PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA

ARQUITETURAS AUTO-RECONFIGURÁVEIS EM SISTEMAS DIGITAIS

Prof. Dr. Ney Laert Vilar Calazans
Orientador

Marcelo Sarmento
Trabalho de conclusão II

Bacharelado em Informática

Porto Alegre, Junho de 2001

ARQUITETURAS AUTO-RECONFIGURÁVEIS EM SISTEMAS DIGITAIS

Marcelo Sarmento

RESUMO

O presente trabalho é um estudo sobre a aplicação de Arquiteturas Reconfiguráveis na resolução de problemas computacionais, com enfoque nas características de reconfiguração dinâmica que podem ser implementadas com essa tecnologia. Durante o seu desenvolvimento, alguns conceitos sobre dispositivos reconfiguráveis, especialmente FPGAs, são abordados. Arquiteturas Reconfiguráveis e suas classificações são revistas. A utilização da densidade funcional como ferramenta de comparação em relação às arquiteturas convencionais é apresentada.

O objetivo do trabalho é a construção de um protótipo de aplicação de Arquiteturas Auto-Reconfiguráveis. Os componentes necessários para a construção desta, tanto a nível de Hardware (uma placa protótipo com recursos de reconfiguração) como de Software (uma aplicação de gerenciamento deste hardware) são desenvolvidos ao longo deste trabalho.

Por fim, são apresentadas as conclusões deste trabalho, bem como sugestões para trabalhos futuros relacionados à Arquiteturas Reconfiguráveis.

ABSTRACT

This work presents a study on the application of Run-Time Reconfigurable systems and their use for the resolution of computational application-specific problems, particulary those related with dynamic reconfiguration. During this study, some concepts about reconfigurable hardware are revised, especially those related whit RAM-based FPGAs. Also, it provides some insights on the characterization and classification of Run-Time Reconfigurable systems. The use of functional density as a metric for balancing the advantages of RTR against its associated reconfiguration costs and against the conventional approaches is presented.

The required components to build the prototype application (a prototype board with hardware resources, and a software application to manage this hardware) are parts of the development of this study.

Finally, the conclusions about this work are presented, as well as some suggestions for future works related with Run-Time Reconfigurable systems.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

INTRODUÇÃO

O presente trabalho é um estudo sobre a aplicação de Arquiteturas Reconfiguráveis na resolução de problemas computacionais, com enfoque nas características de reconfiguração dinâmica que podem ser implementadas com essa tecnologia. Esta técnica, conhecida como RTR (do inglês, *Run-Time Reconfiguration*), possibilita a reconfiguração do dispositivo durante a sua operação, e permite a redução dos recursos de hardware necessários para a execução de tarefas computacionais específicas.

Durante o seu desenvolvimento, diversas etapas compuseram este trabalho:

- Pesquisa da tecnologia envolvida e dos produtos disponíveis no mercado ;
- Estudo de casos e implementações em projetos similares;
- Projeto de uma plataforma de hardware para a aplicação desta tecnologia, bem como das ferramentas de software necessárias para sua utilização;
- Desenvolvimento de uma aplicação utilizando a plataforma a ser desenvolvida e as técnicas pesquisadas.

Este trabalho de conclusão é apresentado da seguinte forma:

Ainda nesta Introdução, apresenta-se o escopo do trabalho, uma breve descrição do histórico dos tópicos abordados e a motivação que originou este estudo. O Capítulo 2 traz as definições e conceitos necessários para à leitura deste texto. O Capítulo 3 descreve os aspectos de arquiteturas que utilizam técnicas de reconfiguração, suas características, possíveis aplicações e comparações em relação aos modelos tradicionais. No Capítulo 4, é feito um detalhamento da implementação realizada, desde a sua construção, funcionamento e ferramentas utilizadas. Os resultados, a experiência adquirida e as dificuldades encontradas, bem como sugestões para a continuidade deste estudo e de outros relacionados com o assunto abordado compõem a Conclusão deste trabalho. Finalizando, Referências Bibliográficas relacionam os autores citados no texto, bem como as demais fontes de pesquisa utilizadas.

Diagramas, listagens dos programas fontes utilizados e demais materiais de apoio utilizados durante o desenvolvimento deste projetos são encontrados nos Anexos.

1.1 Escopo

Durante as ultimas décadas, o projeto de componentes digitais tem se tornado uma peça chave para a construção de sistemas computacionais. Neste período, muitos projetistas têm encontrado um problema freqüente, que consiste em estabelecer a relação correta entre generalidade e velocidade. Um componente pode ser construído visando versatilidade, possibilitando a execução de diferentes aplicações, ou visando desempenho, limitando o dispositivo a um número menor de tarefas, mas com maior velocidade.

Microprocessadores como o Intel Pentium [1] ou o Motorola Power PC [2] são exemplos de componentes de propósito geral. Eles possuem um conjunto de instruções de programação codificadas, cuja combinação pode levar a execução de virtualmente qualquer tarefa computável, segundo modelos teóricos, tais como a máquina de Turing [3].

Por outro lado, componentes dedicados, geralmente conhecidos como ASICs (do inglês, *Application Specific Integrated Circuits*) ou Circuitos Integrados para uma Aplicação Específica, são construídos visando a funcionalidade especifica de uma determinada tarefa. Um chip de aceleração gráfica para a plataforma PC de microcomputadores, por exemplo, pode desenhar linhas ou executar movimentação de objetos na tela 10 a 100 vezes mais rápido do que um microprocessador executando instruções genéricas.

Apesar da maior velocidade, a utilização dos ASICs fica restrita apenas à resolução ou operacionalização do problema para o qual eles foram desenvolvidos, visto que o seu hardware é tradicionalmente fixo. Uma pequena modificação dos requisitos ou a inclusão de uma nova necessidade requer o desenvolvimento de um novo componente.

Contínuos avanços na tecnologia de desenvolvimento de circuitos integrados possibilitaram uma terceira opção, agregando a programação da funcionalidade característica dos sistemas que utilizam microprocessadores com o desempenho dos

ASICs. *Field Programmable Gate Arrays*, ou FPGAs, consistem em uma matriz de elementos agrupados em blocos lógicos configuráveis, que podem ser interconectados de um modo genérico por barramentos de conexões também configuráveis.

A Figura 1 demonstra a relação entre essas tecnologias.

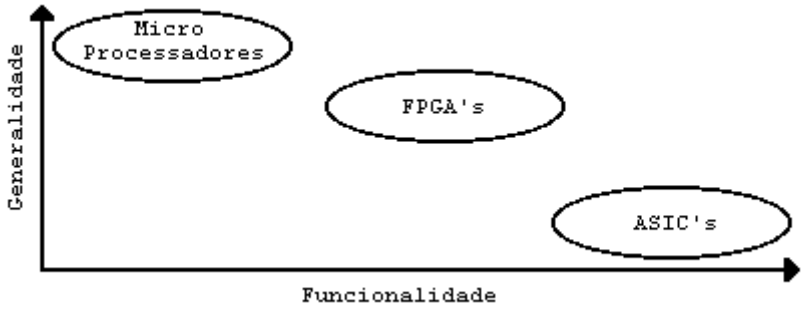


Figura 1 - Gráfico Comparativo Desempenho-Funcionalidade

Sistemas computacionais podem fazer uso da capacidade de reconfiguração dos FPGAs de várias maneiras. Uma das técnicas que estão sendo pesquisadas atualmente e a utilização de FPGAs para construção de dispositivos de hardware cuja funcionalidade pode ser alterada durante o uso do sistema. Esses dispositivos [4], em conjunto com um processador de propósito geral (ou GPP, do inglês *General Purpose Processsor*) e uma estrutura de memória compartilhada (opcional), constituem uma *Arquitetura Reconfigurável.*

Neste contexto, um único hardware (FPGA), pode executar uma seqüência de diferentes tarefas através da reconfiguração dos seus blocos lógicos sob demanda: o equivalente em hardware da execução de um programa, sua remoção da memória do computador e a execução de outro. Uma consideração a respeito da utilização deste modelo é feita em [5]: a especialização destas arquiteturas reconfiguráveis tem melhor desempenho que o seu equivalente em microprocessadores convencionais.

Estas arquiteturas também são descritas como reconfiguráveis dinamicamente, segundo a classificação de Sanchez [6]. Uma implementação destas arquiteturas pode utilizar tanto dispositivos configurados parcialmente como a alocação de um mesmo dispositivo com configurações distintas em instantes de tempo diferentes.

Apesar de alguns dispositivos disponíveis no mercado oferecerem a possibilidade de reconfiguração parcial, essa característica não será abordada nesse trabalho. O estudo estará baseado em dispositivos que necessitam a reconfiguração total sempre que seja necessária a modificação da função que ele deve executar.

1.2 Motivação

Segundo Villasenor e Smith [7], "Uma arquitetura de hardware que tem a possibilidade de reconfigurar a si mesma dinamicamente à medida em que uma tarefa é executada, refinando a sua própria configuração para obter uma melhor performance é a forma mais desafiadora e potencialmente mais poderosa de um sistema computacional configurável".

Tomando como base essa afirmação, o objetivo deste trabalho é implementar uma aplicação que exemplifique a utilização de uma *Arquitetura Auto-Reconfigurável*, verificando as vantagens e desvantagens em relação aos sistemas convencionais.

Dado um determinado problema, pretende-se construir o algoritmo para a resolução desse problema, dividir o algoritmo em etapas não simultâneas (quando possível), e implementar cada uma destas etapas em seu equivalente em hardware.

Na Figura 2, é apresentado o formato convencional para o mapeamento de um algoritmo em hardware. Utilizando uma linguagem de descrição de hardware, como VERILOG, AHDL ou VHDL e sua posterior síntese, é possível configurar um dispositivo FPGA para executar a função desejada. Neste modelo, o hardware executa o mesmo algoritmo até que o agente configurador seja acionado novamente. Como um FPGA tem um número limitado de blocos configuráveis, algoritmos mais complexos necessitam dispositivos maiores (e, por conseqüência, de custo mais elevados) ou o uso de múltiplos dispositivos.

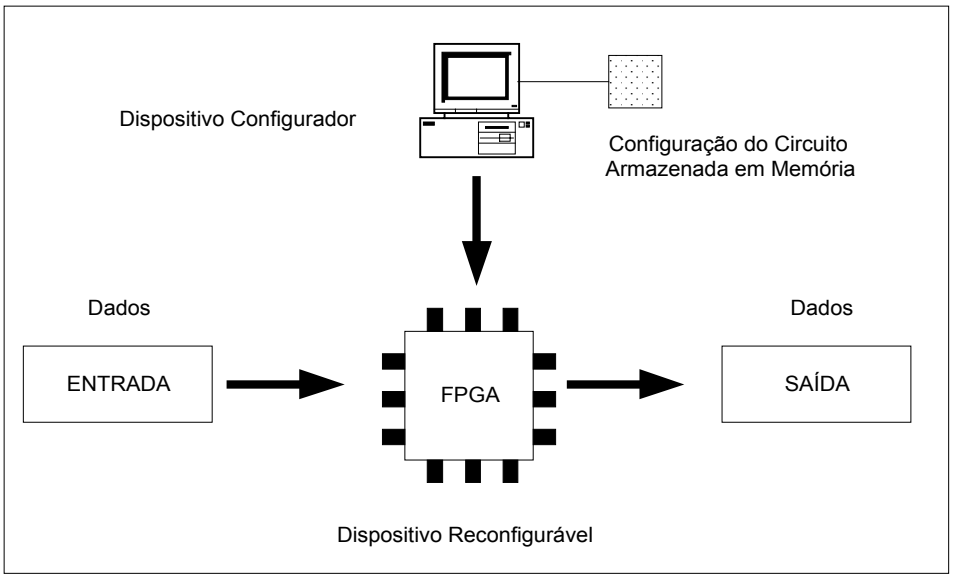


Figura 2 - Modelo convencional para configuração de um FPGA

Na Figura 3, é apresentada a proposta de uma arquitetura auto-reconfigurável destinada à resolução do mesmo problema. Neste caso, o algoritmo é "quebrado" em partes menores, sintetizado, e cada uma das suas partes é armazenada em uma memória de configuração.

O agente configurador faz a programação inicial, e a partir deste momento o próprio sistema determina qual o próximo módulo a ser carregado, seguindo a seqüência do algoritmo ou de acordo com a demanda dos dados recebidos.

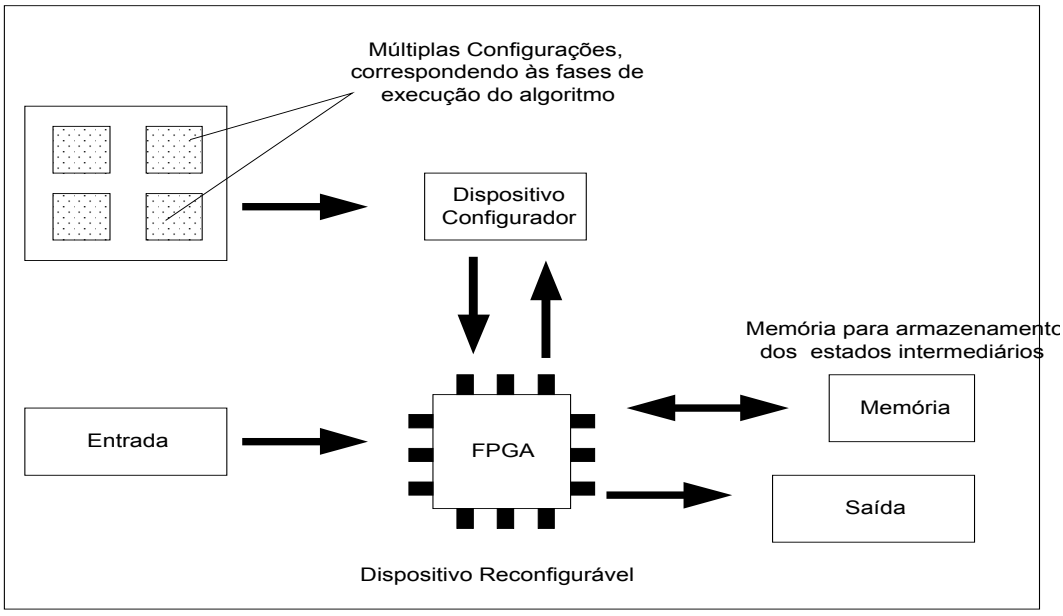


Figura 3 - Arquitetura Auto-Reconfigurável Proposta

Em vez de utilizar uma arquitetura *estática* (arquiteturas convencionais, sem possibilidade de configuração) para o processamento de todas as variações possíveis de um determinado algoritmo, pequenas partições do mesmo algoritmo podem ser usadas para resolver o mesmo problema com maior eficiência utilizando recursos limitados de hardware de um FPGA.

Este recurso de reconfiguração do dispositivo na medida em que é necessária uma nova função oferece outras vantagens além da flexibilidade, já que o FPGA pode ser reconfigurado não apenas antes da execução da aplicação, mas durante o processo. Esta técnica permite a especialização dos recursos do circuito durante o processamento da tarefa, o que pode trazer um aumento de eficiência do circuito reconfigurável dinamicamente como um todo.

Uma das maneiras de aumentar essa eficiência é usar as técnicas de RTR para substituir partes do hardware que estão inativas ou sem uso por outras mais úteis durante a utilização. Esta otimização do circuito permite que o processamento aconteça utilizando um menor número de recursos de hardware.

Além do teste da funcionalidade do circuito reconfigurável dinamicamente, também pretende-se verificar a vantagem (ou desvantagem) em termos do tempo total de processamento na resolução de um determinado problema em relação aos sistemas convencionais, utilizando-se a Densidade Funcional [30] como métrica.

Exemplos de aplicações e dos resultados alcançados através da utilização desta técnica são apresentados na Seção 3.

2 DEFINIÇÕES E CONCEITOS

2.1 Hardware Reconfigurável

Contínuos avanços na tecnologia de circuitos integrados possibilitaram o desenvolvimentos de dispositivos reconfiguráveis. Tradicionalmente, as funções de um circuito integrado são definidas durante a sua fase de projeto e posterior industrialização. Com os recentes *Field-programmable gate arrays* (FPGAs), as funções de um circuito integrado podem ser modificadas, através da execução de um programa. Os FPGAs modificaram a tradicional distinção entre hardware e software, já que enquanto um programa pudesse ser modificado até mesmo em tempo real, a estrutura do hardware era invariável. Com a tecnologia de FPGA é possível configurar as interconexões dos circuitos, compostos de portas lógicas, mesmo depois da industrialização do circuito integrado, modificando assim a sua função.

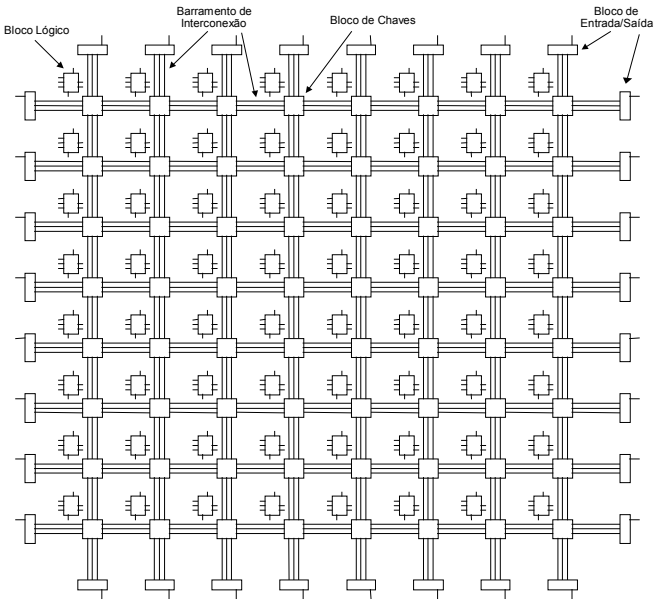


Figura 4 - Arquitetura de um FPGA genérico

A arquitetura de um FPGA, ilustrada pela Figura 4, consiste em uma matriz de elementos agrupados em *Blocos Lógicos* configuráveis, que podem ser interconectados, de um modo genérico, por Barramentos de Interconexão configuráveis. Semelhante a uma PAL (*Programmable Array Logic*), as interconexões entre os elementos são implementadas por *Blocos de Chaves* programáveis pelo usuário. Através de *Blocos de*

Entrada/Saída também configuráveis é realizado a comunicação com o mundo externo ao circuito.

2.1.1 Tecnologias Configuráveis

Em um FPGA, a matriz de blocos lógicos é conectada internamente e programada eletricamente através de chaves eletrônicas configuráveis. As propriedades destas chaves, tais como; tamanho, resistência (em ohms), e capacitância (em farads), delimitam as características elétricas destes circuitos integrados. Nas Seções seguintes serão descritos três tecnologias de chaves mais freqüentemente utilizadas [8].

2.1.1.1 Tecnologia de Configuração SRAM

A tecnologia SRAM (*Static Random Access Memory*) usa bits de memória RAM estática para programar e controlar as chaves eletrônicas, baseadas em transistores de tecnologia CMOS (*Complementary Metal Oxide Semiconductor*) ou multiplexadores como ilustra a Figura 5.

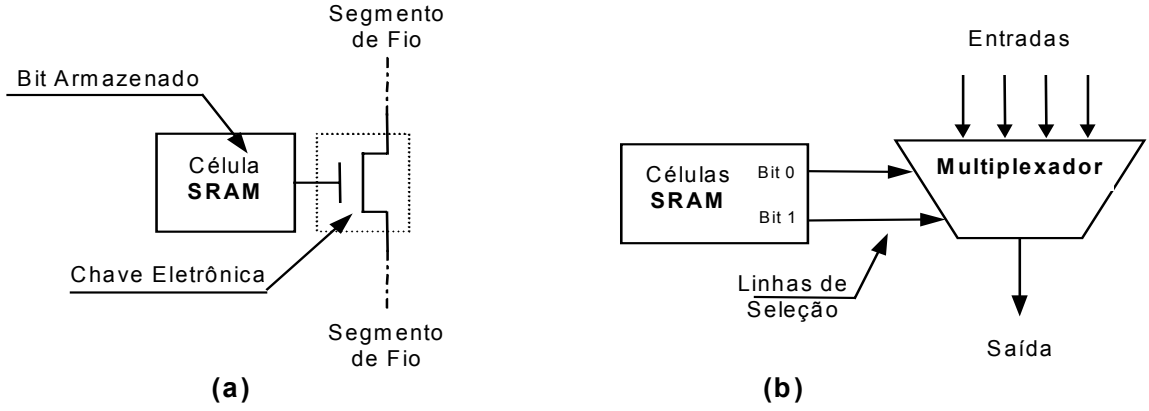


Figura 5 - (a) Chave Eletrônica, (b) Multiplexador

Quando um bit é armazenado na célula SRAM vista na Figura 5 (a) a chave eletrônica funciona como um circuito aberto ou fechado, conforme o valor lógico ("0" ou "1") do bit armazenado, desta forma a chave é usada para fazer uma conexão entre

dois segmentos de fios. Quando um "0" é armazenado, a chave está aberta, e o transistor apresenta uma resistência alta, a qual impede a conexão entre os dois segmentos.

No caso de multiplexador, da Figura 5 (b), os estados das células SRAM estão conectados as linhas de seleção, e estas selecionam uma das entradas do multiplexador para ser conectada à saída. Considerando que a SRAM é volátil, o FPGA deverá ser configurado no momento que for energizado. Isto exige uma memória externa permanente como PROM, EPROM, EEPROM ou outro meio de armazenamento, para prover o vetor de configuração, como será visto na Seção 2.2.2.5.

A necessidade de uma grande área na pastilha de silício que compõe o circuito integrado FPGA é a maior desvantagens da tecnologia de programação SRAM. Consome-se cinco transistores para implementar uma célula de SRAM e um transistor para a chave programável. Porém, esta tecnologia tem duas vantagens principais, a reprogramabilidade rápida, devida às características das SRAMs e a possibilidade de utilizar-se um processo relativamente simples de fabricação do circuito integrado. Ela é usada nos dispositivos da Xilinx [9], Plessey [10], Algotronix [11], Concurrent Logic [12] e Toshiba [13].

2.1.1.2 Tecnologia de Configuração Antifusível

Um *antifusível* é um dispositivo que apresenta uma resistência muito alta entre seus terminais. Quando uma tensão de 11 a 20 volts (dependendo do tipo de antifusível) é aplicada através de seus terminais, o antifusível "queima," criando uma ligação (contrário ao fusível convencional que se interrompe) de baixa de resistência e esta ligação torna-se permanente, conforme ilustrado na Figura 6.

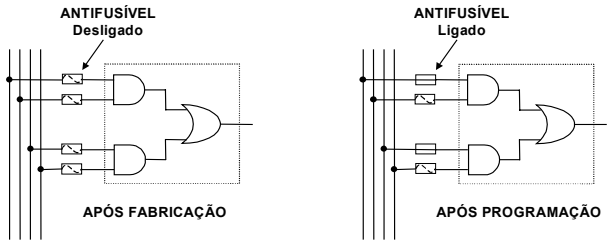


Figura 6 - Tecnologia de Programação Anti-Fusível

A programação de um anti-fusível requer um circuito extra para fornecer a tensão de programação e uma corrente de 5 mA ou mais. Isto é uma desvantagem, pois exige transistores de maior potência para prover a queima de cada anti-fusível. Uma vantagem do antifusível é seu tamanho pequeno. Outra é sua resistência série relativamente baixa e uma pequena capacitância parasita (para o antifusível não configurado), que é significativamente baixa em relação as outras tecnologias de configuração. Dois tipos de antifusíveis são mais utilizados, os fabricados pelo processo Oxigênio-Nitrogênio-Oxigênio (ONO) e o de Silício amorfo. A tecnologia de antifusível é usada nos dispositivos FPGAs da Actel [14], Quicklogic [15] e Crosspoint [16].

2.1.1.3 Tecnologia de Configuração de Porta Flutuante

Esta tecnologia é baseada em configuração por armazenamento de cargas, da mesma forma que utilizada nas EPROM (*Erasable Programmable ROM)* e EEPROM (*Electrically Erasable Programmable ROM*). Cada bit da memória, conforme ilustra a Figura 7, possui um transistor MOS (*Metal Oxide Semiconductor***)** com duas portas*,* uma delas flutuante, não conectada ao barramento da memória (S_0) e isolada por material de muito alta impedância.

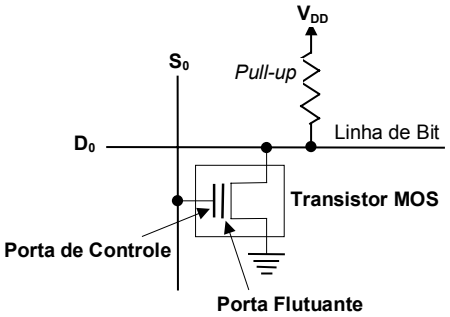


Figura 7 - Tecnologia de Programação Porta Flutuante

Em estado "desligado", como são fabricado, esses transistores não conduzem quando selecionados (S_0) e o conteúdo das posições de memória (D_0) é levado a "1" por resistores de *pull-up*. Para gravar um valor "0" em determinada posição, aplica-se uma tensão elevada, entre a porta de controle (não flutuante) e o dreno, o que causa uma

ruptura no material isolante e permite o acúmulo de cargas na porta flutuante, as quais permanecem, mesmo após o término do pulso de tensão, devido à alta impedância do material isolante. A presença dessas cargas na porta flutuante mantém o transistor em condução quando a posição (S_0) daquele bit, for selecionada. Com isso, a linha de bit é levada para "0". Estas cargas podem ser removidas da porta flutuante expondo-a à luz UV (Ultra Violeta), desfazendo-se desta forma a configuração.

Como no caso de uso de SRAM, uma principal vantagem da tecnologia de porta flutuante é sua reconfigurabilidade. Entretanto, esta tecnologia tem uma vantagem a mais, não é necessária memória externa permanente para programação do conjunto de bits de configuração no momento da inicialização. Porém, a tecnologia de porta flutuante requer três passos a mais do que o processo usual de fabricação CMOS. Duas outras desvantagens são: a alta resistência do transistor quando no estado de condução, e o alto consumo de energia devido ao resistor de *pull-up*, que é conectado à fonte de alimentação (V_{DD}) do dispositivo FPGA.

A tecnologia de porta flutuante é baseada em EEPROM, é usada em dispositivos fabricados pela AMD [17], Lattice [18] e Altera [19]. Esta tecnologia é semelhante a das EPROMs, a não ser pela remoção das cargas da *porta flutuante*, que pode ser feita eletricamente, no circuito, sem luz UV. Isto traz uma vantagem a mais, a fácil reconfigurabilidade, que pode ser muito útil em algumas aplicações. Porém, existe uma desvantagem, a célula de EEPROM é aproximadamente duas vezes o tamanho de uma célula de EPROM.

2.1.2 Arquitetura de Blocos Lógicos

Em alguns FPGAs, um bloco lógico configurável corresponde a uma estrutura semelhante a uma PAL (Programmable Array Logic). Uma definição possível para o bloco lógico é uma estrutura complexa contendo vários circuitos combinacionais, de múltiplas entradas e uma ou mais saídas. A maioria dos blocos lógicos também contém algum tipo de dispositivo de armazenamento, para viabilizar a implementação de circuitos seqüenciais. A Figura 8 ilustra exemplos de blocos lógicos.

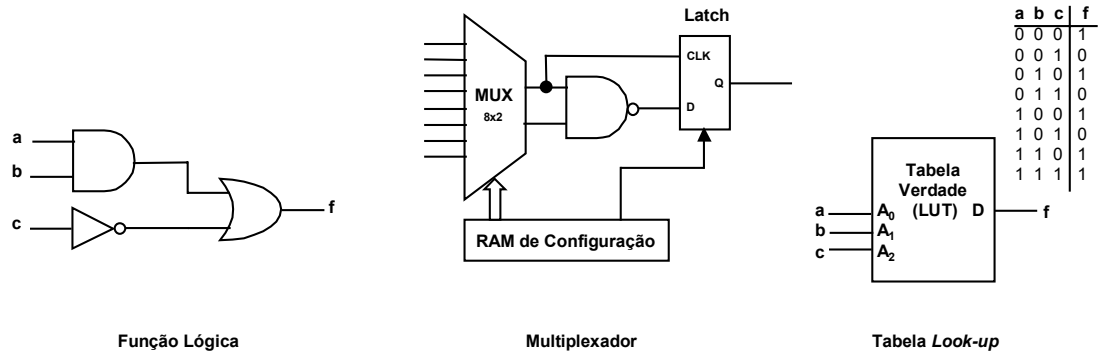


Figura 8 - Exemplos de Estruturas de Blocos Lógicos

Como ilustram os exemplos da Figura 8, em alguns FPGAs os blocos lógicos possuem estrutura bastante simples, tal como uma porta NAND de duas entradas. Em outros, são uma estrutura bem mais complexa, tais como tabelas verdade de 3, 4 ou 5 entradas, denominadas de *look-up tables* (LUTs).

Uma LUT é implementada como um registrador com 2^n biestáveis e um esquema de seleção de um dos bits deste registrador para colocar na saída. O controle desta seleção é realizado pelas variáveis de entrada da função, como ilustrado na Figura 9 para o caso de tabelas de 3 variáveis. Este tipo de bloco lógico possui dimensão relativamente elevada, mas fornece a máxima flexibilidade.

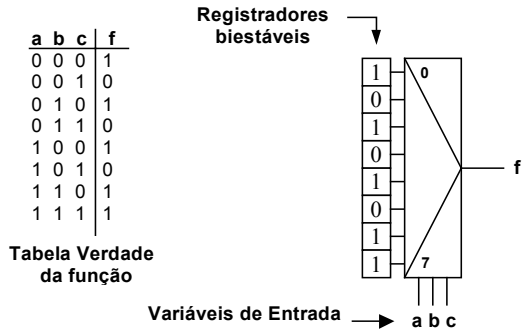
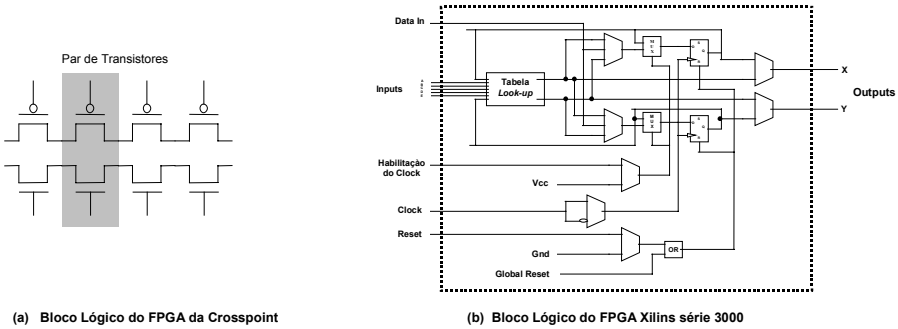


Figura 9 - LUT para tabelas de 3 variáveis

Rose, em [8], observa que os blocos lógicos em FPGAs, como ilustra a Figura 10, diferem grandemente em tamanho e capacidade de implementação. O bloco lógico

de dois transistores usado no FPGA da Crosspoint [16], pode implementar só um inversor, porém é muito pequeno em tamanho, enquanto que o bloco lógico implementado com LUTs, usado nos FPGAs da Xilinx série 3000 pode implementar qualquer função lógica de até cinco variáveis, mas é significativamente maior em tamanho.



(a) Bloco Lógico do FPGA da Crosspoint (b) Bloco Lógico do FPGA Xilins série 3000

Figura 10 - Blocos Lógicos (a) Crosspoint (b) Xilinx série 3000

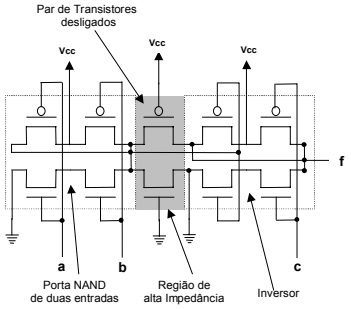
Observando as diferentes estruturas que constituem os blocos lógicos, Rose [8] classifica as arquiteturas dos FPGAs pela sua granularidade. *Granularidade* pode ser definido de vários modos. Exemplos são o número de funções booleanas que o bloco lógico pode implementar, o número equivalente de portas NAND de duas entradas, o número total de transistores, o total de área normalizada, ou o número de entradas e saídas.

O tópico granularidade em uma arquitetura FPGA é bastante complexo. Em algumas arquiteturas, tais como a dos FPGAs Altera [19] ou a dos FPGAs AMD [17], a lógica e o roteamento estão fortemente mesclados e é difícil separar as diferentes características de granularidade para estas arquiteturas. Para simplificar, Rose escolhe classificar a granularidade das arquiteturas comerciais em duas categorias: grão fino e grão grosso.

2.1.2.1 Arquitetura de Grão Fino

O FPGA fabricado pela Crosspoint [16] usa um único par de transistores no bloco lógico, como ilustrado na Figura 10(a). A função $f = ab + \overline{c}$ é implementada com um par de transistores conforme mostrado na Figura 11. Considerando que os

transistores são conectados lado a lado, alternando filas de transistores P e transistores N, portas lógicas CMOS quaisquer podem ser configuradas, bastando desligar um par de transistores nos seus limites laterais para criar uma região de isolamento entre portas adjacentes.



5 blocos lógicos configuráveis

Figura 11 - FPGA Crosspoint configurado para a função $f = ab + \overline{c}$

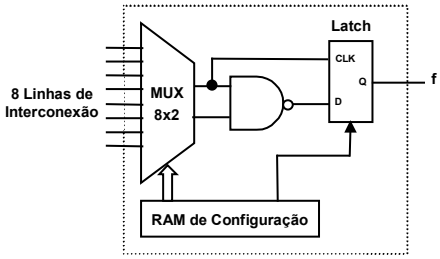


Figura 12 - Bloco Lógico do FPGA Plessey

Um segundo exemplo de uma arquitetura de grão fino é o FPGA de Plessey [10]. Nesta, o bloco básico é uma porta NAND de duas entradas como ilustrado na Figura 12. A lógica é formada do modo habitual conectando-se as entradas da NAND por meio de um multiplexador 8x2 para implementar a função $f = ab + \overline{c}$ desejada. A função é definida na Figura 13(a), e o bloco lógico é configurado para implementar, a função equivalente mostrada na Figura 13(b). Se o *Latch* na saída do bloco não é necessário, então a RAM de configuração é programada para mante-lo permanentemente transparente.

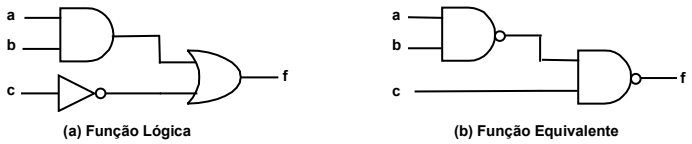


Figura 13 - (a) Função $f = ab + \overline{c}$ e (b) Função equivalente

Vários outros fabricantes usam blocos de grão fino. A Algotronix [11] usa um bloco lógico que pode executar qualquer função de duas variáveis de entrada. Isto é implementado usando um conjunto de multiplexadores configuráveis. Os blocos lógicos dos FPGAs da Concurrent Logic [12] e Toshiba [13] contêm portas AND e NAND de duas entradas.

A vantagem principal de uso de blocos lógicos de grão fino é que os blocos usados para implementar as funções projetadas são utilizados com muita freqüência de forma completa. A exemplo dos blocos utilizados em convencionais MPGAs (*Mask-Programmed Gate Arrays*) e células padrões (*standards cells*) [20], blocos lógicos pequenos podem ser usados mais eficazmente, pois as técnicas de síntese lógica e física, para tais blocos, são semelhantes. A desvantagem principal de blocos lógicos de grão fino é que eles requerem um número relativamente grande de segmentos de fios e chaves programáveis para o roteamento. Estes recursos de roteamento são dispendiosos em tempo e área do circuito integrado. Como resultado, FPGAs empregando blocos de grão fino são em geral mais lentos e tem mais baixas densidades em relação aos que empregam blocos de grão grosso.

2.1.2.2 Arquitetura de Grão Grosso

O bloco lógico do Act-1 da Actel está baseado na capacidade de multiplexadores para implementar diferentes funções lógicas. Ele funciona conectando cada de suas entradas a uma constante ou a um sinal. Este bloco consiste em três multiplexadores e uma porta OR. O Act-1, conforme é ilustrado na Figura 14(a), tem um total de 8 entradas (S_1, S_2, S_3, S_4, x, y, z e w) e uma saída (f), e implementa a função genérica $f = \overline{(s_3 + s_4)}\,(\overline{s_1 w} + s_1 x) + (s_3 + s_4)(\overline{s_2} y + s_2 z)$.

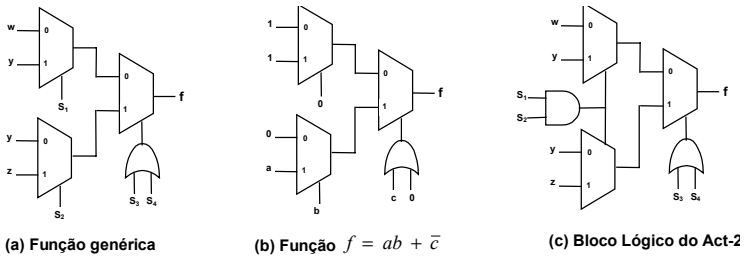


Figura 14 - (a) Função genérica, (b) Função $f = ab + \overline{c}$ e (c) Bloco lógico do Act-2

Conectando cada variável a um sinal de entrada, ou a uma constante, 702 funções lógicas diferentes podem ser implementadas. Por exemplo, a função lógica $f = ab + \overline{c}$ é implementada fixando-se as variáveis como mostrado na Figura 14(b): w = 1, x = 1, S_1 = 0, y = 0, z = a, S_2 = b, S_3 = c, e S_4 = 0. O bloco lógico do Act-2 [21] é semelhante ao do Act-1, a diferença é que a seleção dos dois multiplexadores que configuram as entradas do bloco são unidos e conectados à saída de uma porta AND de duas entradas, como mostrado em Figura 14(c). Esta mudança na estrutura do bloco lógico do Act-2 permite implementar 766 funções lógicas, ou seja, 64 a mais que no Act-1.

Semelhante à lógica da Actel, o bloco lógico no FPGA da QuickLogic [15] emprega multiplexadores 2x1. Cada entrada do multiplexador é alimentada por uma porta AND, como ilustra a Figura 15. Estes blocos baseados em multiplexadores têm a vantagem de permitir um alto grau de funcionalidade para um número relativamente pequeno de transistores. Porém, isto é alcançado às custas de um número grande de entradas (variáveis), 8 no caso da Actel e 14 no caso da QuickLogic, as quais, quando utilizadas impõem elevada demanda nos recursos de roteamento. Este tipo de bloco é mais adequado para FPGAs que usam chaves programáveis de pequeno tamanho, tais como antifusíveis.

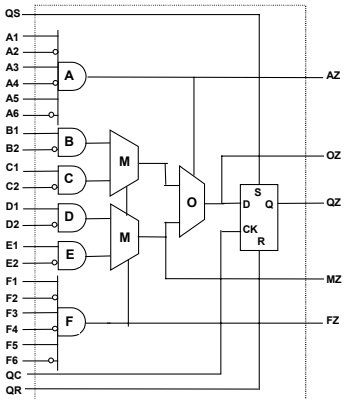


Figura 15 - Bloco lógico da QuickLogic

A base para o bloco lógico da Xilinx [9] são células baseadas em tecnologia de configuração SRAM. O bloco lógico Xilinx usa LUTs. Uma tabela verdade, para funções lógicas de K entradas, é armazenada em uma SRAM 2^K x 1 bits. As linhas de endereço da SRAM funcionam como entradas, e a saída da SRAM fornece o valor da função lógica. Por exemplo, considere a tabela verdade da função lógica $f = ab + \overline{c}$ representada na Figura 16(a). Esta função lógica é implementada utilizando uma LUT de 3 entradas conforme Figura 16(b), para tal a SRAM armazena "1" nos endereços "000," "010, "100," "110", "111" e "0" nos demais como especificado pela tabela verdade.

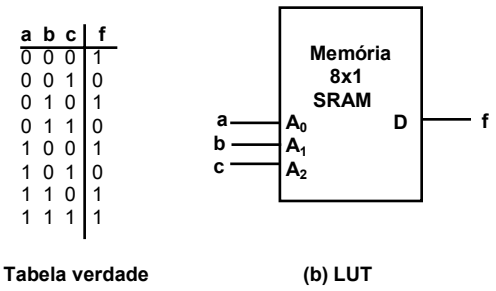


Figura 16 - (a) Tabela Verdade, (b) Bloco lógico do tipo LUT

A vantagem de LUTs é que estas exibem alta funcionalidade, uma tabela *look-up* de K entradas pode implementar qualquer função de K variáveis. A desvantagem é que elas tornam-se muito grandes para funções de mais de cinco entradas. Além disso são necessárias 2^K células de memória para uma LUT de K entradas. Mesmo que o número de funções que podem ser implementadas cresça rapidamente, muitas das

funções adicionais não são usadas freqüentemente em projetos práticos, além de serem de difícil implementação para uma ferramenta de síntese lógica. Desta forma, é freqüente o caso que uma LUT grande ser em grande parte subtilizada.

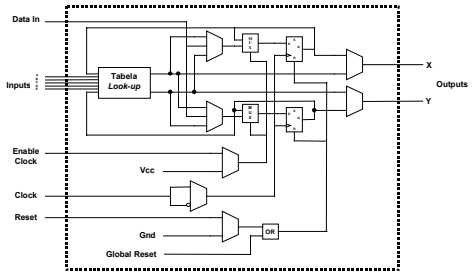


Figura 17 - Bloco Lógico Xilinx série 3000

O bloco lógico da família 3000 de FPGAs da Xilinx [9] contém uma LUT de 5 entradas e 1 saída, como ilustrado na Figura 17. Este bloco pode ser reconfigurado em duas LUTs de quatro entradas, possibilitando uma maior flexibilidade na utilização do bloco lógico, porque muitas funções lógicas mais comuns não requerem mais de cinco variáveis de entrada. O bloco também permite a implementação de lógica seqüencial e possui vários multiplexadores. Estes últimos permitem conectar as entradas combinacionais com as saídas diretamente ou passando por *flip-flops*. Os multiplexadores são controlados por células de SRAM carregadas durante a configuração do FPGA.

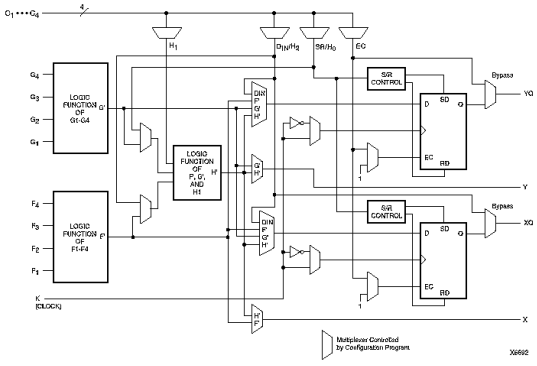
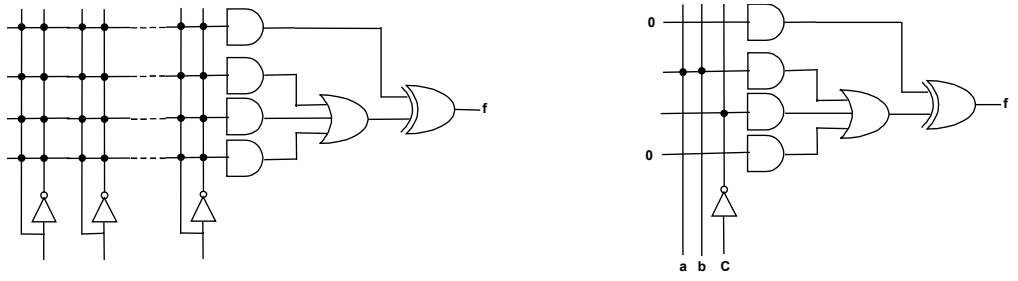


Figura 18 - Bloco lógico da série Xilinx 4000

Na família 4000 de FPGAs Xilinx o bloco lógico contém duas LUTs de 4 entradas e suas saídas conectadas a uma outra LUT de 3 entradas, como apresentado na Figura 18. Este bloco introduz duas mudanças significativas comparadas ao bloco lógico da série 3000. A primeira, dois tamanhos de tabelas *look-up* são utilizadas, permitindo um melhor compromisso entre desempenho e densidade lógica (Ver mais

detalhes sobre este tema na Seção 2.1.4). A segunda na família 4000 é o uso de duas conexões diretas, que ligam as saídas das duas tabelas *look-up* de quatro entradas para as entradas da tabela *look-up* de três entradas. Estas duas conexões são significativamente mais rápidas que qualquer interconexão programável, pois nenhuma chave programável é utilizada em série. Com o uso adequado destas conexões rápidas, pode melhorar muito o desempenho do FPGA. Porém há uma desvantagem para este tipo de conexão. Já que a conexão é permanente, a LUT de três entradas tem sua flexibilidade limitada e freqüentemente não pode ser utilizada, reduzindo a densidade lógica global. O bloco lógico do Xilinx 4000 incorpora várias características adicionais. Cada tabela *look-up* pode ser utilizada diretamente como um bloco de SRAM de pequenas dimensões. Isto permite implementar memórias pequenas mais eficazmente. Outra característica é a inclusão de circuitos, que possibilitam a configuração de somadores rápidos (*fast carry addition circuits*).

A arquitetura do FPGA Altera [19] evoluiu da arquitetura PLA baseada em PLDs de baixa densidade tradicionais [22]. Seu bloco lógico possui um grande numero de entradas (de 20 a mais de 100), este é constituído de portas AND que alimentam portas OR de três a oito entradas. A Figura 19(a) ilustra o bloco lógico do Altera MAX série 5000. As entradas das portas AND podem ser conectadas a qualquer linha vertical do barramento, por meio de chaves eletrônicas com tecnologia de configuração de porta flutuante vista na Seção 2.1.1.3. A Figura 19(b) ilustra a implementação da função lógica $f = ab + \overline{c}$. As conexões na figura indicam quais chaves eletrônicas estão fechadas para configurar o bloco lógico com a função desejada.



(a) Função Genérica (b) Função $f = ab + \overline{c}$ Configurada

Figura 19 - Bloco lógico Altera série 5000, (a) genérico e (b) configurado

A vantagem deste tipo de bloco é que o grande número de entradas pode ser usado para formar funções complexas com poucos blocos lógicos, reduzindo a

necessidade de muitas interconexões programáveis. Isto porém, resulta em perda de densidade, pois é difícil fazer-se uso eficiente de todas entradas e todas portas. Esta perda não é tão significativa, pois é compensada pela alta densidade das portas *Wired-AND*. As próprias conexões servem também para o roteamento, diferente do que ocorre em outras arquiteturas, onde a configuração da lógica e o roteamento estão separados. Uma desvantagem das portas *Wired-AND* é o uso de resistores *pull-up*, que apresentam alto consumo de energia na configuração.

O bloco lógico do Altera MAX 7000 [23] é semelhante ao MAX 5000. Ele tem dois termos produto a mais e também maior flexibilidade, porque blocos adjacentes podem utilizar termos produto um do outro. Vários outros FPGAs fazem o uso de grandes blocos lógicos, estilo AND-OR, entre os quais estão os produzidos pela Concurrent Logic [12], AMD [17] e Lattice [18].

2.1.3 Arquitetura de Roteamento

A arquitetura de roteamento de um FPGA é a maneira na qual são posicionadas as chaves programáveis e segmentos de fios para realizarem as interconexões programáveis dos blocos lógicos. Nesta Seção será apresentada uma arquitetura de roteamento utilizada comercialmente como referência para o estudo das arquiteturas de roteamento de forma genérica, observando-se que cada fabricante possui um modelo de arquitetura particular.

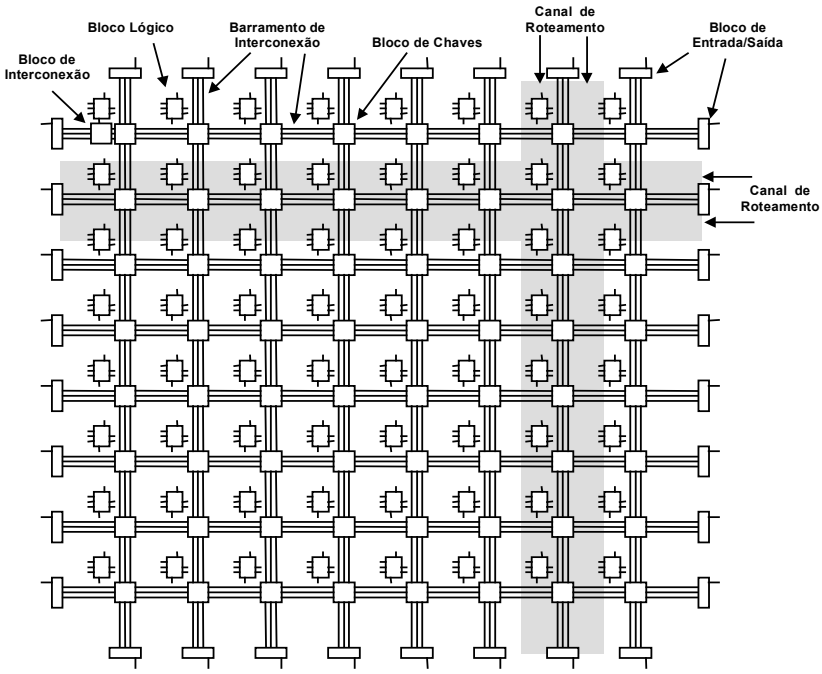


Figura 20 - Arquitetura de roteamento genérica

A Figura 20 define um modelo genérico para descreve as várias arquiteturas de roteamento de FPGAs comerciais. Neste modelo: um *segmento de fio* é contínuo através das chaves programáveis; uma ou mais chaves podem ser ligadas ao segmento de fio; cada final de um segmento de fio esta tipicamente ligado a uma chave programável; uma *trilha* é uma sucessão em linha de um ou mais segmentos fios e um *canal de roteamento* é um grupo de trilhas paralelas.

Conforme a Figura 20, o modelo contém duas estruturas básicas. O primeiro é o *bloco de conexão* que aparece em todas as arquiteturas. Um bloco de conexão provê a conectividade das entradas e saídas de um bloco lógico para e dos segmentos de fios no canal de roteamento. Embora não mostrado na figura, pode haver blocos de conexão na direção vertical como também na direção horizontal. A segunda estrutura é o *bloco de chaves* o qual provê a conectividade dos segmentos de fios horizontais com o os segmentos verticais. O bloco de chaves provê conectividade entre os segmentos incidentes em seus quatro lados. Em algumas arquiteturas, o bloco de chaves é intercalado com o bloco de conexão, e em outros eles são combinados em uma única estrutura. Um último bloco de relevante importância é o *Bloco de Entrada/Saída*, através deste é realizado a conectividade do circuito integrado que compõe o FPGA com o mundo externo ao circuito. Neste bloco os pinos terminais do componente FPGA

são programados para a transferência de sinais do interior do componente ou para o exterior do componente.

A Figura 21 ilustra a arquitetura de roteamento utilizada na família 3000 nos FPGAs da Xilinx [24][25]. As conexões do bloco lógico são feitas, no canal de roteamento, por um bloco de conexão.

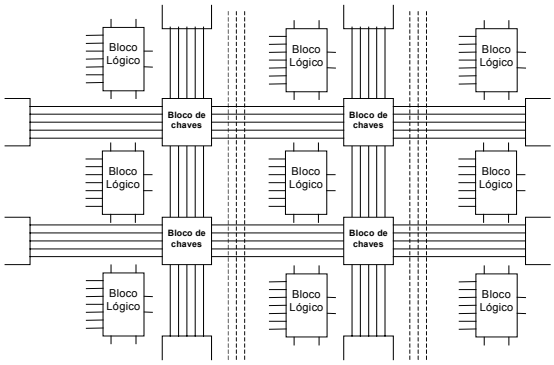


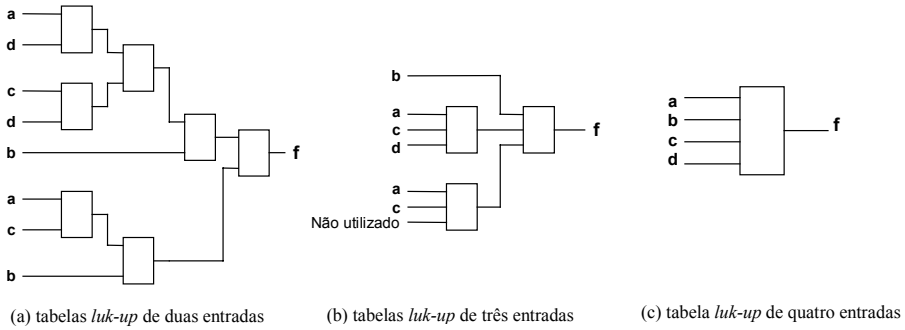
Figura 21 - Arquitetura de roteamento Xilinx 3000

2.1.4 Granularidade, Densidade e Desempenho do FPGA

Quando a granularidade do bloco lógico aumenta, o número de blocos necessários para implementar um projeto deveria diminuir. Por outro lado, um bloco lógico mais funcional (granularidade maior) exige mais interconexões programáveis para implementa-lo, por conseqüência, ocupa mais área. Este situação antagônica sugere a existência de um "ponto ótimo" para a granularidade de bloco lógico, no qual a área de FPGA dedicada a implementação da lógica é minimizada.

A porção de área para a implementação da lógica é relativamente fácil calcular, enquanto que o efeito de granularidade no roteamento não é tão simples, e tem grande influência na área de roteamento de todo o FPGA, esta influência é vista com maiores detalhes na Seção 2.1.3.

O efeito da funcionalidade do bloco sobre a área de lógica é facilmente observado, por exemplo, a implementação da função de lógica $f = abd + bc\overline{d} + \overline{a}\overline{b}\overline{c}$, utilizando blocos lógicos de três granularidades diferentes como ilustrado na Figura 22.



(a) tabelas *luk-up* de duas entradas (b) tabelas *luk-up* de três entradas (c) tabela *luk-up* de quatro entradas

Figura 22 - Implementação da função lógica $f = abd + bc\overline{d} + \overline{a}\,\overline{b}\,\overline{c}$

Cada um dos blocos lógicos são tabelas *look-up* de duas entradas, Figura 22(a), três entradas, Figura 22(b) e quatro entradas, Figura 22(c) respectivamente. A implementação da função com tabelas *look-up* de duas entradas requer sete blocos lógicos, as de três entradas requer três blocos, e a de quatro um único bloco. Considerando como uma medida de área, o número de *bits* de memória necessários para implementar a função lógica *f*, utilizando tabelas *look-up* de **K** entradas. Considerando que cada tabelas *look-up* de **K** entradas requer $\mathbf{2^K}$ *bits*, a implementação da função com tabelas de duas entradas requer um total de 28 *bits*, as de três requerem de 24 *bits* e as de quatro no máximo 16 *bits*. Conclui-se que utilizando esta medida de área como referência, as tabelas *look-up* de quatro entradas requerem menor área de lógica para a implementação da função.

Dados experimentais apontam para uma determinada granularidade de bloco lógico, granularidade esta, que requer a menor área de lógica para a implementação da função. M determinado número de projetos são mapeados em FPGA com diferentes granularidades do bloco, a área total do bloco lógico, bem como a área de roteamento são determinadas para cada mapeamento. São calculadas as médias dos resultados e então comparados. A Figura 23 dá um exemplo de tais resultados experimentais [26].

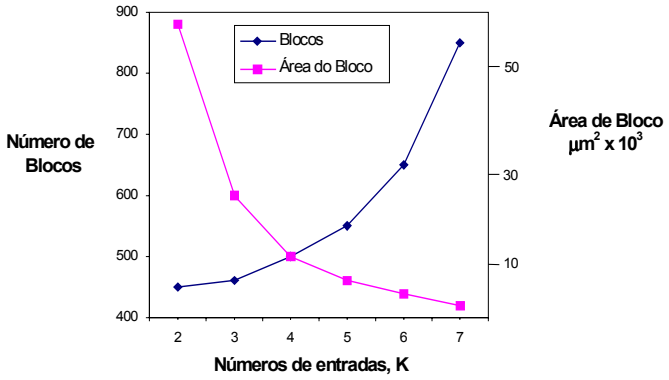


Figura 23 - Número de blocos e área do bloco para um circuito

O número de blocos lógicos diminui rapidamente como o aumento de **K** (número de entradas do bloco lógico), enquanto que o tamanho de bloco aumenta exponencialmente com **K**. A área total do bloco lógico (o produto das duas curvas) alcança um mínimo em **K=4**. A área mínima total do bloco lógico possui uma fraca dependência em relação ao tamanho da chave programável (Seção 2.1.1.1 [14] e [26]).

Área de lógica ativa é parte da área total. A área para o roteamento é normalmente maior que a área ativa, particularmente em FPGAs, representam de 70% a 90% da área total. A Figura 24 mostra a área de roteamento por bloco lógico e o número de blocos usados por número de entradas **K,** para um mesmo projeto experimentado.

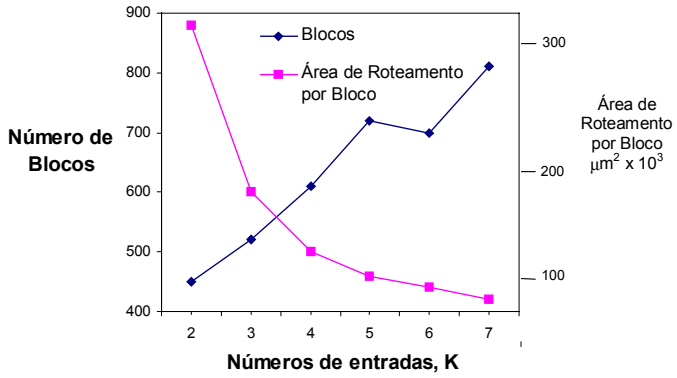


Figura 24 – Número de blocos e área de roteamento/bloco para um circuito

2.2 ARQUITETURAS RECONFIGURÁVEIS

2.2.1 Definições Básicas

Um *sistema computacional* de acordo com a Figura 25 é uma composição de um sistema digital e um software que executa sobre este, tal como definido por Calazans em [27]. Um sistema digital pode ser visto como uma estrutura com entradas e saídas capaz de processar as entradas e gerar as saídas.

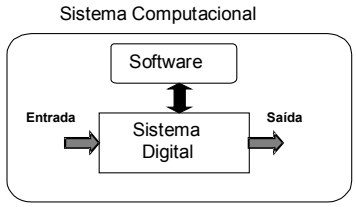


Figura 25 - Sistema Computacional

Em um sistema computacional, o desempenho está relacionado com as tarefas a serem executadas e os resultados nele obtidos. O desempenho pode ser definido por critérios baseados em um subconjunto de fatores, dos quais os mais importantes em geral são: o tempo de execução da tarefa; a energia consumida no tempo (potência consumida) e o preço do sistema como um todo.

Os sistemas computacionais tem seu valor medido por alguns objetivos que são específicos a seu campo de aplicação, por exemplo, desempenho, projeto, facilidade de programabilidade e custo de fabricação. Os sistemas têm diferentes aplicações, e cada aplicação especifica requer uma *arquitetura dedicada*, e esta por sua vez, deve ser projetada para atender a todos os objetivos a dada aplicação. Existe alguns problemas de projeto que freqüentemente são tarefas demoradas e propensas a erros. A grande quantidade de informações envolvida na solução dos problemas, fazem com que dificilmente o projetista consiga aperfeiçoar todos os objetivos, podendo conduzir assim a um produto cujo valor é mais baixo que o seu potencial.

Muitas vezes a melhoria na solução de um problema só é alcançada, construindo-se um sistema computacional especializado com o uso uma *arquitetura reconfigurável*. Uma arquitetura reconfigurável neste trabalho é uma composição de um ou mais *processadores de propósito geral* (do inglês, General Purpose Processor ou GPP), um ou mais *processadores de aplicação específica* (do inglês, Aplication

Specific Processor ou ASP) e uma estrutura de memória compartilhada ou não entre o GPPs e ASPs. A Figura 26 ilustra a topologia de tal arquitetura reconfigurável, onde um barramento padrão(endereços, dados e controle) interconecta GPPs, a estrutura de memória e os ASPs. Os ASPs são dispositivos de hardware, cuja funcionalidade pode ser alterada durante o uso do sistema [4].

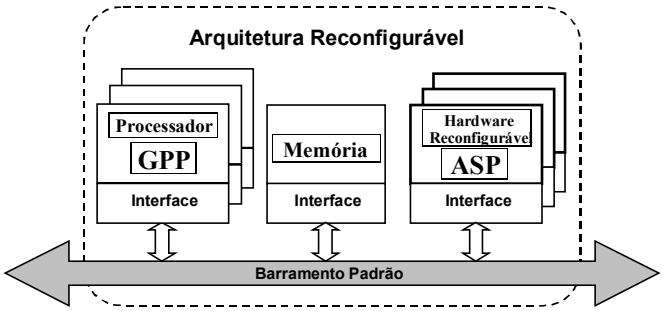


Figura 26 - Arquitetura Reconfigurável

Por outro lado, uma dada arquitetura é dita *fixa* se nenhum de seus componentes de hardware podem ter sua funcionalidade e/ou estrutura alterada durante o uso. Quando é necessário executar uma determinada tarefa computacional que tenha alto desempenho e utilize pouca quantidade de hardware, observamos que a composição de uma arquitetura reconfigurável no sistema computacional, pode vir a superar algumas limitações, por exemplo, o tempo de execução, impostas pela execução da tarefa em uma arquitetura fixa.

Existe um grande número de sistemas computacionais, que podem ser classificados como Arquiteturas Reconfiguráveis. Nesta Seção, a título de classificação, limita-se o escopo ao estudo de uma classe específica de arquiteturas reconfiguráveis de grande utilidade prática: aquelas compostas de apenas um GPP, um ASP e um subsistema de memória.

2.2.2 Classificações das Arquiteturas Reconfiguráveis

Componentes de hardware reconfigurável usados em combinação com GPPs impõe uma nova metodologia para implementar aplicações. A combinação de um processador e um hardware reconfigurável, define uma *arquitetura reconfigurável*, que aponta a um futuro no qual a dinâmica dos sistemas computacionais mudam

radicalmente. Esta mudança na dinâmica dos sistemas computacionais denota o quão bem e flexivelmente eles poderão solucionar problemas cada vez complexos.

Existem diversas classes de sistemas computacionais que podem ser caracterizados como arquiteturas reconfiguráveis, tais como aos descritos em [28],[6], [29] e [31]. Estas arquiteturas podem ser classificadas segundo um conjunto de critérios. Nesta Seção, revisa-se duas das classificações propostas na bibliografia consultada e se propõe alguns novos critérios e classificações de arquiteturas reconfiguráveis baseadas nestes.

2.2.2.1 Classificação de PAGE

Conforme Page [28], os FPGAs estão hoje presentes na maioria das implementações de hardware complexas. A função destes pode ser mudada sob controle do software, oferecendo a possibilidade de arquiteturas que se reconfiguram para apoiar uma aplicação. Estes componentes de hardware reconfiguráveis são usados em combinação com tradicionais processadores (GPPs). Combinações de GPPs e algum hardware de reconfigurável implementam diferentes tipos de aplicações que podem ser classificados por diferentes critérios. Page cita quatro critérios de classificação para arquiteturas reconfiguráveis, os quais serão apresentados a seguir:

Critério: Forma de interação entre o GPP e o ASP

Este critério diz respeito ao modo como a parte reconfigurável, em geral um ASP, se comunica com o processador principal do sistema, que na maioria das aplicações é um GPP. O modelo de interação entre os processadores ASP e GPP possui quatro modos de comunicação:

- **coprocessamento:** A parte reconfigurável recebe as instruções provenientes do processador principal do sistema, de forma similar às instruções enviadas a um coprocessador.

- **chamada de procedimento remoto (RPC, *remote procedure call*)**: O microprocessador emite uma instrução ou sucessão de instruções que são

interpretadas pela parte reconfigurável como uma chamada de procedimento remoto, semelhante ao coprocessamento, podendo operar como um sistema multitarefa.

- **cliente-servidor:** Este modo tem o algoritmo da parte reconfigurável como um processo de servidor, sendo que, num dado momento, este pode solicitar alguma informação sobre os processos no microprocessador, que nesse caso passa a executar como servidor para o GPP.

- **execução paralela:** Os processos executados na parte reconfigurável e o microprocessador principal executam independentes um do outro. O algoritmo da parte reconfigurável ocorre como um processo paralelo. A comunicação entre o dois processadores pode acontecer, a qualquer momento, via troca de mensagens.

Critério: Estrutura de Memória

O algoritmo executado por um ASP, normalmente, requer algum espaço variável de memória temporária para sua operação. O tamanho desse espaço pode variar desde alguns registradores até uma estrutura organizada de memória, incluindo o uso de caches. Page classificou Arquiteturas Reconfiguráveis quanto à estrutura de memória em:

- **sem acesso a memória:** em algumas circunstâncias, o algoritmo executado no ASP pode operar sem memória externa. Esta situação só é aceitável quando o ASP precisa de poucos estados para sua operação, pois a maioria dos ASPs atuais têm relativamente pouca memória interna disponível.

- **acesso compartilhado:** o ASP pode usar qualquer memória associada ao barramento por ele compartilhado. Contudo, isto envolve tempo e sobrecarga no espaço de memória, necessitando desta forma de uma estrutura de arbitragem de acesso à memória. Pode ser necessário existir um controlador de endereçamento, o que tende a reduzir a velocidade de acesso a esta memória para processamento intensivo de dados.

- **acesso a memória local:** o algoritmo da parte reconfigurável é organizado com memória privada, consumindo o mínimo de ciclos de "clock" para cada dado necessário, aumentando o desempenho durante um processamento. Neste caso, contudo, pode existir duplicação de dados em relação ao GPP. Logo, também é necessário um controle de coerência e um meio para troca de mensagens.

Critério: Forma de Operação da Memória local do ASP

Segundo o presente critério, a memória local do ASP pode ser usada de três formas diferentes:

- **memória de bloco:** a memória local é grande bastante para conter um conjunto de dados completo para o processamento. Por exemplo, esta poderia ser uma página completa de vídeo a ser processada por um algoritmo de compressão de imagens.

- **memória de fila (FIFO):** a memória age como fila, enquanto dados e resultados são trocados com o GPP. Um exemplo seria o armazenamento da região completa de apoio para um filtro de tempo real (Resposta de Impulso Finita).

- **cache**: a memória local pode ser acessada como um "cache" sobre uma estrutura de dados maior controlada pelo GPP. Pode ser administrada através de troca preditiva de dados pedidos no microprocessador, ou através de métodos tradicionais, tais como: "cache-line-on-demand methods" [32].

Page salienta que as classificações baseadas nos critérios desta e da Seção anterior não são necessariamente mutuamente exclusivos, ou seja os critérios não são ortogonais.

Critério: Forma de Execução do Algoritmo pelo ASP

Page menciona a existência de cinco formas segunda as quais um algoritmo pode ser implementado numa arquitetura reconfigurável. Estas formas diferem no suporte à execução de algoritmos, explorando diferentes partes do aspecto custo/desempenho de implementação:

- **hardware puro** (Figura 27): O algoritmo é convertido através de uma ferramenta de compilação para hardware, em uma descrição de hardware única e completa, de um circuito que realiza integralmente a mesma função daquele algoritmo e que será carregada na parte reconfigurável. Esta é a tecnologia na qual tudo é construído sem a necessidade de qualquer outro elemento extra para microprogramação ou código executável.

Figura 27 - Hardware puro

- **processador de aplicação específica:** o algoritmo é compilado em um código de máquina abstrato para um processador abstrato, e os dois são então co-otimizados para produzir a descrição de um ASP propriamente dito e um programa em código de máquina para execução no ASP (Figura 28). A descrição do processador é compilada então para uma implementação na parte reconfigurável (*hardware/software codesign*).

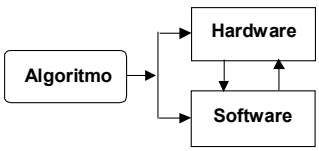


Figura 28 - Processador de aplicação específica

- **reuso seqüêncial:** o algoritmo pode ser muito grande para implementar no dispositivo reconfigurável disponível, ou, por razões econômicas, divide-se o algoritmo em partes, de forma a usar uma configuração parcial, ora utilizando uma ou outra das configurações conforme necessário (Figura 29). As vantagens advindas

da reutilização da uma mesma parte reconfigurável deve ser comparada com o tempo gasto para a reconfiguração.

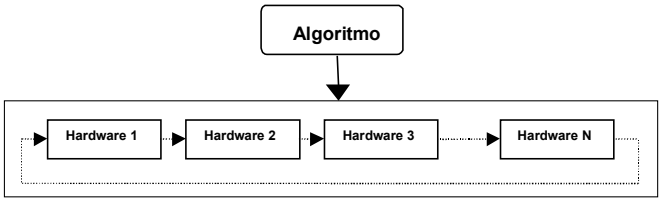


Figura 29 - Reuso Seqüêncial

- **uso múltiplo simultâneo** (Figura 30): se os recursos da parte reconfigurável são consideravelmente extensos, é possível coexistirem vários algoritmos residentes executando suas funções simultaneamente, cada um deles trocando dados entre si ou interagindo com o processador hospedeiro (GPP).

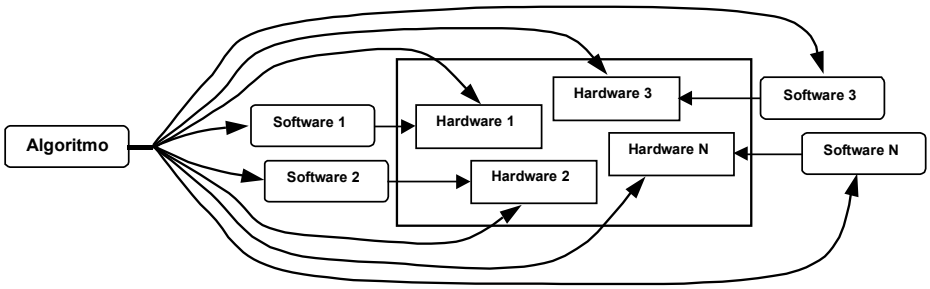


Figura 30 - Uso múltiplo simultâneo

- **uso sob demanda:** aqui há uma coleção relativamente grande de circuitos que podem ser carregados na parte reconfigurável e a atual atividade do sistema depende, a qualquer momento, de um conjunto destes circuitos (Figura 31). Analogamente a sistemas de memória virtual, podemos nos referir a esta arquitetura como a um "hardware virtual". O uso sob demanda cria a necessidade de estruturas complexas, onde o "tempo real" exige um eficiente controle de qual sistema de hardware deve ser construído para a demanda do momento.

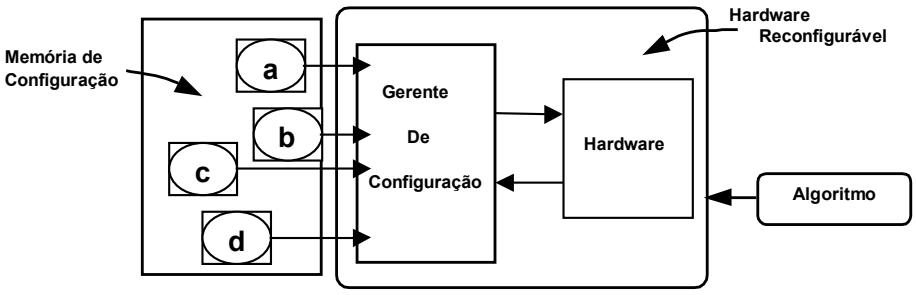


Figura 31 - Uso sob demanda

2.2.2.2 Considerações sobre a taxonomia de Page

Existem diferentes pontos de vistas para classificar arquiteturas reconfiguráveis. Page classifica arquiteturas reconfiguráveis a partir de características comportamentais e estruturais de uma série de estudos de casos. Seus exemplos compõem-se de uma parte reconfigurável e uma parte baseada em um processador de propósito geral (GPP). Sua classificação dá um enfoque privilegiando a processadores de aplicação específica (ASP) [28]. Além disto, os critérios "Forma de execução do algoritmo pelo ASP" e Forma de Operação da Memória local pelo ASP" não são ortogonais.

2.2.2.3 Classificação de Sanchez

Analisando arquiteturas reconfiguráveis de um outro ponto de vista, Sanchez [6] classifica-as levando em consideração o número de configurações possíveis para o circuito e o instante de mudança de configuração. Ele divide os arquiteturas reconfiguráveis em duas grandes classes de reconfigurabilidade, discutidas nas Seções a seguir.

Sistemas estáticos

Nesta classe de arquiteturas, existe pouca ou nenhuma flexibilidade. O sistema é desenvolvido para executar somente uma aplicação específica. Duas subclasses podem ser identificadas:

- **configuráveis estaticamente:** o circuito possui apenas uma programação, que nunca é alterada, nem durante e nem após o processamento. Ou seja, o componente é programado, integralmente, para realizar apenas uma função que não é mais alterada durante toda a atividade do sistema. Estritamente falando, esta subclasse não constitui uma arquitetura reconfigurável.

- **reconfiguráveis estaticamente:** há várias configurações possíveis para o circuito, entretanto a reconfiguração da arquitetura ocorre apenas ao final do processamento de uma dada tarefa. Os dispositivos programáveis podem ser reconfigurados integral (a forma mais comum) ou parcialmente.

Sistemas Dinâmicos

É nesta classe de aplicações que a arquitetura reconfigurável é utilizada da maneira mais flexível. Em um sistema **reconfigurável dinamicamente** existem várias configurações para o circuito, e a reprogramação ocorre durante a execução (*run-time reconfiguration*). Para a implementação, podem ser utilizados tanto dispositivos programáveis parcialmente reconfiguráveis, quanto dispositivos que alocados em instantes de tempo diferentes (enquanto um executa, o outro esta sendo reconfigurado e vice-versa).

2.2.2.4 Considerações sobre a taxonomia de Sanchez

Na análise dos sistemas vistos por Sanchez, em [6] ele descreve quatro projetos de arquitetura reconfigurável, separando-os nas duas classes distintas descritas acima. O autor considera dois sistemas estáticos: SPYDER (sistema de desenvolvimento para um processador reconfigurável) e RENCO (uma rede de computadores reconfigurável) e outros dois como sistemas dinâmicos: Firefly (uma máquina evolutiva) e o BioWatch (um relógio capaz de consertar a si próprio). Estes sistemas enquadram-se nas características discutidas anteriormente (sistemas estáticos e dinâmicos). A classificação de Sanchez é estritamente comportamental, observa os sistemas quanto a dinamicidade

da sua reconfiguração. Comparada à classificação de Page, a última não distingue as arquiteturas reconfiguráveis sob o ponto de vista estrutural.

2.2.2.5 Classificação de Torok

Tanto as classificações de Page [28] quanto as de Sanchez [6] baseiam-se em diferentes critérios, tais como: esquema de comunicação entre GPPs e ASPs, arquiteturas de memória, modos de execução e classes de reconfigurabilidade dos projetos. Os critérios utilizados nem sempre são ortogonais, tendo apenas por distinção o ponto de vista do critério de classificação de cada modelo.

Torok [33] propõe um conjunto adicional de critérios, observando-se a reconfigurabilidade de outro ponto de vista, que pode ser utilizada como forma alternativa ou complementar às classificações apresentadas anteriormente.

Novas classificações são definidas por estes critérios e analisadas do ponto de vista do conjunto de *bits* (aqui denominado vetor de configuração) e do modo com que estes implementam a funcionalidade dos componentes do Hardware (ou seja a configuração do hardware).

Critério: Número de vetores de configuração

Este critério define o número de conjuntos de bits (vetores) que são carregados na arquitetura para a configuração do hardware, possui duas classificações:

- **Simples**: definida por um vetor de configuração específico, o qual é carregado uma única vez na arquitetura configurando-a para uma única aplicação;

- **Múltipla**: caracterizada por mais de um vetor de configuração, a cada nova aplicação corresponde uma nova configuração, o que implica em uma arquitetura reconfigurável.

Critério - Dimensão do vetor mínimo de configuração

Neste critério é observada a quantidade de bits que compõem o vetor de configuração e comparada em relação ao tamanho total, também possui duas classificações:

- **Total**, vista em relação ao vetor de configuração, que tem a dimensão única e máxima, carregada inteiramente na arquitetura, configurando-a totalmente;

- **Parcial** : o vetor de configuração é carregado em partes, configurando a arquitetura parcialmente, tendo sua dimensão dividida em dois tipos básicos (grão grosso ou grão fino), relativo a parte da arquitetura que é configurada.

Critério: Ordem de Configuração

Refere-se a ordem na qual um conjunto de vetores de configuração é carregado na arquitetura, e esta, realiza as tarefas na mesma ordem como os vetores são carregados. O critério é dividido em três classificações:

- **Serial:** quando um conjunto de vetores configuram a arquitetura para diversas aplicações, as quais são utilizadas, uma a uma, de forma seqüencial em uma mesma arquitetura;

- **Circular**: semelhante à classificação Serial o conjunto de vetores usado para configurar a arquitetura é limitado, com a particularidade que as tarefas especializadas são executadas de forma seqüencial e repetitiva;

- **Aleatória**: que não apresenta uma ordem predefinida para a carga de diferentes vetores de configuração.

Critério: Atividade da configuração

Este último baseia-se no tipo de atividade que os vetores impõem na configuração da arquitetura, permitindo ou não sua alteração durante o uso. Para este critério temos duas possíveis classificações:

- **Passiva**: o vetor configurado na arquitetura permanece inalterado até uma nova configuração de uma nova aplicação;

- **Ativa**: o vetor configurado na arquitetura é reconfigurado pela própria aplicação durante o uso.

Os critérios propostos por Torok estão resumidos na Tabela 1 e apresentam cada um suas respectivas classificações para as Arquiteturas Reconfiguráveis.

CRITÉRIO	CLASSIFICAÇÕES
Número de vetores de configuração.	Simples. Múltipla.
Dimensão do vetor mínimo de configuração.	Total. Parcial (grão grosso ou grão fino).
Ordem de configuração.	Serial. Circular. Aleatória.
Atividade da configuração.	Passiva. Ativa.

Tabela 1 - Classificação de Arquiteturas Reconfiguráveis

2.3 DENSIDADE FUNCIONAL

As melhorias em eficiência que as técnicas de reconfiguração trazem não são alcançadas sem custo: um tempo adicional para a reprogramação do dispositivo e mecanismos de acesso à memória são necessários para transferir os bits de configuração da memória externa para a memória do FPGA. Em alguns casos, esse tempo extra obviamente compromete as vantagens dessa especialização. Em alguns outros casos, no entanto, a diferença entre o aumento da eficiência e os recursos adicionais necessários não é claramente perceptível.

Embora muitos sistemas que empregam a técnica de auto-reconfiguração demonstrem ou sugiram aumento de eficiência, poucos consideram ou justificam o custo adicional da reconfiguração. Na prática, essa técnica só será efetivamente utilizada se as vantagens em relação às técnicas convencionais puderem ser claramente verificadas.

Wirthlin e Hutchings [30] propõem o uso de uma métrica que relaciona tempo e área de silício ocupada, denominada Densidade Funcional. Os autores empregam Densidade Funcional como ferramenta para avaliar quantitativamente as vantagens (ou desvantagens) da RTR em termos de redução de área versus o custo adicional do tempo de reconfiguração.

2.3.1 Definição

A vantagem principal da reconfiguração em tempo real é possibilitar especialização da arquitetura de um circuito para resolver um problema específico. Como sugerido anteriormente, circuitos especializados dessa maneira necessitam menos hardware e geralmente operam em velocidade superior as alternativas convencionais.

Utilizando a densidade funcional como métrica, podemos avaliar quantitativamente os benefícios desta técnica de especialização.

A densidade funcional (*D*) de uma implementação de hardware é definida em termos do custo da implementação computacional em hardware. Para circuitos VLSI (do inglês, Very Large Scale of Integration), o custo deste processamento é geralmente definido como o produto entre a área do circuito (*A*) e o tempo de execução (*T*), ou $C = AT$ [34] [35]. Este custo é aplicado aqueles sistemas onde a capacidade de processamento é mais importante que a latência [36].

A densidade funcional (*D*) mede a capacidade de processamento (operações por segundo) dos recursos de hardware de é definido como o inverso do custo computacional:

$$D = \frac{1}{C} = \frac{1}{AT} \textbf{ (1)}$$

T é o tempo operacional total do processamento e inclui o tempo requerido para execução, controle, inicialização e transferência de dados. *A* é a área total necessária para implementar o elemento computacional em hardware.

Esta métrica (densidade funcional), pode ser usada para comparar circuitos estáticos com os circuitos que utilizam reconfiguração dinâmica. Especificamente, a densidade funcional pode identificar as condições em que uma arquitetura reconfigurável tem uma melhor relação *custo* x *benefício* (em termos de área e tempo de execução) do que o seu correspondente estático.

A melhoria (*I*) na densidade funcional do circuito reconfigurável (*Dr*) em relação ao circuito estaticamente configurado (*Ds*) é calculado como a diferença normalizada entre *Dr* e *Ds*:

$$I = \frac{\Delta D}{Ds} = \frac{Dr - Ds}{Ds} = \frac{Dr}{Ds} - 1 \textbf{ (2)}$$

O percentual de melhoria é obtido multiplicando-se o resultado **(2)** por 100.

2.3.1 Tempo de Configuração

A maior diferença entre as arquiteturas reconfiguráveis e seus equivalentes estáticos é o custo adicional de reconfiguração. A maioria dos FPGAs disponíveis no mercado exigem que a configuração do circuito e a execução ocorram separadamente. Esta característica força a adição deste tempo de configuração ao tempo total de operação de um circuito reconfigurável. Assim, para estes circuitos, o tempo operacional total inclui tanto o tempo de execução (*Te*) como o tempo de configuração (*Tc*), ou $T = Te + Tc$. Substituindo T na fórmula **(1)**, temos a redefinição da densidade funcional com a inclusão do custo adicional de reconfiguração.

$$Dr = \frac{1}{A(Te + Tc)} \quad \textbf{(3)}$$

A fórmula **(3)** demonstra claramente que o tempo de configuração reduz a densidade funcional. Durante a reconfiguração do circuito, os recursos de hardware que estão sendo reconfigurados estão essencialmente inativos e não contribuem para o processamento. A medida em que o tempo de reconfiguração aumenta, a densidade funcional do sistema diminui. Embora todos os sistemas reconfiguráveis incluam alguma sobrecarga devida à configuração, aqueles com tempos menores terão maior densidade funcional.

Embora o tempo de configuração absoluto de um sistema seja um parâmetro importante numa arquitetura reconfigurável, o tempo *relativo* de configuração é bem mais informativo. A relação de configuração, $f = Tc / Te$, define este importante parâmetro. O tempo total de operação de uma arquitetura reconfigurável pode ser expresso em termos desta relação como $T = Te\ (1 + f)$. Substituindo-se este tempo em **(3)**, temos a densidade funcional em termos de *f*.

$$Dr = \frac{1}{ATe(1 + f)} \quad \textbf{(4)}$$

Como sugerido em **(4)**, tempos longos de configuração podem ser tolerados se seguidos por longos tempos de execução (com um valor de *f* pequeno, por exemplo).

Sistemas que operam em grandes volumes de dados ou que apresentam uma granularidade de reconfiguração esparsa (poucas reconfigurações entre etapas grandes de processamento), têm demonstrado uma certa tolerância em relação aos tempos relativamente grandes necessários à reconfiguração dos dispositivos atuais.

Se considerarmos $f \rightarrow 0$, o overhead associado a reconfiguração é desprezível. Este sistema atingiria a máxima densidade funcional possível para um sistema reconfigurável. Este valor máximo, (*Dmax*) é calculado desconsiderando-se os efeitos do tempo de reconfiguração (por exemplo, $Dmax_{\lim f \rightarrow 0} = Dr = 1/ATe$). Usando-se este valor, pode-se chegar ao máximo de melhoria (*Imax*) sobre um sistema estático. Este parâmetro é importante porque sugere os benefícios de uma arquitetura reconfigurável e também porque através dele pode-se obter uma boa indicação da aplicabilidade da técnica.

$$Imax = \frac{Dmax}{Ds - 1} \quad \textbf{(5)}$$

2.3.2 Impacto do tempo de configuração *versus* resultado final

Com os tempos relativamente longos de configuração dos dispositivos atuais, muitas das arquiteturas reconfiguráveis são muitos sensíveis em relação a *f* (relação de configuração). Uma maneira de reduzir este impacto é através do aumento do tempo de execução entre duas reconfigurações. Isto é alcançado mais freqüentemente através da execução de mais etapas ou tratamento de um volume maior de dados entre dois passos de configuração. Executando múltiplos processamentos entre cada necessidade de reconfiguração possibilita a amortização do custo de reconfiguração.

O tempo para executar *n* processamentos entre cada passo de configuração é dado por $Tn = nTse + Tc$, onde *Tse* é o tempo necessário para executar um único processamento. O tempo necessário para uma única execução é simplesmente Tn / n**,** ou:

$$To = Tse + \frac{Tc}{n} = Tc\,(1 + \frac{fe}{n}) \quad \textbf{(6)}$$

Como a equação sugere, o tempo de configuração é amortizado à medida em que aumenta o número de execuções entre cada necessidade de reconfiguração. A relação de configuração também é amortizada, reduzida por *n*. Isto reduz o overhead de configuração associado com a densidade funcional:

$$Dr_n = \frac{1}{Ate + (1 + fe / n)} \quad \textbf{(7)}$$

2.3.4 Análise da arquitetura pela densidade funcional

Utilizando-se a densidade funcional como medida é possível estabelecer uma comparação entre uma arquitetura reconfigurável e seu correspondente estático. Mais importante do que isso, é fundamental compreender as condições em que a densidade funcional de uma arquitetura reconfigurável excede a do circuito estático, ou seja, quando $Dr \geq Ds$. Usando Ar e Tr para representar a área e o tempo de processamento do circuito reconfigurável, esta relação pode ser reduzida a:

$$\frac{1}{Ar\,(Tr + Te)} \geq Ds$$

$$\frac{1}{Ds}\left(\frac{1}{ArTr}\right) \geq 1 + \frac{Tc}{Tr} \quad \textbf{(8)}$$

Simplificando, com a utilização de Dmax:

$$\frac{D\max}{Ds - 1} > \frac{Te}{Tr} \quad \textbf{(9)}$$

O lado esquerdo desta equação é o maior ganho possível com a utilização da reconfiguração. Substituindo *Imax*, produz a relação:

$$Imax \geq f \quad \textbf{(10)}$$

A equação **(10)** é um importante resultado que descreve a máxima relação de configuração de uma arquitetura reconfigurável. Esta relação define a condição para que uma arquitetura reconfigurável tenha mais densidade funcional do que o correspondente estático. Esta condição diz que a relação de configuração precisa ser menor que o máximo ganho potencial (*Imax*) do sistema reconfigurável.

Como exemplo, vamos considerar um circuito estático que requer uma área *a* e um tempo *t* para completar uma dada tarefa. Suponha que uma arquitetura reconfigurável complete a mesma tarefa utilizando metade da área (*a*/2) e em dois terços do tempo (2*t*/3). Como demonstrado na Tabela 2, o máximo ganho do circuito configurável é 2 (200 %). Usando **(10)**, este resultado sugere que a densidade funcional será maior do que a do circuito estático sempre que $f < 2$, ou seja, quando o tempo de configuração é menor do que duas vezes o tempo de execução.

	A	T	D*max*	I*max*
Estático	*a*	*t*	1/*at*	0
RTR	*a*/2	2*t*/3	3/*at*	2

Tabela 2 - Circuito Exemplo - Parâmetros

Intuitivamente, este resultado sugere que quanto maior a densidade funcional de uma arquitetura reconfigurável, menor é o impacto do tempo de configuração sobre o seu desempenho.

Este resultado também sugere que sistemas que tragam apenas ganhos modestos em relação aos circuitos convencionais podem ser justificáveis, desde que a relação de configuração seja baixa. Sistemas que executam tarefas por um longo período de tempo (em relação ao tempo de configuração), justificam a utilização de arquiteturas reconfiguráveis na maioria dos casos.

3 EXEMPLOS DE ARQUITETURAS RECONFIGURÁVEIS

Recentemente, a partir do final da década de 80, surgiram as primeiras Arquiteturas Reconfiguráveis. Estas arquiteturas foram desenvolvidas para aplicações específicas, normalmente uma classe de problemas bem definida, tais como o processamento de imagens [29], aritmética especializada, criptografia e processamento de informações genéticas [38]. Algumas destas arquiteturas permitiram resolver alguns problemas com maior desempenho que arquiteturas genéricas, aquelas cujas características construtivas foram desenvolvidas para solução de uma grande variedade de problemas.

Nas Seções a seguir serão analisados quatro exemplos de arquiteturas reconfiguráveis, propostas em trabalhos acadêmicos, para ilustrar as possíveis aplicações desta nova tecnologia. Nestes exemplos serão observadas as classificações segundo Page e Sanchez, a variedade de aplicações, bem como as diferentes arquiteturas reconfiguráveis adaptam a solução de problemas específicos. A Tabela 1 será referenciada na análise e classificação das diferentes arquiteturas reconfiguráveis para identificar o critério aplicado na classificação proposta para as diferentes arquiteturas reconfiguráveis.

3.1 PRISM (Processor Reconfiguration through Instruction-Set Metamorphosis)

O PRISM [37] é um processador para o qual existe um conjunto de ferramentas de desenvolvimento que, dada uma aplicação, instruções são sintetizadas em software e/ou hardware para o processador, sendo que esse processo é repetido para cada nova aplicação. Dois protótipos, o PRISM-I e o PRISM-II foram construídos, utilizando FPGAs Xilinx [9], baseadas nos dispositivos XC3090 e XC4010, respectivamente.

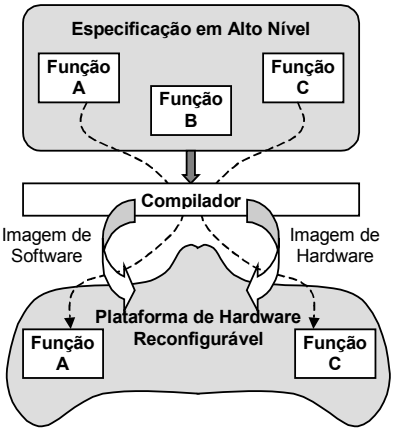


Figura 32 - Arquitetura Reconfigurável do Prism

A Figura 32 ilustra a estrutura geral do PRISM. Esta arquitetura consiste de um compilador interativo especializado de configurações, que recebe uma especificação em linguagem de alto nível. O compilador analisa especificação e apresenta ao programador uma lista funções candidatas a serem sintetizadas, que indica quais aceitar para produzir uma imagem de hardware. As linhas pontilhadas representam a porção das especificações (Funções A, B, C, ...) identificadas para a transformação em estrutura equivalente em hardware. A imagem de hardware consiste em especificações físicas programáveis em uma plataforma de hardware reconfigurável. Semelhante em função, a imagem de software é transformada para uma imagem executável, produzida por um compilador convencional, que consiste em código de máquina, pronto para execução junto com o código que integra a nova função sintetizada.

Nesta arquitetura, a cada nova aplicação corresponde uma reconfiguração, isto implica em uma arquitetura reconfigurável classificada como Múltipla. O processo de particionamento do sistema em hardware e software no PRISM ocorre a partir de uma especificação de alto nível, escrita em linguagem C (funções candidatas a serem sintetizadas). O compilador constrói duas descrições, uma que implementa a imagem de hardware e outra a imagem de software, conjuntos de procedimentos ou funções dentro do programa de aplicação. Estes respectivamente são usados para configurar o hardware e gerar o código executável para a plataforma de hardware reconfigurável. O tempo de compilação e configuração da arquitetura, é da ordem de 1 a dezenas de minutos, dependendo da função a ser implementada. A classificação de configuração Total é aplicável, pois o vetor de configuração, apesar de duplo, reconfigura a plataforma como um todo. Cada nova função é implementada em hardware através de intervenção do

usuário, constituindo uma arquitetura com ordem de configuração Aleatória. Quanto à atividade da configuração, uma vez que a mesma é configurada, permanece inalterada, qualificando a arquitetura como Passiva.

3.2 DISC (Dynamic Instruction-Set Computer)

O DISC [29] é um processador programável reconfigurável, capaz de carregar novas instruções no hardware conforme a necessidade específica de uma aplicação, por demanda, e tem sua estrutura geral representada na Figura 33. O projeto utiliza FPGAs CLAy da National, que são não apenas reconfiguráveis dinamicamente (baseados em SRAM) mas também reconfiguráveis parcialmente, o que viabiliza a abordagem de manter o hardware em funcionamento durante sua alteração. Um exemplo de aplicação DISC na implementação de um algoritmo de refinamento de imagens, alcançou melhorias de até 10 vezes em relação à mesma implementação em software num PC486 a 66 MHz.

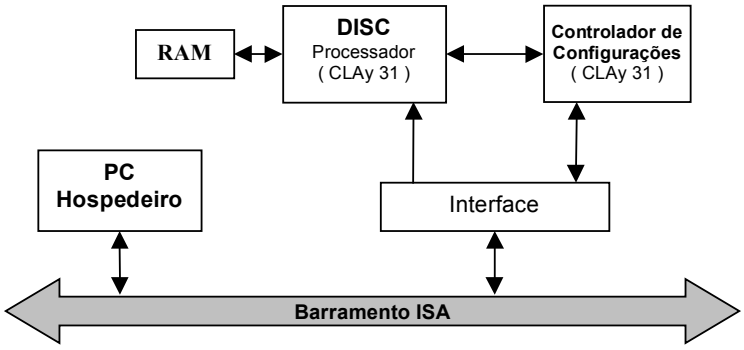


Figura 33 - Estrutura geral do sistema DISC

Observa-se, a partir da Figura 33, que o DISC possui um controlador de configurações. Este controlador dinamicamente altera a configuração do Processador DISC carregando o hardware para executar novas instruções e/ou substituindo hardware de instruções não usadas. Isto implica uma arquitetura reconfigurável Múltipla. Os recursos de hardware específicos para cada instrução são implementados como uma configuração Parcial. Isto é feito configurando cada instrução individualmente no DISC.

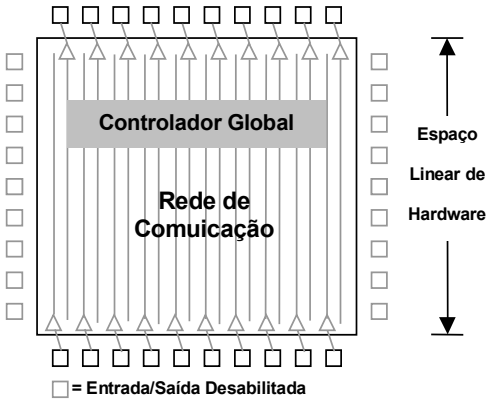


Figura 34 - Descrição física da distribuição interna de recursos de hardware do Processador DISC

O hardware é reconfigurado usando conjuntos de linhas contíguas de blocos lógicos dentro do FPGA que implementa o processador. A unidade de reconfiguração mínima é uma linha de blocos lógicos do FPGA. A estrutura interna que implementa o DISC é ilustrada na Figura 34. A reconfiguração só ocorre para atender demanda do programa de aplicação, caracterizando, desta forma, uma arquitetura com as classificações Aleatória e Ativa. As linhas verticais na Figura 34 representam barramentos de comunicação entre recursos de hardware específicos para cada instrução e o Controlador Global configurado de forma permanente.

3.3 SPLASH

SPLASH [38] é um array sistólico reconfigurável desenvolvido pelo *Supercomputing Research Center* (SRC) em 1988. Sua principal área de aplicação foi em Biologia Computacional, oferecendo um bom desempenho em tarefas como a comparação e emparelhamento de seqüências de DNA. A matriz do SPLASH possui 32 estágios conforme detalhado na Figura 35 - Interface VME, VSB e matriz de 32 estágios, cada um é composto de: um FPGA Xilinx XC3090 e uma SRAM de 128KB. A segunda versão do SPLASH, o SPLASH-2, é um processador matricial reconfigurável mais flexível que SPLASH e que foi desenvolvido em 1992 [39].

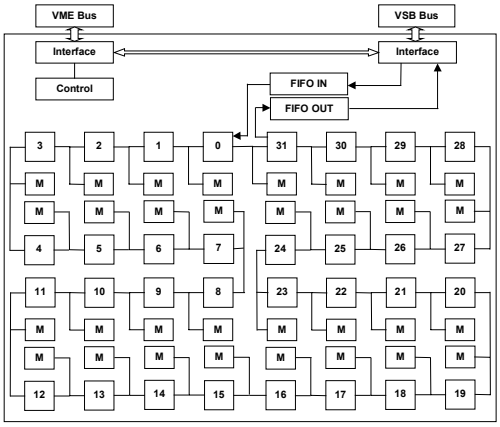


Figura 35 - Interface VME, VSB e matriz de 32 estágios

O hardware do SPLASH representado pela Figura 36, consiste em duas placas conectadas a uma estação de trabalho, hospedeiro (estação de trabalho SUN/UNIX), que compartilham um barramento VSB. Uma das placas é o processador SPLASH ilustrado na Figura 35, com uma interface VME para o hospedeiro, uma interface de VSB para organizar a memória e uma matriz linear de 32 estágios. O outro processador é uma placa com 8 Mbytes de memória conectada aos barramentos VSB e VME.

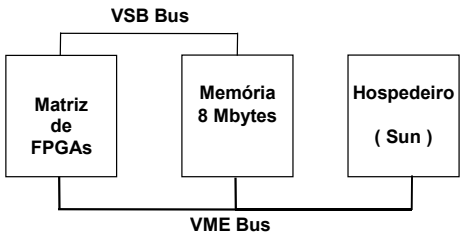


Figura 36 - Sistema SPLASH

A máquina SPLASH é configurada a partir das configurações individuais dos FPGAs, conectados em uma cadeia de 32 estágios. A configuração é realizada através do barramentos VME quando o SPLASH é inicializado. Paralelamente, um programa supervisor de autoria do usuário, executado na estação de trabalho SUN controla a execução dos algoritmos sistólicos no SPLASH.

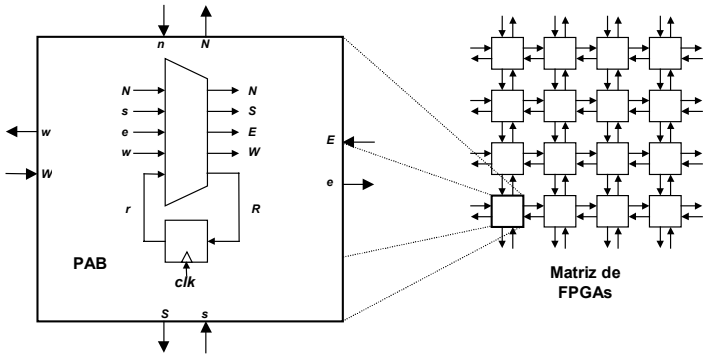
Com estas características de reconfiguração, podemos classificar esta arquitetura como Múltipla, pois pode ser reconfigurada a cada novo algoritmo. Uma linguagem interpretada, denominada *Trigger*, permite aos usuários carregar e depurar os programas. Esta linguagem permite a acesso a uma biblioteca básica de procedimentos,

bem como acesso a funções escritas pelos usuários, e permite que o SPLASH seja visto como um conjunto de 32 processadores.

No SPLASH, cada FPGA pode ser configurado através da interconexão serial de todos os processadores. Deste modo, o SPLASH tem a propriedade de reconfigurabilidade Parcial e esta propriedade é melhor explorada no SPLASH 2, onde é possível a reconfiguração da arquitetura da própria cadeia de interconexão. A escolha do algoritmo sistólico é realizada de acordo com a área de aplicação. Desta forma, a ordem de reconfiguração não é determinada, classificando a arquitetura como Aleatória. Uma vez que as rotinas são definidas e o algoritmo configurado na matriz, esta permanece inalterada até uma nova aplicação, caracterizando uma arquitetura de classificação Passiva.

3.4 DEC-Perle

DEC-Perle [40] é um projeto desenvolvido pelo DEC PRL (*Paris Research Lab*). Trata-se de implementações de uma arquitetura genérica denominada PAM(*Programmable Active Memories*), uma matriz de células idênticas conectadas ortogonalmente, onde cada célula é um PAB (*programmable active bit*), como ilustrado na Figura 37. Cada PAB pode realizar simultaneamente funções Booleanas quaisquer de até 5 entradas. Um protótipo de arquitetura reconfigurável baseada em PAMs, denominado DEC-Perle-0, foi construído e testado utilizando uma matriz de 25 FPGAs Xilinx XC3020. Entre as aplicações implementadas nesta arquitetura reconfigurável estão problemas como aritmética especializada (implementação de multiplicadores complexos, com até 512 bits), compressão de dados e processamento de imagens. Esse protótipo foi sucedido pelo DEC-Perle-1, que utiliza uma matriz de 24 FPGAs XC3090.



Este PAB tem 4 entradas (n,s,e,w), 4 saídas (N,S,E,W), um registrador (flip-flop) com entrada R e saída r, e um circuito combinacional g(n,s,e,w,r) = (N,S,E,W,R).

A tabela verdade de g é especificada por 160 = 5 x 32 bits

Figura 37 - Matriz de FPGAs e detalhe da célula PAB

A estrutura genérica de uma PAM é vista Figura 38. Ela é conectada por uma interface de entrada e saída a um processador hospedeiro. Uma das funções do hospedeiro é carregar o vetor de configuração na PAM. Observa-se que uma PAM é conectada a um barramento de alta velocidade do computador de hospedeiro, como qualquer módulo de memória de RAM. O processador pode escrever na e ler da PAM, porém ao contrário de uma RAM, uma PAM processa dados entre instruções de escrita e leitura, caracterizando PAMs como memórias ativas.

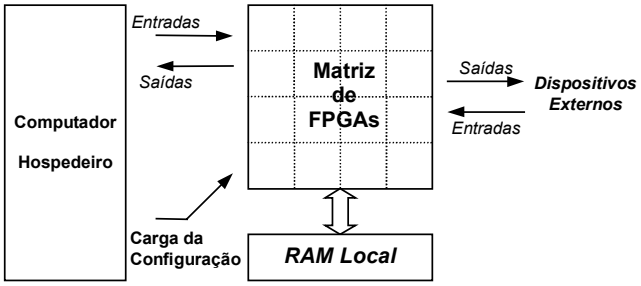


Figura 38 - Arquitetura genérica de um processador baseado em PAMs

Em algumas aplicações, tais como criptografia, não é necessário o uso de memória local. Contudo a maioria das aplicações requerem alguma quantidade de memória RAM local e/ou para reordenar dados locais.

Nas PAMs, um programa é composto de três partes:

- software que controla o hardware e é executado no computador hospedeiro;

- A configuração dos blocos lógico que descreve o hardware síncrono implementado na PAM;

- As diretivas de posicionamento e roteamento que orientam a implementação da lógica no interior do PAM.

O software de controle é escrito em C ou C++ e é integrado a uma biblioteca que encapsula o "driver" do dispositivo. A configuração dos blocos lógicos e as diretivas de posicionamento e roteamento são geradas por algoritmo no programa em C++. Um FPGA extra implementa o *Firmware* (interface de alta velocidade, 1.2GB/s). Este, analogamente a uma ROM, não pode ser configurado pelo usuário e sua função é prover, através do hospedeiro, o controle da PAM, bem como o protocolo para ajudar a reconfiguração da PAM durante execução.

O propósito da PAM é implementar uma máquina virtual, que pode ser configurada dinamicamente em um grande número de dispositivos de hardware específicos. Isto demonstra uma arquitetura configurável que pode ser classificada como Simples. Depois da configuração, a PAM comporta-se, eletricamente e logicamente, como um ASIC (*application-specific integrated circuits*) definido por um vetor de configuração específico, desta forma, a classificação para a dimensão do vetor é Total. Ela pode operar de forma autônoma, conectado a algum dispositivo externo ou operar como um coprocessador sob controle do hospedeiro. A PAM também pode operar como ambos, conectado ao hospedeiro e em algum dispositivo externo, como um dispositivo áudio ou vídeo, ou alguma outra PAM. Não existe definida ordem de reconfiguração, pode-se classificar a arquitetura da PAM, quanto a ordem de configuração, como Aleatória. Quanto a atividade PAMs podem ser classificadas como de configuração Ativa ou Passiva dependendo da aplicação.

4 IMPLEMENTAÇÃO DA ARQUITETURA PROPOSTA

O objetivo deste trabalho é exemplificar a utilização da técnica de reconfiguração dinâmica, através do desenvolvimento de uma Arquitetura Reconfigurável. Sendo assim, foi necessário construir uma arquitetura baseada em um computador hospedeiro para o desenvolvimento (interface com o usuário, programação e síntese das imagens de hardware) e uma plataforma de prototipação que contenha o dispositivo reconfigurável (FPGA), com os recursos de memória necessários para o armazenamento de dados, estados intermediários (dados comunicados entre configurações sucessivas) e múltiplas configurações. A Figura 39 mostra a estrutura geral da arquitetura que é apresentada neste trabalho.

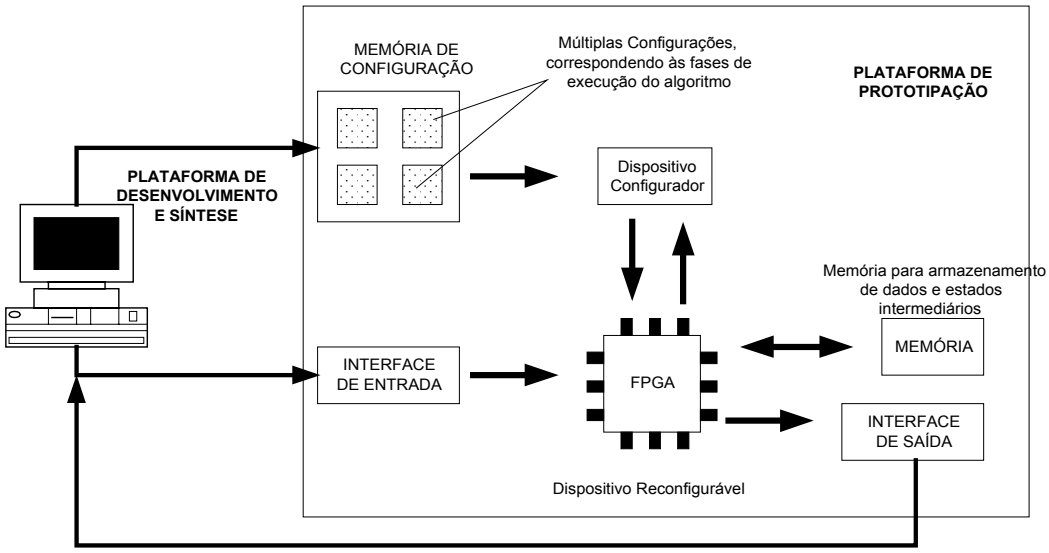


Figura 39 - Diagrama da arquitetura proposta

A implementação do sistema foi dividida em 3 etapas distintas, que serão descritas a seguir :

- Hardware da placa de prototipação / Desenvolvimento do programa básico;
- Software de Gerenciamento da Placa / Transferência de Configurações;
- Aplicação

4.1 HARDWARE

Durante a fase de definição da placa de prototipação utilizada neste trabalho, duas arquiteturas comerciais foram analisadas:

- Xess XS40/XS95 [41]
- AEE 1199^{A} [42]

Estas duas plataformas, cuja característica comum é a utilização de FPGAs da série XC4000 da Xilinx [9], possibilitam a prototipação de projetos envolvendo a programação de dispositivos FPGA de maneira flexível e com baixo custo.

No entanto, nenhuma das placas conta com uma memória específica para armazenamento de configurações. Dada a reduzida área destinada a expansões em ambas as placas, não seria possível adicionar o hardware necessário para implementação desse recurso, essencial para a realização do presente estudo.

Em função disso, chegou-se a conclusão de que seria interessante construir uma nova plataforma de prototipação, que atendesse às necessidades de reconfiguração relacionadas a este trabalho. Usando como referência as plataformas citadas anteriormente, foi projetada e construída uma nova placa de prototipação. O diagrama de blocos desta plataforma é apresentado na Figura 41, e a descrição detalhada dos módulos é feita nas Seções 4.1.1 e 4.1.2.

Como a confecção de uma placa de circuito impresso demandaria muito tempo, optou-se por montar a placa utilizando uma "proto-board", que agilizou a fase de construção e também facilitou as modificações e testes que aconteceram durante o desenvolvimento do projeto. O custo total do projeto também foi reduzido em função disso: todo o material necessário para a montagem do protótipo custou cerca de R$300,00 (Para referência, na data do desenvolvimento deste documento, um dólar equivale a R$ 2,35). A Figura 40 mostra uma foto da placa em seu estágio final.

Figura 40 - Foto da placa de prototipação

Optou-se por utilizar a mesma plataforma de FPGA's (Xilinx Série XC4000) na placa protótipo, já que mantendo a compatibilidade com as placas disponíveis, trabalhos já realizados poderiam ser reutilizados com uma necessidade mínima de alterações.

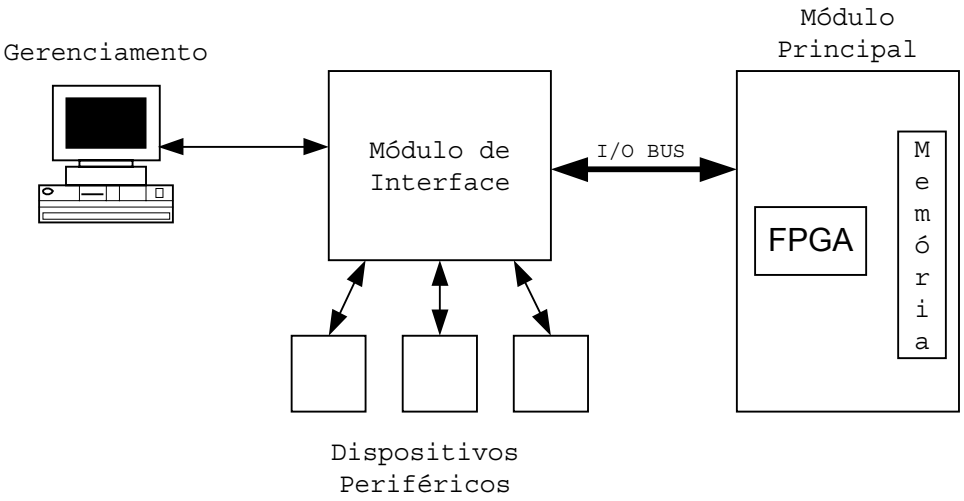


Figura 41 - Diagrama de blocos do Hardware

4.1.1 Módulo Principal

O módulo principal da placa protótipo contém os dispositivos necessários para uma implementação de uma arquitetura auto-reconfigurável. A memória da placa está particionada em 4 segmentos, cada um deles podendo receber uma imagem a ser transferida para o FPGA, ou pode funcionar como memória convencional, para

armazenamento de dados ou manutenção do estado do processamento durante uma troca de configuração.

A estrutura de mapeamento de memória possibilita o chaveamento entre diferentes configurações armazenadas, tornando possível a auto reconfiguração do sistema durante a execução, sem necessidade de um gerenciamento externo ao circuito.

Além disso, optou-se por acrescentar uma memória FLASH adicional, com a finalidade de manter um programa básico, que permitisse ao usuário a gerência das configurações armazenadas sem a necessidade de conexão à plataforma de desenvolvimento. Esse programa é acionado sempre que a placa é ligada, e também pode ser utilizado como um sistema de "recovery", caso aconteça algum problema durante a execução do programa que o sistema estiver executando.

A Figura 42 mostra o diagrama esquemático do módulo principal.

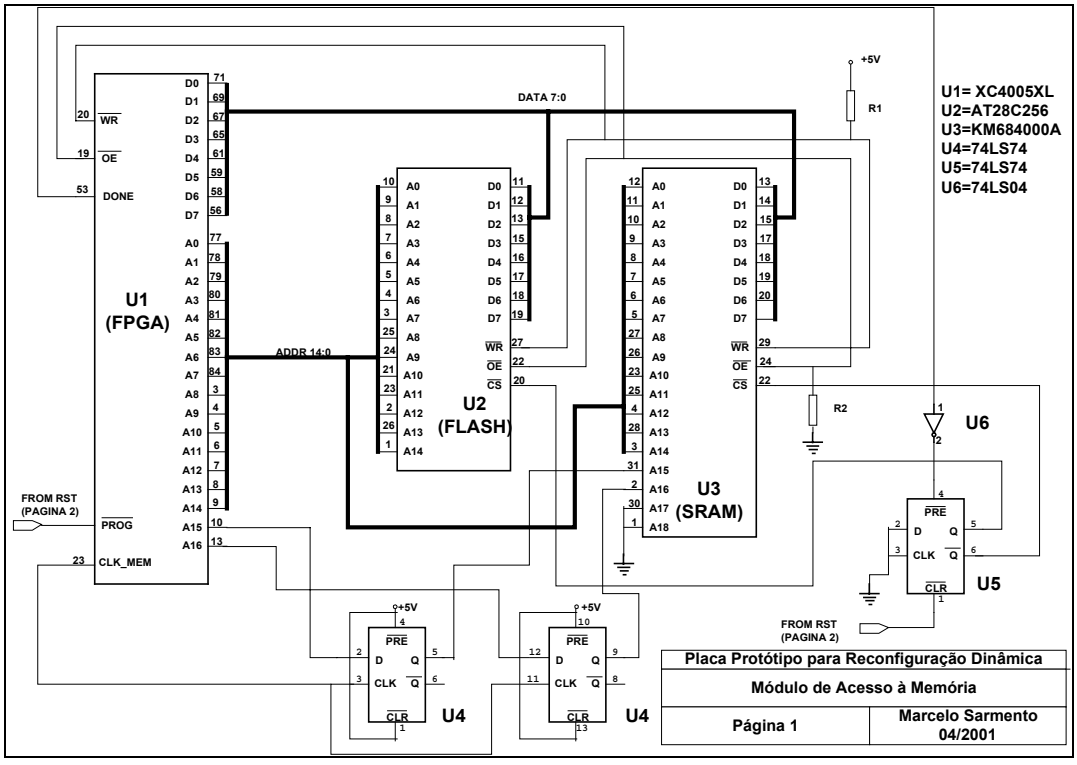


Figura 42 - Diagrama do módulo de controle e acesso à memória

4.1.1.1 Funcionamento do circuito de “startup”

Logo após o “power-up”, o circuito de Reset, aplicado ao pino “Prog” do FPGA (Xilinx XC4005) força a inicialização do sistema. Durante a sua fase de inicialização, o FPGA “limpa” qualquer resíduo de configuração na sua memória interna e inicia a fase de programação. Os níveis aplicados aos pinos M0-M2 determinam o modo de configuração do dispositivo. Nesta arquitetura, a carga da configuração é feita através de uma memória externa de 8 bits, através de um gerenciamento executado pelo próprio FPGA, que controla o endereçamento desta memória e a aquisição dos dados. Este método é denominado Master Paralell Up [9], e requer uma imagem de cerca de 20Kbytes (no caso específico do XC4005, já que dispositivos da mesma família tem requisitos diferentes, de acordo com a sua capacidade) para a sua programação. Essa necessidade determinou a escolha da memória FLASH (U2, AT28C256), que tem uma capacidade de 32Kb.

O circuito de RST também está aplicado ao pino de Clear do U5 (74LS74), responsável pela habilitação das memórias de configuração da placa. Neste estado, o pino Q é levado ao nível lógico 0, selecionando a memória FLASH (que contém o programa básico da placa).

Como pode ser observado através do diagrama da Figura 42, os barramentos de dados e endereços, tanto da memória FLASH (U2) como da memória SRAM (U3), estão conectados diretamente ao FPGA (U1).

Nesta fase de configuração, é necessário controlar parcialmente o acesso à memória, já que o FPGA assume que a memória estará disponível para leitura e gerencia apenas o endereçamento (através de incremento/decremento automático) e a leitura dos dados. Como as saídas do FPGA só atingem um estado lógico estável após a finalização do processo de configuração, foi necessário “forçar” o pino OE (Output Enable) das memórias a nível lógico “0”, o que é feito através do resistor R2.

Ainda com relação ao controle de acesso à memória, o pino WR (Write Enable, ou habilitação para escrita) é mantido em nível lógico “1” durante a configuração pelo resistor R1. Isso foi necessário para que a instabilidade durante a fase de programação

do FPGA, ou uma transicão em um processo de reconfiguração não determinasse uma eventual alteração das imagens (dados) armazenados nas memórias.

Assim que a carga da imagem inicial é finalizada, o FPGA sinaliza o fim do processo através do sinal DONE. Esse sinal (após uma inversão através de U6) é aplicado ao pino PRESET de U5, o que causa a habilitação da memória de configuração (SRAM, U3).

A imagem contida nesta FLASH não faz parte da aplicação, mas programa o FPGA para que seja possível a transferência das imagens da aplicação para a memória de configuração. Decidiu-se utilizar o próprio FPGA para essa função porque isso reduziu drasticamente o número de circuitos integrados a ser utilizado como lógica periférica.

A Figura 43 mostra o diagrama de blocos da imagem contida na FLASH e que é carregada no FPGA e ativada após a inicialização:

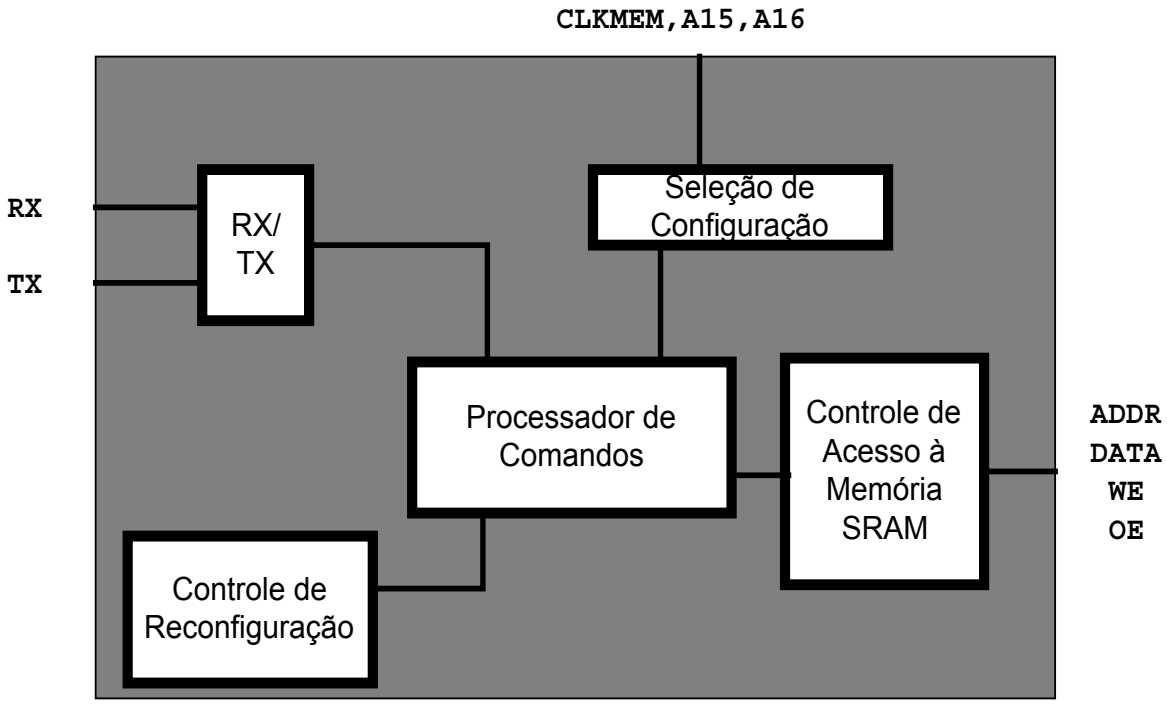


Figura 43 - Diagrama de blocos da imagem de "startup"

A partir da carga inicial, a placa já pode receber as imagens da aplicação, via comunicação padrão RS232 com o computador que está gerenciando a configuração.

O algoritmo implementado na imagem armazenada na FLASH é apresentado no diagrama da Figura 44. A listagem da implementação em VHDL deste algoritmo é apresentada no Anexo II.

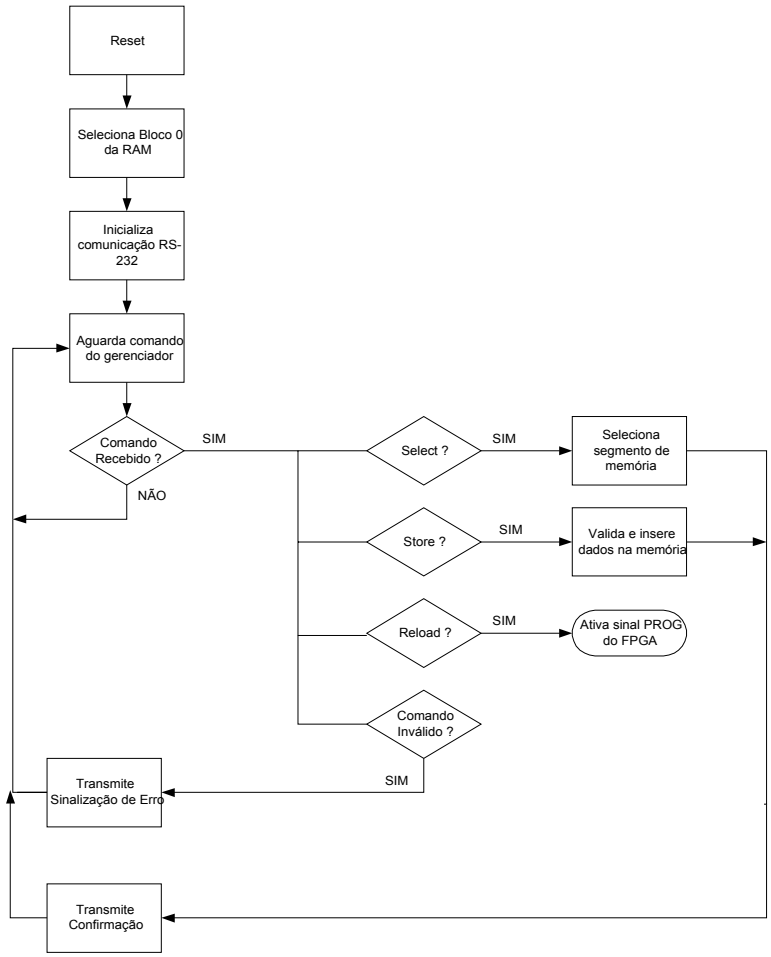


Figura 44 - Algoritmo de inicialização do sistema

4.1.1.2 Circuito de seleção de banco de memória

Para armazenar diferentes configurações, foi necessário particionar a memória SRAM em blocos. Cada partição, ou bloco, recebe uma numeração (0 a 3) e tem 32 Kbytes disponíveis, que podem ser utilizados para armazenamento de dados,

configurações, ou resultados intermediários (troca de informações entre configurações sucessivas).

O controle da seleção destes blocos é feito pelo próprio FPGA, através do controle dos sinais A15 e A16 pela aplicação que está sendo executada.

Para evitar que a informação de banco ativo (ou a ser ativado, durante uma troca de configuração) não seja perdida, utilizou-se uma configuração de "latch" para estes dois sinais, através de U4 (74LS74).

A seqüência para a seleção do banco ativo é demonstrada a seguir :

- Selecionar o banco desejado, através de A15 e A16
- Executar uma transição do estado lógico "0" para "1" no sinal CLK_MEM, o que determina que as saídas Q do 74LS74 assumam o estado de A15 e A16

Desta maneira, o estado dos sinais A15 e A16 é mantido mesmo durante uma reconfiguração do FPGA.

O fragmento de código abaixo mostra um exemplo em VHDL de como o chaveamento dos bancos de memória pode ser implementado.

```
type maquina is (s0,s1,s2,s3); -- máquina de estados
signal controle : maquina;
signal trigger : sdt_logic; -- controle do disparo de chaveamento do banco
if trigger = '0' then
  a15 <= '0' ; -- seleciona, por exemplo,  o banco número 2
  a16 <= '1' ;
  controle <= s0;
elsif clk'event and clk = '1' then
   case controle is
        when s0 => clk_mem <= '0'; controle <= s1; a15 <= '0'; a16 <='1';
        when s1 => clk_mem <= '1'; controle <= s2; -- pulso de clk para o 74LS74
        when s2 => clk_mem <= '0'; controle <= s3;
        when s3 => null;
   end case;
end if;
end process;
```

4.1.1.3 Controle de Reconfiguração

Este bloco do circuito é responsável pela carga de uma nova configuração para o FPGA, e funciona em conjunto com o circuito de seleção de bancos de memória. Uma vez selecionado o banco de memória, o FPGA deve ser inicializado novamente, para que a nova imagem seja carregada da memória para o FPGA. Durante a fase de projeto, um dos pinos de I/O foi selecionado para essa função.

Este pino –REPROG- é ativo em nível 1, e é aplicado através de um inversor (U6) ao pino PROG do FPGA. Uma vez que este pino é ativado (em nível 0, após a passagem pelo inversor), uma nova seqüência de programação do FPGA é iniciada.

É importante ressaltar que neste momento o banco de memória contendo a próxima configuração a ser carregada já deve ter sido selecionada. O mapeamento dos dados e configurações nos bancos de memória e a sua seqüência de ativação são responsabilidade do projetista da aplicação. Após a finalização de um estágio de processamento, a ativação do sinal de REPROG é a última tarefa a ser executada.

Para utilizar o sinal REPROG:

- Manter o sinal REPROG em nível 0 a partir do reset inicial do sistema;
- Aguardar a finalização do processo, ou a necessidade da carga de outro etapa do processamento;
- Selecionar o banco de memória contendo a próxima configuração para o FPGA;
- Ativar o sinal REPROG em nível 1.

Como a entrada PROG é ativa em nível 0, seria mais lógico ligar o pino de REPROG diretamente, porém percebeu-se que imediatamente após a configuração do FPGA, todas as suas saídas são levadas ao nível 0 por alguns ciclos de clock, até que o sistema fique estável. Esse comportamento levaria o FPGA a um novo ciclo de programação, motivo pelo qual optou-se pela inversão desse sinal.

4.1.1.4 Fonte de Alimentação

A placa protótipo é alimentada por uma fonte de 12V DC. Com exceção do FPGA Xilinx 4005, que é alimentado com 3.3V, todos os outros componentes da placa são alimentados com 5V. Para fazer a redução da tensão de alimentação, um circuito com reguladores de tensão foi montado na própria placa. O diagrama completo da fonte de alimentação está no Anexo I.

Para evitar transientes e interferências, um capacitor de 100nF foi adicionado junto à alimentação de cada componente da placa.

4.1.2 Módulo de Interface

O módulo de interface da placa protótipo contém os dispositivos necessários para a aquisição de dados e apresentação de resultados (interface com o usuário), bem como um canal de comunicação entre a placa e o gerenciador de configurações (tipicamente um microcomputador).

Os dispositivos implementados neste módulo são:

- Um canal de comunicação RS-232, através de um conector padrão DB-9;
- Uma interface compatível com o Parallel Download Cable, da Xilinx;
- Uma chave push-button, de uso geral;
- Uma interface para monitores de vídeo padrão VGA, com possibilidade de exibição de 8 cores (Apesar do padrão VGA para cores ser analógico, neste protótipo usamos sinais digitais para a definição das cores RGB);
- Um display de 7 segmentos;
- 2 led's para indicação de estado.

O diagrama de blocos deste módulo é apresentado na Figura 45. O diagrama esquemático destes circuitos está no Anexo I.

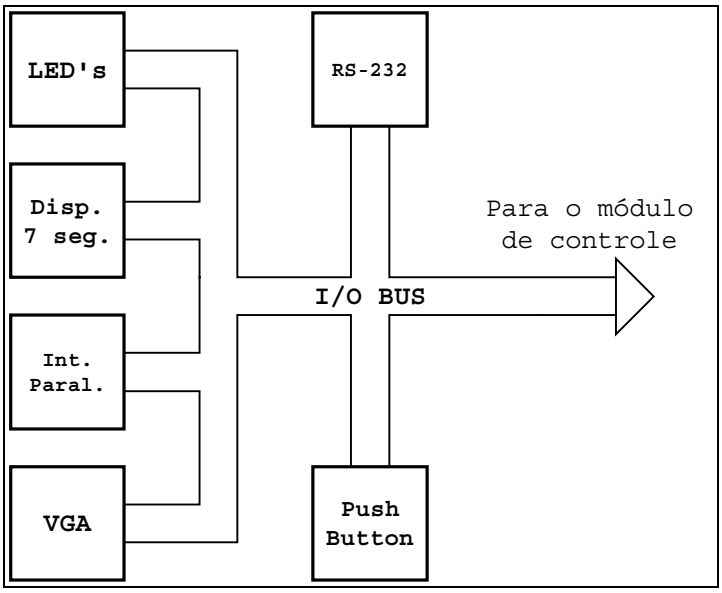


Figura 45 - Diagrama do módulo de interface

4.1.3 Dificuldades na construção do protótipo

Durante a construção da placa protótipo, algumas dificuldades foram encontradas, a maioria delas relacionada com a gravação da memória FLASH. Para que os testes com reconfiguração dinâmica pudessem ser efetuados, era essencial que a transferência de dados para a memória, e os circuitos de seleção de banco e REPROG estivessem funcionando. Essas tarefas estão todas no programa básico da placa (cuja listagem VHDL aparece no anexo II).

Como não havia um gravador de FLASH disponível, decidiu-se utilizar o próprio FPGA para realizar essa tarefa. Utilizando a interface paralela e as ferramentas de programação da Xilinx (Hardware Debugger), essa tarefa foi executada com um mínimo de hardware adicional.

Na construção do programa VHDL para gravação da FLASH, notamos que pequenas modificações no fonte VHDL geravam resultados imprevisíveis, ou mesmo causavam uma instabilidade em alguns sinais do projeto. Após pesquisa na documentação da Xilinx, verificamos que o modo com que são utilizadas algumas construções da linguagem VHDL podem interferir no desempenho do projeto e devem ser evitadas:

- **Utilização de STD_LOGIC em construções do tipo "CASE"**

Neste caso, o problema está relacionado com a quantidade de CLB´s utilizados. Como exemplo, utilizamos o código abaixo :

```
teste : std_logic_vector (1 downto 0);
dspl : std_logic_vector (7 downto 0);

case teste is

   when "00" => -- 0
   dspl <= "00000011";

   when "01" => -- 1

   dspl <= "10011110";

   when "10" => -- 2

   dspl <= "00100101";

   when "11" => -- 3

   dspl <= "00100101";

   when others =>

 dspl <= "00100101";


 end case;
```

Neste caso, como o sinal teste é do tipo STD_LOGIC_VECTOR, a claúsula "when others" precisa ser adicionada ao case, já que este tipo de sinal admite outros valores além de ´0´ e ´1´. Condições como ´1Z' ou ´Z0´ são válidas e precisam ser consideradas. O problema que isso acarreta está ligada à quantidade de hardware gerada pela ferramenta de síntese, necessária ao tratamento desses níveis adicionais. Para contorna essa situação, recomenda-se o uso do tipo BIT_VECTOR, já que este tipo apenas admite os valores ´0´ e ´1´, e a cláusula "when others", quando utilizada, gera menos hardware do que para um sinal do tipo STD_LOGIC_VECTOR.

- **Ferramenta de síntese gera a mensagem "Latch Inferred"**

Neste caso, todo o comportamento do circuito sintetizado torna-se instável. De acordo com a documentação da Xilinx [9], "FPGA Express will synthesize HDL code into a variety of logic elements, including flip flops, combinatorial gates, tri-state buffers, and latches. For Xilinx devices, internal latches are not an efficient use of resources because they will be built using combinatorial logic (if the particular family

targeted does not have internal latches), **so it is in the user's best interest to avoid latch inference whenever possible**. “

O código abaixo é um exemplo de código que gera um latch :

```
teste : std_logic_vector (1 downto 0);

teste <= "01";
```

O mesmo código, sem a geração de um latch :

```
teste : std_logic_vector (1 downto 0);

process (clk)

 begin
if clk´event and clk = ´1´ then

   teste <= “01”;
end if;

end process;
```

4.2 SOFTWARE

4.2.1 Aplicação de Gerenciamento das Configurações

Para possibilitar a seleção e a transferência das configurações para a memória da placa de prototipação, foi necessário o desenvolvimento de um aplicação de gerenciamento.

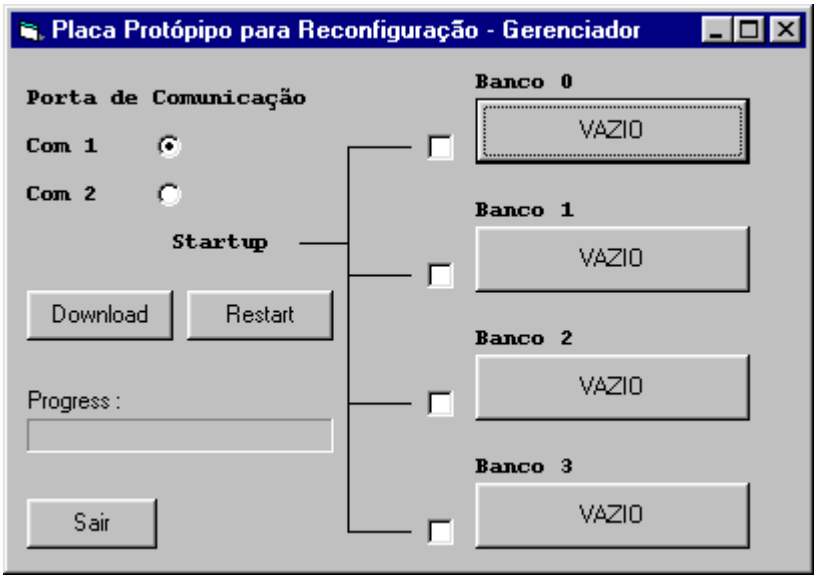


Figura 46 – "Screen Shot" do Software de Gerenciamento - Tela Inicial

Esse software, cuja tela inicial é mostrada na Figura 46, pode ser executado na plataforma de desenvolvimento e síntese e tem como funções principais:

- Seleção das imagens das configurações a serem transferidas para a placa e o segmento de memória destino;

Para cada segmento de memória da placa, existe um botão correspondente na interface do programa de configuração. Um "clic" neste botão abre uma janela secundária, onde um arquivo no formato Intel MCS-86 (extensão .mcs), pode ser selecionado. Este arquivo pode conter tanto uma imagem de configuração, resultado da ferramenta de síntese, como dados para preenchimento da memória, com informações a serem processadas. Após a seleção do arquivo desejado, o nome do arquivo passa a aparecer no texto do botão, como apresentado na Figura 47.

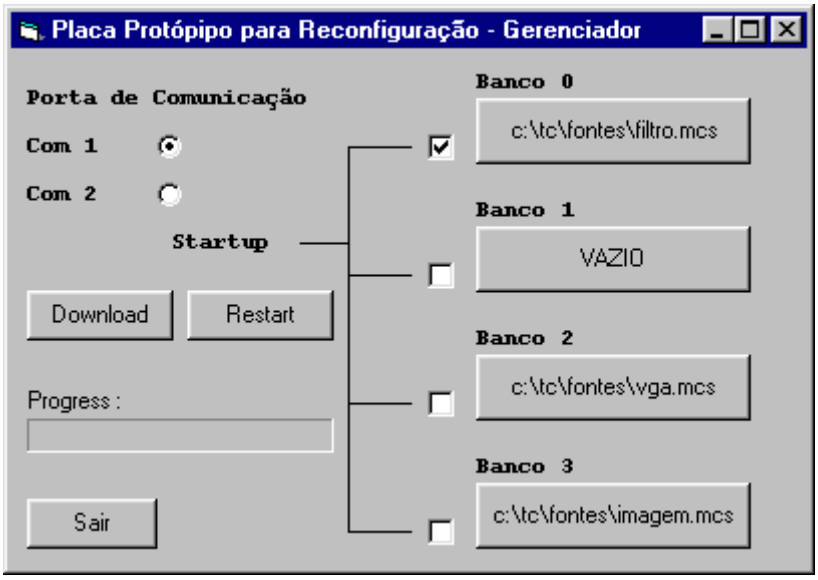


Figura 47 - Programa de configuração após a seleção das imagens

- Determinação da imagem a ser carregada para a placa no momento da inicialização;

É tarefa do gerenciador determinar qual será a imagem a ser transferida para o FPGA na inicialização do sistema. Um "check box" relacionado ao label "Startup" está associado a cada botão correspondente aos segmentos de memória da placa, e seleciona a aplicação que terá o controle da placa logo após o final da fase de transferência de dados. A Figura 48 detalha esse processo.

Figura 48 - Detalhe da aplicação de configuração: Seleção da imagem de "Startup"

- Definição da porta serial a ser utilizada na comunicação;

Permite a seleção da porta de comunicação, recurso útil caso o computador utilizado já tenha algum outro dispositivo conectado às portas principais. A seleção é feita através de um conjunto de "radio buttons" disponível na tela principal do programa e detalhado na Figura 49.

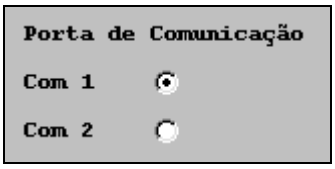


Figura 49 - Detalhe da aplicação de configuração: Seleção da porta de configuração

- Transferência das imagens para a placa de prototipação.

Um "click" no botão **Download** inicia a transferência dos dados para a placa de prototipação. Uma estatística do andamento do processo é indicada durante a transferência através da barra "% transferidos ".

4.2.2 Protocolo de transferência de dados

Para que não exista a possibilidade de perda ou alteração dos dados durante o processo de transferência dos dados entre o hospedeiro que gerencia as imagens e a placa protótipo, decidiu-se utilizar um protocolo de comunicação para garantir a integridade dos dados.

Este protocolo está baseado em um padrão *Request-Response*, onde cada comando é enviado através de um pacote de dados, e é esperada uma resposta de confirmação, indicando o recebimento e correta execução do comando enviado. Neste modelo, a comunicação sempre é iniciada pelo computador hospedeiro gerenciador.

O formato do pacote de dados é apresentado na Figura 50.

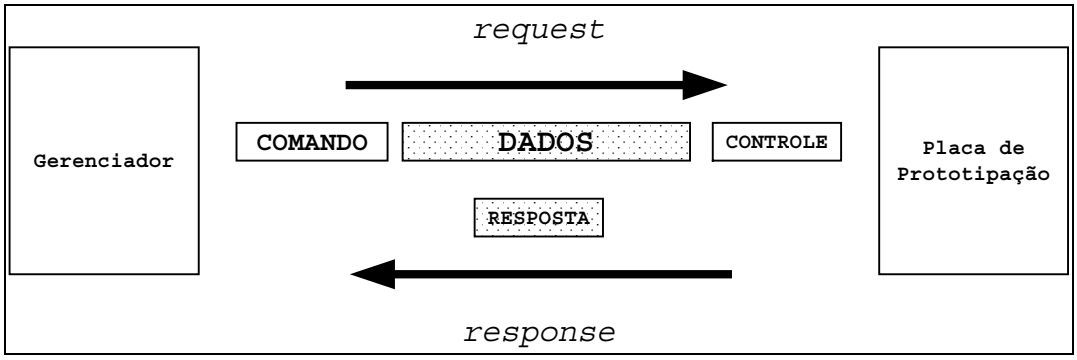


Figura 50 - Protocolo de comunicação

Cada pacote do tipo *request* é subdividido em 3 partes:

- Comando: 1 byte, contém o comando a ser executado;
- Dados: número variável de bytes, dependendo do comando enviado;
- Controle: 1 byte, utilizado para validação dos dados, contém o checksum do campo de dados.

Os pacotes do tipo *response* contém apenas 1 byte, e podem apenas assumir os valores *ACK*, indicando a aceitação do comando, ou *NACK*, indicando rejeição. No caso de recebimento de um *NACK* por parte do gerenciador, o comando que foi rejeitado deve ser enviado novamente.

Nos pacotes do tipo *request*, o campo de comandos pode assumir 3 valores:

- *Select*: Indica uma requisição de seleção do segmento de memória onde os dados enviados através do comando *Store* devem ser armazenados. Neste caso, o campo *dados* do pacote é composto por 1 byte, e contém o segmento de memória a ser selecionado.

- *Store*: Transfere dados para a memória. O campo *dados*, neste caso é composto pelo endereço da página de memória a ser gravada (2 bytes), seguidos por 16 bytes, que são o conteúdo a ser gravado nesta página. Este formato permite o endereçamento de até 1Mbyte por segmento de memória.

- *Reload* : Indica a finalização da transferência dos dados e também sinaliza à placa que o processamento deve iniciar. Neste caso, o campo *dados* do pacote é composto por 1 byte, e contém o segmento de memória a ser selecionado na inicialização do sistema.

4.3 APLICAÇÃO EXEMPLO

Para demonstrar a funcionalidade da placa protótipo das técnicas de auto-reconfiguração abordadas, a implementação de uma aplicação exemplo, um sistema de reconhecimento de padrões em imagens, foi implementado. Além da utilização dos recursos e das técnicas, esta aplicação é utilizada como modelo comparativo em relação às arquiteturas convencionais (entenda-se por arquitetura convencional, neste caso, como um circuito estático, onde todo o processo seria realizado através de um único componente de hardware).

A Figura 51 demonstra o diagrama de blocos da aplicação exemplo.

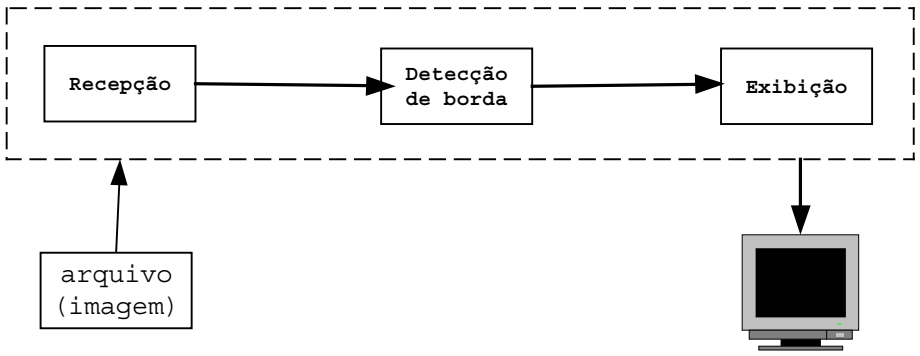


Figura 51 - Diagrama em blocos da aplicação

Como pode ser observado através da Figura 51, o objetivo desta aplicação é a detecção de bordas em uma imagem. Apesar desse problema poder ser abordado como uma tarefa única, esta tarefa pode ser subdividida em etapas menores não simultâneas:

- Aquisição da imagem e armazenamento na memória de trabalho, através da porta de comunicação RS-232. Neste caso, a imagem foi gerada a partir de um arquivo no formato Bitmap (extensão .bmp), e um programa aplicativo foi desenvolvido para converter este formato no formato MCS necessário para que o gerenciador de aplicações pudesse executar o download dos dados.
- tecção das bordas dos objetos contidos na imagem, através de um algoritmo, conforme descrito em [43];

- Exibição dos resultados em um monitor de vídeo, através da interface VGA.

Os programas fonte em VHDL relativos à cada uma destas etapas e a descrição do processo de síntese estão no Anexo II.

4.3.1 Densidade Funcional

Para uma análise quantitativa entre o método convencional e a arquitetura reconfigurável, utilizamos a Densidade Funcional (apresentada anteriormente no Ítem 2.3). Esta métrica visa identificar as condições em que uma arquitetura reconfigurável tem uma melhor relação *custo x benefício* (em termos de área e tempo de execução) do que o seu correspondente estático.

Como a definição da Densidade Funcional está baseada na área do circuito e no tempo de execução, foi necessário determinar esses parâmetros. Para simplificação desta análise, tomamos a liberdade de fazer uma pequena modificação no conceito relacionado à medida da área: ao invés de utilizar a medida física (medindo a área de silício ocupada no chip), decidimos utilizar o número de CLB's (blocos lógicos internos do FPGA) utilizados no projeto, já que essa informação é disponibilizada pela ferramenta de síntese. Com relação aos tempos de execução, outra simplificação: como o algoritmo que implementa cada fase é o mesmo, os tempos são aproximadamente os mesmos. Por isso, o tempo de cada etapa será indicado como uma váriavel, ao invés de um valor absoluto.

A Tabela 3 demonstra o número de CLB's e o tempo de execução para cada uma das fases do processo.

Etapa do Processamento	No. de CLB's	Tempo
Transferência dos Dados/ Armazenamento em Memória	*108*	T_1
Detecção da Borda	*122*	T_2
Exibição da Imagem	98	T_3

Tabela 3 - Circuito Exemplo - Parâmetros

A partir destes dados, é possível calcular a Densidade Funcional para cada um dos modelos.

- Para o modelo estático:

  Neste caso, assumimos que para realizar todas as funções em um mesmo hardware, teríamos que concentrar as 3 etapas do processo em um mesmo componente. Sendo assim, podemos calcular a área necessária como sendo a soma do número de CLB's utilizados em cada uma das etapas, o que resulta em A = 122+108+98 = 328 (Essa é uma aproximação não muito realista, visto que provavelmente uma otimização de alguns recursos seria possível numa síntese única). Agrupando os tempos, $T = T_1 + T_2 + T_3$, podemos utilizar a fórmula (1), chegando a

$$D_{estático} = 1 / 328T.$$

- Para o modelo dinâmico:

  Neste caso, como a área do hardware é variavel, utilizamos o número máximo de CLB's do FPGA, que no caso do XC4005 utilizado na placa protótipo, é de 196.

$$D_{dinâmica} = 1 / 196T.$$

Utilizando a fórmula (2), estabelecemos a relação entre os dois modelos, ou o índice de melhoria na Densidade Funcional do circuito reconfigurável em relação ao estático.

$$I = \frac{\Delta D}{Ds} = \frac{Dr - Ds}{Ds} = \frac{1/196T}{1/328T} - 1 = 0{,}673$$

Como podemos observar, o circuito reconfigurável tem cerca uma Densidade Funcional 67% superior à do seu correspondente estático. Outro detalhe que pode ser observado é que o circuito estático ocuparia mais área do que a disponível no componente (se considerado o número total de CLB's utilizados), e outro com maior capacidade seria necessário para realizar o mesmo trabalho.

A conseqüência imediata, se utilizada uma Arquitetura Reconfigurável, é a redução dos recursos de hardware do componente reconfigurável utilizado no sistema. No entanto, os custos adicionais necessários à reconfiguração devem ser considerados.

Como o processo executado pela aplicação exemplo acontece uma única vez, o tempo de reconfiguração do circuito (cerca de 0,2 segundos, obtidos através da medição com osciloscópio do período de transição do pino INIT do FPGA), não é suficiente para impactar o resultado da análise como um todo. Uma aplicação com um maior número de passos de processamento é necessária para um estudo mais completo deste aspecto da Densidade Funcional. Deixamos como sugestão para um trabalho futuro.

5 CONCLUSÕES E TRABALHO FUTURO

O trabalho desenvolvido consistiu em um estudo sobre arquiteturas auto-reconfiguráveis. Em virtude da ausência de uma plataforma de desenvolvimento que possibilitasse a execução de uma aplicação auto-reconfigurável, uma plataforma protótipo, composta por um módulo de hardware e um módulo de software, foi construída.

Em virtude do longo tempo demandado na construção do protótipo do hardware, apenas uma aplicação de detecção de bordas em imagens estáticas foi utilizada para demonstrar a capacidade de auto-reconfiguração do sistema. Embora simples, essa aplicação é funcional e demonstra a possibilidade de utilização dessa tecnologia.

Como resultado imediato, verificamos que o número de CLB´s necessário para a implementação da aplicação exemplo na arquitetura reconfigurável tende a ser menor do que utilizada no seu correspondente estático.

A partir da construção da placa protótipo, abrem-se várias possibilidades, que sugerimos a seguir como trabalhos que poderiam dar continuidade ou utilizar as ferramentas que foram desenvolvidas ao longo deste estudo :

- Avaliações mais completas, através da utilização de aplicações mais complexas e com maior número de reconfigurações, são necessárias para avaliar o impacto do tempo de reconfiguração neste tipo de arquitetura;
- Utilização de FPGA´s com maior capacidade (maior número de portas) na placa protótipo;
- Comparação entre as técnicas de reconfiguração total versus reconfiguração parcial.

Como sugerido, acredita-se que este trabalho possa vir a ter uma continuação, seja pelo próprio autor ou por qualquer outra pessoa com interesse pelo assunto

abordado no presente trabalho. Contatos podem ser feitos com o autor através dos endereços eletrônicos:

msarmento@hotmail.com

msarmento@netcabo.com.br,

ou junto ao grupo de pesquisa que sediou este trabalho:

gaph-l@inf.pucrs.br.

6 BIBLIOGRAFIA

[1] INTEL. Pentium III Processor.: [http://www.intel.com/PentiumIII/index.html]

[2] Motorola Inc. PowerPC Microprocessors. [http://ebus.mot-sps.com/ProdCat]

[3] COHEN, D., Introduction to Computer Theory, Wiley & Sons, Inc. 1991.

[4] ADÁRIO, Alexandro M. S. *Implementação em FPGA de um Processador de Vizinhança para Aplicação em Imagens Digitais*. Campinas: Instituto de Computação, UNICAMP, 1997. Dissertação de Mestrado.

[5] DEHON, Andre. *DPGA-Coupled Microprocessors: Commodity Ics for the early 21st Century*. MIT Transit Project. [http://www.ai.mit.edu/projects/transit/tn100/tn100.html]

[6] SANCHEZ, E.; SIPPER, M.; HAENNI, J. O.; BEUCHAT, J. L.; STAUFFER, A.; URIBE, A. P. *Static and Dynamic Configurable Systems*. IEEE Transactions on Computers, vol 48, no. 6, Junho 1999.

[7] VILLASENOR, J.; SMITH, W. H. *Configurable Computing* in Scientific American, June 1997

[8] ROSE, J.; VINCENTENELLI, A. S. *Architecture of Field Programmable Gate Arrays*. Proceedings of the IEEE, vol 81 no. 7, Julho 1993.

[9] Xilinx Corporation, Inc. *2000 Xilinx Data Book*. [http://www.xilinx.com]

[10] Plessey Semiconductor. *ERA60IOO preliminary data sheet*, Swindon, England, 1989.

[11] Algotronix Ltd, Edinburg, Scotland. *CAL 1024 Datasheet*., 1989.

[12] Concurrent Logic., *CFA6006 Field Programmable Gate Array Data Sheet*, Sunnyvale, CA, 1991.

[13] MUROGA, H.; MURATA, Y.; SACKI, T.;OHASH, Y.; NUGUCHI, T., e NISHIMURA, T.;*A large scale FPGA with 10K core cells with CMOS 0.8μ 3-layered metal process*, Custom Integrated Circuits Conf, CICC, pp 6.4.1-6.4.4, Maio, 1991.

[14] EL GAMAL, A. et al, *An architecture for eletrically configurable gate arrays*, IEEE JSSC, vol 124, No.2, pp 339-398, Abril, 1989.

[15] BIRKNER, J.; CHAN, A.; CHUA, T.; CHAO, A.; GORDON, K.; KLEIRNAN, B.; KOLZE, R. e WONG, R. *A very high-speed field programmable gate array using metal-to-metal anti-fuse programmable elements*, in New Hardware Product Introduction at CICC, 1991.

[16] MARPLE, D e COOKE, L., *An MPGA compatible FPGA architecture in FPGA,* '92 ACM first international Workshop on Programmable Gate Arrays, pp 39-44, Fevereiro, 1992.

[17] Advanced Micro Devices, *Mach Devices High Density EE Programmable Logic Data Book*, 1990.

[18] BAKER, S. *Lattice Fields FPGA*, Elec. Fog. Times, No. 645, Junho, 1991.

[19] WONG, S.; SO, H.; OU, H. e COSTELLO, *A 5000 gates CMOS EPLD with multiple logic and interconnect arrays*, in Proceedings 1989 CICC, pp 5.8.1-5.8.4, Maio, 1989.

[20] De MICHELLI, G. e SAMI, M., *Hardware/Software Codesign*, pp 1-28, Kluver Academy Publishers, 1996.

[21] AHRENS, M. *et al., An FPGA family optimized for high densities and reduced routing delay,* in Proceedings 1990, CICC, pp 31.5.1-31.5.4, Maio, 1990.

[22] LALA, P., *Digital System Design Using Programmable Logic Devices*, Prentice Hall, 1990.

[23] VIJ, S.; AHANIM, B., *A high density, high speed array erasable programmable logic device with programmable speed/power optimization*, in ACM First mt. Workshop on Field Programmable Gate Arrays, pp 29-32, Fevereiro, 1992.

[24] HSIEH, H. *et al, A second generation user programmable gate array,* in Proc 1987 CICC, pp 15.3.1-15.3.7, Maio 1987.

[25] HSIEH, H. *et al, A 9000 gate user programmable gate array,* in Proc 1988 CICC, pp 31.2.1-31.2.7, Maio 1988.

[26] ROSE, J., FRANCIS, R.; LEWIS, D. e CHOW, P., Architecture of field programmable gate arrays: The effect of logic functionality on area efficiency, IEEE JSSC, Vol 25, pp 1217-1225, Outubro, 1990.

[27] CALAZANS, N., *Projeto Lógico Automatizado de Sistemas Digitais Sequenciais*. Rio de Janeiro, DCC/IM, COPPE / Sistemas, NCE/UFRJ, 1998

[28] PAGE, I., *Reconfigurable Processor Architectures*, [ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Ian.Page/proc_arch.ps.gz], Agosto, 1997.

[29] WIRTHLIN, M. e HUTCHINGS, B., *DISC: The dynamic instruction set computer*, in Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing. John Schewel, Editor, Proceedings of the SPIE 2607, pp 92-103, 1995.

[30] WIRTHLIN, M. e HUTCHINGS, B., *Improving Functional Density Using Run-Time Circuit Reconfiguration*, IEEE Transactions on Very Large Scale Integration Systems, Vol 6, No. 2, Junho, 1998

[31] ELDRIDGE, J., HUTCHINGS, B., BRAD, L., *RRANN: The Run-Time Reconfiguration Artificial Neural Network*, in Proceedings of Custom Integrated Circuit Conference, pp 77-80, 1994, San Diego, California.

[32] UTH, A., SINDAJI, V., SOMANATHAN, S., *Branch effects Reduction Techniques*, IEEE Maio 1997.

[33] TOROK, D., *Integração Hardware Reconfigurável / Redes de Interconexão em Arquiteturas Reconfiguráveis*, Trabalho Individual I, Maio 2000.

[34] CONWAY, L., MEAD, C., *Introduction to VLSI Systems*, Reading, MA: Addison/Wesley, 1980.

[35] SEITZ, C., *Concurrent VLSI Architectures, IEEE Transactions on Computers*, vol C-33, pp1247-1265, Dezembro, 1984.

[36] _____, *VLSI and paralell computation*, in Concurrent Architectures, Morgan Kaufmann, 1990, cap 1., pp 1-84.

[37] ATHANAS, P.; SILVERMAN, H., *Processor Reconfiguration Through Instruction Set Metamorphosis*, IEEE Computer, 26,pp 11-18, Março, 1993.

[38] GOKHALE, M., SPLASH: *A Reconfigurable Linear Logic Array*. [ftp://ftp.super.org/pub/fpga/splash-1/splash-1.ps], Maio, 1997.

[39] BUELL, D.; ARNOLD, W.; KLEINFELDER, W., *Splash2: FPGAs in Custom Computing Machine*, IEEE Computer Society Press, Los Alamitos, CA, 1996.

[40] BERTIN, P., RONCIN, D.; VUILLEMIN, J.*, Introduction to Programmable Active Memories*, Digital Equipment Corporation, Paris Research Lab, Junho 1989.

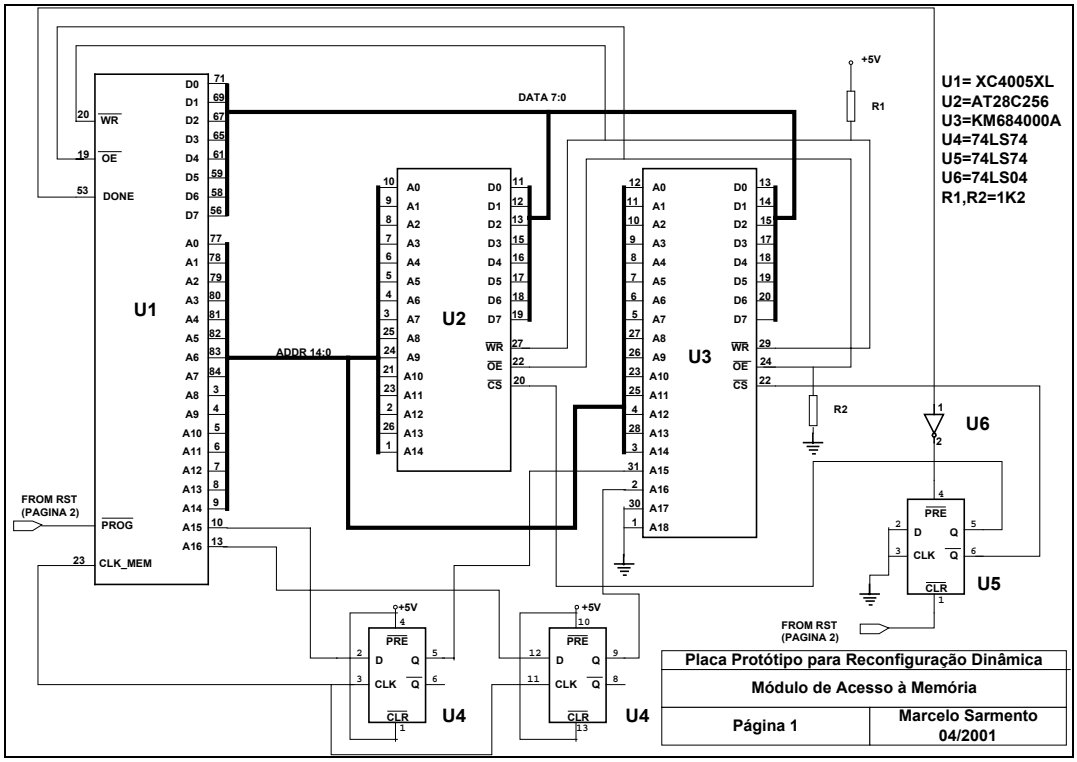
[41] Xess Corporation, *Xstend Board V1.2 Manual*, 1998 [http://www.xess.com]

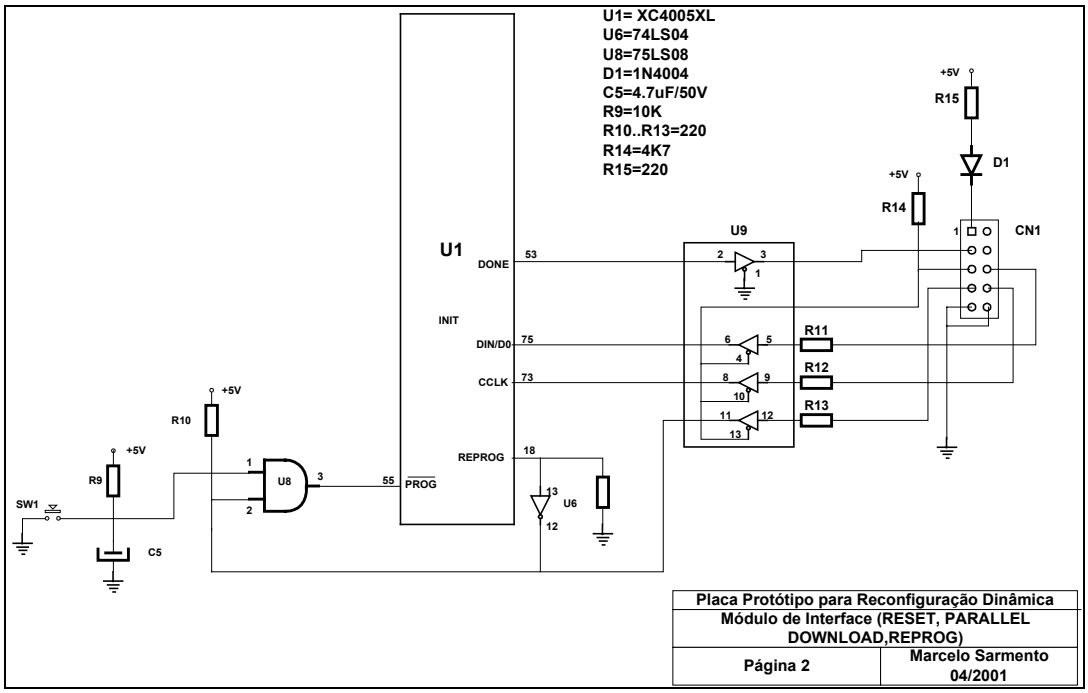
[42] AEE Engenharia Eletronica, *AEE 1199 FPGA Evaluation Board*, [http://www.aee.com.br/i1199.html]

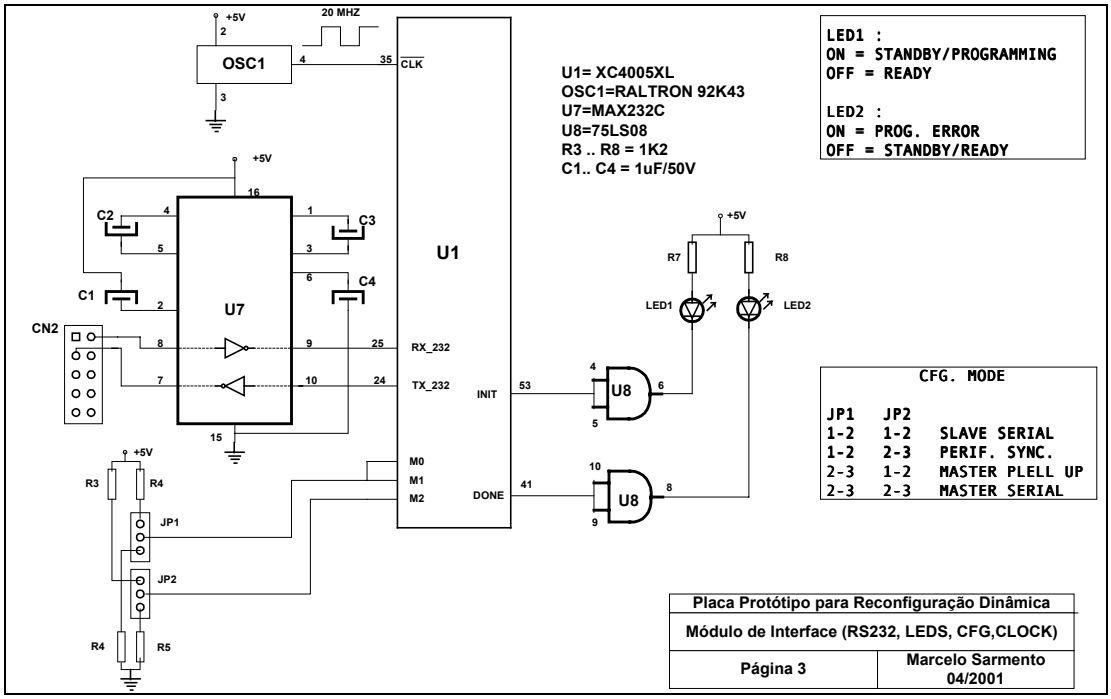
[43] PUCRS, Material de apoio a disciplina de Computação Gráfica, [http://www.inf.pucrs.br/~pinho/CG/Aulas/Img/IMG.htm]

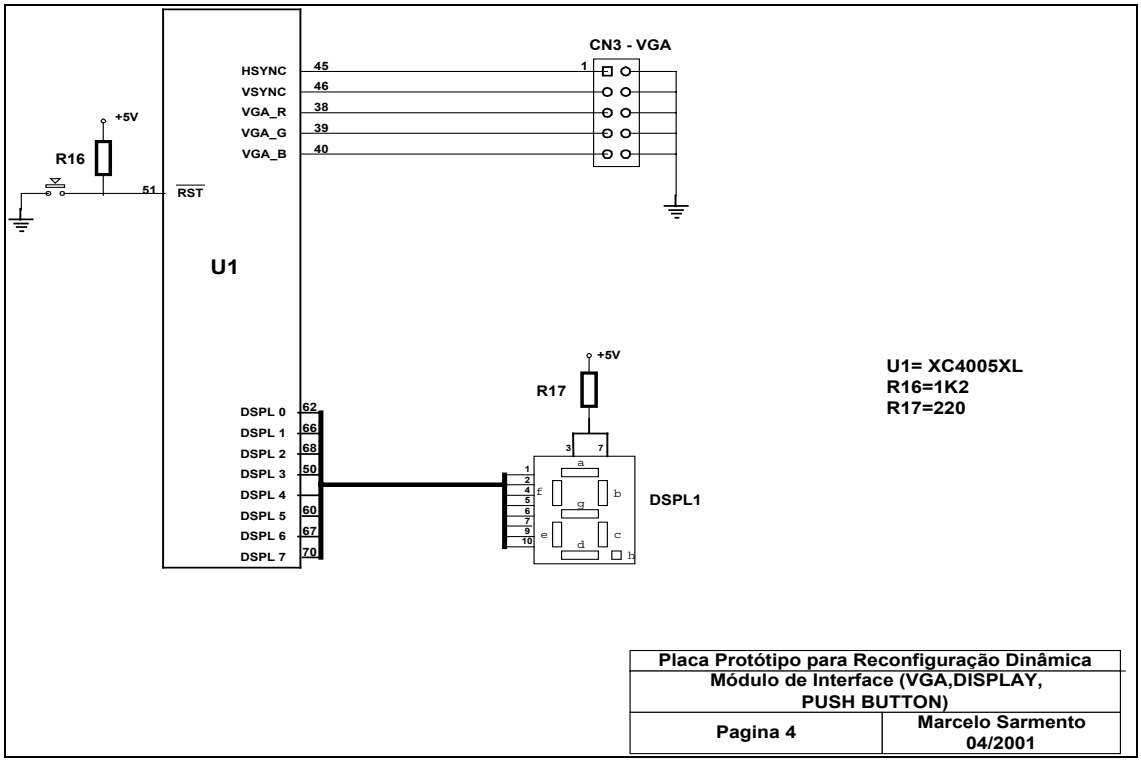
ANEXO I

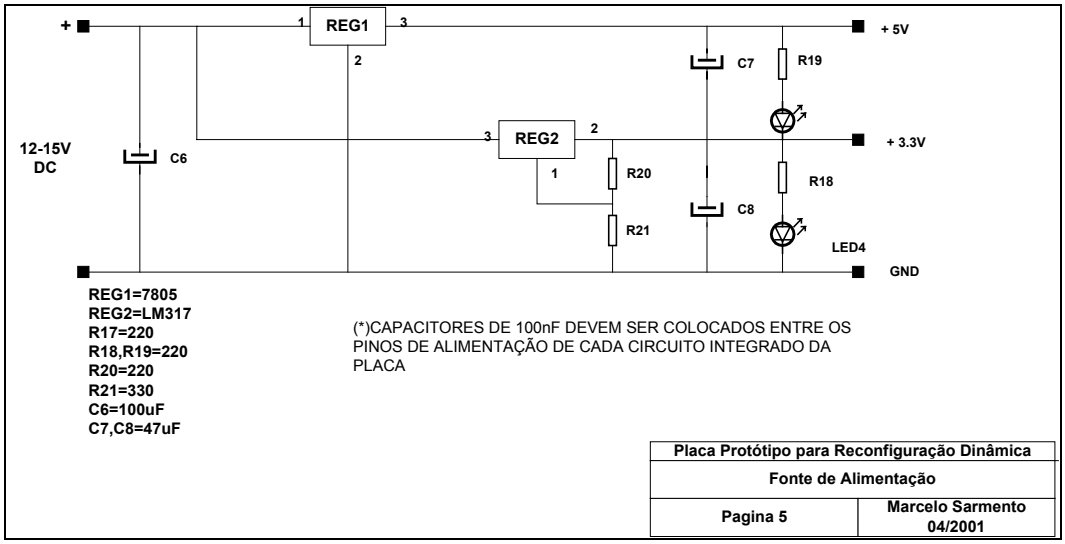
DIAGRAMAS ESQUEMÁTICOS DA PLACA PROTÓTIPO


U1= XC4005XL
U2=AT28C256
U3=KM684000A
U4=74LS74
U5=74LS74
U6=74LS04
R1,R2=1K2
DATA 7:0
ADDR 14:0
U1
U2
U3
U4
U5
U6
R1
R2
+5V
WR
OE
DONE
PROG
CLK_MEM
FROM RST (PAGINA 2)
PRE
CLR
CLK
D
Q
Placa Protótipo para Reconfiguração Dinâmica
Módulo de Acesso à Memória
Página 1
Marcelo Sarmento
04/2001


U1= XC4005XL
U6=74LS04
U8=75LS08
D1=1N4004
C5=4.7uF/50V
R9=10K
R10..R13=220
R14=4K7
R15=220
U1
U9
U8
U6
CN1
D1
R15
R14
R11
R12
R13
R10
R9
SW1
C5
+5V
DONE
INIT
DIN/D0
CCLK
REPROG
PROG
Placa Protótipo para Reconfiguração Dinâmica
Módulo de Interface (RESET, PARALLEL DOWNLOAD,REPROG)
Página 2
Marcelo Sarmento
04/2001


+5V
20 MHZ
OSC1
CLK
U1
U1= XC4005XL
OSC1=RALTRON 92K43
U7=MAX232C
U8=75LS08
R3 .. R8 = 1K2
C1.. C4 = 1uF/50V
LED1 :
ON = STANDBY/PROGRAMMING
OFF = READY
LED2 :
ON = PROG. ERROR
OFF = STANDBY/READY
C1
C2
C3
C4
U7
CN2
RX_232
TX_232
INIT
M0
M1
M2
DONE
U8
R7
R8
LED1
LED2
R3
R4
R5
JP1
JP2
CFG. MODE
JP1 JP2
1-2 1-2 SLAVE SERIAL
1-2 2-3 PERIF. SYNC.
2-3 1-2 MASTER PLELL UP
2-3 2-3 MASTER SERIAL
Placa Protótipo para Reconfiguração Dinâmica
Módulo de Interface (RS232, LEDS, CFG,CLOCK)
Página 3
Marcelo Sarmento
04/2001


CN3 - VGA
HSYNC
VSYNC
VGA_R
VGA_G
VGA_B
+5V
R16
RST
U1
R17
DSPL 0
DSPL 1
DSPL 2
DSPL 3
DSPL 4
DSPL 5
DSPL 6
DSPL 7
DSPL1
U1= XC4005XL
R16=1K2
R17=220
Placa Protótipo para Reconfiguração Dinâmica
Módulo de Interface (VGA,DISPLAY, PUSH BUTTON)
Pagina 4
Marcelo Sarmento
04/2001


+
1
REG1
3
+ 5V
2
C7
R19
12-15V
DC
C6
3
REG2
2
+ 3.3V
1
R20
R18
C8
R21
LED4
GND
REG1=7805
REG2=LM317
R17=220
R18,R19=220
R20=220
R21=330
C6=100uF
C7,C8=47uF
(*)CAPACITORES DE 100nF DEVEM SER COLOCADOS ENTRE OS PINOS DE ALIMENTAÇÃO DE CADA CIRCUITO INTEGRADO DA PLACA
Placa Protótipo para Reconfiguração Dinâmica
Fonte de Alimentação
Pagina 5
Marcelo Sarmento
04/2001

ANEXO II

LISTAGENS VHDL

```
-- Fonte vhdl para o programa base da placa protótipo
-- Marcelo Sarmento - 04/20001
-- File : base_prg.vhd
-----------------------------------------------------

-----------------------
-- IMPORTANTE !!!!!
-- CASO PRECISE MODIFICAR O PROJETO, EVITE CONSTRUCOES DO TIPO :

-- SINAL <= '1'; SEM QUE "SINAL" ESTEJA EM UM BLOCO PROCESS. A FERRAMENTA

-- DE SINTESE INFERE UM LATCH NESTE TIPO DE CONSTRUCAO, O QUE PODE TORNAR

-- O CIRCUITO RESULTANTE INSTAVEL.
-----------------------


LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY base_placa IS
PORT
 (
 clk: IN STD_LOGIC; -- 20 MHZ clock
 init_rst: IN STD_LOGIC; -- asynchronous reset
 address: OUT STD_LOGIC_VECTOR(14 DOWNTO 0);-- address para a SRAM
 a15 : out std_logic; -- enderecamento do banco da SRAM
 a16: out std_logic;
 clk_mem : out std_logic; -- clock para controle do banco

 data: INOUT STD_LOGIC_VECTOR(7 DOWNTO 0); -- data from RAM
 oe: OUT STD_LOGIC; -- RAM output enable
 we: OUT STD_LOGIC; -- RAM write enable
 dspl: OUT STD_LOGIC_VECTOR (7 DOWNTO 0); -- DISPLAY DE 7 SEGMENTOS


 reprog: OUT STD_LOGIC; -- controle de reload

 -- dados rs232

 rx_232: in STD_LOGIC;
 tx_232: out STD_LOGIC

 );
 END base_placa;

 ARCHITECTURE base_placa_arch OF base_placa IS



component serial
        -- componente da interface serial ( definido em serial.vhd)
        port(
                -- general control
                clk             : in  std_logic;      -- clock
                init_rst        : in  std_logic;      -- reset
                -- paralel interface (processor)
                data_in_232     : in  std_logic_vector(7 downto 0);
                data_out_232    : out std_logic_vector(7 downto 0);
                -- serial interface(outdoor)
                rx_232          : in  std_logic;
                tx_232          : out std_logic;
                --
                tx_completed    : out std_logic;
                rx_completed    : out std_logic
        );
        end component;



-- interface com o componente serial

signal serial_data_in: STD_LOGIC_VECTOR (7 DOWNTO 0);
signal serial_data_out: STD_LOGIC_VECTOR (7 DOWNTO 0);
signal serial_tx_completed: STD_LOGIC;
signal serial_rx_completed: STD_LOGIC;
```

```
signal disp_show: bit_vector (3 downto 0);

-- maquina de estados 1 - recepcao do comando

type type_state1 is (S0,S1,S2,S3,S4);
       signal EA1 : Type_STATE1;

-- maquina de estados 3 -> 3o processo
       type type_state3 is (S0, S1,S1A,s11a,S2, S3, S4, S5, S7,S8,s9);
       signal EA3 : Type_STATE3;

       signal trans_done : std_logic;
       signal mem_done : std_logic;
       --
        signal comando : std_logic_vector(7 downto 0);
       signal mem_address : std_logic_vector(14 downto 0);
       signal byte2 : std_logic_vector(7 downto 0);

        signal mem_out : std_logic_vector (7 downto 0);

        signal reprog_aux : std_logic;

 -- maquina de estados para o controle dos enderecos do banco da SRAM
type type_state2 is (S0,S1,S2);
       signal me2 : Type_STATE2;


 BEGIN



u2: serial
       port map
       (
       clk => clk,
       init_rst => init_rst,
       data_in_232 => serial_data_in,
       data_out_232 => serial_data_out,
       rx_232 => rx_232,
       tx_232 => tx_232,
       tx_completed => serial_tx_completed,
       rx_completed => serial_rx_completed
       );


reprogramacao :
process (init_rst,clk)
begin
if init_rst = '0' then
  reprog_aux <= '1';
else
if clk'event and clk = '1' then
   if comando = x"58" then
     reprog_aux <= '0';
   end if;
end if;
end if;
end process;

reprog <= not (reprog_aux);


process (init_rst,clk)
begin
  if init_rst = '0' then
     disp_show <= "0000";
  elsif clk'event and clk='1' then
    if mem_out (0) = '0' then disp_show(0) <= '0'; else disp_show(0) <= '1';end if;
    if mem_out (1) = '0' then disp_show(1) <= '0'; else disp_show(1) <= '1'; end if;
    if mem_out (2) = '0' then disp_show(2) <= '0'; else disp_show(2) <= '1'; end if;
    if mem_out (3) = '0' then disp_show(3) <= '0'; else disp_show(3) <= '1'; end if;

   end if;
end process;


 PROCESS(clk)
```

```
 BEGIN

  if (clk'event and clk='1') then

   case disp_show is

   when "0000" => -- 0
   dspl <= "00000011";

   when "0001" => -- 1

   dspl <= "10011110";

   when "0010" => -- 2

   dspl <= "00100101";

   when "0011" => -- 3

   dspl <= "00001100";

   when "0100" => -- 4

   dspl <= "10011001";

   when "0101" => -- 5
   dspl <= "01001000";

   when "0110" => -- 6
   dspl <= "01000001";

   when "0111" => -- 7
   dspl <= "00011110";

   when "1000" => -- 8
   dspl <= "00000001";

  when "1001" => -- 9
   dspl <= "00001000";

  when "1010" => -- A
   dspl <= "00010001";

  when "1011" => -- B
   dspl <= "11000000";

  when "1100" => -- C
   dspl <= "01100011";

  when "1101" => -- D
   dspl <= "10000100";

  when "1110" => -- E
   dspl <= "01100001";

  when "1111" => -- F
   dspl <= "01110000";

 end case;


end if;


END PROCESS;



process(init_rst, serial_rx_completed)
       begin

       if init_rst = '0' then

        EA1 <= S0;
        trans_done <= '0';
        comando <= x"00";
```

```
         byte2 <= x"00";

        elsif serial_rx_completed'event and serial_rx_completed = '0' then

          case EA1 is

             when S0 => -- armazena comando

               trans_done <= '0';
               comando <= serial_data_out;
               EA1 <= S1;

             when S1 => -- armazena primeiro byte do endereco

               mem_address(14 downto 8) <= serial_data_out(6 downto 0);
               EA1 <= S2;


              when S2 => -- armazena segundo byte do endereco

               mem_address(7 downto 0) <= serial_data_out;
               EA1 <= S3;


             when S3 => -- armazena dado     e ativa sinal

               byte2 <= serial_data_out;
               trans_done <= '1';
               EA1 <= S4;

               when S4 =>

               if mem_done = '1' then
                      byte2 <= (others => 'Z');
               end if;

               EA1 <= S0;

        end case;

               end if;

        end process;


process (init_rst, clk, trans_done)
        begin

               if init_rst = '0' or trans_done = '0' then
                      ea3 <= S0;
                      mem_done <= '0';
                      data <=  (others => 'Z');
                      --address <= (others => '0');
                      oe <='1';
                      --ce <= '1';
                      we <= '1';
                      a15 <= '0';
                      a16 <= '0';
                      --start_tx <= '0';

               elsif clk'event and clk = '1' then

               case ea3 is

                              when S0 =>     -- desativo a memoria e o comando

                                     oe <= '1';
                                     we <= '1';
                                     data <= (others => 'Z');
                                     address <= mem_address (14 downto 0);

                                     if comando = x"57" then       -- W
                                            ea3 <= S1;
                                     elsif comando = x"52" then    -- R
                                            ea3 <= S2;
                                     elsif comando =x"53" then -- S sel. banco
                                             ea3 <= S7;
```

```
                        else
                                ea3 <= S2;
                        end if;

                    when S1 =>     -- escrita e vai para S1a

                        data <=  byte2;
                        ea3 <= S1a;

                      when S1a =>
                             oe <= '0';  -- para a ram
                             --oe <= '1';  -- para a flash
                        we <= '0';
                        ea3 <= s11a;

                     when s11a =>


                     oe <= '0';
                     --ce <= '0';
                     we <= '0';
                     ea3 <=s4;


                    when S2 =>     -- leitura 1

                        oe <= '0';
                        we <= '1';
                        --ce <= '0';
                        ea3 <= S3;

                    when S3 =>     -- leitura 2 e vai para S4
                             -- display deve entrar aqui !!!!

                             mem_out <= data (7 downto 0);
                             ea3 <= S4;

                    when S4 =>     -- desativa a memoria e vai para S5

                        oe <= '1';
                        --ce <= '1';
                        we <= '1';
                        ea3 <= S5;

                    when S5 =>     -- nao faz nada       -- pulso de mem_done

                        mem_done <= '1';
                        --start_tx <= '1';
                        --serial_data_in <= mem_out;
                        data <= (others => 'Z');
                        ea3 <= s5;

                    when s7 =>

                         a15 <= byte2 (0);
                         a16 <= byte2 (1);
                         ea3 <= s8;

                    when s8 =>
                        clk_mem <= '1';
                        ea3 <= s9;

                    when s9 =>
                        clk_mem <='0';
                        ea3 <= s9;

                end case;

            end if;

        end process;



END base_placa_arch;
```

```
-- Modulo de recepção RS-232 para a placa protótipo
-- Marcelo Sarmento - 05/2001
-- File : serial.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;


entity serial is
port
(
        -- general control
        clk                     : in  std_logic;      -- clock
        init_rst                : in  std_logic;      -- reset
        -- paralel interface (processor)
        data_in_232                     : in  std_logic_vector(7 downto 0);
        data_out_232            : out std_logic_vector(7 downto 0);
        -- serial interface(outdoor)
        rx_232                  : in  std_logic;
        tx_232                  : out std_logic;
        --
        tx_completed   : out std_logic;      -- saida indicando a finalizacao da
transmissao de 1 byte
        rx_completed   : out std_logic       -- saida indicando a finalizacao da recepcao
de 1 byte
);
end serial;

architecture beh of serial is

--      BAUD RATE SIGNALS
        signal  clk_counter    : STD_LOGIC_VECTOR(9 downto 0);
        signal  clk_serial     : std_logic;                  -- defined baud rate
         signal  tx_clk_counter       : STD_LOGIC_VECTOR(9 downto 0);
        signal  tx_clk_serial  : std_logic;                  -- defined baud rate

--      AUX BUFFER
        signal  bit_tx         : std_logic;   -- rx aux

        signal  byte_tx,byte_rx              : std_logic_vector(7 downto 0);

--      FSM SIGNALS
        type type_state is (S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,SA);
        type type_state_ctl is (c0,c01,c02,c1,c2,c3,c4,c5,c6,c7,c8,c9,ca,cb);

        signal  EST : Type_STATE;

        signal  ESR : Type_STATE;

        signal  start_Ctl : type_state_ctl;
         signal startbit : std_logic;
         signal bitclock : std_logic_vector (4 downto 0);
         signal start_check : std_logic_vector (3 downto 0);

begin

        process(clk, start_ctl,init_rst)
        begin
        if (clk'event and clk='1') then

        if start_ctl = c01  or init_rst = '0' then
               clk_counter <= (others => '0');
               bitclock <= (others => '0');
          elsif clk_counter = "1000001001" then --clk_divider then
                   bitclock <= bitclock + 1;
                   if bitclock = "10010" then
                      bitclock <= (others => '0');
                   end if;

                   clk_counter <= (others => '0');
               else
                  clk_counter <= clk_counter + 1;

               end if;
        end if;
        end process;
```

```
        clk_serial <= bitclock (0);

        --clk_out <= clk_serial;
        --clk_divider <= "0100000100";      --- 19200 bps com clk_divider = 1042/2



--@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
--              RECEPTION
--@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

        process (init_rst,clk,rx_232,bitclock)

        begin

        if init_rst = '0' then
           startbit <= '1';
           start_ctl <= c0;
           rx_completed <= '1';
           byte_rx <= (others => '0');
           start_check <= (others => '0');
        elsif clk'event and clk='0' then

           case start_ctl is

              when c0 =>
                 rx_completed <= '1';
                 startbit <= '1';
                 if rx_232 = '0' then
                    start_check <= (others => '0');
                    start_ctl <= c01;
                 else
                    start_ctl <= c0;
                 end if;

              when c01 =>
                 if rx_232 = '0' then
                    start_check <= start_check + 1;
                    if start_check = "1000" then
                    start_ctl <= c02;
                    end if;
                 else
                    start_ctl <= c0;
                 end if;

              when c02 =>

                 if rx_232 = '0' then
                    start_ctl <= c1;
                 else
                    start_ctl <= c0;
                 end if;

              when c1 =>
              startbit <='0';
              if bitclock = "00001" then
                  start_ctl <= c2; -- le o start_bit
              else
                  start_ctl <= c1;
              end if;

              when c2 =>
              if bitclock = "00011" then
                  byte_rx(0) <= rx_232;
                  start_ctl <= c3; -- le o bit 0

              else
                  start_ctl <= c2;
              end if;

              when c3 =>
              if bitclock = "00101" then
                  byte_rx(1) <= rx_232;
                  start_ctl <= c4; -- le o bit 1
```

```
                else
                    start_ctl <= c3;
                end if;
                when c4 =>
                if bitclock = "00111" then
                    byte_rx(2) <= rx_232;
                    start_ctl <= c5; -- le o bit 2
                else
                    start_ctl <= c4;
                end if;
                when c5 =>
                if bitclock = "01001" then
                    byte_rx(3) <= rx_232;
                    start_ctl <= c6; -- le o bit 3
                else
                    start_ctl <= c5;
                end if;
                when c6 =>
                if bitclock = "01011" then
                    byte_rx(4) <= rx_232;
                    start_ctl <= c7; -- le o bit 4
                else
                    start_ctl <= c6;
                end if;
                when c7 =>
                if bitclock = "01101" then
                    byte_rx(5) <= rx_232;
                    start_ctl <= c8; -- le o bit 5
                else
                    start_ctl <= c7;
                end if;
                when c8 =>
                if bitclock = "01111" then
                    byte_rx(6) <= rx_232;
                    start_ctl <= c9; -- le o bit 6
                else
                    start_ctl <= c8;
                end if;
                when c9 =>
                if bitclock = "10001" then
                    byte_rx(7) <= rx_232;
                    start_ctl <= ca; -- le o bit 7
                else
                    start_ctl <= c9;

                end if;
                when ca =>
                if bitclock = "10010" then
                        start_ctl <= cb;
                        data_out_232 <= byte_rx;
                        rx_completed <= '0';
                else
                    start_ctl <= ca;
                end if;
                when cb =>
                    start_ctl <= c0;

           end case;

        end if;
        end process;


end beh;
```

```
-- Gerador VGA para a placa protótipo
-- Adaptado do código da placa XS-40
-- Marcelo Sarmento 06-2001
-- File : vga.vhd

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY vga_generator IS
PORT
 (
 clk: IN STD_LOGIC; -- VGA dot clock (20MHZ)
 init_rst: IN STD_LOGIC; -- asynchronous reset
 hsync: OUT STD_LOGIC; -- horizontal (line) sync
 vsync: OUT STD_LOGIC; -- vertical (frame) sync
 vga_r: out std_logic;
 vga_g: out std_logic;
 vga_b: out std_logic;
 extra: out std_logic;
 address: OUT STD_LOGIC_VECTOR(14 DOWNTO 0);-- address into video RAM
 data: IN STD_LOGIC_VECTOR (7 DOWNTO 0); -- data from video RAM
 oe: OUT STD_LOGIC; -- video RAM output enable
 we: OUT STD_LOGIC; -- video RAM write enable
 a15: out std_logic;
 a16: out std_logic;
 reprog: out std_logic;
 clk_mem: out std_logic
 );
 END vga_generator;

 ARCHITECTURE vga_generator_arch OF vga_generator IS
 SIGNAL hcnt: STD_LOGIC_VECTOR(9 DOWNTO 0); -- horizontal pixel counter
 SIGNAL vcnt: STD_LOGIC_VECTOR(9 DOWNTO 0); -- vertical line counter
 SIGNAL pixrg: STD_LOGIC_VECTOR(7 DOWNTO 0); -- byte register for 2 pix
 SIGNAL blank: STD_LOGIC; -- video blanking signal
 SIGNAL pblank: STD_LOGIC; -- pipelined video blanking signal
 SIGNAL int_hsync: STD_LOGIC; -- internal horizontal sync.

 -- maquina de estados para o pixel generator

 TYPE statetype is (S1,S2,S3);
 SIGNAL EA : statetype;
 type estados_banco is (s0,s1,s2,s3);
 signal controle : estados_banco;

 BEGIN

 inc_horiz_pixel_counter:
 PROCESS(clk,init_rst)
 BEGIN
 IF init_rst='0' THEN -- init_rst asynchronously clears pixel counter
 hcnt <= "0000000000";
 ELSIF (clk'EVENT AND clk='1') THEN
 IF hcnt<630 THEN -- pixel counter init_rsts after 580 pixels
 hcnt <= hcnt + 1;
 ELSE
 hcnt <= "0000000000";
 END IF;
 END IF;
 END PROCESS;

 inc_vert_line_counter:
 PROCESS(int_hsync,init_rst)
 BEGIN
 IF init_rst='0' THEN -- init_rst asynchronously clears line counter
 vcnt <= "0000000000";
 ELSIF (int_hsync'EVENT AND int_hsync='1') THEN
 IF vcnt<527 THEN -- vert. line counter rolls-over after 528 lines
 vcnt <= vcnt + 1;
 ELSE
 vcnt <= "0000000000";
 END IF;
 END IF;
 END PROCESS;

 generate_horiz_sync:
```

```
 PROCESS(clk,init_rst)
 BEGIN
 IF init_rst='0' THEN -- reset asynchronously inactivates horiz sync
 int_hsync <= '1';
 ELSIF (clk'EVENT AND clk='1') THEN
 IF (hcnt>=540 AND hcnt<625) THEN
 -- horiz. sync is low in this interval to signal start of new line
 int_hsync <= '0';
 ELSE
 int_hsync <= '1';
 END IF;
 END IF;
 hsync <= int_hsync; -- output the horizontal sync signal
 END PROCESS;

generate_vert_sync:
 PROCESS(int_hsync,init_rst)
 BEGIN
 IF init_rst='0' THEN -- init_rst asynchronously inactivates vertical sync
 vsync <= '1';
 -- vertical sync is recomputed at the end of every line of pixels
 ELSIF (int_hsync'EVENT AND int_hsync='1') THEN
 IF (vcnt>=490 AND vcnt<492) THEN
 -- vert. sync is low in this interval to signal start of new frame
 vsync <= '0';
 ELSE
 vsync <= '1';
 END IF;
 END IF;
 END PROCESS;

 -- blank video outside of visible region: (0,0) -> (255,479)
 blank <= '1' WHEN (hcnt <=16 or hcnt>=512 OR vcnt>=480) ELSE '0';
 -- store the blanking signal for use in the next pipeline stage
 pipeline_blank:
 PROCESS(clk,init_rst)
 BEGIN
 IF init_rst='0' THEN
 pblank <= '0';
 ELSIF (clk'EVENT AND clk='1') THEN
 pblank <= blank;
 END IF;
 END PROCESS;

 -- video RAM control signals
 oe <= '0'; -- enable the RAM
 we <= '1'; -- disable writing to the RAM
 reprog <= '0';



sel_next_bank :

process (init_rst,clk)

begin


if init_rst = '0' then
  a15 <= '1' ; -- seleciona o banco 3 (onde a imagem esta armazenada)
  a16 <= '1' ;
  controle <= s0;

elsif clk'event and clk = '1' then


   case controle is

        when s0 => clk_mem <= '0'; controle <= s1; a15 <= '1'; a16 <='1';
        when s1 => clk_mem <= '1'; controle <= s2;
        when s2 => clk_mem <= '0'; controle <= s3;
        when s3 => null;
   end case;
end if;

end process;
```

```
ram_addr_selection :
process (clk,init_rst)
begin

if  init_rst = '0' then
  address <= (others => '0');
 elsif (clk'event and clk = '1') then
        address <= vcnt(6 downto 0)  & hcnt (8 downto 1);

end if;
end process;



update_pixel_register:
PROCESS(clk,init_rst)
BEGIN
IF init_rst='0' THEN -- clear the pixel register on init_rst
pixrg <= "00000000";
--pixel clock controls changes in pixel register
ELSIF (clk'EVENT AND clk='1') THEN

 -- hcnt = 2 , ram e copiado para o pixrg



IF hcnt(1)= '1' THEN
pixrg <= data; -- load data from RAM
END IF;
end if;

END PROCESS;

 -- the color mapper translates each 2-bit pixel into a 6-bit
 -- color value. When the video signal is blanked, the color
 -- is forced to zero (black).



map_pixel_to_rgb:
PROCESS(clk,init_rst)
BEGIN
 IF init_rst='0' THEN -- blank the video on init_rst
 vga_r <= '0';
 vga_g <= '0';
 vga_b <= '0';

 EA <= S1;

 ELSIF (clk'EVENT AND clk='1') THEN -- update the color every clock

   if pblank = '0' then
    if hcnt (0) = '1' then

              vga_r <= pixrg (0);
              vga_g <= pixrg (1);
              vga_b <= pixrg (2);
              extra <= pixrg (3);
    else
              vga_r <= pixrg (4);
              vga_g <= pixrg (5);
              vga_b <= pixrg (6);
              extra <= pixrg (7);
    end if;
   else
       vga_r <= '0';
       vga_g <= '0';
       vga_b <= '0';

   end if;
 end if;
 END PROCESS;
END;
```