PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SISTEMAS DINAMICAMENTE RECONFIGURÁVEIS COM COMUNICAÇÃO VIA REDES INTRA-CHIP

por

LEANDRO HELENO MÖLLER

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

Prof. Dr. Fernando Gehm Moraes
Orientador

Porto Alegre, Dezembro de 2005.

Resumo

O projeto de SoCs atuais compreende dezenas de núcleos IP (*Intellectual Property*) compreendendo módulos de hardware dedicado e processadores. Núcleos IP obtêm alto desempenho com a utilização de hardware dedicado. Flexibilidade é a característica chave de processadores, visto que é fácil modificar o comportamento dos programas em tempo de execução. A maioria dos SoCs atuais não possui capacidade de modificar o comportamento do hardware ou a sua estrutura depois da fabricação do sistema. Além da inflexibilidade observada no hardware, a integração de dezenas ou centenas de núcleos em um mesmo SoC traz problemas relacionados a escalabilidade e desempenho. Para solucionar estes problemas, redes intra-chip (*Networks on Chip* - NoCs) estão se tornando cada vez mais difundidas. Este trabalho propõe estender a flexibilidade natural do software para módulos de hardware, utilizando NoCs como principal recurso interno de comunicação. Isto é possível a partir da reconfiguração de módulos de hardware em tempo de execução. O presente trabalho possui duas contribuições principais: primeiro, um fluxo de projeto específico para SoCs com módulos reconfiguráveis; segundo, a proposta de um conjunto de características necessárias para habilitar a reconfiguração de módulos de hardware de uma NoC.

Palavras Chave: SoCs, NoCs, CoCs, MPSoCs, FPGAs, SDRs, reconfiguração parcial, reconfiguração dinâmica.

Abstract

A modern SoC design comprises dozens of dedicated IP cores, with dedicated hardware modules and processors. High performance can be obtained by implementing a dedicated hardware. Flexibility is the key feature of processors, since it is easy to modify their applications behavior at runtime. Most current SoCs have no capability to modify the hardware behavior or structure after system fabrication. Besides the observed inflexibility, the integration of dozens or hundreds of IP cores in a single chip introduces problems related to scalability and performance. To deal with these problems, suggestions to employ Networks on Chip (NoCs) are becoming widespread. This work proposes to extend the inherent software flexibility to dedicated IP cores in SoCs, using NoCs as the main internal communication resource. This can be achieved by making IP cores reconfigurable. The work advances two main contributions: first, a straightforward design flow for SoCs with reconfigurable IP cores; second, the proposition of a feature set required for a NoC to enable its usage in the support of IP core reconfiguration.

Keywords: SoCs, NoCs, CoCs, MPSoCs, FPGAs, DRSs, partial reconfiguration, dynamic reconfiguration.

AGRADECIMENTOS

Muitas pessoas contribuíram para a efetivação deste trabalho, e, mesmo correndo o risco de esquecer alguém, não posso deixar de lhes expressar aqui os meus agradecimentos.

Ao meu orientador, Fernando Moraes, pela sua objetividade e constante orientação. Desculpa ter incomodado demais, eu sei que abusei diversas vezes da sua boa vontade! Obrigado por estar presente e preocupado até mesmo quando estavas longe (viajando). Obrigado pela liberdade de me deixar fazer as coisas do jeito que eu queria e depois ainda me apoiar mesmo não tendo sido o caminho que você havia escolhido. Obrigado pelos mais de 5 anos de convivência e aprendizagem, não só técnica, como também de vida.

Ao Ney Calazans, meu informal co-orientador, pelo seu vasto conhecimento e pelos toques finais que agregam valor ao nosso trabalho.

À Aline Mello, que já me acompanha e atura há 6 anos na graduação, no mestrado e no GAPH todos os dias. A sua força e inteligência são ilimitadas, porque até quando não gosta de alguma coisa a faz muito bem feita. Foram estas e outras qualidades, adicionadas a tua constante companhia, que me motivaram a sempre seguir em frente.

Aos bolsistas que me auxiliaram com o presente trabalho, Ismael Grehs, Daniel Camozzato, Alzemiro Henrique e Aline Abreu, os meus sinceros agradecimentos. Não se esqueçam: “passos de bebê”, ou seja, não implementem tudo de uma vez, senão não se consegue encontrar o erro!

A todos os colegas do GAPH, não só áqueles que comparecem atualmente ao laboratório, mas aos que estão mundo afora: tem sido uma honra trabalhar ao lado de tantas mentes brilhantes. Um abraço especial para os autores dos trabalhos os quais este deu procedimento: Daniel Mesquita, José Carlos Palma, Eduardo Brião e Ewerson Carvalho. Bah, teria tantos colegas e amigos para citar aqui que daria estouro de memória no computador!

À minha família, que sempre me proveu uma base sólida de amor e princípios, elementos fundamentais para chegar até aqui. Ao meu pai, Luiz Heleno, pela razão. À minha mãe, Adriana, pelo amor. Ao meu irmão, Luciano, pelo espírito de aproveitar ao máximo a vida! Vocês foram essenciais para a minha criação e para mais esta etapa da vida!

Eu gostaria também de destacar 3 acontecimentos chaves na linha do tempo que me fizeram chegar até esta dissertação: o primeiro aos 6 anos, quando meu pai comprou o primeiro computador Maxxi compatível com a família Apple-II; o segundo aos 17 anos, quando não passei no vestibular da UFRGS, me fazendo vir para a PUCRS; o terceiro aos 19, quando fui convidado pela Aline Mello a fazer parte do GAPH. Cada um destes acontecimentos, que parecem rotineiros, foi de fundamental importância para concretizar o presente trabalho.

A todos o meu muito obrigado.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API	*Application Programming Interface*
ASIC	*Application Specific Integrated Circuit*
ASIP	*Application Specific Instruction-Set Processor*
BRAM	*Block RAM*
CAD	*Computer Aided Design*
CC	Controlador de Configurações
CLB	*Configurable Logic Block*
COC	*Cluster on Chip*
CRC	*Cyclic Redundancy Check*
DCM	*Digital Clock Manager*
DCT	*Discrete Cosine Transform*
DMA	*Direct Memory Access*
DPU	*Datapath Unit*
DRAM	*Dynamic RAM*
EDIF	*Electronic Digital Interchange Format*
E/S	Entrada e Saída
FPGA	*Field Programmable Gate Array*
GPP	*General Purpose Processor*
GPS	*Global Positioning System*
HDL	*Hardware Description Language*
ICAP	*Internal Configuration Access Port*
ILP	*Instruction Level Parallelism*
IOB	*Input/Output Block*
ISE	*Integrated Software Environment*
IP	*Intellectual Property*
LUT	*Look-Up Table*
MPSoC	*Multiprocessed SoC*
NCD	*Native Circuit Description*
NDA	*Non Disclosure Agreement*
NEWS	*North, East, West, South*
NGD	*Native Generic Database*
NGO	*Native Generic Object*
NMC	*Native Macro Circuit*
NoC	*Network on Chip*
PAL	*Programmable Array Logic*
PDA	*Personal Digital Assistant*
PE	*Processing Element*
PLA	*Programmable Logic Array*
PLD	*Programmable Logic Device*
PROM	*Programmable ROM*
RAM	*Random Access Memory*
RAP	*Reconfigurable Arithmetic Processing Unit*
RDPA	*Reconfigurable Datapath Architecture*
RDPU	*Reconfigurable Datapath Unit*
RFU	*Reconfigurable Functional Unit*
RISC	*Reduced Instruction-Set Computer*
RISP	*Reconfigurable Instruction-Set Processor*
ROM	*Read-Only Memory*

RPU	*Reconfigurable Processing Unit*
RTR	*Run-Time Reconfiguration*
SDF	*Standard Delay Format*
SDR	Sistema Dinamicamente Reconfigurável
SoC	*System on Chip*
SRAM	*Static RAM*
UCF	*User Constraints File*
UDP	*User Datagram Protocol*
ULA	Unidade Lógica e Aritmética
VHDL	*VHSIC HDL*
VHSIC	*Very High Speed Integrated Circuit*
XDL	*Xilinx Design Language*

1 INTRODUÇÃO

A crescente demanda por equipamentos eletrônicos cada vez menores, mais duradouros, mais rápidos e com mais funcionalidades no mesmo dispositivo, motivam a pesquisa por novos métodos de implementação. A busca por desempenho em microprocessadores seguiu duas estratégias simultâneas ao longo do tempo: aumento da freqüência de relógio e paralelismo no nível de instrução (ILP – *Instruction Level Parallelism*). O aumento da freqüência está alcançando um limite, devido principalmente a problemas de dissipação de potência e complexidade de projeto. O ILP seguiu o caminho *pipeline superescalar* com múltiplas linhas execução (*multi-thread*) e também está chegando a um limite nos sistemas mono-processados. Companhias como Intel, AMD, Sony, IBM e Toshiba mostraram no ano de 2005 a tendência para futuros microprocessadores: **SoCs multiprocessados** (MPSoCs - *Multiprocessed System on Chip*), ou seja, múltiplos núcleos de processamento em um único chip. MPSoCs aumentam o desempenho, pois diversos processadores cooperam para executar as tarefas e reduzem a latência de comunicação entre os processadores, por estes estarem implementados em um único chip.

No entanto, não somente o fator distância influencia na comunicação entre os elementos de processamento de um SoC, mas também o meio de comunicação em si. Um dos meios de comunicação mais utilizados, devido à simplicidade de implementação e ao baixo consumo de área, são os barramentos. Barramentos, segundo [BEN02], possuem diversas limitações: (*i*) o número de módulos de hardware que podem ser conectados ao barramento é tipicamente na ordem da dezena; (*ii*) apenas uma troca de dados pode ser realizada por vez, pois o meio físico é compartilhado por todos os módulos de hardware, reduzindo o desempenho global do sistema; (*iii*) necessidade de mecanismos inteligentes de arbitragem do meio físico, para evitar desperdício de largura de banda; (*iv*) o uso de linhas globais em um circuito integrado com tecnologia submicrônica impõe sérias restrições ao desempenho do sistema, devido às altas capacitâncias e resistências parasitas inerentes aos fios longos. Estas limitações podem ser parcialmente contornadas através do uso de, por exemplo, hierarquias de barramentos, onde o problema continua existindo, sendo apenas reduzido.

Uma maneira de minimizar os problemas oriundos da arquitetura de barramentos é através da utilização de redes de comunicação internas ao circuito integrado [DAL01] [WIN01], no que se denomina hoje de **NoC** (*Network on Chip*). Uma NoC é uma rede intra-chip [BEN02] composta por módulos de hardware conectados a roteadores, que por sua vez são conectados entre si através de canais de comunicação. Este meio de comunicação possibilita que diferentes módulos da NoC se comuniquem simultaneamente, sendo este um meio de comunicação promissor para futuros MPSoCs [YE03] e CoCs (*Cluster on Chip*) [OTT00].

Uma característica que pode ser agregada a uma NoC, quando a mesma for implementada em **dispositivos parcial e dinamicamente reconfiguráveis** (e.g. algumas famílias de FPGAs), é a possibilidade de substituir módulos conectados à NoC enquanto outros continuam operando. Esta substituição de módulos em tempo de execução pode trazer novos benefícios aos SoCs, como por

exemplo reduzir o consumo de potência e o tamanho do sistema, aumentar o tempo de vida do produto, implementar SoCs tolerantes a falhas e possuir um hardware virtualmente maior que o fisicamente existente, já que os módulos são configurados no sistema conforme a demanda.

Seguindo as tendências apresentadas inicialmente, presume-se que futuros SoCs possuirão dezenas de módulos de hardware interconectados por uma NoC, sendo cada módulo reconfigurado com um hardware (e.g. um processador ou um hardware dedicado) que melhor atenda as aplicações que estão sendo executadas no momento pelo SoC. Este trabalho versa exatamente sobre a junção destas tendências (NoC e reconfiguração), objetivando auxiliar o projeto destes SoCs.

1.1 Cenário Atual de Projeto em Sistemas Embarcados

Sistemas Embarcados atuais (e.g. celulares, computadores de mão, máquinas fotográficas digitais, tocadores de música) utilizam basicamente dois tipos de tecnologia: circuitos integrados de propósito específico (ASICs) e processadores (GPPs, DSPs e ASIPs). A Figura 1 ilustra diferentes tecnologias disponíveis para a implementação de sistemas embarcados em um gráfico, quantificando consumo de potência, flexibilidade e desempenho destas.

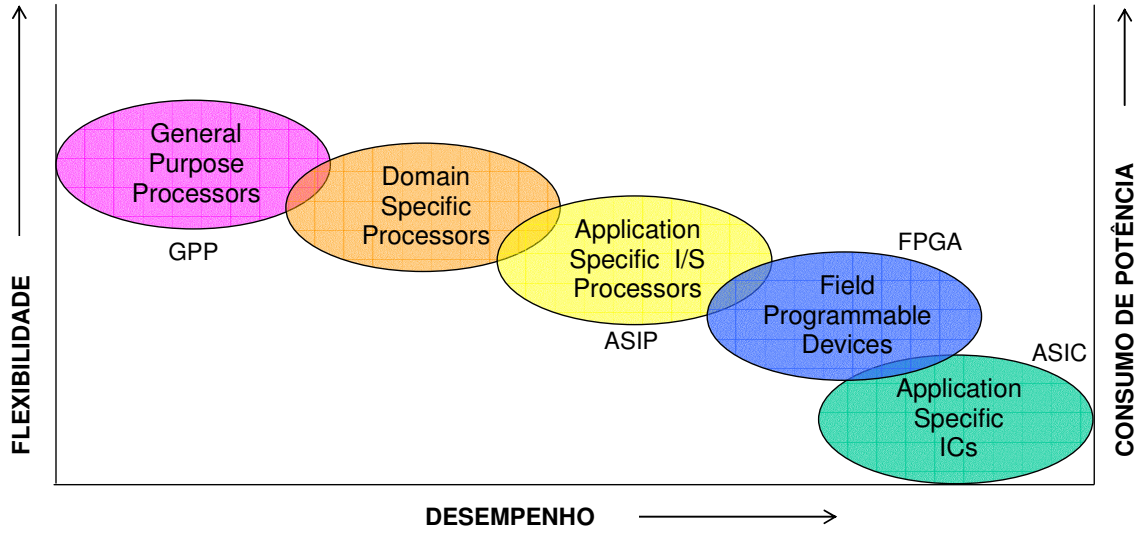


Figura 1 – Tecnologias atuais para implementação de sistemas embarcados.

Conforme mostra o gráfico, processadores são utilizados quando flexibilidade é um requisito importante do projeto, e ASICs são empregados quando desempenho e/ou baixo consumo de potência é almejado. No entanto, quando ambas as características são desejadas, processadores e ASICs têm sido utilizados em um mesmo sistema embarcado. As conseqüências deste fato são: (*i*) um grande consumo de área para a implementação do circuito integrado e do(s) processador(es); (*ii*) elevado consumo de potência devido ao alto consumo de área e à implementação de interfaces entre as diferentes tecnologias empregadas.

Segundo o gráfico da Figura 1, a tecnologia de dispositivos reconfiguráveis é uma alternativa intermediária entre processadores e ASICs para sistemas embarcados que necessitam

simultaneamente de flexibilidade e desempenho. Por que esta tecnologia ainda não é largamente empregada para a implementação de sistemas embarcados com as características citadas?

Dispositivos reconfiguráveis, surgidos na década de 80, são relativamente novos em relação a tecnologias já maduras como ASICs e processadores, lançados nas décadas de 60 e 70, respectivamente. Desta forma, estas tecnologias se encontram mais otimizadas para a implementação de sistemas embarcados atuais. Outro ponto a ser considerado é que os recursos configuráveis de dispositivos reconfiguráveis também consomem área e potência do sistema embarcado como um todo.

Dispositivos reconfiguráveis são utilizados tanto na prototipação de sistemas digitais, quanto em produtos finais. A vantagem da utilização destes em produtos finais é a capacidade de poder modificar o hardware, permitindo: (*i*) atualizar não somente o software, como também o hardware; (*ii*) aumentar o tempo de vida do produto, instalando novas funcionalidades de hardware; (*iii*) aumentar o desempenho do produto, migrando tarefas para serem executadas em hardware; (*iv*) reduzir o tamanho do dispositivo, configurando apenas o hardware necessário para uso imediato; (*v*) reduzir o consumo de potência, como conseqüência da redução da área; (*vi*) implementar sistemas tolerantes a falhas, reconfigurando o hardware para corrigir defeitos do sistema; (*vii*) reduzir o 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 reconfigurável é comprado pronto, necessitando apenas a configuração do dispositivo com a lógica específica do produto. Algumas desvantagens de dispositivos reconfiguráveis são: (*i*) custo elevado quando comparado a ASICs produzidos em grande quantidade; (*ii*) maior consumo de área e potência quando comparando uma única configuração utilizada no dispositivo reconfigurável com um circuito integrado equivalente implementado em ASIC.

Dispositivos reconfiguráveis permitem que um mesmo produto seja tanto específico quanto genérico. Considere um produto genérico como sendo um computador de mão e um produto específico como sendo 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 reconfiguráveis, 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 reconfiguráveis utilizados na indústria são EPROMs, PLAs, PALs, PLDs e FPGAs. Entre estes, se destaca o FPGA, pela sua alta densidade de elementos lógicos e rápida reconfigurabilidade. Estes componentes são baseados em um arranjo bidimensional de células básicas de lógica configurável repetidas, possuindo um paralelismo nativo. Além deste arranjo básico, outras estruturas mais complexas como blocos de memória ou mesmo processadores são integrados ao componente, aumentando consideravelmente seu poder de computação.

1.2 Definições e Conceitos

Esta Seção apresenta definições e conceitos que permitem associar a linguagem do presente documento com a literatura da área de sistemas reconfiguráveis e redes intra-chip.

Dispositivo reconfigurável: Circuito integrado que habilita definir a interconexão de seus componentes internos, bem como a função destes componentes através do processo que se denomina configuração. Podem-se citar os dispositivos Virtex [XIL04d] e APEX [ALT04] como exemplos de dispositivos reconfiguráveis.

Arquitetura reconfigurável: Arquitetura que conecta um ou mais dispositivos reconfiguráveis a outros componentes de hardware, permitindo a interação com o mundo externo. Pode-se citar a XSV-800 [XES00] como exemplo de arquitetura reconfigurável com um dispositivo reconfigurável, e o PRISM [ATH93] como exemplo de arquitetura reconfigurável com diversos dispositivos reconfiguráveis.

Sistema reconfigurável: Projeto e implementação de hardware que tira proveito das características reconfiguráveis de arquiteturas reconfiguráveis. Podem-se citar os trabalhos [HUB04b] e [MEL05] como exemplos de sistemas reconfiguráveis.

Bitstream: Conjunto de bits que define a função dos recursos reconfiguráveis de um sistema reconfigurável, conferindo uma funcionalidade específica ao mesmo.

Configuração: Processo de escrever um bitstream na memória do sistema reconfigurável, de forma a definir a funcionalidade do mesmo.

Reconfiguração: Processo de sobrescrever um bitstream na memória do sistema reconfigurável, de forma a redefinir a funcionalidade do mesmo.

Reconfiguração total: É a reconfiguração onde a memória do sistema reconfigurável é inteiramente sobrescrita.

Reconfiguração parcial: É a configuração que permite que somente uma porção da memória do sistema reconfigurável seja sobrescrita.

Reconfiguração estática: É uma reconfiguração que afeta a operação do sistema inteiro, implicando uma parada geral durante o processo de reconfiguração. Também chamada de reconfiguração disruptiva.

Reconfiguração dinâmica: As porções do sistema que não estão sendo reconfiguradas permanecem completamente funcionais durante o ciclo de reconfiguração. Também chamada de *run-time reconfiguration* (RTR), *on-the-fly reconfiguration, in-circuit reconfiguration* ou reconfiguração não-disruptiva. Todas essas expressões podem ser traduzidas também como reconfiguração em tempo de execução.

Projeto reconfigurável: É um conjunto de descrições abstratas que pode, através de um processo automático ou semi-automático, gerar um bitstream. Este bitstream ou projeto reconfigurável é uma das possíveis configurações que compõe um sistema reconfigurável ao longo do tempo.

Bloco lógico: É a célula básica de dispositivos reconfiguráveis para a implementação de funções lógicas.

FPGA: É um tipo de dispositivo reconfigurável que consiste de uma matriz de blocos lógicos, chaves de interconexão, fios e pinos de E/S configuráveis. Os blocos lógicos são conectados entre si ou a pinos de E/S através de um ou mais fios e chaves de interconexão.

Chave de interconexão: Recurso de hardware que permite conectar dois fios distintos. Pode-se citar como exemplos de chaves de interconexão *tristates* e *switch boxes*.

Recursos internos do FPGA: Blocos lógicos, fios, chaves de interconexão e pinos de E/S.

Grão: Estrutura básica em um sistema reconfigurável, replicada para gerar o todo ou a parte fundamental do sistema.

Granularidade: Característica de um sistema reconfigurável, dependente da dimensão do grão deste.

Sistema reconfigurável de grão pequeno: Aquele onde o grão é relativamente simples. Por exemplo, no caso de FPGAs da família Virtex-II da Xilinx, o grão é um CLB.

Sistema reconfigurável de grão grande: Os sistemas reconfiguráveis de grão grande possuem como grão unidades lógicas e aritméticas (ULAs) e/ou pequenos microprocessadores e memórias. No caso do Chess [MAR99], o grão é uma ULA.

Núcleo: É um sistema digital complexo pré-projetado, pré-verificado e prototipado em hardware pelo menos uma vez. Os núcleos são utilizados como componentes de aplicações, sendo a reusabilidade destes em diferentes aplicações uma característica importante para o projeto de sistemas digitais. Os núcleos são chamados também usando as terminologias em inglês, *core* ou *IP core*.

Invólucro: O invólucro é uma camada de lógica inserida sobre o núcleo, permitindo adequar a interface e o protocolo de comunicação original do núcleo ao meio de comunicação. Também chamado de *wrapper*.

Módulo: É o núcleo pronto para ser conectado a um meio de comunicação, ou seja, o núcleo com o seu *wrapper*.

Roteador: É o núcleo de hardware básico de uma rede intra-chip, responsável por fazer o repasse de pacotes.

CPU bound: característica de um sistema onde o tempo de computação é principalmente determinado pelo tempo despendido com processamento.

I/O bound: característica de um sistema onde o tempo de computação é principalmente determinado pelo tempo despendido com operações de entrada e saída (E/S).

Top: arquivo de mais alta hierarquia em um projeto de hardware.

1.3 Objetivos

O objetivo estratégico do presente trabalho é dominar a tecnologia de sistemas digitais reconfiguráveis, utilizando redes intra-chip como meio de comunicação entre módulos de propriedade intelectual reconfiguráveis.

Os objetivos específicos deste trabalho são: (*i*) propor um fluxo de projeto que simplifique o processo de implementação de SDRs (Sistemas Dinamicamente Reconfiguráveis); (*ii*) implementar uma plataforma multiprocessada dinamicamente reconfigurável construída sobre uma rede de interconexão de baixo consumo de área.

A rede de interconexão empregada no presente trabalho está limitada a uma rede intra-chip direta, com topologia malha, chaveamento por pacotes e repasse de pacotes pelo método wormhole. O motivo destas restrições de projeto ocorrem devido à disponibilidade do código fonte da NoC Hermes, que possui tais características, e desta NoC poder ser implementada em FPGAs junto a outros módulos de hardware conectados a ela no mesmo dispositivo.

1.4 Organização do Documento

Este trabalho está organizado como segue. O Capítulo 2 apresenta a evolução de sistemas e dispositivos reconfiguráveis, contemplando também processadores reconfiguráveis e sistemas reconfiguráveis. Neste Capítulo é dada ênfase a dispositivos parcial e dinamicamente reconfiguráveis, os quais possuirão esta característica explorada no restante do documento.

O Capítulo 3 apresenta fluxos de projeto para a reconfiguração parcial e dinâmica de SoCs. Tal estudo faz-se necessário uma vez que os fabricantes de dispositivos reconfiguráveis nem sempre fornecem um conjunto completo de ferramentas de CAD que possibilite a reconfiguração parcial e dinâmica de seus dispositivos.

O Capítulo 4 apresenta um novo fluxo de projeto para a implementação de sistemas parcial e dinamicamente reconfiguráveis, sendo esta a primeira contribuição deste trabalho. Este fluxo é baseado em fluxos estudados no Capítulo 3 e ferramentas de CAD específicas do fabricante do dispositivo o qual o sistema foi embarcado.

O Capítulo 5 apresenta a rede intra-chip Hermes[1], que foi modificada para suportar reconfiguração parcial e dinâmica de módulos conectados a ela. Esta nova rede, denominada Artemis[2], é a segunda contribuição deste trabalho.

O Capítulo 6 apresenta diferentes estudos de caso, mostrando como utilizar o fluxo de projeto proposto no Capítulo 4 e a rede Artemis introduzida no Capítulo 5. A finalidade destes estudos de caso é apresentar as principais dificuldades para a implementação de sistemas parcial e dinamicamente reconfiguráveis, e como estas podem ser resolvidas.

Por fim, as conclusões e os trabalhos futuros são apresentados no Capítulo 7.

[1] Segundo a mitologia grega, Hermes é o mensageiro dos Deuses.
[2] Segundo a mitologia grega, Artemis possui o poder de transformar pessoas em animais.

2 SISTEMAS RECONFIGURÁVEIS

Há duas décadas lógica reconfigurável é empregada em arquiteturas reconfiguráveis, ocupando diferentes posições dentro destas. A Figura 2 apresenta diferentes posições na qual a lógica reconfigurável pode ser empregada em arquiteturas reconfiguráveis [HAU98]: como unidade funcional (RFU - *Reconfigurable Functional Unit*) de um processador (A), como coprocessador (B), anexada ao barramento de um processador (C) e de forma autônoma (D).

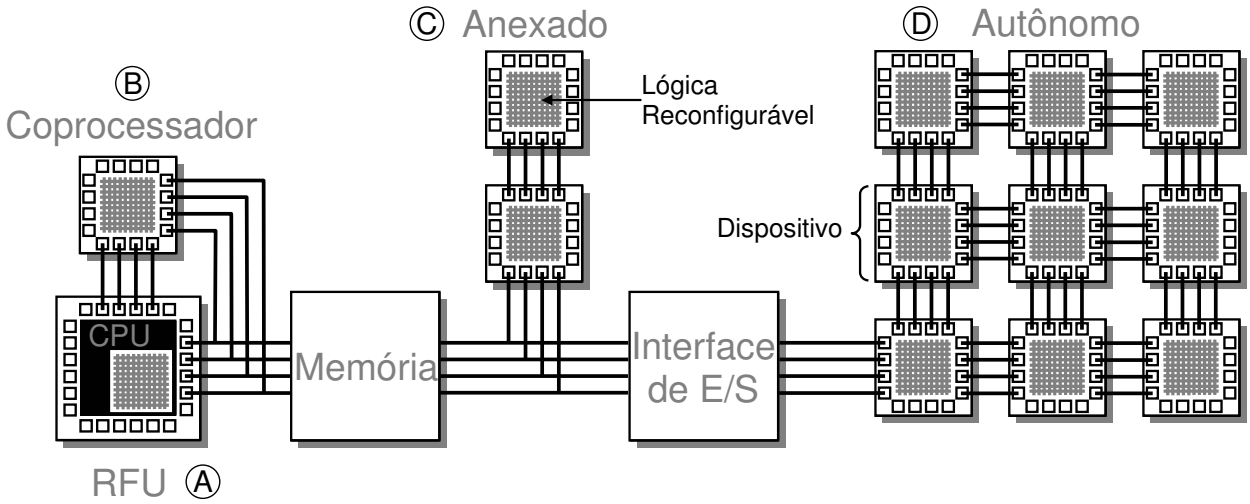


Figura 2 – Tipos de arquiteturas reconfiguráveis [HAU98].

Os primeiros três tipos de arquiteturas (A, B e C) apresentadas na Figura 2 são chamadas de processadores reconfiguráveis e possuem a lógica reconfigurável conectada a um processador mestre, sendo essas chamadas de acopladas. As arquiteturas podem ser fortemente acopladas (RFUs (A)) ou fracamente acopladas (coprocessadores (B) e anexados (C)), devido à distância da lógica reconfigurável ao processador mestre.

Em (A), a lógica reconfigurável é posicionada dentro do processador, sendo normalmente inserida no estágio de execução do mesmo. Esta lógica reconfigurável permite estender o conjunto de instruções do processador e é acessada com baixa latência devido à proximidade entre a lógica reconfigurável e o processador. Em (B), a lógica reconfigurável possui uma conexão dedicada com o processador, sendo esta tratada como um coprocessador. Neste caso a lógica reconfigurável pode, opcionalmente, acessar diretamente a memória do processador e/ou recursos de E/S. Em (C), a lógica reconfigurável não possui uma conexão dedicada com o processador, sendo esta comunicação compartilhada com outros recursos do processador.

Com base nos diferentes tipos de arquiteturas apresentadas, a Figura 3 apresenta a evolução de dispositivos, processadores e sistemas reconfiguráveis ao longo do tempo.

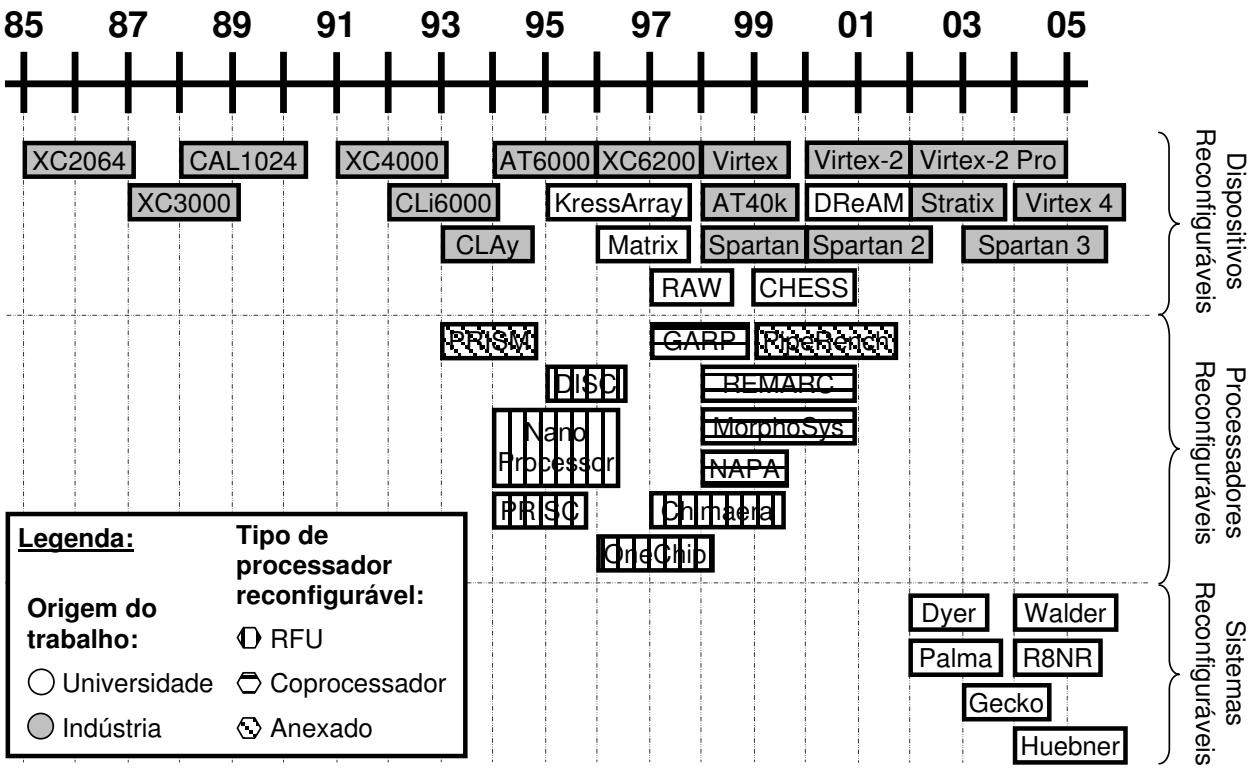


Figura 3 – Evolução de dispositivos, processadores e sistemas reconfiguráveis nas últimas duas décadas.

O primeiro FPGA a ser construído foi o XC2064 em 1985 pela fabricante Xilinx. Após, outros fabricantes surgiram no mercado de FPGAs: Actel, Advanced Micro Devices (AMD), Algotronix, Altera, Concurrent Logic, CrossPoint Solutions, DynaChip, Fairchild, National Semiconductor, Plessey, Plus, QuickLogic, entre outros. Por volta de 1990, trabalhos acadêmicos apresentaram arquiteturas com múltiplos FPGAs comerciais (multi-chip), como por exemplo DeCPerle, CHS2x4, MORRPH, Splash-2 e outros apresentados em [GUC05]. Por volta de 1995, trabalhos acadêmicos mostraram diversas formas de conectar FPGAs a processadores, sendo alguns exemplos destes os processadores reconfiguráveis apresentados na Figura 3. Por volta da mesma época o meio acadêmico desenvolve seus próprios dispositivos reconfiguráveis, sendo estes também apresentados na Figura 3. Estes, de forma geral, são dispositivos reconfiguráveis de grão grande, e não se popularizaram devido à generalidade do grão restringir o conjunto de aplicações que obtêm um desempenho adequado no dispositivo. Por volta do ano 2000 os FPGAs comerciais já possuíam um milhão de portas lógicas equivalentes disponíveis ao projetista, permitindo a construção de aplicações com processadores e hardware dedicado em um único FPGA. Os FPGAs do ano de 2005 já possuem dez milhões de portas lógicas equivalentes disponíveis ao projetista e processadores embarcados.

Uma característica disponível em alguns FPGAs é a reconfiguração parcial e dinâmica. As vantagens destes sobre outros que não possuem esta característica é a possibilidade de modificar parte do hardware do sistema enquanto o restante permanece executando. O primeiro FPGA com esta característica foi o dispositivo CAL1024 da Algotronix, lançado no ano de 1988. Após este, surgiram outros dispositivos que hoje não são mais comercializados: ERA60100 [PLE90], CLi6000 [CON92], CLAy-10 e CLAy-31 [NAT93], AT6000 [ATM94] e XC6200 [XIL97]. Os FPGAs

parcial e dinamicamente reconfiguráveis disponíveis atualmente no mercado são os dispositivos da Atmel e da Xilinx. A Atmel comercializa a família de dispositivos AT40k [ATM98], enquanto a Xilinx vende as famílias Virtex, Virtex-II, Virtex-II Pro e Virtex 4.

No entanto, poucos sistemas reconfiguráveis tiraram proveito da reconfiguração parcial e dinâmica dos FPGAs até o momento. Um dos principais motivos é a inexistência ou a falta de documentação dos fabricantes de FPGAs em como utilizar esta característica. Alguns exemplos de sistemas parcial e dinamicamente reconfiguráveis são apresentados na Figura 3, possuindo estes uma baixa latência de comunicação por terem sido implementados em um único chip. O Gecko é o único sistema reconfigurável que utiliza NoCs para conectar módulos reconfiguráveis, enquanto os outros sistemas utilizam barramento.

As Seções 2.1, 2.2 e 2.3 apresentam, respectivamente, processadores reconfiguráveis, dispositivos reconfiguráveis e sistemas reconfiguráveis, conforme delimitado na Figura 3.

2.1 Processadores Reconfiguráveis

Aplicações com computações intensivas passam em média 90% do tempo em apenas 10% do código [HEN90]. Este fato motiva otimizar o processador para executar os trechos de código responsáveis por este maior consumo de tempo de processamento diretamente em hardware. Deve-se observar que nem sempre o hardware dedicado é a solução para ganho de desempenho, pois em determinados casos o gargalo é o conjunto de operações de E/S ou acessos à memória externa.

Processadores com um conjunto de instruções específicas para a aplicação (*Application-Specific Instruction-Set Processor* - ASIP) provêm flexibilidade e desempenho a um custo de área adicional para cada nova função implementada em hardware. Um exemplo de ASIP é o Xtensa [TEN02] da Tensilica. Processadores com um conjunto de instruções reconfiguráveis (*Reconfigurable Instruction-Set Processor* - RISP) não apresentam um custo adicional de hardware para cada nova instrução, mas sim um custo pré-definido de área onde lógica reconfigurável pode ser programada em tempo de execução.

Segundo [BAR02], a posição da lógica reconfigurável em relação ao processador tem um impacto direto no desempenho dos RISPs e nos tipos de aplicações que são beneficiadas pela lógica reconfigurável. Obter um ganho de desempenho executando um pedaço de código em hardware reconfigurável depende de dois aspectos: tempo de comunicação e tempo de execução [WIT96]. O tempo para executar uma operação é a soma do tempo necessário para enviar e receber dados da/para lógica reconfigurável, e o tempo de processamento. Se este tempo total é menor que o tempo levado pelo processador executar a operação sozinho, então um ganho de desempenho é obtido. Se a lógica reconfigurável ainda não foi configurada para executar uma operação específica, então ainda é necessário adicionar o tempo de configuração [BAR02].

As próximas Seções são organizadas conforme a Tabela 1.

Tabela 1 – Exemplos de processadores reconfiguráveis organizados segundo o critério de acoplamento.

Lógica Reconfigurável Anexada (2.1.1)	Lógica Reconfigurável como Coprocessador (2.1.2)	Lógica Reconfigurável como RFU (2.1.2.3)
PRISM (2.1.1.1) PipeRench (2.1.1.2)	Garp (2.1.2.1) MorphoSys (2.1.2.2) Systolic Ring (2.1.2.3)	Nano Processor (2.1.3.1) DISC (2.1.3.2) OneChip (2.1.3.3) Chimaera (2.1.3.4)

2.1.1 Lógica Reconfigurável Anexada

A principal característica de processadores reconfiguráveis que utilizam lógica reconfigurável anexada a um meio de comunicação (freqüentemente um barramento) é a flexibilidade. Qualquer processador pode ser estendido para utilizar a lógica reconfigurável anexada ao processador através de instruções de leitura e escrita mapeadas em memória. Para isso é necessário inserir um hardware que selecione, a partir do endereço de acesso da instrução, se deve ser acessada a memória ou a lógica reconfigurável.

Uma vantagem desta técnica é que a lógica reconfigurável funciona em paralelo com o processador, liberando o mesmo para outras tarefas ou para a gerência de recursos, enquanto a lógica reconfigurável trata da computação. Outra vantagem é que como a lógica reconfigurável está conectada no mesmo meio de comunicação de outros periféricos do processador, logo a lógica reconfigurável pode fazer uso direto, por exemplo, da memória e periféricos de E/S sem interromper o processador. Tal vantagem pode ser de extrema valia para aplicações I/O bound.

2.1.1.1 PRISM

O PRISM-I (*Processor Reconfiguration through Instruction-Set Metamorphosis*) [ATH93] teve sua primeira publicação por Peter Athanas et al. da Universidade do Estado da Virginia. Ele foi desenvolvido como prova de conceito para mostrar a viabilidade de incorporar um conjunto de instruções configurável a um processador de propósito geral. O PRISM-I consiste na interconexão de um nodo de processamento Armstrong com uma plataforma com hardware reconfigurável por meio de um barramento, conforme ilustrado na Figura 4.

O nodo de processamento Armstrong é baseado em um processador M68010 da Motorola, com freqüência de operação de 10MHz. A plataforma com hardware reconfigurável possui 4 FPGAs 3090 da Xilinx [XIL98], cada um com seis mil portas lógicas equivalentes e 144 pinos de entrada e saída. A interconexão das duas plataformas foi feita pela interface de coprocessador de 16 bits do M68010.

Para avaliar o sistema, o processador faz chamadas de funções implementadas em hardware reconfigurável. As operações testadas consumiram de 48 a 72 ciclos de relógio para completar a operação e entregar a resposta de volta ao processador.

Os testes executados, embora tenham sido iniciais e prejudicados pelo baixo desempenho do meio de comunicação, mostraram que é possível obter ganhos com a implementação de instruções em hardware reconfigurável. Dentre as diversas operações executadas, a que obteve

melhor desempenho foi a função de logaritmo na base 2, executando 54 vezes mais rápido que em software.

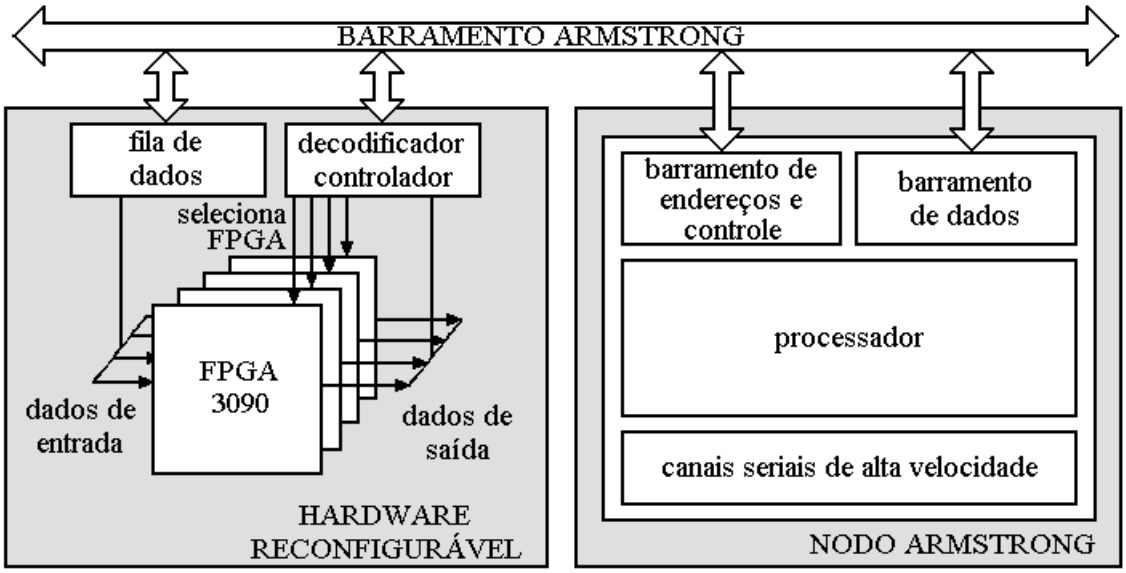


Figura 4 – Arquitetura do PRISM-I.

A segunda versão do PRISM [WAZ93] teve como principais objetivos estender o número de instruções suportadas pelo compilador e reduzir o tempo de comunicação entre o processador e o hardware reconfigurável. Enquanto o PRISM-I leva em média 100ns para devolver a resposta ao processador, o tempo gasto na nova versão é de 30ns. Outra modificação feita foi no tamanho da palavra de dados, que no PRISM-I era fixa em 32 bits, e que no PRISM-II é de 64 bits para entrada e 32 bits para saída. O PRISM-I também possuía a limitação que uma instrução tinha que caber em um único FPGA. A arquitetura do PRISM-II permite que uma única operação seja dividida em até três FPGAs.

Para prover um melhor balanceamento entre hardware e software o processador M68010 foi substituído pelo AMD Am29050, com freqüência de 33MHz e desempenho aproximado de 28 MIPS. Os FPGAs 3090 foram substituídos pelos 4010 com aproximadamente 10 mil portas lógicas equivalentes e 160 pinos de entrada e saída. Todas essas modificações resultaram em um fator de aceleração de aproximadamente 86 vezes em relação a software para uma função de reversão de bits em uma palavra de 32 bits. Isso mostra que essa versão obteve um ganho significativo de desempenho em relação ao PRISM-I, pois quando a mesma função foi executada na primeira versão ela obteve um fator de aceleração de apenas 26 vezes em relação a software.

2.1.1.2 PipeRench

O PipeRench [GOL99] foi proposto por Seth Goldstein et al. da Universidade de Carnegie Mellon. A arquitetura é composta por uma rede de elementos lógicos e de armazenamento interconectados, que atuam como um coprocessador. Esta arquitetura é organizada de forma a implementar um *pipeline* de configurações, sendo particularmente interessante para aplicações

baseadas em fluxos uniformes (como processamento de imagens ou sons), ou qualquer computação simples e regular sobre grandes conjuntos de pequenos elementos de dados.

A Figura 5 apresenta uma visão abstrata de dois estágios do PipeRench e uma visão detalhada de um estágio. Cada PE contém uma ULA e um conjunto de registradores para comunicação entre estágios (registrador de passagem). Cada ULA contém LUTs e controle extra para propagação de *carry*, detecção de zero, e outros bits de controle. Lógica combinacional pode ser implementada usando um conjunto de *n* ULAs de B bits. Funções combinacionais complexas podem ser obtidas com o cascateamento das linhas de propagação de *carry*, através da rede de interconexão.

A partir das redes de interconexão os PEs podem acessar operandos dos estágios anteriores que estão armazenados nos registradores de saída, bem como saídas de outros PEs do mesmo estágio. O barramento global é utilizado tanto pela aplicação para entrada e saída de dados quanto pelos PEs para alcançar os seus destinos, porque os estágios do *pipeline* de uma dada aplicação podem estar fisicamente em qualquer um dos estágios da arquitetura.

Com este dispositivo foi obtido um ganho de desempenho médio de 189 vezes sobre um Ultrasparc II com freqüência de 300 MHz.

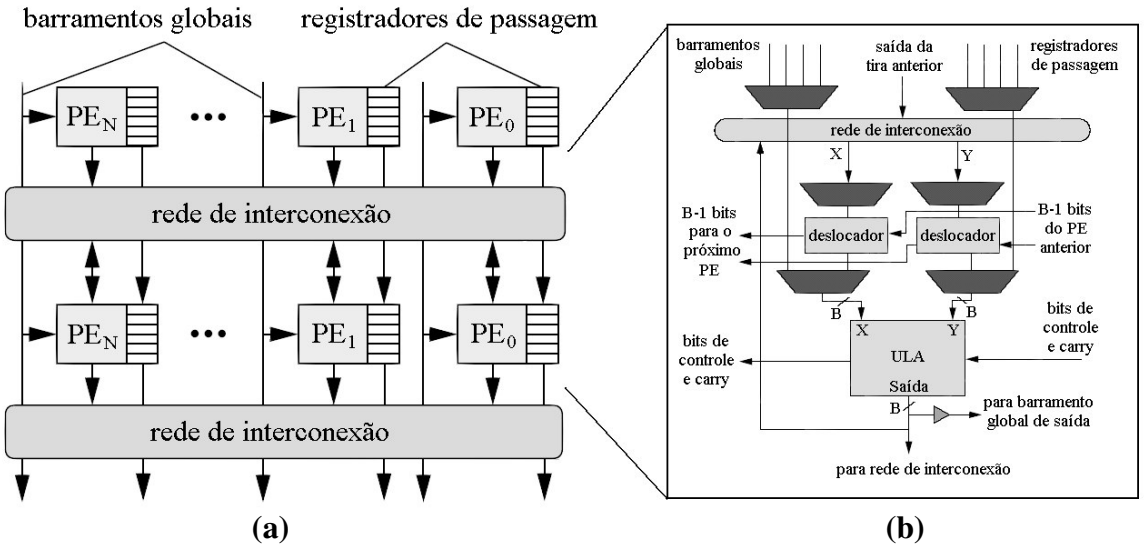


Figura 5 – Arquitetura do PipeRench: (a) duas linhas de PEs e redes de interconexão; (b) arquitetura interna de um estágio do PipeRench.

2.1.2 Lógica Reconfigurável como Coprocessador

Quando a lógica reconfigurável é conectada como coprocessador, existe uma interface dedicada entre processador e coprocessador. Neste caso a comunicação entre processador e coprocessador ocorre sem intervenções externas, pois não há a concorrência de outros periféricos neste canal de comunicação. Em contraponto, não é qualquer processador que pode fazer uso deste tipo de acoplamento, porque este necessita de uma interface específica para conectar um

coprocessador.

Neste tipo de acoplamento a lógica reconfigurável pode, opcionalmente, acessar diretamente a memória do processador e/ou recursos de E/S. No entanto, esta comunicação só pode ocorrer quando o processador não estiver acessando a memória ou recursos de E/S, pois a comunicação com estes periféricos é, normalmente, compartilhada entre processador e lógica reconfigurável.

Este tipo de acoplamento apresenta um custo benefício intermediário aos tipos anexado e RFU, pois aplicações IO-bound concorrem com o processador para acessar a memória e aplicações CPU-bound possuem um tempo intermediário de comunicação entre processador e lógica reconfigurável em relação aos outros tipos de acoplamento.

2.1.2.1 Garp

O Garp [HAU97a] teve sua primeira publicação por John Hauser da Universidade da Califórnia em Berkeley e foi projetado com o principal objetivo de acelerar laços de aplicações de propósito geral. O projeto implementa um coprocessador conectado ao MIPS, que consiste em uma malha bidimensional de CLBs interconectadas por recursos de roteamento programáveis (Figura 6). A célula lógica básica da matriz reconfigurável é uma unidade de 2 bits com quatro entradas de 2 bits cada. Quatro canais de 32 bits de dados e um canal de 32 bits de endereços comunicam o processador à matriz reconfigurável.

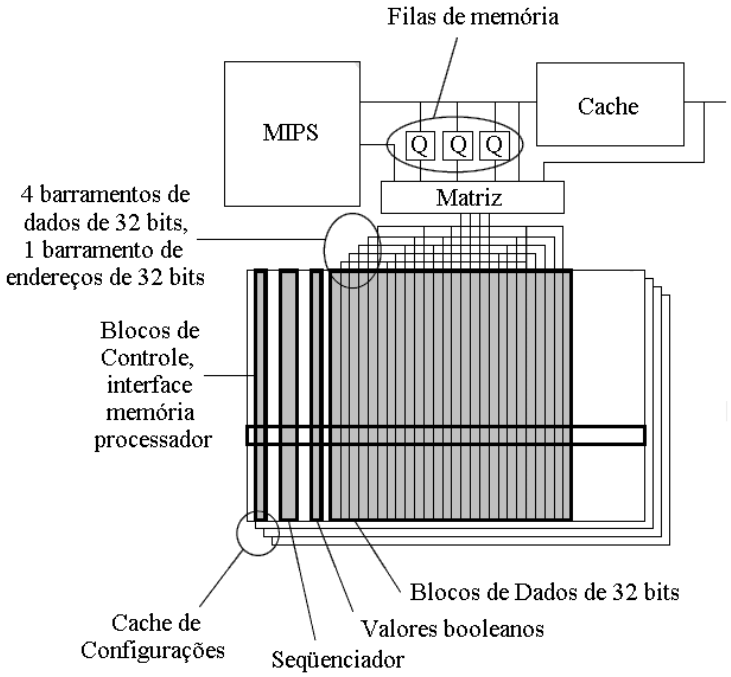


Figura 6 – Arquitetura do Garp.

A matriz reconfigurável possui pelo menos 24 blocos lógicos em uma coluna, dos quais 16 blocos a partir da primeira posição são conectados ao barramento compartilhado processador-memória. Na tentativa de minimizar os acessos ao barramento compartilhado, o projeto emprega a

utilização de uma memória cache que pode armazenar quatro configurações totais da matriz ou diversas configurações parciais. Uma reconfiguração total buscada da cache pode entrar em funcionamento em até cinco ciclos de relógio.

Para avaliar o desempenho do Garp foi utilizado um algoritmo de filtro de imagem que calcula a média dos pixels. A execução do Garp obteve um desempenho 43 vezes melhor que um Ultrasparc 170.

2.1.2.2 Morphosys

O Morphosys foi proposto por Hartej Singh et al. [SIN98] da Universidade da Califórnia em Irvine e consiste em sete componentes principais (Figura 7): um processador baseado no MIPS (Tiny_RISC), uma malha 8x8 de células reconfiguráveis, uma memória de contextos, uma fila, uma cache de dados, um controlador de DMA e duas memórias principais (uma do processador e outra de contextos da malha reconfigurável).

Cada célula da malha reconfigurável pode ser individualmente reconfigurada e possui uma ULA de 16 bits com módulo de multiplicação acoplado, uma unidade de deslocamento, dois multiplexadores que selecionam as entradas da célula, um registrador de contextos e quatro registradores de propósito geral.

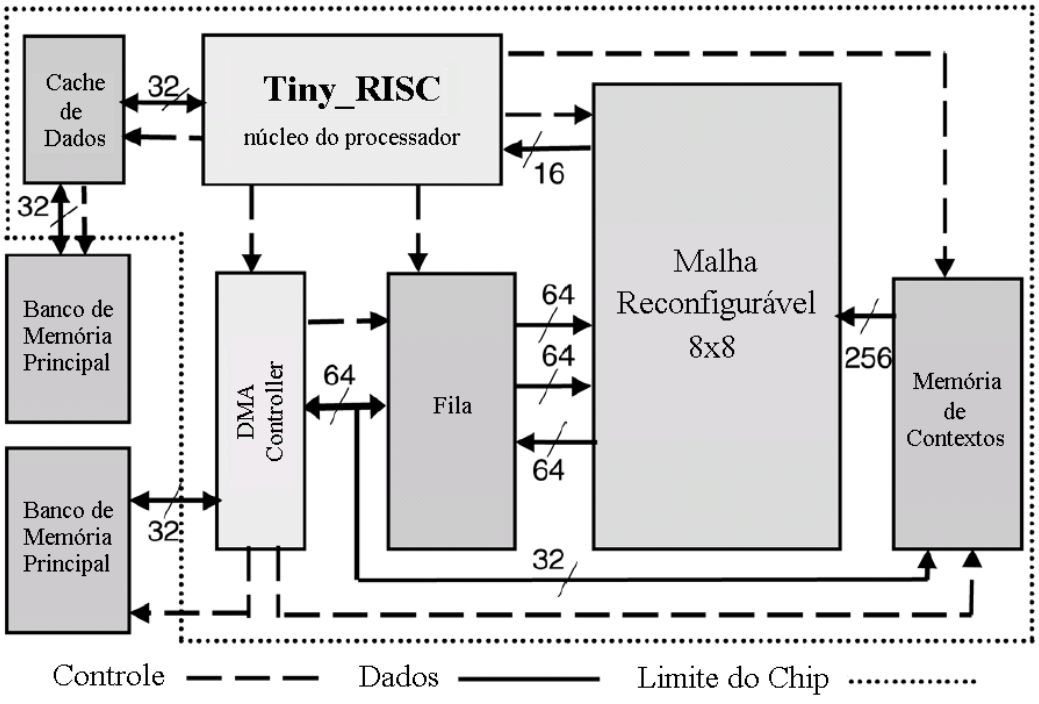


Figura 7 – Arquitetura do MorphoSys.

Enquanto o processador executa as tarefas seqüenciais e de controle de transferência de dados entre o hardware reconfigurável e a memória, o hardware reconfigurável explora o paralelismo disponível do algoritmo da aplicação. O processador RISC possui duas classes de instruções específicas para controlar os componentes do MorphoSys. Uma dessas classes é responsável por carregar contextos da memória principal para a memória de contextos e a outra é

responsável por iniciar transferências de dados entre a memória principal e a fila, sendo essa classe denominada de instruções de DMA.

A fila é uma memória de dados interna para a malha reconfigurável, provendo 128 palavras de 16 bits de memória. Um barramento de operandos de 128 bits conecta a fila com a matriz de células reconfiguráveis. Todas as células reconfiguráveis de uma mesma linha compartilham 16 bits do barramento de operandos. Uma característica importante da malha reconfigurável é a sua interconexão em três níveis, o qual pode ser vista na Figura 8. O primeiro nível acessa o vizinho mais próximo da mesma linha/coluna. O segundo nível provê conectividade completa com a linha/coluna do mesmo quadrante. O terceiro nível provê conectividade inter-quadrante, que consiste em barramentos que percorrem todas as linhas e colunas, cruzando as bordas dos quadrantes.

Um dos algoritmos utilizados para avaliar o desempenho do MorphoSys foi o de DCT, sendo executado em 37 ciclos de relógio. Este tempo foi mais de seis vezes menor que a execução do algoritmo em um computador com processador Pentium MMX com instruções otimizadas.

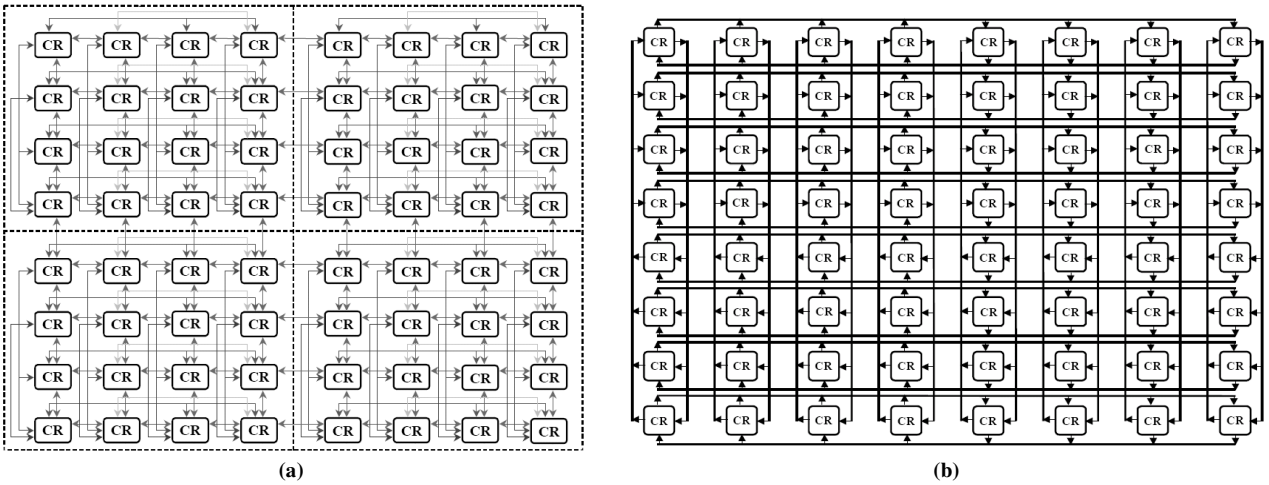


Figura 8 – Conectividade das 8x8 Células Reconfiguráveis do MorphoSys em três níveis: (a) primeiro e segundo níveis; (b) terceiro nível.

2.1.2.3 Systolic Ring

O Systolic Ring foi proposto por Gilles Sassatelli et al. [SAS02] da universidade de Montpellier e foi projetado com o principal objetivo de acelerar computações intensivas orientadas a dados (e.g. multimídia e processamento de sinais). A Figura 9 apresenta um diagrama de blocos da arquitetura do Systolic Ring.

O Systolic Ring possui uma rede em anel como meio de interconexão de blocos lógicos, denominados D-Nodes. Cada D-Node possui uma ULA de 16 bits, um banco de registradores 4x16 bits e uma unidade de controle. A unidade de controle possui 9 registradores, um contador para até 8 estados e um multiplexador 8x1. Esta unidade de controle gerencia o D-Node quando o mesmo está operando em modo local. O D-Node também pode executar em modo global (normal), o qual

executa microinstruções originadas do controlador de configurações. O controlador de configurações é formado por um processador RISC personalizado para funcionar como um controlador de configurações. A tarefa dele é gerenciar dinamicamente as configurações da rede e controlar a comunicação de dados com um processador hospedeiro (suprimido da Figura 9).

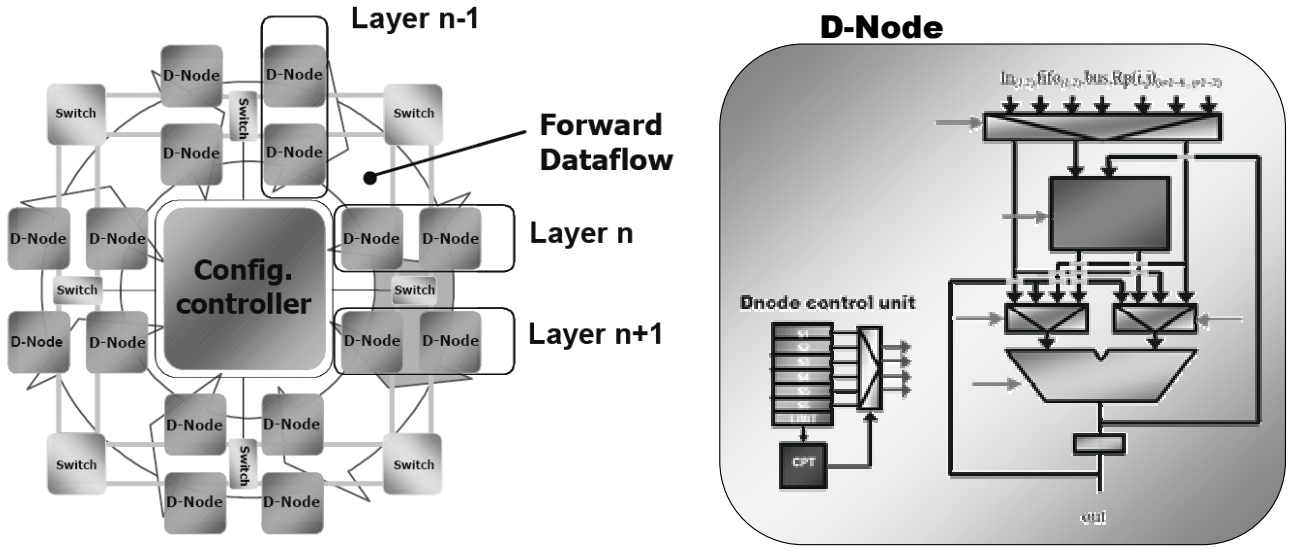


Figura 9 – Arquitetura do Systolic Ring.

O Systolic Ring opera como segue. O processador hospedeiro inicia a execução de uma aplicação e envia o código objeto a ser executado pelo processador do controlador de configurações. Ao ser recebido o código objeto, o controlador de configurações pode configurar o Systolic Ring ciclo a ciclo se assim for configurado e inicia a recepção de dados a serem computados do processador hospedeiro.

Um Systolic Ring com 8 D-Nodes possui um poder máximo de computação de 1600 MIPS a 200 MHz, enquanto um Pentium II 450 MHz possui um poder máximo de computação de 400 MIPS.

2.1.3 Lógica Reconfigurável como RFU

Quando a lógica reconfigurável é conectada a um processador como RFU, esta é posicionada dentro do processador, sendo normalmente inserida no estágio de execução do mesmo. Este posicionamento permite estender o conjunto de instruções do processador com um tempo de acesso inferior aos oferecidos pelos outros tipos de acoplamento.

2.1.3.1 Nano Processor

O Nano Processor [WIR94] foi proposto por Michael Wirthlin et al. da Universidade de Brigham Young. O seu principal objetivo é poder ser implementado em FPGAs da época, por isso ele dispõe de uma arquitetura simples, ocupando pouca área. Ele possui seis instruções nativas e permite um total de 32 instruções. O processador é organizado hierarquicamente em três níveis,

conforme apresentado na Figura 10.

O nível mais interno do processador é o núcleo do Nano Processor, que possui as seis instruções nativas e não foi desenvolvido para ser modificado. Ele possui um registrador de instruções (*Instruction Register* - IR) de 5 bits, um registrador de página de endereços (*Page Address Register* - PAR) de 3 bits, um contador de programa (*Program Counter* - PC) de 8 bits, um registrador de endereços (*Address Register* - AR) de 8 bits e um acumulador de 8 bits com 1 bit de *carry*. Nesta arquitetura apenas instruções aritméticas de adição e subtração foram implementadas. O núcleo do Nano Processor ocupa 40 CLBs da família 3000 de FPGAs da Xilinx [XIL98].

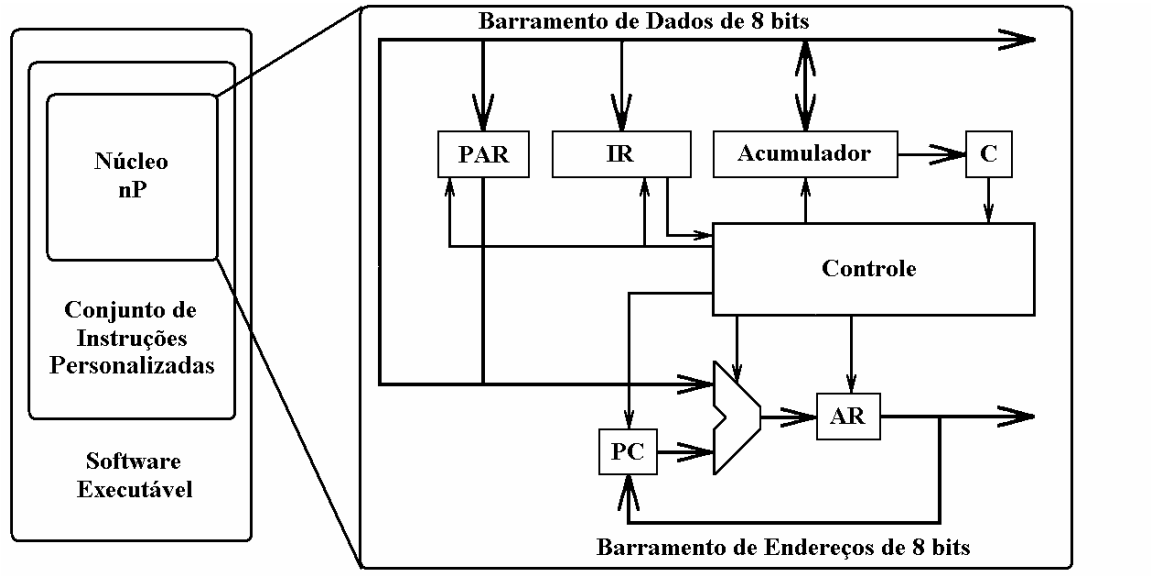


Figura 10 – Organização e arquitetura do Nano Processor.

O nível intermediário do processador é chamado de Conjunto de Instruções Personalizadas. Nela são implementadas as instruções específicas ao problema a ser tratado pelo processador. Essas instruções podem ser selecionadas de uma biblioteca previamente desenvolvida ou criadas a partir de editores de esquemáticos ou ferramentas de síntese. O nível mais externo é o Software Executável. Nele é implementado a aplicação em código assembly onde são feitas as chamadas das instruções nativas e personalizadas.

O Nano Processor foi prototipado em uma plataforma da National Technologies. A plataforma inclui dois FPGAs 3090 da Xilinx [XIL98], duas memórias SRAMs 32k x 8, uma memória DRAM de 1 Mb, um codec stereo de 16 bits e uma interface com o PC. O melhor ganho de desempenho obtido com esta plataforma foi de 240 vezes em relação a um processador 486 executando a 33 MHz para a aplicação *saturating mixer* [WIR94].

2.1.3.2 DISC

O DISC (*Dynamic Instruction Set Computer*) [WIR95] também foi proposto por Michael Wirthlin et al. da Universidade de Brigham Young, um ano após a publicação do Nano Processor. Ele foi desenvolvido para suportar modificações no seu conjunto de instruções em tempo de execução. Para isso ele deve ser implementado em dispositivos que suportem reconfigurações

parciais e dinâmicas. No DISC todas as instruções são implementadas como módulos de hardware e são substituídas quando não existe mais espaço disponível para a instrução seguinte. Essa abordagem possui a vantagem de não só utilizar uma área em hardware virtualmente maior que a existente fisicamente, mas também de permitir implementar funções complexas com melhor desempenho quando comparado a software. A arquitetura do DISC e alguns exemplos de instruções são apresentados na Figura 11.

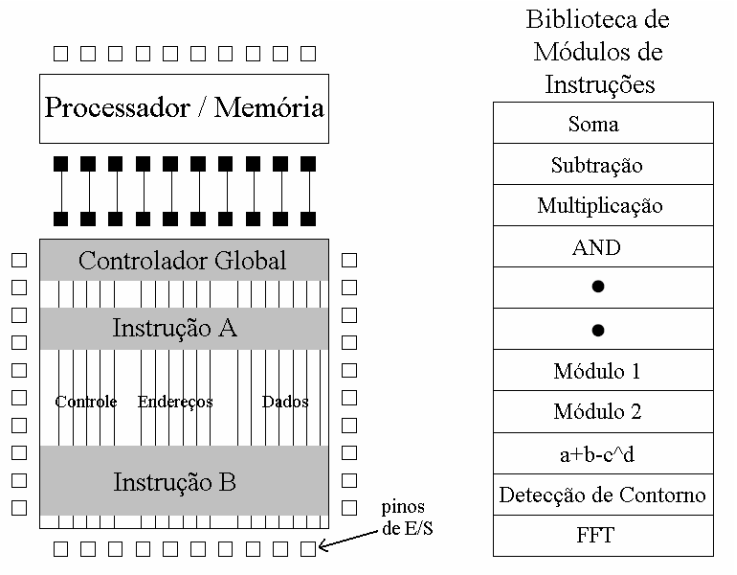


Figura 11 – Arquitetura do DISC e exemplo de biblioteca de módulos de instruções.

Antes de iniciar o DISC, um programa é carregado na memória, o módulo de controle global é carregado em hardware e o espaço de instruções dinâmicas é liberado. A seguir o DISC inicia a execução do programa. Quando for feita a chamada de uma instrução que não se encontra configurada, o processador é bloqueado e uma requisição de configuração de instrução é enviada ao computador hospedeiro pelos pinos de entrada e saída do sistema. O computador hospedeiro verifica o tamanho do módulo requisitado, seleciona onde o mesmo será posicionado, remove um ou mais módulos se não existir espaço suficiente no dispositivo, e reposiciona o módulo para onde ele havia sido selecionado. A seguir, o computador hospedeiro envia a nova configuração para o sistema, o processador sai do estado de bloqueio e a instrução é executada.

Para avaliar o desempenho da arquitetura foi desenvolvido um algoritmo de filtro de imagem tanto com instruções de propósito geral quanto com instruções específicas. O ganho de desempenho obtido pela versão com instruções específicas foi de 23,5 vezes sobre a implementação com instruções de propósito geral.

2.1.3.3 OneChip

O OneChip [WIT96] teve a sua primeira publicação por Ralph Wittig et al. da Universidade de Toronto. Nesse projeto é defendida a idéia que o processador deve estar fortemente acoplado ao hardware reconfigurável, minimizando os bloqueios do processador com uma comunicação rápida e eficiente entre eles.

Os recursos reconfiguráveis foram integrados no estágio de execução do processador, sendo chamados de Unidades Funcionais Programáveis (*Programmable Functional Unit* - PFU). As PFUs são adicionadas em paralelo às Unidades Funcionais Básicas (*Basic Functional Unit* - BFU) do processador. As PFUs implementam funções específicas a uma aplicação, podendo ser circuitos seqüenciais ou combinacionais. As PFUs também podem ser utilizadas como lógica de cola para que o processamento de uma determinada função venha do mundo externo. As BFUs são responsáveis pelas operações lógicas e aritméticas elementares de qualquer aplicação.

O OneChip foi prototipado em um Transmogrifier-1 [LEW98], que possui quatro FPGAs 4010 da Xilinx [XIL99], dois chips de interconexão Aptix AX1024 e quatro memórias SRAMs 32k x 9. O processador DLX [HEN03] baseado no MIPS foi particionado em três FPGAs, deixando um FPGA para PFUs do usuário. O OneChip obteve um fator de aceleração de quase 50 vezes na aplicação de uma DCT (*Discrete Cosine Transform*) em relação à execução em uma Workstation SGI Indy rodando a 150MHz com processador MIPS R4400.

A versão posterior do OneChip [JAC99], chamada de OneChip-98, foi publicada por Jeffrey Jacob e Paul Chow ambos da Universidade de Toronto. Realizando uma análise detalhada sobre o OneChip eles detectaram que o gargalo não era mais a interface entre o processador e a lógica reconfigurável, mas sim a interface destes com a memória. Por este motivo, o OneChip-98, apresentado na Figura 12, busca uma maior taxa de transferência de dados com a memória, permitindo o acesso concorrente a mesma pela lógica reconfigurável e pelo processador.

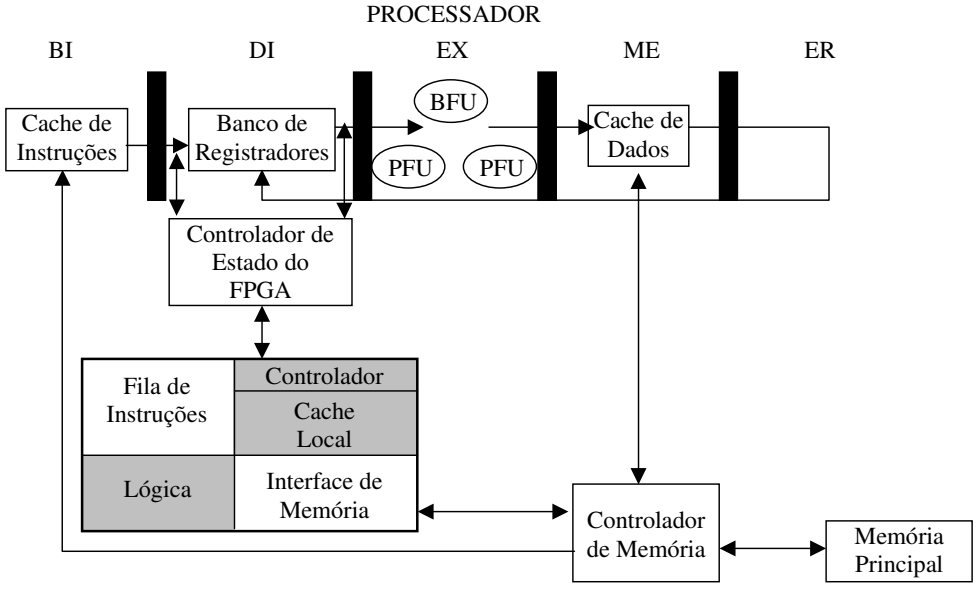


Figura 12 – Arquitetura do OneChip-98.

Para aumentar a taxa de transferência com a memória principal, caches foram adicionadas ao sistema, implementando uma hierarquia de memória semelhante à empregada em computadores pessoais. Conseqüentemente, um mesmo dado pode ser trazido tanto para a cache do processador quanto para a cache do FPGA, causando um possível problema de coerência de caches. O tratamento deste problema é analisado no artigo [JAC99].

O OneChip-98 foi prototipado em um Transmogrifier-2 [LEW98] com dois PLDs Altera Flex10K50. A aplicação de um filtro FIR 64-tap obteve uma aceleração de 32 vezes sobre um Ultrasparc 2 rodando a 300MHz. Os autores apontam como aplicações alvo problemas que acessam muito a memória.

2.1.3.4 Chimaera

O Chimaera [HAU97b] foi proposto por Scott Hauck et al. da Universidade de Washington. Ele possui Unidades Funcionais Reconfiguráveis (*Reconfigurable Functional Unit* - RFU) pequenas o suficiente para serem integradas no próprio processador, buscando minimizar o gargalo de comunicação entre processador e lógica reconfigurável. Esta abordagem permite utilizar a lógica reconfigurável para acelerar as computações críticas das aplicações, enquanto aproveita a flexibilidade dos processadores para a criação de software. A Figura 13 apresenta a arquitetura do Chimaera.

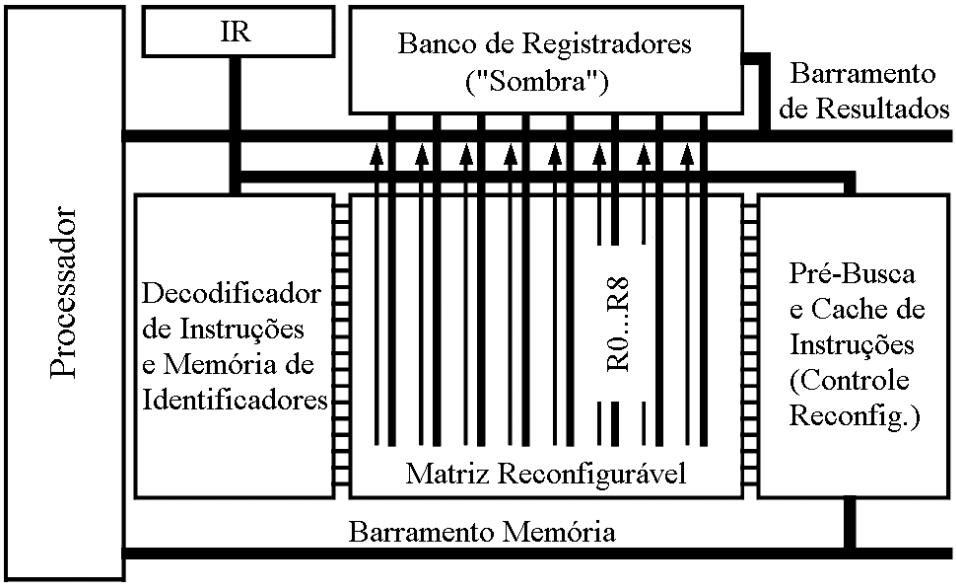


Figura 13 – Arquitetura do Chimaera.

O principal componente do sistema é a matriz reconfigurável, que possui uma arquitetura própria baseada no Triptych [EBE95], na família FLEX 8000 da Altera [ALT04] e no PRISC [RAZ94]. A matriz recebe entradas do banco de registradores chamado “sombra”, que possui cópia de um subconjunto de valores do banco de registradores do processador. Cada modificação feita no banco de registradores do processador é propagada pelo barramento, assim atualizando o “sombra”. As instruções que forem carregadas na matriz podem utilizar uma ou mais linhas de RFUs. Sempre que uma linha de RFUs for configurada, uma memória conectada à matriz armazena, em uma posição específica para a linha da RFU, o identificador da instrução que foi carregado na linha. No momento em que uma operação da matriz for executada, as linhas de saída da matriz que possuem o identificador da instrução são habilitadas a colocar o resultado no barramento.

Enquanto o processador executa instruções nativas, o módulo de pré-busca verifica a cada

ciclo de relógio o Registrador de Instruções (*Instruction Register* - IR), de forma "especulativa". Isso permite iniciar a execução da operação na matriz antes mesmo da chamada ter sido feita pelo processador, assim reduzindo ou eliminando o bloqueio do mesmo à espera do resultado. Quando a chamada é de fato executada pelo processador, a matriz escreve o resultado no barramento de resultados caso a operação já tenha sido computada, caso contrário ela bloqueio o processador até a computação ser efetuada. Também pode ocorrer da operação não estar presente na matriz, assim causando um bloqueio no processador até que a operação seja buscada da cache de instruções ou da memória, parcialmente configurada na matriz e executada. Neste caso o tempo de reconfiguração pode ser significativo, logo o programador deve evitar ficar recarregando a RFU constantemente.

O Chimaera possui nove registradores que podem servir como origem de uma operação a ser executada nas RFUs. O programador deve saber quais destes registradores devem ser carregados antes de chamar a operação a ser executada na matriz, pois quando o processador chama uma operação implementada na RFU é passado como parâmetro somente o identificador da operação e o registrador de retorno.

Uma célula da matriz reconfigurável com sua estrutura de roteamento é apresentada na Figura 14. Em cada linha da matriz, existe uma célula por bit do processador. Por exemplo, um processador de 32 bits possui 32 células por linha, portanto uma célula N tem acesso ao enésimo bit dos registradores R0 a R8. Cada célula possui quatro entradas (I1-I4) e quatro saídas (O1-O4) que são utilizadas para comunicação com outras células.

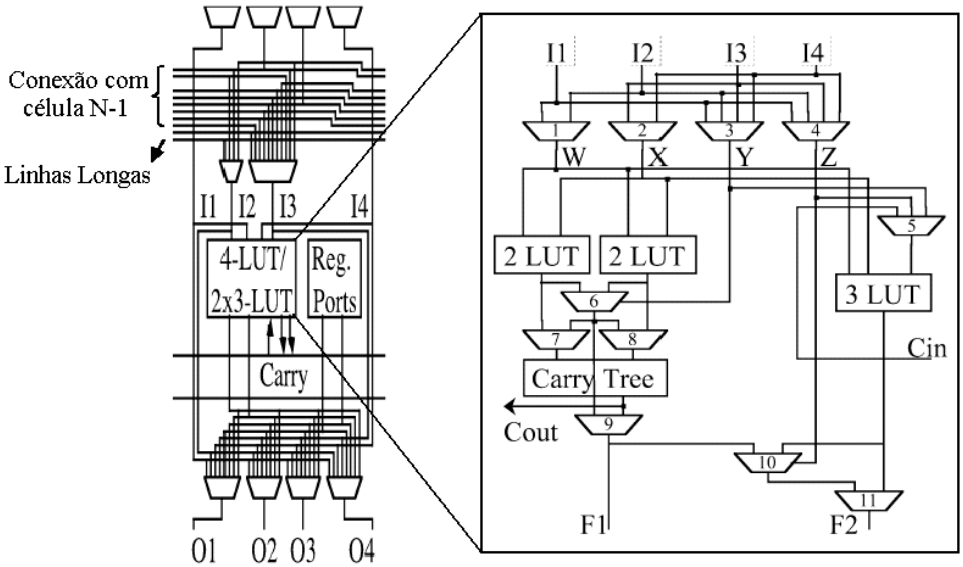


Figura 14 – Célula da matriz reconfigurável e estrutura de roteamento do Chimaera.

As RFUs do Chimaera foram desenvolvidas para tratar com eficiência instruções típicas de um processador como AND, OR, XOR, XNOR, operações aritméticas simples, condições de salto e combinações destas. Essa arquitetura, embora simples, apresentou desempenhos até 160 vezes superiores do que a execução do processador sem lógica reconfigurável.

2.2 Dispositivos Reconfiguráveis

Esta Seção apresenta exemplos de como a lógica reconfigurável pode ser utilizada de forma autônoma, ou seja, sem levar em consideração um processador. Neste sentido são apresentadas nas próximas Seções dispositivos reconfiguráveis, organizados conforme a Tabela 2.

Tabela 2 – Exemplos de dispositivos reconfiguráveis apresentados na Seção 2.2.

Dispositivos Reconfiguráveis Acadêmicos (2.2.1)	Dispositivos Reconfiguráveis Comerciais (2.2.2)
KressArray (2.2.1.1)	CAL1024 (2.2.2.1)
Matrix (2.2.1.2)	ERA60100 (2.2.2.2)
Chess (0)	XC6200 (2.2.2.3)
Dream (2.2.1.4)	Virtex-II (2.2.2.4)

2.2.1 Dispositivos Reconfiguráveis Acadêmicos

A maior parte dos dispositivos reconfiguráveis acadêmicos encontrados na literatura são dispositivos de granularidade grande. Um dos principais motivos desta tendência é uma maior simplicidade de mapeamento e roteamento para este tipo de dispositivo, reduzindo a complexidade da implementação de ferramentas de CAD e o tempo de execução destas ferramentas. Além disso, como os blocos lógicos trabalham com palavras de dezenas de bits, o número de elementos de interconexão é reduzido, pois todos os fios de uma palavra são roteados através de um único elemento de chaveamento. A conseqüência deste fato é um menor consumo de energia e área. Outra vantagem é um menor tempo de reconfiguração em comparação com os dispositivos de grão pequeno, pois a complexidade é reduzida devido a um menor número de chaves de roteamento e elementos reconfiguráveis. A desvantagem de dispositivos reconfiguráveis de grão grande é que as operações e o tamanho da palavra são limitadas pelo grão, tratando operações e tamanhos de palavra múltiplos a sua arquitetura. Em certos casos, este fato pode causar uma perda de eficiência em consumo de potência e desempenho.

As próximas Seções apresentam alguns exemplos de dispositivos reconfiguráveis acadêmicos de grão grande.

2.2.1.1 KressArray

O KressArray-I [HAR95], publicado por Rainer Kress et al. da Universidade de Kaiserslautern, pode ser considerado como a primeira arquitetura de grão grande reconfigurável, pois suas Unidades de Blocos de Dados (*Datapath Unit* - DPU) manipulam dados de 32 bits. O conjunto de DPUs do KressArray-I é chamado de Arquitetura de Blocos de Dados Reconfiguráveis (*Reconfigurable Datapath Architecture* - RDPA), possuindo topologia em malha e DPUs com dois registradores de entrada e dois registradores de saída, conforme apresentado na Figura 15. As operações são dirigidas a dados, ou seja, elas são avaliadas logo que todos os operandos estão disponíveis, utilizando para isso um protocolo semelhante a um *handshake* para a sincronização de

dados entre DPUs.

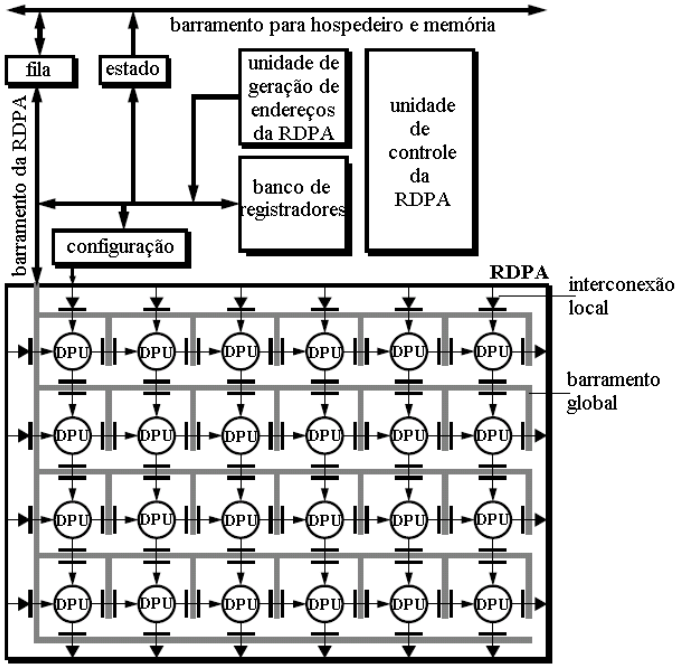


Figura 15 – Arquitetura do KressArray-I.

O KressArray-I provê barramentos locais e globais. O barramento local interconecta os vizinhos e é implementado como uma malha unidirecional para reduzir gastos em área. O barramento global interconecta cada DPU da arquitetura, sendo utilizado para entrada e saída do KressArray-I e para propagação de resultados intermediários de DPUs distantes. Outra alternativa de projeto empregada para reduzir área foi a multiplexação das interconexões globais, passando assim a tarefa de gerência para um escalonador externo. O escalonamento do acesso a dados é feito pelo ambiente de software, que calcula o tempo de execução de toda a aplicação e escalona os dados buscando atingir o melhor desempenho.

O KressArray-III [HAR97] é uma versão posterior do KressArray-I que passou a chamar as DPUs de RDPUs. Nesta arquitetura as conexões passaram a ser bidirecionais, conectando-se as quatro RDPUs vizinhas. Aqui "bidirecional" significa que a direção é selecionada em tempo de configuração, ou seja, ela é fixa durante a execução. Também foi inserido nesta arquitetura um hardware adicional responsável por paralelizar o acesso à memória, permitindo acesso em modo rajada.

No KressArray-III as RDPUs podem servir tanto para roteamento quanto para a aplicação de uma determinada função. O conjunto de funções também foi estendido, agora podendo executar qualquer operação aritmética não ponto-flutuante da linguagem C.

2.2.1.2 MATRIX

A arquitetura MATRIX (*Multiple ALU Architecture wiTh Reconfigurable Interconnect eXperiment*) [MIR96] foi proposta por Ethan Mirsky e André DeHon do MIT. Ela possui topologia em malha que conecta Unidades Básicas Funcionais (*Basic Functional Unit* - BFU) de 8 bits. Cada

BFU (Figura 16) contém 256 palavras de 8 bits de memória, uma ULA de 8 bits com uma unidade de multiplicação agregada e uma unidade de controle implementada através de PLA.

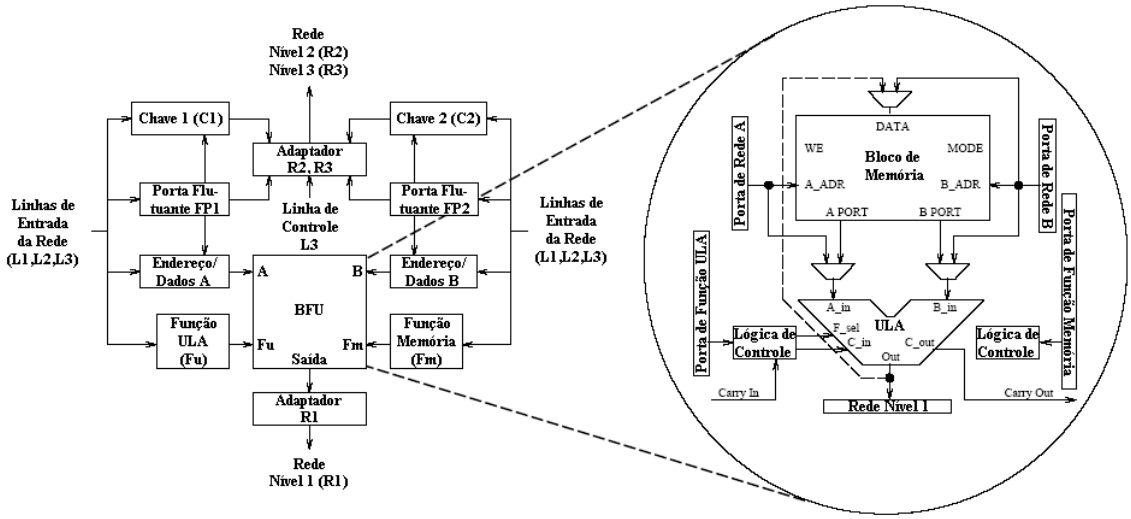
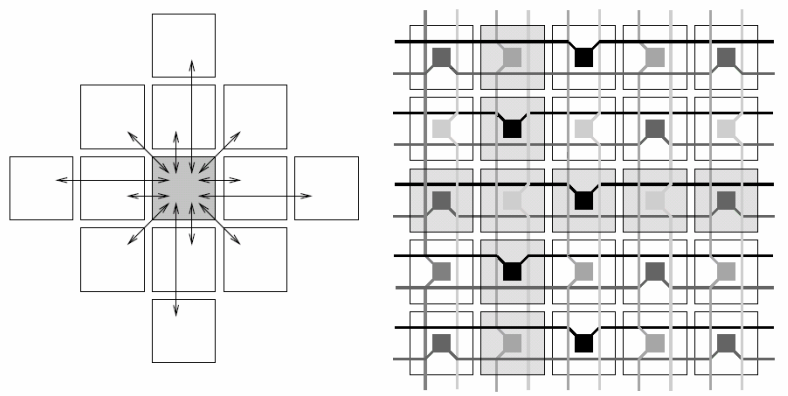


Figura 16 – BFU da MATRIX.

A rede MATRIX é uma coleção hierárquica de barramentos de 8 bits que possui a opção de dinamicamente trocar as conexões. A rede interconecta os vizinhos mais próximos (Figura 17A), vizinhos a uma distância de quatro posições (Figura 17B) e também interconecta BFUs por linhas globais. Essa estrutura hierárquica de interconexão atinge alta taxa de transferência de dados e, em contraponto, dificulta o mapeamento da aplicação.



(A) (B)

Figura 17 – Interconexão da MATRIX: (A) com vizinhos próximos;
(B) com vizinhos a uma distância de 4 posições.

2.2.1.3 CHESS

O CHESS [MAR99] foi desenvolvido por Alan Marshall et al. da HP Labs e possui topologia em malha. A unidade de computação fundamental é uma ULA de quatro bits, que possui um conjunto de 16 instruções básicas. Ao contrário de outras arquiteturas aritméticas reconfiguráveis, o CHESS possui uma arquitetura no formato de um tabuleiro de xadrez, conforme

apresentado na Figura 18. Cada ULA contém barramentos de quatro bits de largura de entrada e saída em cada um dos seus quatro lados. Isto possibilita o envio e recebimento de dados de/para qualquer uma das oito ULAs à sua volta.

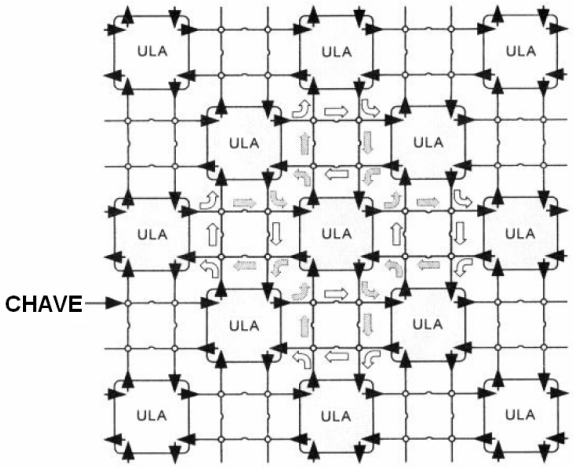


Figura 18 – Arquitetura CHESS.

Em tempo de execução qualquer uma das chaves pode ser utilizada como memória de 16 palavras de 4 bits. Neste modo a chave permanece desconectada, no entanto, barramentos que passam sobre a chave ainda podem continuar operando. Logo, se um grande número de chaves for utilizado como RAM a capacidade de roteamento fica reduzida. Para evitar este problema, o CHESS suporta a inserção de BRAMs.

O CHESS provê a capacidade de encadeamento de ULAs, o que é útil para computações seriais de nibbles. As instruções executadas pelas ULAs podem ser estáticas ou dinâmicas. Instruções estáticas são armazenadas como parte da configuração fixa da ULA. Instruções dinâmicas são geradas pelo encadeamento de ULAs. Isso permite que instruções sejam modificadas ciclo-a-ciclo, suportando determinadas execuções como macros, implementando processadores específicos ou provendo um efeito de arquitetura reconfigurável de grão fino.

A arquitetura do CHESS foi otimizada para reduzir atrasos de comunicação, favorecendo interconexões locais entre os elementos de processamento e não possuindo barramentos globais. Interconexões longas podem ser obtidas pela conexão de múltiplos segmentos de barramentos ou pela utilização do CHESS como um *pipeline*.

2.2.1.4 DReAM

O DReAM (*Dynamically Reconfigurable Architecture for Mobile Systems*) [BEC00] foi desenvolvido na Universidade de Darmstadt por Jürgen Becker et al. O DReAM possui uma topologia em malha de unidades de processamento reconfiguráveis (*Reconfigurable Processing Unit* - RPU). Ele foi fabricado pela Mietec/Alcatel com tecnologia CMOS 0.35µm. O DReAM é composto pela interconexão de 8 componentes, conforme apresentado na Figura 19. Enquanto o barramento local liga RPUs vizinhas, a interconexão global é composta por barramentos segmentados por chaves.

A estrutura da RPU foi otimizada para requisitos de sistemas de comunicação móvel, sendo desenvolvida para executar manipulações de dados aritméticos. A RPU possui duas RAMs de 16 por 8 bits dupla porta e duas unidades de execução de 8 bits dinamicamente reconfiguráveis, chamadas de Unidades Reconfiguráveis de Processamento Aritmético (*Reconfigurable Arithmetic Processing Unit* - RAP). As RAMs de dupla presentes nas RPUs são utilizadas tanto como LUTs nas operações de multiplicação quanto como memória de dados ou como fila para armazenar resultados intermediários. Cada RAP integra dois deslocadores e uma ULA.

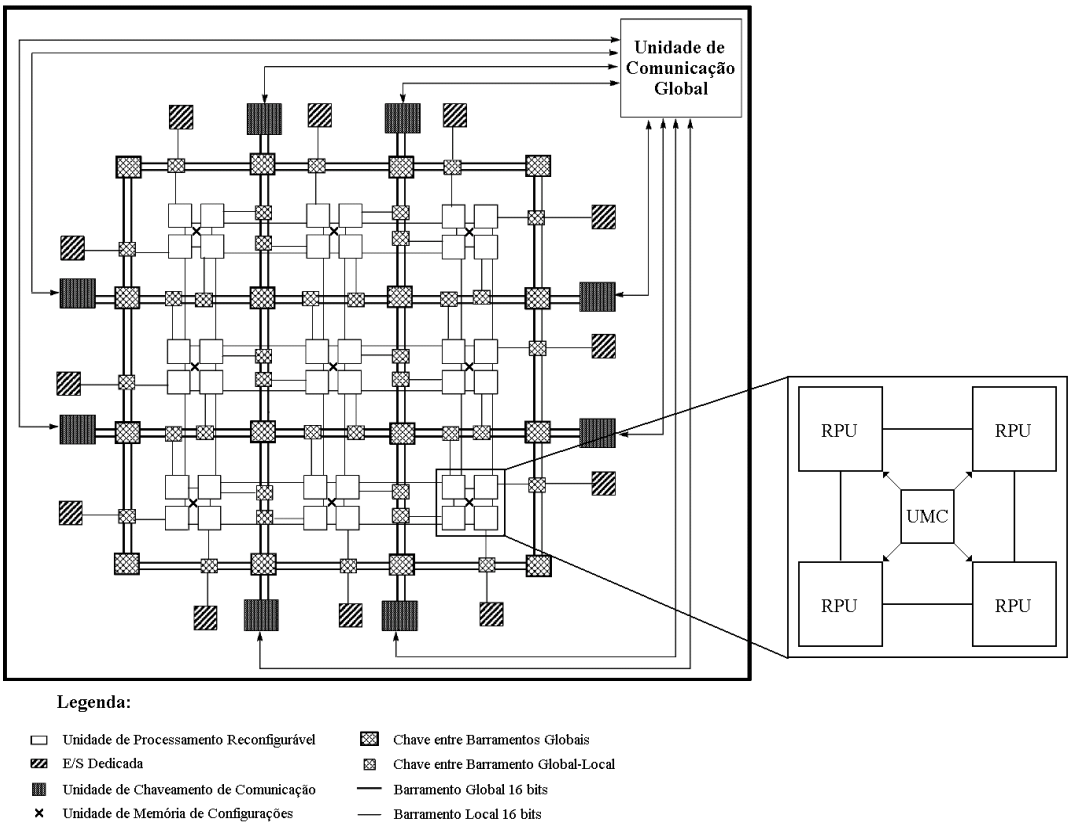


Figura 19 – Arquitetura DReAM.

2.2.2 Dispositivos Reconfiguráveis Comerciais

Esta Seção apresenta a arquitetura interna de FPGAs parcial e dinamicamente reconfiguráveis das últimas duas décadas.

2.2.2.1 CAL1024

O dispositivo CAL1024 da Algotronix foi o primeiro dispositivo parcial e dinamicamente reconfigurável a ser fabricado. As especificações técnicas do dispositivo surgiram da tese de doutorado de Tom Kean [KEA88]. O dispositivo possui uma matriz de 32 por 32 células (1024 células) configuráveis, com 5 mil portas lógicas equivalentes. O dispositivo possui 128 pinos de E/S, permitindo implementar plataformas com múltiplos FPGAs.

A estrutura básica de uma célula do CAL1024 é apresentada na Figura 20. A célula possui

conexões de entrada e saída de um bit com cada uma das quatro células vizinhas NEWS (*North*, *East*, *West*, *South*) e duas entradas globais (G_1 e G_2). Os sinais globais foram implementados com o propósito de serem utilizados como sinais de relógio do circuito, no entanto estes podem ser utilizados pela lógica do usuário. A célula pode implementar qualquer uma das 20 funções apresentadas na Tabela 3. Observar que ao invés de utilizar LUTs, a função da célula básica é definida por um conjunto de multiplexadores.

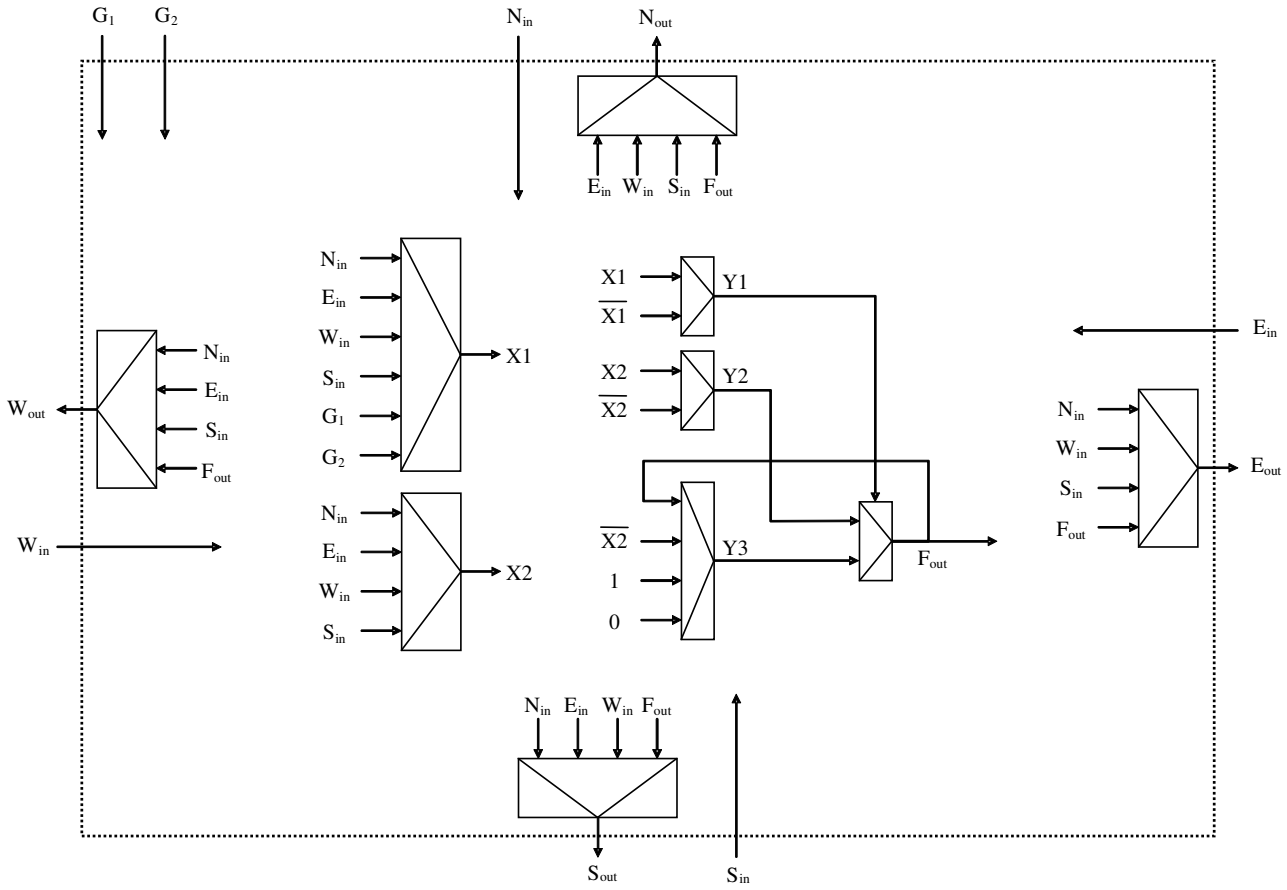


Figura 20 – Diagrama de blocos de uma célula do CAL1024.

Tabela 3 – Tabela de possíveis configurações de uma célula do CAL1024.

Número	Função	Número	Função
1	zero	11	$\overline{X1} \cdot X2$
2	um	12	$\overline{X1} \cdot \overline{X2}$
3	X1	13	$X1 + X2$
4	$\overline{X1}$	14	$X1 + \overline{X2}$
5	X2	15	$\overline{X1} + X2$
6	$\overline{X2}$	16	$\overline{X1 \oplus X2}$
7	$X1 \cdot X2$	17	$\overline{X1} + \overline{X2}$
8	$X1 \cdot \overline{X2}$	18	$X1 \oplus X2$
9	D Clk Latch	19	$\overline{D}$ Clk Latch
10	D $\overline{Clk}$ Latch	20	$\overline{D}$ $\overline{Clk}$ Latch

A memória de configuração do CAL1024 possui 18k bits, sendo 18 bits para cada célula. Cada célula está organizada na memória de configuração em 2 linhas de 9 colunas, conforme ilustrado na Tabela 4. Novos valores podem ser escritos individualmente na memória de configuração, automaticamente reconfigurando parcialmente o dispositivo.

Tabela 4 – Organização de uma célula na memória de configuração do dispositivo CAL1024.

Coluna	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Linha 1	Mux North		X1 [0:1]		F_{out}	X1 [2]	Y1	Mux East	
Linha 0	Mux West		X2		Y2	Y3		Mux South	

2.2.2.2 ERA60100

O dispositivo ERA60100 da Plessey [PLE90] possui uma matriz de 50 por 50 células, sendo a célula deste dispositivo a mais simples de todos os FPGAs [HOW92]. A Figura 21 apresenta o diagrama de blocos da célula, podendo esta ser configurada como uma porta lógica NAND, como um inversor ou como um flip-flop.

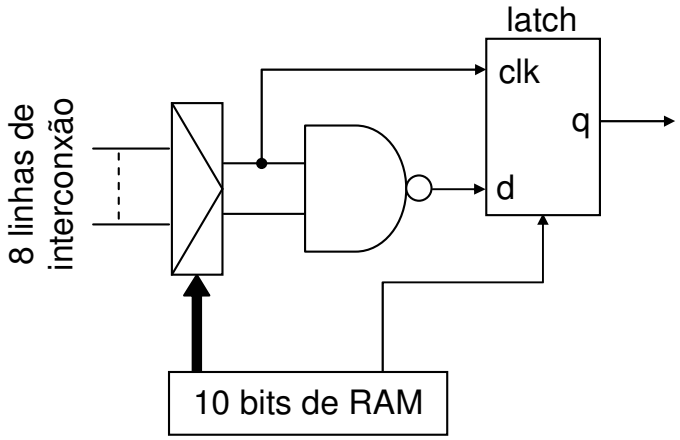


Figura 21 – Diagrama de blocos da célula do dispositivo ERA60100.

O ERA60100 possui quatro tipos de roteamento: local, curta distância, longa distância e periferia. O roteamento local conecta as células vizinhas mais próximas. Os roteamentos de curta e longa distância são barramentos verticais e horizontais que se estendem, respectivamente, por 10 células e pelo chip inteiro. O roteamento de periferia é um barramento de 10 bits que se estende junto com os pinos de E/S localizados na periferia do chip.

O ERA pode ser configurado a partir de 3 diferentes fontes: por um ROM, por outro FPGA ou por um dispositivo autônomo como, por exemplo, um processador. O dispositivo pode ser parcialmente reconfigurado por linha, permitindo que uma parte do dispositivo seja modificado enquanto o restante permanece funcionando. O tempo de reconfiguração é proporcional ao número de linhas a ser configurado. Uma configuração total pode ser efetuada em aproximadamente 350 μs.

2.2.2.3 XC6200

Em 1993 a Algotronix foi comprada pela Xilinx, tendo sido a tecnologia da Algotronix utilizada como base para a construção da família de dispositivos XC6200. Apenas os dispositivos XC6216 e XC6264 foram construídos desta família, possuindo, respectivamente, uma matriz de 64 por 64 e 128 por 128 células. A Figura 22 apresenta o diagrama de blocos de uma célula desta família de dispositivos e sua respectiva unidade funcional. Observar a semelhança entre as Figura 20 e Figura 22.

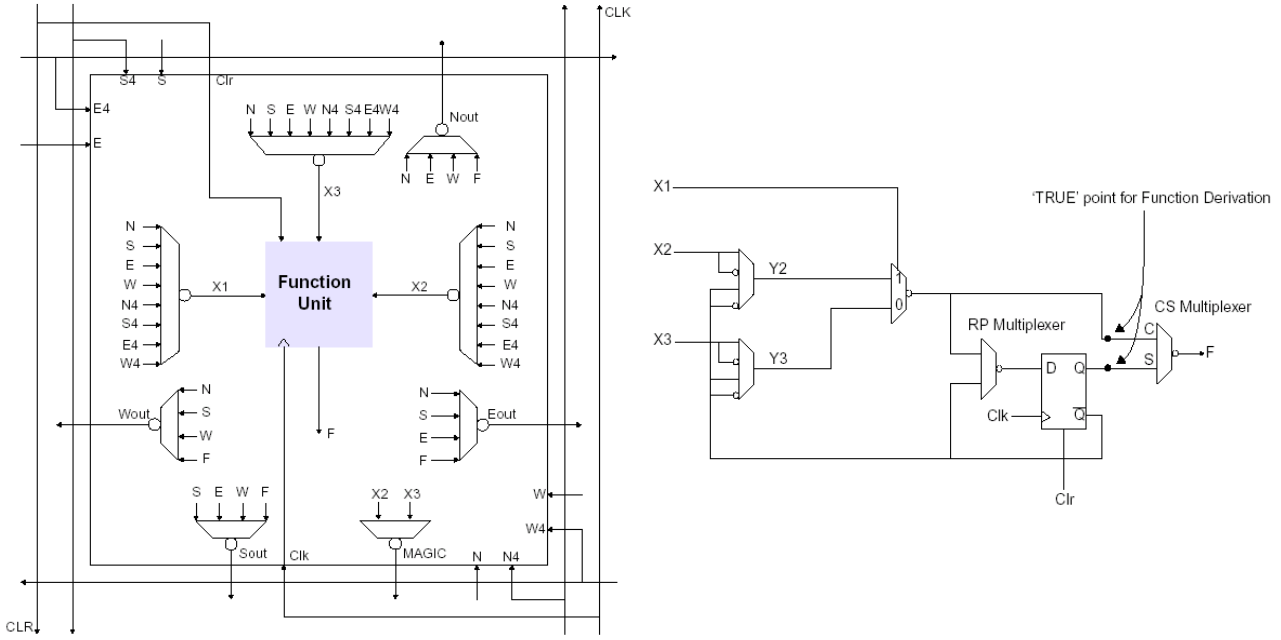


Figura 22 – Diagrama de blocos de uma célula do XC6200 e respectiva unidade funcional.

A célula recebe três entradas de vizinhos NEWS a uma célula de distância (N, E, W, S) e a quatro células de distância (N4, E4, W4, S4). Além destas entradas, a célula recebe as entradas globais CLR e CLK utilizadas pelo flip-flop D da unidade funcional. A célula possui como saída conexão direta com os quatro vizinhos NEWS e uma saída (MAGIC) para a conexão com células localizadas a uma distância maior ou igual a quatro células.

O tempo mínimo de leitura e escrita na memória de configuração da família de dispositivos XC6200 é, respectivamente, 30ns e 40ns. Cada célula está organizada na memória de configuração em três bytes, conforme ilustrado na Tabela 5.

Tabela 5 – Formato de configuração de uma célula da família de dispositivos XC6200.

Deslocamento da coluna	DATA BIT							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
00	North		East		West		South	
01	CS	X1 [2:0]			X2 [1:0]		X3 [1:0]	
10	M	RP	Y2 [1:0]		Y3 [1:0]		X3 [2]	X2 [2]

2.2.2.4 Virtex-II e Virtex-II Pro

A família de dispositivos Virtex da Xilinx foi a sucessora da família XC6200, tendo esta sido descontinuada quando a primeira foi lançada em 1998. A família Virtex evoluiu para a família Virtex-II em 2000 e para Virtex-II Pro em 2002. Devido às famílias Virtex-II e Virtex-II Pro terem herdado todas as características da Virtex, apenas estas serão contempladas no decorrer desta Seção. Para informações específicas do dispositivo Virtex o leitor é remetido a [MES02] e a [XIL01].

A Figura 23 apresenta os principais recursos dos dispositivos Virtex-II.

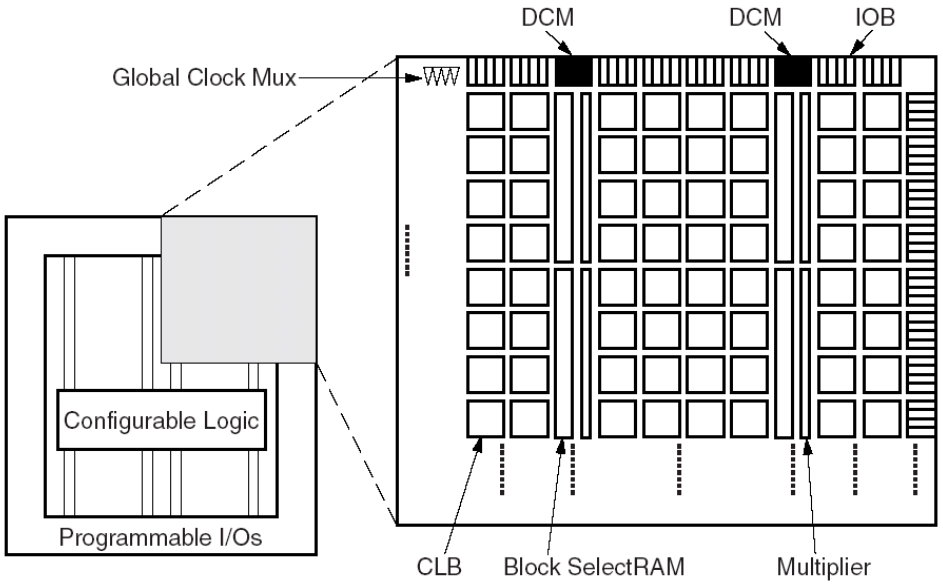


Figura 23 – Arquitetura interna do dispositivo Virtex-II.

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 digitais de relógio (DCMs – *Digital Clock Manager*), multiplicadores e recursos de roteamento que interconectam todos estes componentes. As CLBs provêem elementos funcionais para a implementação de lógica combinacional e/ou seqüencial. Cada CLB possui quatro *slices* e dois *tristates*. A Figura 24 apresenta a metade superior de uma CLB e compreende geradores de função de quatro entradas (Look-Up Tables - *LUTs*), flip-flops para armazenamento de um bit e recursos como propagação rápida de *carry*. Cada BRAM possui 18Kb de memória dupla porta, programável em diversas larguras de dados e endereços. A Virtex-II Pro possui todos os recursos dos dispositivos Virtex-II e mais dois processadores Power PC. A Tabela 6 apresenta a quantidade desses e outros recursos existentes em cada dispositivo das famílias Virtex-II e Virtex-II Pro.

A arquitetura de (re)configuração dos dispositivos Virtex é organizada 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 7

apresenta o número de bits de um *frame* dos dispositivos apresentados na Tabela 6, assim como o tempo de reconfiguração de um bitstream total para diversos modos de configuração.

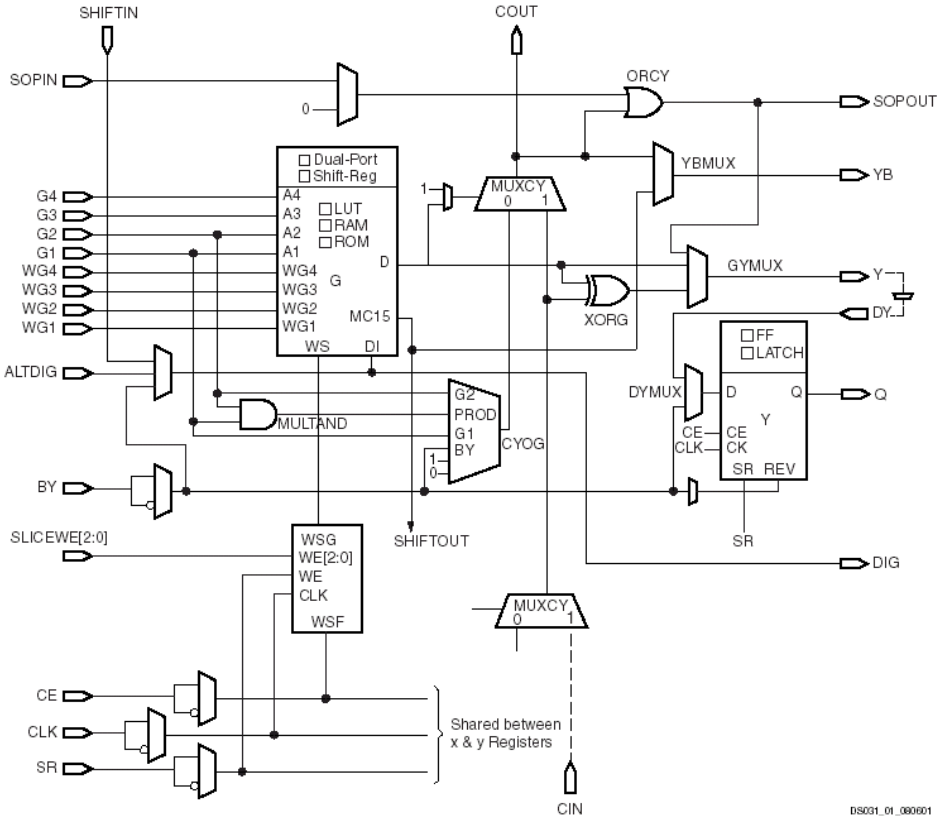


Figura 24 – Metade superior de um *slice* da Virtex-II.

Tabela 6 – Quantificação dos recursos internos de cada dispositivo das famílias Virtex-II e Virtex-II Pro.

	Dispositivo	Blocos de PowerPC	1 CLB = 4 slices = 128 bits RAM			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)		
Virtex-II Pro	XC2V40	0	8x8	256	8	4	4	72	4	88
	XC2V80	0	16x8	512	16	8	8	144	4	120
	XC2V250	0	24x16	1.536	48	24	24	432	8	200
	XC2V500	0	32x24	3.072	96	32	32	576	8	264
	XC2V1000	0	40x32	5.120	160	40	40	720	8	432
	XC2V1500	0	48x40	7.680	240	48	48	864	8	528
	XC2V2000	0	56x48	10.752	336	56	56	1.008	8	624
	XC2V3000	0	64x56	14.336	448	96	96	1.728	12	720
	XC2V4000	0	80x72	23.040	720	120	120	2.160	12	912
	XC2V6000	0	96x88	33.792	1.056	144	144	2.592	12	1.104
	XC2V8000	0	112x104	46.592	1.456	168	168	3.024	12	1.108
Virtex-II	XC2VP2	0	16x22	1.408	44	12	12	216	4	204
	XC2VP4	1	40x22	3.008	94	28	28	504	4	348
	XC2VP7	1	40x34	4.928	154	44	44	792	4	396
	XC2VP20	2	56x46	9.280	29	88	88	1.584	8	564
	XC2VP30	2	80x46	13.696	428	136	136	2.448	8	692
	XC2VP40	2	88x58	19.392	606	192	192	3.456	8	804
	XC2VP50	2	88x70	23.616	738	232	232	4.176	8	852
	XC2VP70	2	104x82	33.088	1.034	328	328	5.904	8	996
	XC2VP100	2	120x94	44.096	1.378	444	444	7.992	12	1.164
	XC2VP125	4	136x106	55.616	1.738	556	556	10.008	12	1.200

Tabela 7 – Informações para a reconfiguração de dispositivos Virtex-II e Virtex-II Pro.

	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
Virtex-II	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
Virtex-II Pro	XC2VP2	884	1472	1.301.248	1.305.376	3,26	26,11	39,56
	XC2VP4	884	3392	2.998.528	3.006.496	7,52	60,13	91,11
	XC2VP7	1320	3392	4.477.440	4.485.408	11,21	89,71	135,92
	XC2VP20	1756	4672	8.204.032	8.214.560	20,54	164,29	248,93
	XC2VP30	1756	6592	11.575.552	11.589.920	28,97	231,80	351,21
	XC2VP40	2192	7232	15.852.544	15.868.192	39,67	317,37	480,86
	XC2VP50	2628	7232	19.005.696	19.021.344	47,55	380,43	576,41
	XC2VP70	3064	8512	26.080.768	26.098.976	65,24	521,98	790,88
	XC2VP100	3500	9792	34.272.000	34.292.768	85,73	685,86	1039,18

Os dispositivos da família Virtex-II possuem uma porta interna de acesso de configuração denominada ICAP (*Internal Configuration Access Port*). Este componente, apresentado na Figura 25, 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]. A interface de comunicação do ICAP, apresentada na Tabela 8, é semelhante à interface SelectMAP, não possuindo os pinos utilizados exclusivamente para reconfiguração total (DONE, INIT e PROGRAM) e os pinos de modo de configuração (M0, M1 e M2).

Tabela 8 – Significado dos pinos de entrada e saída do ICAP.

Porta ICAP	Descrição
I [0:7]	Barramento de entrada de dados por onde é feita uma configuração. O pino I [0] 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 O [0] 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 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, quando BUSY estiver em nível lógico baixo.
CE	Habilita o ICAP. Este pino é ativo em nível lógico baixo, embora esta polaridade possa ser invertida através da ferramenta FPGA Editor.
WRITE	Indica se está sendo feita uma configuração ou um *readback* do FPGA. É ativado em nível lógico baixo, podendo ser invertido via FPGA Editor, assim como o sinal CE. Quando em nível lógico baixo, WRITE indica que um byte está sendo escrito através do barramento I [0:7]. Quando em nível lógico alto, WRITE indica que um byte está sendo lido através do barramento O [0:7].
CCLK	Sincroniza as leituras e escritas dos barramentos de configuração e *readback*.

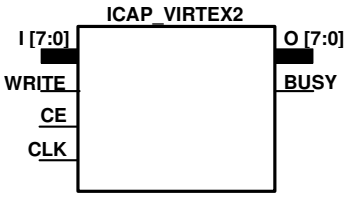


Figura 25 – Interfaces de entrada e saída do ICAP.

Outra característica das famílias de dispositivos Virtex da Xilinx é que eles são *glitchless*, permitindo que áreas que sobrescreverem os mesmos dados de configuração aos existentes não apresentem nenhum tipo de transitório. No entanto, caso sejam enviados dados de configuração diferentes dos previamente existentes no FPGA, *glitches* podem ser observados em sinais que ligam a área que está sendo reconfigurada ao restante da lógica que está fixa no dispositivo.

2.3 Sistemas Reconfiguráveis

Esta Seção apresenta sistemas parcial e dinamicamente reconfiguráveis. Estes sistemas estão organizados a partir do meio de comunicação empregado, mostrando a evolução destes meios em SoCs.

Para esta Seção foram selecionados apenas sistemas que reconfiguram um módulo inteiro, sendo descartado aqueles que fazem a reconfiguração de "parâmetros", onde apenas alguns valores em memória são modificados para que um módulo opere com novos valores de entrada.

Um ponto em comum entre os trabalhos apresentados é que todos utilizam as famílias Virtex e Virtex-II da Xilinx. O motivo principal desta escolha em comum é que a Xilinx é a única empresa atualmente no mercado a comercializar dispositivos parcial e dinamicamente reconfiguráveis com alta densidade de porta lógicas, assim possibilitando a construção deste tipo de sistema.

As próximas Seções apresentam exemplos de sistemas reconfiguráveis, organizados conforme a Tabela 9.

Tabela 9 – Exemplos de sistemas reconfiguráveis apresentados na Seção 2.3.

Módulos Reconfiguráveis conectados ponto a ponto (2.3.1)	Módulos Reconfiguráveis conectados por barramento (2.3.2)	Módulos Reconfiguráveis conectados por NoC (0)
Dyer (2.3.1.1)	Palma (2.3.2.1) Huebner (2.3.2.2) Walder (2.3.2.3) R8NR (2.3.2.4)	Gecko (2.3.3.1)

2.3.1 Ponto a Ponto

Quando apenas um módulo de hardware deve ser conectado a outro, a conexão direta, também chamada de ponto a ponto, é normalmente empregada. Este tipo de interconexão permite

extrair o máximo de desempenho da comunicação, pois é construído especificamente para atender os requisitos dos dois módulos que estão se comunicando.

2.3.1.1 Dyer

O sistema de Dyer et al. [DYE02a] foi validado em uma plataforma XSV800 [XES00] que possui um dispositivo FPGA Virtex 800. A Figura 26 apresenta o diagrama de blocos do sistema de Dyer, que implementa no mesmo FPGA um processador LEON e um coprocessador reconfigurável. Foram desenvolvidos dois decodificadores de áudio como coprocessadores para o LEON, um PCM e um Intel/DVI compatível com o decodificador ADPCM. O módulo ADPCM executa a 50MHz e utiliza 430 *slices* (4,5% dos recursos da XCV800); o módulo PCM utiliza 35 *slices* (0,4% dos recursos da XCV800). Neste sistema um módulo Ethernet recebe pacotes UDP com áudio codificado, o processador LEON desempacota o áudio codificado e passa para o módulo reconfigurável decodificá-lo. O resultado é automaticamente ouvido pela saída de som da plataforma de prototipação.

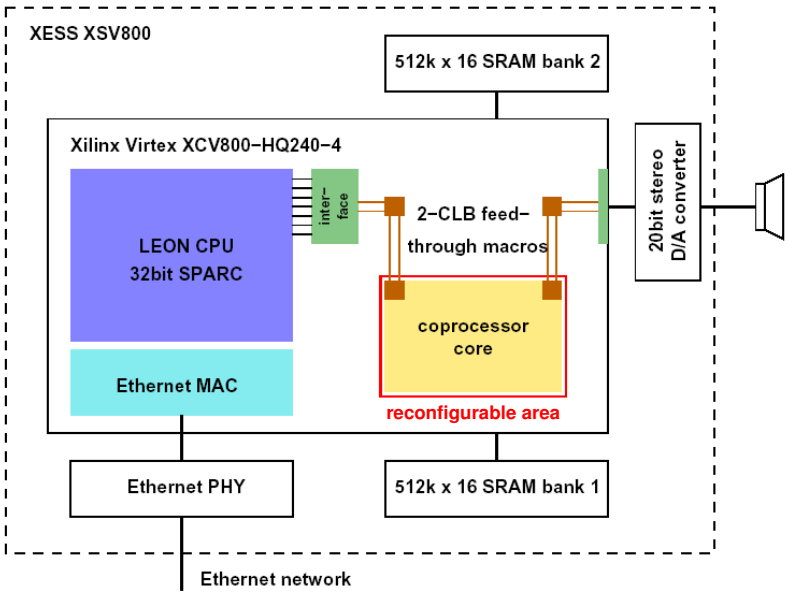


Figura 26 – Diagrama de blocos do sistema de Dyer [DYE02a].

O sistema proposto por Dyer é dito ponto a ponto porque define uma interface única entre o coprocessador e os módulos a ele conectado (Leon e D/A converter). Dois conjuntos de macros de passagem (feed-through) são utilizados. O primeiro conecta o Leon ao coprocessador e o segundo o coprocessador ao D/A converter. O sistema não é escalável, não permitindo agregar novos componentes ao sistema, como ocorre em barramentos ou NoCs.

2.3.2 Barramento

Barramento, conforme apresentado no Capítulo 1, é um meio de comunicação simples e que ocupa pouca área, sendo adequado para a comunicação de poucos módulos de hardware devido a este ser compartilhado por todos a ele conectado. Para sistemas reconfiguráveis

unidimensionalmente, como os dispositivos Virtex, barramentos são normalmente fixados horizontalmente na parte inferior ou superior do dispositivo, de forma a comunicar os módulos reconfiguráveis posicionados verticalmente.

2.3.2.1 Palma

O sistema de Palma et al. [PAL02a] foi implementado em uma plataforma VW300 [VCC04] que possui um dispositivo FPGA Virtex 300. A Figura 27 apresenta o diagrama de blocos do sistema de Palma, que implementa um barramento de dados de um bit, um árbitro do barramento, um módulo mestre e até três módulos escravos.

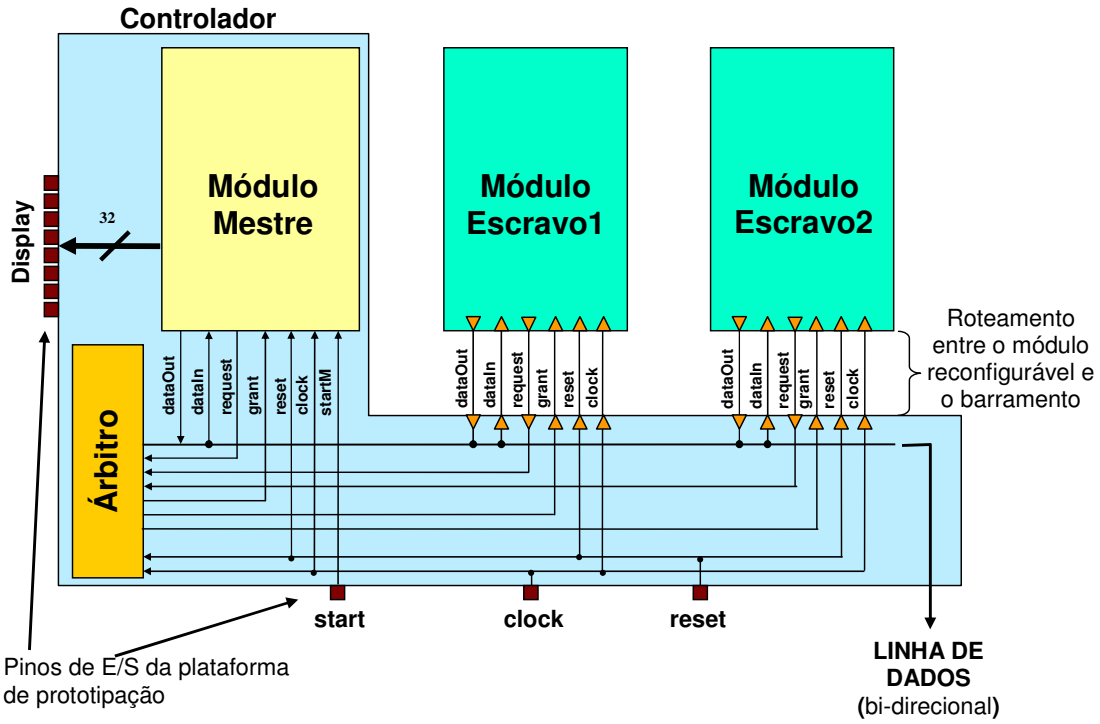


Figura 27 – Diagrama de blocos do sistema de Palma [PAL02b].

Neste sistema os módulos escravos (reconfiguráveis) executam operações aritméticas. O módulo mestre é responsável por fazer as solicitações de computação aos módulos escravos. Para que o módulo mestre possa fazer a transferência de dados pelo barramento para os módulos escravos, o módulo mestre deve antes requisitar acesso ao árbitro para que o mesmo possa utilizar o meio de comunicação compartilhado. Quando o acesso é liberado, o módulo mestre começa a enviar bit a bit os valores a serem calculados. Uma vez que o módulo escravo tenha efetuado a operação, o módulo escravo utiliza a mesma política de comunicação com o arbitro do barramento para enviar a resposta da operação para o módulo mestre.

Neste sistema o módulo mestre envia para o *display* os resultados das operações efetuadas e as solicitações de reconfiguração. Uma vez que um pedido de reconfiguração foi enviado para o *display* o sistema permanece parado até que o usuário faça a reconfiguração parcial manualmente do módulo escravo requisitado e aperte no botão de *start* da plataforma de prototipação.

2.3.2.2 Huebner

O sistema de Huebner et al. [HUB04b] foi validado em um dispositivo Virtex-II 3000. A Figura 27 apresenta o diagrama de blocos do sistema de Huebner, que implementa um barramento, um árbitro do barramento, quatro módulos reconfiguráveis, um módulo de controle de reconfiguração, uma unidade de descompactação de bitstreams e um módulo de acesso a auto-reconfiguração (ICAP).

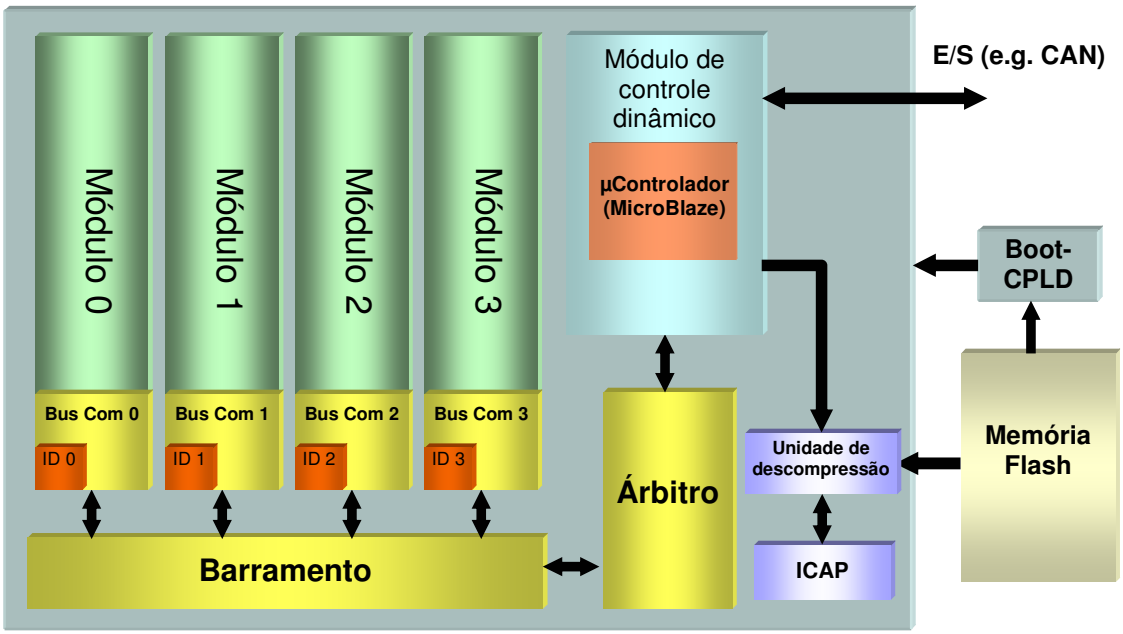


Figura 28 – Diagrama de blocos do sistema de Huebner [HUB04b].

Neste sistema o microcontrolador é responsável por gerenciar os módulos reconfiguráveis durante a execução do sistema. Quando o microcontrolador necessita de um módulo reconfigurável que não se encontra configurado, este envia uma requisição à unidade de descompactação, que por sua vez inicia a busca do bitstream específico na memória flash e envia para o módulo ICAP.

Não foram divulgados quais foram os módulos reconfiguráveis utilizados para este sistema, apenas que as aplicações alvo são multimídia e automotivas.

2.3.2.3 Walder

O sistema de Walder et al. [WAL04] foi validado em um dispositivo Virtex-II 3000. A Figura 29 apresenta o diagrama de blocos do sistema de Walder, que implementa um barramento na parte superior do dispositivo e interliga diversas tarefas ao sistema operacional, distribuído a esquerda e a direita do dispositivo. Quando o sistema é iniciado, apenas tarefas do sistema operacional e tarefas que implementam o barramento (tarefa nula) são carregadas, conforme apresentado a esquerda da Figura 29. À medida que novas tarefas são necessárias, reconfigurações parciais são efetuadas com tarefas do usuário. A direita da Figura 29, duas tarefas T1 e T2, de diferentes tamanhos, são apresentadas. Note que as tarefas devem ser múltiplas de um tamanho determinado como mínimo de uma tarefa para que o barramento não seja desconectado.

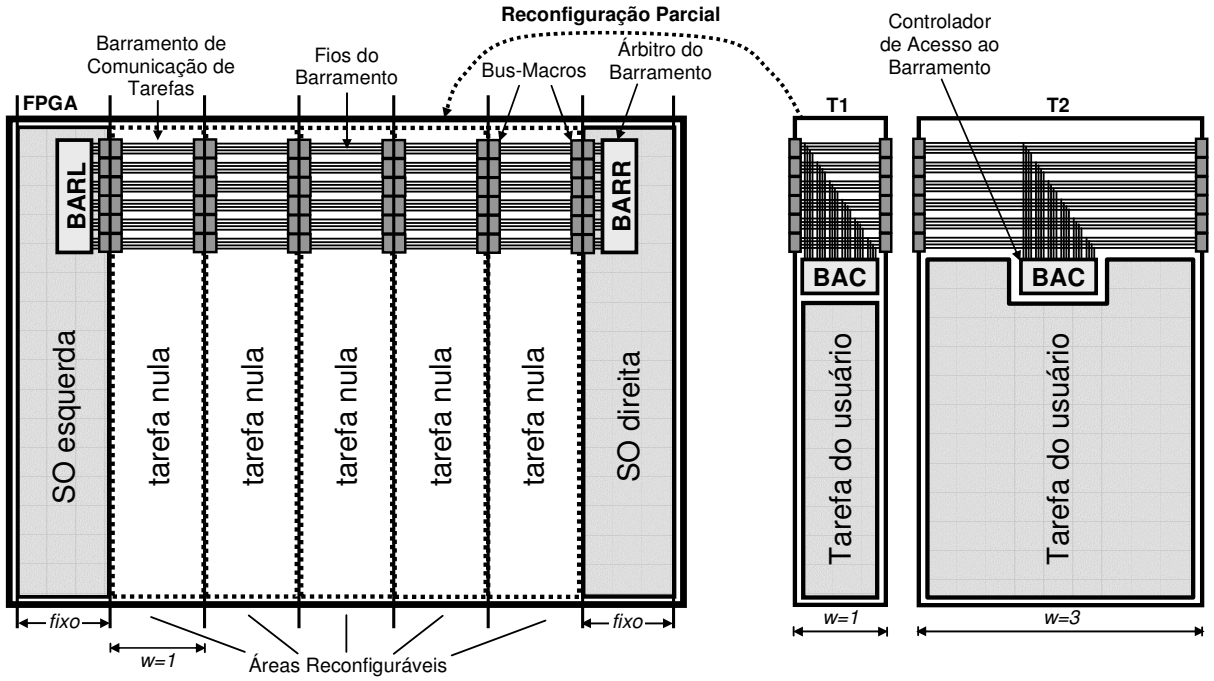


Figura 29 – Diagrama de blocos do sistema de Walder [WAL04].

Não foram divulgados quais foram os módulos reconfiguráveis utilizados para teste deste sistema.

2.3.2.4 R8NR

O sistema R8NR [MÖL04], que também pode ser caracterizado como coprocessador reconfigurável, foi validado em uma plataforma V2MB1000 [MEM02] que possui um dispositivo Virtex-II 1000. A Figura 30 apresenta o diagrama de blocos do sistema R82R com duas áreas reconfiguráveis. O sistema é composto por três módulos: (*i*) um computador hospedeiro, provendo uma interface para o sistema do usuário; (*ii*) uma memória de configurações, contendo todos os bitstreams parciais a serem utilizados durante a execução do sistema; (*iii*) um FPGA, contendo áreas fixas e reconfiguráveis do sistema R82R. A área fixa do FPGA é um sistema computacional completo, incluindo o processador R8R, a sua memória local contendo instruções e dados, um barramento do sistema controlado por um árbitro e periféricos (interface serial e controlador de configurações). A interface serial provê comunicação com o computador hospedeiro através de uma interface RS-232 padrão. O controlador de configurações (CC) é um periférico desenvolvido para agir como um módulo escravo do processador R8R ou do computador hospedeiro.

Foram desenvolvidas três instruções aritméticas como coprocessadores para o processador R8R: divisão, multiplicação e raiz quadrada. O principal objetivo deste sistema foi avaliar quando uma determinada implementação em hardware começa a ser vantajosa com relação a uma implementação em software, baseada no número de vezes que este hardware é empregado antes de ser reconfigurado. Os módulos multiplicador, divisão e raiz quadrada começam a ser vantajosos em relação à implementação em software a partir de 750, 260 e 200 execuções, respectivamente.

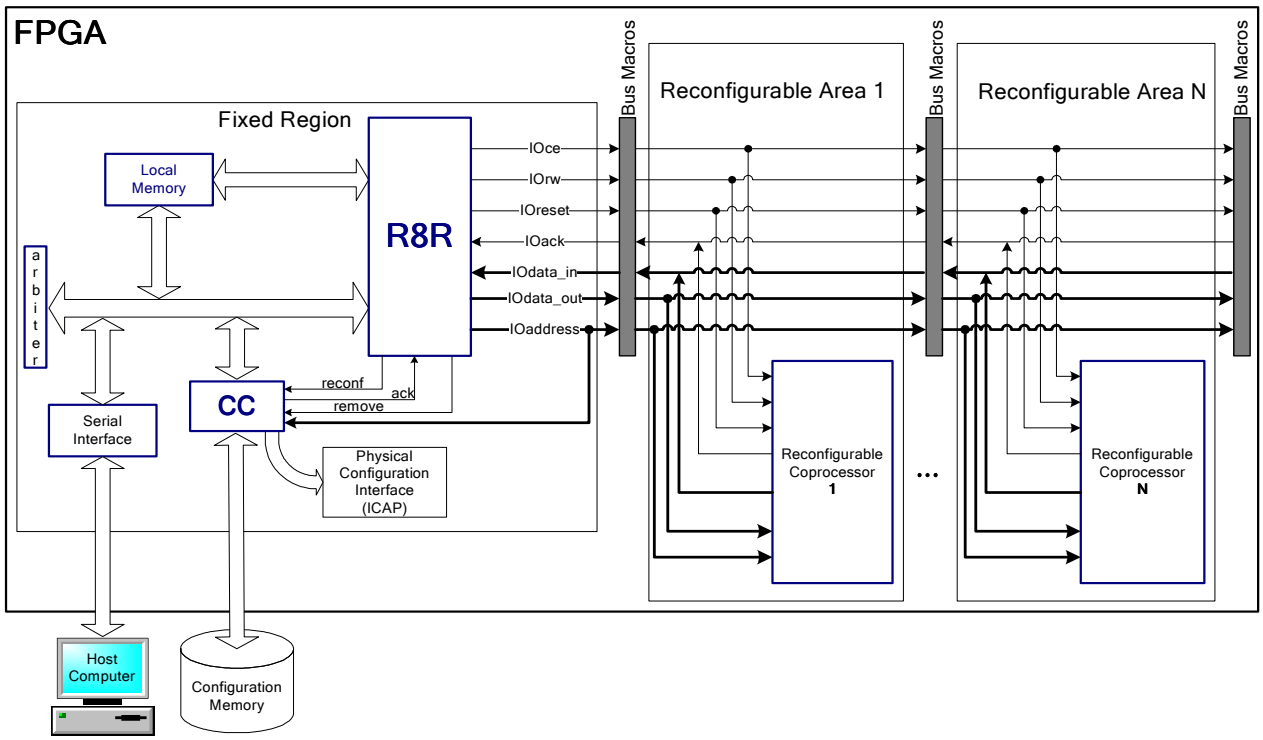


Figura 30 – Diagrama de blocos do sistema R82R [MÖL04].

2.3.3 NoC

Para interconectar diversos módulos de hardware que demandam altas taxas de transferência de dados, barramentos podem não prover a qualidade de serviço desejada. Neste contexto, NoCs parecem ser uma boa solução para a interconexão destes módulos de hardware. A seguir é apresentado o sistema Gecko, sendo este o único exemplo de sistema encontrado na literatura até o presente momento a utilizar módulos dinamicamente reconfiguráveis conectados a uma NoC. Embora a DyNoC [BOB05] também permita a reconfiguração dinâmica de módulos, nesta a reconfiguração é disruptiva, pois interrompe a execução de outros módulos e roteadores posicionados na mesma coluna. Outro problema da DyNoC é que esta não trata os pacotes existentes nos roteadores antes de reconfigurá-los.

2.3.3.1 Gecko

O sistema Gecko [MAR04], apresentado na Figura 31, é composto por um computador de mão iPAQ 3760 com processador StrongArm (206MHz) e um FPGA Virtex-II 6000. O sistema operacional OS4RS, executado sobre o iPAQ, controla as tarefas de hardware executadas no FPGA. O FPGA implementa uma NoC 3x3 com topologia malha como meio de comunicação dos seguintes módulos de hardware: cinco processadores RISC de 16 bits, um bloco 2D IDCT, uma interface para uma CPU externa e dois processadores reconfiguráveis de grão fino.

A NoC do Gecko é composta de três redes, conforme ilustrado na Figura 32: uma rede de dados, uma rede de controle e uma rede de reconfiguração. Estas três redes distintas foram criadas

porque cada rede possui propósitos e necessidades diferentes, permitindo assim tratar cada uma de forma eficiente.

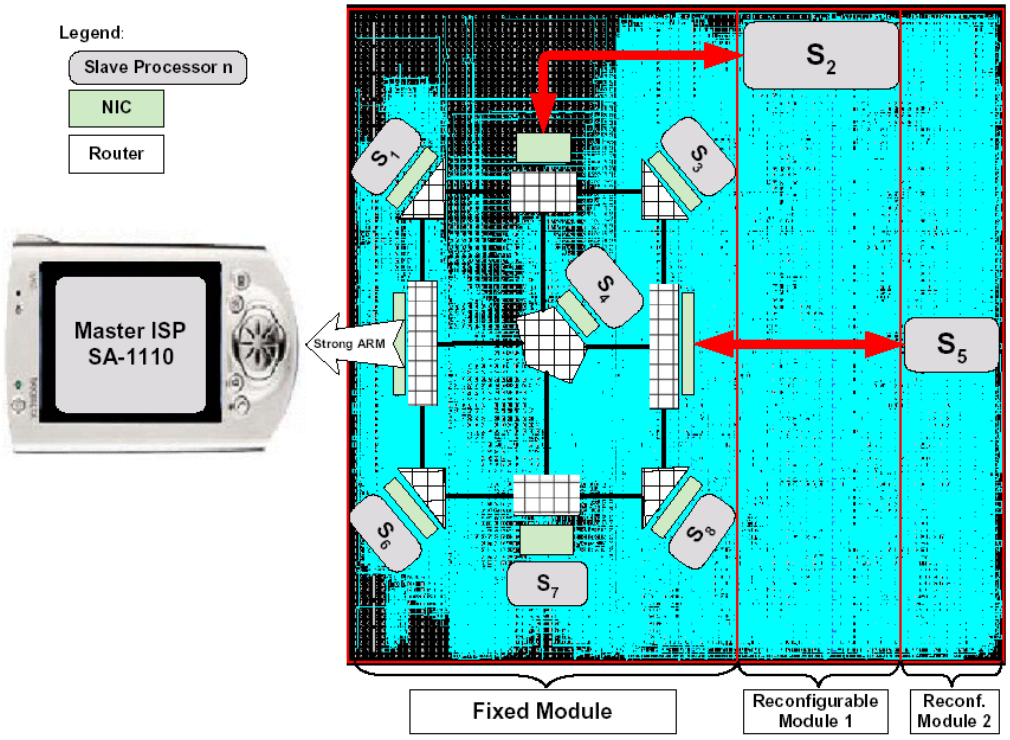


Figura 31 – Planta baixa e diagrama de blocos do sistema Gecko.

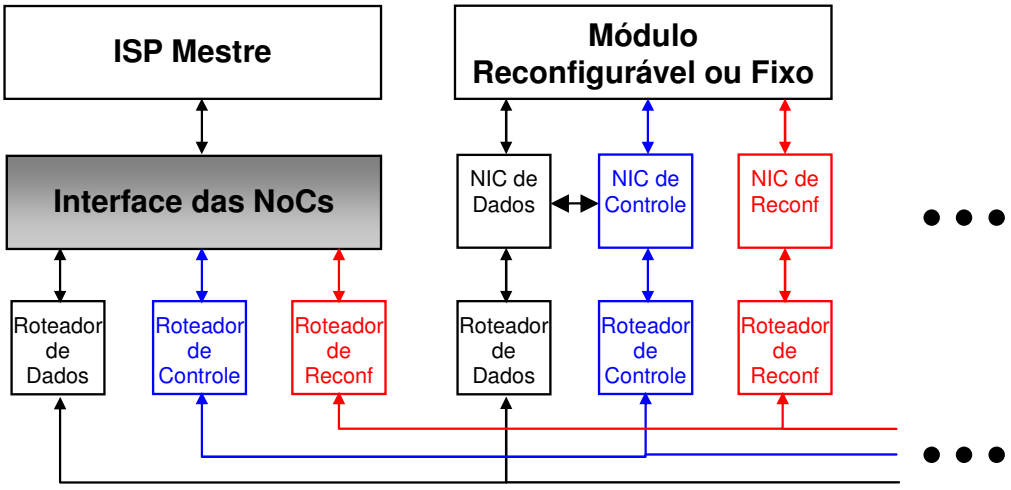


Figura 32 – Interface entre os módulos reconfiguráveis e o meio de comunicação do Gecko.

Um exemplo de aplicação executado sobre o Gecko é ilustrado na Figura 33. Em primeiro momento, um vídeo está sendo executado no PDA (1), utilizando todos os recursos de hardware disponíveis. A seguir, iniciam-se os comerciais e o usuário minimiza a janela do filme para executar outro aplicativo em primeiro plano (2). Quando isso ocorre, o número de recursos alocados ao vídeo é reduzido, pois não é necessário executar o vídeo em alta definição e taxa de amostragem. Ao executar, por exemplo, um jogo em primeiro plano (3), este pode utilizar os recursos de hardware previamente liberados.

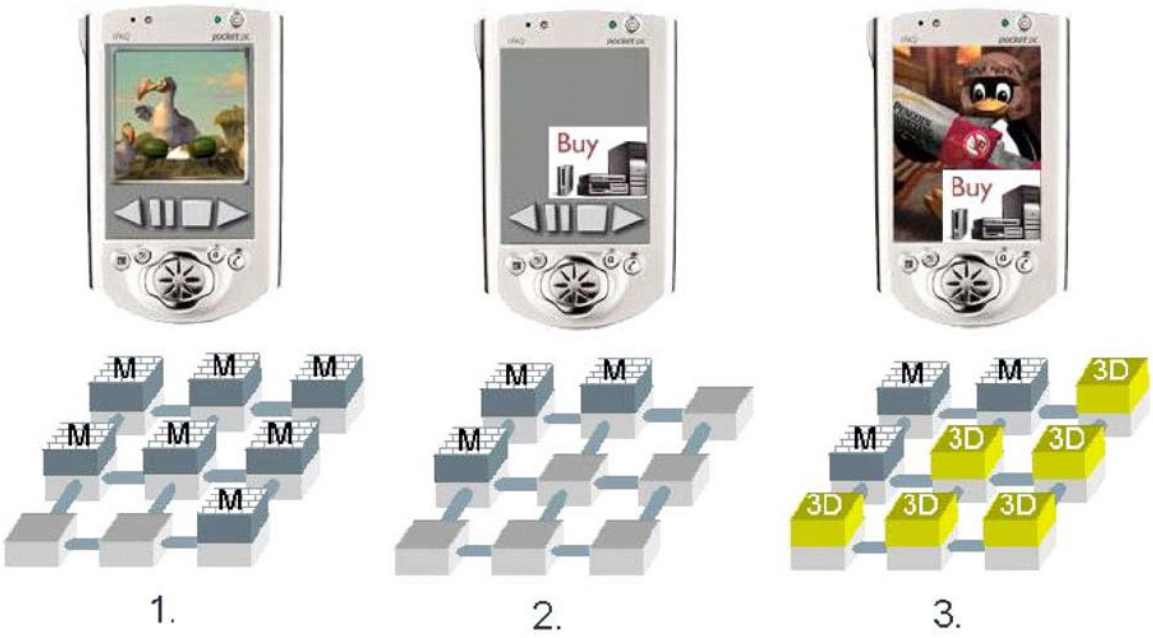


Figura 33 – Exemplo de aplicativo executado sobre o Gecko.

2.4 Conclusões

Este Capítulo apresentou diferentes processadores, dispositivos e sistemas reconfiguráveis, enfatizando-se sua evolução ao longo do tempo. Observa-se nitidamente três gerações nesta evolução: (*i*) sistemas com múltiplos FPGAs; (*ii*) dispositivos reconfiguráveis de grão grande; (*iii*) sistemas reconfiguráveis de grão grande prototipados sobre dispositivos reconfiguráveis de grão fino com elevado número de portas lógicas equivalentes.

O objetivo perseguido em sistemas reconfiguráveis desde 1985 é adicionar ao hardware a flexibilidade alcançada no software. A primeira geração era penalizada pelo baixo desempenho do meio de comunicação e baixa densidade dos dispositivos reconfiguráveis. A segunda geração era penalizada pela pouca flexibilidade do grão, pois o mesmo é implementado na forma de ULAs ou processadores, os quais realizam um conjunto pequeno de operações. A terceira geração provê flexibilidade, pois permite que qualquer função seja implementada nos módulos reconfiguráveis, com razoável desempenho no meio de comunicação.

O sistema Gecko é semelhante à arquitetura proposta na presente Dissertação, pois utiliza NoC como meio de comunicação, provendo ao usuário as características de escalabilidade e elevado desempenho no meio de comunicação.

O dispositivo de grão fino utilizado nos sistemas da terceira geração (Virtex) é adequado para arquiteturas que utilizam barramentos, dada a natureza unidimensional do processo de configuração. Sistemas baseados em NoCs têm uma natureza bidimensional no posicionamento dos módulos, o que requereria dispositivos que permitissem a reconfiguração individual de blocos lógicos, como ocorria em CAL1024 (Seção 2.2.2.1), ERA60100 (Seção 2.2.2.2) e XC6200 (Seção 2.2.2.3).

3 FLUXOS DE PROJETO PARA SDRS

O Capítulo 2 apresentou a evolução de diversos tipos de sistemas reconfiguráveis nas últimas duas décadas, convergindo nos últimos anos para FPGAs comerciais que possibilitam reconfigurar parcial e dinamicamente módulos de hardware do sistema. Dentre estes pode se observar que os dispositivos Xilinx são os que possuem maior densidade de portas lógicas equivalente e documentação relacionada à reconfiguração parcial e dinâmica [DYE02a] [PAL02a] [HUB04a] [WAL04] [MÖL04] [MAR04].

Embora os dispositivos Xilinx possuam a característica de ser parcial e dinamicamente reconfiguráveis e existam sistemas com esta característica, ainda não há um fluxo de projeto específico e adequado para a implementação destes sistemas, conforme relatado na revista Xcell da própria fabricante [BUT04]. Exemplos de fluxos de projeto utilizados para a implementação dos sistemas parcial e dinamicamente reconfiguráveis citados no Capítulo 2 são detalhados neste Capítulo, ressaltando as características de cada um a partir dos dados disponibilizados pelos autores dos sistemas.

A Seção 3.1 apresenta uma breve introdução sobre o fluxo de projeto padrão para FPGAs Xilinx, o qual deve ser conhecido antes de inserir características de reconfiguração parcial e dinâmica sobre o mesmo. A seguir, um fluxo de projeto genérico para a construção de SDRs é apresentado na Seção 3.2, apontando os principais passos a serem executados e problemas a serem resolvidos por fluxos reais. A Seção 3.3 e 3.4 apresentam, respectivamente, os fluxos utilizados pelos sistemas de Dyer (Seção 2.3.1.1) e Palma (Seção 2.3.2.1). A Seção 3.5 apresenta o fluxo de Projeto Modular, utilizado pelos sistemas de Walder (Seção 2.3.2.3), R8NR (Seção 2.3.2.4), Gecko (Seção 2.3.3.1) e outros. O fluxo de Horta é apresentado na Seção 3.6. Por fim, a Seção 3.7 apresenta o fluxo de projeto utilizado pelo sistema de Huebner (Seção 2.3.2.2).

3.1 Fluxo de Projeto Padrão para FPGAs Xilinx

Esta Seção apresenta uma introdução ao fluxo utilizado na implementação de projetos de hardware para FPGAs Xilinx 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 pela 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 aplicadas a SDRs.

O primeiro passo apresentado na Figura 34 é 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 aplicação de vetores de teste à entrada do mesmo. O passo seguinte é a síntese lógica do hardware. Algumas alternativas de 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 selecionáveis pelo usuário, como

maior controle do nível de esforço para redução de 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 HDL com atraso de portas (em formato 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.

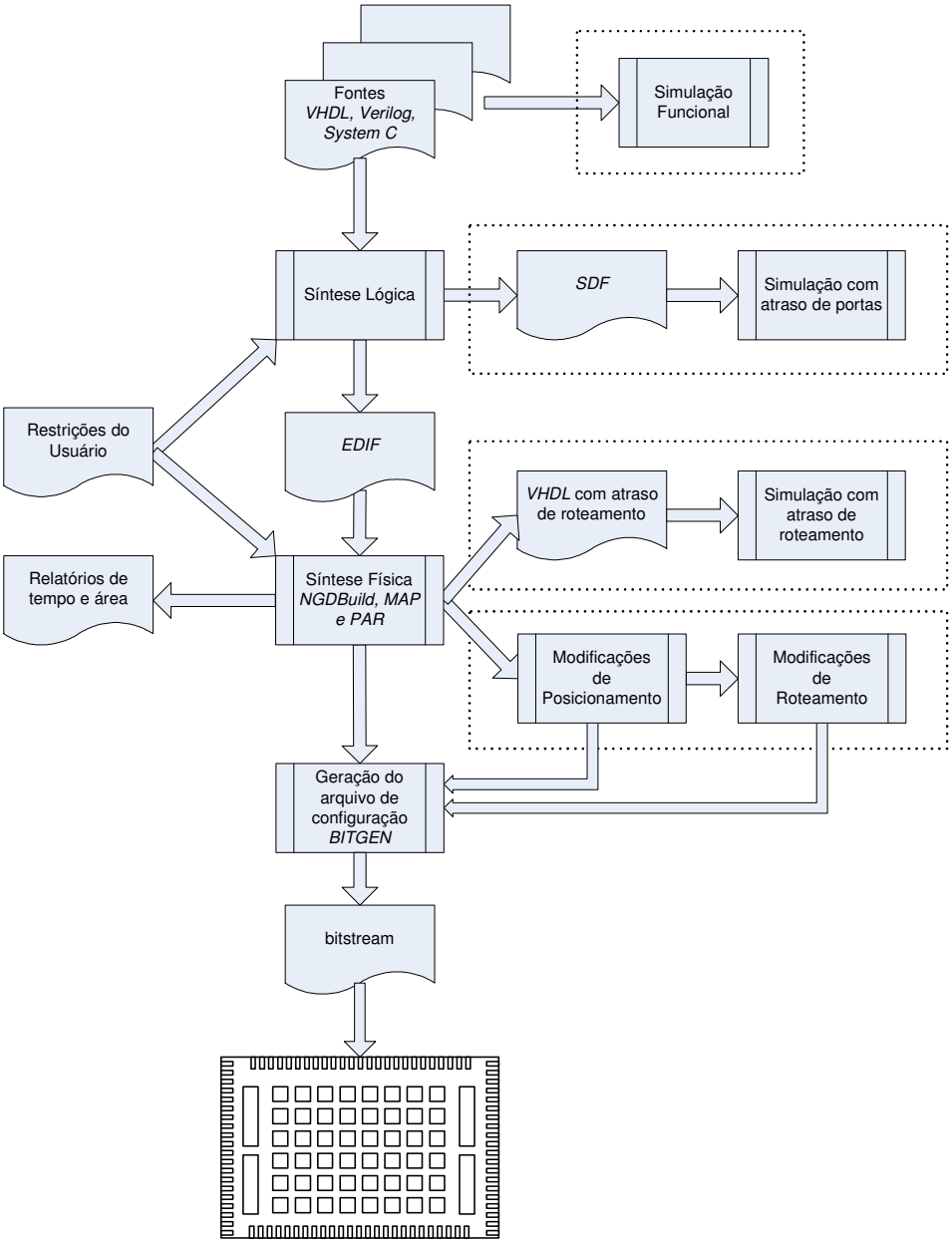


Figura 34 – Fluxo básico de projeto em FPGAs Xilinx, iniciando com o projeto em linguagem de descrição de hardware, e terminando com o bitstream gerado. Caixas pontilhadas representam passos opcionais do fluxo.

O arquivo EDIF, gerado pela ferramenta de síntese lógica, acompanhado do arquivo de restrições de recursos do usuário (UCF – *User Constraints File*), específico para o FPGA alvo, são as entradas para a fase de síntese física. A síntese física é responsável pelo mapeamento das portas

lógicas do 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, um bitstream total e uma série de relatórios são gerados. Entre os principais relatórios gerados, os mais importantes são os de velocidade e o de área, que informam os atrasos dos caminhos críticos do projeto e a ocupação de área deste no FPGA. O bitstream total do projeto é gerado pela ferramenta BitGen da Xilinx, podendo este ser configurado no FPGA a partir da execução da ferramenta Impact.

3.2 Fluxo de Projeto Genérico para SDRs

A Seção 3.1 versou sobre a execução do fluxo de projeto padrão para projetos de hardware sem características reconfiguráveis. As seções 3.2.1 a 3.2.6 apresentam um fluxo de projeto genérico para SDRs. Neste contexto faz-se antes necessário definir os termos *sistema* e *projeto*. Utilizar-se-á *sistema* como sendo o equivalente a SDR e possuindo diversos projetos. Cada *projeto* é um dos possíveis estados atuais de configuração do sistema.

3.2.1 Definir projeto inicial do SDR

O primeiro passo do fluxo de projeto é definir a estrutura geral do SDR, escolhendo a quantidade de áreas reconfiguráveis que farão parte deste e quais módulos estarão inicialmente configurados nestas áreas reconfiguráveis. É aconselhável prototipar este projeto usando configuração total, de forma a garantir que as restrições temporais deste sejam atendidas e que o mesmo opere em hardware. O motivo para esta prototipação inicial é garantir a operacionabilidade do ponto de partida do SDR, uma vez que serão posteriormente adicionados novos elementos, agregando complexidade ao todo.

3.2.2 Instanciar componentes de comunicação

O passo seguinte é a instanciação de componentes provedores de comunicação controlada entre os módulos reconfiguráveis e o restante do projeto. Estes componentes, também conhecidos como *macros* na literatura, fixam a posição das entradas e saídas entre cada um dos módulos reconfiguráveis instanciados em diferentes projetos do sistema.

3.2.3 Restringir posicionamento

Para que as macros, instanciadas na Seção 3.2.2, fixem corretamente as entradas e saídas de cada um dos módulos de cada um dos projetos do sistema, tanto os módulos quanto as macros correspondentes em cada projeto devem ser posicionados nas mesmas coordenadas do FPGA. Esta restrição de posicionamento, obrigatória para SDRs, pode ser feita com a ferramenta gráfica Floorplanner da Xilinx. O Floorplanner permite selecionar a posição destino dos componentes do projeto dentro do FPGA e definir o formato de alocação de lógica dos módulos no mesmo. O

Floorplanner gera como saída um arquivo de restrições do usuário (*User Constraints File* - UCF). Uma vez gerado o primeiro UCF de um projeto, este pode ser reutilizado para gerar novos projetos do mesmo SDR.

3.2.4 Verificar roteamento

Infelizmente, ao restringir a lógica de um módulo a uma determinada posição do FPGA, não obrigatoriamente os fios que interconectam os componentes internos deste módulo estarão posicionados dentro dos limites definidos para a lógica. Com freqüência, os fios internos de um módulo invadem a área de outro. Caso um ou mais fios cruzem a área pré-definida, estes ficarão desconectados após a reconfiguração, podendo ocasionar o não funcionamento do projeto e até mesmo um dano físico no dispositivo. A ferramenta FPGA Editor da Xilinx permite visualizar os fios do sistema e rotear manualmente ou automaticamente conexões que apresentem erros.

3.2.5 Gerar bitstreams

O quinto passo para o fluxo de projeto genérico de SDRs é a geração de bitstreams. Um *bitstream total* é o resultado de uma síntese física para FPGA e serve para inicializar o sistema. Bitstream totais são gerados a partir da ferramenta BitGen da Xilinx. *Bitstreams parciais* são utilizados para atualizar um módulo de hardware em uma área específica do FPGA. A 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 e sua relação com a arquitetura de reconfiguração do dispositivo. Como os fabricantes não têm o interesse de divulgar detalhes sobre estas arquiteturas, apenas eles possuem informações suficientes para prover ferramentas capazes de configurar parcialmente os seus dispositivos. Outro motivo que inibe os fabricantes de dispositivos reconfiguráveis de fornecer mais informações de suas arquiteturas é que seja feita engenharia reversa de algum projeto, podendo assim colocar em risco a propriedade intelectual de produtos dos clientes.

3.2.6 Realocar módulos do sistema

Duas situações necessitam que um fluxo de projeto para SDR seja repetido. A primeira é para gerar outro módulo de hardware para uma mesma área reconfigurável. A segunda é para posicionar um mesmo módulo de hardware em outras áreas reconfiguráveis. É possível evitar a segunda situação se o mesmo bitstream puder ser posicionado em diferentes áreas reconfiguráveis. Este procedimento é chamado de realocação e permite que um módulo originalmente sintetizado para uma área possa ser movido para outra, sem re-síntese.

Realocação pode ser útil em pelo menos quatro diferentes situações: (*i*) quando existem muitas áreas reconfiguráveis na qual pode-se posicionar um módulo; (*ii*) quando é difícil restringir um módulo a uma área específica; (*iii*) quando um módulo possui restrições temporais difíceis de

serem alcançadas; (*iv*) quando dois módulos que se comunicam freqüentemente ou possuem uma grande quantidade de dados a ser transferida se encontram distantes. Além disso, a realocação de módulos também reduz os requisitos de memória para armazenar bitstreams parciais, diminuindo o custo do sistema.

3.3 Dyer

Segundo Dyer e Wirz [DYE02b], quatro fluxos para geração de SDRs podem ser utilizados levando em consideração a biblioteca de manipulação de bitstreams JBits [GUC99]:

- Fluxo 1: desenvolver todos os módulos com ferramentas de implementação padrão[3] e extrair os bitstreams parciais manualmente a partir dos bitstream totais.
- Fluxo 2: desenvolver todos os módulos com JBits.
- Fluxo 3: desenvolver o bitstream total com ferramentas de implementação padrão e aplicar posteriores modificações ao roteamento com JBits.
- Fluxo 4: desenvolver todos os módulos com ferramentas de implementação padrão e utilizar JBits para conectar os módulos.

JBits é um conjunto de classes Java que fornece uma API para manipular o arquivo de configuração da família Virtex, tendo sido originalmente criado para suportar reconfiguração dinâmica. Com este conjunto de classes é possível realizar modificações em um circuito sem a necessidade de se refazer o posicionamento e roteamento com as ferramentas padrão. O JBits pode ser utilizado como uma ferramenta autônoma ou como base para produzir outras ferramentas, incluindo aplicações de CAD tradicionais de posicionamento e roteamento. Pode-se citar como exemplos de classes JBits a leitura e escrita de valores de componentes internos como LUTs, CLBs, BRAMs e IOBs, o cálculo de CRC de um bitstream e a geração de bitstreams parciais ou totais com as modificações efetuadas.

As próximas Seções apresentam os quatro fluxos propostos por Dyer e Wirz.

3.3.1 Fluxo 1: ferramentas padrão

Sem utilizar JBits, a reconfiguração parcial consiste na manipulação manual de bitstreams. Os bitstreams totais são gerados por ferramentas de síntese e implementação padrão. Destes bitstreams, módulos são extraídos copiando apenas os *frames* destes em um novo bitstream parcial. Os *frames* correspondentes do FPGA são substituídos pelos novos do bitstream parcial através de reconfiguração dinâmica.

Uma restrição inerente a este fluxo é que o roteamento entre o módulo reconfigurável e o restante do sistema não é feito automaticamente. Isto requer que o módulo seja conectado ao restante do sistema através de macros, conforme apresentado na Seção 3.2.2, assegurando que cada módulo reconfigurável utilize os mesmos recursos de roteamento para as conexões.

[3] Ferramentas do próprio fabricante, como MAP, PAR e NGDBuild.

3.3.2 Fluxo 2: apenas JBits

Se todos os módulos forem desenvolvidos com JBits, é possível facilmente utilizar a reconfiguração dinâmica neste projeto. A qualquer momento o programa pode escrever modificações no projeto como um bitstream parcial e reconfigurar o dispositivo. JBits possui um roteador automático, o qual pode refazer o roteamento com novos módulos inseridos no sistema.

Uma restrição deste fluxo é que ele é aplicável apenas a projetos de pequeno porte, pois a ferramenta de roteamento é limitada, não havendo outras ferramentas para implementar máquinas de estado ou outros modelos de construção.

3.3.3 Fluxo 3: JBits para pequenas modificações

Um projeto inicial é criado com ferramentas de síntese e implementação padrão, configurando inicialmente o FPGA. A partir disso todas as modificações subseqüentes são aplicadas com JBits. As modificações não são capturadas de outro bitstream, mas criadas pelo JBits. Este fluxo é apropriado para projetos que necessitam de poucas modificações como alteração de parâmetros de um algoritmo ou mudança na conectividade de fios.

Uma dificuldade inerente a este fluxo é desenvolver o circuito inicial de forma que o JBits consiga encontrar os recursos específicos novamente. Por este motivo o projeto inicial necessita de um floorplanning que contenha além do posicionamento de cada módulo, o posicionamento das LUTs que são origem ou destino das conexões que serão alteradas. Outra restrição é quanto a temporização de sinais. A ferramenta de implementação utiliza um algoritmo baseado em atrasos para efetuar o roteamento. Se estas conexões são removidas pelo JBits a temporização não é mais garantida, a menos que o programa verifique explicitamente os atrasos das novas rotas.

3.3.4 Fluxo 4: JBits para unificar módulos

Este fluxo é semelhante ao Fluxo 1, com a diferença que o JBits é utilizado para extrair os módulos de bitstreams totais e unificar com outros bitstream para construir um novo bitstream parcial. Existem duas vantagens deste fluxo sobre o Fluxo 1: (*i*) o JBits não fica restrito a extrair *frames*, podendo este ler CLBs com todos os seus recursos e escrever como um bitstream parcial para o dispositivo; (*ii*) um módulo pode ser conectado à interface através do roteador do JBits.

Este fluxo permite utilizar dois métodos diferentes para realizar a reconfiguração dinâmica. O primeiro, apresentado na Figura 35, utiliza macros na interface do módulo reconfigurável com o restante do sistema. O segundo, apresentado na Figura 36, utiliza o roteador do JBits para fazer a interconexão entre o módulo reconfigurável e o restante do sistema. Uma restrição inerente a este segundo método é, assim como o Fluxo 3, que ao utilizar o roteador do JBits a temporização da interface entre o módulo reconfigurável e o restante do sistema não é garantida. Em contraponto, o primeiro método possui a desvantagem de necessitar a edição manual da macro para implementação da interface.

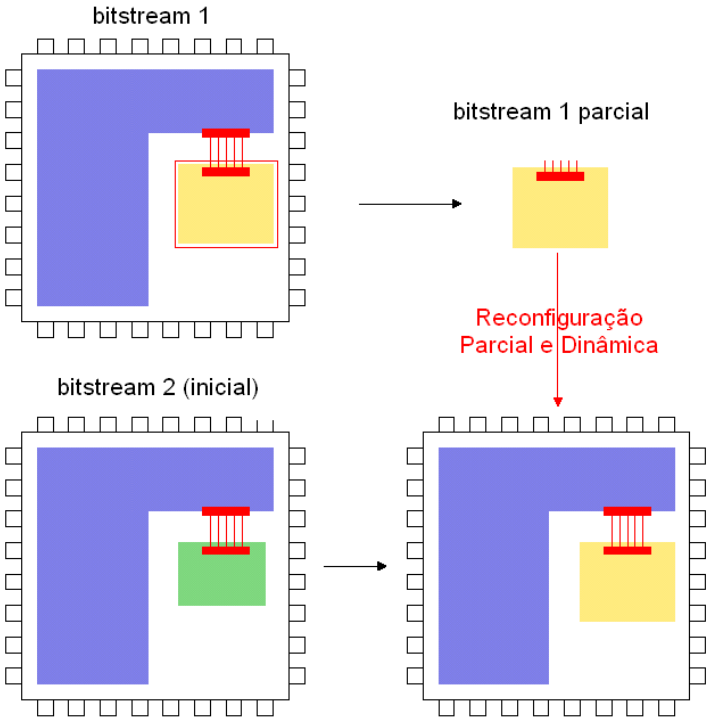


Figura 35 – Geração de bitstream parcial com roteamento estático.

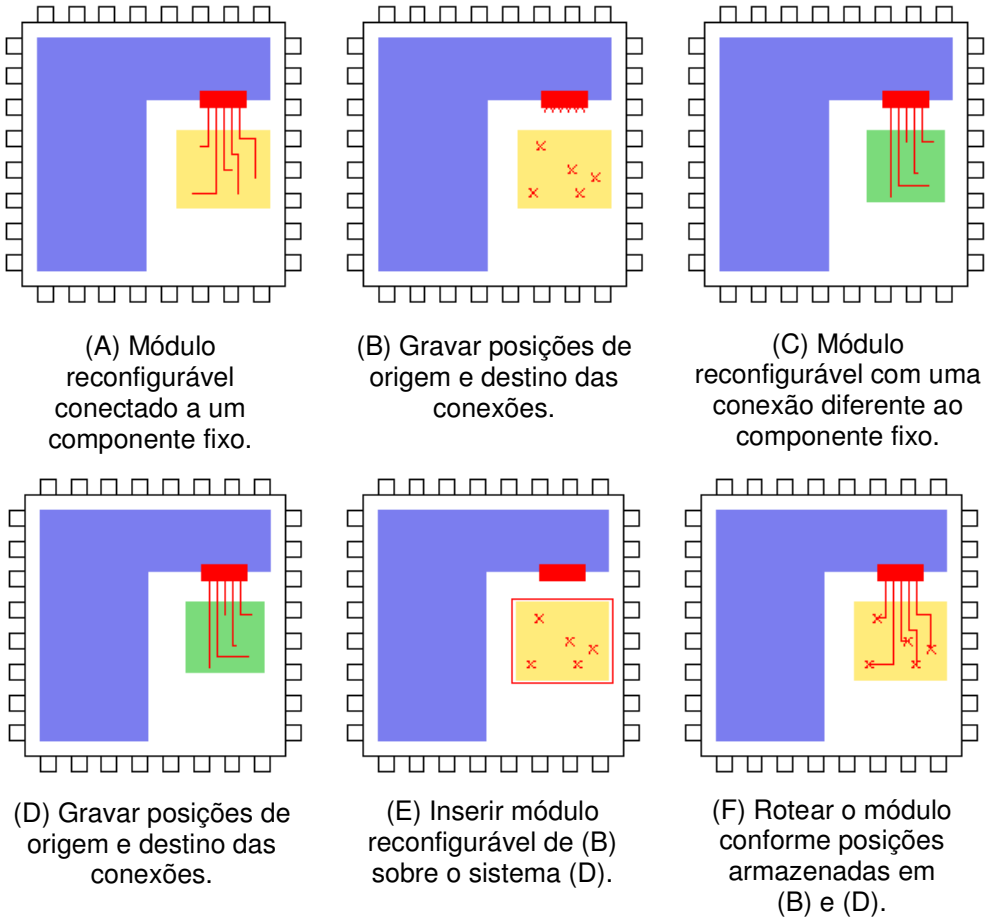


Figura 36 – Geração de bitstream parcial com roteamento automático através do JBits.

3.3.5 Fluxo utilizado por Dyer

Dentre os quatro fluxos apresentados por Dyer e Wirz, os dois métodos do Fluxo 4 foram utilizados para implementar o sistema apresentado na Seção 2.3.1.1. O método descrito na Figura 35 foi chamado de roteamento estático, enquanto o método descrito na Figura 36 foi chamado de roteamento automático. As Seções 3.3.5.1 a 3.3.5.3 apresentam os passos utilizados pelo Fluxo 4 ressaltando quando um passo é específico para um determinado método.

3.3.5.1 Posicionamento

Este passo é equivalente ao descrito na Seção 3.2.3, utilizando-se a ferramenta Floorplanner da Xilinx para fazer restrições de área de módulos e componentes. Para que seja possível fazer a restrição de posicionamento de módulos é necessário que seja mantida a hierarquia do projeto, sendo esta uma opção que deve ser habilitada na ferramenta de síntese. O Leon, a interface e o coprocessador tiveram a sua área restringida conforme apresentado na Figura 26 da página 34.

3.3.5.2 Roteamento

Conforme já mencionado, a restrição de posicionamento não necessariamente restringe as linhas de roteamento. Dyer e Wirz denominaram as linhas cruzando a área reconfigurável como "linhas perturbantes". As soluções de roteamento manual ou automático foram descartadas, devido ao tempo necessário ao usuário realizar um roteamento manual correto e às limitações da ferramenta JBits, respectivamente.

A solução adotada no fluxo é denominada "roteamento guiado", conforme ilustra a Figura 37. O princípio desta técnica é obrigar um fio a seguir um determinado caminho através da inserção de uma macro de passagem, denominada "CLB macro simples" e apresentada na Figura 38A.

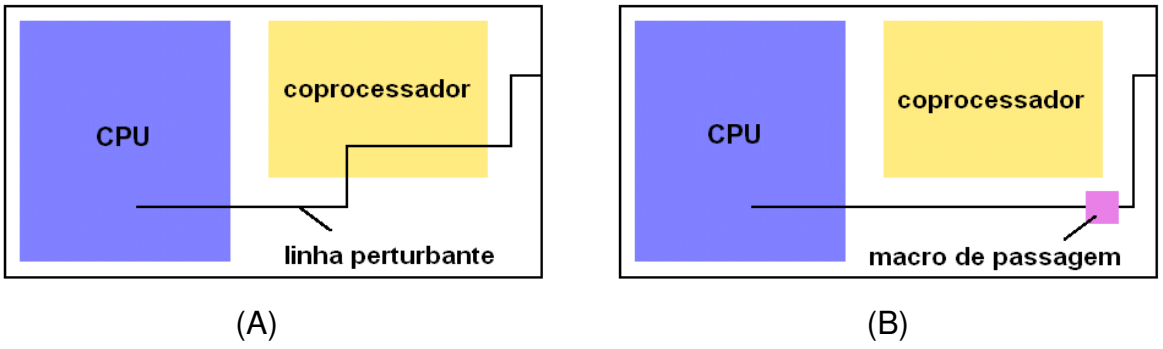


Figura 37 – Linha perturbante: (A) ilustração do problema de invasão de um módulo; (B) macro de passagem inserida para forçar o caminho da linha perturbante apresentada em (A).

A "CLB macro simples" nada mais é que todas as LUTs de uma CLB configuradas com a função identidade. Este componente pode ser restringido a uma posição específica a partir de restrições no arquivo UCF ou pelo Floorplanner. Para decidir onde posicionar a CLB macro simples ainda é necessário utilizar o FPGA Editor para capturar as coordenadas de onde este componente

pode ser posicionado de forma a evitar que o roteamento invada um módulo. A vantagem deste método é que isto só deve ser feito uma única vez e não a cada nova implementação como os métodos de roteamento manual e roteamento automático. A Figura 26 da página 34 apresenta onde as CLBs macro simples foram posicionadas no sistema de Dyer.

Figura 38 – Macros de Dyer implementadas para dispositivos Virtex: (A) CLB macro simples e (B) CLB macro dupla.

A macro denominada "CLB macro dupla", ilustrada na Figura 38B, é utilizada para fixar a interface do módulo reconfigurável com o restante do sistema, conforme apresentado na Seção 3.2.2. No sistema de Dyer diversas macros deste tipo estão posicionadas entre a interface do coprocessador reconfigurável e o Leon. Elas são formadas por duas CLBs interconectadas, sendo cada uma programada também como passagem de dados. A CLB macro dupla permite comunicar quatro bits e possui uma rota fixa entre duas CLBs.

3.3.5.3 Geração de bitstream

Para gerar um bitstream parcial pelo método de roteamento automático, o usuário pode escrever um programa em Java utilizando a API JBits, executando basicamente os seguintes passos: (*i*) abrir o bitstream total contendo o módulo origem; (*ii*) abrir o bitstream total contendo o módulo destino; (*iii*) habilitar o cálculo de CRC do bitstream destino; (*iv*) informar que nenhum *frame* foi modificado do bitstream destino; (*v*) encontrar e desconectar os fios que conectam o módulo reconfigurável ao módulo fixo no bitstream destino; (*vi*) armazenar os pontos de interface dos fios desroteados no módulo fixo; (*vii*) copiar o módulo origem; (*viii*) colar o módulo origem no bitstream destino; (*ix*) encontrar os pontos de interface do novo módulo reconfigurável no bitstream destino; (*x*) criar fios para conectar os pontos de interface do módulo origem ao módulo destino; (*xi*) rotear os fios criados, conectando os pontos de interface do módulo reconfigurável aos respectivos pontos de interface do módulo fixo; (*xii*) calcular o CRC do bitstream destino; (*xiii*) salvar o bitstream destino como um bitstream parcial.

A Tabela 10 apresenta algumas funções do JBits a serem utilizadas no processo de implementação de um programa em Java/JBits que permite criar bitstreams parciais pelo método de roteamento automático.

Tabela 10 – Algumas funções úteis do JBits para a manipulação de bitstreams.

Função	Descrição
`jbits.read(<BitstreamFile>);`	Abre um bitstream
`jbits.enableCRC(true);`	Habilita o cálculo de CRC
`jbits.clearDirty();`	Informa que nenhum *frame* foi modificado
`jbits.write(<BitstreamFile>);`	Gera um bitstream total
`jbits.writePartial(<PartialFile>);`	Gera um bitstream parcial
`rf=ResourceFactory.getResourceFactory(jbits);`	Captura todos os recursos associados a um bitstream
`rf.fillResourceFactory();`	Atualiza todos os recursos modificados para o bitstream
`net.addSource(<pin>);`	Adiciona uma origem a um fio
`net.addSink(<pin>);`	Adiciona um destino a um fio
`jbits.get(<row>,<col>,S1F1.S1F1);`	Captura um recurso de uma CLB
`Util.Compare(S1F1,S1F1.OFF)`	Descobre o valor de um componente de uma CLB
`jbits.set(<row>,<col>,LUT.SLICE0_F,nullLut);`	Escreve um valor em um recurso de uma CLB

3.4 Palma

O fluxo de projeto adotado em [PAL02b] foi implementado para permitir a reconfiguração parcial de módulos de hardware e busca minimizar a complexidade da interface de comunicação dos módulos do sistema. Tenta-se reduzir a complexidade da 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 verificar manualmente se os diferentes projetos utilizam a mesma trilha desde o módulo origem até o módulo destino. Este processo dificulta 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 criar SDRs com este fluxo foi utilizar *tristates* nas entradas e saídas dos módulos reconfiguráveis. Isto se deve ao fato que as trilhas de roteamento com *tristates* apresentam um roteamento fixo entre as CLBs vizinhas em dispositivos Virtex. As trilhas de roteamento com *tristates* dos dispositivos Virtex permitem construir um barramento fixo entre os diferentes projetos, 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 ilustrado na Figura 27 (pág. 35).

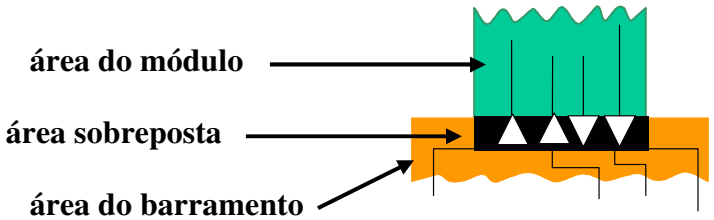


Figura 39 – Conexão do módulo com o barramento através de uma camada de *tristates*.

Duas camadas de *tristates* entre o barramento e o módulo reconfigurável são necessárias.

Isto se deve ao fato que um *tristate* pertence a uma CLB, a qual possui LUTs, *flip-flops* e roteamento. Caso apenas uma camada de *tristate* fosse utilizada, como na Figura 39, haveria sobreposição da área de um módulo sobre a lógica do barramento, assim interrompendo o funcionamento do sistema.

3.4.1 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.

3.4.1.1 Modularização do sistema

Esta etapa pressupõe que os módulos a serem inseridos no SDR já tenham sido simulados, apresentando um funcionamento correto. Isto é importante, pois esta etapa implicará em um aumento de complexidade do módulo. A tarefa a ser executada nesta etapa consiste em inserir os módulos de comunicação *Send* e *Receive*. Estes módulos são responsáveis por fazer a interface com o barramento que será inserido na Seção 3.4.1.2. A Figura 40 apresenta a conexão do módulo do usuário aos módulos *Send* e *Receive*, que por sua vez se conectam ao barramento.

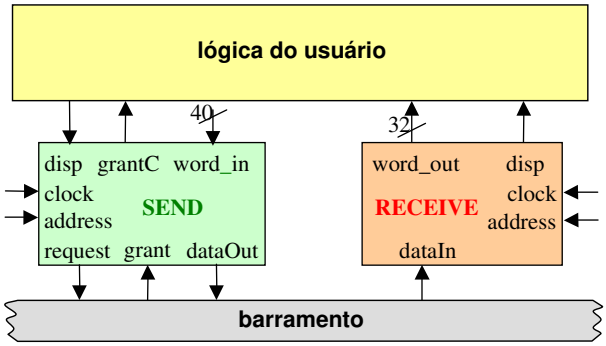


Figura 40 – Interface dos módulos Send e Receive com a lógica do usuário e com o barramento.

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 é liberado ao módulo, o sinal de *grant* é ativado e os bits de *word_in* são 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*.

3.4.1.2 Inserção do barramento

Nesta etapa é necessário instanciar o barramento serial de Palma no arquivo *top* do projeto

e tantos módulos quantos 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 *tristates* 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.

3.4.1.3 Criação de arquivos de restrição do usuário

Nesta etapa deve-se fazer a restrição dos módulos do sistema, conforme apresentado na Seção 3.2.3. 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 devem ocupar uma área específica do sistema para que uma posterior reconfiguração possa ocorrer.

3.4.1.4 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 41 apresenta à direita a interface entre o primeiro slot e o barramento. Ao centro da Figura 41 o caminho percorrido pelo sinal *dataIn* é ressaltado, desde sua origem até o seu destino. À esquerda da Figura 41 é feito um zoom sobre o destino do sinal *dataIn*, permitindo visualizar diversos outros caminhos que poderiam ter sido tomados pelo sinal.

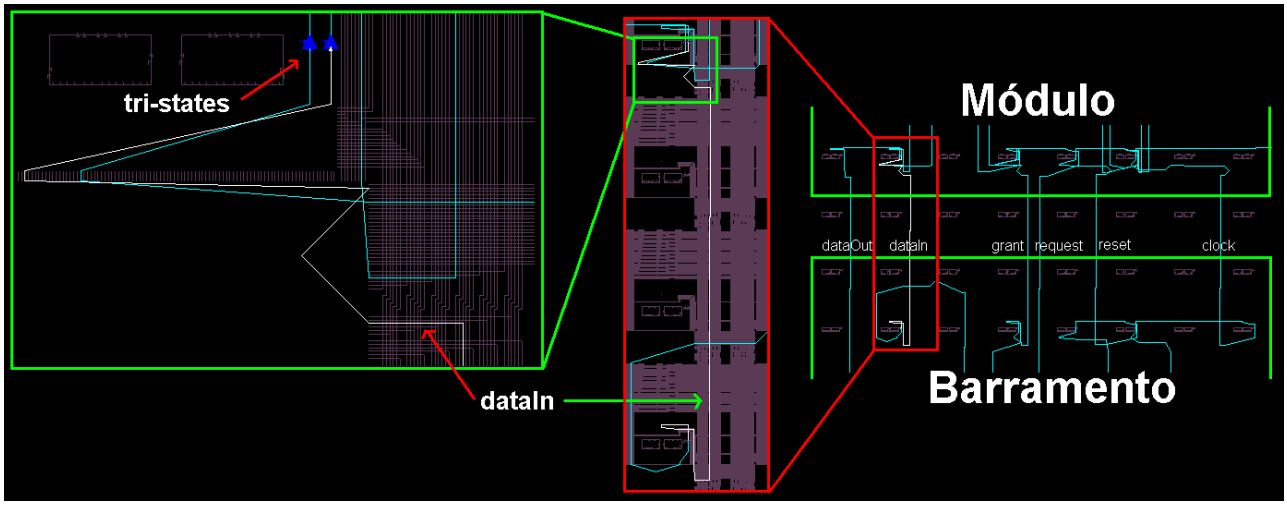


Figura 41 – À direita, interconexão de um módulo do SDR ao barramento; ao meio, ilustração do sinal dataIn; à esquerda, diversos outros caminhos que poderiam ser tomados pelo destino do sinal dataIn.

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.

3.4.1.5 Geração de bitstreams totais

Uma vez garantido o idêntico roteamento das interfaces de todos os projetos é possível utilizar a ferramenta BitGen 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.

3.4.1.6 Geração de bitstreams parciais

Utilizando a ferramenta CoreUnifier [MES03] 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, o qual é denominado *bitstream* base. A seguir devem ser abertos os outros *bitstreams* que possuem algum módulo diferente do 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 *tristates* 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.

3.4.2 Estudo de Caso – Calculadora Reconfigurável

A aplicação utilizada para testar o fluxo de projeto proposto por Palma é uma calculadora reconfigurável. O sistema é composto basicamente por três módulos, conforme apresentado pela Figura 27: controlador (mestre, árbitro e *tristates* 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.

É função do *árbitro* do sistema evitar que haja escritas simultâneas no 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 *tristates 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 3.4.1.1 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 realizarem uma dada função, retornam o resultado novamente para o módulo mestre.

O Anexo I apresenta a descrição VHDL do projeto *top* da calculadora reconfigurável. Após a síntese do *top* é possível utilizar o Floorplanner para restringir o posicionamento dos módulos e dos *tristates* 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 aos *tristates*. Cada módulo adicionado ao sistema demanda 12 *tristates* (6 do lado do barramento e 6 do lado do módulo). Ao posicionar estes *tristates*, também com a ferramenta Floorplanner, recomenda-se fixar os *tristates* 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 *tristates* 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 42.

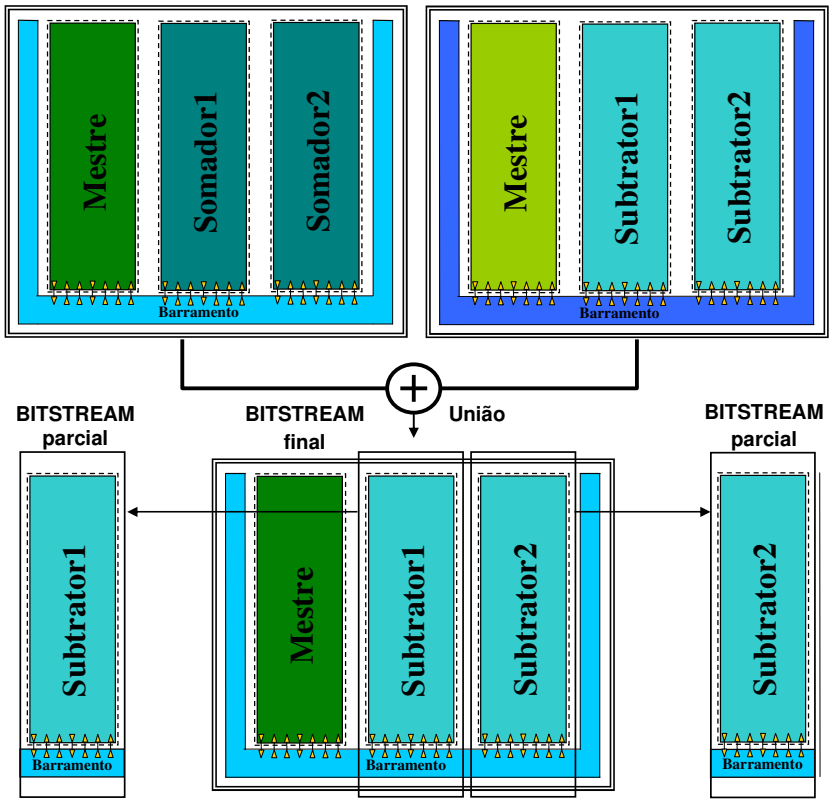


Figura 42 – Ilustração da geração de bitstreams efetuada pelo CoreUnifier. Dois bitstreams totais geram um novo total modificando apenas os módulos a serem substituídos. Posterior geração dos bitstreams parciais destes módulos.

De posse dos bitstreams totais com dois somadores e com dois subtratores, manualmente roteados, verificou-se o correto funcionamento destes em FPGA. Observado o correto funcionamento, utilizou-se a ferramenta CoreUnifier 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.

3.5 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 testar e validar os seus módulos sem que os módulos de outras equipes afetem os seus. Desta forma cada equipe gera um bitstream parcial do seu módulo e ao final do fluxo completo do Projeto Modular têm-se os arquivos de inicialização do sistema (bitstream total). Apesar de utilizado para sistemas reconfiguráveis, originalmente o fluxo de Projeto Modular não foi concebido para esta finalidade.

Antes de executar o fluxo de Projeto Modular em si, abordado na Seção 3.5.2, alguns passos devem ser efetuados para que os arquivos de entrada do fluxo sejam corretamente criados e posicionados em uma árvore de diretórios [LIM04]. Estes passos de preparação para execução do fluxo de Projeto Modular são apresentados na Seção 3.5.1.

3.5.1 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.

3.5.1.1 Modularização do sistema

Esta etapa pressupõe que o sistema, sem características reconfiguráveis, já foi prototipado pelo menos uma vez, conforme abordado na Seção 3.2.1, e que o sistema já está dividido em módulos.

3.5.1.2 Inserção de macros, LUTs e DCMs

Nesta etapa são adicionados os componentes isoladores, conforme abordado na Seção 3.2.2, e alguns componentes específicos do dispositivo no arquivo *top* do projeto. Exemplos destes componentes são as LUTs e os DCMs. DCMs são utilizados para evitar o escorregamento de relógio. LUTs são utilizadas para a geração de constantes lógicas. O motivo desta geração manual de constantes lógicas com LUTs é devido a estas poderem ter a sua posição restringida. Quando não

são feitas restrições de posicionamento, pode ocorrer de constantes serem buscadas de um componente que se encontra em outra área reconfigurável, assim causando uma desconexão do fio no momento da reconfiguração.

A Bus Macro [LIM04] [XIL04b], ilustrada na Figura 43, é o componente proposto pela Xilinx para interconectar módulos reconfiguráveis ao restante do sistema. A Bus Macro é formada por oito *tristates* e disponibiliza quatro fios de um bit para a comunicação do sistema com o módulo reconfigurável ou vice-versa.

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 *tristates* 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.

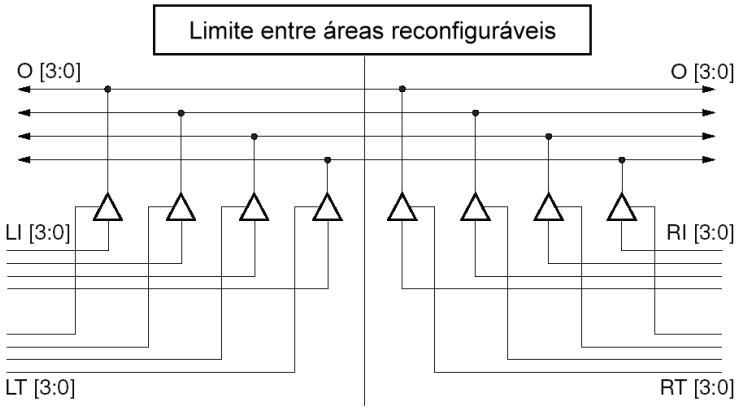


Figura 43 – Implementação física de uma Bus Macro.

3.5.1.3 Criação de arquivos de restrições do usuário

A criação de arquivos de restrições do usuário, abordado na Seção 3.2.3, pode ser feita manualmente, com a ferramenta Floorplanner ou com a ferramenta PACE. Estas ferramentas restringem a posição de módulos, macros e pinos de E/S através das linhas:

```
AREA_GROUP = "module_name" RANGE = SLICE_X0Y0:SLICE_X9Y55 , RAMB16_X0Y0:X0Y1 ;

INST "macro_name" LOC = "SLICE_X38Y68" ;

NET "pin_name" LOC = "A11" ;
```

onde:

- **`AREA_GROUP`** define que o módulo **`"module_name"`** deve ser posicionado entre as coordenadas de slices 0,0 e 9,55, e as coordenadas de blockRAMs 0,0 e 0,1;
- **`INST`** restringe a macro **`"macro_name"`** as coordenadas 38,68;
- **`NET`** restringe o pino **`"pin_name"`** ao pino de E/S **`"A11"`** do FPGA;

Este passo deve ser executado para todos os módulos, macros e pinos de E/S do projeto.

3.5.1.4 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.

3.5.1.5 Criação de scripts de síntese

Para facilitar a tarefa de síntese de cada um dos módulos do sistema é aconselhável escrever 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.

3.5.1.6 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.

3.5.2 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.

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 do módulo ativo 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 44 apresenta as fases do Projeto Modular.

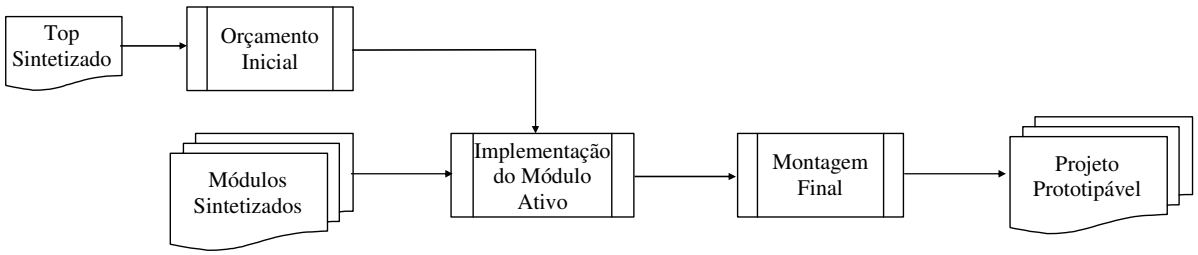


Figura 44 – Fluxo de Projeto Modular.

3.5.2.1 Fase de orçamento inicial

A fase de orçamento inicial tem como principal objetivo montar a estrutura do projeto. Nesta fase é feita a definição de quais pinos serão utilizados e onde estão posicionados os módulos e componentes inseridos no arquivo de mais alto nível do projeto. A Figura 45 apresenta o fluxo de execução da fase de orçamento inicial.

Inicialmente, os fontes HDL do projeto são sintetizados, gerando-se os arquivos de descrição lógica do circuito (EDIF). A seguir o NGDBuild posiciona os módulos do projeto como caixas preta, e fixa os componentes inseridos na Seção 3.5.1.2. 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.

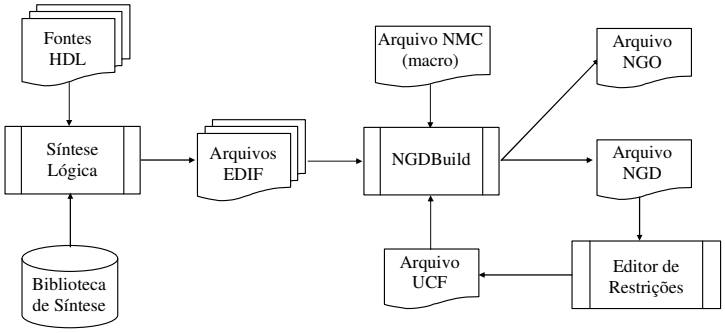


Figura 45 – Fluxo da fase de orçamento inicial do Projeto Modular que deve ser executado para cada projeto pertencente ao SDR.

3.5.2.2 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 46 ilustra o fluxo executado nesta fase.

3.5.2.3 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 47.

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.

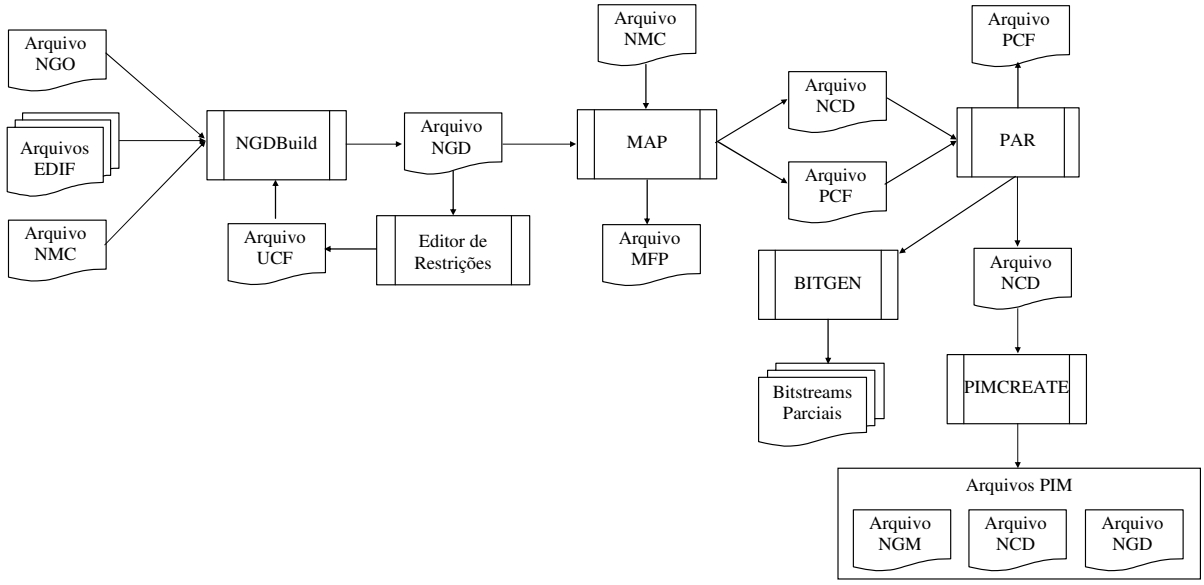


Figura 46 – Fluxo da fase de implementação do módulo ativo do Projeto Modular que deve ser executado para cada módulo do SDR.

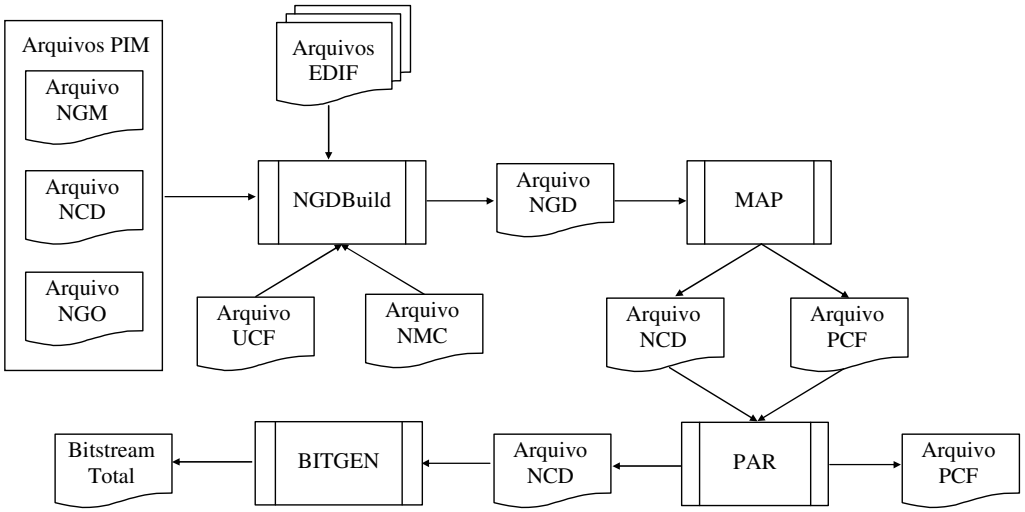


Figura 47 – Fluxo da fase montagem final do Projeto Modular que deve ser executado para cada projeto do SDR.

3.5.3 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, um de adição e um de subtração. O sistema também possui um módulo não reconfigurável responsável por efetuar E/S. A Figura 48 apresenta um diagrama de blocos do sistema.

O módulo E/S 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 deste módulo 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 E/S.

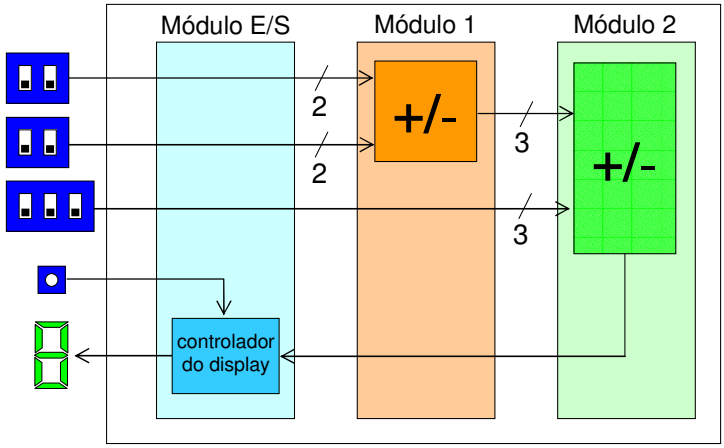


Figura 48 – Diagrama de blocos da calculadora reconfigurável para utilização com o fluxo de Projeto Modular.

As próximas Seções seguem o fluxo de Projeto Modular da mesma forma como ele foi abordado na Seção 3.5.1 e 3.5.2, partindo desde a preparação da execução do fluxo até a fase de montagem final do mesmo.

3.5.3.1 Módulos do sistema

O sistema da calculadora reconfigurável possui dois projetos denominados *top* e *top1*. O projeto *top* instancia dois módulos de adição e o projeto *top1* instancia dois módulos de subtração. O Anexo II apresenta o código VHDL do projeto *top*, sem características reconfiguráveis.

3.5.3.2 Inserção de macros, LUTs e DCMs

Para tornar o projeto *top*, apresentado no Anexo II, um projeto reconfigurável, é necessário modificar apenas o arquivo *top.vhd*, inserindo os componentes apresentados na Seção 3.2.2. O Anexo III apresenta o arquivo *top.vhd* com os componentes que permitem a reconfiguração do

sistema já inseridos.

3.5.3.3 Criação de arquivos de restrições do usuário

As áreas reconfiguráveis e os componentes inseridos na etapa anterior 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 feitas no arquivo UCF, que pode ser editado manualmente ou pela ferramenta Floorplanner. A Figura 49 apresenta o arquivo UCF para o estudo de caso da calculadora reconfigurável. A Figura 50 apresenta a representação gráfica do dispositivo com as restrições de posicionamento inseridas no arquivo UCF apresentado na Figura 49.

```
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 49 – Arquivo de restrições do usuário para o estudo de caso da calculadora reconfigurável.

3.5.3.4 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*.

3.5.3.5 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 51 apresenta os scripts de síntese para alguns arquivos VHDL do estudo de caso da calculadora reconfigurável utilizando a ferramenta Leonardo Spectrum. A Figura 51A apresenta um script de síntese para um *top*, que neste caso é o projeto *top.vhd*. A Figura 51B 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 VHDL de entrada.

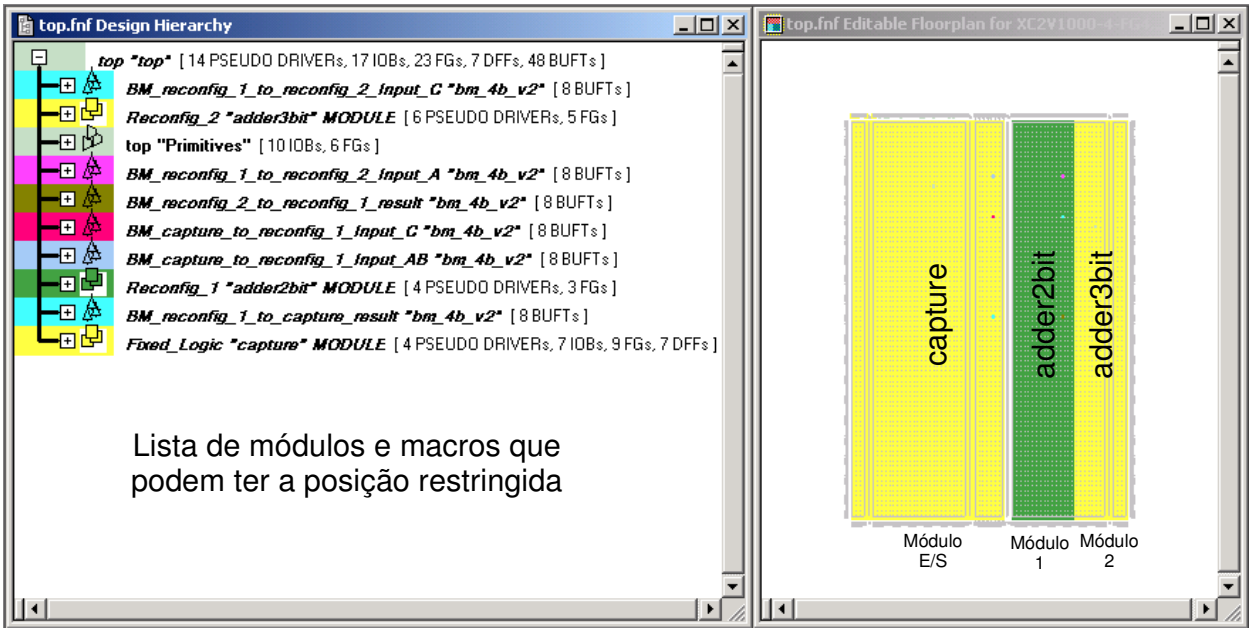


Figura 50 – Tela da ferramenta Floorplanner para o estudo de caso da calculadora reconfigurável.

```
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 51 – (A) script de síntese lógica de um *top*; (B) script de síntese lógica de um módulo.

3.5.3.6 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 52.

3.5.3.7 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 53), que deve ser gerado pelo projetista. As linhas de 5 a 7 e de 9 a 11 da Figura 53 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 apenas a ferramenta NGDBuild é executada para gerar a estrutura do projeto.

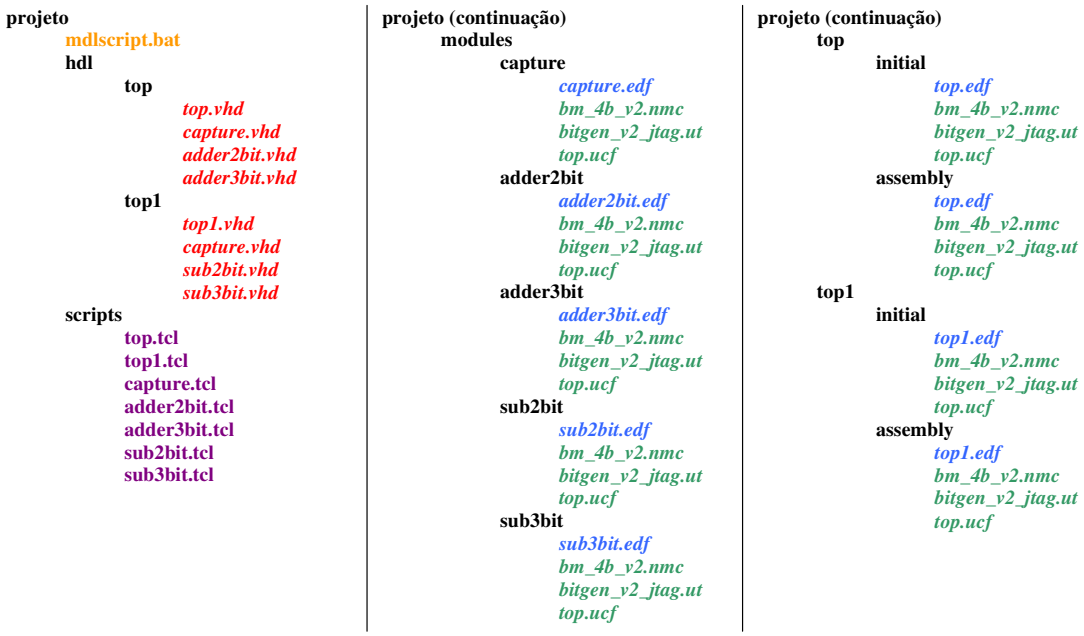


Figura 52 – Organização de diretórios sugerida para a execução do fluxo de Projeto Modular.

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

3.5.3.8 Fase de implementação do módulo ativo

Nesta fase os módulos reconfiguráveis do sistema são implementados, sendo gerado um bitstream parcial para cada módulo. A Figura 54 apresenta os quatros 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 3.5.2.2.

3.5.3.9 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 55 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 3.5.2.3.

```
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 sub2bit
32 ngdbuild -p xc2v1000fg456-4 -modular module -active sub2bit ..\..\top1\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 sub3bit
40 ngdbuild -p xc2v1000fg456-4 -modular module -active sub3bit ..\..\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 ..
```

Figura 54 – Continuação do script do Projeto Modular para o estudo de caso da calculadora reconfigurável. Fase de implementação do módulo ativo.

```
46 cd ..
47
48 cd top\assemble
49 ngdbuild -p xc2v1000fg456-4 -modular assemble -pimpath ..\..\pims top.edf
50 map -cm area top.ngd -o top_map.ncd top.pcf
51 par -rl 5 -w top_map.ncd top.ncd top.pcf
52 bitgen -f bitgen_v2_jtag.ut top.ncd
53 cd ..\..
54
55 cd top1\assemble
56 ngdbuild -p xc2v1000fg456-4 -modular assemble -pimpath ..\..\pims top.edf
57 map -cm area top.ngd -o top_map.ncd top.pcf
58 par -rl 5 -w top_map.ncd top.ncd top.pcf
59 bitgen -f bitgen_v2_jtag.ut top.ncd
60 cd ..\..
```

Figura 55 – Final do script do Projeto Modular para o estudo de caso da calculadora reconfigurável. Fase de montagem final.

3.6 Horta

O fluxo de SDR proposto por Horta [HOR04] possui como entrada módulos de hardware descritos a partir de bitstreams totais ou parciais, podendo estes terem sido implementados para qualquer dispositivo da família Virtex I da Xilinx. As vantagens deste fluxo são: (*i*) flexibilidade para realocar módulos de hardware em qualquer FPGA da família Virtex I; (*ii*) redução de tempo de projeto, pois o módulo não é re-sintetizado; (*iii*) propriedade intelectual do módulo garantida; (*iv*) características de temporização do módulo fixo. Um ponto negativo desta abordagem é não poder modificar a interface nativa de comunicação do módulo, necessitando a inserção de lógica adicional (*wrapper*) para tornar o módulo compatível com uma interface de comunicação específica.

Um requisito apontado por Horta para que o fluxo funcione corretamente é que a interface de comunicação entre o módulo reconfigurável e a lógica estática esteja bem definida. Essas interfaces foram chamadas de Gasket em [HOR02] e redefinidas para Bus Macros em [HOR04]. Para a geração de bitstreams parciais segundo este fluxo, pode-se utilizar a ferramenta PARBIT (*PARtial BItfile Transformer*) em três diferentes modos de operação:

- *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 a mesma que a posição de origem no bitstream total.
- *Block*: o usuário informa qual área 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.
- BIP (*Bitstream Intellectual Property*): o usuário informa o bitstream total que possui o módulo a ser extraído e o arquivo *top* VHDL que instancia o módulo. Devem ser inseridos no *top*, na forma de comentário, o número de linhas e colunas de CLBs ocupados pelo módulo e a coordenada superior esquerda do módulo.

O PARBIT funciona por linha de comando e não utiliza o JBits para acessar o bitstream, fazendo manipulação direta do arquivo de configuração.

3.7 Huebner

Huebner, em [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 *tristates*. A Figura 56 ilustra este problema. 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. 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.

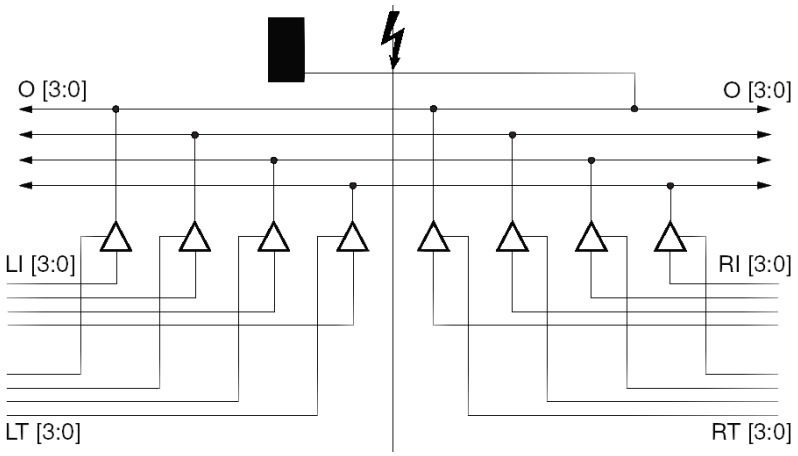


Figura 56 – Problema reportado por Huebner de fios que cruzam áreas reconfiguráveis.

Devido aos problemas supracitados das Bus Macros, Huebner desenvolveu um barramento seguindo a idéia de comunicação em SDRs proposta por Palma (Seção 3.4), onde os elementos que comunicam as diversas áreas reconfiguráveis ficam na parte inferior do FPGA. A vantagem desta abordagem 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 somente porque a reconfiguração das famílias de dispositivos Virtex da Xilinx são *glitchless*.

A princípio, Huebner tentou instanciar múltiplas macros iguais lado a lado de forma a implementar um barramento. Devido à arquitetura da Virtex-II apresentar mistura de colunas de CLBs com BRAMs e blocos multiplicadores, problemas de roteamento foram detectados. Por este motivo as macros que seriam instanciadas foram todas roteadas manualmente, construindo assim dois barramentos unidirecionais para a comunicação de módulos reconfiguráveis. A Figura 57 ilustra o diagrama de esquemático dos barramentos de leitura e escrita, enquanto a Figura 58 apresenta a planta baixa de um destes barramentos.

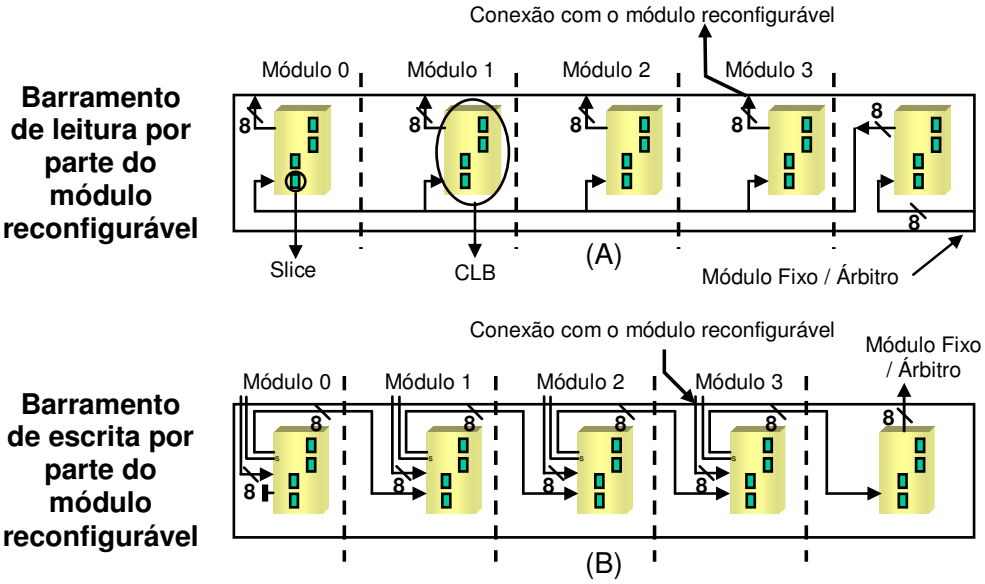


Figura 57 – Barramentos unidirecionais de Huebner.

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 de acesso ao mesmo.

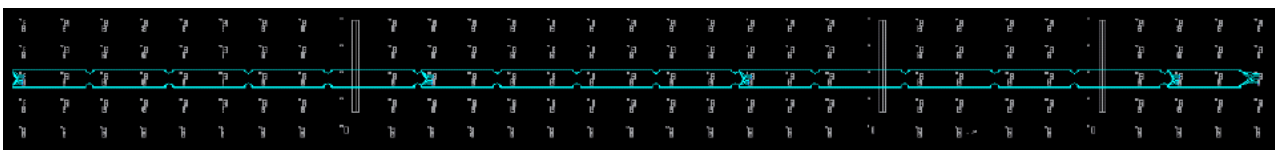


Figura 58 – Planta baixa do barramento de Huebner.

A Figura 59 apresenta como as CLBs dos barramentos de leitura e escrita estão configuradas. As CLBs do barramento de leitura estão configuradas como somente passagem de dados, enquanto as CLBs do barramento de escrita estão configuradas como multiplexadores. 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. O problema desta arquitetura de barramentos é o procedimento de arbitragem, Daisy-chain, onde a prioridade de acesso ao barramento é fixa.

Huebner, assim como Dyer, utiliza o JBits para gerar bitstreams parciais, mas a partir de um fluxo ainda diferente dos propostos por Dyer. No fluxo de Huebner todos os módulos são desenvolvidos com ferramentas de implementação padrão e os bitstreams parciais são extraídos com o JBits. Isto é possível marcando os *frames* de um bitstream total como modificados e gerando um bitstream parcial deste. A Figura 60 apresenta o trecho de código em linguagem Java utilizando a API JBits 3 responsável por gerar bitstreams parciais para dispositivos Virtex-II.

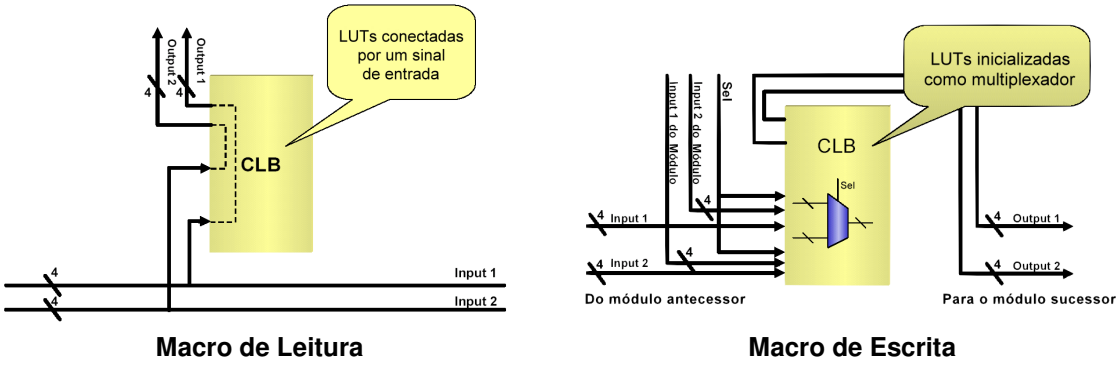


Figura 59 – Configuração das CLBs conectadas aos barramentos de Huebner.

```
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 da coluna
11    for (int y=0; y<22; y++)
12    {
13          //configurando o frame a ser capturado
14          res[0][1] = y;
15
16          //captura o frame y da linha x
17          int value[] = jbits.getCLBBits(0,x,res);
18
19          //escreve o frame lido de volta no bitstream
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 60 – Trecho de código Java/JBits para geração de bitstreams parciais de módulos.

3.8 Conclusões

A Tabela 11 apresenta um resumo dos fluxos estudados neste Capítulo, apontando as principais características de cada um.

Tabela 11 – Fluxos de projeto para SDRs.

Fluxo	Utilização de Macro	Síntese	Bitstream Parcial	Método de Geração	Compatibilidade
Dyer 1	sim	simples	manual	extração *frames*	Virtex I
Dyer 2	não	JBits 2	JBits 2	construção nativa	Virtex I
Dyer 3	sim/não	simples	JBits 2	modificação de parâmetros	Virtex I
Dyer 4	sim/não	simples	JBits 2	extração *frames*	Virtex I
Palma	sim	simples	CoreUnifier	extração *frames*	Virtex I
Projeto Modular	sim	avançada	BitGen	construção nativa	Virtex I, II, II Pro
Horta	sim	simples	ParBit	extração *frames*	Virtex I
Huebner	sim	simples	JBits 3	extração *frames*	Virtex II

A segunda coluna da tabela refere-se à utilização ou não de macros. Observa-se que, exceto os fluxos propostos por Dyer, sempre há a utilização de macros para fixar os pinos de interface dos módulos reconfiguráveis. Uma alternativa à utilização de macros é empregar o roteador do JBits, porém este é extremamente limitado em suas funcionalidades [DYE02a].

A terceira coluna refere-se ao método de síntese dos módulos reconfiguráveis. O conjunto de classes JBits não permite gerar módulos reconfiguráveis complexos, devido ao reduzido número de recursos oferecidos por ela. O fluxo de Projeto Modular utiliza parâmetros de síntese avançados, que com freqüência dificultam a síntese do projeto em comparação à síntese simples. A síntese simples aplica os parâmetros padrão, não fazendo outras restrições ao projeto que possam dificultar o processo de síntese.

A quarta e a quinta coluna apresentam a ferramenta que gera o bitstream parcial e o método de geração deste bitstream. Dentre os fluxos apresentados, apenas as alternativas propostas pela Xilinx, BitGen e JBits, constroem os bitstreams parciais de forma nativa. Os demais fluxos utilizam ferramentas próprias, gerando bitstreams parciais através da extração de *frames* de bitstream totais.

A última coluna apresenta com quais dispositivos um determinado fluxo é compatível. Nestes fluxos apresentados pode-se perceber que a maioria é compatível apenas com os FPGAs da família Virtex-I, devido a arquitetura desta ser disponibilizada publicamente. Apenas o fluxo de Projeto Modular e o fluxo de Huebner são compatíveis com FPGAs mais recentes. Segundo a análise realizada e excluindo o fluxo de Projeto Modular que possui as desvantagens apresentadas, o fluxo de Huebner é o que mais se adeqüa a um fluxo simples, rápido e compatível com dispositivos atuais, no entanto, este ainda possui uma limitação: não extrai apenas o módulo reconfigurável do projeto, mas sim *frames* inteiros onde o módulo está inserido.

O próximo Capítulo apresenta o fluxo de projeto para SDRs proposto por esta dissertação de mestrado, introduzindo um fluxo que supra algumas das restrições presentes nos atuais.

4 FLUXO DE PROJETO PROPOSTO PARA SDRS

Este Capítulo apresenta a primeira contribuição deste trabalho: um fluxo de projeto para SDRs. Este fluxo tem por objetivo simplificar os procedimentos de síntese para o usuário, utilizando ferramentas padrão. Os elementos isoladores não são baseados em *tristates*, reduzindo a propensão a erros observada em fluxos como o Projeto Modular. Além da simplicidade e de macros próprias, este fluxo contempla realocação de módulos em duas dimensões.

Esse fluxo de projeto de SDRs é ilustrado na Figura 61. Ele inicia com um projeto em linguagem de descrição de hardware que compõe o projeto inicial (Figura 61A) e é finalizado pela geração de bitstreams parciais (Figura 61L ou Figura 61M). Este fluxo é divido em três etapas: (*i*) geração e validação do projeto inicial, que não possui características reconfiguráveis (Figura 61A a Figura 61D); (*ii*) projeto inicial com características reconfiguráveis (Figura 61E a Figura 61H); (*iii*) geração de bitstreams parciais (Figura 61I a Figura 61M). As Seções a seguir apresentam os passos do fluxo de projeto proposto, possuindo correspondência direta com a Figura 61.

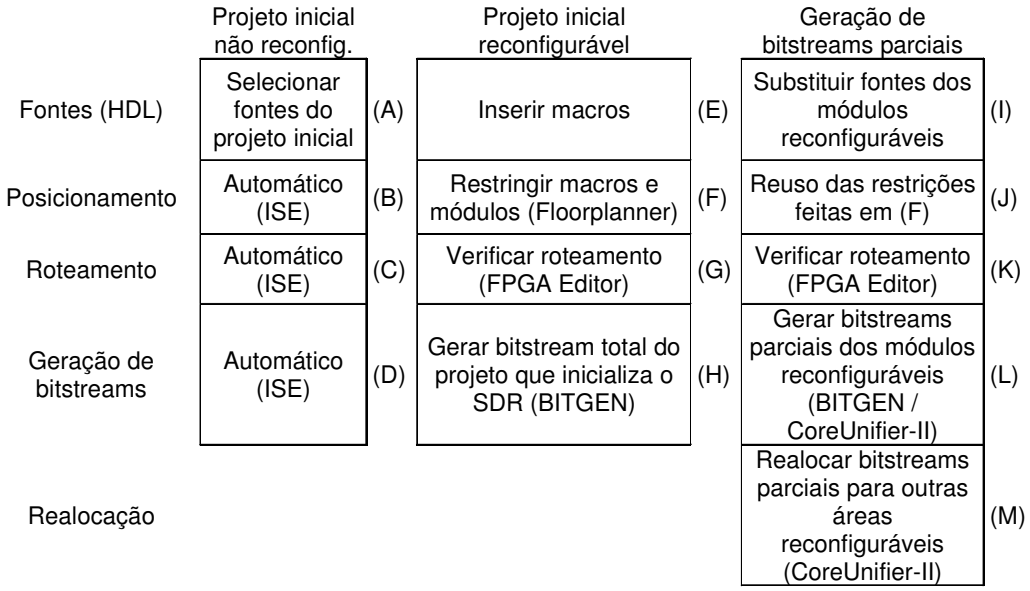


Figura 61 – Etapas de projeto para o fluxo de SDRs proposto neste trabalho.

4.1 Projeto Inicial Não Reconfigurável

O primeiro passo do fluxo de projeto é definir a estrutura geral do SDR, escolhendo a quantidade de áreas reconfiguráveis que farão parte deste e quais módulos estarão inicialmente configurados nestas áreas reconfiguráveis (Figura 61A). A escolha da quantidade de áreas reconfiguráveis é feita segundo o tamanho dos módulos reconfiguráveis a serem utilizados no sistema, o tamanho do FPGA e a necessidade de utilizar módulos reconfiguráveis em paralelo. É aconselhável prototipar este projeto usando configuração total (Figura 61B a Figura 61D), de forma a garantir que as restrições temporais deste sejam atendidas e que o mesmo funcione corretamente.

O motivo para esta prototipação inicial é garantir a operacionabilidade do ponto de partida do SDR, uma vez que serão posteriormente adicionados novos elementos, agregando complexidade ao todo.

A Figura 62 apresenta o diagrama de blocos do projeto inicial, com duas áreas reconfiguráveis (*Reconf1* e *Reconf2*) e uma área fixa.

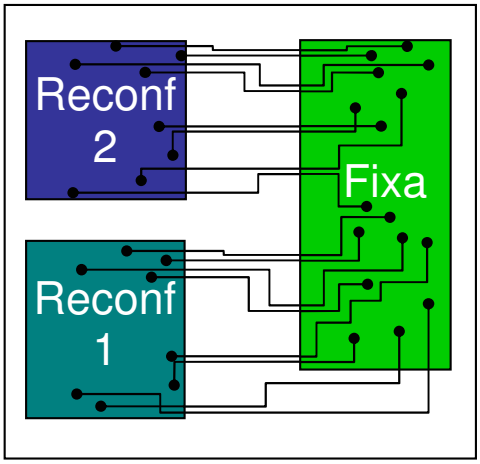


Figura 62 – Projeto Inicial.

4.2 Inserção de macros

O passo seguinte do fluxo corresponde à inserção de macros (Figura 61E) entre os módulos reconfiguráveis e o restante do projeto, conforme ilustrado na Figura 63. Estes componentes, conforme explicado na Seção 3.2.2, fixam a posição das entradas e saídas entre cada um dos módulos reconfiguráveis instanciados em diferentes projetos do sistema.

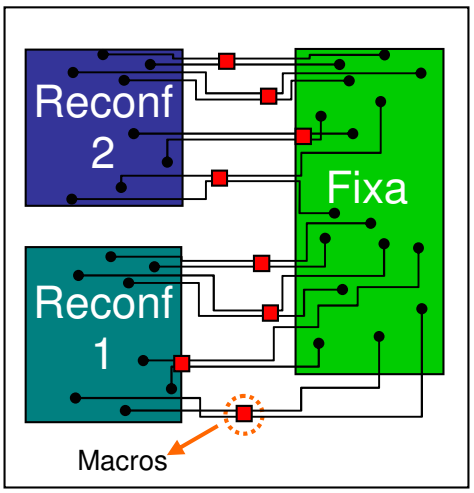


Figura 63 – Projeto com macros inseridas.

Este fluxo de projeto permite utilizar qualquer uma das macros apresentadas nos fluxos do Capítulo 3. No entanto, sugere-se a utilização das macros apresentadas na Seção 4.2.1, pois elas não utilizam *tristates* e protegem a parte fixa do sistema de receber dados espúrios durante a

reconfiguração. As Seção 4.2.2 e 4.2.3 apresentam, respectivamente, como construir as macros propostas na Seção 4.2.1 e como instanciá-las na linguagem VHDL.

4.2.1 Macros propostas

Além de fixar a comunicação dos módulos reconfiguráveis nos diferentes projetos do SDR, uma das macros propostas neste trabalho possui a função de isolar o restante do sistema enquanto um módulo está sendo substituído. Este isolamento é necessário porque a reconfiguração instantaneamente modifica o módulo, podendo assim ocorrer chaveamentos transitórios em sua saída e por conseqüência interferir no funcionamento do restante do sistema que continua operando. Este comportamento foi de fato identificado através da utilização de analisador lógico no momento da reconfiguração parcial e dinâmica do sistema. A Figura 64 apresenta as duas macros propostas neste trabalho para SDRs que possuem a área fixa à esquerda e a área reconfigurável à direita. A macro LR, apresentada na Figura 64A, possui apenas a função de fixar a comunicação da área fixa para a área reconfigurável nos diferentes projetos do SDR. A macro RLC, apresentada na Figura 64B, possui a função de fixar a comunicação da área reconfigurável para a área fixa, e adicionalmente, permitir à área fixa evitar que haja o envio de dados para si durante a reconfiguração. Caso a área reconfigurável esteja à esquerda, é possível implementar macros invertidas (RL e LRC) em relação às apresentadas.

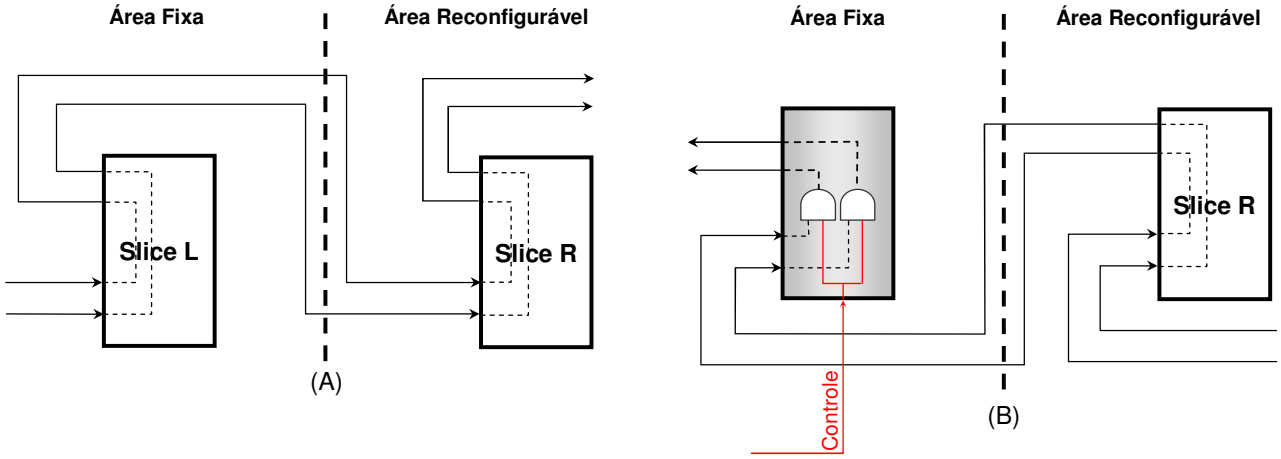


Figura 64 – Estrutura atômica das macros: (A) LR e (B) RLC. A macro LR envia dados da esquerda para a direita. A macro RLC envia dados da direita para esquerda apenas quando o módulo não estiver sendo reconfigurado, o que é indicado pelo sinal *controle*.

4.2.2 Criação de macros

Pelo conhecimento do Autor existem três métodos para projetar uma macro destinada a FPGAs Xilinx: (*i*) através da utilização da ferramenta FPGA Editor; (*ii*) escrita em linguagem XDL; (*iii*) escrita em linguagem HDL. O primeiro método de implementação é demorado e trabalhoso, mas é exatamente o hardware a ser configurado no FPGA. O método (*ii*) utiliza a linguagem de descrição da Xilinx (XDL) para descrever de forma textual a macro. Este método necessita de um

conhecimento avançado da arquitetura do dispositivo para de fato obter sucesso no projeto de uma macro, não sendo normalmente indicado para construir macros, mas sim realizar um grande número de pequenas modificações em macros ou projetos, já que é baseado em texto e pode ser rapidamente modificado. O Anexo V apresenta o projeto de uma porta lógica "E" através da linguagem XDL. O método (*iii*) utiliza o fluxo de projeto padrão para FPGAs Xilinx descrito na Seção 3.1. Este método de implementação é rápido e menos suscetível a erros que os outros métodos, mas o hardware gerado pode ser bem diferente do esperado, devido ao processo de posicionamento e roteamento ser não determinístico. Por este motivo, este método é indicado apenas para macros grandes, realizando-se as eventuais correções de posicionamento e/ou roteamento através dos métodos (*i*) ou (*ii*).

Devido à simplicidade das macros propostas e a necessidade de construí-las apenas uma única vez, o método (*i*) foi utilizado. Os passos para a construção da macro segundo o método (*i*) são apresentados no Anexo IV, enquanto a Figura 65 ilustra a macro RLC criada pelo FPGA Editor. A macro LR é semelhante à apresentada na Figura 65, com a diferença de não possuir o sinal *ctrl* e as saídas (*o*) estarem posicionadas no lugar das entradas (*i*) e vice-versa.

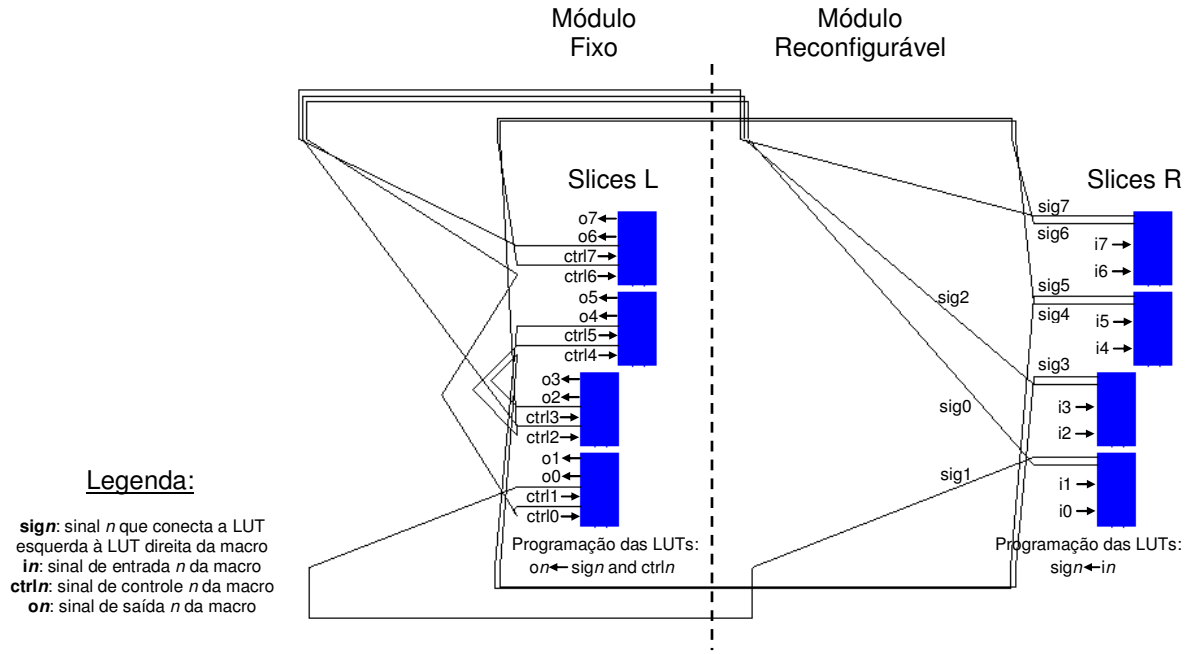


Figura 65 – Macro RLC.

4.2.3 Instanciação de macros

A Figura 66 apresenta como a macro RLC, ilustrada na Figura 65, é instanciada na linguagem VHDL. Esta macro possui 8 bits de entrada de dados (*i*), 8 bits de saída (*o*) e 8 bits que controlam a passagem de dados (*ctrl*). Os sinais *data_in*, *data_out* e *control* representam, respectivamente, os sinais do módulo reconfigurável com destino à parte fixa do sistema, os sinais oriundos do módulo reconfigurável que já passaram pela macro e os sinais do módulo fixo que controlam a passagem de dados da macro. É aconselhável que antes de executar uma

reconfiguração, o sinal de controle da macro seja desabilitado, para que chaveamentos temporários de *data_in* não sejam percebidos em *data_out*, permitindo que a parte fixa do SDR continue operando durante o processo de reconfiguração. Caso seja necessário uma interface entre o módulo fixo e reconfigurável com mais de 8 bits, mais macros devem ser instanciadas, conforme necessário.

A instanciação da macro LR funciona de forma análoga à macro RLC, considerando apenas que a mesma usa sinais com direção de propagação da informação invertida e não possui o sinal de controle (*ctrl*).

```
01 entity top is
02 port(
03    --portas do usuario adicionadas aqui
04 );
05 end top;
06
07 architecture arch of top is
08
09 signal data_in : std_logic_vector(7 downto 0);
10 signal data_out: std_logic_vector(7 downto 0);
11 signal control : std_logic;
12
13 component RLC is
14 port(
15    i0: in std_logic;
16    i1: in std_logic;
17    i2: in std_logic;
18    i3: in std_logic;
19    i4: in std_logic;
20    i5: in std_logic;
21    i6: in std_logic;
22    i7: in std_logic;
23
24    ctrl0: in std_logic;
25    ctrl1: in std_logic;
26    ctrl2: in std_logic;
27    ctrl3: in std_logic;
28    ctrl4: in std_logic;
29    ctrl5: in std_logic;
30    ctrl6: in std_logic;
31    ctrl7: in std_logic;
32
33    o0: out std_logic;
34    o1: out std_logic;
35    o2: out std_logic;
36    o3: out std_logic;
37    o4: out std_logic;
38    o5: out std_logic;
39    o6: out std_logic;
40    o7: out std_logic
41 );
42 end component;
44 begin
45
46 RLC1: RLC port map (
47    i0 => data_in(0),
48    i1 => data_in(1),
49    i2 => data_in(2),
50    i3 => data_in(3),
51    i4 => data_in(4),
52    i5 => data_in(5),
53    i6 => data_in(6),
54    i7 => data_in(7),
55
56    ctrl0 => control,
57    ctrl1 => control,
58    ctrl2 => control,
59    ctrl3 => control,
60    ctrl4 => control,
61    ctrl5 => control,
62    ctrl6 => control,
63    ctrl7 => control,
64
65    o0 => data_out(0),
66    o1 => data_out(1),
67    o2 => data_out(2),
68    o3 => data_out(3),
69    o4 => data_out(4),
70    o5 => data_out(5),
71    o6 => data_out(6),
72    o7 => data_out(7)
73 );
74
75 end arch;
```

Figura 66 – Código VHDL ilustrando a instanciação da macro RLC.

4.3 Restringir posicionamento

O terceiro passo no fluxo de implementação de SDRs é restringir o posicionamento dos módulos reconfiguráveis e das macros (Figura 61F). Esta restrição pode ser feita pela ferramenta gráfica Floorplanner da Xilinx, que permite estabelecer restrições para definir uma região sobre a superfície de silício do FPGA que deve ser ocupada pelos componentes do sistema. O Floorplanner gera como saída um arquivo de restrições do usuário (*User Constraints File* - UCF) que é utilizado durante as fases de posicionamento e roteamento do projeto.

A Figura 67 apresenta um exemplo de como os módulos reconfiguráveis e as macros podem ser posicionados para que o projeto possa ser posteriormente dinamicamente reconfigurado. O importante deste posicionamento é que os módulos reconfiguráveis ocupem subconjuntos de colunas de recursos (CLBs) distintos do FPGA, e que as macros estejam verticalmente alinhadas nas interfaces dos módulos reconfiguráveis. O motivo destas restrições de posicionamento serem

implementadas verticalmente é que os dispositivos Xilinx são reconfigurados em colunas.

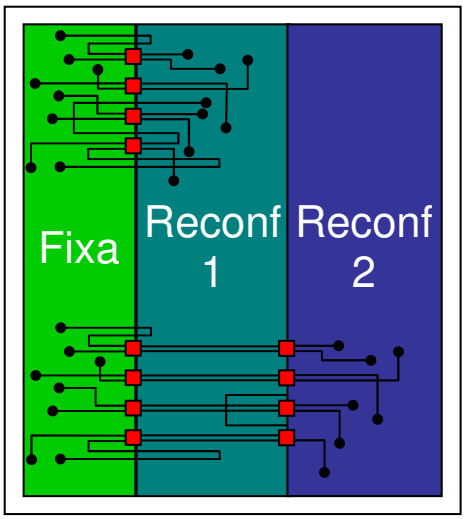


Figura 67 – Restrição de posicionamento de macros e módulos.

4.4 Roteamento

Uma vez executado os passos de posicionamento e roteamento, é necessário verificar se o roteamento efetuado automaticamente atende os requisitos de um SDR. Como já mencionado, ao restringir a lógica de um módulo a uma determinada área do FPGA, não obrigatoriamente os fios que interconectam os componentes internos deste módulo permanecerão dentro dos limites definidos pelas restrições de posicionamento. Com freqüência, os fios internos de um módulo invadem a área de outro, conforme ilustrado na Figura 68A.

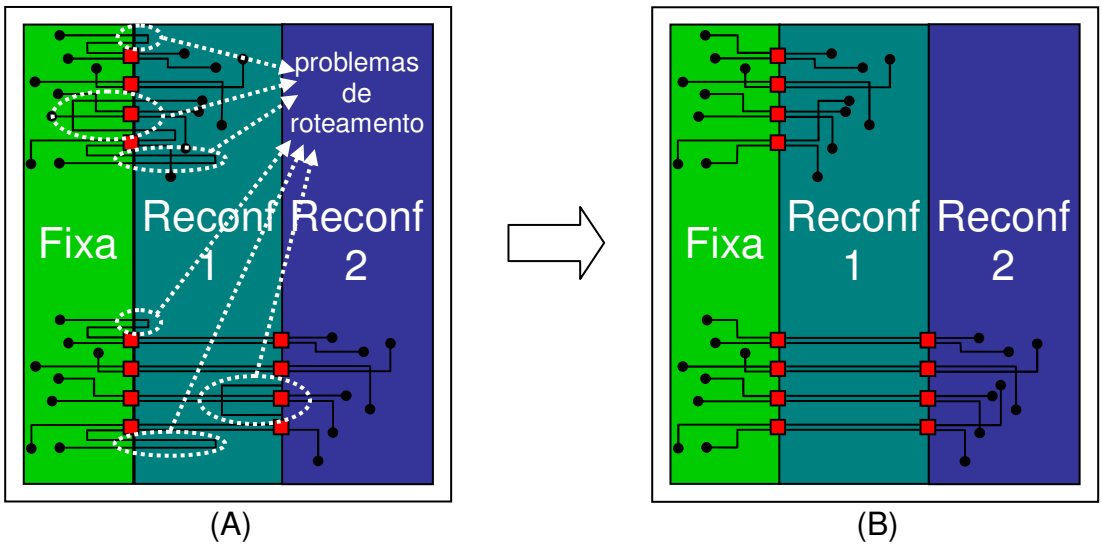


Figura 68 – Exemplos de problemas de roteamento, mesmo com o uso de macros: (A) identificação dos problemas de roteamento; (B) problemas de roteamento corrigidos.

Caso um ou mais fios cruzem a área pré-definida, estes poderão ficar desconectados após a

reconfiguração, podendo ocasionar o não funcionamento do projeto e até mesmo um dano físico no dispositivo. Por este motivo, deve-se verificar as interfaces dos módulos reconfiguráveis (Figura 61G), buscando obter um roteamento como o da Figura 68B. As interfaces dos módulos reconfiguráveis podem ser verificadas pela ferramenta XDLAnalyzer, proposta nesta dissertação, ou pela ferramenta FPGA Editor. Ambas alternativas de verificação de roteamento serão tratadas nas Seções 4.4.1 e 4.4.2.

4.4.1 FPGA Editor

Através da ferramenta FPGA Editor é possível visualizar e manipular a forma como os recursos estão fisicamente alocados no FPGA. Ao se abrir o projeto já roteado nesta ferramenta, deve-se localizar as macros no mesmo. As macros propostas na Seção 4.2.1 utilizam CLBs de duas colunas adjacentes, sendo uma pertencente a um módulo fixo/reconfigurável e outra pertencente a um módulo reconfigurável. A fronteira entre duas colunas de CLBs são demarcadas no FPGA Editor por *linhas longas*[4] (em inglês, *long lines*) verticais. Uma vez encontrada uma macro deve-se verificar se algum fio cruza a fronteira ao longo de toda a coluna sem passar por alguma macro. O sinal de relógio é o único fio que pode cruzar livremente o FPGA sem passar por macros, pois este utiliza uma árvore específica de roteamento.

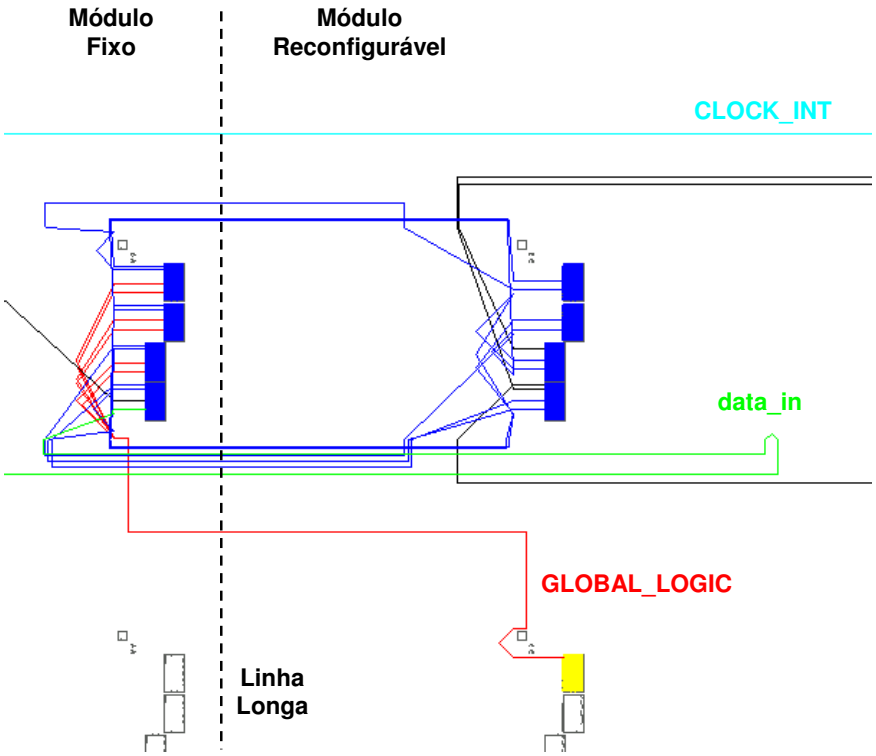


Figura 69 – Macro LR com dois fios invadindo o módulo reconfigurável.

A Figura 69 apresenta uma macro LR com um sinal de relógio passando livremente pela macro (*CLOCK_INT*) e dois casos típicos de problemas: (*i*) um fio de dados que cruza a fronteira e depois retorna para entrar na macro (*data_in*) e (*ii*) um fio de constante posicionado no módulo

[4] *Linha longa* refere-se a um recurso de roteamento que interconecta CLBs distantes.

reconfigurável mas que é conectado à macro no módulo fixo (*GLOBAL_LOGIC*). Para resolver o problema (*ii*) basta remover o roteamento completo do fio, arrastar a *slice* fonte do sinal para qualquer lugar disponível do módulo fixo e refazer o roteamento de forma automática. Para resolver o problema (*i*) não é aconselhado utilizar essa mesma técnica de movimentação do *slice* fonte do sinal, pois este está conectado a outros *slices* que podem ter suas características temporais modificadas e por conseqüência resultar no posterior não funcionamento do circuito. Neste caso é mais indicado voltar ao passo anterior e tentar novas restrições de posicionamento. Este procedimento de "tentativa e erro" também ocorre no fluxo de Projeto Modular, sendo a principal dificuldade dos fluxos de SDR.

Existem casos onde não se consegue evitar que um determinado fio não cruze a fronteira. Quando esse fio é originado de um módulo fixo e invade um módulo reconfigurável, esse fato não representa maiores problemas, contanto que o SDR não utilize realocação e que o projetista verifique para cada projeto diferente se o fio invasor utilizou exatamente as mesmas trilhas do FPGA. A realocação não pode ser utilizada em SDRs que possuem fios do módulo fixo invadindo a área reconfigurável, porque os módulos construídos em outras áreas reconfiguráveis não possuirão a configuração do fio invasor, e se a reconfiguração fosse executada, o módulo fixo pararia de funcionar, ou mesmo causar danos ao FPGA.

4.4.2 XDLAnalyzer

A ferramenta XDLAnalyzer, proposta pelo Autor, possui a função de analisar todos os sinais de um projeto, verificando se algum deles atravessa a fronteira do módulo de origem sem passar por macros. A partir disso, pode-se utilizar a ferramenta FPGA Editor para visualizar/corrigir os sinais com erros de roteamento ou tentar um novo posicionamento sem sequer abrir o FPGA Editor.

O XDLAnalyzer utiliza como entrada o arquivo UCF do projeto, o qual possui as informações das fronteiras dos módulos, e a descrição XDL do projeto, que descreve em formato texto o caminho percorrido para cada sinal, conforme exemplificado nas linhas 82 a 106 da Figura 107 do Anexo V. O arquivo XDL é gerado pela ferramenta XDL da Xilinx após a execução da mesma com a linha de comando `XDL -ncd2xdl arquivo.ncd`, onde `arquivo.ncd` é o nome do projeto gerado pela ferramenta de síntese física ou modificado pela ferramenta FPGA Editor. O primeiro passo da ferramenta é converter as coordenadas utilizadas pelo UCF para as coordenadas utilizadas pelo XDL. A seguir o XDLAnalyzer percorre o arquivo XDL até encontrar o primeiro sinal (*net*). Para cada sinal encontrado, deve-se determinar as coordenadas do primeiro ponto de interconexão (*pip*) de sua descrição (*LIOIR17* = coluna de E/S esquerda, linha 17). A partir dessas coordenadas é possível determinar no arquivo UCF a qual módulo o sinal pertence e verificar se algum dos *pip*s seguintes possuem as coordenadas fora do módulo origem. Se isso ocorrer, este sinal é reportado ao usuário. O procedimento é repetido até o final do arquivo XDL.

4.5 Geração de bitstream total

A ferramenta BitGen da Xilinx é utilizada para gerar o bitstream total que inicializa o sistema (Figura 61H). O BitGen pode ser executado tanto através do ambiente de desenvolvimento ISE quanto via linha de comando. Via linha de comando, um bitstream total pode ser gerado pelo comando `bitgen arquivo.ncd`, onde `arquivo.ncd` é o nome do arquivo gerado pela ferramenta de síntese física ou modificado pela ferramenta FPGA Editor da Xilinx. O `arquivo.bit` gerado pelo BitGen pode ser utilizado pela ferramenta Impact para configurar o FPGA.

4.6 Inserir novo módulo reconfigurável

A construção de projetos com novos módulos reconfiguráveis é feita com o reuso do projeto inicial e substituindo-se o fonte HDL específico do módulo reconfigurável (Figura 61I).

4.7 Reuso de restrições de posicionamento

O arquivo UCF gerado na Seção 4.3 é reutilizado neste passo para fixar o novo módulo reconfigurável exatamente na mesma posição do módulo pelo qual está sendo substituído (Figura 61J).

4.8 Verificar roteamento

Neste passo, o projeto deve ser verificado conforme apresentado na Seção 4.4. Esta verificação pode ser feita com o FPGA Editor ou com o XDLAnalyzer (Figura 61K).

4.9 Geração de bitstreams parciais

Esta Seção apresenta duas ferramentas para extrair módulos reconfiguráveis a partir de bitstreams totais (Figura 61L), gerando bitstreams parciais conforme ilustra a Figura 70. Ambas as ferramentas utilizam o método de extração de *frames* e são compatíveis com os dispositivos Virtex-II e Virtex-II Pro.

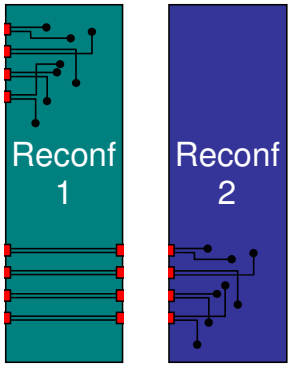


Figura 70 – Módulos reconfiguráveis extraídos do projeto.

4.9.1 BitGen e BitMask

Bitstreams parciais podem ser extraídos de projetos totais descritos no formato NCD com a ferramenta BitGen da Xilinx. Três tipos de colunas podem ser incluídos no bitstream parcial através dos seguintes parâmetros na linha de comando do BitGen:

- **`-g PartialMask0`**: colunas de CLB;
- **`-g PartialMask1`**: colunas de BRAM;
- **`-g PartialMask2`**: colunas de interconexão de BRAM;

A documentação disponível da ferramenta BitGen [XIL04a] informa apenas que para selecionar as colunas específicas de um determinado tipo de coluna utiliza-se uma máscara com valores em hexadecimal, mas não informa como gerar esta máscara. Através de um termo de cooperação de pesquisa assinado com regras de sigilo (*Non Disclosure Agreement* - NDA) entre o GAPH e a Xilinx obteve-se dados para gerar essas máscaras. Por este motivo, foi criada a ferramenta BITMask, que transforma as coordenadas dos módulos do projeto (disponíveis no arquivo UCF ou no FPGA Editor) na máscara a ser utilizada pelo BitGen.

O BITMask é executado pela linha de comando: **`BITMask inicio fim`**. Três máscaras são geradas pelo BITMask, uma para cada **`PartialMask`**. Portanto, o usuário decide se os valores informados dos parâmetros **`inicio`** e **`fim`** correspondem ao número da coluna de *slices* (**`PartialMask0`**) ou ao número da coluna de BRAMs (**`PartialMask1`** e **`PartialMask2`**). No primeiro caso, **`inicio`** e **`fim`** correspondem as *slices* inicial e final do módulo, no segundo caso, **`inicio`** e **`fim`** correspondem à primeira coluna e à última de BRAMs. Se o módulo ocupar apenas uma coluna de recursos, basta informar os mesmos valores para **`inicio`** e **`fim`**.

Para criar a máscara para o parâmetro **`PartialMask0`**, BITMask divide **`inicio`** e **`fim`** por dois para transformar colunas de *slices* em colunas de CLBs, e soma três a cada variável para desconsiderar as primeiras três colunas do FPGA, que são a coluna de relógio (coluna 0) e as colunas de IOBs (colunas 1 e 2). A seguir deve-se gerar um valor binário que possua os bits entre **`inicio`** e **`fim`** habilitados, onde o bit mais a direita é o bit menos significativo. A Figura 71 apresenta um exemplo de como gerar a máscara para o parâmetro **`PartialMask0`** de um módulo implementado entre as *slices* 2 e 9.

Para criar a máscara para os parâmetros **`PartialMask1`** e **`PartialMask2`**, o BITMask deve apenas gerar um valor binário que possua os bits entre **`inicio`** e **`fim`** habilitados, onde o bit mais a direita é o bit menos significativo.

4.9.2 CoreUnifier-II Pro

O CoreUnifier [MES03] [MOR03b] é uma ferramenta gráfica desenvolvida pelo Autor 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-I disponibilizadas no Application Note 151 [XIL04d].

O CoreUnifier-II Pro, assim como o CoreUnifier, também gera bitstreams parciais. Além disto, permite realocação de bitstreams e é destinado a manipular bitstreams dos dispositivos Virtex-II e Virtex-II Pro. Ele foi construído com base no conhecimento obtido do CoreUnifier e da documentação disponível no JBits 3. Tais fontes de informações foram importantes para o seu desenvolvimento porque a organização do bitstream em relação a sua arquitetura não foi aberta ao público, como feito para o dispositivo Virtex-I.

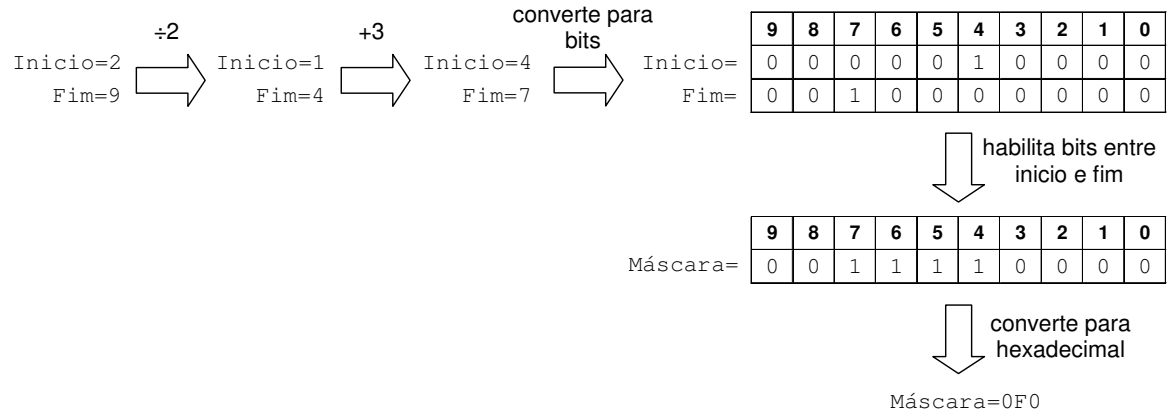


Figura 71 – Exemplo de como gerar a máscara para o parâmetro PartialMask0 de um módulo implementado entre as *slices* 2 e 9.

A ferramenta CoreUnifier-II Pro possui dois modos de operação:

- Extração: gera um bitstream parcial das colunas selecionadas a partir de um bitstream total. Este modo de operação é automaticamente utilizado quando um único bitstream total é aberto na interface gráfica. Neste modo de operação, também é possível realocar as colunas selecionadas antes de salvar o bitstream parcial, o que é feito arrastando o conjunto de colunas para a nova posição.
- Unificação: gera um bitstream parcial a partir da unificação de dois bitstreams totais. Esse modo de operação é automaticamente selecionado quando dois bitstreams totais são abertos na interface gráfica. O primeiro bitstream total aberto é denominado *mestre*, sendo este o projeto inicializa o SDR. O segundo bitstream total aberto possui o módulo reconfigurável a ser inserido sobre o mestre. A diferença deste modo de operação para o primeiro é que neste, o módulo reconfigurável não precisa ocupar toda a coluna, podendo ser mantida lógica do projeto inicial na mesma coluna. Este modo de operação permite a realocação de um módulo nas duas dimensões: linha e coluna.

As próximas Seções descrevem a relação entre arquitetura, memória de configuração e bitstreams dos dispositivos Virtex-II e Virtex-II Pro que foram compiladas para a implementação da ferramenta CoreUnifier-II Pro. Estas informações complementam o estudo efetuado na dissertação de Daniel Mesquita [MES02], podendo servir como base para novas ferramentas que manipulam bitstreams. Cabe ressaltar que não foram utilizadas informações disponíveis do NDA para a implementação da ferramenta CoreUnifier-II Pro, tendo sido estas deduzidas a partir de [XIL04b] e [XIL04d].

4.9.2.1 Memória de configuração da Virtex-II e Virtex-II Pro

Conforme mencionado anteriormente, aa memóriaa de configuração das famílias Virtex-II e Virtex-II Pro podem ser vistas como uma matriz de bits. Esses bits são agrupados em *frames* verticais com um bit de largura, que se estendem do topo à base da matriz. Existem três tipos de colunas: CLB, BRAM e interconexão de BRAM. Recursos de E/S (IOBs) e de relógio fazem parte das colunas de CLBs e podem conter um número diferente de *frames*, conforme ilustrado na Figura 72.

Cada CLB é cortada verticalmente por 22 *frames* e horizontalmente por 80 bits. A Figura 73 apresenta uma abstração da composição de um *frame* de uma coluna de CLBs do dispositivo Virtex-II 1000. Os 96 primeiros bits de um *frame* correspondem aos IOBs do topo da coluna, e os 96 últimos, aos IOBs da base da coluna. Entre esses grupos, há conjuntos de 80 bits correspondentes às CLBs de uma coluna do dispositivo. Para o dispositivo Virtex-II 1000, há 32 desses conjuntos, conforme apresentado na Tabela 6, da página 31.

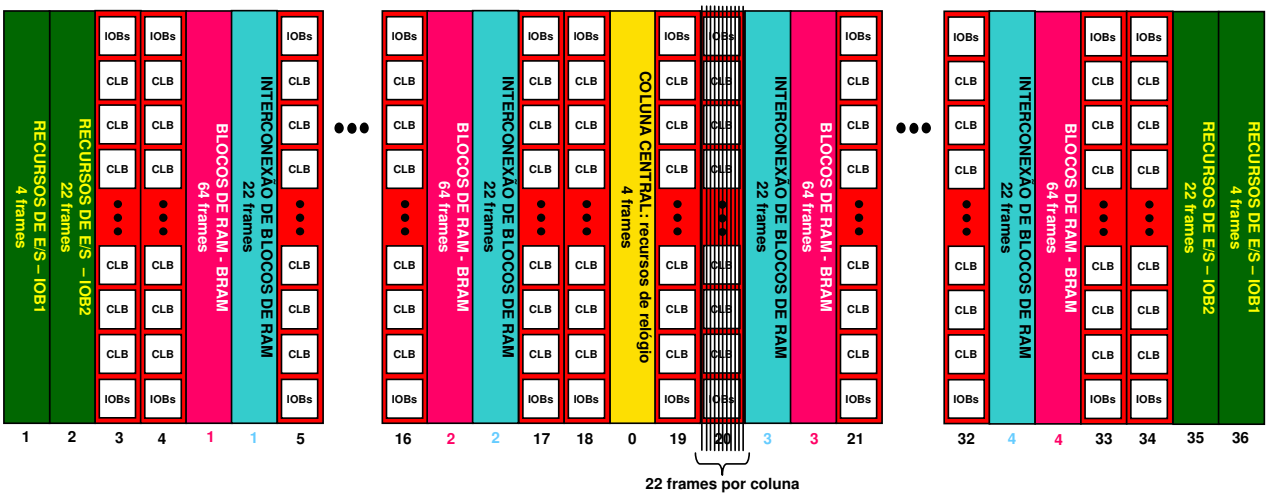


Figura 72 – Disposição em colunas dos elementos do FPGA Virtex-II 1000.

4.9.2.2 Arquivo de configuração da Virtex-II e Virtex-II Pro

A ferramenta BitGen pode gerar bitstreams em dois formatos. O primeiro é em formato binário, normalmente utilizado para configurar FPGAs (possui a extensão ".bit"). O segundo é em formato texto, utilizado para manipular o bitstream (possui a extensão ".rbt"). Um FPGA pode ser configurado pela ferramenta Impact a partir de ambos os formatos de arquivo.

A Figura 74 apresenta a estrutura de um bitstream total do dispositivo Virtex-II 1000 gerado pelo BitGen, composto por 127797 palavras de 32 bits. Destas, apenas as palavras de dados (117130 palavras) possuem a função de configurar o FPGA. As outras ou são desnecessárias ou são utilizadas pelo processo de configuração. Neste bitstream existem 10522 palavras desnecessárias, pois repetem dados que já foram enviados nas palavras de dados. O Autor utilizou a ferramenta CoreUnifier para extrair estas 10522 palavras do bitstream e a configuração do FPGA foi executada corretamente. Para informações completas sobre as palavras utilizadas pelo processo de

configuração, remete-se o leitor a [XIL04b].

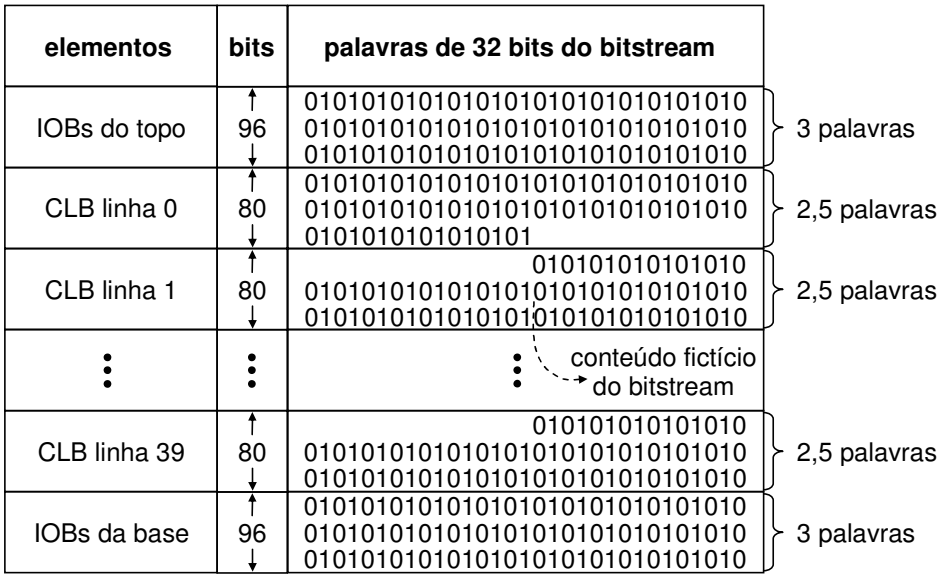


Figura 73 – Abstração de um *frame* do FPGA Virtex-II 1000 em um bitstream.

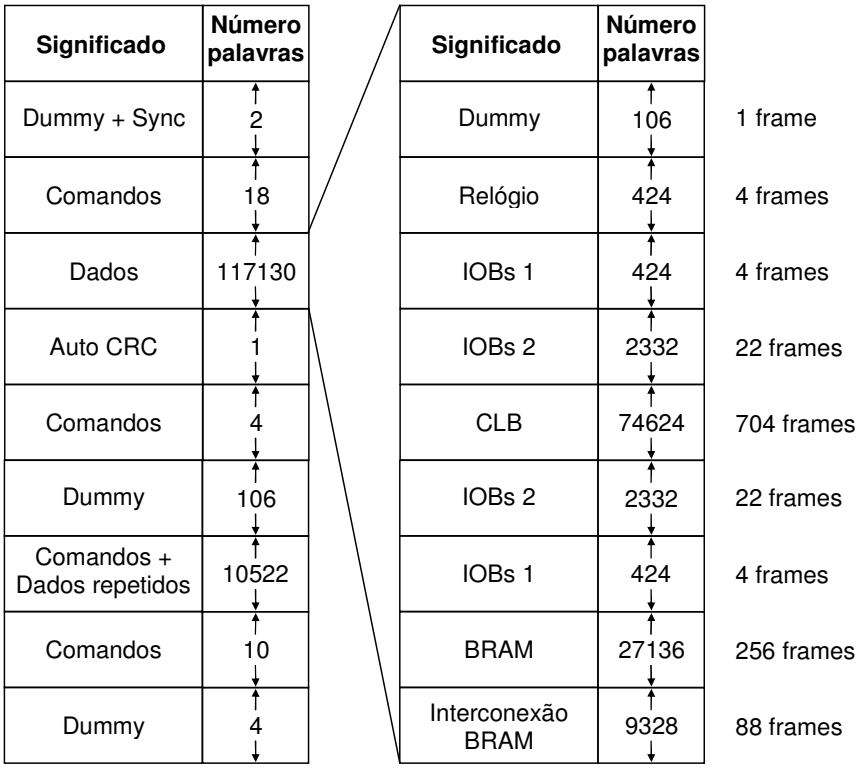


Figura 74 – Estrutura do arquivo de configuração da Virtex-II 1000.

A Figura 75 apresenta um bitstream parcial gerado pelo CoreUnifier-II Pro. A linha 5 indica que o bitstream foi gerado para um dispositivo Virtex-II 1000 e a linha 7 informa que a partir desta linha existem 597664 bits a serem enviados para o dispositivo. As linhas 8 e 9 são utilizadas para sincronizar o hardware de recepção de dados com a velocidade na qual o bitstream está sendo enviado. As linhas 10 e 11 inicializam o cálculo de CRC para posterior verificação de dados de

configuração enviados para o dispositivo. As linhas 12 e 13 são escritas para garantir que o bitstream enviado é destinado ao dispositivo correto. As linhas 14 e 15 informam que está sendo feito o download de uma configuração para o dispositivo, e não readback. As linhas 16 e 17 informam que os dados enviados configuram o FPGA a partir da coluna de CLB 20 (MJA 23-3) *frame* 0 (MNA 0). As linhas 18 e 19 informam que a partir deste ponto serão enviadas 18656 palavras de 32 bits, ou seja, 8 colunas de CLBs (18656 ÷ 106 ÷ 22 = 8, onde 106 é número de palavras necessárias para configurar um *frame* de 3392 bits (3392 ÷ 32 = 106) e 22 é o número de *frames* que possui uma coluna de CLBs). Os dados de configuração da coluna de CLB 20 a coluna 27 são enviados entre as linhas 20 e 18675. A linha 18676 apresenta o cálculo de CRC automático efetuado após o envio de dados. As linhas 18677 a 18680 executam uma nova verificação total antes de terminar o bitstream. As linhas 18681 a 18682 retiram, por segurança, a sincronização efetuada no início do bitstream, impossibilitando o envio de dados até uma nova sincronização. As linhas 18683 a 18684 atualizam a sincronização feita nas duas linhas anteriores.

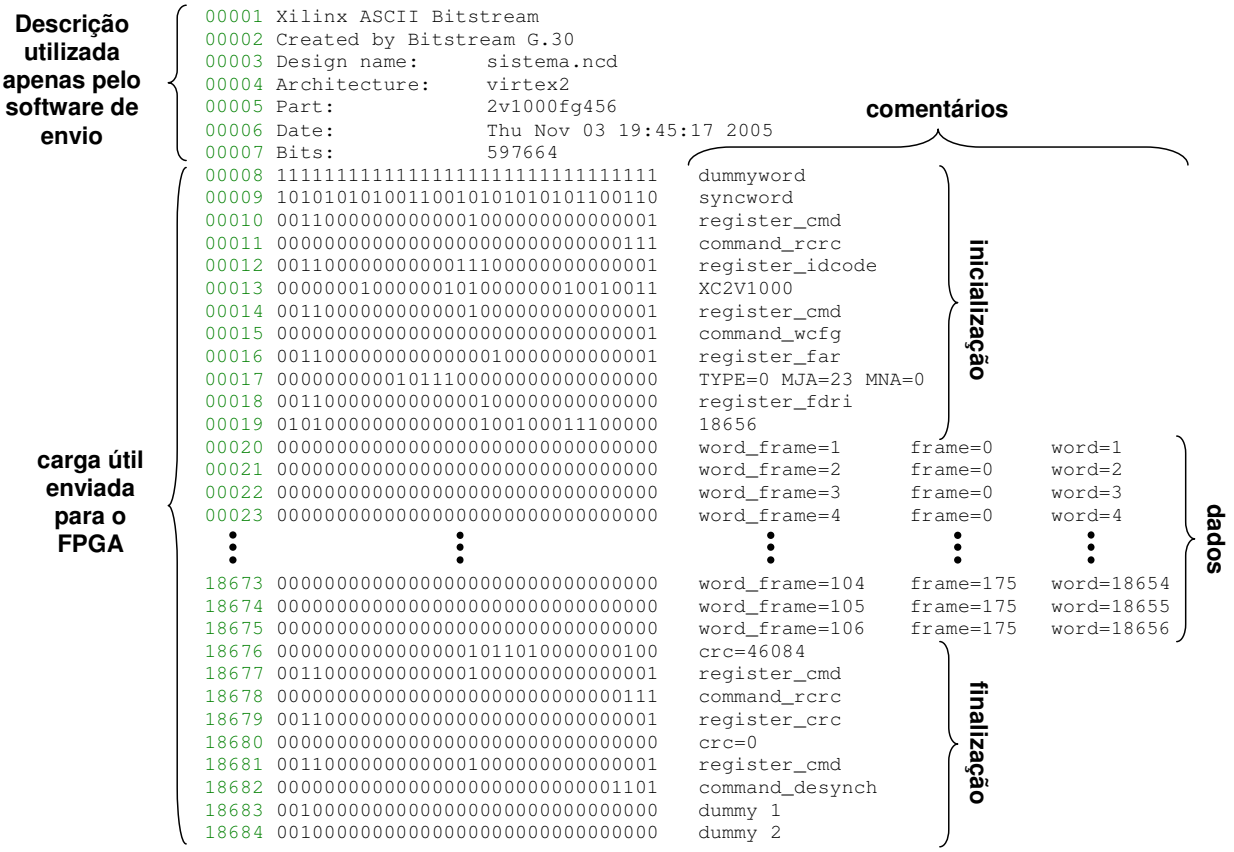


Figura 75 – Bitstream parcial gerado pelo CoreUnifier-II Pro.

4.9.2.3 Endereçamento de CLBs da Virtex-II e Virtex-II Pro

Para que seja factível a reconfiguração parcial de módulos a partir de bitstreams totais, faz-se necessária a localização de CLBs dentro do arquivo de configuração. Segundo a Figura 74, para chegar até o primeiro *frame* de CLBs do arquivo de configuração deve-se deslocar 31 *frames* de dados (1 de dummy, 4 de relógio, 4 de IOBs 1, 22 de IOBs 2) a partir do comando FDRI. Logo, a

localização da palavra de 32 bits inicial de qualquer coluna de CLBs pode ser realizada através da Equação 1.

$$clb\_st\_wd = FL \times (31 + (22 \times CLB\_Col)) \quad \text{Equação 1}$$

onde:

- *clb_st_wd*: é o índice da palavra que inicia uma coluna de CLBs;
- *FL*: é o número de palavras de um *frame*. Esse valor é obtido dividindo o tamanho do *frame* em bits do dispositivo utilizado (informação disponível na Tabela 7 da página 31) por 32.
- *CLB_Col*: é o número da coluna de CLB desejada, sendo a coluna mais à esquerda do dispositivo a coluna zero.

Como já mencionado, uma coluna de CLBs é formada verticalmente por 22 *frames*. A Equação 2 permite acessar individualmente cada *frame* de uma coluna de CLBs.

$$fm\_st\_wd = clb\_st\_wd + (MNA \times FL) \quad \text{Equação 2}$$

onde:

- *fm_st_wd*: é o índice da palavra que inicia um *frame* de uma coluna de CLB;
- *MNA*: é o índice do *frame* desejado dentro de uma coluna de CLBs. Este valor varia de 0 a 21.

Conforme apresentado na Figura 73, cada CLB dentro de um *frame* possui 80 bits (2,5 palavras de 32 bits). A Equação 3 permite localizar o índice da palavra que contém os bits de programação de uma CLB específica dentro de um *frame*. Cabe ressaltar que se o resultado não for inteiro, os bits da linha da CLB requisitada iniciam a partir da metade da palavra de 32 bits resultante.

$$row\_st\_wd = fm\_st\_wd + 3 + (CLB\_Row \times 2{,}5) \quad \text{Equação 3}$$

onde:

- *row_st_wd*: é o índice da palavra onde inicia a programação de uma CLB dentro de um *frame*;
- *CLB_Row*: é o número da linha de CLB desejada, sendo a linha de CLB superior a linha zero.

4.9.2.4 Realocação nas famílias Virtex-II e Virtex-II Pro

Considere a realocação da CLB de coordenadas 0,0 para as coordenadas 1,1 em um dispositivo Virtex-II 1000, conforme apresentado na Figura 76. O primeiro passo é determinar o índice da palavra do bitstream onde inicia a configuração da CLB de origem. Isto é feito aplicando

as três Equações apresentadas na Seção anterior. Substituindo-se as coordenadas 0 e 0 em *CLB_Col* e *CLB_Row*, 106 (3392 ÷ 32) em *FL* e 0 em *MNA* obtêm-se que os primeiros 80 bits de programação da CLB 0,0 iniciam na palavra 3289. O segundo passo é determinar o índice da palavra da CLB de destino dentro do bitstream. Isto é feito aplicando as mesma três equações com as coordenas 1 e 1 para *CLB_Col* e *CLB_Row*, o que resulta no índice 5623,5. Neste caso deve-se copiar 80 bits a partir do inicio da palavra 3289 para a segunda metade da palavra 5623 e as duas palavras seguintes. Para realocar a CLB inteira basta reaplicar os dois passos para os outros 21 *frames* variando o valor de *MNA*.

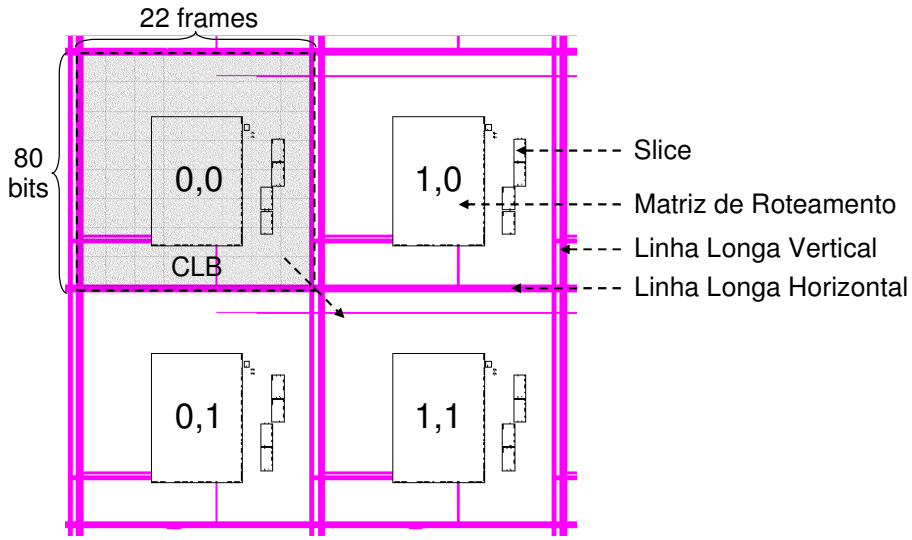


Figura 76 – Exemplo de realocação de uma CLB do dispositivo Virtex-II 1000.

4.10 Realocação de módulos

Existe ainda um passo opcional para a implementação de SDRs que é a realocação dos módulos reconfiguráveis do sistema (Figura 61M). Embora a ferramenta CoreUnifier-II Pro permita de forma gráfica facilmente realocar módulos de posição, conforme descrito na Seção anterior, fazer um projeto com esta característica requer um esforço de projeto ainda maior que fazer um projeto apenas reconfigurável. Quatro dificuldades adicionais sobre o fluxo de SDRs são citadas: (*i*) perda de um grau de liberdade no roteamento, não sendo mais possível que nenhum fio cruze a fronteira entre os módulos, pois o módulo realocado não possuirá o complemento do fio de outro módulo conforme o módulo original possuía; (*ii*) as interfaces dos dois módulos implementados em posições distintas do FPGA devem possuir interface idêntica, caso contrário a comunicação do módulo realocado com o restante do sistema será prejudicada; (*iii*) possibilidade de um módulo realocado não atender mais as características de temporização; (*iv*) um módulo não pode ser realocado para qualquer posição do FPGA, devido aos FPGAs Virtex-II e Virtex-II Pro possuírem uma arquitetura irregular (processadores PowerPC, BRAMs e blocos multiplicadores).

4.11 Conclusões

Este Capítulo apresentou um fluxo de projeto que busca simplificar a implementação de SDRs em relação a outros fluxos apresentados no Capítulo 3. A principal simplificação deste fluxo em relação a propostas anteriores ocorre durante a síntese do projeto, a qual, por ser baseada em parâmetros padrão, reporta menos erros durante a execução do fluxo. Uma fonte de problemas de fluxos para SDRs, como é o caso do Projeto Modular, é a utilização de macros baseadas em *tristates*. Este problema foi tratado no fluxo atual com a utilização de macros baseadas em LUTs, conforme as macros utilizadas por Dyer e Huebner. A ferramenta CoreUnifier-II Pro permite gerar bitstreams parciais pelo método de extração de *frames* e realocar módulos do sistema.

O fluxo apresentado ainda está distante de um fluxo ideal para SDRs. O principal problema enfrentado é a dificuldade em restringir o roteamento, já que os módulos são compostos por blocos lógicos e roteamento, e os blocos lógicos podem ter as suas posições restritas. Este controle do roteamento é necessário para que os módulos possam ser posicionados em posição escolhida do FPGA e posteriormente reconfigurados, enquanto o restante do sistema continua operando.

Uma tarefa passível de ser automatizada nos fluxos de SDR é a instanciação de macros no *top* do projeto. Como todas as conexões dos módulos reconfiguráveis (com exceção do sinal de relógio) devem ser feitas por macros, basta saber a direção da conexão e inserir a macro correspondente no meio.

5 ARTEMIS NETWORK ON CHIP

Este Capítulo apresenta a segunda contribuição deste trabalho: uma proposta de NoC que permita a reconfiguração parcial e dinâmica de módulos conectados a ela. Esta NoC, denominada Artemis, foi construída a partir da NoC Hermes, visando minimizar a necessidade de conectar um número elevado de módulos que não são usados simultaneamente, conseqüentemente reduzindo a área reservada para os roteadores da NoC, adicionar flexibilidade permitindo que novos módulos sejam carregados sem interromper o sistema como um todo e aumentar o desempenho do sistema utilizando módulos de hardware específicos para resolver uma computação solicitada.

O trabalho de Marescaux [MAR04], apresentado na Seção 2.3.3.1, possui três redes distintas: uma para tráfego de dados, uma para controle e uma exclusiva para reconfiguração. Por um lado, três redes distintas garantem uma melhor qualidade de comunicação e menor latência. Por outro lado, a sobreposição de redes de comunicação aumenta o número de fios entre os módulos do sistema, conseqüentemente aumentando a complexidade para a conexão de módulos reconfiguráveis e a área para o controle de cada uma destas redes em cada uma das interfaces dos módulos.

A proposta deste trabalho é utilizar a própria rede intra-chip de comunicação de dados para controlar também a reconfiguração. Para a implementação desta, a rede Hermes é utilizada como base, sendo sucintamente apresentada na Seção 5.1. Na seqüência, duas políticas de acesso a módulos reconfiguráveis são analisadas, na Seção 5.2. As modificações necessárias na rede Hermes para a conexão de módulos reconfiguráveis são apresentadas na Seção 5.3. Essa rede intra-chip modificada para permitir a substituição de módulos de hardware em tempo de execução foi denominada Artemis.

5.1 Hermes

A topologia de uma NoC é definida pela estrutura de conexão dos seus roteadores. Na topologia malha, utilizada neste trabalho, roteadores distintos possuem um número de portas diferentes, dependendo de sua posição na rede, como mostrado na Figura 77C. Por exemplo, o roteador central de uma NoC Hermes 3x3 possui cinco portas, conforme apresentado na Figura 77B. As cinco portas bidirecionais do roteador são: *East*, *West*, *North*, *South* e *Local*. Cada porta possui uma fila de entrada para o armazenamento temporário de *flits*[5]. A porta *Local* estabelece a comunicação entre o roteador e seu módulo local. As demais portas ligam o roteador aos roteadores vizinhos. O modo de chaveamento adotado pelo roteador da NoC Hermes é *wormhole*, onde cada pacote que trafega pela rede é passado *flit* a *flit* pelos canais físicos. O formato dos pacotes que trafegam pela rede é apresentado na Figura 77A. A lógica de controle implementa a *lógica de arbitragem* e o *algoritmo de roteamento*.

[5] Flit: menor quantidade de informação transmitida entre roteadores de uma NoC.

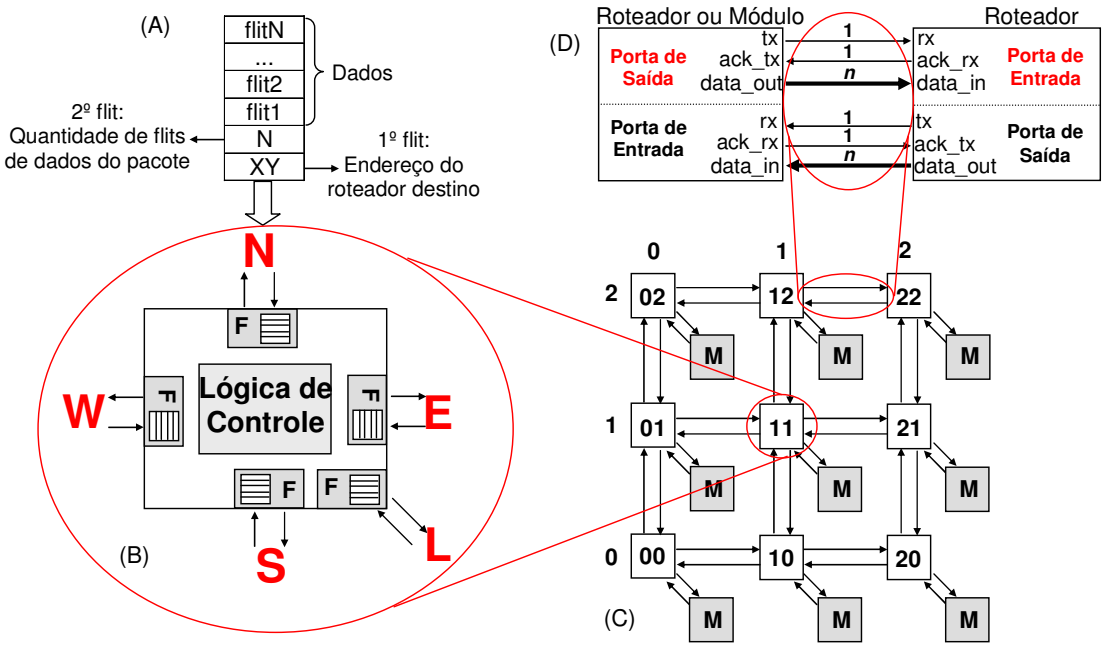


Figura 77 – Estrutura geral da NoC Hermes: (a) formato dos pacotes que trafegam pelos roteadores; (b) diagrama de blocos do roteador; (c) NoC Hermes 3x3; (d) interface roteador-roteador ou módulo-roteador.

Um roteador pode ser requisitado a estabelecer até 5 conexões simultâneas (uma conexão por porta de entrada do roteador). A *lógica de arbitragem* é utilizada para escolher qual das portas de entrada será roteada primeiro. Para isto, é utilizado um algoritmo de arbitragem dinâmica rotativa (round-robin), onde a prioridade de uma determinada porta é dada em função da última porta a ter a requisição de arbitragem atendida. Este algoritmo evita que uma mesma porta de entrada tenha sempre a maior prioridade para requisitar uma porta de saída, não permitindo acesso das outras portas de entrada às portas de saída.

O *algoritmo de roteamento* utilizado pelo roteador é o XY determinístico, no qual os pacotes trafegam pela rede sempre em uma mesma ordem: (*i*) trafegam pela rede no eixo X até chegar na coluna do roteador destino; (*ii*) trafegam pela rede no eixo Y até chegar na linha do roteador destino; (*iii*) estando no roteador destino, o pacote é encaminhado para a porta local para alcançar o módulo destino.

Quando um *flit* é bloqueado em um determinado roteador, o desempenho da rede é afetado, pois os *flits* do mesmo pacote podem ser bloqueados em roteadores intermediários da rede, dependendo dos tamanhos do pacote e das filas. O tamanho das filas utilizadas para armazenar *flits* bloqueados é parametrizável em tempo de projeto. Mais detalhes sobre a NoC Hermes podem ser obtidos em [MOR04].

5.2 Políticas de acesso a módulos reconfiguráveis

A posição de um dado módulo reconfigurável na NoC pode variar ao longo da execução do sistema. Isto ocorre porque estes saem do sistema para dar espaço a outros e até mesmo para melhorar o desempenho do sistema como um todo (e.g. reposicionar módulos origem e/ou destino

de forma que fiquem mais próximos quando estes se comunicam com freqüência ou quando transferem grande quantidade de informação). Para isso, é necessária uma entidade que gerencie a qual roteador um módulo reconfigurável vai ser conectado e que módulo reconfigurável pode ser retirado para a entrada de outro. A esta entidade dá-se o nome de Controlador de Configurações (CC).

Dois CCs foram propostos como trabalhos acadêmicos [BLO03] [CUR04], tendo sido estes analisados por Carvalho [CAR04]. Carvalho implementou um CC completamente em hardware chamado RSCM no escopo do mesmo grupo do Autor. Encontra-se em andamento a implementação de um CC em software [SOA05], para que seja feita a análise dos prós e contras em relação ao trabalho de Carvalho. Não pertence ao escopo deste trabalho conectar um CC ao SDR proposto, ficando esta atividade como uma sugestão de trabalho futuro. No entanto, este trabalho se preocupa em prover um suporte de hardware e de comunicação entre os módulos reconfiguráveis e o CC.

Logo, para efetuar um acesso a um módulo reconfigurável da rede deve-se acessar antes o CC para se determinar onde o módulo será configurado. A Figura 78 apresenta duas políticas de acesso ao CC para posterior acesso ao módulo reconfigurável.

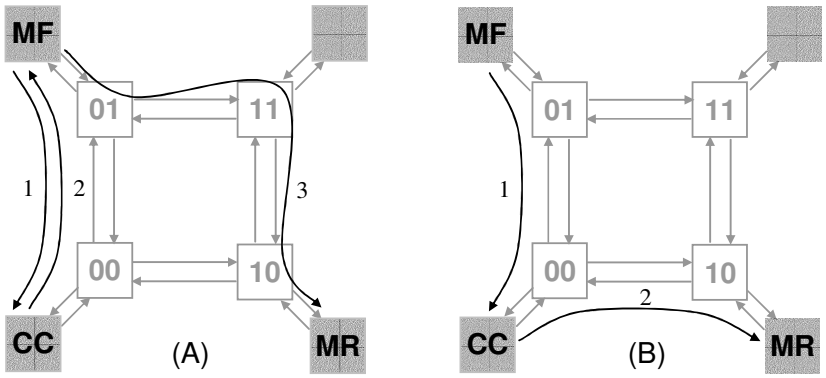


Figura 78 – Dois exemplos de políticas de acesso para comunicação de dados com módulos reconfiguráveis. MF representa um módulo fixo, MR representa um módulo reconfigurável e CC é o controlador de configurações. Em (a) o CC é acessado apenas para determinar o endereço de MR, em (b) o CC é acessado toda vez que MR for acessado.

Os passos sugeridos na Figura 78A são descritos a seguir:

1. MF, que deseja acessar MR, acessa CC para obter o endereço de MR.
2. CC identifica se o MR realmente se encontra configurado ou se o mesmo deve ser configurado e depois responde de volta para o MF o endereço do MR.
3. MF acessa diretamente MR.

O passo 3 se repete para cada acesso posterior de MF a MR.

A seguir são descritos os passos sugeridos na Figura 78B:

1. MF envia pacote destinado a MR para CC enviá-lo.

2. CC identifica se MR realmente se encontra configurado ou se o mesmo deve ser configurado e depois encaminha o pacote de MF para MR.

A política descrita na Figura 78B é a mais simples de ser implementada, pois quando deseja-se acessar um MR, o pacote é enviado para o CC e este gerencia qual é o endereço que corresponde ao MR. No entanto, esta política possui uma latência maior, pois o CC é sempre acessado antes de acessar um MR. A necessidade de sempre acessar o CC antes de acessar um MR pode criar também um gargalo no sistema quando existirem muitos MRs.

A política descrita na Figura 78A é a que mais se aproxima à política de comunicação já implementada na NoC, pois uma vez que o endereço de destino do pacote é fornecido pelo CC, os acessos a MRs são semelhantes aos acessos a MFs. Esta política permite efetuar a comunicação com MRs com uma baixa latência de comunicação, já que não é necessário acessar o CC sempre antes de acessar o MR. Em contraponto, cada módulo conectado ao meio de comunicação deve possuir uma tabela local que relacione o endereço dos MRs utilizados pelo módulo. Esta é uma tabela limitada, contendo apenas um subconjunto de registros da tabela total do CC. O número de registros existentes na tabela de CC é igual ao número de MRs do sistema, enquanto a tabela existente em cada módulo é parametrizável. A vantagem desta implementação é que um módulo pode acessar mais de um MR sem consultar o CC. A desvantagem é que criar tabelas nos módulos que podem acessar MRs pode trazer, além de um aumento significativo em área, a necessidade de controle da validade do endereço de MR.

5.3 Artemis

O objetivo da NoC Artemis é permitir o *controle da reconfiguração* de módulos conectados a ela pela mesma rede intra-chip na qual dados são transmitidos. Entende-se por *controle da reconfiguração* o processo de permitir que um módulo seja corretamente configurado sem prejudicar o funcionamento do restante do sistema que continua operando. Verificou-se durante a execução do presente trabalho, que a reconfiguração de um novo módulo resulta em *glitche*s para a entidade ao qual o módulo está conectado. No caso específico da NoC, este problema é percebido no sinal *rx* do roteador (ver Figura 77D na página 87), o qual erroneamente oscila algumas vezes devido ao processo de reconfiguração, levando o roteador a possíveis estados inválidos.

Para contornar este problema e permitir a reconfiguração de módulos conectados à NoC, quatro modificações no roteador da Hermes foram feitas: (*i*) suporte a pacotes de controle; (*ii*) acréscimo de sinais nas interfaces de E/S; (*iii*) inserção de lógica nas filas para receber/repassar pacotes de controle; (*iv*) descarte de pacotes de dados enviados para módulos reconfiguráveis durante a reconfiguração do módulo. Estas modificações são apresentadas na Seção 5.3.1. O mecanismo de isolamento do módulo reconfigurável conectado à porta local do roteador é apresentado na Seção 5.3.2. A Seção 5.3.3 apresenta a validação por simulação funcional da NoC Artemis. A prototipação da Artemis é realizada em uma série de estudos de caso no Capítulo 6.

5.3.1 Roteador da Artemis

O roteador da NoC Hermes é formado basicamente pela lógica de controle, pelas filas e pelas portas de E/S, como pode ser observado na Figura 77B na página 87. As modificações necessárias no roteador da Hermes são apresentadas a seguir e envolvem apenas as portas de E/S e as filas do roteador, não sendo necessárias alterações na lógica de controle. Para informações específicas sobre a lógica de controle o leitor é remetido a [MEL03].

5.3.1.1 Pacotes de controle

Para que a rede Artemis possa garantir o correto funcionamento da NoC em SDRs, *pacotes de controle* foram criados em complemento aos *pacotes de dados* suportados pela rede Hermes. Pacotes de controle são utilizados na Artemis para que os roteadores possam controlar a comunicação com o módulo reconfigurável conectado em sua porta local. Estes pacotes são identificados por sinais extras nas portas de entrada dos roteadores, detalhados na Seção seguinte. Dois serviços habilitados pela chegada de pacotes de controle foram adicionados aos roteadores de módulos reconfiguráveis da Artemis: os serviços de *estabelecimento* e *interrupção de comunicação* entre o roteador e o módulo reconfigurável. A Figura 79 apresenta estes pacotes.

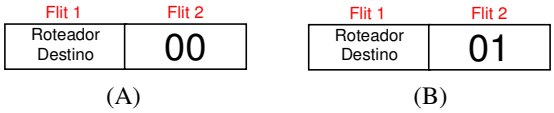


Figura 79 – Pacotes de controle da rede Artemis: (a) estabelecimento da comunicação entre roteador e módulo reconfigurável; (b) interrupção da comunicação entre roteador e módulo reconfigurável.

Pacotes de controle possuem apenas dois *flits*. O primeiro é o destino do pacote e o segundo é a operação que o roteador deve executar. Com esta estrutura de pacotes de controle a NoC pode diretamente suportar até $2^{(\text{tamanho do } flit \text{ em bits})}$ serviços disponibilizados pelos roteadores. No caso dos dois pacotes que controlam a reconfiguração apresentados na Figura 79, estes devem ser gerados por um módulo que executa o papel de um controlador de configurações (CC). Este CC pode ser interno ao sistema e estar implementado em hardware [CAR04], em software [SOA05] ou ser implementado externamente e fazer interface com o sistema através de um módulo conectado à NoC.

Uma vez que um pacote de interrupção da comunicação é recebido pelo roteador de um módulo reconfigurável, o sinal de *reconf* deste roteador, que é conectado a(s) macro(s) através dos sinais *ctrl*, é ativado. Quando isto ocorre a(s) macro(s) inibe(m) a passagem de dados. Depois da recepção de um pacote de restabelecimento da comunicação, o sinal *reconf* é desativado e a(s) macro(s) volta(m) a repassar dados para o módulo reconfigurável.

5.3.1.2 Interface Roteador-Roteador

Para diferenciar pacotes de controle de pacotes de dados foram adicionados nas portas dos roteadores os sinais *ctrl_in* e *ctrl_out*, conforme ilustrado na Figura 80. Desta forma, a cada *flit*

enviado por um roteador através dos sinais *data_out* e *tx*, adiciona-se o sinal *ctrl_out*, o qual informa se o *flit* é de controle ou de dados. O roteador destino do *flit* recebe *flits* de forma análoga pelos sinais *data_in*, *rx* e *ctrl_in*.

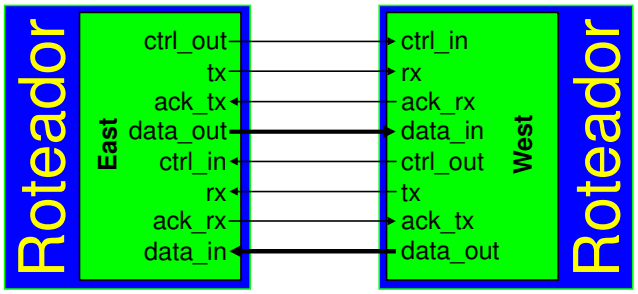


Figura 80 – Interface entre as portas East e West de dois roteadores da rede Artemis.

5.3.1.3 Lógica para recepção/repasse de pacotes de controle

Para que pacotes de controle cheguem ao roteador destino é necessário inserir lógica em cada uma das portas dos roteadores da NoC para repassar ou capturar os pacotes de controle. As primeiras tarefas do roteador destino são verificar o destino (primeiro *flit*) de um pacote recebido, e se este é de controle (*ctrl_in* = '1'). Caso o destino de um pacote de controle seja igual ao endereço do roteador atual, então este pacote será capturado pelo roteador atual. A chegada do segundo *flit* do pacote informa se será estabelecida (*data_in* = x"00") ou interrompida (*data_in* = x"01") a comunicação do roteador com o módulo reconfigurável. Caso o destino do pacote seja um endereço diferente do roteador atual, então este *flit* deve entrar na fila pré-existente (*fila de dados*) e uma nova fila criada no roteador da Artemis (*fila de controle*) deve armazenar no mesmo índice da fila de dados a informação de que aquele dado é de controle (*ctrl_in* = '1'). A fila de controle possui o mesmo tamanho da fila de dados, largura um bit e sempre opera com o mesmo índice da fila de dados. O segundo *flit* deste pacote, contendo a operação a ser efetuada pelo roteador destino, é igualmente inserido na fila de dados, inserindo também o valor recebido de *ctrl_in* na fila de controle. Uma vez que um pacote de controle se encontra na fila do roteador, o processo de repasse é feito de forma semelhante aos pacotes de dados, apenas levando em consideração que deve ser realizado o envio de *ctrl_out* com o valor armazenado na fila de controle, para cada *flit* enviado.

5.3.1.4 Controle de fluxo durante a reconfiguração

Independente da política de acesso a módulos reconfiguráveis utilizada, assunto tratado na Seção 5.2, deve-se garantir que nenhuma modificação feita no sistema para suportar reconfiguração parcial e dinâmica de módulos leve a NoC a um estado de bloqueio irrecuperável. O caso citado na Figura 78A é um destes exemplos. O problema ocorre quando MR está sendo reconfigurado e um pacote chega de um MF para o roteador do MR (passo 3 da Figura 78A), ficando este bloqueado porque o hardware do módulo reconfigurável ainda não está presente para receber os *flits* do pacote enviado. Neste caso duas questões devem ser analisadas: (1) este pacote foi destinado ao módulo que existia antes da reconfiguração ou ao módulo que está sendo configurado? (2) o que fazer com

o pacote recebido durante o processo de reconfiguração?

Quando um pacote de dados destinado à porta local do roteador for recebido durante o processo de reconfiguração, o pacote em questão fica aguardando na fila do roteador do módulo reconfigurável e, dependendo do tamanho do pacote, em outros roteadores intermediários da NoC. Se isto ocorrer, o pacote de controle enviado pelo CC não chegará até o roteador do módulo que foi configurado, jamais estabelecendo a conexão com o mesmo. Devido a este motivo e à dúvida de não saber se o pacote foi destinado ao módulo que estava previamente configurado ou ao módulo que está sendo configurado, decidiu-se *descartar* pacotes de dados recebidos pelo roteador de um módulo reconfigurável durante o processo de configuração de um módulo. Fica a cargo da política de acesso evitar que pacotes de dados não sejam descartados pelo roteador do módulo que está sendo reconfigurado, ou tratar deste caso usando e.g. uma estratégia de reenvio.

Um pacote deve ser descartado apenas quando três eventos simultâneos forem verificados: (*i*) o módulo local encontra-se em processo de reconfiguração (*reconf*='1'); (*ii*) o pacote recebido é destinado ao módulo conectado à porta local do roteador (*data_in=address*, onde *address* é o endereço XY do roteador atual na NoC e *data_in* é o primeiro *flit* do pacote que indica o destino do mesmo); (*iii*) o pacote é de dados (*ctrl_in*='0'). Uma vez detectado que o pacote deve ser descartado na chegada do primeiro *flit*, espera-se a chegada do segundo *flit*, o qual possui a informação de quantos *flits* ainda serão recebidos. Esta informação é utilizada para inicializar um contador que é decrementado a cada *flit* recebido, onde cada *flit* é dito "descartado" por não ser inserido na fila de entrada de forma a ser repassado para o módulo local.

5.3.2 Interface Roteador-Módulo

A interface entre o roteador e o módulo reconfigurável é apresentada na Figura 81. Foram necessárias duas macros LR para conectar os 11 bits da interface do roteador com o módulo reconfigurável e duas macros RLC para conectar os outros 10 bits de interface no sentido inverso de comunicação.

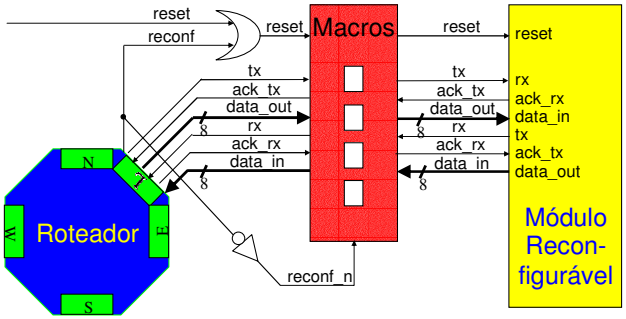


Figura 81 – Interface da rede Artemis com um módulo reconfigurável passando através das macros propostas. No estudo de caso implementado duas macros LR e duas macros RLC foram utilizadas para comunicar os 21 bits de interface do módulo reconfigurável com a Artemis.

Na interface entre o roteador e um módulo reconfigurável percebe-se que os sinais *reconf* e *reset* foram adicionados e os sinais *ctrl_in* e *ctrl_out* foram suprimidos, quando comparado à

interface roteador-roteador apresentada na Figura 80. Os sinais *ctrl_in* e *ctrl_out* foram suprimidos da interface entre o roteador e o módulo reconfigurável porque módulos reconfiguráveis não enviam nem recebem pacotes de controle, sendo estes pacotes consumidos pelo roteador. O *reset* é um sinal global que deve ser conectado apenas aos módulos, sendo utilizado para colocar as máquinas de estados dos módulos no estado inicial. O sinal *reconf* é utilizado, assim como *reset*, para inicializar as máquinas de estados, com a exceção que o sinal de *reconf* inicializa apenas o módulo local do roteador a ele conectado. O sinal de *reconf* negado da Figura 81 é conectado ao sinal *controle* da macro RLC, conforme apresentado na Figura 64B (pág. 71), permitindo a passagem de dados do módulo reconfigurável para o roteador apenas quando o módulo não estiver sendo reconfigurado.

5.3.3 Validação da NoC Artemis

A rede Artemis foi primeiramente validada através de simulação funcional. A Figura 82 apresenta, por meio de formas de onda, a validação do processo de interrupção da comunicação com um módulo reconfigurável, e o posterior restabelecimento desta comunicação após a reconfiguração. Os passos apresentados a seguir possuem correspondência com a numeração indicada na Figura 82, e pressupõem o uso do sistema descrito na Figura 78.

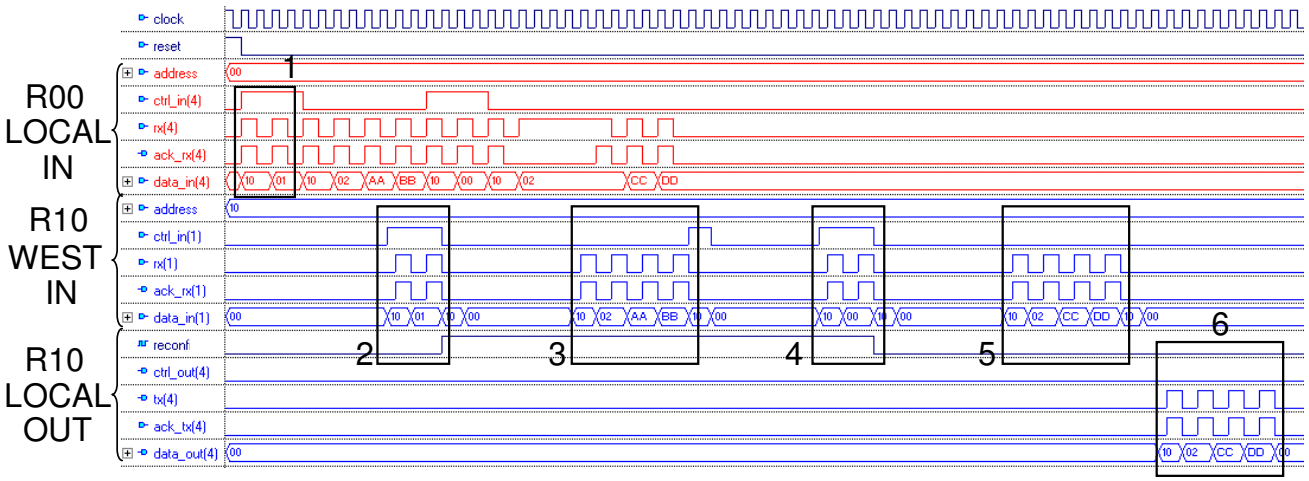


Figura 82 – Simulação da rede Artemis, mostrando o tráfego de pacotes de controle para interromper / restabelecer conexão com os módulos reconfiguráveis e de pacotes de dados repassados ou descartados.

1. Envio de pacote de controle, requisitando a interrupção da comunicação do roteador 10 com o seu módulo reconfigurável. Este pacote foi originado pelo CC, que está conectado na porta local do roteador 00.
2. Chegada do pacote de interrupção de comunicação com o módulo reconfigurável no roteador 10. Roteador ativa o sinal de *reconf* informando à macro que não mais repasse os sinais de saída do módulo reconfigurável para o roteador.
3. Teste de chegada de um pacote de dados durante a reconfiguração. Este pacote é totalmente descartado, conforme discutido na Seção 5.3.1.4. O posterior tratamento de outros pacotes por

parte do roteador indica que o mesmo garante que pacotes recebidos durante o processo de reconfiguração não o bloqueiam.

4. Chegada do pacote de restabelecimento de comunicação com o módulo reconfigurável. Roteador desativa sinal de *reconf*, informando à macro que volte a repassar os sinais de saída do módulo reconfigurável para o roteador.
5. Chegada de pacote de dados após a reconfiguração do módulo reconfigurável.
6. Envio de pacote de dados do roteador 10 para o módulo reconfigurável.

5.4 Conclusões

Poucas modificações foram necessárias na rede Hermes para habilitar a reconfiguração parcial e dinâmica de módulos conectados a ela. Dentre as modificações, a primeira refere-se à implementação de dois tipos de pacotes de controle, os quais não são repassados ao módulo conectado à porta local, e sim processados diretamente pelo roteador. No contexto deste trabalho, os pacotes de controle foram utilizados para isolar o módulo antes da reconfiguração e reativar a comunicação após a reconfiguração. Em trabalhos futuros, estes pacotes poderão ser utilizados para configurar o roteador, ou recolher estatísticas de desempenho, por exemplo. A segunda modificação a ressaltar é habilitar o descarte de pacotes durante a reconfiguração de um módulo, pois pacotes de dados direcionados a este último seriam bloqueados pelo roteador, podendo colocar a rede em situação de bloqueio permanente.

6 ESTUDOS DE CASO

Este Capítulo apresenta uma série de estudos de caso para validar o fluxo de projeto de SDRs proposto no Capítulo 4 e a NoC Artemis, proposta no Capítulo 5. A Seção 6.1 primeiramente apresenta o ambiente de hardware e software utilizado para validar os estudos de caso apresentados nas Seções seguintes. A Seção 6.1 ainda provê uma introdução sobre os módulos de hardware utilizados nos estudos de casos. O estudo de caso da calculadora reconfigurável é apresentado na Seção 6.2, com o objetivo de seguir passo a passo o fluxo de projeto proposto. A Seção 6.3 possui a finalidade de mostrar que o fluxo proposto é robusto, no sentido de permitir a construção de interfaces entre um barramento e um módulo reconfigurável com mais de 100 sinais. A Seção 6.4 apresenta a conexão dos módulos reconfiguráveis de multiplicação, divisão e raiz quadrada à NoC Artemis. Um sistema multiprocessado que permite intercambiar módulos processadores de 32 bits por processadores de 16 bits é apresentado na Seção 6.5. As Seções 6.6 e 6.7 apresentam sistemas realocáveis vertical e horizontalmente, respectivamente.

6.1 Ambiente

Nesta Seção apresenta-se o ambiente de hardware e software utilizado para desenvolver e validar o presente trabalho. Primeiro, descreve-se as plataformas de desenvolvimento utilizadas e os módulos de hardware que nelas foram prototipados. Em seguida, descreve-se quais softwares comerciais foram empregados e os aplicativos implementados para habilitar o desenvolvimento dos estudos de caso.

6.1.1 Hardware

Os testes iniciais do fluxo de projeto proposto, módulos reconfiguráveis conectados à NoC Artemis e de realocação, foram todos executados em uma plataforma V2MB1000 da Memec Insight. Esta plataforma possui um FPGA Virtex-II 1000 e suporte a módulos de extensão padrão da Memec denominados P160. Utilizou-se o módulo de prototipação para estender as interfaces de E/S da plataforma principal para a conexão em analisadores lógicos da Agilent e HP. As reconfigurações foram efetuadas através do cabo Parallel Cable IV da Xilinx.

À medida que os estudos de caso foram se tornando mais complexos, migrou-se para uma plataforma maior, a Xilinx University Program Virtex-II Pro Development System. Esta possui um dispositivo Virtex-II Pro XC2VP30 e foi reconfigurada através de um cabo USB.

As próximas Seções apresentam os módulos de hardware utilizados nos sistemas implementados. É importante ressaltar que todos os módulos foram desenvolvidos no escopo do GAPH.

6.1.1.1 Módulo serial

O módulo serial é responsável por prover comunicação entre o sistema do usuário em um computador hospedeiro e os módulos do sistema interconectados através da NoC. Esta comunicação é realizada através do protocolo padrão RS-232C. A Figura 83 apresenta a interface externa do módulo serial.

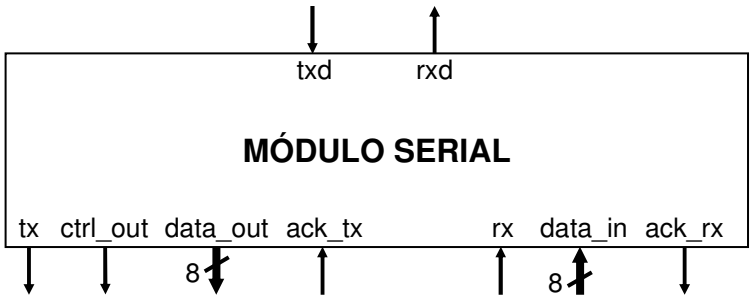


Figura 83 – Interface externa do módulo serial.

Os sinais apresentados na parte superior da Figura 83 conectam o módulo ao computador hospedeiro, enquanto os sinais da parte inferior conectam o módulo com a NoC. A função de cada sinal é:

- *clock*: sinal de relógio do sistema (não explicitado na Figura 83);
- *reset*: sinal de inicialização do sistema (não explicitado na Figura 83);
- *txd*: recebe dados do computador hospedeiro bit-a-bit;
- *rxd*: envia dados para o computador hospedeiro bit-a-bit;
- *tx*, *ctrl_out*, *data_out*, *ack_tx*, *rx*, *data_in*, *ack_rx*: interface com a NoC.

Além dos sinais de interface padrão do protocolo da NoC descritos na Figura 77D (pág. 87), pode-se perceber o sinal *ctrl_out*. Este habilita que o módulo serial envie pacotes de controle para os módulos conectados à Artemis, permitindo controlar externamente o estabelecimento ou a interrupção da comunicação com módulos reconfiguráveis.

A primeira tarefa do módulo serial é ajustar a transmissão/recepção da taxa de transferência de dados do computador hospedeiro com a taxa de transmissão/recepção do módulo serial. Isto é feito enviando um byte 55h do computador hospedeiro para o módulo serial. Este procedimento precisa ser efetuado cada vez que o sistema for inicializado. Após a inicialização, o módulo espera por dados do computador hospedeiro ou da NoC, transmitindo o dado recebido para o lado oposto.

O módulo serial ocupa em média 284 *slices* de um dispositivo Virtex-II (Pro).

6.1.1.2 Módulo R8

O processador R8 [MOR03a] é um processador 16 bits *load-store* com arquitetura Von Neumann (memória de instrução e dados unificada), com CPI (Ciclos Por Instrucao) entre 2 e 4. O conjunto completo de instruções do processador é apresentado na Tabela 12. O bloco de dados

contém 16 registradores de propósito geral, registrador de instruções (IR), contador de programa (PC), ponteiro para pilha (SP), e 4 flags de estado (negativo, zero, vai-um e transbordo).

A interface externa do módulo R8, assim como os seus principais blocos internos, são apresentados na Figura 84. O módulo R8 inclui o processador R8 e uma memória cache de 4K palavras de 16 bits.

Tabela 12 – Conjunto de instruções do processador R8.

INSTRUÇÃO	FORMATO DA INSTRUÇÃO				SEMÂNTICA
	15 – 12	11 - 8	7 - 4	3 - 0	
ADD Rt, Rs1,Rs2	**0**	R target	R source1	R source2	Rt ← Rs1 + Rs2 ; Inz ; Icv
SUB Rt, Rs1,Rs2	**1**	R target	R source1	R source2	Rt ← Rs1 - Rs2 ; Inz ; Icv
AND Rt, Rs1,Rs2	**2**	R target	R source1	R source2	Rt ← Rs1 *and* Rs2 ; Inz
OR Rt, Rs1,Rs2	**3**	R target	R source1	R source2	Rt ← Rs1 *or* Rs2 ; Inz
XOR Rt, Rs1,Rs2	**4**	R target	R source1	R source2	Rt ← Rs1 *xo* Rs2 ; Inz
ADDI Rt, *cte8*	**5**	R target	Constante		Rt ← Rt + ("00000000" & constante) ; Inz ; Icv
SUBI Rt, *cte8*	**6**	R target	Constante		Rt ← Rt - ("00000000" & constante) ; Inz ; Icv
LDL Rt, *cte8*	**7**	R target	Constante		Rt ← RtH & constante
LDH Rt, *cte8*	**8**	R target	Constante		Rt ← constante & RtL
LD Rt, Rs1,Rs2	**9**	R target	R source1	R source2	Rt ← MEM (Rs1+Rs2)
ST Rt, Rs1,Rs2	**A**	R target	R source1	R source2	MEM (Rs1+Rs2) ← Rt
NOT Rt, Rs1	**B**	R target	R source1	**4**	Rt ← not (Rs1) ; Inz
NOP	**B**	0	0	**5**	Sem ação
HALT	**B**	0	0	**6**	Suspende o fluxo de busca e execução de instruções
LDSP Rs1	**B**	0	R source1	**7**	SP ← Rs1 (inicializar o ponteiro da pilha)
RTS	**B**	0	0	**8**	PC ←MEM(SP+1) ; SP←SP+1
POP Rt	**B**	R target	0	**9**	Rt ←MEM(SP+1) ; SP←SP+1
PUSH Rt	**B**	R target	0	**A**	MEM(SP)←Rt; SP←SP-1
JMPR Rs1	**C**	0	R source1	**0**	PC ← PC + Rs1 (não depende de nenhum flag)
JMPNR Rs1	**C**	0	R source1	**1**	if (*n* =1) PC ← PC + Rs1
JMPZR Rs1	**C**	0	R source1	**2**	if (*z* =1) PC ← PC + Rs1
JMPCR Rs1	**C**	0	R source1	**3**	if (*c* =1) PC ← PC + Rs1
JMPVR Rs1	**C**	0	R source1	**4**	if (*v* =1) PC ← PC + Rs1
JMP Rs1	**C**	0	R source1	**5**	PC ← Rs1 (não depende de nenhum flag)
JMPN Rs1	**C**	0	R source1	**6**	if (*n* =1) PC ← Rs1
JMPZ Rs1	**C**	0	R source1	**7**	if (z =1) PC ← Rs1
JMPC Rs1	**C**	0	R source1	**8**	if (*c* =1) PC ← Rs1
JMPV Rs1	**C**	0	R source1	**9**	if (*v* =1) PC ← Rs1
JSRR Rs1	**C**	0	R source1	**A**	MEM(SP)←PC; SP←SP-1; PC← PC+ Rs1
JSR Rs1	**C**	0	R source1	**B**	MEM(SP)←PC; SP←SP-1; PC← Rs1
JMPD deslocamento	**D**	**0**	Deslocamento (10 bits)		PC ← PC + sinal_ext & deslocamento
JMPND deslocamento	**E**	**0**	Deslocamento (10 bits)		if (n =1) PC ← PC + sinal_ext & deslocamento
JMPZD deslocamento	**E**	**1**	Deslocamento (10 bits)		if (z=1) PC ← PC + sinal_ext & deslocamento
JMPCD deslocamento	**E**	**2**	Deslocamento (10 bits)		if (c =1) PC ← PC + sinal_ext & deslocamento
JMPVD deslocamento	**E**	**3**	Deslocamento (10 bits)		if (v =1) PC ← PC + sinal_ext & deslocamento
JSRD deslocamento	**F**	Deslocamento (12 bits)			MEM(SP)←PC; SP←SP-1; PC← PC + sinal_ext & deslocamento

No módulo R8 existe ainda uma lógica de controle que comanda a execução do processador R8, colocando o mesmo em estado de espera toda a vez que o processador executa uma instrução de *load-store* (ver sinal *waitR8* na Figura 84). Operações de *load-store* podem acessar: (*i*) a cache local ao processador, a cache de outro processador ou uma memória remota; (*ii*) dispositivos de E/S; (*iii*) outros processadores, por motivos de sincronização. Estes três modos de acesso são detalhados a seguir.

Para determinar qual elemento de armazenamento do sistema (cache local, cache remota, memória) o processador R8 está acessando através de instruções *load-store*, faixas de endereçamento foram definidas para cada tipo de elemento de armazenamento. A Figura 85 apresenta um exemplo das faixas de endereçamento de um sistema que contém dois processadores e uma memória.

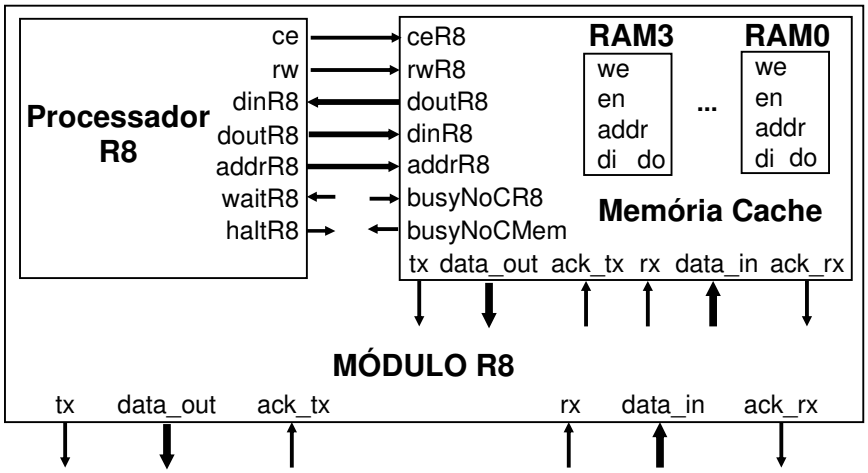


Figura 84 – Diagrama de blocos do módulo processador R8.

```
01 if(address>=0 && address<4096 ){
02         globalAddress = address;
03         addressCore=1; //cache local
04 }
05 else if(address>=4096 && address<8192){
06         globalAddress = 4096 - address;
07         addressCore=2; //cache remota
08 }
09 else if(address>=8192 && address<12288){
10         globalAddress = 8192 - address;
11         addressCore=3; //memória
12 }
```

Figura 85 – Código em linguagem C ilustrando o acesso a cache local/remota ou a uma memória.

As operações de E/S são mapeadas no endereço de memória FFFFh. Portanto, quando uma instrução de *store* é executada neste endereço, uma operação de *printf* é realizada pelo módulo de E/S do sistema (módulo Serial no presente trabalho), e quando uma operação de *load* é executada no mesmo endereço uma operação de *scanf* é realizada.

Em sistemas multiprocessados um mecanismo de sincronização é necessário para implementar aplicações distribuídas. Esta sincronização foi realizada no presente trabalho através de comandos *wait* e *notify*, ambos mapeados em memória [MEL05]. O comando *wait* é responsável por bloquear a execução do processador até que um comando de *notify* seja recebido. O comando *wait* é identificado através de uma instrução de *store* no endereço FFFEh com o número do processador que vai reiniciar a execução do processador com um comando de *notify*. O comando de notify é identificado através de uma instrução de store no endereço FFFDh com o número do processador que vai ser reiniciado.

O módulo R8 ocupa em média 502 *slices* de um dispositivo Virtex-II (Pro).

6.1.1.3 Módulo MR2

O processador MR2 [CAL04] implementa um subconjunto das instruções da arquitetura MIPS [PAT00]. O processador MR2 é um processador 32 bits *load-store* com arquitetura Harvard. Ele possui um banco de registradores de uso geral de 32 posições, denominados de $0 a $31. O registrador $0 não é realmente um registrador, mas sim a constante 0, disponível para uso em

instruções que necessitem usar este valor. O conjunto completo de instruções do processador é apresentado na Tabela 13.

A interface externa do módulo MR2, assim como os seus principais blocos internos, são apresentados na Figura 86. O módulo MR2 inclui o processador MR2 e uma memória cache de 2K palavras de 32 bits, sendo 1K palavras reservada para instruções e 1K palavras para dados.

Tabela 13 – Conjunto de instruções do processador MR2.

Instrução	FORMATO DA INSTRUÇÃO						AÇÃO
	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0	
ADDU Rd, Rs, Rt	00	Rs	Rt	Rd	00	21	Rd← Rs + Rt
SUBU Rd, Rs, Rt	00	Rs	Rt	Rd	00	23	Rd ← Rs – Rt
AND Rd, Rs, Rt	00	Rs	Rt	Rd	00	24	Rd ← Rs and Rt
OR Rd, Rs, Rt	00	Rs	Rt	Rd	00	25	Rd ← Rs or Rt
XOR Rd, Rs, Rt	00	Rs	Rt	Rd	00	26	Rd ← Rs xor Rt
SLL Rd, Rt, shamt	00	00	Rt	Rd	shamt	00	Rd ← Rt deslocado shamt bits à esquerda (0s à direita)
SRL Rd, Rt, shamt	00	00	Rt	Rd	shamt	02	Rd ← Rt deslocado shamt bits à direita (0s à esquerda)
ADDIU Rt, Rs, Imed16	09	Rs	Rt	Imed16			Rt ← Rs + (Imed16 com sinal estendido)
ANDI Rt, Rs, Imed16	0C	Rs	Rt	Imed16			Rt ← Rs and 0x0000 & (Imed16)
ORI Rt, Rs, Imed16	0D	Rs	Rt	Imed16			Rt ← Rs or 0x0000 & (Imed16)
XORI Rt, Rs, Imed16	0E	Rs	Rt	Imed16			Rt ← Rs xor 0x0000 & (Imed16)
LUI Rt, Imed16	0F	0	Rt	Imed16			Rt ← (Imed16 & 0x0000)
LBU Rt, Imed16(Rs)	20	Rs	Rt	Imed16			Rt ← 0x000000 & PMEMD(Imed16 com sinal estendido+Rs)
LW Rt, Imed16(Rs)	23	Rs	Rt	Imed16			Rt ← PMEMD(Imed16 com sinal estendido+Rs) (4 bytes)
SB Rt, Imed16(Rs)	28	Rs	Rt	Imed16			PMEMD(Imed16 com sinal estendido+Rs) ← Rt [7:0] (1 byte)
SW Rt, Imed16(Rs)	2B	Rs	Rt	Imed16			PMEMD(Imed16 com sinal estendido+Rs) ← Rt (4 bytes)
SLT Rd, Rs, Rt	00	Rs	Rt	Rd	00	2A	Rd ← 1 se Rs menor que Rt (c/sinal), senão Rd ← 0
SLTU Rd, Rs, Rt	00	Rs	Rt	Rd	00	2B	Rd ← 1 se Rs menor que Rt (s/sinal), senão Rd ← 0
SLTI Rt, Rs, Imed16	0A	Rs	Rt	Imed16			Rt ← 1 se Rs menor que Imed16 (c/sinal), senão Rt ← 0
SLTIU Rt, Rs, Imed16	0B	Rs	Rt	Imed16			Rt ← 1 se Rs menor que Imed16 (s/sinal), senão Rt ← 0
BEQ Rs, Rt, rótulo	04	Rs	Rt	Imed16			PC ← PC + (Imed16 & "00" com sinal estendido), se Rs=Rt
BGEZ Rs, rótulo	01	Rs	01	Imed16			PC ← PC + (Imed16 & "00" com sinal estendido), se Rs=>0
BLEZ Rs, rótulo	06	Rs	00	Imed16			PC ← PC + (Imed16 & "00" com sinal estendido), se Rs<=0
BNE Rs, Rt, rótulo	05	Rs	Rt	Imed16			PC ← PC + (Imed16 & "00" com sinal estendido), se Rs≠Rt
J rótulo	02	Imed26					PC ← PC[31:28] & Imed26 & "00"
JALR Rd, Rs	00	Rs	00	Rd	00	09	Rd ← NPC; PC ← Rs
JR Rs	00	Rs	0000			08	PC ← Rs

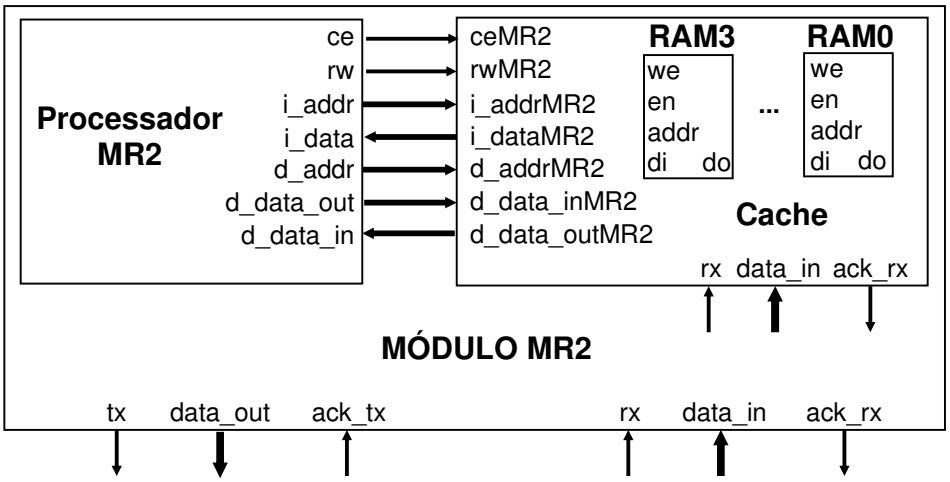


Figura 86 – Diagrama de blocos do módulo processador MR2.

O porte do processador MR2 para a NoC e a criação de drivers para comunicação com outros módulos da NoC também são contribuições do presente trabalho. No processador R8 drivers

não são necessários, pois uma vez que uma operação mapeada em memória é executada, um hardware específico se responsabiliza por montar um pacote e enviar a outro módulo da NoC. Se por um lado, esta abordagem permite rapidamente liberar o processador em chamadas não bloqueantes, por outro lado ela se torna extremamente custosa em hardware à medida que novas operações com novos módulos são adicionadas a NoC. Por este motivo decidiu-se implementar a comunicação com outros módulos da NoC por software. Neste caso, os drivers possuem toda a lógica de envio e recebimento de pacotes para a comunicação com outros módulos da NoC. O Anexo VI apresenta o código completo dos drivers criados.

Com as modificações efetuadas no módulo MR2, as únicas operações executadas por hardware são: enviar *flit*, receber *flit* e escrever *flit* na cache. A escrita na cache foi a única operação que não foi possível implementar em software, pois quando o sistema é inicializado ele não possui o software para preencher a sua cache de instruções com a aplicação a ser executada. Espera-se no futuro utilizar a reconfiguração parcial para inicializar a cache de instruções da MR2, tornando opcional o hardware hoje existente para este fim.

O módulo MR2 ocupa em média 1021 *slices* de um dispositivo Virtex-II (Pro).

6.1.1.4 NoC

A NoC Artemis foi modificada a partir da NoC Hermes utilizada na MultiNoC [MEL05]. A MultiNoC é uma plataforma biprocessada otimizada para baixo consumo de área, tendo sido esta prototipada em um dispositivo Spartan-II com 200 mil portas lógicas equivalentes junto a dois processadores R8, um módulo serial e um módulo de memória. Esta NoC é uma rede malha 2×2, com *flits* de 8 bits e filas de armazenamento temporário para 4 *flits*.

A NoC Artemis com a configuração apresentada ocupa em média 1360 *slices* de um dispositivo Virtex-II (Pro).

6.1.1.5 Multiplicação

O módulo de multiplicação recebe como entrada dois operandos de 32 bits, provendo uma resposta de 64 bits. A Figura 87 apresenta os quatro formatos de pacotes utilizados para interagir com o módulo de multiplicação. Os pacotes A e B são utilizados para enviar os dois operandos de 32 bits e os pacotes C e D são utilizados para recuperar os 64 bits do resultado (um pacote é para recuperar os 32 bits mais significativos da resposta, outro é para recuperar os bits menos significativos).

O módulo de multiplicação ocupa em média 260 *slices* de um dispositivo Virtex-II (Pro).

6.1.1.6 Divisão

O módulo de divisão divide um operando de 32 bits por outro de 32 bits, resultando em 32 bits de resposta e 32 bits de resto da divisão inteira. A Figura 88 apresenta os quatro formatos de pacotes utilizados para interagir com o módulo de divisão. O pacote A é utilizado para enviar o dividendo da operação, o pacote B é utilizado para enviar o divisor, o pacote C é utilizado para

recuperar o quociente e o pacote D para recuperar o resto da divisão inteira.

O módulo de divisão ocupa em média 254 *slices* de um dispositivo Virtex-II (Pro).

6.1.1.7 Raiz Quadrada

O módulo de raiz quadrada calcula a raiz quadrada de um operando de 64 bits, provendo uma resposta de 32 bits. A Figura 89 apresenta os três formatos de pacotes utilizados para interagir com o módulo de raiz quadrada. Os pacotes A e B são utilizados para enviar, respectivamente, a parte alta e parte baixa do operando a extrair a raiz quadrada, enquanto o pacote C é utilizado para recuperar o resultado da operação.

O módulo de raiz quadrada ocupa em média 343 *slices* de um dispositivo Virtex-II (Pro).

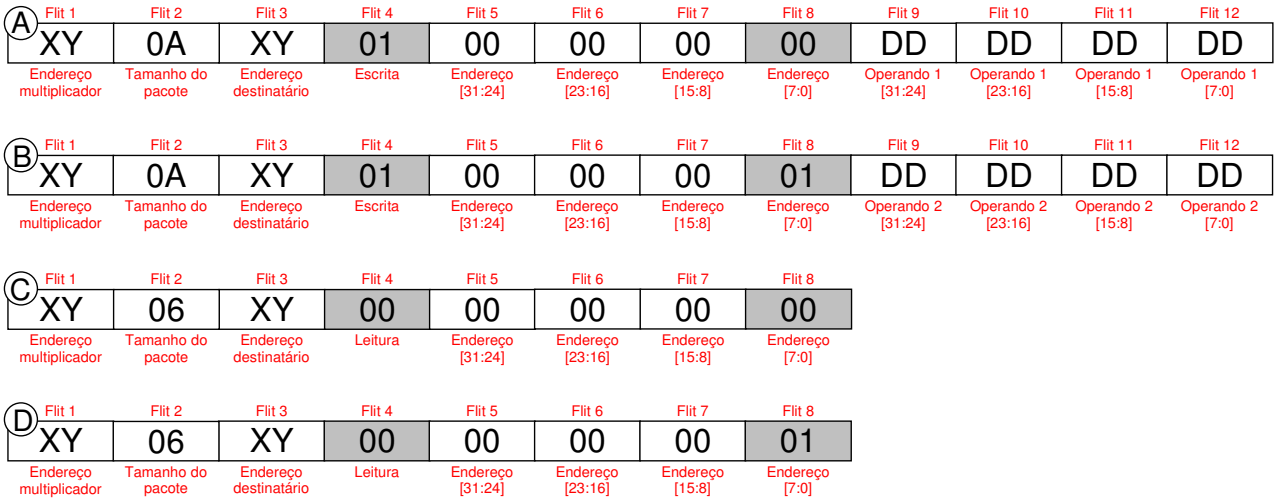


Figura 87 – Formatos de pacotes utilizados pelo módulo de multiplicação.

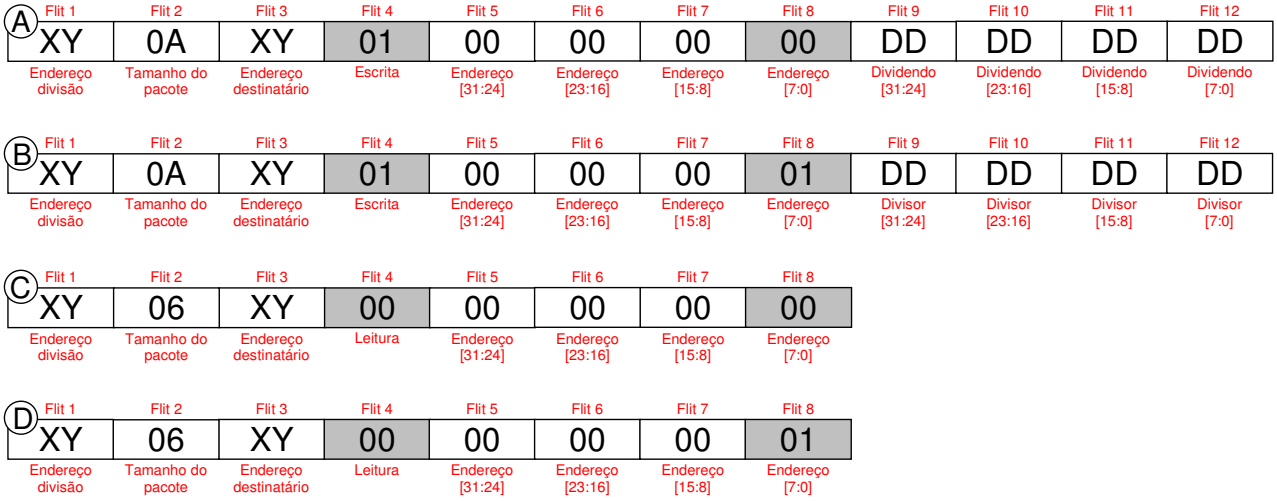


Figura 88 – Formatos de pacotes utilizados pelo módulo de divisão.

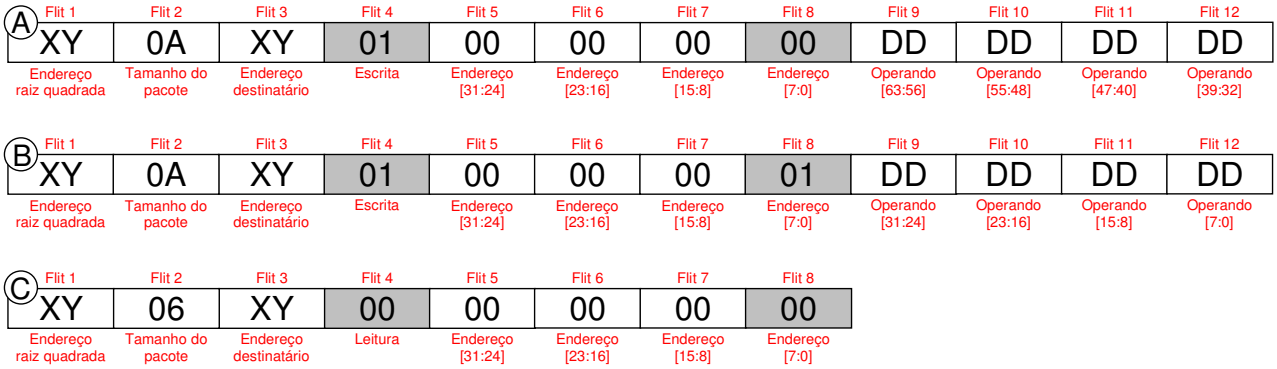


Figura 89 – Formatos de pacotes utilizados pelo módulo de raiz quadrada.

6.1.2 Software

Além dos softwares para o fluxo de projeto apresentados no Capítulo 4 (CoreUnifier-II Pro, XDLAnalyzer, FPGA Editor e BitGen), outros softwares adicionais são utilizados com a finalidade de obter o SDR funcionando na plataforma de prototipação. As Seções 6.1.2.1 e 6.1.2.2 apresentam, respectivamente, os simuladores dos processadores R8 e MR2. A Seção 6.1.2.3 apresenta a ferramenta Packer, responsável por preparar os códigos objetos dos processadores para serem enviados para um simulador HDL ou para um computador hospedeiro através da interface serial. A Seção 6.1.2.4 apresenta o software utilizado para comunicar o computador hospedeiro com o SDR executado na plataforma de prototipação.

6.1.2.1 Simulador R8

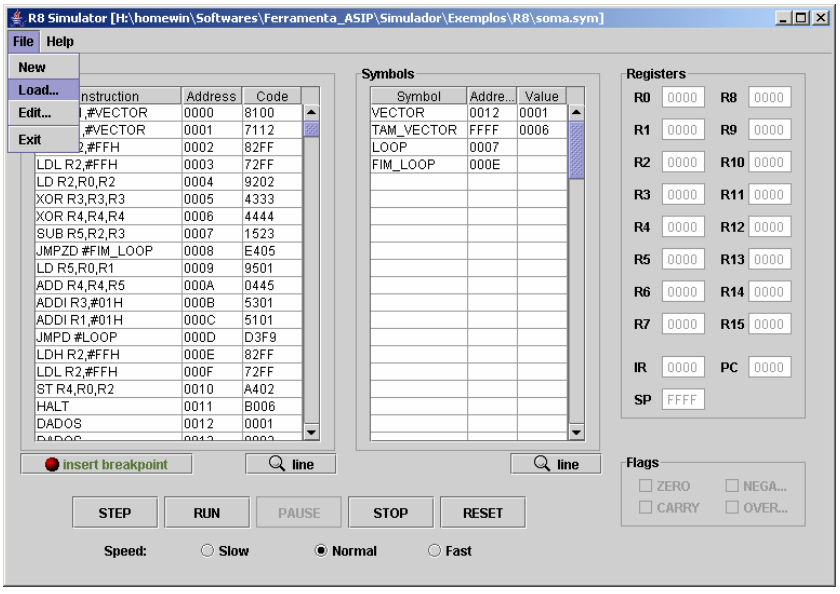


Figura 90 – Interface gráfica do R8 Simulator.

O código em linguagem de montagem para o processador R8 deve ser escrito conforme o seu conjunto de instruções apresentado na Tabela 12. A seguir o R8 Simulator [GAP04a], apresentado na Figura 90, permite simular o código desenvolvido e automaticamente gerar o código

objeto da aplicação. Infelizmente, o R8 Simulator não permite simular uma aplicação multiprocessada, sendo desconsideradas as instruções que se comunicam com outros processadores.

6.1.2.2 SPIM

O código assembly para o processador MR2 deve ser escrito conforme o seu conjunto de instruções apresentado na Tabela 13. A seguir o SPIM [LAU05], apresentado na Figura 91, permite simular o código desenvolvido e automaticamente gerar o código objeto da aplicação. Infelizmente, o SPIM, assim como o R8 Simulator, não permite simular uma aplicação multiprocessada, sendo desconsideradas as instruções que se comunicam com outros processadores.

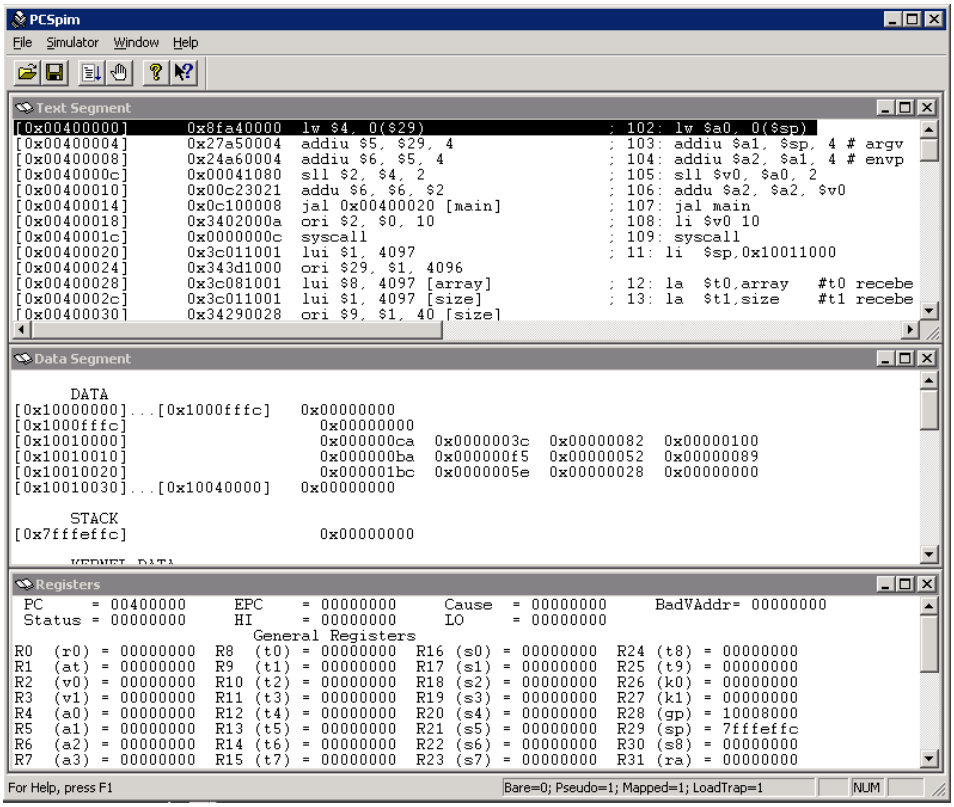


Figura 91 – Interface gráfica do SPIM.

6.1.2.3 Packer

Após ter sido gerado o código objeto pelo simulador SPIM na Seção anterior, o código objeto precisa ser transformado em pacotes para que estes sejam enviados através da NoC até as caches dos processadores do SDR. A Figura 92 apresenta a interface gráfica da ferramenta Packer [GAP05], responsável por quebrar códigos objetos em pacotes. Um campo que deve ser obrigatoriamente preenchido, além dos arquivos de origem e destino dos pacotes, é qual o endereço XY na NoC do processador MR2 que está sendo carregado. Pela interface gráfica da ferramenta também é possível selecionar se o arquivo de saída vai ser utilizado por um simulador HDL (opção TB NoC na Figura 92) ou se o arquivo será enviado para o SDR a partir de uma interface de E/S do computador hospedeiro (opção Serial na Figura 92). Os campos restantes são opcionais.

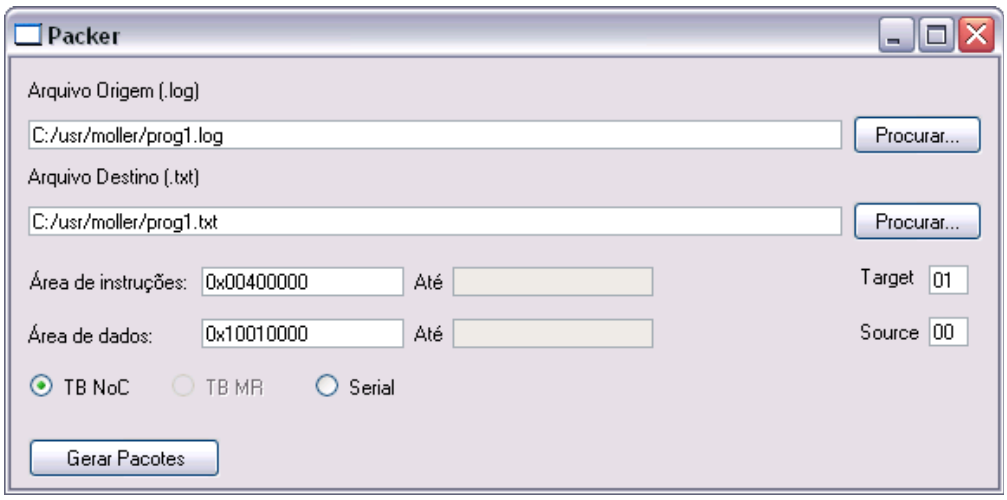


Figura 92 – Interface gráfica do Packer.

Até o presente momento não existe suporte da ferramenta Packer para os códigos objetos do processador R8, devendo o processo de geração de pacotes ser efetuado manualmente.

6.1.2.4 SerialApp

O SerialApp [GAP04b] provê acesso à plataforma de prototipação através da interface serial RS-232C de um computador. A partir dele é possível configurar as caches de dados e instruções dos processadores, disparar a execução dos processadores e efetuar operações de E/S dos módulos do sistema. A Figura 93 apresenta a interface gráfica do SerialApp.

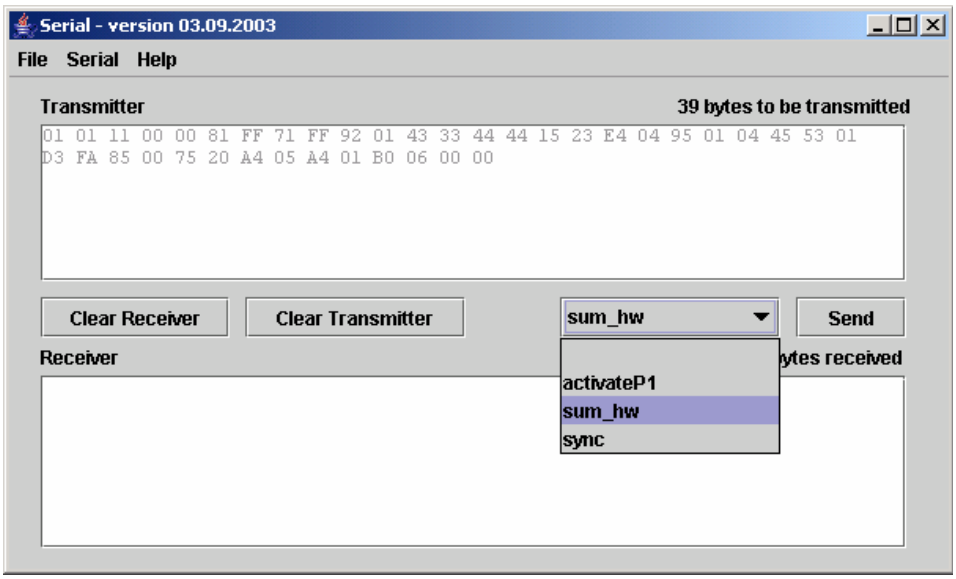


Figura 93 – Interface gráfica do SerialApp.

O SerialApp tem algumas funções para simplificar o uso da interface serial. Entre elas a possibilidade de carregar arquivos texto de um diretório do computador hospedeiro (conforme mostrado na Figura 93) e abrir terminais específicos para cada processador, como apresenta a Figura 94.

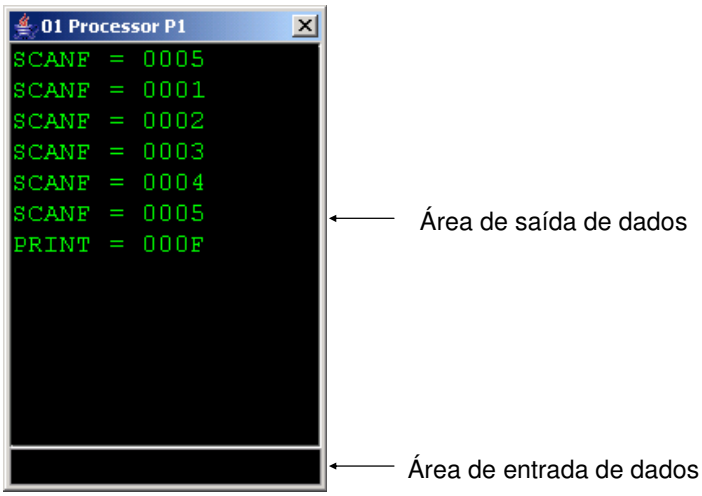


Figura 94 – Terminal para interação direta com um processador específico do sistema.

6.2 Calculadora Reconfigurável

Este estudo de caso tem o objetivo de apresentar um SDR simples, dando ênfase ao fluxo de projeto proposto no Capítulo 4. Para este estudo de caso utilizou-se um dispositivo Virtex-II 1000. Os módulos de hardware deste estudo de caso são os mesmos utilizados pela calculadora reconfigurável apresentada no Anexo II. A única modificação é no arquivo *top*, que instancia as macros conforme explicado na Seção 4.2. Para este estudo de caso, utilizou-se macros LR e RL sem o sinal de controle, funcionando apenas como passagem de dados, da mesma forma que a Bus Macro utilizada pelo fluxo de Projeto Modular. O novo *top* do projeto, instanciando dois módulos *adder*, é apresentado no Anexo VII. Para implementar o segundo projeto do SDR com os dois módulos *sub* basta substituir as ocorrências da string "*adder*" por "*sub*" no arquivo *top*.

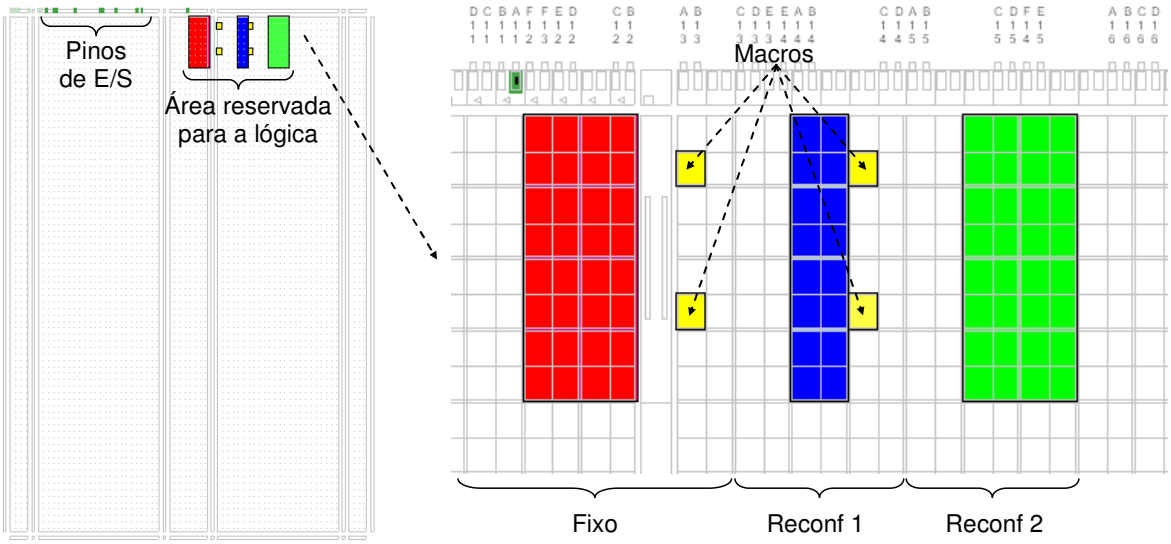


Figura 95 – Restrições de posicionamento da calculadora reconfigurável.

A Figura 95 apresenta as restrições de posicionamento feitas com a ferramenta Pace sobre os recursos instanciados no *top*. Note-se à esquerda da Figura 95 que a lógica do projeto foi

posicionada à direita do dispositivo, de forma a evitar que o roteamento dos pinos de E/S não invadam os módulos reconfiguráveis.

A Figura 96 apresenta o roteamento do projeto restringido na Figura 95. Note que fios do módulo fixo invadiram o módulo reconfigurável. Tentou-se diversas vezes voltar para o passo anterior e modificar o posicionamento para melhorar o roteamento, sendo o roteamento apresentado o melhor que foi obtido. Para este estudo de caso esta invasão de área é permitida desde que os fios ocupem as mesmas trilhas na área reconfigurável 1 do projeto *top* que instancia módulos *sub*.

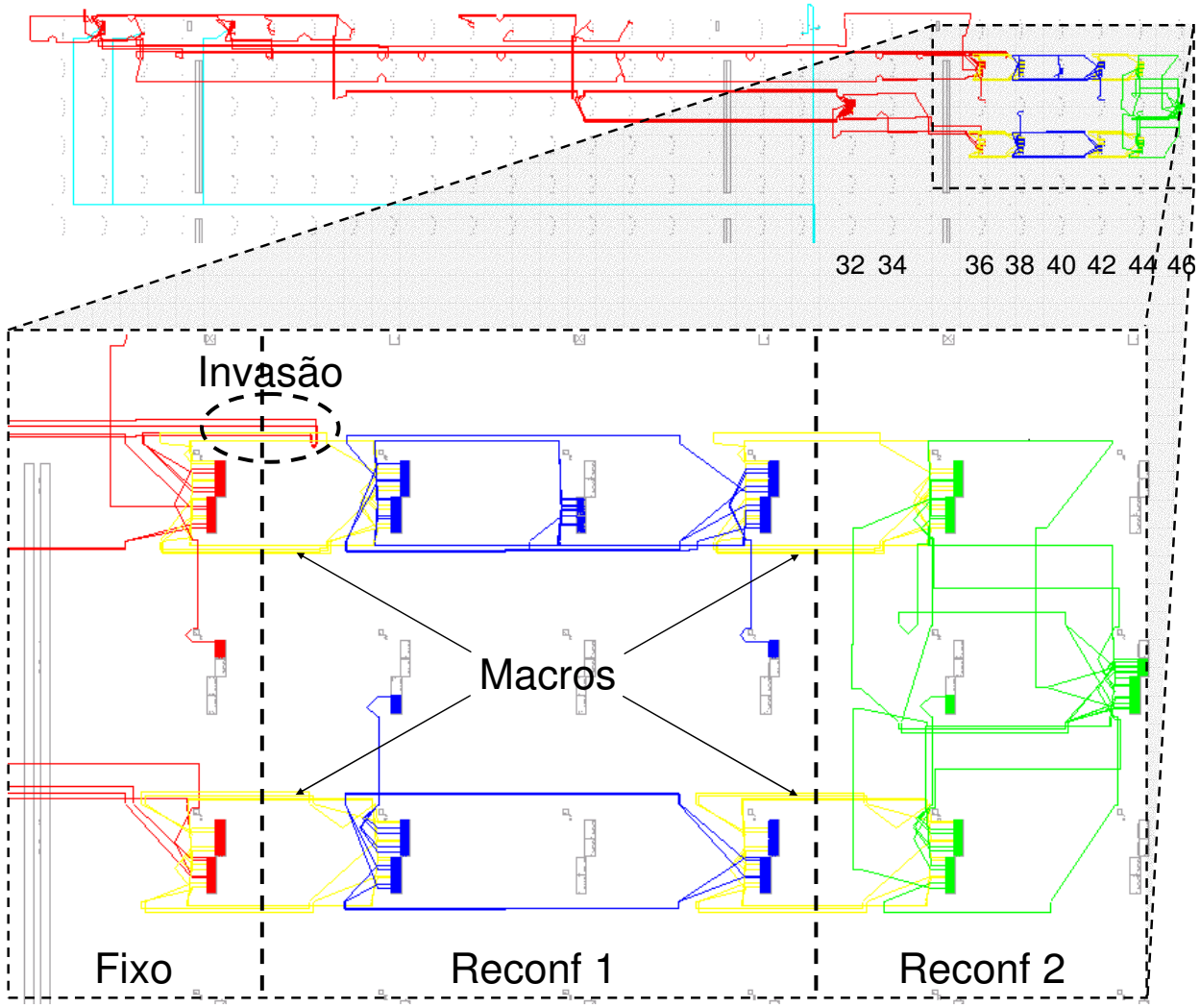


Figura 96 – Roteamento da calculadora reconfigurável.

De posse dos dois projetos *top*, um com dois módulos *adder* instanciados outro com dois módulos *sub* instanciados, pode-se gerar os bitstreams parciais. Utilizando a ferramenta CoreUnifier-II Pro basta abrir cada um dos *top*s, selecionar as áreas dos módulos reconfiguráveis e gerar o bitstream parcial. Para utilizar o BitGen, deve-se antes descobrir as máscaras das áreas reconfiguráveis com a ferramenta BitMask. Para este exemplo deve-se executá-la duas vezes, uma para cada área. Uma vez com as coordenadas da primeira área reconfigurável (`bitmask 38 42`) e uma segunda vez com as coordenadas da segunda área reconfigurável (`bitmask 44 46`). As respectivas máscaras 1C00000 e 6000000 retornadas pelo BitMask devem ser passadas para o

BitGen da seguinte forma:

```
bitgen –g PartialMask0:1C00000 –g ActiveReconfig:yes top.ncd adder_1.bit

bitgen –g PartialMask0:6000000 –g ActiveReconfig:yes top.ncd adder_2.bit

bitgen –g PartialMask0:1C00000 –g ActiveReconfig:yes top.ncd sub_1.bit

bitgen –g PartialMask0:6000000 –g ActiveReconfig:yes top.ncd sub_2.bit
```

Um detalhe importante é que as primeiras duas execuções do BitGen devem ocorrer dentro da pasta onde se encontra o *top* que possui os módulos *adder* instanciados, e as duas últimas execuções na pasta onde se encontra o *top* que instancia os módulos *sub*.

Para verificar o correto funcionamento do SDR, prototipou-se o projeto inicial com os dois módulos somadores e, a partir dos *switch buttons* da plataforma, requisitou-se a operação 3+2 para o primeiro somador e o resultado da primeira operação somado ao valor 1 para o segundo somador. O resultado 6 ((3+2)+1) foi apresentado no *display* da plataforma. A seguir, se configurado o primero subtrator, o resultado se altera instantaneamente para 2 ((3-2)+1), se configurado o segundo, o resultado se altera para 4 ((3+2)-1) e se configurado ambos os subtratores, o resultado se altera para 0 ((3-2)-1).

6.3 Contador Progressivo / Regressivo

O fluxo de projeto para SDRs, apresentado no Capítulo 4, foi estendido por Rafael Soares [SOA05] em seu trabalho de mestrado executado em paralelo ao do Autor, permitindo a reconfiguração de módulos do usuário conectados ao barramento CoreConnect. A este barramento podem ser conectados processadores (e.g. Microblaze, PowerPC), módulos do usuário e outros módulos disponibilizados no ambiente de desenvolvimento EDK da Xilinx.

Em [SOA05] foi apresentado um SDR auto-reconfigurável, no qual um processador Microblaze posicionado na parte fixa do sistema dispara a reconfiguração ora de um contador progressivo ora de um contador regressivo. A Figura 97 apresenta a planta baixa do sistema contador progressivo / regressivo para um dispositivo Virtex-II Pro 30.

Devido à grande quantidade de sinais que interligam o barramento CoreConnect ao módulo do usuário, um total de 11 macros RL e LRC foram utilizadas. O ambiente EDK também provê o controlador *hwicap*, que fornece acesso ao dispositivo ICAP, descrito na Seção 2.2.2.4, para aplicações executadas sobre o processador Microblaze ou PowerPC. Desta forma, aplicações executadas sobre estes processadores podem alterar a memória de configuração do dispositivo, assim modificando o hardware existente.

A diferença do fluxo utilizado em [SOA05] para o fluxo apresentado no Capítulo 4 é na etapa inicial, na qual todo o projeto é implementado no EDK e em um segundo momento as macros são inseridas não no *top* do projeto, mas sim dentro do módulo reconfigurável. O restante do fluxo segue de forma semelhante à apresentada no Capítulo 4.

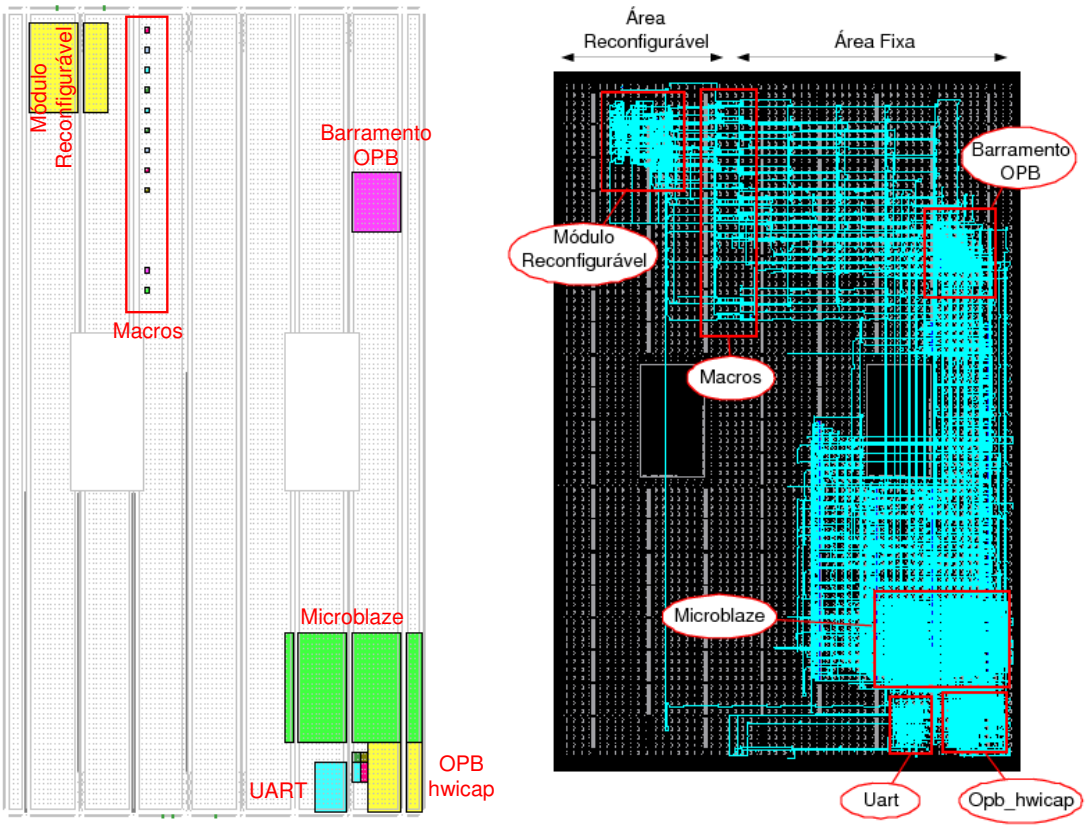


Figura 97 – Planta baixa do sistema contador progressivo / regressivo apresentado em [SOA05].

6.4 Multiplicação / Divisão / Raiz Quadrada

Este estudo de caso apresenta a NoC Artemis conectada a dois módulos reconfiguráveis. Três diferentes módulos reconfiguráveis podem ser substituídos em tempo execução neste sistema: multiplicação (Seção 6.1.1.5), divisão (Seção 6.1.1.6) e raiz quadrada (Seção 6.1.1.7). A Figura 98 apresenta as restrições de posicionamento e roteamento do projeto que possui dois módulos reconfiguráveis de raiz quadrada conectados à NoC. Os outros dois projetos (um que possui dois módulos de divisão conectados à NoC e outro que possui dois módulos de multiplicação) possuem restrições de posicionamento e roteamento idênticas na parte fixa. A Figura 99 apresenta o roteamento dos três módulos reconfiguráveis posicionados na parte superior do FPGA (módulo reconfigurável 01). Este estudo de caso utilizou uma Virtex-II Pro XC2VP30.

Dois detalhes importantes para aumentar a probabilidade do SDR funcionar corretamente são escolher os maiores módulos reconfiguráveis para iniciar o SDR e tentar restringir ao máximo a área dos módulos reconfiguráveis. Estes dois cuidados evitam que a árvore de relógio, que não é reconfigurada junto com o módulo, possua uma configuração diferente em cada um dos projetos do sistema, conforme evidenciado na Figura 99. Verificou-se na prática que a reconfiguração parcial do módulo raiz quadrada não funciona utilizando os projetos dos módulos multiplicação ou divisão como iniciais, somente quando o mesmo é o inicial. Isto ocorre porque o módulo de raiz quadrada utilizou dois ramos da árvore principal de relógio, enquanto os outros dois módulos utilizaram somente um. Logo, apenas o projeto com dois módulos de raiz quadrada possui a configuração do

segundo ramo da árvore de relógio que possibilita o correto funcionamento para estes módulos reconfiguráveis e não apresenta problemas para os módulos de divisão e multiplicação onde este segundo ramo não é utilizado.

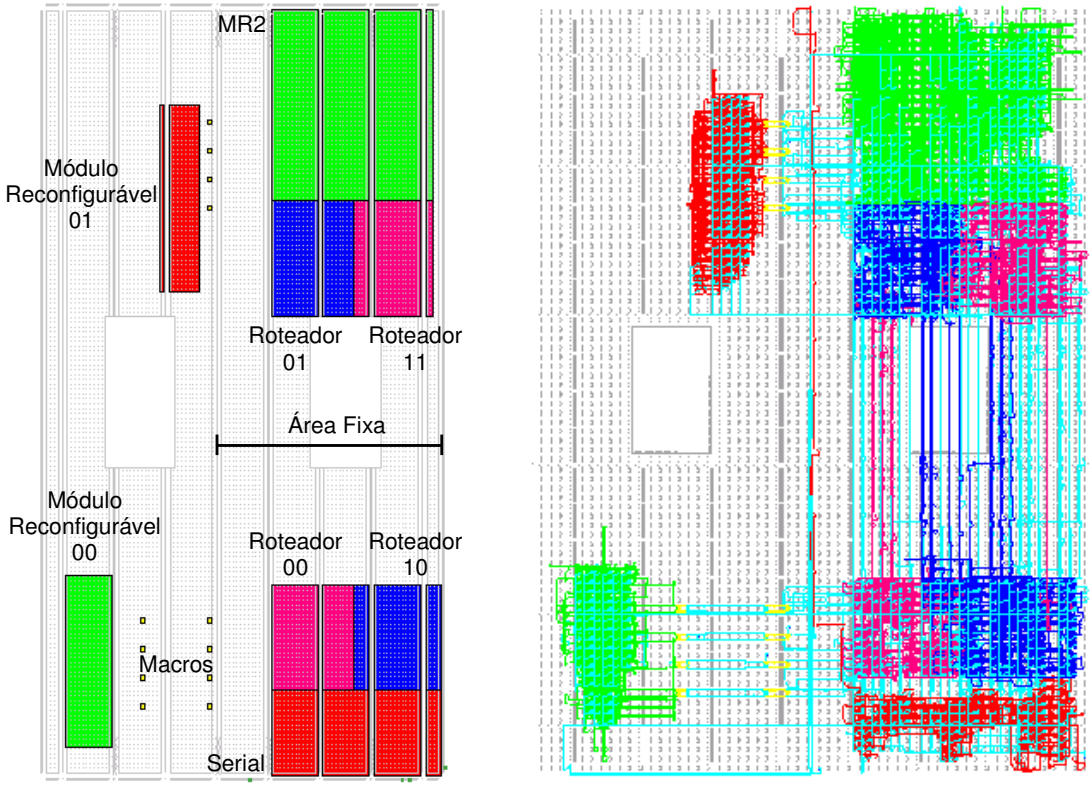


Figura 98 – Posicionamento e roteamento de dois módulos reconfiguráveis conectados à NoC Artemis.

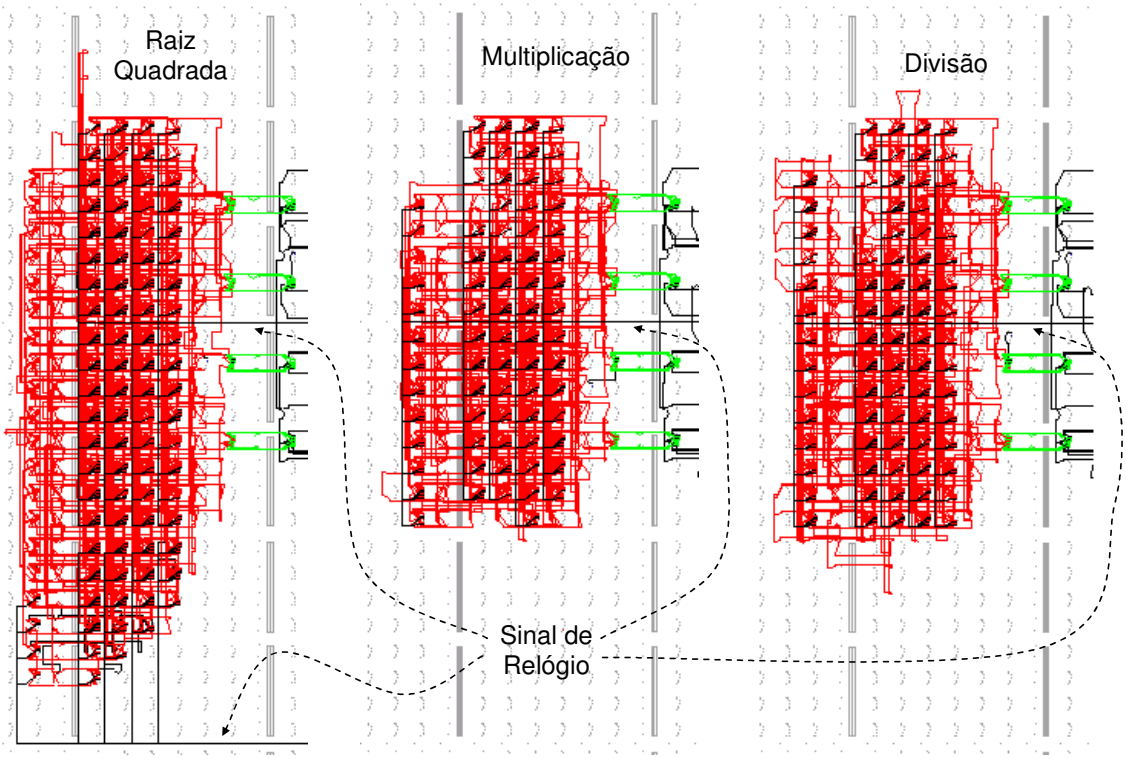


Figura 99 – Roteamento dos módulos reconfiguráveis 01: raiz quadrada, divisão e multiplicação.

Para verificar o correto funcionamento do SDR, prototipou-se o projeto inicial com os dois módulos de raiz quadrada e a partir do computador hospedeiro enviou-se um pacote através do módulo serial para interromper a comunicação da NoC com o módulo a ser reconfigurado. A seguir efetuou-se a configuração do bitstream parcial com o módulo de divisão ou raiz quadrada e restabeceu-se a comunicação da NoC com o módulo reconfigurável. Após isso todos os módulos foram verificados, garantindo que o SDR continua funcionando conforme o esperado após a reconfiguração parcial e dinâmica.

6.5 Reconfiguração de Processadores em Sistemas Multiprocessados

Sob o ponto de vista de sistemas parcial e dinamicamente reconfiguráveis não há diferença entre reconfigurar processadores e reconfigurar módulos. A Figura 100 apresenta um SDR semelhante ao apresentado na Seção anterior, com a diferença de possuir um processador MR2 ou R8 conectado ao roteador inferior esquerdo da NoC Artemis. O processador MR2 pode ser substituído em tempo execução pelo processador R8 e vice-versa ou pelos módulos reconfiguráveis previamente apresentados de multiplicação, divisão e raiz quadrada.

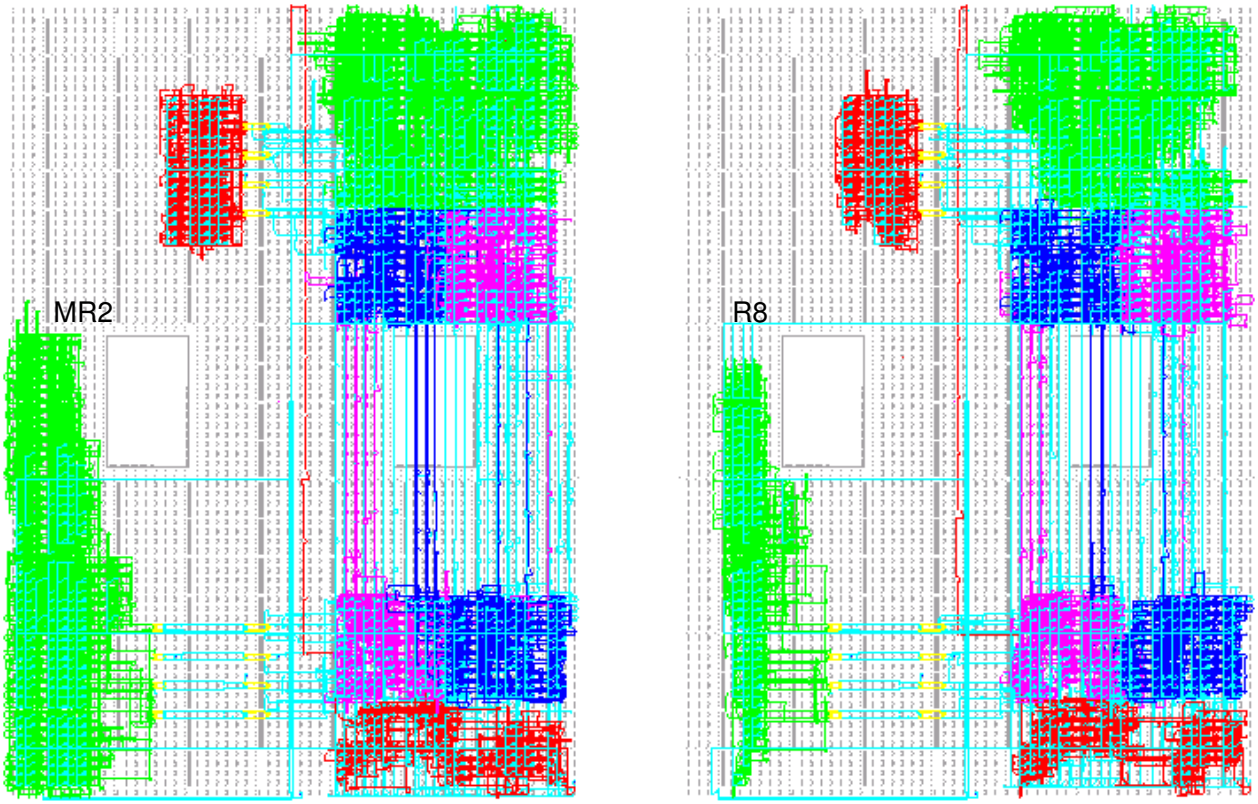


Figura 100 – Roteamento de dois projetos de um SDR que possui processadores reconfiguráveis conectados à Artemis.

Aplicações em MPSoCs possuem requisitos diferenciados. Por exemplo, para operações que não exigem alto desempenho, uma CPU simples de 16 bits pode ser suficiente. Para aplicações mais complexas, como tratamento digital de sinais, processadores mais complexos são necessários.

Este experimento mostra que é possível configurar os processadores conforme a aplicação a ser executada e que é possível também utilizar processadores especializados para determinadas tarefas, sem necessariamente tê-los configurados o tempo todo.

6.6 Realocação estática de módulos

Até o momento da escrita deste documento não foi possível rotear os projetos apresentados nas Seções 6.4 ou 6.5 de forma que estes sejam realocáveis. Os projetos apresentados nessas Seções não são realocáveis porque o módulo reconfigurável superior (ver posicionamento das macros na Figura 98 ou na Figura 100) é conectado à NoC por macros implementadas sobre colunas de BRAMs / multiplicadores, enquanto o módulo reconfigurável inferior é conectado por macros implementadas sobre colunas de CLBs.

A Figura 101 apresenta o roteamento do projeto inicial de módulos estaticamente realocáveis conectados à NoC Artemis. Como este estudo de caso foi implementado em um dispositivo Virtex-II XC2V1000 (menor que o dispositivo Virtex-II Pro XC2VP30 utilizado na Seção anterior), apenas 3 módulos foram conectados à NoC para simplificar o roteamento: um módulo serial posicionado na parte fixa do sistema e dois módulos reconfiguráveis denominados “double” e “not”. O módulo “double” soma um parâmetro de 16 bits recebido com ele mesmo e o módulo “not” inverte todos os bits de um parâmetro de 16 bits. Estes módulos ocupam 75 *slices* e 77 *slices*, respectivamente.

A reconfiguração destes módulos é dita estática porque uma parte do sistema (o segundo módulo reconfigurável posicionado na mesma coluna) possui a sua execução interrompida durante a reconfiguração. Dois bitstreams parciais foram gerados a partir da ferramenta CoreUnifier-II Pro. Um realocando o módulo “not” sobre a área do módulo “double”, e o outro realocando o módulo “double” sobre a área do módulo “not”. Estes bitstreams parciais permitem que a NoC possua dois módulos “not” ou dois módulos “double” conectados a ela a partir de um único projeto inicial através de realocação.

A macro utilizada pelo sistema, também apresentada na Figura 101, possui duas diferenças em relação às macros apresentadas na Seção 4.2.1. Estas ocupam 4 colunas de CLBs e utilizam apenas 4 fios e 4 *slices*. A utilização de um número maior de colunas pode ser o último recurso, quando não se está preocupado com área do FPGA, para evitar que linhas longas cruzem os limites das áreas, conforme apresentado à direita da Figura 101. Também foi detectado em alguns projetos que, dependendo da posição origem dos sinais conectados à macro, podem não existir recursos de roteamento adequados em um única linha de CLBs do FPGA. A utilização de um roteamento inadequado na interface de módulos pode ser percebida quando linhas longas são utilizadas para interconectar pontos de origem e destino próximos ou quando diversas linhas curtas são utilizadas para interconectar pontos de origem e destino distantes. Nestes casos, uma alternativa é utilizar um número maior de macros em diferentes linhas de CLBs do FPGA, com um número menor de fios por macro, conforme a macro apresentada à direita da Figura 101.

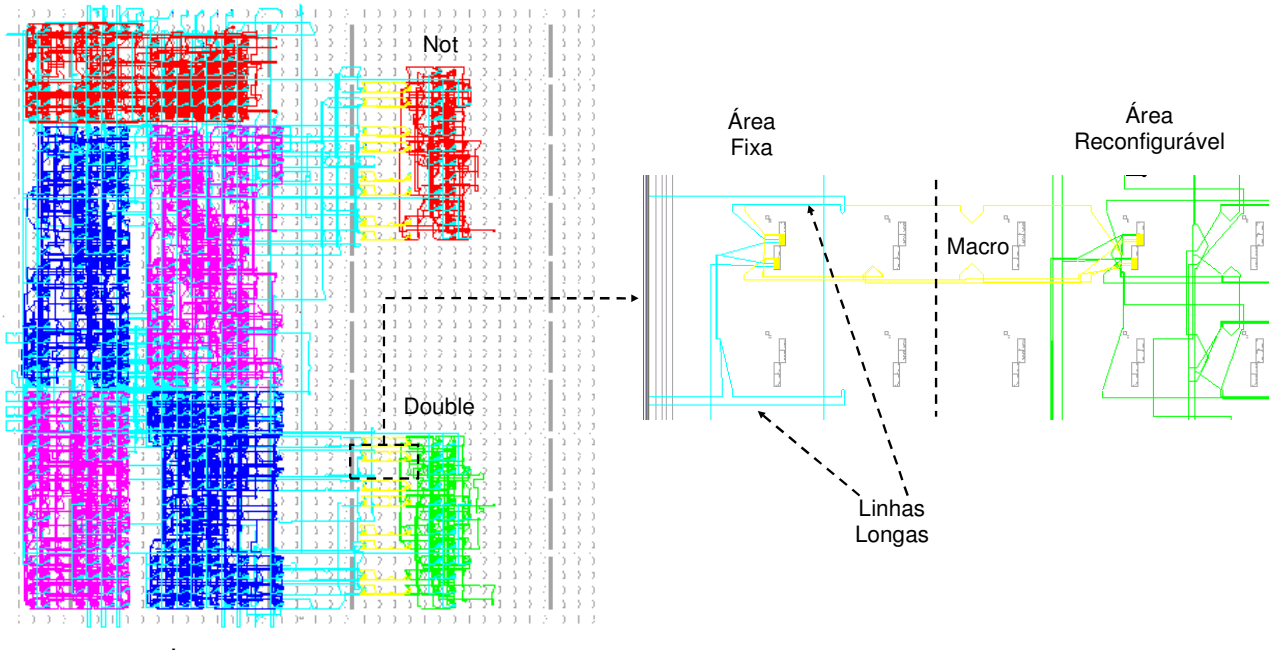


Figura 101 – Roteamento de dois módulos estaticamente reconfiguráveis conectados à NoC Artemis.

Apesar de simples o conceito de realocação, este estudo de caso evidenciou a dificuldade prática da implementação desta técnica em dispositivos Xilinx. A opção de realizar inicialmente a realocação vertical teve por objetivo verificar a factibilidade de realocação. Comprovou-se neste estudo de caso que realocação é possível nestes dispositivos.

6.7 Realocação dinâmica de CLBs

A Seção anterior apresentou a realocação vertical parcial e estática de módulos de hardware. O objetivo desta Seção é validar a realocação horizontal parcial e dinâmica, assim validando a etapa opcional do fluxo de SDRs proposto no Capítulo 4 e a ferramenta CoreUnifer-II Pro. No entanto, ainda não foi possível rotear módulos, conforme apresentados nos estudos de casos anteriores, apenas rotear CLBs para serem horizontalmente realocáveis. Esta dificuldade na realocação horizontal ocorre devido à irregularidade dos FPGAs Xilinx com relação às colunas de multiplicadores e BRAMs. Estas colunas tornam-se "âncoras" para módulos reconfiguráveis, que só podem ser realocados horizontalmente de 6 em 6 colunas de CLBs, no caso dos dispositivos Virtex-II Pro, assim dificultando as opções de posicionamento e, por conseqüência, roteamento.

Quando o CoreUnifier-II Pro reconfigurar uma CLB, será exibido no *display* um índice de CLB, com um novo conteúdo, atribuído por realocação.

O estudo de caso apresentado nesta Seção possui 8 CLBs realocáveis, onde cada CLB possui uma constante diferente armazenada. O projeto inicial possui na CLB 0, a constante 8; na CLB 1, a constante 9; e assim por diante. Ao pressionar-se um *push-button* da plataforma de prototipação, exibe-se em *display* de 7 segmentos o número de uma CLB e a sua respectiva constante armazenada. Ao pressionar-se novamente o mesmo *push-button*, exibe-se o número da próxima CLB e a sua respectiva constante armazenada. Desta forma quando uma CLB é realocada

através do CoreUnifier-II Pro a constante da CLB de origem é mostrada na CLB de destino da realocação. O roteamento deste projeto é apresentado na Figura 102. Este sistema é dito dinamicamente reconfigurável porque a modificação de uma parte do sistema não afeta outras.

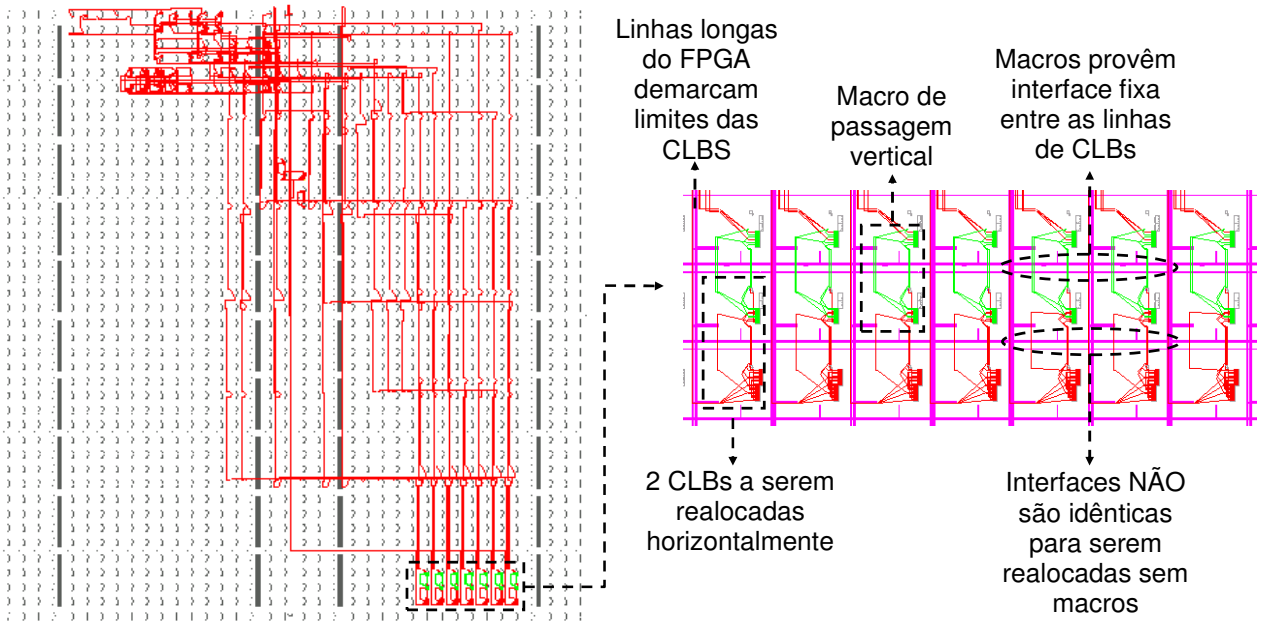


Figura 102 – SDR que possui CLBs realocáveis horizontalmente.

7 CONCLUSÕES E TRABALHOS FUTUROS

A computação reconfigurável é indubitavelmente uma área promissora para sistemas que necessitam de flexibilidade e desempenho. Potência é uma terceira característica importante, porém não contemplada no escopo do presente trabalho. Intuitivamente, pode-se argumentar que havendo menos componentes no SoC, haverá um menor consumo de potência. Entretanto, Becker et al. [BEC03] mostraram que durante o processo de reconfiguração o consumo de potência é elevado. A demanda por estes requisitos (flexibilidade, desempenho e potência) tem sido verificada principalmente em dispositivos portáteis. Estes devem ser compatíveis com diversos protocolos de redes sem fio (802.11, Bluetooth, Wi-Fi, GSM, UTMS), fotografar, filmar, jogar, habilitar a comunicação por voz, navegar na Internet, escutar música armazenada local ou remotamente, enviar e receber mensagens de texto, funcionar como localizador geográfico, reproduzir vídeos e ainda executar algoritmos avançados de criptografia (AES, DES, BlowFish) para não permitir que a comunicação ou os dados armazenados no dispositivo possam ser interceptados por um agente externo mal intencionado.

No entanto, a computação reconfigurável ainda não é amplamente utilizada em produtos comerciais. Um dos motivos que inibe o seu uso é a dificuldade de conseguir tirar proveito adequado desta tecnologia para alcançar o desempenho esperado. Foi visando suprir esta demanda que o presente trabalho buscou contribuir.

As próximas Seções apresentam um resumo do trabalho desenvolvido, uma análise das contribuições e indicações de trabalhos futuros.

7.1 Resumo do trabalho desenvolvido

O estudo do estado da arte apontou as diferentes formas de se fazer uso da computação reconfigurável e diferentes dispositivos nos quais sistemas reconfiguráveis podem ser implementados. Entre estes, os dispositivos Xilinx chamam a atenção devido à disponibilidade de documentação e trabalhos relacionados no assunto, tanto no meio acadêmico quanto na indústria, de ferramentas que permitem interagir nas fases intermediárias da descrição de hardware à prototipação em FPGA. O mesmo fabricante se destaca dos demais neste assunto quanto à diversidade de plataformas e à alta densidade em termos de portas lógicas dos dispositivos oferecidos.

A partir disso, o restante do documento explorou especificamente as características dos FPGAs Xilinx, estudando os fluxos para a implementação de sistemas parcial e dinamicamente reconfiguráveis. Em relação a estes fluxos, constatou-se uma série de dificuldades e limitações no emprego dos mesmos, motivando o desenvolvimento de um novo fluxo que contornasse alguns dos problemas apontados.

A proposta de um novo fluxo de projeto para a implementação de SDRs evitou o uso de

macros baseadas em *tristates*, utilizou ferramentas baseadas no fluxo de projeto padrão e apresentou dois métodos para a geração de bitstreams parciais compatíveis com os dispositivos Virtex-II e Virtex-II Pro. A este fluxo, adicionou-se a possibilidade de gerar bitstreams parciais a partir da realocação de módulos de um SDR através da ferramenta CoreUnifier-II Pro. No entanto, o fluxo proposto sofre dos mesmos problemas enfrentados pelos fluxos existentes em relação ao roteamento. Para auxiliar neste quesito, diversos artifícios de projeto e implementação foram sugeridos ao longo do documento.

De forma a atender as demandas de comunicação de futuros SDRs, uma NoC foi modificada para permitir que módulos reconfiguráveis fossem dinamicamente conectados a ela. Esta NoC, denominada Artemis, habilita o isolamento dos módulos reconfiguráveis para que a reconfiguração do FPGA não interfira com o restante do sistema que continua operando. Outra característica da Artemis é um mecanismo de segurança em hardware que descarta pacotes de dados quando estes são enviados (erroneamente) para um módulo que se encontra em processo de reconfiguração, evitando assim situações de erro que possam conduzir a bloqueios permanentes da rede.

O fluxo de projeto proposto e a NoC Artemis foram empregados na prática em estudos de caso apresentados no Capítulo 6. O fluxo, uma vez dominado e seguido conforme apresentado, não reportou erro durante a sua execução. A reconfiguração parcial e dinâmica de módulos foi atingida na prática. A realocação dinâmica de módulos conectados à NoC Artemis não foi alcançada até o fechamento do presente trabalho, apenas a realocação estática de módulos e a realocação dinâmica de poucas CLBs.

7.2 Contribuições

Segundo a visão do Autor, este trabalho contribui em 7 diferentes aspectos:

- **Fluxo de projeto para SDRs**: propôs-se uma nova forma de implementar a reconfiguração parcial e dinâmica de dispositivos Xilinx baseada nas ferramentas padrões. A vantagem deste fluxo é ser menos suscetível a erros (em relação a fluxos que utilizam Bus Macros), ser mais simples de ser executado (em relação ao fluxo de Projeto Modular e ao fluxo de Palma executado pelo Autor em [MÖL04] e [PAL02a]) e ser compatível, a princípio, com todos os FPGAs atuais que suportam reconfiguração parcial e dinamica da Xilinx.
- **NoC Artemis**: uma NoC que habilita a reconfiguração parcial e dinâmica de módulos conectados a ela. A NoC Artemis provê mecanismos (e.g. pacotes de controle, isolamento do módulo reconfigurável, descarte de pacotes) que possibilitam a reconfiguração de uma parte do FPGA sem interferência no restante do sistema.
- **Macros:** este trabalho propôs a utilização de macros baseadas em LUTs, eliminando a possibilidade de fios se conectarem no lado errado da macro, conforme verificado nas Bus Macros. As macros propostas podem ser modificadas para facilitar o roteamento do SDR.
- **Geração de bitstreams parciais:** duas alternativas para a geração de bitstreams parciais através

do método de extração de *frames* foram apresentadas. Uma alternativa é através da ferramenta BitGen com o auxílio da ferramenta BitMask. A segunda alternativa é através da ferramenta CoreUnifier-II Pro.

- **Realocação:** também foi adicionada à ferramenta CoreUnifier-II Pro a possibilidade de realocar módulos de hardware de SDRs. A realocação, embora difícil de ser empregada na prática, pode reduzir o gasto em memória de sistemas auto-reconfiguráveis para armazenamento de bitstreams com configurações do sistema.
- **Ferramenta de verificação de roteamento:** a ferramenta XDLAnalyzer foi proposta para analisar todos os sinais de um projeto, verificando se algum deles atravessa a fronteira do módulo de origem sem passar por macros.
- **Interconexão do processador MR2 à NoC:** foi no contexto do presente trabalho que ocorreu o porte do processador MR2 para a NoC. Para isto foi criado um *wrapper* para o processador MR2 que conecta a NoC e as caches de instruções e dados ao processador. Este porte buscou reduzir ao máximo o consumo em área, resultando na criação de drivers para a comunicação com outros módulos da NoC.

7.3 Trabalhos futuros

Entre as atividades previstas para o futuro imediato e de médio prazo, cita-se:

- Atividades relacionadas à arquitetura reconfigurável:
  - Utilizar o ICAP para fazer a auto-reconfiguração do SDR;
  - Sinalizar o término do processamento de módulos reconfiguráveis para o processador MR2 através de interrupções.
- Atividades relacionadas a ferramentas para SDR:
  - Implementar de forma efetiva realocação em hardware;
  - Atualizar a ferramenta CoreUnifier-II Pro para torná-la compatível com a Virtex 4;
  - Utilizar uma metodologia semelhante ao fluxo de Projeto Modular onde cada módulo do sistema é uma macro, metodologia esta que pode ser denominada *macro-based design*;
  - Automatizar o processo de inserção de macros em SDRs;
  - Permitir que o XDLAnalyzer refaça roteamentos que não atendem o fluxo de SDRs de forma automática;
  - Salvar e recuperar o contexto de módulos reconfiguráveis, permitindo a reconfiguração de um módulo que compartilhe a mesma coluna de outro módulo reconfigurável.
- Atividades relacionadas a redes intra-chip:
  - Inserir a capacidade de gerar a NoC Artemis na ferramenta Maia;
  - Desenvolver um adaptador para NoCs, de forma que outras NoCs possam acoplar módulos dinamicamente reconfiguráveis;

- Permitir a reconfiguração do meio de comunicação;
- Conectar os processadores PowerPC, Microblaze e Picoblaze à NoC Artemis.

8 REFERÊNCIAS BIBLIOGRÁFICAS

[ALT04] Altera, Inc. "**Altera Devices**". Capturado em: http://www.altera.com, Set. 2005.

[ATH93] Athanas, P.; Silverman, H. "**Processor Reconfiguration Through Instruction-Set Metamorphosis**". Computer, vol. 26 (3), Mar. 1993, pp. 11-18.

[ATM94] Atmel Corporation. "**AT6000 Series Datasheet**". Capturado em: http://www.atmel.com, Out. 2005.

[ATM98] Atmel Corporation. "**AT40k Series Datasheet**". Capturado em: http://www.atmel.com, Out. 2005.

[BAR02] Barat, F.; Lauwereins, R.; Deconinck, G. "**Reconfigurable Instruction Set Processors from a Hardware/Software Perspective**". IEEE Transactions on Software Engineering, vol. 28 (9), Set. 2002, pp. 847-862.

[BEC00] Becker, J.; Piontek, T.; Glesner, M. "**DReAM: A Dynamically Reconfigurable Architecture for Future Mobile Communications Applications**". In: Field Programable Logic and Applications (FPL), 2000, pp. 312-321.

[BEC03] Becker, J.; Huebner, M.; Ullmann, M. "**Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-Offs and Limitations**". In: Brazilian Symposium on Integrated Circuits and Systems Design (SBCCI), 2003, pp. 283-288.

[BEN02] Benini, L.; De Micheli, G. "**Networks on Chips: a New SoC Paradigm**". Computer, vol. 35 (1), Jan. 2002, pp. 70-78.

[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.

[BOB05] Bobda, C.; Ahmadinia, A. "**Dynamic Interconnection of Reconfigurable Modules on Reconfigurable Devices**". IEEE Design & Test of Computers, vol. 22 (5), Set. 2005, pp. 443-451.

[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.

[BUT04] Butel, P.; Habay, G.; Rachet, A. "**Managing Partial Dynamic Reconfiguration in Virtex-II Pro FPGAs**". Xcell Journal, vol. 50, Ago. 2004, pp. 32-37.

[CAL04] Calazans, N.; Moraes, F. "**Especificação da Arquitetura MR2**". Capturado em: http://www.inf.pucrs.br/~calazans/undergrad/orgcomp/arq_MR2_V1.6.pdf, Mar. 2004.

[CAR04] Carvalho, E. "**RSCM – Controlador de Configurações para Sistemas de Hardware Reconfigurável**". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2004, 127 p.

[CON92] Concurrent Logic. "**CLi6000 Series Field-Programmable Gate Arrays**". 1992.

[CUR04] Curd, D. "**Partial Reconfiguration of RocketIO Pre-Emphasis and Differential Swing Control Attributes**". Xilinx Application Note 660 (XAPP660 v2.2), 2004, 9 p.

[DAL01] Dally, W.; Towles, B. "**Route Packets, not Wires: On-Chip Interconnection Networks**". In: Design Automation Conference (DAC), 2001, pp. 684-689.

[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, Suíça, 2002, 82 p.

[EBE95] Ebeling, C.; McMurchie, L.; Hauck, S.; Burns, S. "**Placement and Routing Tools for the Triptych FPGA**". IEEE Transactions on Very Large Scale Integration Systems, vol. 3 (4), Dez. 1995, pp. 473-482.

[GAP04a] GAPH – Hardware Design Support Group. "**R8 Simulator**". Capturado em: http://www.inf.pucrs.br/~gaph/homepage/download/software/simulatorR8.zip, Set. 2004.

[GAP04b] GAPH – Hardware Design Support Group. "**Serial Software**". Capturado em: http://www.inf.pucrs.br/~gaph/homepage/download/software/SerialSoftware.zip, Set. 2004.

[GAP05] GAPH – Hardware Design Support Group. "**Packer**". Capturado em: http://www.inf.pucrs.br/~gaph/homepage/download/software/Packer.zip, Nov. 2005.

[GOL99] Goldstein, S.; Schmit, H.; Moe, M.; Budiu, M.; Cadambi, S.; Taylor, R.; Laufer, R. "**PipeRench: a Coprocessor for Streaming Multimedia Acceleration**". In: International Symposium on Computer Architecture (ISCA), 1999, pp. 28-39.

[GUC99] Guccione, S., Levi, D., Sundararajan, P. "**JBits: Java-Based Interface for Reconfigurable Computing**". In: Military and Aerospace Applications of Programmable Devices and Technologies Conference, 1999.

[GUC05] Guccione, S. "**List of FPGA-Based Computing Machines**". Capturado em: http://www.io.com/~guccione/HW_list.html, Out. 2005.

[HAR95] Hartenstein, R.; Kress, R. "**A Datapath Synthesis System for the Reconfigurable Datapath Architecture**". In: Asia and South Pacific Design Automation Conference (ASP-DAC), 1995, pp. 479-484.

[HAR97] Hartenstein, R.; Becker, J.; Herz, M.; Nageldinger, U. "**An Innovative Platform for Embedded System Design**". In: Architekturen von Rechensystemen (ARCS), 1997, pp. 143-152.

[HAU97a] Hauser, J.; Wawrzynek, J. "**Garp: a MIPS Processor with a Reconfigurable Coprocessor**". In: FPGAs for Custom Computing Machines (FCCM), 1997, pp. 12-21.

[HAU97b] Hauck, S.; Fry, T.; Hosler, M.; Kao, J. "**The Chimaera Reconfigurable Functional Unit**". In: FPGAs for Custom Computing Machines (FCCM), 1997, pp. 87-96.

[HAU98] Hauck, S. "**The Roles of FPGAs in Reprogrammable Systems**". Proceedings of the IEEE, vol. 86 (4), Abr. 1998, pp. 615-638.

[HEN90] Hennessy, J.; Patterson, D. "**Computer Architecture: A Quantitative Approach**". Estados Unidos, San Mateo, Morgan Kauffmann Publishers, 1990, 594 p.

[HEN03] Henkel, J. "**Closing the SoC Design Gap**". Computer, vol. 36 (9), Set. 2003, pp. 119-121.

[HOR02] Horta, E.; Lockwood, J.; Taylor, D.; Parlour, D. "**Dynamic Hardware Plugins in an FPGA with Partial Run-Time Reconfiguration**". In: Design Automation Conference (DAC), 2002. pp. 343-348.

[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.

[HOW92] Howard, N.; Taylor, R. "**Reconfigurable Logic: Technology and Applications**". Computing & Control Engineering Journal, vol. 3 (5), Set. 1992, pp. 235-240.

[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.

[JAC99] Jacob, J.; Chow, P. "**Memory Interfacing and Instruction Specification for Reconfigurable Processors**". In: Field Programmable Gate Arrays (FPGA), 1999, pp. 145-154.

[KEA88] Kean, T. "**Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation**". Tese de Doutorado, Universidade de Edinburgh, Edinburgh, Inglaterra, 1988, 286 p.

[LAU05] Laurus, J. "**SPIM - A MIPS32 Simulator**". Capturado em: http://www.cs.wisc.edu/~larus/spim.html, Nov. 2005.

[LEW98] Lewis, D.; Galloway, D.; Ierssel, M.; Rose, J.; Chow, P. "**The Transmogrifier-2: A 1 Million Gate Rapid Prototyping System**". In: IEEE Transactions on Very Large Scale Integration Systems, 1998, pp. 188-198.

[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.

[MAR99] Marshall, A.; Stansfield, T.; Kostarnov, I.; Vuillemin, J.; Hutchings, B. "**A Reconfigurable Arithmetic Array for Multimedia Applications**". In: Field Programmable Gate Arrays (FPGA), 1999, pp. 135-143.

[MAR04] Marescaux, T.; Nollet, V.; Mignolet, J.; Bartic, A.; Moffat, W.; Avasare, P.; Coene, P.; Verkest, D.; Vernalde, S.; Lauwereins, R. "**Run-Time Support for Heterogenous Multitasking on Reconfigurable SoCs**". Integration The VLSI Journal, vol. 38 (1), Out. 2004, pp. 107-130.

[MEL03] Mello, A.; Möller, L. "**Arquiteturas Multiprocessadas em SoCs: Estudo de Diferentes Topologias de Interconexão**". Trabalho de Conclusão de Graduação, Pontifícia Universidade Católica do Rio Grande do Sul (Faculdade de Informática - FACIN), Porto Alegre, Brasil, 2003, 110 p.

[MEL05] Mello, A.; Möller, L.; Calazans, N.; Moraes, F. "**A Multiprocessing System Enabled by a Network on Chip**". In: Design, Automation and Test in Europe Conference and Exhibition (DATE), 2005, pp. 502-507.

[MEM02] Memec Insight. "**Virtex-II V2MB1000 Development Board User's Guide**". Memec Insight User's Guide (v3.0), 2002, 44 p.

[MES02] Mesquita, D. "**Contribuições para Reconfiguração Parcial, Remota e Dinâmica de FPGAs**". Dissertação de Mestrado, Pontifícia Universidade Católica do Rio Grande do Sul (Programa de Pós-Graduação em Ciência da Computação - PPGCC), Porto Alegre, Brasil, 2002, 99 p.

[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.

[MIR96] Mirsky, E.; DeHon, A. "**MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources**". In: FPGAs for Custom Computing Machines (FCCM), 1996, pp. 157-166.

[MÖL03] Möller, L.; Moraes, F.; Calazans, N. "**Ferramentas de Reconfiguração Parcial, Remota e Dinâmica de FPGAs Virtex**". Relatório Técnico (TR035), Pontifícia Universidade Católica do Rio Grande do Sul (Programa de Pós-Graduação em Ciência da Computação - PPGCC), 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**". Capturado em: http://www.inf.pucrs.br/~gaph/Projects/R8/public/R8_arq_spec_eng.pdf, Dez. 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.

[MOR04] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "**HERMES: an Infrastructure for Low Area Overhead Packet-Switching Networks on Chip**". Integration The VLSI Journal, vol. 38 (1), Out. 2004, pp. 69-93.

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

[OTT00] Otten, R.; Stravers, P. "**Challenges in Physical Chip Design**". In: Computer Aided Design (ICCAD), 2000, pp. 84-91.

[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.

[PAT00] Hennessy, J.; Patterson, D. "**Organização e Projeto de Computadores: a Interface Hardware/Software**". LTC, 2000, 551 p.

[PLE90] Plessey Semiconductors. "**ERA60100 Electrically Reconfigurable Array Data Sheet**". GEC Plessey Semiconductors LTDA, Swindon, Wiltshire SN2 2QW, UK, 1990.

[RAZ94] Razdan, R.; Brace, K.; Smith, D.; "**PRISC Software Acceleration Techniques**". In: IEEE International Conference on Computer Design (ICCD), VLSI in Computer & Processors, 1994, pp. 145-149.

[SAS02] Sassatelli, G.; Torres, L.; Benoit, P.; Gil, T.; Diou, C.; Cambon, G.; Galy, J. "**Highly Scalable Dynamically Reconfigurable Systolic Ring-Architecture for DSP Applications**". In: Design, Automation and Test in Europe (DATE), 2002, pp. 553-558.

[SIN98] Singh, H.; Lee, M.; Lu, G.; Kurdahi, F.; Bagherzadeh, N.; Filho, E. "**MorphoSys: a Reconfigurable Architecture for Multimedia Applications**". In: Brazilian Symposium on Integrated Circuits and Systems Design (SBCCI), 1998. pp. 134-139.

[SOA05] Soares, R. "**Controladores de Configuração para Arquiteturas Reconfiguráveis: Características, Comparação e Evolução**". Seminário de Andamento, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2005, 15 p.

[TEN02] Tensilica, Inc. "**Xtensa Microprocessor**". Overview Handbook, 2002, 100 p.

[VCC04] Virtual Computer Corporation. "**The Virtual Workbench"**. Capturado em: http://www.vcc.com, Ago. 2004.

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

[WAZ93] Wazlowski, M.; Agarwal, L.; Lee, T.; Smith, A.; Lam, E.; Athanas, P.; Silverman, H.; Ghosh, S. "**PRISM-II Compiler and Architecture**". In: FPGAs for Custom Computing Machines (FCCM), 1993, pp. 9-16.

[WIN01] Wingard, D. "**MicroNetwork-Based Integration for SoCs**". In: Design Automation Conference (DAC), 2001, pp. 673-677.

[WIR94] Wirthlin, M.; Hutchings, B.; Gilson, K. "**The Nano Processor: a Low Resource Reconfigurable Processor**". In: FPGAs for Custom Computing Machines (FCCM), 1994, pp. 23-30.

[WIR95] Wirthlin, M.; Hutchings, B. "**A Dynamic Instruction Set Computer**". In: FPGAs for Custom Computing Machines (FCCM), 1995, pp. 99-107.

[WIT96] Wittig, R.; Chow, P. "**OneChip: an FPGA Processor with Reconfigurable Logic**". In: FPGAs for Custom Computing Machines (FCCM), 1996, pp. 126-135.

[XES00] Xess Corporation. "**XSV-800 Board Manual**". XSV Board V1.0 Manual, 2000, 60 p.

[XIL97] Xilinx, Inc. "**XC6200 Field Programmable Gate Arrays**". Xilinx Product Description (v1.10), 1997, 73 p.

[XIL98] Xilinx, Inc. "**XC3000 Series Field Programmable Gate Arrays**". Xilinx Product Description (v3.1), 1998, 76 p.

[XIL99] Xilinx, Inc. "**XC4000E and XC4000X Series Field Programmable Gate Arrays**". Xilinx Product Description (v1.6), 1999, 68 p.

[XIL01] Xilinx, Inc. "**Virtex 2.5V Field Programmable Gate Arrays**". Xilinx Product Specification (v2.5), 2001, 74 p.

[XIL04a] Xilinx, Inc. "**Development System Reference Guide – BitGen**". Xilinx, 2004, pp. 265-284.

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

[XIL04c] Xilinx, Inc. “**Development System Reference Guide – Modular Design**”. Xilinx, 2004, pp. 79-116.

[XIL04d] Xilinx, Inc. “**Virtex Series Configuration Architecture – User Guide**”. Xilinx Application Note 151 (XAPP151 v1.7), 2004, 45 p.

[YE03] Ye, T.; Benini, L.; De Micheli, G. “**Packetized On-Chip Interconnect Communication Analysis for MPSoC**”. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), 2003, pp. 344-349.

ANEXO I – DESCRIÇÃO VHDL DO ESTUDO DE CASO DE PALMA (SEÇÃO 3.4)

A Figura 103 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 *tristates*. A camada de *tristates* do lado do módulo é instanciada dentro do *bus_interface*. A camada de *tristates* 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*).

```
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 103 – Arquivo *top* da calculadora reconfigurável com dois módulos somadores.

ANEXO II – DESCRIÇÃO VHDL INICIAL DA CALCULADORA RECONFIGURÁVEL

Nesta Seção é apresentada a descrição VHDL completa do ponto de partida da calculadora reconfigurável, ou seja, ainda sem os componentes que habilitam a reconfigurabilidade do projeto. O projeto *top*, descrito na Figura 104A, instancia três módulos: o módulo de E/S (Figura 104B), denominado *capture*; o módulo adição de dois bits (Figura 105A), denominado *adder2bit*; o módulo adição de três bits (Figura 105B), denominado *adder3bit*.

```
01 entity top is
02 port(
03    reset:    in  std_logic;
04    clock:    in  std_logic;
05    input_A:  in  std_logic_vector(1 downto 0);
06    input_B:  in  std_logic_vector(1 downto 0);
07    input_C:  in  std_logic_vector(2 downto 0);
08    get_data: in  std_logic;
09    displayA: out std_logic_vector(6 downto 0)
10);
11 end top;
12
13 architecture arq_top of top is
14
15 signal dadoA: std_logic_vector(1 downto 0);
16 signal dadoB: std_logic_vector(1 downto 0);
17 signal dadoC: std_logic_vector(2 downto 0);
18 signal res1:  std_logic_vector(2 downto 0);
19 signal res2:  std_logic_vector(3 downto 0);
20
21 begin
22
23    Interface: entity capture
24    port map (
25           input_A => input_A,
26           input_B => input_B,
27           input_C => input_C,
28          output_A => dadoA,
29          output_B => dadoB,
30          output_C => dadoC,
31             reset => reset,
32             clock => clock,
33          get_data => get_data,
34             input => res2,
35          displayA => displayA
36     );
37
38    Module_1: entity adder2bit
39    port map (
40          input_A => dadoA,
41          input_B => dadoB,
42          output  => res1
43    );
44
45    Module_2: entity adder3bit
46    port map (
47          input_A => res1,
48          input_B => dadoC,
49          output  => res2
50    );
51
52 end arq_top;
```

(A)

```
01 entity capture is
02 port(
03    input_A:  in  std_logic_vector(1 downto 0);
04    input_B:  in  std_logic_vector(1 downto 0);
05    input_C:  in  std_logic_vector(2 downto 0);
06    output_A: out std_logic_vector(1 downto 0);
07    output_B: out std_logic_vector(1 downto 0);
08    output_C: out std_logic_vector(2 downto 0);
09    reset:    in  std_logic;
00    clock:    in  std_logic;
11    get_data: in  std_logic;
12    input:    in  std_logic_vector(3 downto 0);
13    displayA: out std_logic_vector(6 downto 0)
14 );
15 end capture;
16
17 architecture arq_capture of capture is
18 begin
19
20 process(clock, get_data, reset)
21 begin
22    if reset ='0' then
23          output_A <= (others=>'0');
24          output_B <= (others=>'0');
25          output_C <= (others=>'0');
26    else
27          if clock = '1' and clock'event then
28                if (get_data = '0') then
29                      output_A <= input_A;
30                      output_B <= input_B;
31                      output_C <= input_C;
32                end if;
33          end if;
34    end if;
35 end process;
36
37 -- decodificador do display utilizado na V2MB1000
38 with input select
39 displayA <=
40          "0111111" when "0000",
41          "0000110" when "0001",
42          "1011011" when "0010",
43          "1001111" when "0011",
44          "1100110" when "0100",
45          "1101101" when "0101",
46          "1111101" when "0110",
47          "0000111" when "0111",
48          "1111111" when "1000",
49          "1101111" when "1001",
50          "1000000" when "1111",
51          "0110000" when others;
52 end arq_capture;
```

(B)

Figura 104 – (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 105 – (A) código VHDL do módulo *adder2bit.vhd*; (B) código VHDL do módulo *adder3bit.vhd*.

ANEXO III – DESCRIÇÃO VHDL DA CALCULADORA RECONFIGURÁVEL PARA O PROJETO MODULAR

A seguir é apresentado o arquivo *top.vhd* com os componentes que permitem a reconfiguração do sistema a partir do fluxo de Projeto Modular. As linhas 40 a 50 inibem a remoção de alguns recursos que podem ser otimizados pelas ferramentas de síntese. As linhas 54 a 74 são responsáveis por criar constantes que podem ser manualmente restringidas a correta área reconfigurável do sistema. As linhas 76 a 97 apresentam a instanciação dos módulos *capture*, *adder2bit* e *adder3bit* já descritos no Anexo II. As linhas 99 a 225 apresentam a instanciação das Bus Macros.

```
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;
```

ANEXO IV – MACRO: CONSTRUÇÃO

O objetivo deste Anexo é apresentar os passos básicos de como criar uma macro, sem entretanto mostrar quais os componentes constituem a macro. 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 selecionado. 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 conectar os pinos dos componentes selecionados através de fios.
5. A seguir adicione os pinos externos de entrada e saída dos componentes editados e nomeie cada pino. Memorize os nomes dos pinos porque estes serão posteriormente utilizados quando a macro for instanciada.
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 HDL.
10. Após estes passos a ferramenta de síntese física estará apta a utilizar a macro como um componente já roteado.

ANEXO V – XDL

XDL (*Xilinx Design Language*) é uma linguagem de descrição física de projeto para dispositivos Xilinx e também uma ferramenta que transforma projetos descritos em formato proprietário (NCD e NMC) para a linguagem XDL.

A ferramenta XDL opera em linha de comando em 3 modos distintos: (*i*) provendo um relatório completo sobre os recursos disponíveis de um determinado dispositivo (**`xdl -report`**); (*ii*) convertendo arquivos NCD/NMC para XDL (**`xdl -ncd2xdl`**); (*iii*) convertendo arquivos XDL para NCD/NMC (**`xdl -xdl2ncd`**).

A linguagem XDL foi disponibilizada pela Xilinx para que os usuários possam examinar o conteúdo dos NCDs/NMCs e escrever as suas próprias ferramentas de acordo com as suas necessidades individuais. A descrição de um projeto em XDL contém informações de mapeamento, posicionamento e roteamento de um circuito desenvolvido. A Figura 106 apresenta o projeto de uma porta lógica "E" descrito em VHDL e em esquemático específico para dispositivos Xilinx, enquanto a Figura 107 apresenta a respectiva descrição da porta lógica "E" em linguagem XDL. A descrição XDL do projeto descreve nas linhas 6 a 62 a configuração dos pinos de E/S da porta lógica "E", das linhas 64 a 80 a configuração da *slice* que implementa a porta lógica "E" e das linhas 81 a 106 o roteamento dos fios que conectam os pinos de E/S à *slice*.

```
01 library  IEEE;
02 use ieee.std_logic_1164.all;
03
04 entity top is
05 port
06 (
07    a : in  std_logic;
08    b : in  std_logic;
09    c : out std_logic;
10 );
11 end top;
12
13 architecture arch of top is
14 begin
15    c <= a and b;
16 end arch;
```

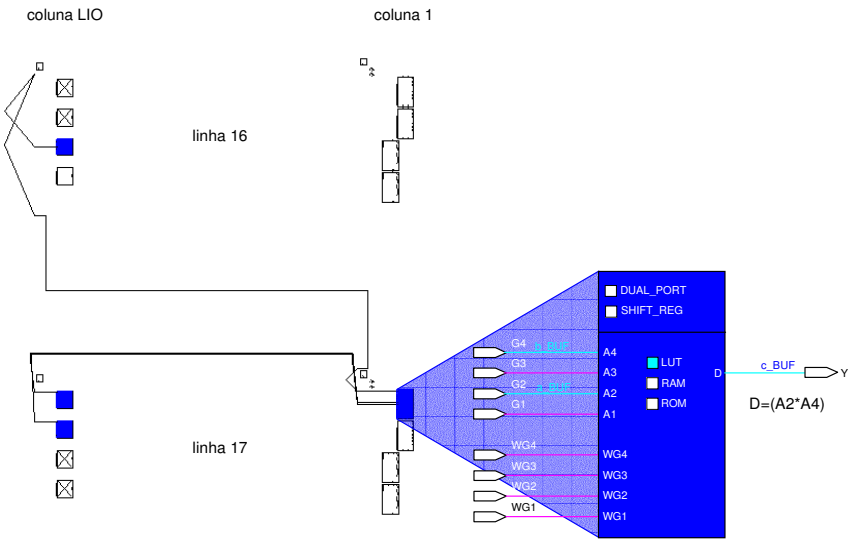


Figura 106 – Projeto de uma porta lógica "E" em VHDL e em esquemático para dispositivos Xilinx.

```
001 design "top" xc2v1000fg456-4 v2.38 ,
002  cfg "
003  _DESIGN_PROP::PIN_INFO:c c OUTPUT
004  _DESIGN_PROP::PIN_INFO:b b INPUT
005  _DESIGN_PROP::PIN_INFO:a a INPUT"
006 ;
007 inst "a" "IOB" , placed LIOIR17 K4 ,
008 cfg  "SRINV::#OFF REVINV::#OFF O2INV::#OFF OCEINV::#OFF OFF2::#OFF
009      OMUX::#OFF O1INV::#OFF T2INV::#OFF T1INV::#OFF TCEINV::#OFF
010      TMUX::#OFF OFF1::#OFF TFF2::#OFF TFF1::#OFF IFF2::#OFF
011      IFFATTRBOX::#OFF OFFATTRBOX::#OFF TFFATTRBOX::#OFF
012      ICLK2INV::#OFF IFF1::#OFF ICLK1INV::#OFF
013      TFF1_INIT_ATTR::#OFF TFF1_SR_ATTR::#OFF TFF2_SR_ATTR::#OFF
014      TFF2_INIT_ATTR::#OFF OFF1_SR_ATTR::#OFF
015      OFF1_INIT_ATTR::#OFF OFF2_SR_ATTR::#OFF
016      OFF2_INIT_ATTR::#OFF IFF1_SR_ATTR::#OFF
017      IFF1_INIT_ATTR::#OFF IFF2_SR_ATTR::#OFF
018      IFF2_INIT_ATTR::#OFF PULL::#OFF PAD:a: INBUF:a_IBUF:
019      T_USED::#OFF IOATTRBOX::LVTTL OSR_USED::#OFF TSR_USED::#OFF
020      ISR_USED::#OFF IREV_USED::#OFF OREV_USED::#OFF
021      TREV_USED::#OFF IDELMUX::1 IMUX::1 IFFDELMUX::#OFF
022      IFFDMUX::#OFF TSMUX::#OFF OTCLK1INV::#OFF OTCLK2INV::#OFF
023      ICEINV::#OFF GTSATTRBOX::#OFF SLEW::#OFF DRIVEATTRBOX::#OFF
024      DRIVE_0MA::#OFF"
025 ;
026 inst "b" "IOB" , placed LIOIR17 K3 ,
027  cfg "SRINV::#OFF REVINV::#OFF O2INV::#OFF OCEINV::#OFF OFF2::#OFF
028      OMUX::#OFF O1INV::#OFF T2INV::#OFF T1INV::#OFF TCEINV::#OFF
029      TMUX::#OFF OFF1::#OFF TFF2::#OFF TFF1::#OFF IFF2::#OFF
030      IFFATTRBOX::#OFF OFFATTRBOX::#OFF TFFATTRBOX::#OFF
031      ICLK2INV::#OFF IFF1::#OFF ICLK1INV::#OFF
032      TFF1_INIT_ATTR::#OFF TFF1_SR_ATTR::#OFF TFF2_SR_ATTR::#OFF
033      TFF2_INIT_ATTR::#OFF OFF1_SR_ATTR::#OFF
034      OFF1_INIT_ATTR::#OFF OFF2_SR_ATTR::#OFF
035      OFF2_INIT_ATTR::#OFF IFF1_SR_ATTR::#OFF
036      IFF1_INIT_ATTR::#OFF IFF2_SR_ATTR::#OFF
037      IFF2_INIT_ATTR::#OFF PULL::#OFF PAD:b: INBUF:b_IBUF:
038      T_USED::#OFF IOATTRBOX::LVTTL OSR_USED::#OFF TSR_USED::#OFF
039      ISR_USED::#OFF IREV_USED::#OFF OREV_USED::#OFF
040      TREV_USED::#OFF IDELMUX::1 IMUX::1 IFFDELMUX::#OFF
041      IFFDMUX::#OFF TSMUX::#OFF OTCLK1INV::#OFF OTCLK2INV::#OFF
042      ICEINV::#OFF GTSATTRBOX::#OFF SLEW::#OFF DRIVEATTRBOX::#OFF
043      DRIVE_0MA::#OFF"
044 ;
045 inst "c" "IOB" , placed LIOIR16 K6 ,
046  cfg "SRINV::#OFF REVINV::#OFF O2INV::#OFF OCEINV::#OFF OFF2::#OFF
047      OMUX::O1 O1INV::O1 T2INV::#OFF T1INV::#OFF TCEINV::#OFF
048      TMUX::#OFF OFF1::#OFF TFF2::#OFF TFF1::#OFF IFF2::#OFF
049      IFFATTRBOX::#OFF OFFATTRBOX::#OFF TFFATTRBOX::#OFF
050      ICLK2INV::#OFF IFF1::#OFF ICLK1INV::#OFF
051      TFF1_INIT_ATTR::#OFF TFF1_SR_ATTR::#OFF TFF2_SR_ATTR::#OFF
052      TFF2_INIT_ATTR::#OFF OFF1_SR_ATTR::#OFF
053      OFF1_INIT_ATTR::#OFF OFF2_SR_ATTR::#OFF
054      OFF2_INIT_ATTR::#OFF IFF1_SR_ATTR::#OFF
055      IFF1_INIT_ATTR::#OFF IFF2_SR_ATTR::#OFF
056      IFF2_INIT_ATTR::#OFF PULL::#OFF PAD:c: OUTBUF:c_OBUF:
057      T_USED::#OFF IOATTRBOX::LVTTL OSR_USED::#OFF TSR_USED::#OFF
058      ISR_USED::#OFF IREV_USED::#OFF OREV_USED::#OFF
059      TREV_USED::#OFF IDELMUX::#OFF IMUX::#OFF IFFDELMUX::#OFF
060      IFFDMUX::#OFF TSMUX::#OFF OTCLK1INV::#OFF OTCLK2INV::#OFF
061      ICEINV::#OFF GTSATTRBOX::#OFF SLEW::SLOW DRIVEATTRBOX::12
062      DRIVE_0MA::#OFF"
063 ;
064 inst "c_OBUF" "SLICE" , placed R17C1 SLICE_X1Y47 ,
065  cfg "CYSELF::#OFF CYSELG::#OFF COUTUSED::#OFF YUSED::0
066      XUSED::#OFF F5USED::#OFF YBMUX::#OFF CYINIT::#OFF
067      DYMUX::#OFF DXMUX::#OFF F::#OFF REVUSED::#OFF FXMUX::#OFF
068      SYNC_ATTR::#OFF SRFFMUX::#OFF FRY_SR_ATTR::#OFF FFX::#OFF
069      FFY::#OFF FFX_SR_ATTR::#OFF G_ATTR::#OFF DIG_MUX::#OFF
070      CY0G::#OFF GYMUX::G FXUSED::#OFF DIF_MUX::#OFF
071      SLICEWE0USED::#OFF F_ATTR::#OFF CY0F::#OFF DIGUSED::#OFF
072      SHIFTOUTUSED::#OFF BYOUTUSED::#OFF FFX_INIT_ATTR::#OFF
073      FFY_INIT_ATTR::#OFF G:c1:#LUT:D=(A2*A4) XBMUX::#OFF
074      BYINVOUTUSED::#OFF SLICEWE2USED::#OFF SLICEWE1USED::#OFF
075      WF1USED::#OFF WF2USED::#OFF WF3USED::#OFF WF4USED::#OFF
076      WG1USED::#OFF WG2USED::#OFF WG3USED::#OFF WG4USED::#OFF
077      YBUSED::#OFF BXOUTUSED::#OFF BYINV::#OFF BXINV::#OFF
078      CEINV::#OFF CLKINV::#OFF SRINV::#OFF SOPOUTUSED::#OFF
079      SOPEXTSEL::#OFF
080 PINMAP:48:0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,21,18,20,
19,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,
43,44,45,46,47"
```

Configuração da LUT

```
081 ;
082 net "a_IBUF" ,
083  outpin "a"                I                    ,
084  inpin  "c_OBUF"           G2                   ,
085  pip LIOIR17 I3_PINWIRE -> I3 ,
086  pip LIOIR17 I3 -> OMUX8 ,
087  pip R17C1 OMUX_E8 -> G2_B3 ,
088  pip R17C1 G2_B3 -> G2_B_PINWIRE3 ,
089  # net "a_IBUF" loads=1 drivers=1 pips=4 rtpips=0
090  ;
091 net "b_IBUF" ,
092  outpin "b"                I                    ,
093  inpin  "c_OBUF"           G4                   ,
094  pip LIOIR17 I2_PINWIRE -> I2 ,
095  pip LIOIR17 I2 -> OMUX13 ,
096  pip R17C1 OMUX_E13 -> G4_B3 ,
097  pip R17C1 G4_B3 -> G4_B_PINWIRE3 ,
098  # net "b_IBUF" loads=1 drivers=1 pips=4 rtpips=0
099  ;
100 net "c_OBUF" ,
101  outpin "c_OBUF"           Y                    ,
102  inpin  "c"                O1                   ,
103  pip R17C1 Y3 -> OMUX10 ,
104  pip LIOIR16 OMUX_NW10 -> IOIS_FAN_BX2 ,
105  pip LIOIR16 IOIS_FAN_BX2 -> O1_B1 ,
106  # net "c_OBUF" loads=1 drivers=1 pips=3 rtpips=0
107  ;
```

Figura 107 – Descrição XDL do projeto de uma porta lógica “E” apresentado na Figura 106.

ANEXO VI – DRIVERS DO MR2

Com o objetivo de reduzir o consumo em área do wrapper da MR2 para trafego de pacotes, o envio e recebimentos destes é tratado por software. A seguir são apresentados os drivers responsáveis por comunicar o processador MR2 com outros módulos conectados à NoC. Os drivers estão descritos em linguagem assembly do processador MR2 e comentados linha a linha.

```
001 ###############################################################################
002 # AREA DE DADOS                                                               #
003 ###############################################################################
004
005         .data       # Data declaration section
006 str:    .ascii      "OK"
007 ixTest: .word       handler_test_read       #0x00
008         .word       handler_test_write      #0x01
009         .word       handler_test_activate   #0x02
010         .word       handler_test_printf     #0x03
011         .word       handler_test_scanf      #0x04
012         .word       handler_test_default    #0x05
013         .word       handler_test_default    #0x06
014         .word       handler_test_default    #0x07
015         .word       handler_test_notify     #0x08
016         .word       handler_test_default    #0x09
017         .word       handler_test_default    #0x10
018         .word       handler_test_reconf     #0x11
019
020 ###############################################################################
021 # INICIALIZAO                                                                 #
022 ###############################################################################
023
024         .text             # Code declaration section
025  main:                    # Start of code section
026         la    $sp,0x3FFC
027         la    $ra,main
028         j     my_main
029
030 ###############################################################################
031 ### Activate(Target)                                                          #
032 ###############################################################################
033
034 activate:
035         la    $t1,0       # CONST PAYLOAD SIZE (WORDS)
036         la    $t2,2       # CONST ACTIVATE
037         lw    $t0,4($sp)  # Carrega parâmetro Target.
038         addiu $sp,$sp,-16 # Aloca espaço na pilha.
039         sw    $ra,0($sp)  # Empilha $ra.
040         sw    $t0,4($sp)  # Empilha target.
041         sw    $t1,8($sp)  # Empilha size do payload em words.
042         sw    $t2,12($sp) # Empilha command.
043         jal   sndpacket   # Chama montador de pacotes.
044         lw    $ra,0($sp)  # Recupera $ra.
045         addiu $sp,$sp,16  # Desaloca pilha.
046         jr    $ra         # Retorna.
047
048 ###############################################################################
049 ### Notify(Target)                                                            #
050 ###############################################################################
051
052 notify:
053         la    $t1,0       # CONST PAYLOAD SIZE (WORDS)
054         la    $t2,0x08    # CONST NOTIFY
055         lw    $t0,4($sp)  # Carrega parâmetro Target.
056         addiu $sp,$sp,-16 # Aloca espaço na pilha.
057         sw    $ra,0($sp)  # Empilha $ra.
058         sw    $t0,4($sp)  # Empilha target.
059         sw    $t1,8($sp)  # Empilha size do payload em words.
060         sw    $t2,12($sp) # Empilha command.
061         jal   sndpacket   # Chama montador de pacotes.
062         lw    $ra,0($sp)  # Recupera $ra.
063         addiu $sp,$sp,16  # Desaloca pilha.
064         jr    $ra         # Retorna.
065
```

```
066 ##############################################################################
067 ### Wait                                                                     #
068 ##############################################################################
069
070 wait:
071         la    $t1,0x08    # CONST NOTIFY
072         addiu $sp,$sp,-8  # Aloca pilha.
073         sw    $ra,0($sp)
074         sw    $t1,4($sp)
075         jal   waitpacket  # Aguarda retorno de algum lugar.
076         lw    $ra,0($sp)
077         addiu $sp,$sp,8
078         jr    $ra         # Retorna.
079
080 ##############################################################################
081 ### Read(Target, Address) = Data                                             #
082 ##############################################################################
083
084 read:   la    $t1,1       # CONST PAYLOAD SIZE (WORDS)
085         la    $t2,0x00    # CONST READ
086         la    $t4,0x09    # CONST READ RETURN
087         lw    $t0,4($sp)  # Carrega target
088         lw    $t3,8($sp)  # Carrega address
089         addiu $sp,$sp,-20 # Aloca espaço na pilha.
090         sw    $ra,0($sp)  # Empilha $ra.
091         sw    $t0,4($sp)  # Empilha target.
092         sw    $t1,8($sp)  # Empilha size do payload em words.
093         sw    $t2,12($sp) # Empilha command.
094         sw    $t3,16($sp) # Empilha address.
095         jal   sndpacket   # Chama montador de pacotes.
096         lw    $ra,0($sp)  # Recupera $ra.
097         addiu $sp,$sp,20  # Desaloca pilha.
098         addiu $sp,$sp,-12 # Aloca pilha ($ra + payload de retorno)
099         sw    $ra,0($sp)
100         sw    $t4,4($sp)
101         jal   waitpacket  # Aguarda retorno
102         lw    $ra,0($sp)
103         lw    $t0,8($sp)
104         addiu $sp,$sp,12
105         sw    $t0,12($sp) # Guarda retorno do read
106         jr    $ra         # Retorna.
107
108 ##############################################################################
109 ### Reconf(Módulo) = Address                                                 #
110 ##############################################################################
111
112 reconf: la    $t0,0x01    # CONST GERENCIADOR
113         la    $t1,0x01    # CONST PAYLOAD SIZE (WORDS)
114         la    $t2,0x11    # CONST RECONF
115         la    $t4,0x12    # CONST RECONF RETURN
116         lw    $t3,4($sp)  # Carrega módulo.
117         addiu $sp,$sp,-20 # Aloca espaço na pilha.
118         sw    $ra,0($sp)  # Empilha $ra.
119         sw    $t0,4($sp)  # Empilha target.
120         sw    $t1,8($sp)  # Empilha size do payload em words.
121         sw    $t2,12($sp) # Empilha command.
122         sw    $t3,16($sp) # Empilha módulo.
123         jal   sndpacket   # Chama montador de pacotes.
124         lw    $ra,0($sp)  # Recupera $ra.
125         addiu $sp,$sp,20  # Desaloca pilha.
126         addiu $sp,$sp,-12 # Aloca pilha.
127         sw    $ra,0($sp)
128         sw    $t4,4($sp)
129         jal   waitpacket  # Aguarda retorno
130         lw    $ra,0($sp)
131         lw    $t0,8($sp)
132         addiu $sp,$sp,12
133         sw    $t0,8($sp)  # Guarda retorno do read
134         jr    $ra         # Retorna.
135
```

```
136 ################################################################################
137 ### Write(Target, Address, Data)                                               #
138 ################################################################################
139
140 write:  la    $t1,2       # CONST PAYLOAD SIZE (WORDS)
141         la    $t2,1       # CONST WRITE
142         lw    $t0,4($sp)  # Carrega parâmetro target.
143         lw    $t3,8($sp)  # Carrega parâmetro address.
144         lw    $t4,12($sp) # Carrega parâmetro data.
145         addiu $sp,$sp,-24 # Aloca espaço na pilha.
146         sw    $ra,0($sp)  # Empilha $ra.
147         sw    $t0,4($sp)  # Empilha target.
148         sw    $t1,8($sp)  # Empilha size do payload em words.
149         sw    $t2,12($sp) # Empilha command.
150         sw    $t3,16($sp) # Empilha address.
151         sw    $t4,20($sp) # Empilha data.
152         jal   sndpacket   # Chama montador de pacotes.
153         lw    $ra,0($sp)  # Recupera $ra.
154         addiu $sp,$sp,24  # Desaloca pilha.
155         jr    $ra         # Retorna.
156
157 ################################################################################
158 ### Scanf() = Data                                                             #
159 ################################################################################
160
161 scanf:  la    $t0,1       # CONST TARGET
162         la    $t1,0       # CONST PAYLOAD SIZE (WORDS)
163         la    $t2,4       # CONST SCANF
164         addiu $sp,$sp,-16 # Aloca espaço na pilha.
165         sw    $ra,0($sp)  # Empilha $ra.
166         sw    $t0,4($sp)  # Empilha target.
167         sw    $t1,8($sp)  # Empilha size do payload em words.
168         sw    $t2,12($sp) # Empilha command.
169         jal   sndpacket   # Chama montador de pacotes.
170         lw    $ra,0($sp)  # Recupera $ra.
171         addiu $sp,$sp,16  # Desaloca pilha.
172         addiu $sp,$sp,-12 # Aloca pilha ($ra + payload)
173         sw    $ra,0($sp)
174         sw    $t2,4($sp)  # Parâmetro de comando.
175         jal   waitpacket  # Aguarda retorno do Scanf.
176         lw    $ra,0($sp)
177         lw    $t0,8($sp)
178         addiu $sp,$sp,12
179         sw    $t0,4($sp)  # Guarda retorno do scanf.
180         jr    $ra         # Retorna.
181
182 ################################################################################
183 ### Printf(Payload)                                                            #
184 ################################################################################
185
186 printf: la    $t0,1       # CONST TARGET
187         la    $t1,1       # CONST PAYLOAD SIZE (WORDS)
188         la    $t2,3       # CONST PRINTF
189         lw    $t3,4($sp)  # Carrega parâmetro.
190         addiu $sp,$sp,-20 # Aloca espaço na pilha.
191         sw    $ra,0($sp)  # Empilha $ra.
192         sw    $t0,4($sp)  # Empilha target.
193         sw    $t1,8($sp)  # Empilha size do payload em words.
194         sw    $t2,12($sp) # Empilha command.
195         sw    $t3,16($sp) # Empilha payload.
196         jal   sndpacket   # Chama montador de pacotes.
197         lw    $ra,0($sp)  # Recupera $ra.
198         addiu $sp,$sp,20  # Desaloca pilha.
199         jr    $ra         # Retorna.
200
```

```
201 ##############################################################################
202 ### WaitPacket(Command) = Payload                                           #
203 ##############################################################################
204
205 waitpacket:
206         addiu $t7,$sp,8   # Indice para armazenamento do payload.
207         lw    $t6,4($sp)  # Carrega código do comando.
208         la    $t5,0x0002  # Máscara.
209 waitpacket_init:
210         lw    $t4,0xFFF0  # Carrega bits de status do wrapper.
211         and   $t4,$t4,$t5 # Testa penúltimo bit.
212         beq   $t4,$zero,waitpacket_init
213         lw    $t2,0xFFF2  # Carrega Size
214         lw    $t3,0xFFF3  # Carrega Command
215         bne   $t3,$t6,waitpacket_wrong
216         beq   $t2,$zero,waitpacket_unlock
217 waitpacket_load:
218         lw    $t1,0xFFFF  # Carrega Payload
219         sw    $t1,0($t7)
220         addiu $t7,$t7,4
221         addiu $t2,$t2,-1
222         beq   $t2,$zero,waitpacket_end
223 waitpacket_wait:
224         lw    $t4,0xFFF0  # Carrega bits de status do wrapper.
225         and   $t4,$t4,$t5 # Testa penúltimo bit.
226         beq   $t4,$zero,waitpacket_wait
227         j     waitpacket_load
228 waitpacket_wrong:
229         beq   $t2,$zero,waitpacket_wrong_unlock
230 waitpacket_wrong_waste:
231         lw    $t1,0xFFFF  # Carrega Payload
232         addiu $t2,$t2,-1
233         beq   $t2,$zero,waitpacket_init
234 waitpacket_wrong_unlock:
235         lw    $t4,0xFFFF  # Desbloqueia.
236         j     waitpacket_init # Espera próximo pacote.
237 waitpacket_unlock:
238         lw    $t6,0xFFFF  # Desbloqueia.
239 waitpacket_end:
240         jr    $ra
241
242 ##############################################################################
243 ### SendPacket(Target, Size, Command, Payload)                               #
244 ##############################################################################
245
246 sndpacket:
247         li    $t5,0x0001
248
249         lw    $t0,4($sp)  # Carrega target.
250         lw    $t1,8($sp)  # Carrega size do payload em words.
251         lw    $t2,12($sp) # Carrega command.
252         addiu $t7,$sp,16  # $t7 será o inicio do payload.
253         sw    $t0,0xFFF1  # Seta target.
254         sw    $t1,0xFFF2  # Seta size do payload em words.
255         sw    $t2,0xFFF3  # Seta command.
256
257 sndpacket_wait:
258         beq   $t1,$zero,sndpacket_end
259         lw    $t6,0xFFF0  # Carrega bits de status do wrapper.
260         and   $t6,$t6,$t5 # Verifica ultimo bit (busy).
261         bne   $t6,$zero,sndpacket_wait
262         lw    $t3,0($t7)  # Carrega payload.
263         sw    $t3,0xFFFF  # Envia pacote.
264         addiu $t1,$t1,-1  # Diminui um no tamanho.
265         addiu $t7,$t7,4   # Endereça pro próximo pacote.
266         j     sndpacket_wait    # Espera.
267
268 sndpacket_end:
269         jr    $ra         # Retorna
```

ANEXO VII – DESCRIÇÃO VHDL DA CALCULADORA RECONFIGURÁVEL PARA O FLUXO PROPOSTO

A seguir é apresentado o arquivo *top.vhd* do estudo de caso da calculadora reconfigurável com os componentes que permitem a reconfiguração do sistema a partir do fluxo de projeto proposto no Capítulo 4. As linhas 153 a 231 apresentam as macros instanciadas responsáveis por fazer interface entre os módulos reconfiguráveis do sistema.

```
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 architecture arq_top of top is
014
015 component capture is
016 port (
017   input_A:        in  std_logic_vector(1 downto 0);
018   input_B:        in  std_logic_vector(1 downto 0);
019   input_C:        in  std_logic_vector(2 downto 0);
020   output_A:       out std_logic_vector(1 downto 0);
021   output_B:       out std_logic_vector(1 downto 0);
022   output_C:       out std_logic_vector(2 downto 0);
023   reset:          in  std_logic;
024   clock :         in  std_logic;
025   get_data:       in  std_logic;
026   input_reconfig: in  std_logic_vector(3 downto 0);
027   displayA:       out std_logic_vector(6 downto 0)
028 );
029 end component;
030
031 component adder2bit is
032 port(
033   input_A: in std_logic_vector(1 downto 0);
034   input_B: in std_logic_vector(1 downto 0);
035   output: out std_logic_vector(2 downto 0)
036 );
037 end component;
038
039 component adder3bit is
040 port(
041   input_A: in std_logic_vector(2 downto 0);
042   input_B: in std_logic_vector(2 downto 0);
043   output: out std_logic_vector(3 downto 0)
044 );
045 end component;
046
047 component LRmacro is
048 port(
049   i0: in std_logic;
050   i1: in std_logic;
051   i2: in std_logic;
052   i3: in std_logic;
053   i4: in std_logic;
054   i5: in std_logic;
055   i6: in std_logic;
056   i7: in std_logic;
057
058   o0: out std_logic;
059   o1: out std_logic;
060   o2: out std_logic;
061   o3: out std_logic;
062   o4: out std_logic;
063   o5: out std_logic;
064   o6: out std_logic;
065   o7: out std_logic
066 );
067 end component;
068
069 component RLmacro is
070 port(
071   i0: in std_logic;
072   i1: in std_logic;
073   i2: in std_logic;
074   i3: in std_logic;
075   i4: in std_logic;
076   i5: in std_logic;
077   i6: in std_logic;
078   i7: in std_logic;
079
080   o0: out std_logic;
081   o1: out std_logic;
082   o2: out std_logic;
083   o3: out std_logic;
084   o4: out std_logic;
085   o5: out std_logic;
086   o6: out std_logic;
087   o7: out std_logic
088 );
089 end component;
090
091 signal going_nowhere_1, going_nowhere_2 : std_logic;
092 signal going_nowhere_3, going_nowhere_4 : std_logic;
093 signal going_nowhere_5, going_nowhere_6 : std_logic;
094 signal going_nowhere_7, going_nowhere_8 : std_logic;
095 signal going_nowhere_9, going_nowhere_10: std_logic;
096 signal going_nowhere_11,going_nowhere_12: std_logic;
097 signal going_nowhere_13,going_nowhere_14: std_logic;
099 signal going_nowhere_15,going_nowhere_16: std_logic;
100 signal going_nowhere_17,going_nowhere_18: std_logic;
101 signal going_nowhere_19,going_nowhere_20: std_logic;
102 signal going_nowhere_21,going_nowhere_22: std_logic;
103
104 -- Fixo
105 signal dadoA_1    : std_logic_vector(1 downto 0);
106 signal dadoB_1    : std_logic_vector(1 downto 0);
107 signal dadoC_1    : std_logic_vector(2 downto 0);
108 signal resultado_3: std_logic_vector(3 downto 0);
109
110 -- Reconfig 1 Signals
111 signal dadoA_2    : std_logic_vector(1 downto 0);
112 signal dadoB_2    : std_logic_vector(1 downto 0);
113 signal dadoC_2    : std_logic_vector(2 downto 0);
114 signal dadoA_3    : std_logic_vector(2 downto 0);
115 signal resultado_2: std_logic_vector(3 downto 0);
116
117 -- Reconfig 2 Signals
118 signal dadoA_4    : std_logic_vector(2 downto 0);
119 signal dadoC_3    : std_logic_vector(2 downto 0);
120 signal resultado_1: std_logic_vector(3 downto 0);
121
122 begin
123
124 fixed : capture
125 port map (
126   input_A => input_A,
127   input_B => input_B,
128   input_C => input_C,
129   output_A => dadoA_1,
130   output_B => dadoB_1,
131   output_C => dadoC_1,
132   reset => reset,
133   clock => clock,
134   get_data => get_data,
135   input_reconfig => resultado_3,
136   displayA => displayA
137 );
138
```

```
139 reconf1: adder2bit
140 port map (
141   input_A => dadoA_2,
142   input_B => dadoB_2,
143   output => dadoA_3
144 );
145
146 reconf2: adder3bit
147 port map (
148   input_A => dadoA_4,
149   input_B => dadoC_3,
150   output => resultado_1
151 );
152
153 LRmacro1: LRmacro port map (
154   i0 => dadoA_1(0),
155   i1 => dadoA_1(1),
156   i2 => dadoB_1(0),
157   i3 => dadoB_1(1),
158   i4 => dadoC_1(0),
159   i5 => dadoC_1(1),
160   i6 => dadoC_1(2),
161   i7 => going_nowhere_1,
162
163   o0 => dadoA_2(0),
164   o1 => dadoA_2(1),
165   o2 => dadoB_2(0),
166   o3 => dadoB_2(1),
167   o4 => dadoC_2(0),
168   o5 => dadoC_2(1),
169   o6 => dadoC_2(2),
170   o7 => going_nowhere_2
171 );
172
173 LRmacro2: LRmacro port map (
174   i0 => dadoA_3(0),
175   i1 => dadoA_3(1),
176   i2 => dadoA_3(2),
177   i3 => dadoC_2(0),
178   i4 => dadoC_2(1),
179   i5 => dadoC_2(2),
180   i6 => going_nowhere_3,
181   i7 => going_nowhere_4,
182
183   o0 => dadoA_4(0),
184   o1 => dadoA_4(1),
185   o2 => dadoA_4(2),
186   o3 => dadoC_3(0),
187   o4 => dadoC_3(1),
188   o5 => dadoC_3(2),
189   o6 => going_nowhere_5,
190   o7 => going_nowhere_6
191 );
192
193 RLmacro1: RLmacro port map (
194   i0 => resultado_1(0),
195   i1 => resultado_1(1),
196   i2 => resultado_1(2),
197   i3 => resultado_1(3),
198   i4 => going_nowhere_7,
199   i5 => going_nowhere_8,
200   i6 => going_nowhere_9,
201   i7 => going_nowhere_10,
202
203   o0 => resultado_2(0),
204   o1 => resultado_2(1),
205   o2 => resultado_2(2),
206   o3 => resultado_2(3),
207   o4 => going_nowhere_11,
208   o5 => going_nowhere_12,
209   o6 => going_nowhere_13,
210   o7 => going_nowhere_14
211 );
212
213 RLmacro2: RLmacro port map (
214   i0 => resultado_2(0),
215   i1 => resultado_2(1),
216   i2 => resultado_2(2),
217   i3 => resultado_2(3),
218   i4 => going_nowhere_15,
219   i5 => going_nowhere_16,
220   i6 => going_nowhere_17,
221   i7 => going_nowhere_18,
222
223   o0 => resultado_3(0),
224   o1 => resultado_3(1),
225   o2 => resultado_3(2),
226   o3 => resultado_3(3),
227   o4 => going_nowhere_19,
228   o5 => going_nowhere_20,
229   o6 => going_nowhere_21,
230   o7 => going_nowhere_22
231 );
232
233 end arq_top;
```