respeitar a semântica de cada pino do componente intermediário para o correto funcionamento do sistema. Outra atribuição do componente intermediário é isolar o restante do sistema enquanto um módulo está sendo reconfigurado. Este isolamento é necessário porque os dados recebidos da reconfiguração instantaneamente modificam o módulo, podendo assim ocorrer chaveamentos temporários que modifiquem o valor da saída do módulo, interferindo no funcionamento do restante do sistema que continua operando. Com um componente intermediário é possível isolar a área que está sendo reconfigurada e assim não interferir com o funcionamento do restante do sistema.

Este Capítulo apresenta componentes intermediários utilizados em SDRs. Estes componentes são denominados na literatura de macros. Os passos para a criação de uma macro genérica são apresentados na Seção 2.1. A Seção 2.2 apresenta a bus macro, que é formada por oito *tri-states* e permite conectar quatro fios de um bit entre uma área reconfigurável e o restante do sistema. A Seção 2.3 apresenta a macro de Huebner [HUB04b] que instancia uma CLB da Virtex-II e permite transferir 8 bits pelo componente intermediário. A CLB macro simples e dupla, componente proposto por Dyer [DYE02b] para a família Virtex, é estudada na Seção 2.4.

2.1 Criação de macros

As macros utilizadas como elementos de interconexão entre um módulo reconfigurável e o restante do sistema são formadas por componentes existentes na arquitetura do dispositivo FPGA. Uma forma de criar uma macro é através da ferramenta FPGA Editor, que apresenta de forma gráfica os componentes internos dos FPGAs da Xilinx. Esta ferramenta permite selecionar os componentes a serem utilizados, assim como fazer a ligação entre eles. Após o término da montagem da macro, instâncias da mesma devem ser conectadas às interfaces dos módulos reconfiguráveis, o que é feito no arquivo de mais alto nível de descrição do projeto de hardware (designado como *top*). Para que todos os projetos do SDR possuam as macros posicionadas em locais pré-determinados, restrições manuais devem ser explicitamente inseridas nos arquivos de restrição de projeto.

O objetivo desta Seção é apresentar os passos básicos de como criar uma macro, sem entretanto mostrar quais os componentes constituem a macro, pois três implementações diferentes serão abordadas nas seções 2.2, 2.3 e 2.4. Cabe ressaltar também que Dyer [DYE02b] apresentou passos semelhantes aos aqui apresentados, mas específicos a sua macro e à família Virtex. A seguir os passos genéricos para a construção de uma macro são enumerados.

1. No FPGA Editor criar um novo projeto, selecionar a criação de uma *macro*, inserir um nome para a mesma e configurar o lugar onde ela será criada. Selecionar a família de FPGAs para a qual a macro está sendo desenvolvida. Dispositivo, encapsulamento e velocidade não são importantes.

2. A janela principal do FPGA Editor é iniciada mostrando a arquitetura do dispositivo sele-

cionado. Adicione cada um dos componentes que farão parte da macro e configure-os.

3. Coloque o componente em modo de edição e selecione os recursos a serem utilizados. No caso de LUTs é possível editar a equação da mesma e no caso de BRAMs valores de inicialização podem ser preenchidos.

4. De volta à janela principal da ferramenta deve-se adicionar os pinos de entrada e saída dos componentes editados e inserir um nome para cada pino.

5. A seguir selecione os pinos que devem ser conectados por um fio e nomeie-o. Repita este passo para cada fio da macro.

6. Salve a macro.

7. Instancie a macro no código VHDL.

8. Faça a restrição de posicionamento de cada uma das instâncias da macro no arquivo UCF.

9. Coloque o arquivo *.nmc* salvo pelo FPGA Editor no mesmo diretório que o código VHDL.

Após estes passos a ferramenta de síntese física estará apta a utilizar a macro como um componente já roteado.

2.2 Bus macro

A bus macro [LIM04][XIL04c] é o componente intermediário proposto pela Xilinx para a interconexão entre módulos reconfiguráveis e o restante do sistema. Vários trabalhos [MÖL04][BLO04][WAL04] já utilizaram com sucesso as bus macros como componente de interconexão de áreas reconfiguráveis utilizando o fluxo de Projeto Modular, proposto pela Xilinx. A bus macro é composta por oito *tri-states* e permite conectar quatro fios de um bit entre uma área reconfigurável e o restante do sistema. A Figura 6 apresenta o diagrama esquemático da bus macro.

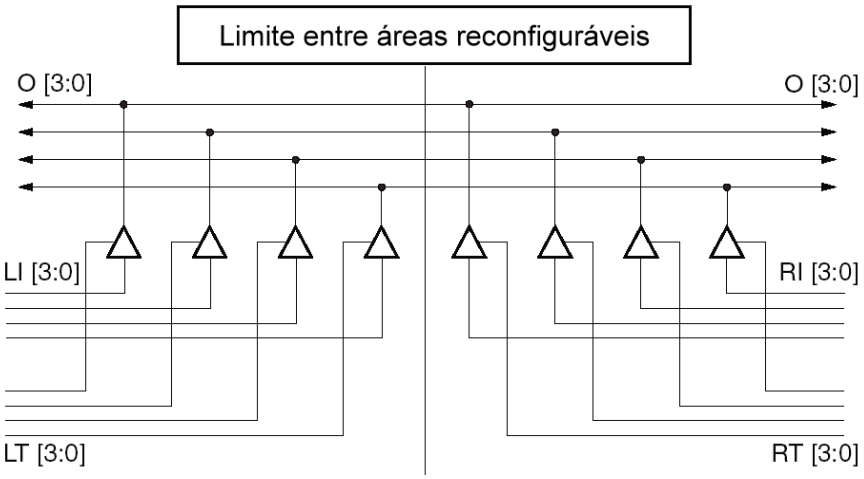


Figura 6 – Implementação física da bus macro.

A bus macro possui quatro saídas (O) bidirecionais. Cada sinal de saída pode estar ligado na entrada esquerda (LI) ou na entrada direita (RI). Desta forma apenas o toggle esquerdo (LT) ou o toggle direito (RT) de um sinal de saída podem estar ativados simultaneamente de forma a não causar um curto circuito. É importante ressaltar que os *tri-states* são ativos em nível lógico baixo e que para desativar uma determinada saída da bus macro basta colocar LT e RT em nível lógico alto.

O trabalho [MÖL04] apresenta um SDR que utiliza bus macros para interconectar módulos

reconfiguráveis. Este sistema possui duas áreas reconfiguráveis onde cada área pode estar configurada com um módulo de raiz quadrada, multiplicação ou divisão. O sistema possui um controlador de configurações disparado pelo software que é executado no processador R8 [MOR03a] localizado na área fixa. Os módulos são fracamente acoplados ao processador.

Um ponto negativo da utilização de bus macros implementadas com *tri-states* é que estes recursos são escassos nos FPGAs Virtex-II, existindo apenas dois por CLB. Huebner [HUB04b] aponta que a ferramenta de roteamento algumas vezes não respeita as restrições feitas pelo projetista e cruza o limite entre as áreas reconfiguráveis na implementação de bus macros com *tri-states*. O principal motivo para a ocorrência de erros é o fato da linha de roteamento do sinal de saída ser fisicamente a mesma em ambos os lados da macro. A Figura 7 ilustra este problema. O retângulo preto é um componente do módulo esquerdo. A ferramenta de roteamento conectou o sinal indicado utilizando um fio conectado ao lado direito, pelo fato do sinal ser equivalente em ambos os lados. A conseqüência deste ato é um simples mau funcionamento do sistema ou até um curto circuito que pode danificar o FPGA.

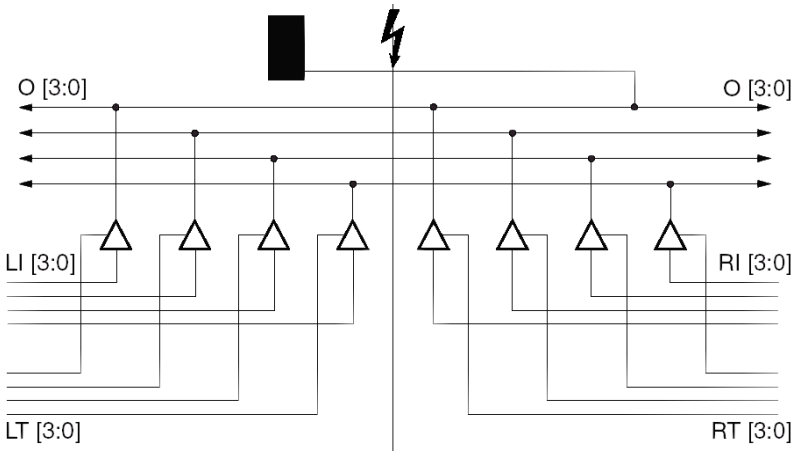


Figura 7 – Problema reportado por Huebner de cruzamento de áreas reconfiguráveis sem a passagem por macros.

2.3 Macro de Huebner

Huebner [HUB04b] desenvolveu macros de forma a seguir a idéia de comunicação em SDRs proposta por Palma [PAL02a], onde os elementos que comunicam as diversas áreas reconfiguráveis ficam na parte inferior do FPGA. Foram encontradas duas vantagens na utilização desta abordagem. A primeira é a possibilidade de isolar a computação na parte superior do FPGA da comunicação fixada na parte inferior. Desta forma é possível reconfigurar a computação de um módulo sem interromper a comunicação dos outros que estão em execução no sistema. Isto é factível pois a reconfiguração de dispositivos FPGAs Xilinx é *glitchless*, permitindo que áreas previamente configuradas não tenham nenhum tipo de transitório se os mesmos dados de configuração forem sobrepostos. A segunda vantagem desta abordagem é que as macros estão dispostas horizontalmente, apresentando assim uma melhor ocupação de área, porque se as macros estiverem dispostas verticalmente são necessárias tantas colunas reservadas para macros quantos forem os módulos reconfiguráveis do sistema.

O trabalho de Huebner apresenta barramentos unidirecionais na parte inferior do FPGA utilizando macros. No princípio Huebner tentou instanciar múltiplas macros iguais lado a lado de forma

a implementar um barramento. Devido à arquitetura Virtex-II apresentar mistura de colunas de CLBs com BRAMs e blocos multiplicadores, problemas de roteamento foram detectados. Por este motivo as diversas macros que seriam instanciadas foram todas roteadas manualmente, construindo um barramento inteiro, conforme apresentado na Figura 8.

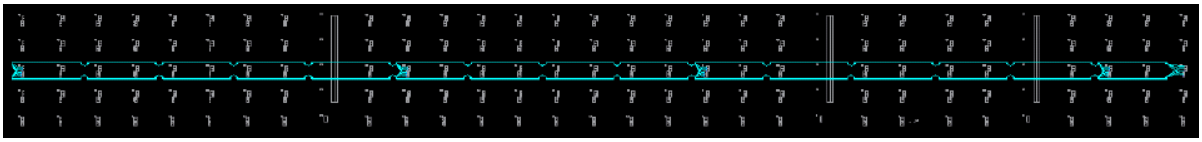


Figura 8 – Barramento de Huebner.

O motivo de Huebner ter desenvolvido um barramento unidirecional é que um barramento para a escrita possui uma arquitetura diferente de um barramento de leitura. Quando um módulo deseja ler dados de um barramento ele deve receber os dados se os mesmos forem destinados a ele. No caso de escrita um árbitro deve selecionar qual o módulo pode escrever no barramento para que não ocorra conflito. Devido a este comportamento as macros de leitura e escrita são diferentes, conforme mostra a Figura 9.

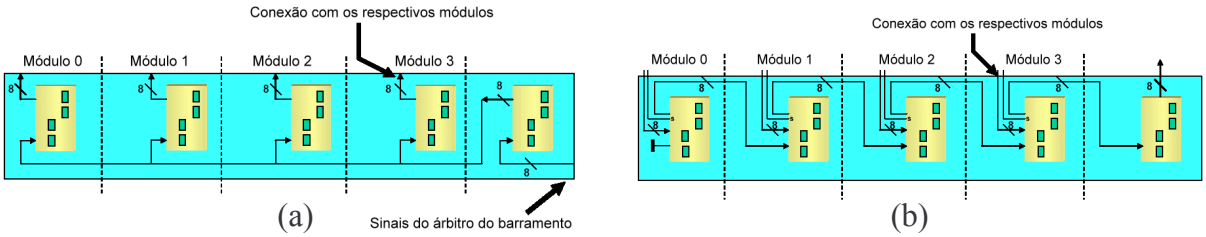


Figura 9 – (a) macro de leitura; (b) macro de escrita.

A Figura 10 apresenta como as slices da macro de leitura e escrita estão configuradas. As slices da macro de leitura estão configuradas como somente passagem de dados, enquanto as slices da macro de escrita estão configuradas como multiplexadores. Isto é necessário para prover acesso de escrita quando existe mais de um módulo conectado ao barramento.

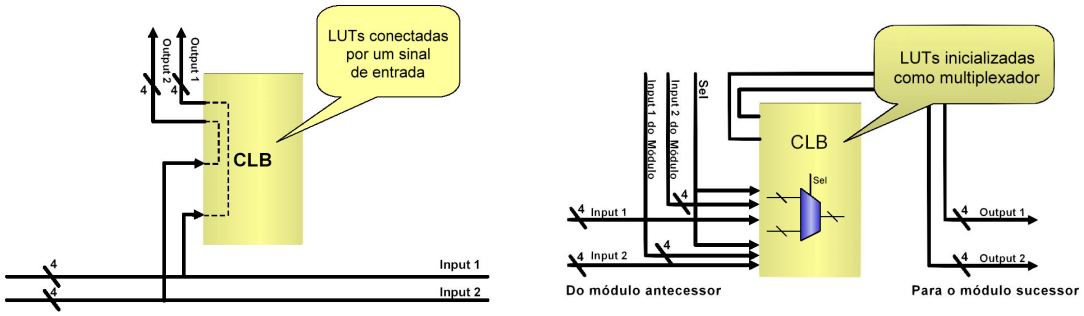


Figura 10 – (a) configuração de uma CLB da macro de leitura; (b) configuração de uma CLB da macro de escrita.

Cada barramento tem largura de 8 bits e o atraso, segundo Huebner, é de 5,5 ns entre o módulo mais à esquerda do FPGA e o árbitro que está posicionado mais à direita da Figura 8. O problema desta arquitetura de barramentos é o procedimento de arbitragem, Daisy-chain, onde a prioridade de

acesso ao barramento é fixa.

Aplicações automotivas foram utilizadas para comprovar o correto funcionamento das macros apresentadas em SDRs. Neste estudo foi utilizado um FPGA Virtex-II XC2V3000, onde foi adicionado o processador Microblaze e quatro módulos reconfiguráveis.

2.4 Macro de Dyer

Dyer e Wirz propuseram em [DYE02b] duas macros para solucionar os problemas de roteamento em SDRs. A macro denominada "CLB macro simples" é programada como uma macro de passagem de dados e é utilizada para auxiliar no roteamento de conexões que invadem um determinado módulo. Desta forma é possível posicionar uma ou mais macros de passagem na rota a ser seguida pelo fio, assim forçando um determinado caminho entre a origem e o destino. O problema e a solução sugerida são apresentados na Figura 11.

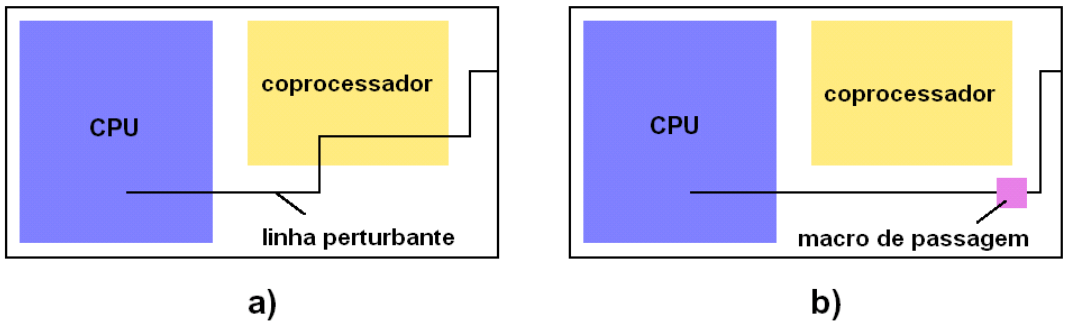


Figura 11 – (a) linha que liga um pino de entrada e saída à CPU cruza a área do coprocessador; (b) macro de passagem inserida para forçar o caminho da linha perturbante apresentada em (a).

A macro denominada "CLB macro dupla" é utilizada para isolar um módulo reconfigurável do restante do sistema. Ela é formada por duas CLBs interconectadas, conforme apresentado na Figura 12, e programados como passagem de dados. A CLB macro dupla permite comunicar quatro bits e possui uma rota fixa entre as duas CLBs. Ambas as macros foram implementadas para dispositivos Virtex. Os métodos utilizados por Dyer e Wirz para a geração de bitstreams parciais são apresentado na Seção 3.3.

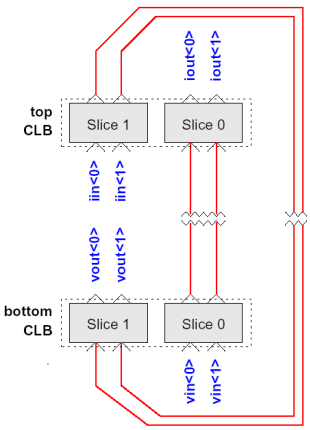


Figura 12 – CLB macro dupla.

As macros propostas por Dyer não são um meio de comunicação entre módulos do sistema

como as macros propostas por Huebner [HUB04b] e Palma [PAL02a]. As macros utilizadas por Dyer são apenas utilizadas para guiar as conexões entre os módulos do sistema.

Parar testar as macros propostas uma aplicação de decodificação de áudio foi implementada sobre uma Virtex XCV800. Neste sistema um módulo Ethernet recebe pacotes UDP com áudio codificado e um processador LEON desempacota o áudio e passa para o módulo reconfigurável decodificá-lo. O resultado é automaticamente ouvido pela saída de som da plataforma de prototipação.

3 MÉTODOS DE GERAÇÃO DE BITSTREAMS PARCIAIS

A fase de geração de bitstreams parciais é um processo específico até mesmo para dispositivos de uma mesma família de um mesmo fabricante. Com freqüência apenas o próprio fabricante cria ferramentas para a geração de bitstreams parciais, pois para efetuar este passo é necessário conhecer em profundidade a organização do bitstream em relação à arquitetura do dispositivo. Como os fabricantes não têm o interesse de divulgar detalhes sobre suas arquiteturas, com o receio de cópia da mesma, apenas eles possuem informações suficientes para fazer ferramentas capazes de configurar parcialmente os seus dispositivos. Outro motivo que inibe os fabricantes de dispositivos programáveis a fornecerem mais informações de suas arquiteturas é que seja feita engenharia reversa de algum circuito, podendo assim colocar em risco a propriedade intelectual de produtos dos clientes.

Este Capítulo apresenta algumas ferramentas de geração de bitstreams parciais voltados a dispositivos Virtex, tanto implementadas pelo fabricante quanto implementadas por terceiros. No caso da família Virtex, ferramentas de terceiros foram criadas porque a Xilinx divulgou detalhes da organização do bitstream. Infelizmente estes dados não foram divulgados para a família Virtex-II, praticamente impossibilitando a atualização das ferramentas de geração de bitstreams parciais existentes.

A Tabela 4 apresenta, além de alguns dados já informados na Tabela 2, o tempo de reconfiguração parcial de um frame de diferentes dispositivos da família Virtex-II. As duas últimas colunas da tabela apresentam os tempos de reconfiguração parcial aproximados para a reconfiguração a partir da interface SelectMAP e a partir do módulo ICAP. É importante ressaltar que com exceção de pequenas reconfigurações, por exemplo reconfiguração de parâmetros, toda uma coluna será modificada. Os diferentes tipos de colunas e o número de frames gasto para reconfigurar um tipo de coluna inteira é apresentado na Tabela 5. Desta forma é possível saber que se um módulo ocupa uma coluna de CLBs, então é necessário reconfigurar 22 frames, onde 22 frames de um dispositivo XC2V40 utilizando a interface ICAP demora (22 x 1,726) 37,972 μs para ser reconfigurada.

Tabela 4 – Tempos de reconfiguração de diferentes frames da família Virtex-II.

Dispositivo	Número de frames por dispositivo	Tamanho do frame em bits	Tempo de reconfiguração total SelectMAP (50 MHz) em μs	Tempo de reconfiguração de um frame SelectMAP (50 MHz) em μs	Tempo de Reconfiguração de um frame ICAP (66 MHz) em μs
XC2V40	404	832	840	2,079	1,726
XC2V80	404	1472	1490	3,688	3,061
XC2V250	752	2112	3970	5,279	4,382
XC2V500	928	2752	6380	6,875	5,707
XC2V1000	1104	3392	9360	8,478	7,037
XC2V1500	1280	4032	12900	10,078	8,365
XC2V2000	1456	4672	17010	11,683	9,697
XC2V3000	1804	5312	23960	13,282	11,024
XC2V4000	2156	6592	35530	16,480	13,678
XC2V6000	2508	7872	49360	19,681	16,335
XC2V8000	2860	9152	65440	22,881	18,992

Tabela 5 – Número de frames por tipo de coluna para dispositivos da família Virtex-II.

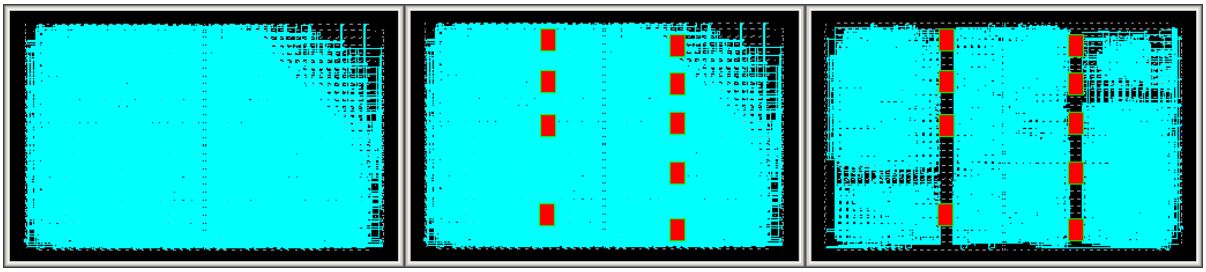
Tipo de coluna	Número de frames por coluna	Número de colunas por dispositivo
IOB1	4	2
IOB2	22	2
CLB	22	Nro de colunas de CLBs
BRAM	64	Nro de colunas de BRAMs
BRAM Interconnect	22	Nro de colunas de BRAMs
GCLK	4	1

A Seção 3.1 faz a conexão do Capítulo anterior com o atual, buscando mostrar os passos intermediários entre a inserção de macros e a geração do bitstream parcial. A ferramenta BitGen, apresentada na Seção 3.2, é a ferramenta do próprio fabricante para a geração de bitstreams totais e parciais. Uma biblioteca de classes para a linguagem Java, chamada JBits, que manipula bitstreams é apresentada na Seção 3.3. A Seção 3.4 apresenta o CoreUnifier que gera bitstreams parciais a partir de totais.

3.1 Restrição de posicionamento e verificação de roteamento

O Capítulo 2 abordou a fase de inserção de componentes intermediários, sendo este um passo importante para que diferentes projetos reconfiguráveis tenham pontos fixos por onde conectar as interfaces dos módulos ao restante do sistema. Esta Seção possui o objetivo de mostrar como restringir os recursos de um módulo a uma determinada área do dispositivo e onde fixar as macros que separam os módulos reconfiguráveis ao restante do sistema.

A Figura 13 apresenta diferentes momentos da implementação de um SDR. Em (a) tem-se um sistema funcionando em FPGA sem características reconfiguráveis. Em (b) é ressaltado a inserção dos componentes intermediários no sistema, conforme apresentado no Capítulo 2. Em (c) são feitas as restrições de posicionamento das diferentes áreas fixas e reconfiguráveis, conteúdo tratado nesta Seção.



(a) (b) (c)

Figura 13 – (a) sistema sem características reconfiguráveis; (b) sistema com componentes isoladores; (c) módulos do sistema com áreas restringidas.

3.1.1 Posicionamento de módulos

Como já mencionado na Seção 1.1, os dispositivos Xilinx são configurados em colunas. Portanto, recomenda-se posicionar os recursos de um módulo em um menor número de colunas, para que o tempo de reconfiguração seja mínimo. Este tipo de restrição é feito pela ferramenta Floor-

planner da Xilinx. Para que seja possível visualizar o projeto separado por módulos, de forma a facilitar o posicionamento dos mesmos, deve-se utilizar a opção de manter a hierarquia na ferramenta de síntese utilizada. Ao abrir o projeto no Floorplanner, os diferentes módulos serão exibidos, permitindo que os mesmos sejam selecionados e arrastados para a posição específica a ser ocupada do FPGA. A Figura 14 apresenta a janela da ferramenta gráfica Floorplanner com um sistema que possui cinco módulos, sendo que três já tiveram as suas posições e tamanhos restringidos.

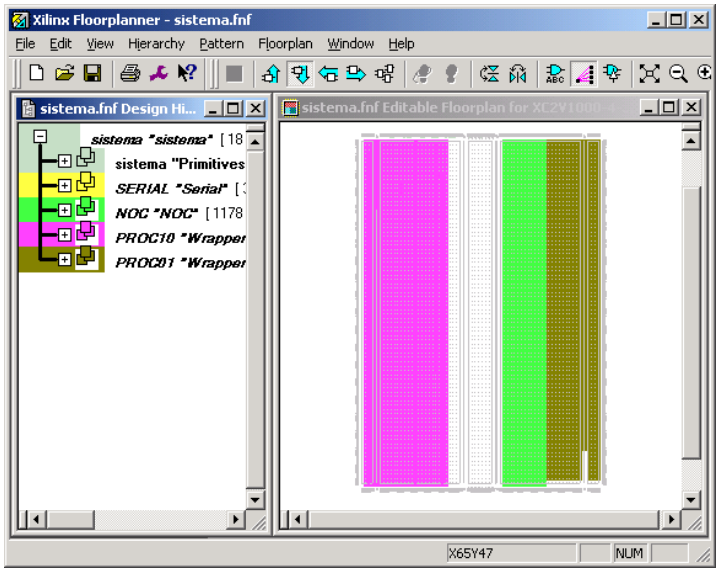


Figura 14 – Tela da ferramenta Floorplanner com três de cinco módulos já restringidos.

Uma vez feita a restrição de posicionamento no Floorplanner, o fluxo básico de projetos VLSI em FPGAs deve ser executado novamente para que as restrições impostas ao projeto sejam levadas em consideração. Deve-se ficar atento aos relatórios gerados pela ferramenta de síntese, verificando se as restrições efetuadas foram atendidas. Caso alguma falhe, a ferramenta reportará qual módulo não pode ser sintetizado, necessitando uma nova execução do Floorplanner.

3.1.2 Verificação de roteamento

Uma vez que a síntese tenha procedido sem erros é recomendável verificar na ferramenta FPGA Editor se não existem conexões que tenham cruzado algum módulo, não respeitando a restrição feita na Seção 3.1.1. Caso isto tenha acontecido pode-se voltar ao Floorplanner para modificar alguma restrição de área ou pode-se tentar fazer o roteamento manual da conexão em questão pelo FPGA Editor.

A ferramenta FPGA Editor trabalha com arquivos de extensão NCD (*Native Circuit Description*) e NMC, conforme apresentado na Seção 2.1. Como o FPGA Editor é uma ferramenta gráfica de interação com componentes internos do FPGA, modificações repetitivas podem ser demoradas de serem feitas visualmente. Por isso, existem duas formas para facilitar este processo. Uma forma é utilizar o console de comandos da ferramenta FPGA Editor. Outra alternativa é utilizar a ferramenta XDL da Xilinx para transformar o NCD em um arquivo texto e vice-versa. Em ambas alternativas deve-se pesquisar nas respectivas documentações como aplicar as mudanças desejadas.

3.2 BitGen

A ferramenta BitGen é uma ferramenta que opera por linha de comando e permite gerar bitstreams parciais e totais a partir de arquivos NCD. Um bitstream total pode ser gerado a partir da linha de comando **`bitgen arquivo.ncd`**, onde **`arquivo.ncd`** é o nome do projeto gerado pela ferramenta de síntese física ou modificado pelo FPGA Editor.

Bitstreams parciais com pequenas modificações podem ser gerados a partir do comando **`bitgen -g ActiveReconfig:Yes -g persist:yes -r in.bit mod.ncd par.bit`** que permite gerar um bitstream parcial (**`par.bit`**) a partir de um bitstream inicial (**`in.bit`**) e o projeto do inicial com alguma modificação (**`mod.ncd`**) feita pelo FPGA Editor. O parâmetro **`ActiveReconfig:Yes`** informa que uma reconfiguração parcial está em andamento e por isso não se deve modificar os valores dos flip flops. O parâmetro **`persist:yes`** é utilizado para manter os pinos de configuração após a configuração atual.

Bitstreams parciais de módulos podem ser gerados a partir do comando **`bitgen -f param.ut top.ncd`**, onde **`top.ncd`** é o módulo já roteado e **`param.ut`** é um arquivo com parâmetros a serem utilizados pela ferramenta BitGen. Os parâmetros utilizados no arquivo **`param.ut`** podem ser visualizados na Figura 15. Para que o bitstream parcial **`top.bit`** seja corretamente gerado, é necessário que o arquivo **`top.ncd`** seja sintetizado com alguns parâmetros adicionais. Estes parâmetros são utilizados no fluxo de Projeto Modular, estudado na Seção 5.1.

```
01 -d
02 -w
03 -l
04 -m
05 -g ActiveReconfig:yes
06 -g ReadBack
07 -g DebugBitstream:No
08 -g CRC:Enable
09 -g ConfigRate:4
10 -g CclkPin:PullUp
11 -g M0Pin:PullUp
12 -g M1Pin:PullUp
13 -g M2Pin:PullUp
14 -g ProgPin:PullUp
15 -g DonePin:PullUp
16 -g DriveDone:No
17 -g PowerdownPin:PullUp
18 -g TckPin:PullUp
19 -g TdiPin:PullUp
20 -g TdoPin:PullNone
21 -g TmsPin:PullUp
22 -g UnusedPin:PullUp
23 -g UserID:0xFFFFFFFF
24 -g DCMShutDown:Disable
25 -g DisableBandgap:No
26 -g StartUpClk:JtagClk
27 -g DONE_cycle:4
28 -g GTS_cycle:5
29 -g GWE_cycle:6
30 -g LCK_cycle:NoWait
31 -g Match_cycle:NoWait
32 -g Security:None
33 -g Persist:Yes
34 -g DonePipe:No
35 -g Encrypt:No
36 -g ActivateGCLK:Yes
```

Figura 15 – Parâmetros da ferramenta BitGen para a geração de bitstreams parciais de módulos.

3.3 JBits

JBits é um conjunto de classes Java que provê ao usuário acesso em alto nível de abstração à bitstreams da família Virtex e Virtex-II. Atualmente a versão 3 do JBits provê suporte apenas aos bitstreams da família Virtex-II, enquanto as versões anteriores provêm acesso apenas a família Virtex [XIL04d].

Entre as principais características do JBits pode-se citar a leitura e a escrita de valores de componentes internos como LUTs, CLBs, BRAMs e IOBs, e a geração de bitstreams parciais ou totais com as modificações efetuadas. Uma vantagem do JBits é a sua velocidade de geração e modificação de bitstreams quando comparados à execução de demoradas sínteses para a geração de um bitstream.

O JBits tem sido utilizado para construir circuitos completos, para modificar circuitos existentes que foram gerados por ferramentas de síntese padrão e para a construção de outras ferramentas.

Isto inclui ferramentas de projeto tradicionais para executar tarefas como posicionamento e roteamento do circuito, bem como ferramentas de aplicação específica, como por exemplo ferramentas de reconfiguração parcial e remota [MÖL03] ou modificação de parâmetros internos do projeto [MES03].

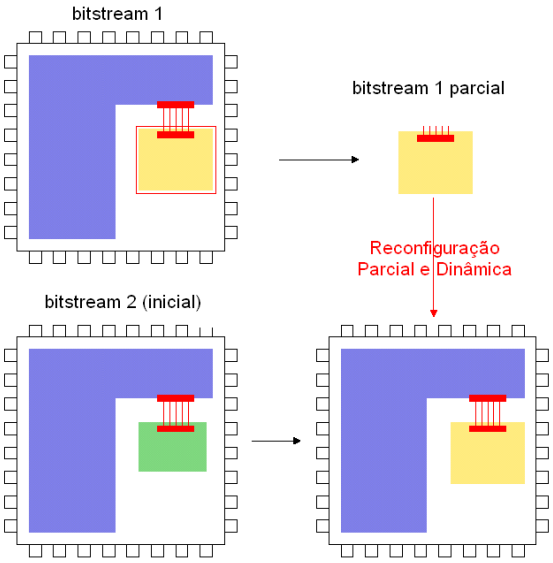


Figura 16 – Geração de bitstream parcial com roteamento estático.

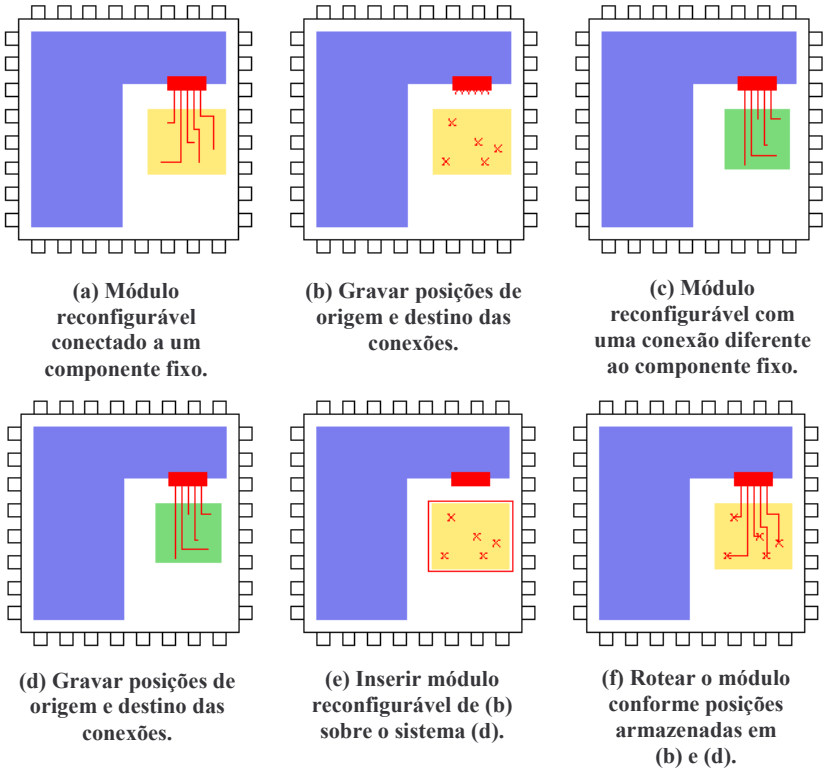


Figura 17 – Geração de bitstream parcial com roteamento dinâmico.

A não adoção do JBits na construção de circuitos completos reside basicamente na limitação das ferramentas de validação e otimização providas pelo JBits. Em função desta limitação a Xilinx

retirou do JBits 3 as ferramentas BoardScope, DeviceSimulator e outras [XIL04d] que permitem validar um projeto a partir do bitstream. Este fato foi constatado também por Dyer [DYE02a], que apontou duas abordagens para utilizar o JBits na implementação de SDRs. Ambas as abordagens iniciam por bitstreams totais gerados e validados por ferramentas padrões, conforme o fluxo apresentado na Seção 1.2. A primeira abordagem utiliza o JBits para unir os módulos, mas não roteia as conexões. As localizações dos módulos reconfiguráveis e da interface entre módulos estáticos e reconfiguráveis estão em posições fixas, conforme é ilustrado na Figura 16. A segunda abordagem posiciona e roteia os módulos dinamicamente a partir do JBits, conforme é apresentado na Figura 17.

Uma abordagem utilizada por Huebner para gerar bitstreams parciais a partir do JBits é marcar os frames de um bitstream total como modificados e gerar um bitstream parcial deste. A Figura 18 apresenta o trecho de código em linguagem Java utilizando a API JBits 3 responsável por gerar bitstreams parciais segundo este método.

```
01 //instanciando uma matriz de recursos
02 int[][] res=new int[device.getTileRows()][device.getTileCols()];
03
04 // cria a instancia de um bitstream parcial
05 byte[] partial = jbits.generatePartial();
06
07 // percorre as colunas
08 for (int x=startColumn; x<endColumn; x++)
09 {
10    //percorre os frames
11    for (int y=0; y<22; y++)
12    {
13          //atribuindo um valor para o calculo do MinorFrame
14          res[0][1] = y;
15
16          //captura a primeira linha do frame
17          int value[] = jbits.getCLBBits(0,x,res);
18
19          //escreve o valor lido de volta no frame
20          jbits.setCLBBits(0,x,res,value);
21    }
22 }
23 //habilita o calculo de CRC
24 jbits.enableCrc(true);
25
26 //gera o bitstream parcial
27 jbits.writePartial(outFileName);
```

Figura 18 – Trecho de código Java/JBits para geração de bitstreams parciais de módulos.

3.4 CoreUnifier

O CoreUnifier [MES03][MOR03b] é uma ferramenta gráfica desenvolvida em linguagem Java, e possui o objetivo de gerar bitstreams parciais a partir da seleção de áreas de bitstreams totais. Ela foi implementada a partir das equações de endereçamento do dispositivo Virtex disponibilizadas no Application Note 151 [XIL04e]. A proposta de fluxo a ser utilizada com o CoreUnifier é que bitstreams totais sejam construídos segundo o fluxo padrão de projetos VLSI com a adição das etapas de inserção de macros (Capítulo 2) e de restrição de área dos módulos (Seção 3.1). Desta forma deve-se criar um projeto completo para cada combinação do SDR e utilizar o CoreUnifier para gerar os bitstreams parciais a partir de cada um dos totais de cada projeto.

O primeiro bitstream total aberto no CoreUnifier é o bitstream chamado de *base*. Todos os bitstreams abertos posteriormente são mostrados em uma nova aba da ferramenta e são denominados de *secundários*. Cada área selecionada em cada um dos bitstreams secundários é inserida sobre a mesma posição no bitstream base. Após a seleção de áreas dos bitstreams secundários, é possível verificar na aba do bitstream base os novos módulos que foram inseridos. A partir disso pode ser

gerado um bitstream total ou um bitstream parcial de uma área selecionada do bitstream base. A Figura 19 apresenta a tela da interface gráfica da ferramenta em questão, com um bitstream base e um bitstream secundário carregados.

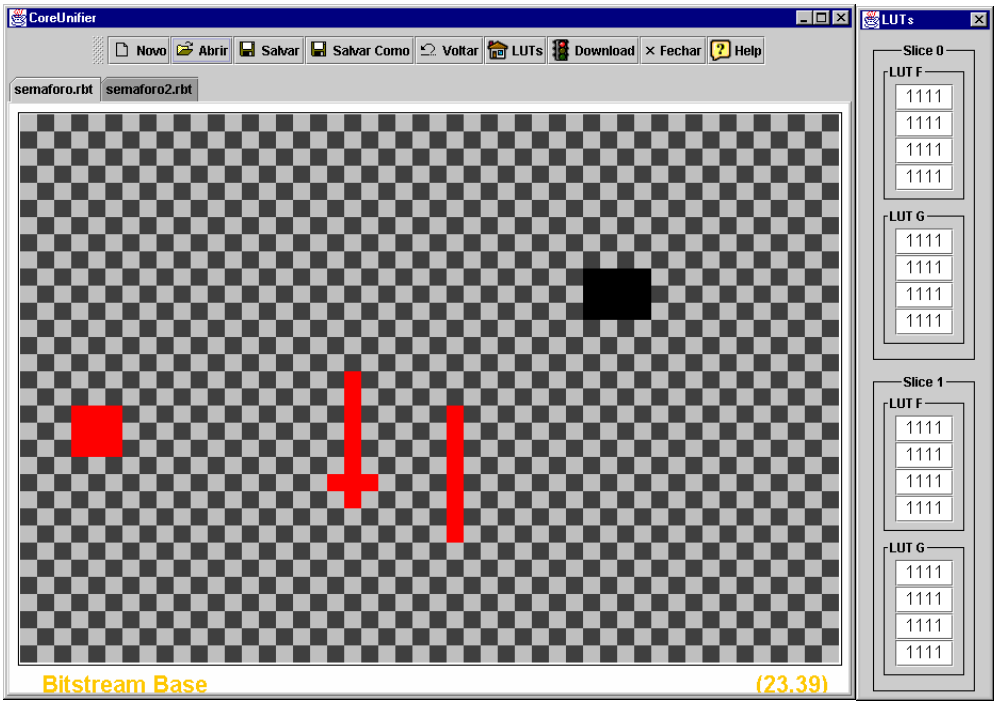


Figura 19 – Tela da ferramenta CoreUnifier com dois bitstreams totais carregados.

A Figura 19 apresenta uma matriz de 32 linhas por 48 colunas, representando cada um dos CLBs da Virtex 300. Quadrados cinza claros e cinza escuros representam CLBs não utilizadas, ou que possuem as LUTs com os valores padrão. Quadrados vermelhos representam CLBs utilizadas no bitstream base. Quadrados com cores diferentes representam módulos inseridos. Os valores das LUTs são visualizados na janela auxiliar a direita da Figura 19 ao passar o mouse sobre alguma CLB.

Encontra-se em andamento o estudo do bitstream da família Virtex-II para a construção da versão II da ferramenta CoreUnifier. Recente acordo de cooperação de pesquisa entre o GAPH e a Xilinx, através da assinatura de um NDA (*Non-Disclosure Agreement*), permitirá a atualização da ferramenta CoreUnifier para a família Virtex-II.

Além das características já citadas do CoreUnifier, planeja-se permitir a leitura do arquivo de restrição do usuário na ferramenta CoreUnifier-II. Conhecendo as restrições de posicionamento feitas pelo usuário é possível facilitar a visualização dos módulos e auxiliar na tarefa de geração de bitstreams parciais. Para melhor inserir a ferramenta CoreUnifier-II a um fluxo de construção de SDRs, será permitido uma interação por linha de comando, de forma a automatizar o processo de geração de bitstreams. Também se planeja permitir a modificação do posicionamento dos módulos a partir do CoreUnifier-II, característica esta chamada de realocação.

4 REALOCAÇÃO DE MÓDULOS DE HARDWARE

Assim como sistemas microprocessados alocam e desalocam memória segundo a demanda do sistema operacional, SDRs configuram e reconfiguram o hardware conforme a exigência das aplicações que estão sendo executadas sobre ele. Por este motivo SDRs sofrem de um problema semelhante, que é a fragmentação interna e externa. É denominada fragmentação interna a lógica configurável não utilizada dentro da área reservada a um módulo de hardware. Fragmentação externa ocorre quando existe lógica configurável livre suficiente para um módulo de hardware, entretanto esta lógica se encontra toda particionada impossibilitando a configuração do módulo.

Uma alternativa para contornar o problema de fragmentação em SDRs é utilizar a reconfiguração parcial e dinâmica para realocar a posição dos módulos de hardware do sistema e liberar espaço para que outros módulos sejam configurados. Outro motivo para realocar módulos de hardware, ao invés de sintetizá-los novamente para uma nova posição do dispositivo, é que os módulos podem possuir restrições temporais que só podem ser obtidas depois de demoradas etapas de posicionamento e roteamento. Neste caso o módulo é sintetizado uma única vez, sendo posteriormente realocado para alguma posição do sistema com a vantagem de manter as suas características temporais. Módulos que possuem alta taxa de comunicação com outros módulos longínquos dentro do sistema podem ser realocados para ocuparem posições próximas, reduzindo o congestionamento do meio de comunicação, aumentando a eficiência do sistema como um todo.

Embora realocação seja uma boa alternativa para solucionar o problema de fragmentação do hardware, a implementação desta técnica envolve o tratamento de uma série de questões para que ela efetivamente possa ser utilizada em SDRs: (*i*) como descobrir se o sistema está fragmentado? (*ii*) se estiver fragmentado, como salvar o estado atual do módulo para continuá-lo do ponto onde havia parado quando for posicionado em uma nova posição? (*iii*) como descobrir uma nova posição para realocar o módulo? (*iv*) quem será responsável por gerar um novo bitstream parcial com o módulo realocado? um computador hospedeiro, um processador do sistema ou um módulo dedicado do sistema? (*v*) se for um processador do sistema ou um módulo dedicado do sistema, como ativar a reconfiguração de dentro do próprio sistema para que o módulo realocado reinicie a sua execução? (*vi*) se nem mesmo realocando outros módulos existir espaço suficiente para um módulo que deve entrar em execução no sistema, como escolher um módulo do sistema a ser retirado? (*vii*) a forma do módulo a ser realocado pode ser modificada?

A resposta a estas questões tem motivado o estudo em quatro diferentes linhas de pesquisa: (*i*) controladores de configuração; (*ii*) sistemas auto-reconfiguráveis; (*iii*) escalonamento de módulos de hardware; (*iv*) posicionamento de módulos. Englobando uma ou mais destas características, diversos autores justificam o trabalho na área de Sistemas Operacionais para Hardware Reconfigurável (RHOS – *Reconfigurable Hardware Operating Systems*) [WAL03][NOL03][MER98].

De tantas questões a serem tratadas sobre este assunto, este Capítulo apresenta apenas uma introdução sobre realocação e fatores relacionados a configuração parcial e dinâmica de módulos realocados. A seção 4.1 apresenta algumas características que o SDR deve respeitar para aceitar módulos realocáveis. Um estudo sobre realocação e desfragmentação é apresentado na seção 4.2. A seção 4.3 apresenta o ParBit, uma ferramenta elaborada por Horta [HOR04] para permitir a realocação de módulos de hardware em dispositivos Virtex.

4.1 Características de um SDR que aceita a realocação de módulos

Para que um SDR suporte a realocação de módulos é necessário que as interfaces dos módulos possuam a mesma semântica que os componentes isoladores do sistema. Assim os módulos podem ser realocados e a comunicação com o restante do sistema permanece inalterada. Por este motivo é importante criar uma interface padrão entre os módulos de forma que todos utilizem o mesmo protocolo de comunicação.

A tendência do projetista é construir um SDR que se adapte aos módulos que o mesmo deve conectar no sistema. Neste caso o meio de comunicação deve ocupar uma posição fixa do sistema, possuir a sua interface de comunicação com os módulos bem definidos e reservar espaço para que os módulos sejam inseridos no sistema sem sobrepor o meio de comunicação. Quanto mais interfaces de comunicação forem adicionadas ao sistema maior será o número de módulos que podem executar simultaneamente.

Outra questão importante em um SDR que permite realocação é que os módulos devem possuir a interface padrão posicionada em uma determinada borda do módulo. Com freqüência a localização do meio de comunicação dita a posição das interfaces dos módulos. A Figura 20 apresenta um SDR hipotético que utiliza bem a área do dispositivo e consegue mapear todos os pinos de entrada e saída, no entanto necessita que um mesmo módulo seja sintetizado tanto com a interface na borda superior do módulo quanto na borda inferior.

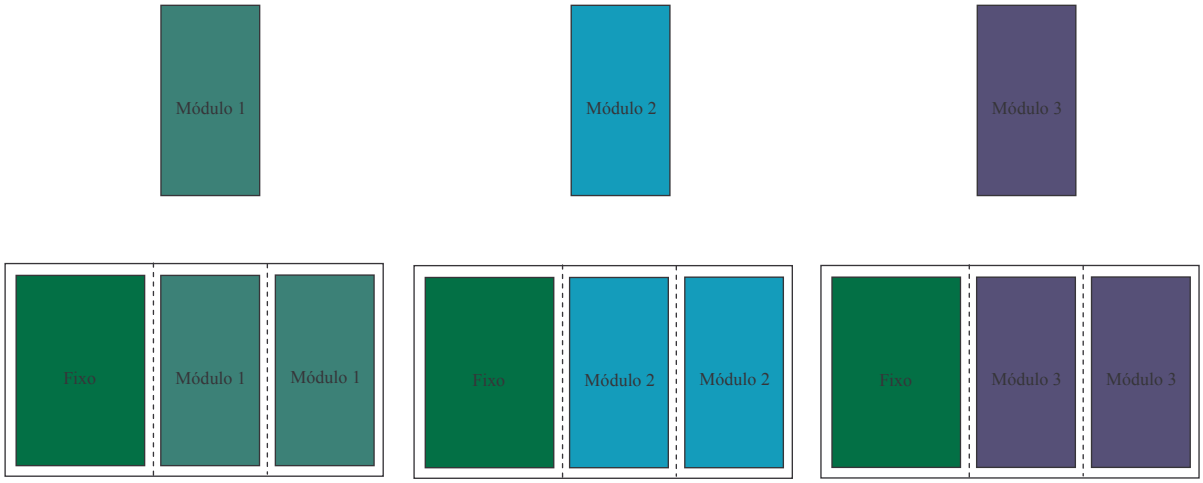


Figura 20 – Sistema hipotético que mapeia todos os pinos de entrada e saída, no entanto, necessita que os módulos sejam gerados com interface de comunicação na parte superior e inferior do mesmo.

A Figura 21 apresenta diagramas de blocos de SDRs que foram implementados sobre dispositivos programáveis da Xilinx. Estes sistemas ainda apresentam uma organização de posicionamento de módulos reconfiguráveis relativamente simples, pois estão restritas a reconfiguração em colunas do dispositivo reconfigurável, conforme visto na Seção 1.1. O fato de toda a coluna ser alterada durante a reconfiguração, inibe a inserção de mais de um módulo reconfigurável na mesma coluna, pois a reconfiguração de um módulo interromperia o funcionamento do outro.

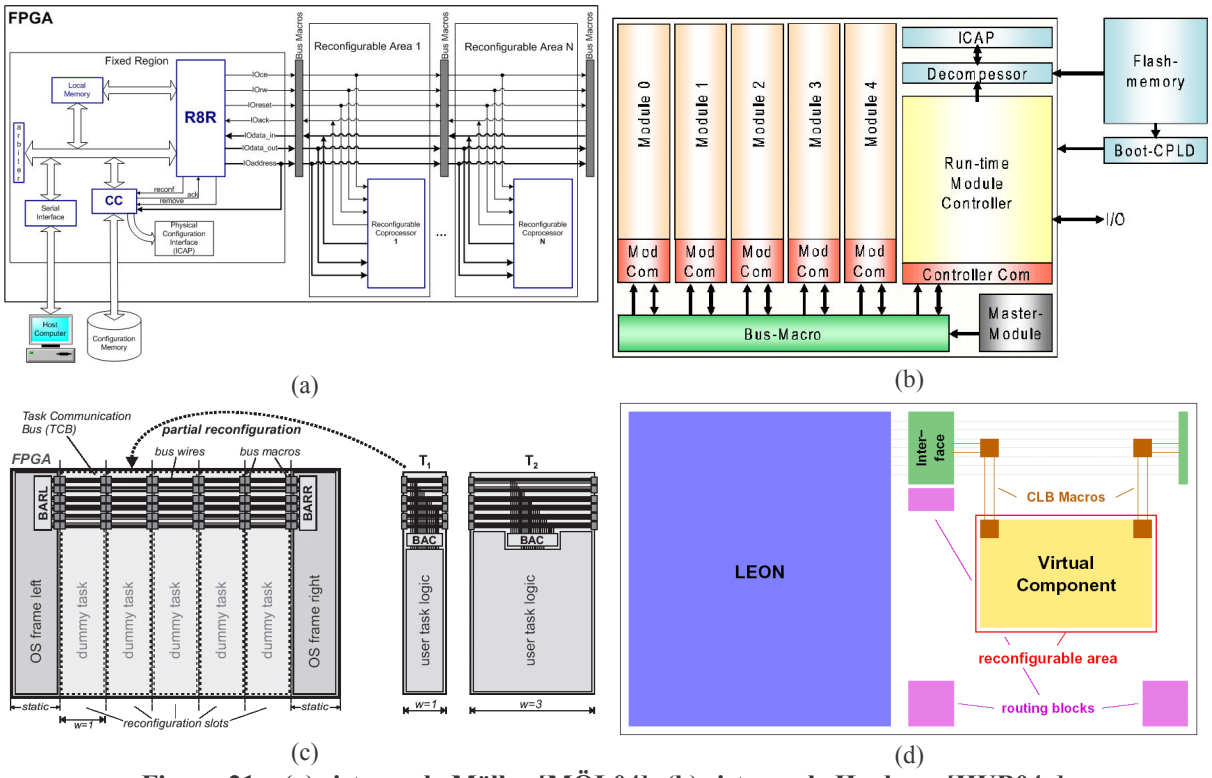


Figura 21 – (a) sistema de Möller [MÖL04]; (b) sistema de Huebner [HUB04a]; (c) sistema de Walder [WAL04]; (d) sistema de Dyer [DYE02a].

4.2 Realocação e desfragmentação

Um caso onde a realocação de configurações em FPGAs pode ser utilizada é ilustrado na Figura 22. Neste caso a configuração inicial é apresentada por quadrados cinza escuro, enquanto quadrados cinza claros apresentam uma nova lógica a ser configurada no FPGA. Quadrados marcados por um **X** mostram células em comum nas duas configurações e que não podem estar presentes simultaneamente em um único FPGA. Entretanto, um FPGA com realocação pode modificar a segunda configuração de uma forma que as células em comum entre as duas configurações não colidam, conforme é ilustrado à direita da Figura 22.

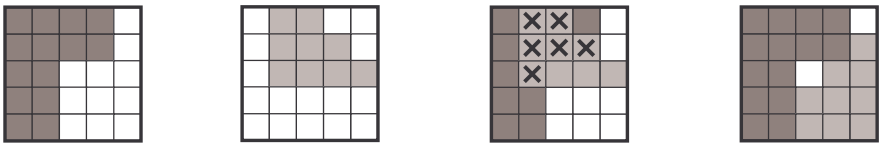


Figura 22 – Utilização de realocação para resolver a fragmentação do dispositivo durante a execução.

Katherine Compton [COM00] elaborou uma série de equações que permitem realocar uma dada configuração a partir de cinco operações básicas (inverter verticalmente, inverter horizontalmente, rotacionar em 90º graus, deslocar na vertical e deslocar na horizontal). A Figura 23 apresenta um pipeline de operações para a realocar o exemplo da Figura 22.

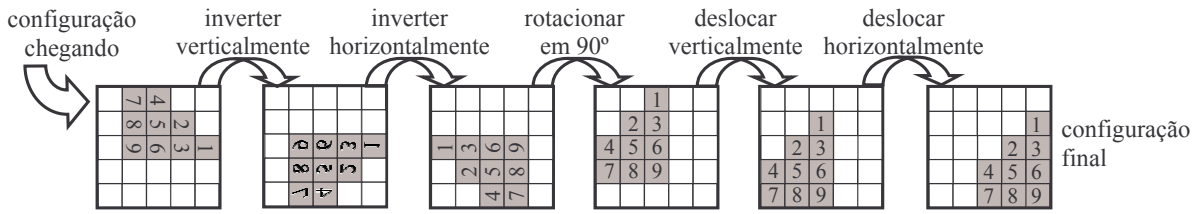


Figura 23 – Pipeline de realocação.

Também é reportado por Compton que desfragmentar configurações bidimensionais é um processo que consome tempo e é normalmente executado em tempo de projeto. Uma estrutura em linhas, onde cada linha seria uma estrutura atômica para configuração do FPGA, removeria a complexidade da realocação e desfragmentação. Isto se dá porque reduz as configurações em objetos unidimensionais, onde a única variação da área de configuração permitida é no número de linhas utilizadas. Operações de rotação ou deslocamento não são mais necessárias. A única operação requerida para realocação de uma configuração é a mudança no deslocamento vertical.

4.3 PARBIT

O PARBIT (*PARtial BItfile Transformer*) é uma ferramenta que funciona por linha de comando e foi desenvolvida por Horta [HOR04] para construir bitstreams parciais e realocar módulos de hardware em dispositivos Virtex. A ferramenta não utiliza o JBits para acesso ao bitstream, fazendo acesso direto ao arquivo de configuração, conforme abordado no Application Note 151 da Xilinx [XIL04e]. O PARBIT aceita dois modos de operação básicos:

- Slice: o usuário informa quais colunas de CLBs de um bitstream total de entrada deve ser construído um bitstream parcial. A localização do bitstream parcial é sempre a mesma que a posição selecionada do bitstream total.
- Block: o usuário informa qual a posição de um bitstream original deve ser copiada para qual posição de um bitstream alvo. Este modo de operação permite que a área selecionada do bitstream original seja de um dispositivo diferente que o bitstream alvo, desde que ambos os bitstreams tenham sido sintetizados para uma mesma família de FPGAs.

5 ESTUDOS DE CASO

Os Capítulos anteriores apresentaram etapas obrigatórias e opcionais para a geração de SDRs independentes do fluxo a serem seguidos. Este Capítulo apresenta dois fluxos para a construção de SDRs. O primeiro é o Projeto Modular [LIM04] apresentado na Seção 5.1. O segundo, apresentado na Seção 5.2, é o fluxo de projeto de Palma [PAL02b]. Ambas as Seções focam no fluxo de geração de SDRs, por este motivo não analisam os resultados das aplicações executadas sobre os SDRs. Uma análise dos prós e contras de cada um dos fluxos estudados são apresentados na Seção 5.3.

5.1 Fluxo de Projeto Modular

O fluxo de Projeto Modular [XIL04c] foi criado para que equipes distintas trabalhem de forma distribuída em um mesmo projeto, onde cada equipe produz um ou mais módulos e um projetista "centralizador" instancia os módulos de cada equipe em um único sistema. As vantagens desta abordagem são que o fluxo não necessita de roteamento manual e que cada equipe pode gerar o bitstream do seu módulo sem que os módulos de outras equipes afetem os seus. Ao final do fluxo completo do Projeto Modular têm-se os arquivos de inicialização do sistema (bitstream total) e os arquivos de configuração dos módulos (bitstreams parciais).

Antes de executar o fluxo de Projeto Modular em si, abordado na Seção 5.1.2, alguns passos devem ser efetuados para que os arquivos de entrada do fluxo estejam corretamente criados e posicionados em uma árvore de diretórios sugerida pela Xilinx [LIM04]. Estes passos de preparação para execução do fluxo de Projeto Modular são apresentados na Seção 5.1.1. A Seção 5.1.3 apresenta a implementação de uma calculadora reconfigurável a partir do fluxo de Projeto Modular.

5.1.1 Preparação para execução do fluxo de Projeto Modular

As seis subseções subseqüentes apresentam os passos que devem ser efetuados antes da execução do fluxo de Projeto Modular.

5.1.1.1 Modularização do sistema

Esta etapa pressupõe que o sistema, sem características reconfiguráveis, já foi prototipado pelo menos uma vez, conforme abordado nos primeiros dois passos da Seção 1.3, e que o sistema já está dividido em módulos.

5.1.1.2 Inserção de macros, LUTs e DCMs

Nesta etapa são adicionados os componentes isoladores, conforme abordado no terceiro passo da Seção 1.3 e aprofundado no Capítulo 2, e alguns componentes específicos do dispositivo no arquivo de mais alta hierarquia do projeto. Exemplos destes componentes são as LUTs e os DCMs. DCMs são utilizados para evitar o escorregamento de relógio. Recomenda-se utilizar constantes VCC e GND geradas a partir de LUTs porque estes componentes podem ter a sua posição restringida. Quando não são feitas restrições de posicionamento, pode ocorrer de a constante ser buscada de um componente que se encontra em outra área reconfigurável e assim causar uma desconexão do fio no momento da reconfiguração.

5.1.1.3 Criação de arquivos de restrições do usuário

A criação de arquivos de restrições do usuário, abordado no passo 4 da Seção 1.3 e aprofundado na Seção 3.1, pode ser feita manualmente, com a ferramenta Floorplanner ou com a ferramenta PACE. A ferramenta Floorplanner restringe o posicionamento de módulos, adicionando ao arquivo UCF linhas como ***AREA_GROUP = usergroup RANGE=SLICE_X3Y1: SLICE_X3Y33*** e ***INST "bus_macroX" LOC = "SLICE_X38Y68"***. A ferramenta PACE restringe o posicionamento de pinos adicionando linhas ao UCF como ***NET "clock" LOC = "A11"***. Este passo deve ser executado para todos os componentes inseridos na Seção 5.1.1.2 e para os pinos de entrada e saída do projeto.

5.1.1.4 Replicação de códigos de entrada

Caso um mesmo módulo seja utilizado em mais de uma área reconfigurável é necessário replicar o código fonte VHDL para cada área reconfigurável, gerando um VHDL para cada combinação. Isto é necessário porque o Projeto Modular utiliza o nome do arquivo de mais alta hierarquia do módulo para construir um diretório específico para cada módulo de cada área reconfigurável.

5.1.1.5 Criação de scripts de síntese

Para facilitar a tarefa de síntese de cada um dos módulos do sistema é aconselhável construir scripts que executam a ferramenta de síntese lógica para cada um dos arquivos VHDL de entrada do sistema. Os arquivos EDIF são criados após a execução deste script.

5.1.1.6 Criação e cópia de diretórios do Projeto Modular

O Application Note 290 da Xilinx [LIM04] sugere uma determinada árvore de diretórios para a organização de um SDR implementado a partir do fluxo de Projeto Modular. Para que o fluxo seja executado com sucesso é recomendável criar os diretórios conforme o modelo sugerido e inserir os arquivos de entrada nesses diretórios.

5.1.2 Fases do Projeto Modular

O fluxo de Projeto Modular nada mais é que a aplicação de uma série de ferramentas da Xilinx segundo uma seqüência e parâmetros que permitem modularizar o sistema, validar individualmente cada um dos módulos e finalmente unir todos os módulos. Para auxiliar no processo de implementação de SDRs utilizando este fluxo gerou-se um *script* que invoca as ferramentas necessárias com os parâmetros voltados à criação de SDRs.

O fluxo de Projeto Modular pode ser dividido em três fases. A fase de orçamento inicial determina qual é a estrutura do arquivo de nível hierárquico mais alto do projeto. Na fase de implementação dos módulos ativos cada módulo é sintetizado e implementado separadamente. Na fase de montagem final são gerados os projetos com os módulos já implementados, utilizados para inicializar o sistema. A Figura 24 apresenta as fases do Projeto Modular.

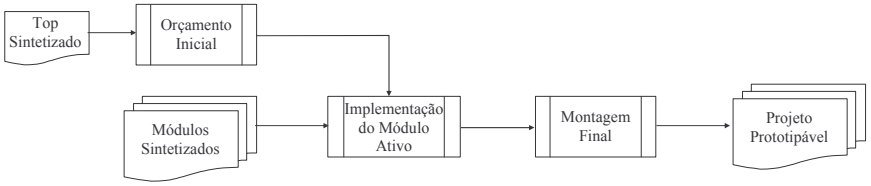


Figura 24 – Fluxo de Projeto Modular.

5.1.2.1 Fase de orçamento inicial

A fase de orçamento inicial tem como principal objetivo montar a estrutura do arquivo de mais alto nível do projeto. Nesta fase é feita a definição de quais pinos serão utilizados do FPGA e onde estão posicionados os módulos e componentes inseridos no arquivo de mais alto nível do projeto. A Figura 25 apresenta o fluxo de execução da fase de orçamento inicial.

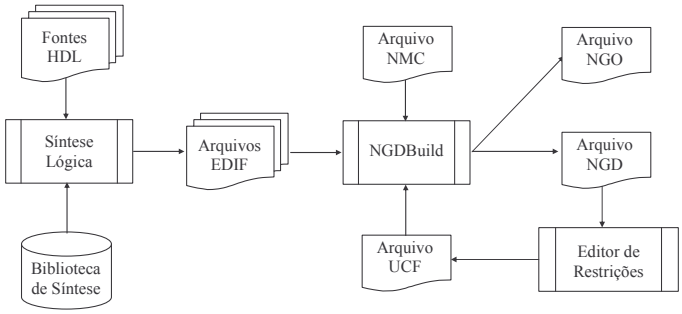


Figura 25 – Fluxo da fase de orçamento inicial do Projeto Modular que deve ser executado para cada projeto pertencente ao SDR.

A única ferramenta executada nesta fase é o NGDBuild que posiciona os módulos do projeto como caixas preta e fixa as componentes inseridos na Seção 5.1.1.2. As entradas desta ferramenta são o arquivo EDIF do arquivo de mais alta hierarquia do projeto, o arquivo NMC que contém a bus macro pré-roteada e o arquivo UCF de restrições do usuário. Arquivos NGD e NGO são gerados, porém apenas este último será utilizado como entrada da próxima fase.

5.1.2.2 Fase de implementação do módulo ativo

Na fase de implementação do módulo ativo são feitos o mapeamento, posicionamento e roteamento de cada um dos módulos do sistema, tanto fixos quanto reconfiguráveis. A entrada desta fase é o arquivo EDIF de um módulo e o arquivo NGO gerado na fase de orçamento inicial. Estes arquivos são utilizados pelo NGDBuild para gerar o arquivo NGD que possui a descrição lógica do módulo em função de primitivas da Xilinx. O arquivo NGD do módulo é mapeado pela ferramenta MAP para um arquivo NCD, não roteado, que possui a descrição do módulo em função dos recursos específicos do dispositivo. A ferramenta PAR posiciona e roteia os recursos do módulo, gerando um novo arquivo com extensão NCD. O arquivo NCD é utilizado como entrada para a ferramenta BITGen, que gera o bitstream parcial do módulo, e para a ferramenta PIMCreate, que copia e renomeia arquivos de entrada de acordo com as necessidades do Projeto Modular. A Figura 26 ilustra o fluxo executado nesta fase.

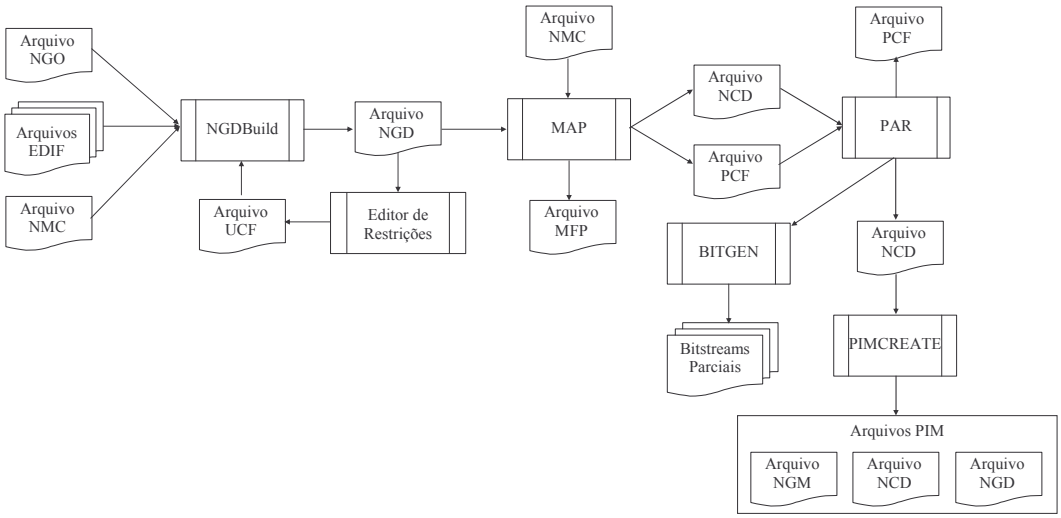


Figura 26 – Fluxo da fase de implementação do módulo ativo do Projeto Modular que deve ser executado para cada módulo do SDR.

5.1.2.3 Fase de montagem final

Na fase de montagem final todos os módulos implementados fisicamente que fazem parte do arquivo de mais alta hierarquia do projeto 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 da fase de montagem final é ilustrado na Figura 27.

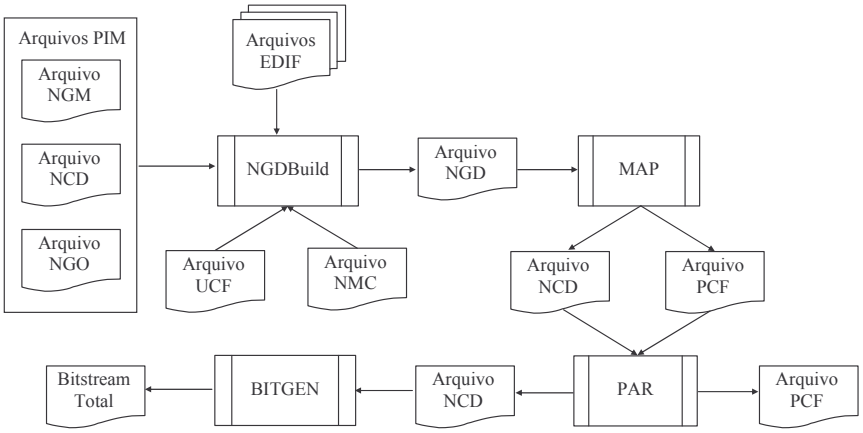


Figura 27 – Fluxo da fase montagem final do Projeto Modular que deve ser executado para cada projeto do SDR.

A primeira ferramenta utilizada nesta fase é o NGDBuild. Esta ferramenta utiliza como entrada o arquivo de restrições do usuário (UCF), o arquivo da bus macro (NMC), o arquivo de mais alta hierarquia do projeto (EDIF) e todo o conjunto de arquivos PIM gerados na fase anterior. As ferramentas MAP e PAR são utilizadas para mapear e posicionar os módulos no arquivo de mais alta hierarquia do projeto e fazer o roteamento entre os módulos já roteados. Por fim a ferramenta BITGen é utilizada para gerar o bitstream total que permite inicializar o FPGA.

5.1.3 Estudo de Caso – Calculadora Reconfigurável

O estudo de caso apresentado nesta Seção já foi abordado na dissertação de mestrado de Brião [BRI04] e aqui é apresentado por ser um sistema simples e que permite focar no fluxo de Projeto Modular em si. O estudo de caso consiste em uma calculadora que pode possuir dois módulos reconfiguráveis. Até o presente momento foram implementados os módulos de adição (*adder*) e subtração (*sub*). O sistema também possui um módulo não reconfigurável responsável por efetuar a entrada e saída de dados do dispositivo (*capture*). A Figura 28 apresenta um diagrama de blocos do sistema.

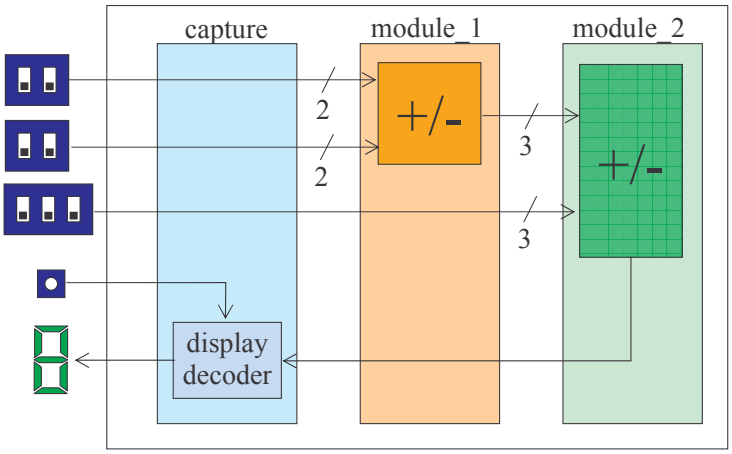


Figura 28 – Diagrama de blocos da calculadora reconfigurável para utilização com o fluxo de Projeto Modular.

O módulo *capture* é responsável por fazer a interface entre a entrada e a saída do FPGA com os módulos que fazem o cálculo dos valores requisitados. As entradas do capture são chaves que representam os operandos dos módulos que efetuam o cálculo e um botão que atualiza o display de saída. O primeiro módulo reconfigurável executa a soma ou a subtração de dois operandos de dois bits. O segundo módulo reconfigurável executa a soma ou a subtração de dois operandos de três bits e retorna o resultado para o módulo *capture*.

As próximas Seções seguem o fluxo de Projeto Modular da mesma forma como ele foi abordado nas Seções 5.1.1 e 5.1.2, partindo desde a preparação da execução do fluxo até a fase de montagem final do mesmo.

5.1.3.1 Módulos do sistema

O sistema da calculadora reconfigurável possui dois projetos denominados *top* e *top1*. O projeto *top* instancia dois módulos *adder* e o projeto *top1* instancia dois módulos *sub*. A Figura 29 e a Figura 30 apresentam os códigos VHDL do projeto *top*, sem características reconfiguráveis.

```vhdl
entity top is
port(
   reset:    in  std_logic;
   clock:    in  std_logic;
   input_A:  in  std_logic_vector(1 downto 0);
   input_B:  in  std_logic_vector(1 downto 0);
   input_C:  in  std_logic_vector(2 downto 0);
   get_data: in  std_logic;
   displayA: out std_logic_vector(6 downto 0)
);
end top;

architecture arq_top of top is

signal dadoA: std_logic_vector(1 downto 0);
signal dadoB: std_logic_vector(1 downto 0);
signal dadoC: std_logic_vector(2 downto 0);
signal res1:  std_logic_vector(2 downto 0);
signal res2:  std_logic_vector(3 downto 0);

begin

   Interface: entity capture
   port map (
          input_A => input_A,
          input_B => input_B,
          input_C => input_C,
         output_A => dadoA,
         output_B => dadoB,
         output_C => dadoC,
            reset => reset,
            clock => clock,
        get_data => get_data,
           input => res2,
        displayA => displayA
    );

   Module_1: entity adder2bit
   port map (
         input_A => dadoA,
         input_B => dadoB,
         output  => res1
   );

   Module_2: entity adder3bit
   port map (
         input_A => res1,
         input_B => dadoC,
         output  => res2
   );

end arq_top;
```

(a)

```vhdl
entity capture is
port(
   input_A:  in  std_logic_vector(1 downto 0);
   input_B:  in  std_logic_vector(1 downto 0);
   input_C:  in  std_logic_vector(2 downto 0);
   output_A: out std_logic_vector(1 downto 0);
   output_B: out std_logic_vector(1 downto 0);
   output_C: out std_logic_vector(2 downto 0);
   reset:    in  std_logic;
   clock:    in  std_logic;
   get_data: in  std_logic;
   input:    in  std_logic_vector(3 downto 0);
   displayA: out std_logic_vector(6 downto 0)
);
end capture;

architecture arq_capture of capture is
begin

process(clock, get_data, reset)
begin
   if reset ='0' then
         output_A <= (others=>'0');
         output_B <= (others=>'0');
         output_C <= (others=>'0');
   else
         if clock = '1' and clock'event then
               if (get_data = '0') then
                     output_A <= input_A;
                     output_B <= input_B;
                     output_C <= input_C;
               end if;
         end if;
   end if;
end process;

-- decoder used to V2MB1000 board
with input select
displayA <=
         "0111111" when "0000",
         "0000110" when "0001",
         "1011011" when "0010",
         "1001111" when "0011",
         "1100110" when "0100",
         "1101101" when "0101",
         "1111101" when "0110",
         "0000111" when "0111",
         "1111111" when "1000",
         "1101111" when "1001",
         "1000000" when "1111",
         "0110000" when others;
end arq_capture;
```

(b)

Figura 29 – (a) código VHDL do *top.vhd*; (b) código VHDL do módulo *capture.vhd*.

```vhdl
entity adder2bit is
 port(input_A: in std_logic_vector(1 downto 0);
      input_B: in std_logic_vector(1 downto 0);
      output: out std_logic_vector(2 downto 0)
     );
end adder2bit;

architecture arq_adder2bit of adder2bit is

signal output_sig: std_logic_vector (1 downto 0);
signal carry_sig: std_logic;

begin

  process(input_A, input_B)
  variable carry, prev_carry: std_logic;
  begin
     for w in 0 to 1 loop

         if w = 0 then
             prev_carry :='0';
             carry:='0';
         end if;

         output_sig(w) <= input_A(w) xor
                          input_B(w) xor
                          carry;

         carry := (input_A(w) and input_B(w)) or
                  (input_A(w) and carry) or
                  (input_B(w) and carry);
     end loop;

     carry_sig <= carry;

  end process;

  output <= carry_sig & output_sig;

end arq_adder2bit;
```

(a)

```vhdl
entity adder3bit is
 port(input_A: in std_logic_vector(2 downto 0);
      input_B: in std_logic_vector(2 downto 0);
      output: out std_logic_vector(3 downto 0)
     );
end adder3bit;

architecture arq_adder3bit of adder3bit is

signal output_sig: std_logic_vector (2 downto 0);
signal carry_sig: std_logic;

begin

  process(input_A, input_B)
  variable carry, prev_carry: std_logic;
  begin
     for w in 0 to 2 loop

         if w = 0 then
             prev_carry :='0';
             carry:='0';
         end if;

         output_sig(w) <= input_A(w) xor
                          input_B(w) xor
                          carry;

         carry := (input_A(w) and input_B(w)) or
                  (input_A(w) and carry) or
                  (input_B(w) and carry);
     end loop;

     carry_sig <= carry;

  end process;

  output <= carry_sig & output_sig;

end arq_adder3bit;
```

(b)

Figura 30 – (a) código VHDL do módulo *adder2bit.vhd*; (b) código VHDL do módulo *adder3.bit*.

5.1.3.2 Inserção de macros, LUTs e DCMs

Para tornar o projeto *top*, apresentado na Seção 5.1.3.1, um projeto reconfigurável, é necessário modificar apenas o arquivo *top.vhd*, inserindo os componentes apresentados na Seção 5.1.1.2. A Figura 31 apresenta o arquivo *top.vhd* com os componentes que permitem a reconfiguração do sistema já inseridos.

```
001 entity top is
002 port
003 (
004   reset:    in  std_logic;
005   clock:    in  std_logic;
006   input_A:  in  std_logic_vector(1 downto 0);
007   input_B:  in  std_logic_vector(1 downto 0);
008   input_C:  in  std_logic_vector(2 downto 0);
009   get_data: in  std_logic;
010   displayA: out std_logic_vector(6 downto 0)
011 );
012
013 attribute preserve_driver: boolean;
014 attribute preserve_signal: boolean;
015 end top;
016
017 architecture arch of top is
018
019 -- sinais que devem ser ligados a Bus Macro para inibir a remoção
020 signal going_nowhere1, going_nowhere2, going_nowhere3: std_logic;
021
022 -- constantes
023 signal VCC_fixed, VCC_reconfig_1, VCC_reconfig_2: std_logic;
024 signal GND_fixed, GND_reconfig_1, GND_reconfig_2: std_logic;
025
026 -- sinais conectados as Bus Macros
027 signal  inBM_reconfig_1_input_A:     std_logic_vector(1 downto 0);
028 signal outBM_reconfig_1_input_A:     std_logic_vector(1 downto 0);
029 signal  inBM_reconfig_1_input_B:     std_logic_vector(1 downto 0);
030 signal outBM_reconfig_1_input_B:     std_logic_vector(1 downto 0);
031 signal  inBM_reconfig_1_pass_C:      std_logic_vector(2 downto 0);
032 signal outBM_reconfig_1_pass_C:      std_logic_vector(2 downto 0);
033 signal  inBM_reconfig_1_pass_result: std_logic_vector(3 downto 0);
034 signal outBM_reconfig_1_pass_result: std_logic_vector(3 downto 0);
035 signal  inBM_reconfig_2_input_A:     std_logic_vector(2 downto 0);
036 signal outBM_reconfig_2_input_A:     std_logic_vector(2 downto 0);
037 signal outBM_reconfig_2_input_C:     std_logic_vector(2 downto 0);
038 signal outBM_capture_pass_result:    std_logic_vector(3 downto 0);
039
040 -- inibe a remoção da LUT
041 attribute NOOPT: boolean;
042 attribute NOOPT of LUT1: component is TRUE;
043
044 -- inibe a remoção da Bus Macro
045 attribute PRESERVE_DRIVER of going_nowhere1: signal is TRUE;
046 attribute PRESERVE_SIGNAL of going_nowhere1: signal is TRUE;
047 attribute PRESERVE_DRIVER of going_nowhere2: signal is TRUE;
048 attribute PRESERVE_SIGNAL of going_nowhere2: signal is TRUE;
049 attribute PRESERVE_DRIVER of going_nowhere3: signal is TRUE;
050 attribute PRESERVE_SIGNAL of going_nowhere3: signal is TRUE;
052 begin
053
054 internal_VCC_fixed: LUT1
055 generic map (INIT => b"11")
056 port map (
057 O=>VCC_fixed,
058 I0=>VCC_fixed);
059
060 Internal_VCC_reconfig_1: LUT1
061 generic map (INIT => b"11")
062 port map (
063 O=>VCC_reconfig_1,
064 I0=> VCC_reconfig_1);
065
066 Internal_VCC_reconfig_2: LUT1
067 generic map (INIT => b"11")
068 port map (
069 O=>VCC_reconfig_2,
070 I0=> VCC_reconfig_2);
071
072 GND_fixed  <= not VCC_fixed;
073 GND_reconfig_1 <= not VCC_reconfig_1;
074 GND_reconfig_2 <= not VCC_reconfig_2;
075
076 Fixed_Logic: capture port map (
077 input_A => input_A,
078 input_B => input_B,
079 input_C => input_C,
080 output_A => inBM_reconfig_1_input_A,
081 output_B => inBM_reconfig_1_input_B,
082 output_C => inBM_reconfig_1_pass_C,
083 reset => reset,
084 clock => clock,
085 get_data => get_data,
086 input_reconfig => outBM_capture_pass_result,
087 displayA => displayA );
088
089 Reconfig_1: adder2bit port map (
090 input_A => outBM_reconfig_1_input_A,
091 input_B => outBM_reconfig_1_input_B,
092 output => inBM_reconfig_2_input_A );
093
094 Reconfig_2: adder3bit port map (
095 input_A => outBM_reconfig_2_input_A,
096 input_B => outBM_reconfig_2_input_C,
097 output => inBM_reconfig_1_pass_result );
```

```
099 BM_capture_to_reconfig_1_Input_C:
100 bm_4b_v2 port map (
101 LI(3) => VCC_fixed,
102 LI(2) => inBM_reconfig_1_pass_C(2),
103 LI(1) => inBM_reconfig_1_pass_C(1),
104 LI(0) => inBM_reconfig_1_pass_C(0),
105 LT(3) => VCC_fixed,
106 LT(2) => GND_fixed,
107 LT(1) => GND_fixed,
108 LT(0) => GND_fixed,
109 RI(3) => VCC_reconfig_1,
110 RI(2) => VCC_reconfig_1,
111 RI(1) => VCC_reconfig_1,
112 RI(0) => VCC_reconfig_1,
113 RT(3) => VCC_reconfig_1,
114 RT(2) => VCC_reconfig_1,
115 RT(1) => VCC_reconfig_1,
116 RT(0) => VCC_reconfig_1,
117 O(3)  => going_nowhere1,
118 O(2)  => outBM_reconfig_1_pass_C(2),
119 O(1)  => outBM_reconfig_1_pass_C(1),
120 O(0)  => outBM_reconfig_1_pass_C(0) );
121
122 BM_reconfig_1_to_reconfig_2_Input_C:
123 bm_4b_v2 port map (
124 LI(3)=> going_nowhere1,
125 LI(2)=> outBM_reconfig_1_pass_C(2),
126 LI(1)=> outBM_reconfig_1_pass_C(1),
127 LI(0)=> outBM_reconfig_1_pass_C(0),
128 LT(3) => GND_reconfig_1,
129 LT(2) => GND_reconfig_1,
130 LT(1) => GND_reconfig_1,
131 LT(0) => GND_reconfig_1,
132 RI(3) => VCC_reconfig_2,
133 RI(2) => VCC_reconfig_2,
134 RI(1) => VCC_reconfig_2,
135 RI(0) => VCC_reconfig_2,
136 RT(3) => VCC_reconfig_2,
137 RT(2) => VCC_reconfig_2,
138 RT(1) => VCC_reconfig_2,
139 RT(0) => VCC_reconfig_2,
140 O(3) => going_nowhere3,
141 O(2) => outBM_reconfig_2_input_C(2),
142 O(1) => outBM_reconfig_2_input_C(1),
143 O(0) => outBM_reconfig_2_input_C(0) );
```

```
145 BM_capture_to_reconfig_1_Input_AB:
146 bm_4b_v2 port map (
147 LI(3)=> inBM_reconfig_1_input_A(1),
148 LI(2)=> inBM_reconfig_1_input_A(0),
149 LI(1)=> inBM_reconfig_1_input_B(1),
150 LI(0)=> inBM_reconfig_1_input_B(0),
151 LT(3) => GND_fixed,
152 LT(2) => GND_fixed,
153 LT(1) => GND_fixed,
154 LT(0) => GND_fixed,
155 RI(3) => VCC_reconfig_1,
156 RI(2) => VCC_reconfig_1,
157 RI(1) => VCC_reconfig_1,
158 RI(0) => VCC_reconfig_1,
159 RT(3) => VCC_reconfig_1,
160 RT(2) => VCC_reconfig_1,
161 RT(1) => VCC_reconfig_1,
162 RT(0) => VCC_reconfig_1,
163 O(3) => outBM_reconfig_1_input_A(1),
164 O(2) => outBM_reconfig_1_input_A(0),
165 O(1) => outBM_reconfig_1_input_B(1),
166 O(0) => outBM_reconfig_1_input_B(0) );
167
168 BM_reconfig_1_to_reconfig_2_Input_A:
169 bm_4b_v2 port map (
170 LI(3)=> VCC_reconfig_1,
171 LI(2)=> inBM_reconfig_2_input_A(2),
172 LI(1)=> inBM_reconfig_2_input_A(1),
173 LI(0)=> inBM_reconfig_2_input_A(0),
174 LT(3) => VCC_reconfig_1,
175 LT(2) => GND_reconfig_1,
176 LT(1) => GND_reconfig_1,
177 LT(0) => GND_reconfig_1,
178 RI(3) => VCC_reconfig_2,
179 RI(2) => VCC_reconfig_2,
180 RI(1) => VCC_reconfig_2,
181 RI(0) => VCC_reconfig_2,
182 RT(3) => VCC_reconfig_2,
183 RT(2) => VCC_reconfig_2,
184 RT(1) => VCC_reconfig_2,
185 RT(0) => VCC_reconfig_2,
186 O(3) => going_nowhere2,
187 O(2) => outBM_reconfig_2_input_A(2),
188 O(1) => outBM_reconfig_2_input_A(1),
189 O(0) => outBM_reconfig_2_input_A(0) );
```

```
191 BM_reconfig_2_to_reconfig_1_result:
192 bm_4b_v2 port map (
193 LI(3) => VCC_reconfig_1,
194 LI(2) => VCC_reconfig_1,
195 LI(1) => VCC_reconfig_1,
196 LI(0) => VCC_reconfig_1,
197 LT(3) => VCC_reconfig_1,
198 LT(2) => VCC_reconfig_1,
199 LT(1) => VCC_reconfig_1,
200 LT(0) => VCC_reconfig_1,
201 RI => inBM_reconfig_1_pass_result,
202 RT(3) => GND_reconfig_2,
203 RT(2) => GND_reconfig_2,
204 RT(1) => GND_reconfig_2,
205 RT(0) => GND_reconfig_2,
206 O => outBM_reconfig_1_pass_result );
207
208 BM_reconfig_1_to_capture_result:
209 bm_4b_v2 port map (
210 LI(3) => VCC_fixed,
211 LI(2) => VCC_fixed,
212 LI(1) => VCC_fixed,
213 LI(0) => VCC_fixed,
214 LT(3) => VCC_fixed,
215 LT(2) => VCC_fixed,
216 LT(1) => VCC_fixed,
217 LT(0) => VCC_fixed,
218 RI => outBM_reconfig_1_pass_result,
219 RT(3) => GND_reconfig_1,
220 RT(2) => GND_reconfig_1,
221 RT(1) => GND_reconfig_1,
222 RT(0) => GND_reconfig_1,
223 O => outBM_capture_pass_result );
224
225 end arch;
```

Figura 31 – Código VHDL do top.vhd com os componentes que permitem a reconfiguração do sistema.

5.1.3.3 Criação de arquivos de restrições do usuário

As áreas reconfiguráveis e os componentes inseridos na Seção 5.1.3.2 no projeto *top* devem ter seus posicionamentos restringidos para que os componentes isoladores ocupem sempre as mesmas posições e para que os módulos não invadam as áreas de outros módulos. Estas restrições de posicionamento são feitos no arquivo UCF, que pode ser editado manualmente ou pela ferramenta Floorplanner. A Figura 32 apresenta o arquivo UCF para o estudo de caso da calculadora reconfigurável. A Figura 33 apresenta a representação gráfica do dispositivo com as restrições de posicionamento inseridas no arquivo UCF apresentado na Figura 32.

```
NET "clock" LOC = "A11";
NET "reset" LOC = "D7"; #Switch 5
NET "get_data" LOC = "A6"; #Switch 6
NET "input_A(1)" LOC = "C5";
NET "input_A(0)" LOC = "C4";
NET "input_B(1)" LOC = "A4";
NET "input_B(0)" LOC = "B4";
NET "input_C(2)" LOC = "C6";
NET "input_C(1)" LOC = "D6";
NET "input_C(0)" LOC = "A5";
NET "displayA(6)" LOC ="E9";
NET "displayA(5)" LOC ="E10";
NET "displayA(4)" LOC ="E8";
NET "displayA(3)" LOC ="E7";
NET "displayA(2)" LOC ="B8";
NET "displayA(1)" LOC ="A8";
NET "displayA(0)" LOC ="B9";
INST "BM_capture_to_reconfig_1_Input_AB"       LOC = "TBUF_X32Y68"  ;
INST "BM_reconfig_1_to_reconfig_2_Input_A"     LOC = "TBUF_X48Y68"  ;
INST "BM_capture_to_reconfig_1_Input_C"        LOC = "TBUF_X32Y60"  ;
INST "BM_reconfig_1_to_reconfig_2_Input_C"     LOC = "TBUF_X48Y60"  ;
INST "BM_reconfig_2_to_reconfig_1_result"      LOC = "TBUF_X48Y40"  ;
INST "BM_reconfig_1_to_capture_result"         LOC = "TBUF_X32Y40"  ;
INST "Internal_VCC_reconfig_2"                 LOC = "SLICE_X57Y58" ;
INST "Internal_VCC_reconfig_2"                 LOCK_PINS;
INST "Internal_VCC_reconfig_1"                 LOC = "SLICE_X42Y52" ;
INST "Internal_VCC_reconfig_1"                 LOCK_PINS;
INST "Internal_VCC_fixed"                      LOC = "SLICE_X19Y66" ;
INST "Internal_VCC_fixed"                      LOCK_PINS;
INST "Fixed_Logic" AREA_GROUP = "AG_Fixed_Logic" ;
AREA_GROUP "AG_Fixed_Logic" RANGE = SLICE_X0Y79:SLICE_X35Y0 ;
AREA_GROUP "AG_Fixed_Logic" MODE = RECONFIG;
INST "Reconfig_1" AREA_GROUP = "AG_Reconfig_1" ;
AREA_GROUP "AG_Reconfig_1" RANGE = SLICE_X36Y79:SLICE_X51Y0 ;
AREA_GROUP "AG_Reconfig_1" MODE = RECONFIG;
INST "Reconfig_2" AREA_GROUP = "AG_Reconfig_2" ;
AREA_GROUP "AG_Reconfig_2" RANGE = SLICE_X52Y79:SLICE_X63Y0 ;
AREA_GROUP "AG_Reconfig_2" MODE = RECONFIG;
```

Figura 32 – Arquivo de restrições do usuário para o estudo de caso da calculadora reconfigurável.

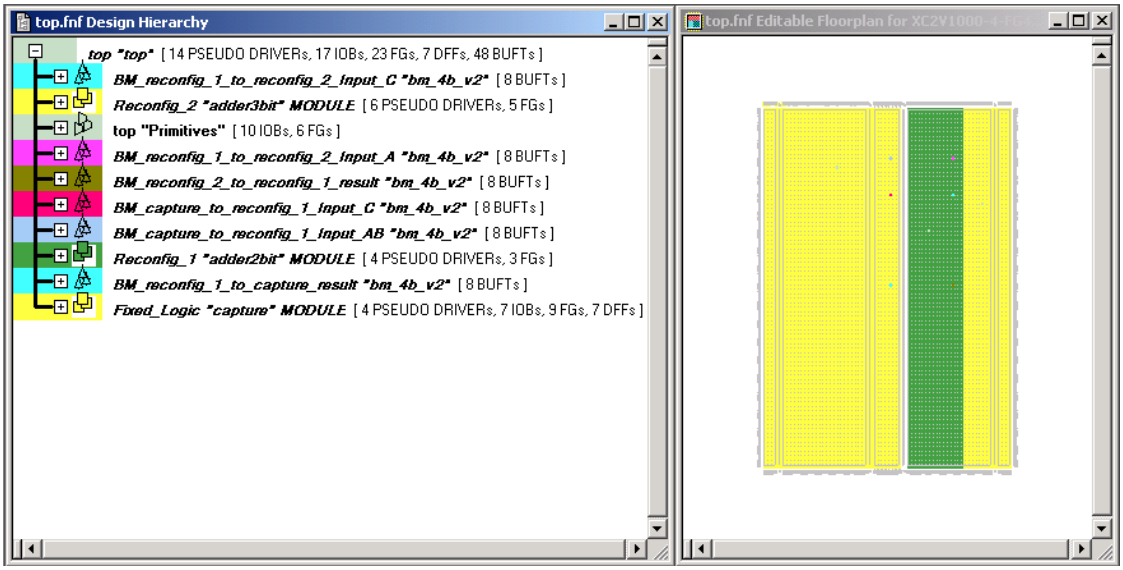


Figura 33 – Tela do Floorplanner para o estudo de caso da calculadora reconfigurável.

5.1.3.4 Replicação de códigos de entrada

Uma vez feito o projeto *top*, fica fácil replicar os passos para o projeto *top1*. O arquivo de mais alta hierarquia do projeto *top1* é o mesmo que o projeto *top*, com a única diferença de instanciar não os módulos *adder*, mas sim os módulos sub. O arquivo de restrições do usuário é exatamente o mesmo para os projeto *top* e *top1*.

5.1.3.5 Criação de scripts de síntese

Para automatizar o processo de síntese do sistema e utilizar os parâmetros corretos para a geração de SDRs é aconselhável criar scripts tanto para os módulos quanto para o arquivo de mais alta hierarquia dos projetos. A Figura 34 apresenta os scripts de síntese para alguns arquivos VHD do estudo de caso da calculadora reconfigurável utilizando a ferramenta Leonardo Spectrum. A Figura 34(a) apresenta um script de síntese para um *top*, que neste caso é o projeto *top.vhd*. A Figura 34(b) apresenta um script de síntese para um módulo, que neste caso é o *adder2bit.vhd*. Estes scripts devem ser replicados segundo estes modelos para cada arquivo VHD de entrada.

```
01 set part 2V1000-4fg456
02 load_library xcv2
03 cd c:/2reconfig/hdl/top
04 read ./top.vhd
05 set blackbox Fixed_Logic
06 set blackbox Reconfig_1
07 set blackbox Reconfig_2
08 optimize -target xcv2 -chip -hier preserve
09 present_design .work.top.arq_top
10 auto_write -format edf top.edf
```

(a)

```
01 set part 2V1000-4fg456
02 load_library xcv2
03 cd c:/2reconfig/hdl/top
04 read ./adder2bit.vhd
05 optimize -target xcv2 -macro -hier preserve
06 present_design .work.adder2bit.arq_adder2bit
07 auto_write -format edf adder2bit.edf
```

(b)

Figura 34 – (a) script de síntese lógica do um *top*; (b) script de síntese lógica de um módulo.

5.1.3.6 Criação e cópia de diretórios do Projeto Modular

Nesta etapa deve-se colocar todos os arquivos que serão utilizados na execução do fluxo do Projeto Modular nos diretórios recomendados pela Xilinx. Para este estudo de caso os diretórios e os arquivos devem estar organizados conforme apresentado na Figura 35.

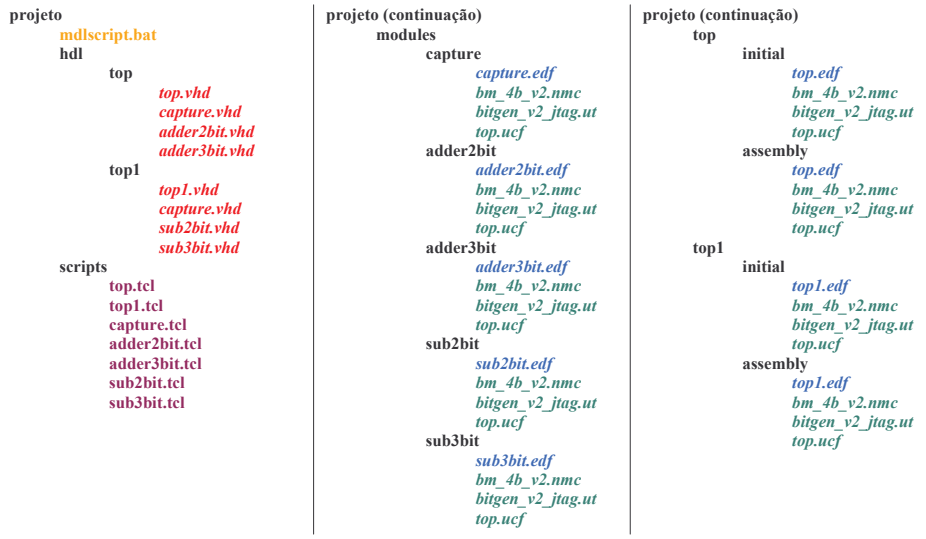


Figura 35 – Organização de diretórios sugerida para a execução do fluxo de Projeto Modular.

5.1.3.7 Fase de orçamento inicial

Antes da fase de orçamento inicial alguns parâmetros devem ser executados para que as ferramentas da Xilinx se configurem para o Projeto Modular. Estes parâmetros são apresentados nas linhas de 1 a 3 do script de execução do Projeto Modular (Figura 36), que deve ser gerado pelo projetista. As linhas de 5 a 7 e de 9 a 11 da Figura 36 apresentam, respectivamente, a execução da fase de orçamento inicial dos dois projetos de mais alta hierarquia do SDR, denominados *top* e *top1*. Na fase de orçamento inicial, como explicado na Seção 5.1.2.1, apenas a ferramenta NGDBuild é executada para gerar a estrutura do projeto.

```
01 set XIL_MAP_ALLOW_MULTI_OUT_PORTS=TRUE
02 set XIL_MAP_MULTI_TBUF_PORTS=TRUE
03 set XIL_MAP_LOCWARN=TRUE
04
05 cd top\initial
06 ngdbuild -p xc2v1000fg456-4 -modular initial top.edf
07 cd ..\..
08
09 cd top1\initial
10 ngdbuild -p xc2v1000fg456-4 -modular initial top.edf
11 cd ..\..
12
```

Figura 36 – Início do script do Projeto Modular para o estudo de caso da calculadora reconfigurável. Fase de orçamento inicial.

5.1.3.8 Fase de implementação do módulo ativo

Nesta fase cada um dos módulos do sistema, tanto fixos quanto reconfiguráveis, são completamente implementados. A Figura 37 apresenta os cinco trechos do script que representam a execução da fase de implementação do módulo ativo para cada um dos módulos do sistema. Cada uma das ferramentas executadas são explicadas na Seção 5.1.2.2.

```
13 cd modules
14
15 cd adder2bit
16 ngdbuild -p xc2v1000fg456-4 -modular module -active adder2bit ..\..\top\initial\top.ngo
17 map -cm area top.ngd -o top_map.ncd top.pcf
18 par -rl 5 -w top_map.ncd top.ncd top.pcf
19 bitgen -d -f bitgen_v2_jtag.ut -g ActiveReconfig:yes top.ncd
20 pimcreate -ncd top.ncd -ngm top_map.ngm ..\..\pims
21 cd ..
22
23 cd adder3bit
24 ngdbuild -p xc2v1000fg456-4 -modular module -active adder3bit ..\..\top\initial\top.ngo
25 map -cm area top.ngd -o top_map.ncd top.pcf
26 par -rl 5 -w top_map.ncd top.ncd top.pcf
27 bitgen -d -f bitgen_v2_jtag.ut -g ActiveReconfig:yes top.ncd
28 pimcreate -ncd top.ncd -ngm top_map.ngm ..\..\pims
29 cd ..
30
31 cd capture
32 ngdbuild -p xc2v1000fg456-4 -modular module -active capture ..\..\top\initial\top.ngo
33 map -cm area top.ngd -o top_map.ncd top.pcf
34 par -rl 5 -w top_map.ncd top.ncd top.pcf
35 bitgen -d -f bitgen_v2_jtag.ut -g ActiveReconfig:yes top.ncd
36 pimcreate -ncd top.ncd -ngm top_map.ngm ..\..\pims
37 cd ..
38
39 cd sub2bit
40 ngdbuild -p xc2v1000fg456-4 -modular module -active sub2bit ..\..\top1\initial\top.ngo
41 map -cm area top.ngd -o top_map.ncd top.pcf
42 par -rl 5 -w top_map.ncd top.ncd top.pcf
43 bitgen -d -f bitgen_v2_jtag.ut -g ActiveReconfig:yes top.ncd
44 pimcreate -ncd top.ncd -ngm top_map.ngm ..\..\pims
45 cd ..
46
47 cd sub3bit
48 ngdbuild -p xc2v1000fg456-4 -modular module -active sub3bit ..\..\top1\initial\top.ngo
49 map -cm area top.ngd -o top_map.ncd top.pcf
50 par -rl 5 -w top_map.ncd top.ncd top.pcf
51 bitgen -d -f bitgen_v2_jtag.ut -g ActiveReconfig:yes top.ncd
52 pimcreate -ncd top.ncd -ngm top_map.ngm ..\..\pims
53 cd ..
54
```

Figura 37 – Continuação do script do Projeto Modular para o estudo de caso da calculadora reconfigurável. Fase de implementação do módulo ativo.

5.1.3.9 Fase de montagem final

Na fase de montagem final para o estudo de caso da calculadora reconfigurável pelo menos um dos projetos *top* ou *top1* deve ser gerado para inicializar o sistema com o bitstream total. A Figura 38 apresenta a fase de montagem final dos dois projetos que compõe o sistema, gerando assim um bitstream total que possui dois módulos de soma e outro bitstream total que possui dois módulos de subtração. Cada uma das ferramentas executadas nesta fase são explicadas na Seção 5.1.2.3.

```
55 cd ..
56
57 cd top\assemble
58 ngdbuild -p xc2v1000fg456-4 -modular assemble -pimpath ..\..\pims top.edf
59 map -cm area top.ngd -o top_map.ncd top.pcf
60 par -rl 5 -w top_map.ncd top.ncd top.pcf
61 bitgen -f bitgen_v2_jtag.ut top.ncd
62 cd ..\..
63
64 cd top1\assemble
65 ngdbuild -p xc2v1000fg456-4 -modular assemble -pimpath ..\..\pims top.edf
66 map -cm area top.ngd -o top_map.ncd top.pcf
67 par -rl 5 -w top_map.ncd top.ncd top.pcf
68 bitgen -f bitgen_v2_jtag.ut top.ncd
69 cd ..\..
```

Figura 38 – Final do script do Projeto Modular para o estudo de caso da calculadora reconfigurável. Fase de montagem final.

5.2 Fluxo de Projeto de Palma

O fluxo de projeto adotado em [PAL02b] foi implementado para permitir a reconfiguração parcial de módulos de hardware e busca minimizar a interface de comunicação dos módulos do sistema. Tenta-se reduzir a interface de comunicação porque durante o desenvolvimento do trabalho as ferramentas disponíveis não possuíam as funcionalidades para reconfiguração parcial e operações com macros. Por este motivo, era necessário utilizar a ferramenta FPGA Editor para garantir manualmente que os diferentes projetos do SDR utilizassem a mesma trilha desde o módulo origem até o módulo destino. Este processo inviabiliza a criação de SDRs, pois não há como salvar o roteamento efetuado para um projeto e reaplicá-lo após uma nova síntese.

A forma encontrada para facilitar a criação de SDRs com este fluxo foi utilizar *tri-states* nas entradas e saídas dos módulos reconfiguráveis. Isto se deve ao fato que as linhas de *tri-states* apresentam um roteamento fixo entre as CLBs vizinhas em dispositivos Virtex, conforme apresentado na Figura 39.

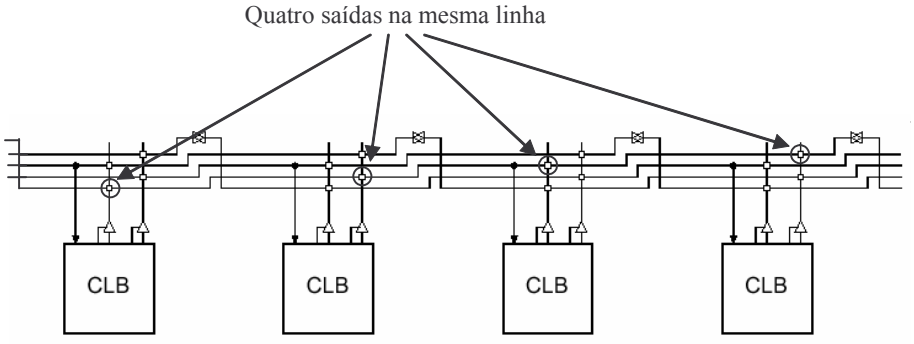


Figura 39 – CLBs possuem roteamento fixo através de linhas *tri-states*.

As linhas *tri-states* dos dispositivos Virtex permitem construir um barramento fixo entre os

diferentes projetos do SDR, assim descartando a necessidade de ter que verificar se o barramento inteiro foi roteado da mesma forma em todos os projetos. No entanto, o problema de rotear fios manualmente persiste na conexão entre o barramento e o módulo reconfigurável, conforme pode ser observado pelas linhas comuns de roteamento apresentadas na Figura 40.

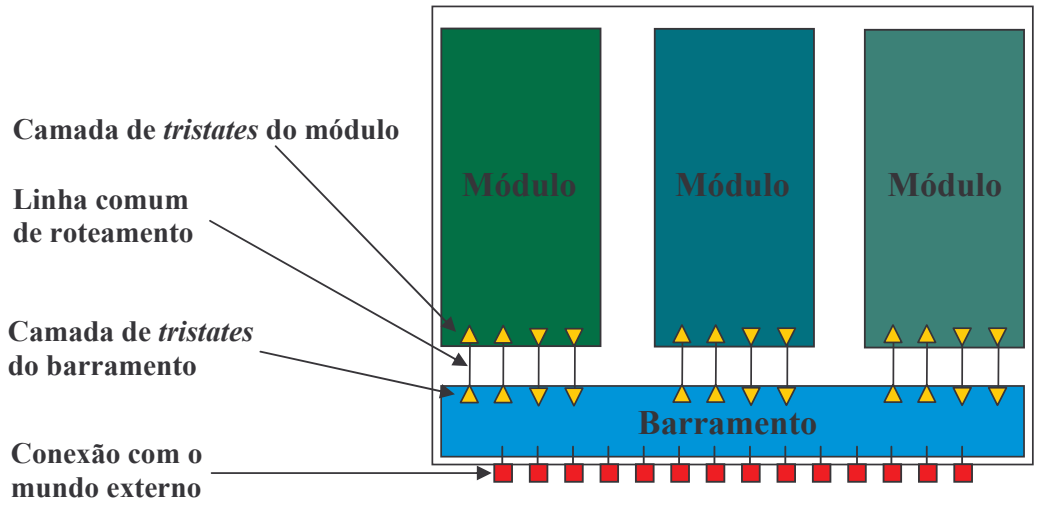


Figura 40 – Estrutura de SDR proposta por Palma.

As duas camadas de *tri-states* são necessárias porque um *tri-state* pertence a uma CLB que possui LUTs, *flip-flops* e roteamento. A Figura 41 apresenta um caso onde ocorre a sobreposição da área de um módulo sobre a lógica do barramento, assim interrompendo o funcionamento do sistema.

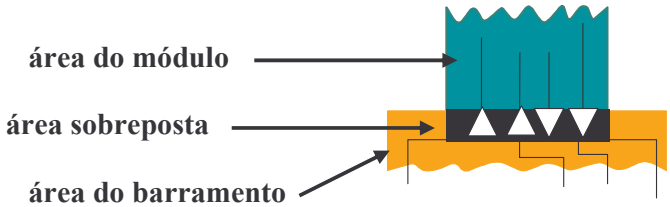


Figura 41 – Conexão do módulo com o barramento através de uma camada de *tri-states*.

Conforme ilustrado na Figura 40, a abordagem de Palma utiliza um barramento localizado na parte inferior do dispositivo como componente isolador entre áreas reconfiguráveis. Os passos para executar o fluxo de construção de SDRs de Palma é abordado na Seção 5.2.1. A Seção 5.2.2 apresenta como estudo de caso uma calculadora reconfigurável, assim como a Seção 5.1.3, com a diferença de utilizar o fluxo de Palma ao invés do Projeto Modular.

5.2.1 Fases do fluxo de projeto de Palma

As seis subseções subseqüentes apresentam os passos que devem ser executados para a construção de um SDR segundo o fluxo proposto por Palma.

5.2.1.1 Modularização do sistema

Esta etapa pressupõe que os módulos a serem inseridos no SDR já tenham sido no mínimo simulados, apresentando um funcionamento correto. Isto é importante, pois esta etapa implicará em

um aumento de complexidade do módulo, dificultando inclusive na sua posterior simulação. A tarefa a ser executada nesta etapa é adaptar o módulo previamente construído e validado com a interface de comunicação dos módulos *Send* e *Receive* disponibilizados por Palma. Estes módulos são responsáveis por fazer a interface com o barramento que será inserido na Seção 5.2.1.2. A Figura 42 apresenta a conexão do módulo do usuário aos módulos *Send* e *Receive*, que por sua vez se conectam ao barramento.

O módulo *Send* é responsável por enviar um pacote de 40 bits (8 bits de endereço, 32 bits de dados) para um módulo destino. Quando a lógica do módulo do usuário ativa o sinal *disp* e disponibiliza um pacote em *word_in*, o módulo Send requisita acesso ao barramento através do sinal *request*. Quando o acesso ao barramento foi liberado ao módulo, o sinal de *grant* será ativado e os bits de *word_in* serão enviados serialmente por *dataOut*.

O módulo *Receive* é responsável por receber serialmente dados enviados por módulos Send. Para isso ele sempre captura os primeiros 8 bits recebidos pelo sinal *dataIn*. Caso o endereço recebido seja igual ao endereço do seu módulo (*address*), os 32 bits restantes serão recebidos, caso contrário serão descartados. Após o completo recebimento, o sinal *disp* é ativado informando ao módulo do usuário a chegada de dados pelo sinal *word_out*.

É recomendável que o usuário simule o módulo gerado para a validação das conexões efetuadas.

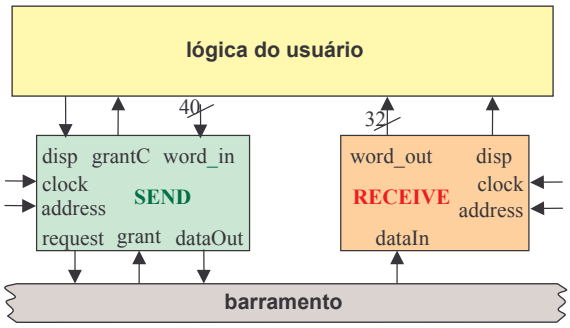


Figura 42 – Interface dos módulos *Send* e *Receive* com a lógica do usuário e com o barramento.

5.2.1.2 Inserção do barramento de Palma

Esta Seção apresenta a geração dos arquivos *top* de todos os projetos do SDR. Recomenda-se que seja seguida a sugestão de organização de arquivos *top* apresentada na **Erro! A origem da referência não foi encontrada.**, para que seja construído o menor número possível de arquivos *top*.

Nesta etapa é necessário instanciar o barramento de Palma no arquivo *top* e tantos módulos quantas forem as áreas reconfiguráveis. Cada área reconfigurável no FPGA é denominada de *slot.* Todos os *slots* devem ter conexões ao barramento. Esta restrição ocorre por duas razões: (*i*) a ferramenta de síntese suprime os *tri-states* não conectados ao barramento; (*ii*) não se deve deixar o sinal de *request* flutuante, pois transições espúrias podem comprometer a operação de todo o sistema. A política adotada por Palma foi construir módulos *dummy*, sem função lógica, para serem conectados aos *slots* sem módulos reconfiguráveis.

5.2.1.3 Criação de arquivos de restrição do usuário

Além de seguir os passos para restrição dos módulos do sistema apresentados na Seção 3.1.1, deve-se atentar para que os módulos que foram conectados em um determinado *slot* do barramento

fiquem posicionados sobre esta posição. Uma vez feita as restrições de posicionamento através da interface gráfica da ferramenta Floorplanner para um projeto *top*, é possível copiar as linhas do arquivo UCF que fazem referência aos módulos e aplicar a todos os outros projetos *top*. Isto é factível porque todos os módulos deverão ocupar uma área específica do sistema para que uma posterior reconfiguração possa ocorrer.

5.2.1.4 Roteamento manual

Esta é a fase que consome mais tempo, e exige um maior esforço do projetista. Neste passo deve-se garantir que a interconexão entre os módulos e o barramento seja idêntica em todos os projetos. Lembrar que havendo, por exemplo, dois módulos reconfiguráveis, são gerados dois projetos totais, denominados *projeto 1* e *projeto 2,* cada um com um dado módulo reconfigurável em todos os *slots.* Por exemplo, o fio *dataIn* do projeto 1 que interliga o primeiro *slot* do barramento ao módulo deve possuir as mesma conexões efetuadas pelo fio *dataIn* que interliga o primeiro *slot* do barramento ao módulo no projeto 2. A Figura 43 apresenta à direita a interface entre o primeiro slot e o barramento. Ao centro da Figura 43 o caminho percorrido pelo sinal *dataIn* é ressaltado, desde sua origem até o seu destino. À esquerda da Figura 43 é feito um zoom sobre o destino do sinal *dataIn*, permitindo visualizar diversos outros caminhos que poderiam ter sido tomados pelo sinal.

Caso exista diferença nas conexões da interface do projeto 1 em relação ao projeto 2 deve-se desconectar o fio em questão do projeto 2 e reproduzir exatamente as mesmas conexões feitas pelo respectivo fio no projeto 1.

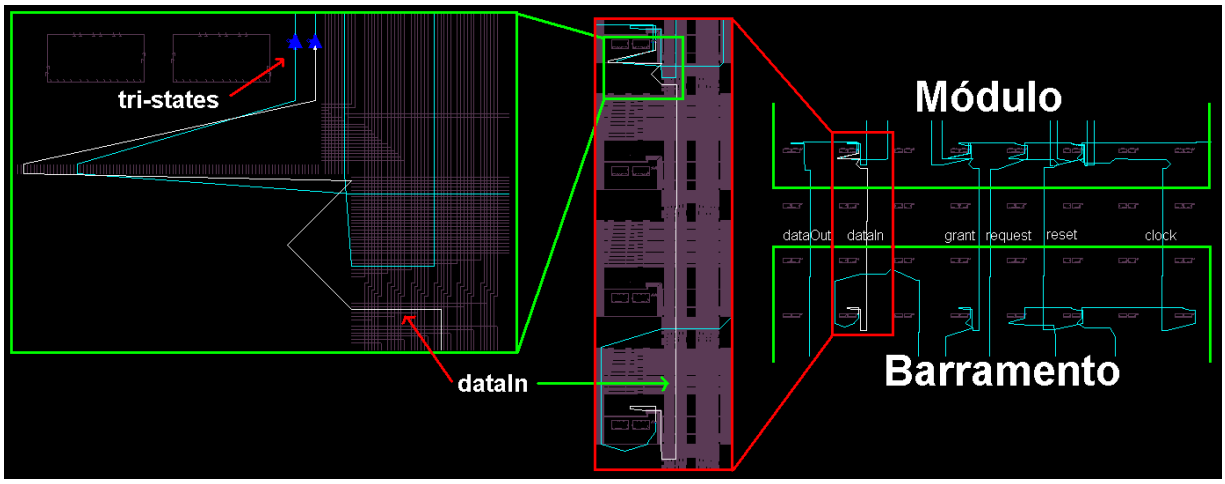


Figura 43 – À direita, interconexão de um módulo do SDR ao barramento; ao meio, ilustração do sinal *dataIn*; à esquerda, diversos outros caminhos que poderiam ser tomados pelo destino do sinal *dataIn*.

5.2.1.5 Geração de bitstreams totais

Uma vez garantido o idêntico roteamento das interfaces de todos os projetos é possível utilizar a ferramenta BitGen, apresentada na Seção 3.2, para gerar os bitstreams totais de todos os projetos pertencentes ao SDR. Nenhum parâmetro é necessário à ferramenta BitGen para a geração do arquivo total, apenas o nome do projeto com extensão NCD.

5.2.1.6 Geração de bitstreams parciais

Utilizando a ferramenta CoreUnifier, apresentada na Seção 3.4, é possível criar os bitstreams

parciais dos módulos que transformarão o sistema com uma nova configuração. O *bitstream* total que será utilizado para inicializar o SDR deverá ser o primeiro *bitstream* aberto no CoreUnifier para que o mesmo seja o *bitstream* base. A seguir devem ser abertos os outros *bitstreams* que possuem algum módulo diferente ao inicial, para que estes modifiquem a configuração do sistema.

A seleção do módulo de um *bitstream* secundário deve ser efetuada pela interface gráfica de forma a englobar toda a lógica do módulo, se estendendo até a camada de *tri-states* do mesmo. A ferramenta gerará um *bitstream* parcial que possui a parte inferior (barramento) originada do bitstream base e a parte superior (módulo) originada da seleção do bitstream secundário.

5.2.2 Estudo de Caso – Calculadora Reconfigurável

A aplicação utilizada para testar o fluxo de projeto proposto por Palma possui a mesma funcionalidade que a apresentada no fluxo de Projeto Modular: uma calculadora reconfigurável. O sistema é composto basicamente por três módulos, conforme apresentado pela Figura 44: controlador (mestre, árbitro e *tri-states* de interface com os módulos), módulo escravo1 e módulo escravo2.

O módulo *mestre* é responsável por prover entrada e saída de dados do sistema. As entradas do sistema são feitas a partir da reconfiguração de componentes LUTRAM internas ao módulo mestre, configuradas a partir da ferramenta BITProgrammer ou Circuit Customizer [MÖL03]. Estas ferramentas permitem fazer leitura e escrita de valores a partir de reconfiguração parcial. Também é responsabilidade do módulo mestre enviar a resposta da operação dos módulos para o display do FPGA.

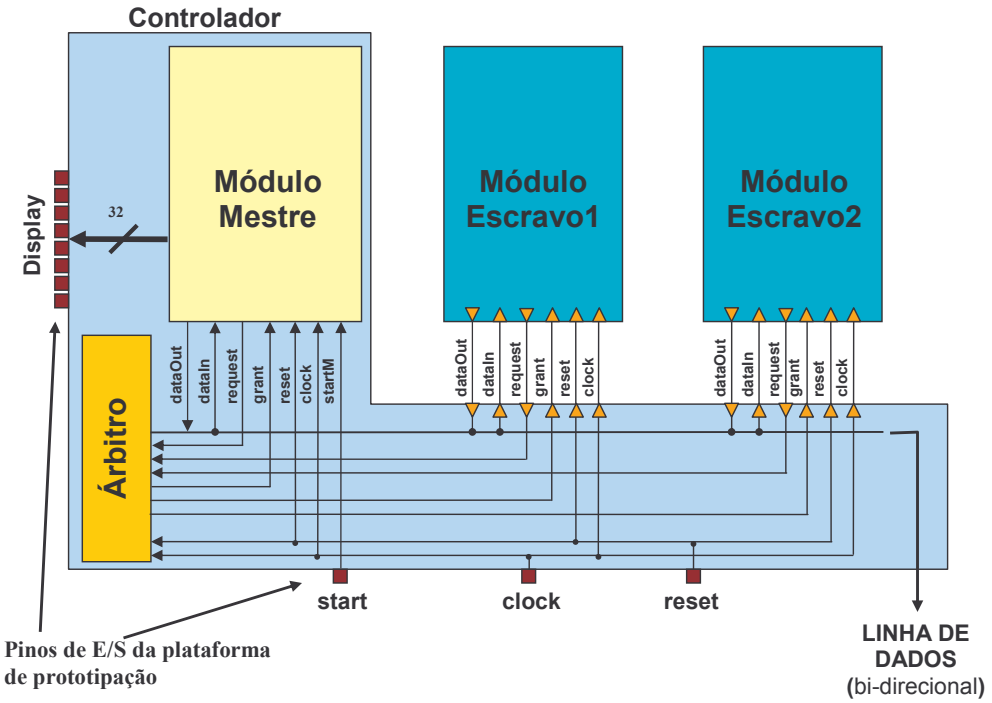


Figura 44 – Diagrama de blocos da calculadora reconfigurável para utilização com o fluxo de projeto de Palma.

É função do *árbitro* do sistema evitar que haja conflito de acessos simultâneos de escrita ao barramento, decidindo qual módulo pode colocar dados no meio compartilhado em um determinado instante de tempo. Neste sistema é utilizado um árbitro com prioridade rotativa, onde a cada ciclo

de relógio um módulo do sistema possui prioridade caso exista mais de um pedido de acesso simultâneo.

Os *tri-states de interface* entre os módulos reconfiguráveis e o barramento são os componentes isoladores no fluxo de projeto de Palma. A interface dos módulos com o barramento é feito pelos quatro sinais descritos em 5.2.1.1 e os sinais de controle *clock* e *reset*.

Os módulos *escravo1* e *escravo2* podem ser configurados como somadores e subtratores. Estes módulos recebem dois operandos de 32 bits do módulo mestre e, depois de computado, retornam o resultado novamente para o módulo mestre.

A Figura 45 apresenta um arquivo *top* do sistema da calculadora reconfigurável, instanciando um árbitro (linhas 33 e 34), um módulo mestre (linhas 37 a 52), dois escravos (linhas 54 a 69 e 71) e um display (linha 73). O barramento utilizado é de um único bit (sinal *dataLine*). Os módulos do sistema, tanto mestre quanto escravos, se conectam ao *bus_interface*, que por sua vez instanciam os módulos *send* e *receive* para a comunicação com as duas camadas de *tri-states*. A camada de *tri-states* do lado do módulo é instanciada dentro do *bus_interface*. A camada de *tri-states* do lado do barramento é instanciada no próprio *top* (*tri_interface*). Cada instância de *tri_interface* de cada módulo possui conexões de controle específicas com o árbitro e uma comunicação de dados compartilhada com todos os módulos (sinal *dataLine*).

Após uma síntese do projeto *top* apresentado na Figura 45 é possível utilizar o Floorplanner para restringir o posicionamento dos módulos e dos *tri-states* que conectam cada módulo ao barramento. Sugere-se utilizar para cada módulo o menor número de colunas para que a reconfiguração leve o menor tempo possível. Outra restrição que deve ser feita é quanto os *tri-states*. Cada módulo adicionado ao sistema demanda 12 *tri-states* (6 do lado do barramento e 6 do lado do módulo). Ao posicionar estes *tri-states*, também com a ferramenta Floorplanner, recomenda-se fixar os *tri-states* que ligam o mesmo fio em uma mesma coluna, de forma que o barramento seja conectado ao módulo pelo menor caminho possível, gerando um menor atraso. Os posicionamentos efetuados tanto dos módulos quanto de *tri-states* devem ser replicados para cada projeto pertencente ao sistema. No caso da calculadora reconfigurável existem dois projetos. Um com dois módulos somadores e outro com dois módulos subtratores, conforme pode ser observado na Figura 46.

```
01 entity coresBarramento is
02 port
03 (
04    clock   : in  std_logic;
05    reset   : in  std_logic;
06    startT  : in  std_logic;
07    RST_N_D : out std_logic;
08    CE_N_D  : out std_logic;
09    FL_N_D  : out std_logic;
10    WR_N_D  : out std_logic;
11    RD_N_D  : out std_logic;
12    CLK_SEL : out std_logic;
13    display : out std_logic_vector(x downto 0) -- especifico da plataforma
14 );
15 end coresBarramento;
16
17 architecture coresBarramento of coresBarramento is
18
19 signal R, C, GB, DI, DO, RqB, DV, enable: std_logic_vector(3 downto 0);
20 signal rst, CLK, GC, RqC, grant, request: std_logic_vector(3 downto 0);
21 signal dataLine: std_logic;
22 type word40 is array (3 downto 0) of std_logic_vector(39 downto 0);
23 signal WdIn : word40;
24 type word32 is array (3 downto 0) of std_logic_vector(31 downto 0);
25 signal WdOut: word32;
26 type word8  is array (3 downto 0) of std_logic_vector(7 downto 0);
27 signal Ad   : word8;
28
29 begin
30
31 dataLine <= '1' when enable="1111" else 'Z';
32
33 CTRL : arbitro port map (
34 clock=>clock, reset=>reset, request=>request, grant=>grant, tout=>enable );
35
36 -- modulo mestre
37 TRI0: tri_interface port map (
38 resetBus=>R(0), clockBus=>C(0), grantBus=>GB(0), dataInBus=>DI(0),
39 dataOutBus=>DO(0), requestBus=>RqB(0), resetCtl=>reset, clockCtl=>clock,
40 dataInCtl=>dataLine, dataOutCtl=>dataLine, grantCtl => grant(0),
41 requestCtl => request(0), enableWrite_n => enable(0) );
42
43 BI0: bus_interface port map (
44 resetBus=>R(0), clockBus=>C(0), grantBus=>GB(0), dataInBus=>DI(0),
45 dataOutBus=>DO(0), requestBus=>RqB(0), reset=>rst(0), clock =>clk(0),
46 grantC=>GC(0), reqC=>RqC(0), wordIn=>WdIn(0), dataValid=>DV(0),
47 wordOut=>WdOut(0), addressCore=>ad(0) );
48
49 Core0: master port map (
50 clk=>clk(0), reset=>rst(0), receive=>DV(0), grant=>GC(0),
51 word_out=>WdIn(0), send=>RqC(0), word_in=>WdOut(0),  address=>ad(0),
52 visor=>visor, start=>startT );
53
54 -- modulo somador 1
55 TRI1: tri_interface port map (
56 resetBus=>R(1), clockBus=>C(1), grantBus=>GB(1), dataInBus=>DI(1),
57 dataOutBus=>DO(1), requestBus=>RqB(1), resetCtl=>reset, clockCtl=>clock,
58 dataInCtl=>dataLine, dataOutCtl=>dataLine, grantCtl=>grant(1),
59 requestCtl => request(1), enableWrite_n => enable(1) );
60
61 BI1: bus_interface port map (
62 resetBus=>R(1), clockBus=>C(1), grantBus=>GB(1), dataInBus=>DI(1),
63 dataOutBus=>DO(1), requestBus=>RqB(1), reset=>rst(1), clock =>clk(1),
64 grantC=>GC(1), reqC=>RqC(1), wordIn=>WdIn(1), dataValid=>DV(1),
65 wordOut=>WdOut(1), addressCore=>ad(1) );
66
67 CoreSomador: slave1 port map (
68 clock=>clk(1), reset=>rst(1), receive=>DV(1), send=>RqC(1),
69 word_out=> WdIn(1), word_in=>WdOut(1), address=>ad(1), grant=>GC(1) );
70
71 -- modulo somador 2: idêntico ao bloco anterior, só que com índices 2
72
73 -- modulo display: port map especifico da plataforma de prototipação
74
75 end coresBarramento;
```

Figura 45 – Arquivo *top* da calculadora reconfigurável com dois módulos somadores.

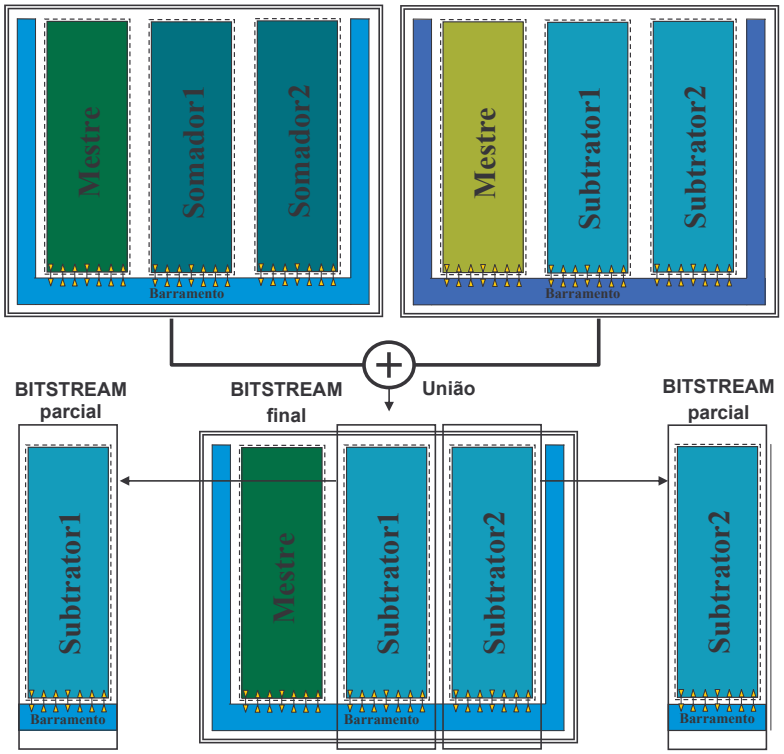


Figura 46 – Ilustração da geração de bitstreams efetuada pelo CoreUnifier. Dois bitstreams totais geram um novo total modificando apenas os módulos a serem substituídos. Posterior geração dos bitstreams parciais destes módulos.

De posse dos bitstreams totais com dois somadores e com dois subtratores, manualmente roteado, verificou-se o correto funcionamento destes em FPGA. Observado o correto funcionamento, utilizou-se a ferramenta CoreUnifier, apresentada na Seção 3.4, para a construção dos bitstreams parciais. Para transformar um sistema inicialmente configurado com dois módulos somadores para módulos subtratores, deve-se abrir primeiramente como bitstream base o bitstream do projeto somador e posteriormente o projeto do subtrator. Seleciona-se a área correspondente ao módulo subtrator1 e gera-se o bitstream parcial do mesmo. A seguir utiliza-se o mesmo processo para gerar o módulo subtrator2.

O teste do sistema no FPGA é feito configurando inicialmente o bitstream total que contém os módulos somadores e a seguir configurando o FPGA com os bitstreams parciais gerados. Sem interromper a execução do sistema os módulos subtratores entram em funcionamento.

5.3 Análise dos Fluxos estudados

Esta Seção apresenta críticas e comentários sobre o fluxo de Projeto Modular, abordado na Seção 5.1, e o fluxo de projeto de Palma, apresentado na Seção 5.2.

Um dos itens mais importantes para a construção de SDRs é, sem dúvida, a necessidade de roteamento manual. Tal processo dificulta o desenvolvimento do sistema por ser suscetível a erros e limitar a complexidade do sistema. O fluxo proposto por Palma envolve roteamento manual, enquanto o fluxo de Projeto Modular necessita que o roteamento seja apenas verificado.

O fluxo de projeto de Palma é mais controlável quando comparado ao fluxo de Projeto Modular. Durante a execução do fluxo de Projeto Modular inúmeros erros de difícil compreensão e correção são reportados pelas ferramentas que fazem parte do fluxo. O leitor é direcionado à Seção 4.3 e

anexo B de [BRI04] para alguns dos problemas obtidos com o fluxo de Projeto Modular. No fluxo de Palma o roteamento é o único problema, pois cada projeto do sistema é gerado pelo fluxo padrão e funciona individualmente. Uma alternativa para automatizar o fluxo de Palma é através da utilização de macros pré-roteadas e pré-validadas, inexistentes na época que o trabalho de Palma foi implementado.

A automatização das fases do fluxo de projeto de Palma é baixa em relação ao Projeto Modular. No Projeto Modular uma vez efetuados todos os passos de preparação e o script para execução do fluxo, alterações podem ser feitas no código fonte que os bitstreams do SDR são automaticamente gerados ao final da execução do fluxo. No fluxo de projeto de Palma toda a vez que um código fonte for modificado o roteamento entre os módulos e o barramento deve ser refeito.

Conforme apontado no anexo B de [BRI04] algumas vezes é necessário adicionar mais componentes ao SDR para que a reconfiguração do sistema seja bem sucedida. Este é normalmente o caso presenciado quando o sistema não consegue atingir restrições de tempo, necessitando de módulos DCM para evitar o escorregamento de relógio e, por conseguinte, LUTs. Como nem sempre estes componentes estão presentes na mesma área reconfigurável que o componente ao qual eles devem ser conectados pode ser necessário adicionar mais bus macros para a comunicação dos mesmos.

Ambos os fluxos de Projeto Modular e Palma utilizam componentes isoladores baseados em *tri-states*. O número de componentes *tri-states* em dispositivos Virtex e Virtex-II é baixo, podendo tornar a comunicação das diferentes áreas um fator limitante do SDR. Um trabalho que não utiliza componentes isoladores baseados em *tri-states* foi desenvolvido por Huebner [HUB04b] e abordado na Seção 2.3.

6 CONCLUSÕES E TRABALHOS FUTUROS

Os Capítulos 2, 3 e 4 deste trabalho abordaram genericamente etapas do fluxo de projeto de SDRs, enquanto o Capítulo 5 apresentou dois fluxos completos para a construção de SDRs. O trabalho utilizou dispositivos Virtex, por estes possuírem grande densidade de portas lógicas, permitirem reconfiguração parcial e dinâmica e terem uma grande difusão no mercado. Embora os dispositivos Virtex permitam reconfiguração parcial e dinâmica, sua arquitetura para reconfiguração é unidimensional, diferentemente das famílias XC6200 [XIL97] (Xilinx) e Clay [NAT93] (National), ambas descontinuadas. Ambas as famílias de dispositivos permitiam reconfiguração parcial no nível de blocos lógicos, possuindo uma arquitetura bidimensional para reconfiguração. Outra família de FPGAs reconfiguráveis disponível no mercado é a ATMEL AT40k, porém sua baixa densidade em termos de portas lógicas não a habilita para a implementação de SDRs.

O presente trabalho identificou três fatores que dificultam o avanço da tecnologia de reconfiguração parcial e dinâmica. O primeiro deles reside em ferramentas de CAD específicas para a construção de SDRs. Das etapas de projeto apresentadas no decorrer deste trabalho, destaca-se a etapa de restrição de roteamento como sendo a etapa mais problemática do desenvolvimento. O segundo item é a falta de uma arquitetura de sistema que dê suporte a SDRs. Não existe uma interface de comunicação bem definida que permita que um módulo reconfigurável acesse outras áreas reconfiguráveis e pinos de entrada e saída. Ambas as comunicações, quando desejadas, devem ser implementadas pelo projetista. O terceiro item que dificulta a utilização de reconfiguração parcial e dinâmica é o tempo elevado de reconfiguração de dispositivos de granularidade baixa, como é o caso de FPGAs Xilinx. Atualmente o tempo de reconfiguração médio de um módulo de hardware em dispositivos de grão fino fica em torno de centenas de microssegundos, enquanto o relógio do sistema oscila na escala de nano segundos. Este elevado tempo de reconfiguração conduz à necessidade de particionar o sistema em desenvolvimento de tal forma que o tempo de execução do módulo reconfigurável amortize o tempo de reconfiguração. O projetista possui duas opções: (*i*) módulo reconfigurável que execute tarefas com elevado tempo de processamento; ou (*ii*) módulo reconfigurável que é re-utilizado muitas vezes, como apresentado em [MÖL04].

Frente a estes fatores limitantes, trabalhos futuros buscarão contribuir nos dois primeiros itens, CAD e arquitetura. O terceiro item, tempo de reconfiguração, não é foco do trabalho por ser dependente de tecnologia. O primeiro item será abordado com a proposta de um novo fluxo de construção de SDRs que possua a automatização do Projeto Modular e utilize o fluxo básico de projetos VLSI para FPGAS conforme o fluxo de Palma o faz. Uma rede intra-chip (NOCs – Network On Chip) será desenvolvida para proporcionar um meio de comunicação escalável entre módulos reconfiguráveis.

7 REFERÊNCIAS BIBLIOGRÁFICAS

[BLO03] Blodget, B.; James-Roxby, P.; Keller, E.; McMillan, S.; Sundararajan, P. "**A Self-reconfiguring Platform**". In: Field Programmable Logic and Applications (FPL), 2003, pp. 565-574.

[BLO04] Blodget, B.; Bobda, C.; Huebner, M.; Niyonkuru, A. "**Partial and Dynamically Reconfiguration of Xilinx Virtex-II FPGAs**". In: Field Programmable Logic and Applications (FPL), 2004, pp. 801-810.

[BRI04] Brião, E. "**Reconfiguração Parcial e Dinâmica para Núcleos de Propriedade Intelectual**". Dissertação de mestrado, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brasil, 2004, 123 p.

[COM00] Compton, K.; Cooley, J.; Knol, S.; S. Hauck. "**Configuration Relocation and Defragmentation for FPGAs**". Relatório Técnico, Universidade Northwestern, (Departamento de Elétrica e Engenharia da Computação), Madison, Estados Unidos, 2000, 15 p.

[DYE02a] Dyer, M.; Plessl, C.; Platzner, M. "**Partially Reconfigurable Cores for Xilinx Virtex**". In: Field Programmable Logic and Applications (FPL), 2002, pp. 292-301.

[DYE02b] Dyer, M.; Wirz, M. "**Reconfigurable System on FPGA**". Dissertação de mestrado, Instituto Técnico Federal de Zurique, Zurique, Suiça, 2002, 82 p.

[HOR04] Horta, E.; Lockwood, J. "**Automated Method to Generate Bitstream Intellectual Property Cores for Virtex FPGAs**". In: Field Programmable Logic and Applications (FPL), 2004, pp. 975-979.

[HUB04a] Huebner, M.; Ullmann, M.; Braun, L.; Klausmann, A.; Becker, J. "**Scalable Application-Dependent Network on Chip Adaptivity for Dynamical Reconfigurable Real-Time Systems**". In: Field Programmable Logic and Applications (FPL), 2004, pp. 1037-1041.

[HUB04b] Huebner, M.; Becker, T.; Becker, J. "**Real-Time LUT-Based Network Topologies for Dynamic and Partial FPGA Self-Reconfiguration**". In: Brazilian Symposium on Integrated Circuits and Systems Design (SBCCI), 2004, pp. 28-32.

[LIM04] Lim, D.; Peattie, M. "**Two Flows For Partial Reconfiguration: Module Based or Small Bit Manipulations**". Xilinx Application Note 290 (XAPP290 v1.2), 2004, 28 p.

[MER98] Merino, P.; Lopez, J.; Jacome, M.; "**A Hardware Operating System for Dynamic Reconfigurations of FPGAs**". In: Field Programmable Logic and Applications (FPL), 1998, pp. 431-435.

[MES03] Mesquita, D.; Moraes, F.; Palma, J.; Möller, L.; Calazans, N. "**Remote and Partial Reconfiguration of FPGAs: tools and trends**". In: Parallel and Distributed Processing Symposium (IDPDS), 2003, pp. 177.

[MÖL03] Möller, L. "**Ferramentas de Reconfiguração Parcial, Remota e Dinâmica de FPGAs Virtex**". Relatório Técnico, Pontifícia Universidade Católica do Rio Grande do Sul (Programa de Pós-Graduação em Ciência da Computação), Porto Alegre, Brasil, 2003, 29 p.

[MÖL04] Möller, L.; Calazans, N.; Moraes, F.; Brião, E.; Carvalho, E.; Camozzato, D. "**FiPRe: An Implementation Model to Enable Self-Reconfigurable Applications**". In: Field Programmable Logic and Applications (FPL), 2004, pp. 1042-1046.

[MOR03a] Moraes F.; Calazans, N. "**R8 Processor Architecture and Organization Specification and Design Guidelines**". Available at: http://www.inf.pucrs.br/~gaph/Projects/R8/public/R8_arq_spec_eng.pdf, 2003.

[MOR03b] Moraes, F.; Mesquita, L.; Palma, J.; Möller, L.; Calazans, N. "**Development of a Tool-Set for Remote and Partial Reconfiguration of FPGAs**". In: Design, Automation and Test in Europe Conference and Exhibition (DATE), 2003, pp. 1122-1123.

[NAT93] National Semiconductor. "**Configurable Logic Array (CLAy)**". Data Sheet, 1993.

[NOL03] Nollet, V.; Coene, P.; Verkest, D.; Vernalde, S.; Lauwereins, R. "**Designing an Operating System for a Heterogeneous Reconfigurable SoC**". In: Parallel and Distributed Processing Symposium (IDPDS), 2003, 7 p.

[PAL02a] Palma, J.; Mello, A.; Möller, L.; Moraes, F.; Calazans, N. "**Core Communication Interface for FPGAs**". In: Brazilian Symposium on Integrated Circuits and Systems Design (SBCCI), 2002, pp. 183-188.

[PAL02b] Palma, J. "**Métodos de desenvolvimento e distribuição de IP-cores**". Dissertação de mestrado, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brasil, 2002, 94 p.

[VCC04] Virtual Computer Corporation. "**The Virtual Workbench"**. Available at: http://www.vcc.com.

[WAL03] Walder, H.; Platzner, M. "**Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations**". In: Eng. of Reconfigurable Systems and Algorithms (ERSA), 2003, pp. 284-287.

[WAL04] Walder, H.; Platzner, M. "**A Runtime Environment for Reconfigurable Hardware Operating Systems**". In: Field Programable Logic and Applications (FPL), 2004, pp. 831-835.

[XIL97] Xilinx, Inc. "**XC6200 Field Programmable Gate Arrays**". Xilinx, 1997, 73 p.

[XIL04a] Xilinx, Inc. **http://www.xilinx.com**. 2004.

[XIL04b] Xilinx, Inc. "**Virtex-II Platform Platform FPGA Handbook**". Xilinx User Guide 002 (UG002 v1.9), 2004, 490 p.

[XIL04c] Xilinx, Inc. "**Development System Reference Guide – Modular Design**". Xilinx, 2004, 484 p.

[XIL04d] Xilinx, Inc. "**JBits 3.0 SDK documentation**". Available at: http://www.xilinx.com/labs/projects/jbits, 2004.

[XIL04e] Xilinx, Inc. "**Virtex Series Configuration Architecture – User Guide**". Xilinx Application Note 151 (XAPP151 v1.7), 2004, 45 p.