SÍNTESE EVOLUCIONÁRIA DE CIRCUITOS DIGITAIS EMPREGANDO FPGA´S

Aluno: Rogério Cortez B. L. Póvoa
Orientador: Marco Aurélio C. Pacheco

1. Introdução

1.1. Motivação

O hardware evolutivo (*Evolvable Hardware* – EHW) é projetado para se adaptar às mudanças na sua estrutura física, no ambiente ou no seu objetivo, devido a sua capacidade de reconfiguração dinâmica. Desta forma este hardware se difere do hardware tradicional cuja estrutura é fixa [1]. EHW foi o resultado dos estudos da área da Eletrônica Evolutiva, que surgiu devido ao grande interesse de pesquisadores em otimizar e/ou sintetizar circuitos eletrônicos por algoritmos evolutivos [2].

Algoritmos evolutivos são divididos em classes e foram desenvolvidos a partir de observações de processos adaptativos e evolutivos da natureza, tendo como principais bases a seleção natural, a recombinação do material genético e a mutação [3]. Dentre as principais classes estão os algoritmos genéticos [4] [5], a programação genética [6], a evolução diferencial [7], os algoritmos culturais [8] [9] e a programação evolutiva [10].

A Eletrônica Evolutiva é classificada de acordo com o tipo de projeto, a natureza do projeto e a plataforma evolutiva. O tipo de projeto determina se o mesmo será utilizado para otimização ou síntese de circuitos, a natureza do projeto define se as aplicações serão analógicas ou digitais e a plataforma evolutiva abrange as classificações intrísica e extrínseca para uma aplicação.

Uma aplicação extrínseca consiste em avaliar circuitos através de simuladores, já na intrínseca a avaliação é realizada através do uso de circuitos integrados reconfiguráveis, esta metodologia de evolução de circuitos é denominada *Evolvable Hardware* [2], como já mencionada anteriormente.

Projetos de Hardware Evolucionário para evolução intrínseca consistem na utilização de um hardware programável, como FPAA, FPGA ou FPTA, e uma técnica de algoritmo evolutivo, como o Algoritmo Genético. O algoritmo evolutivo é compilado em um software e uma configuração de bits é gerada para reconfigurar o hardware programável. Trabalhos como [11] e [12], utilizando a plataforma reconfigurável *Field Programmable Transistor Array* (FPTA), são realizados na NASA (*National Aeronautics and Space Administration*) e são exemplos de projetos de Hardware Evolucionário.

2. FPGA

2.1. Conceito de FPGA

Field Programmable Gate Array (FPGA) é uma plataforma reconfigurável que contém uma matriz de células lógicas genéricas e chaves programáveis. Estas células lógicas são configuradas para desempenhar uma função simples, enquanto as chaves programáveis interconectam as células lógicas segundo uma configuração.

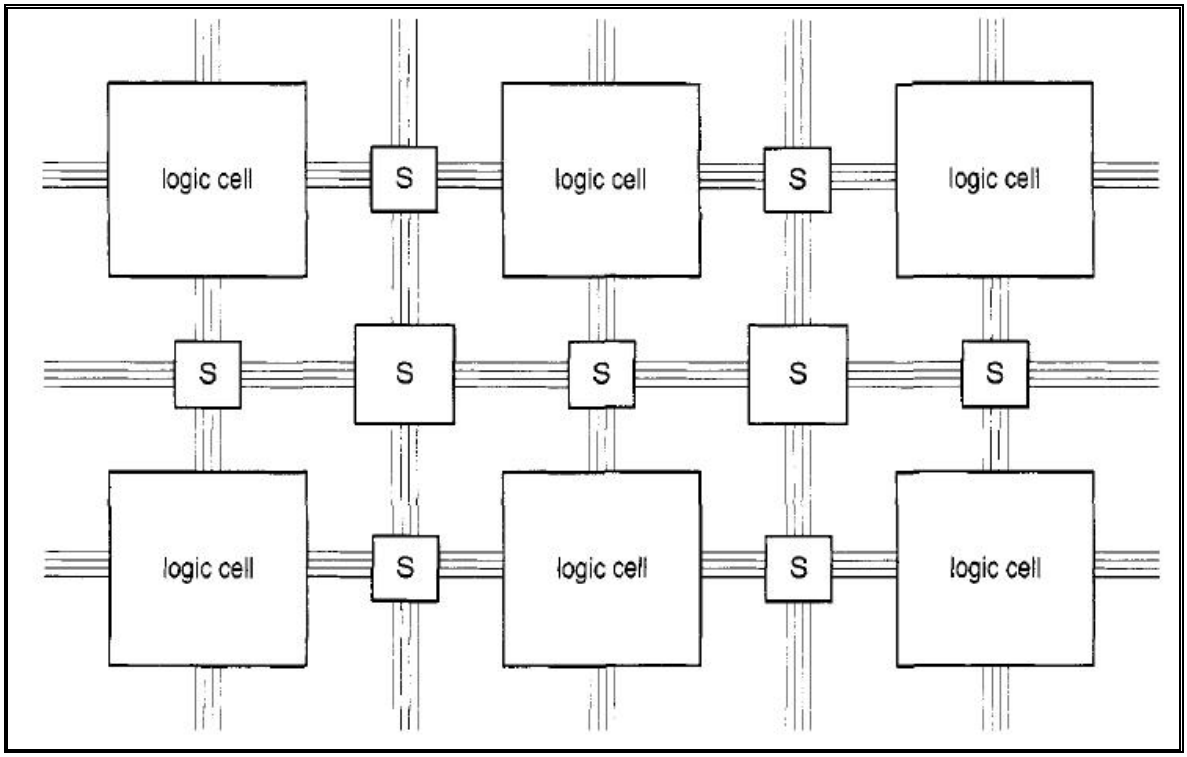

FIGURA 2.1. Estrutura Conceitual de uma FPGA.

Na implementação de um projeto em FPGA, ocorre a especificação da função de cada célula lógica além da configuração da conexão de cada chave. Com a etapa de síntese do projeto concluída, é enviado um arquivo de configuração para FPGA por um cabo. Este processo é feito "em campo" e não "em fábrica", sendo por este motivo atribuído o nome *Field Programmable* à placa. [13]

2.2. LUT-based logic cell

Normalmente uma célula lógica é constituída de uma LUT (*look-up table*), ou tabela de busca, que é um circuito combinacional com um flip-flop tipo D. Uma n-LUT pode ser considerada como uma memória 2n por 1 e se configurada de forma correta pode funcionar como uma função combinacional de n entradas. A figura 2.2 é um exemplo de célula lógica baseada em *look-up table* de 3 entradas. [13]

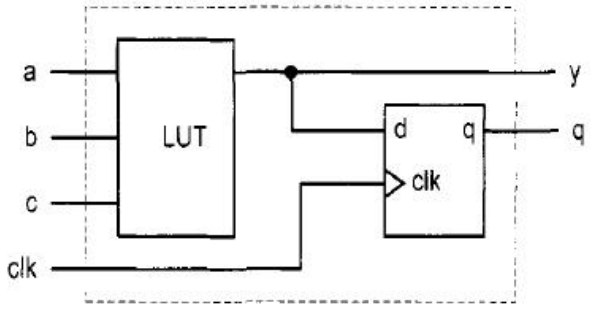

Diagrama Conceitual

a b c	y
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1

LUT = a ⊕ b ⊕ c

FIGURA 2.2. Célula lógica baseada em *look-up table* de 3 entradas.

2.3. Projeto FPGA

O projeto e programação de uma FPGA consistem em seis etapas: definição do comportamento da FPGA, geração de uma *netlist*, processo de roteamento, validação do mapeamento, geração do arquivo binário e configuração da FPGA.

A definição do comportamento da FPGA é feita através da geração de um projeto esquemático (gráfico) por uma linguagem de descrição de hardware (HDL – *Hardware Description Language*). As principais linguagens utilizadas são VHDL e Verilog. Este projeto utilizará a linguagem VHDL (*Very-high-speed integrated circuit* HDL).

A netlist é gerada por uma ferramenta de EDA (Eletronic Design Automation). Esta netlist descreve a conectividade de um circuito, como componentes, pinos e portas. O processo de roteamento é o ajuste da *netlist* à arquitetura da FPGA.

O mapeamento é validado através de análises temporais e simulações, a partir desta validação um arquivo binário é gerado e transferido para a FPGA para configurar a mesma. Esta transferência é feita através de uma interface serial utilizando o protocolo JTAG. [13]

3. Algoritmos Genéticos

3.1. Conceito

Técnica de busca e otimização paralela, desenvolvida por John Holland na Universidade de Michigan, que se baseia na seleção natural proposta por Charles Darwin, onde uma solução potencial se dá através de uma estrutura semelhante a um cromossomo. Este cromossomo é submetido a três operadores, seleção, *crossover* e mutação, além de uma função de avaliação de aptidão. [14]

3.2. Estrutura Básica

Segue abaixo, a estrutura básica de um Algoritmo Genético simples.

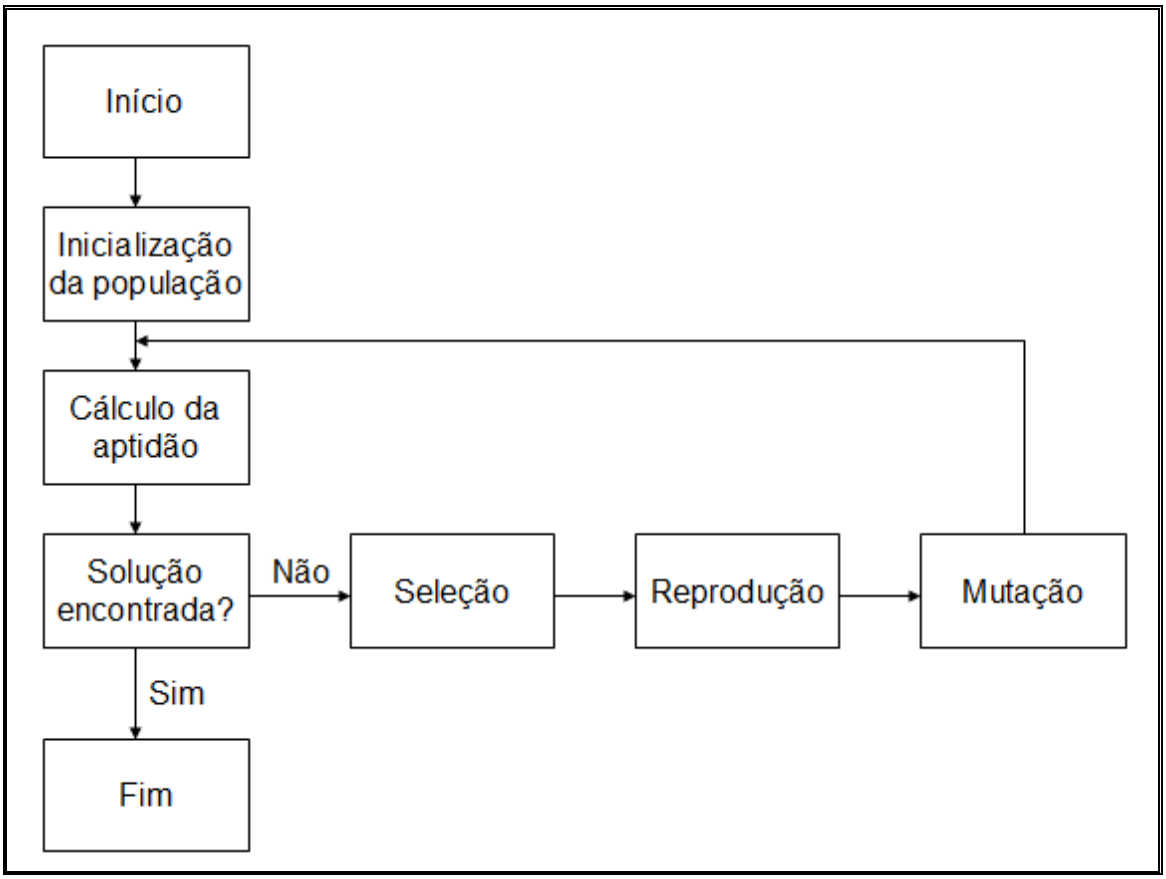

FIGURA 3.1. Estrutura básica de um Algoritmo Genético simples.

Para determinar a solução de um problema, uma população aleatória de possíveis soluções é gerada. Os cromossomos pertencentes a esta população são avaliados quanto aos seus desempenhos e ordenados quanto a sua aptidão. Após este procedimento, é avaliado se a solução para o problema foi encontrada, caso tenha sido encontrada o procedimento é encerrado, caso contrário os procedimentos de seleção, reprodução e mutação serão aplicados para gerar novos indivíduos na população.

Por esta técnica estar baseada na teoria da seleção natural, a seleção é feita de forma a privilegiar a sobrevivência dos indivíduos mais aptos. Privilégio concedido pelo fato da aptidão dos indivíduos estar diretamente relacionada à probabilidade de serem selecionado e assim contribuírem com uma nova geração de indivíduos.

Após o procedimento de seleção, cromossomos são selecionados aleatoriamente em pares para que o *crossover* seja realizado. A aplicação de tal evento é probabilística, ou seja, há a possibilidade de ocorrer ou não. Caso ocorra, partes do material genético (número binário) de um indivíduo é trocado com o outro indivíduo, de maneira que os dois descendentes possuam material genético dos dois indivíduos progenitores (figura 3.2). Caso não ocorra, os códigos dos cromossomos são preservados.

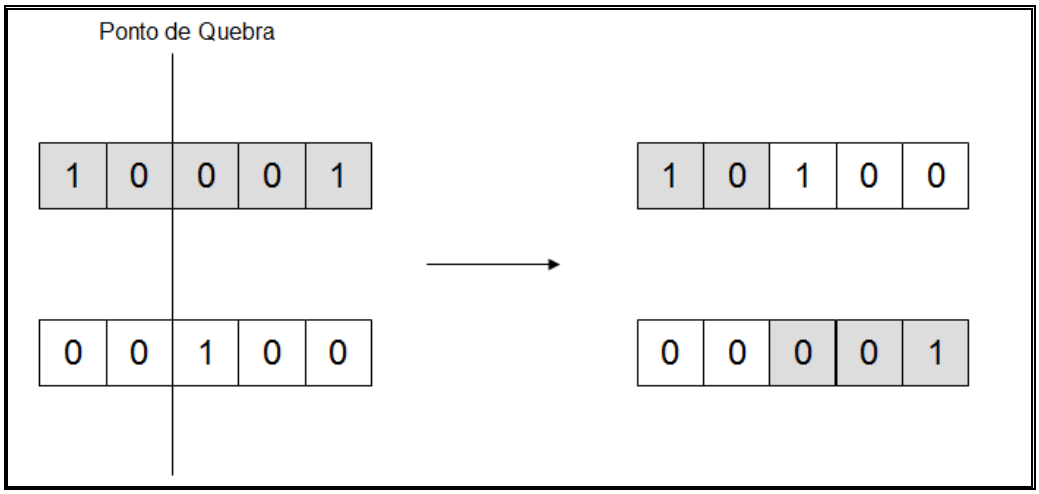

FIGURA 3.2. *Crossover*.

Cada bit dos novos indivíduos é submetido ao procedimento de mutação, que ocorre de forma probabilística. Ao ocorrer uma mutação o bit é alterado, como ilustrado na figura 3.3. Após a mutação, os indivíduos resultantes passam a fazer parte da população, sendo então reiniciado o procedimento, conforme a figura 3.1. [14]

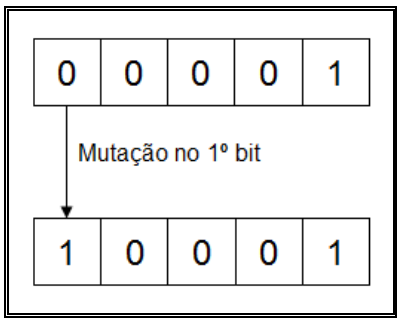

FIGURA 3.3. Mutação.

4. Objetivo

Este projeto tem com objetivo estudar projetos de Hardware Evolucionário utilizando a plataforma reconfigurável *Field Programmable Gate Array* (FPGA). Projetos deste tipo requerem uma constante comunicação entre o software que utiliza o algoritmo evolutivo e o hardware programável, além do uso de uma FPGA requerer passos como programação, configuração de portas e envio de dados. Logo será estudada a possibilidade de transferências de dados para a placa, de forma a utilizá-la de maneira eficiente.

O projeto também tem como objetivo implantar uma aplicação de EHW utilizando uma FPGA. Para esta etapa final, serão avaliados os recursos disponíveis e os conhecimentos adquiridos na área.

4.1. Recurso

Para a realização do projeto, o modelo da FPGA escolhido foi o *Spartan-3E*, do fabricante XILINX. Que está disponível no pacote *Spartan-3E Starter Kit*.

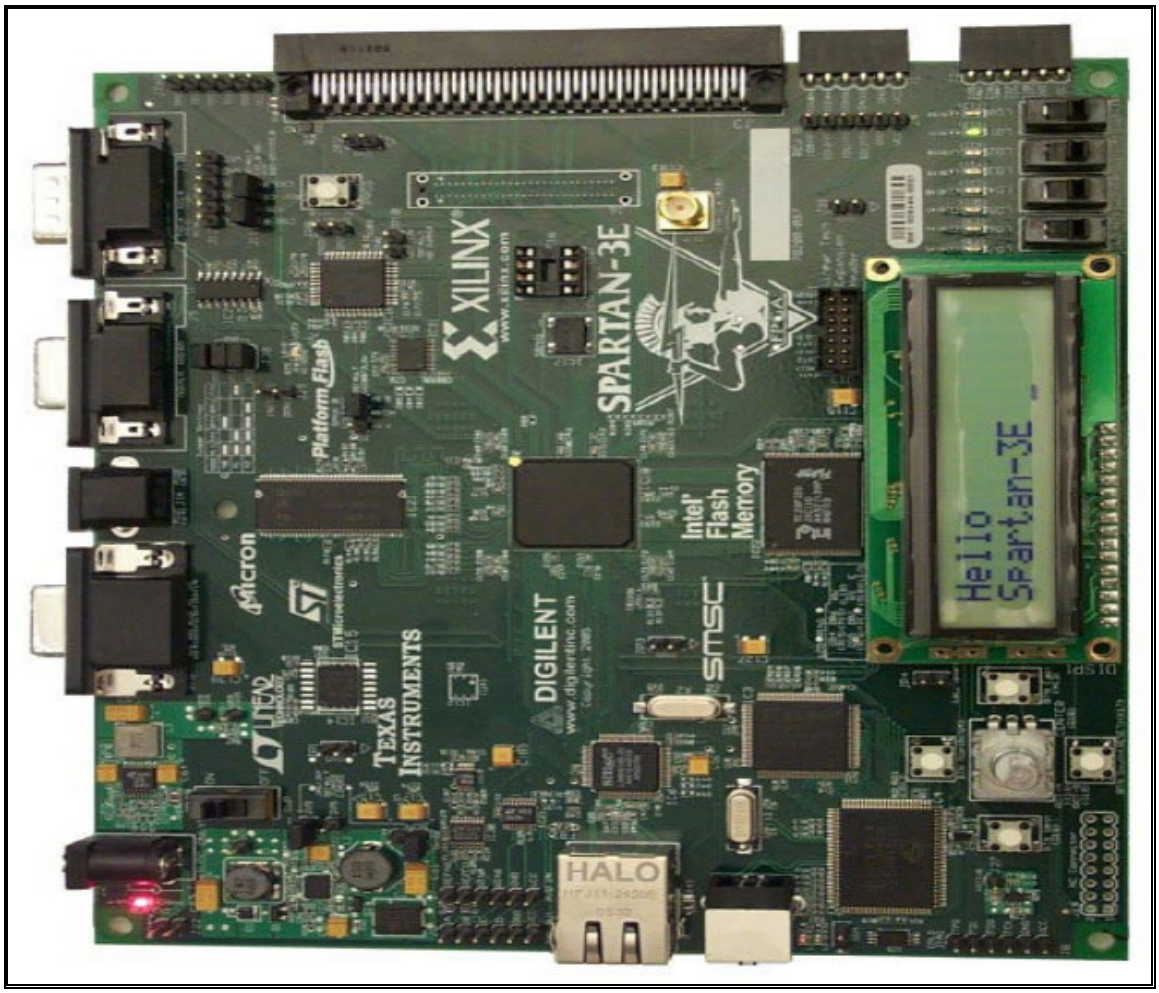

FIGURA 4.1. ***Spartan-3E Starter Kit*****.**

5. Atividades Realizadas

5.1. Projeto FPGA e Pesquisa

O início do projeto foi determinado pelo entendimento da estrutura conceitual e do projeto de uma FPGA, implementando a metodologia de execução de um projeto de FPGA, descrita no capítulo 2 deste projeto. A lembrar, as seis etapas de programação e projeto de uma FPGA são: definição de comportamento da FPGA, geração de uma *netlist*, processo de roteamento, validação do mapeamento, geração do arquivo binário e configuração da FPGA.

Após a compreensão e a implementação de um projeto estático, se fez necessário o estudo de Algoritmos Genéticos, conforme o capítulo 3. Desta forma foi constatada a importância de se utilizar este método para a evolução de códigos e para uma futura programação evolucionária.

Para o desenvolvimento de projetos de Hardware Evolucionário utilizando FPGA's atuando em tempo hábil, foi necessário pesquisar possíveis formas de evoluir um projeto. A criação de um Algoritmo Genético durante a criação de um código VHDL como solução do problema, por mais trivial que se pode parecer, não satisfaz o requisito de utilização do projeto no dia a dia, em tempo satisfatório. Isto ocorre porque um projeto de uma FPGA necessita de um tempo relativamente grande para execução de suas etapas [13]. Além disso, o número de evoluções que seriam realizadas pelo Algoritmo Genético para programação da

plataforma reconfigurável seria grande, de forma a prolongar mais o tempo para encontrar uma configuração ótima.

5.2. Definição de Solução

Dentre as soluções de projetos relacionados a este assunto, a evolução do arquivo binário, gerado a partir da criação de um código VHDL e do mapeamento da FPGA, se apresentou como a mais adequada para o projeto. Pois para a evolução do arquivo binário só é necessário realizar a geração deste arquivo uma única vez, ou seja, as seis etapas de programação e projeto de uma FPGA serão executadas somente uma vez, sendo a partir deste momento a programação da FPGA evoluída por uma API (Application Programming Interface) chamada JBits (abordada no próximo capítulo).

6. JBits

6.1. Conceito

JBits é um conjunto de classes Java que fornece uma API (Application Programming Interface) para acessar um arquivo binário, desenvolvido para modelos de FPGA do fabricante Xinlinx. A interface permite a reconfiguração das células lógicas e conexões de dispositivos da FPGA. Desta forma, sua utilização permite a configuração dinâmica de circuitos. [15]

6.2. Funcionamento

Esta biblioteca de classes dá acesso completo a configuração de todos os dispositivos da plataforma reconfigurável e por se tratar de classes Java pré-compiladas, o resultado para configuração da FPGA não é um *bitstream* estático, mas sim um código executável que gera um *bitstream* para a placa. Este código é executado nos momentos de configuração, conforme a figura 7.1.

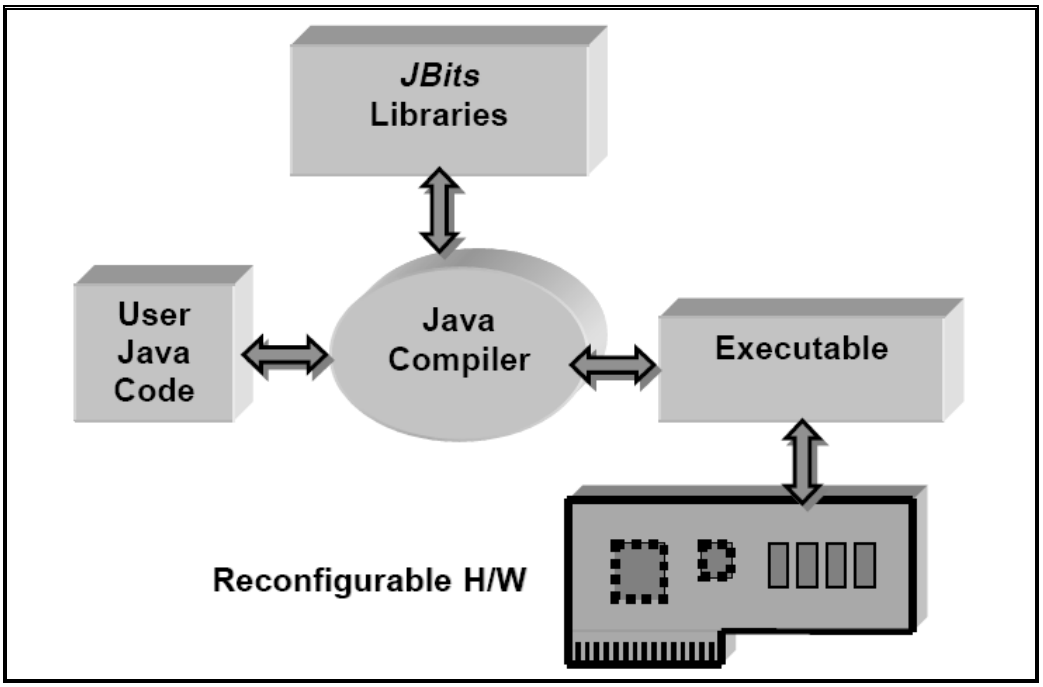

FIGURA 6.1. Modo de funcionamento do JBits.

A informação é compartilhada entre o processador que executa o código Java e a FPGA, o que difere de outros sistemas de computação reconfigurável onde a configuração da FPGA é completamente independente da interface do software. Esta diferença evita erros que são difíceis de encontrar e corrigir. Com um módulo único e integrado de software que faz tanto a configuração do circuito e gerenciamento do processador, a coerência entre o software e a FPGA é mantida de uma forma simples. [15]

6.3. Forma de utilização do JBits

A figura 7.2 ilustra a forma de utilização do JBits. A aplicação em Java para o usuário utiliza o JBits para configurar os dispositivos da FPGA. Cada chamada de função no nível da interface JBits faz uma ou mais chamadas para o nível de bits, onde um bit é alterado no *bitstream*. Ocorrendo então uma abstração na alteração de bits, evitando alterações diretas no arquivo binário, por parte do usuário.

Esta interface do nível de bit gerencia o fluxo contínuo de dados do dispositivo e fornece suporte para leitura e escrita *bitstreams* de e para arquivos. Além disso, a classe *Bitstream* pode ler dados do hardware e mapeá-lo para o formato de dados de um bitstream (*readback*), o que é necessário para uma reconfiguração dinâmica. A API JBits utiliza o software XHWIF para fazer o download e *readback* do hardware.

A biblioteca central é a coleção de classes Java que definem núcleos (*Macro cell*) geralmente parametrizáveis e realocáveis dentro de um dispositivo. São exemplos de núcleos: contadores, somadores e multiplicadores. [15]

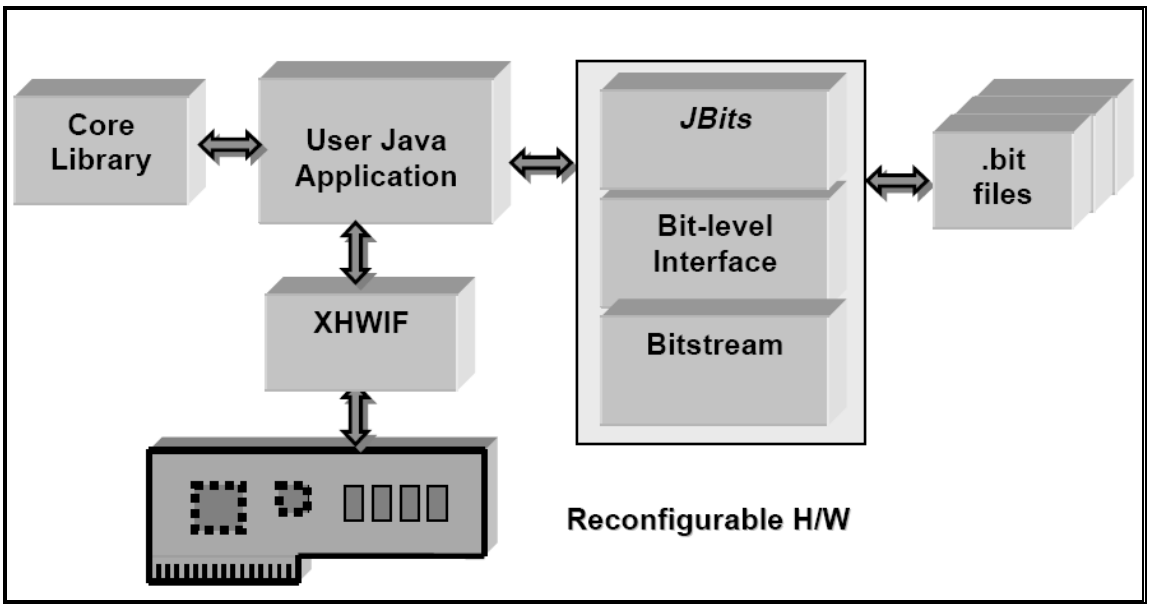

FIGURA 6.2. Forma de utilização do JBits.

```
/* Configure the F LUT of the Slice 0 of row, col to be XOR */
set(row, col, Slice0_FLUT, XOR);

/* Get the value of the Clock Input at row,col */
c = get(row, col, ClockInput);
```

FIGURA 6.3. Exemplo de código JBits para o modelo da FPGA Virtex.

6.4. Limitações do JBits

Uma das principais limitações do JBits é sua natureza manual, onde todos os comandos enviados para a FPGA devem ser explicitados no código fonte. Para facilitar a programação destes códigos, os núcleos (*Macro cell*) da FPGA são utilizados.

Outra limitação importante é a necessidade de um conhecimento avançado sobre a arquitetura da plataforma reconfigurável, conhecimento este, que nunca foi exigido dos usuários de FPGA's. Para solução deste problema é necessário estudo sobre as documentações de cada modelo de FPGA utilizado em um projeto. [15]

O fabricante desta interface, XINLINX, não garante o funcionamento correto das classes para modelos de FPGA's diferentes do modelo XC4000 ou da família Virtex (até a Virtex II). Sendo a ultima versão, JBits 3.0, desenvolvida especificamente para o modelo Virtex II.

6.5. *BoardScope*

Ferramenta gráfica para análise do funcionamento de FPGA's em qualquer placa de computação reconfigurável. Verifica o funcionamento do projeto e resultados produzidos por operações do hardware. Permite uma verificação mais detalhada e uma depuração durante a comunicação da FPGA com outros dispositivos.

O *BoardScope* utiliza a interface JBit para acessar recursos do *bitstream* da FPGA. Em seguida, utilizando o XHWIF, o *bitstream* é transferido para configurar a FPGA, ou o *readback* é realizado para analisá-lo.

Os estados dos Blocos Lógicos Configuráveis (CLB) são representados graficamente pelo BoardScope. A figura 7.4 mostra o StateView no BoardScope de uma placa com quatro FPGA's do modelo XC4028EX. Cada quadrado representa a CLB de uma FPGA, é colorido com a cor azul para representar um estado de baixa e verde para um estado de alta. Se os estados forem alterados as cores são alteradas indicando os novos estados. A representação possibilita a análise de todos as CLB's e as alterações de estados. [15]

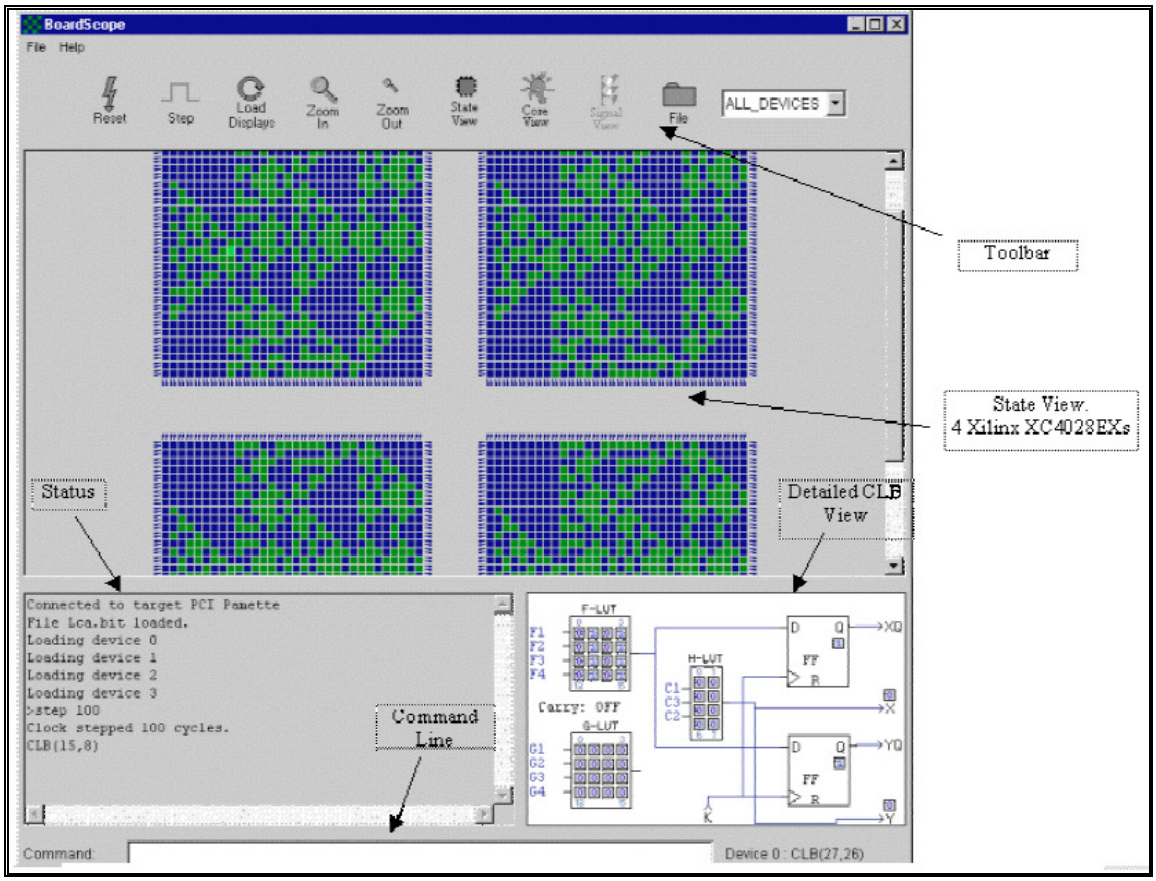

FIGURA 6.4. *StateView* no *BoardScope*.

6.6. XHWIF

É uma interface Java para comunicação com placas FPGA. Possui métodos para ler e escrever *bitstreams* das FPGA's e métodos para descrever os tipos e número de FPGA's contidas em uma placa. Também possui métodos para incrementar o *clock* e para leitura e escrita nas memórias da placa. Através deste software há uma padronização na comunicação entre os aplicativos e o hardware, permitindo comunicação de aplicativos com placas diferentes.

Outra vantagem desta interface é permitir que aplicativos como o *BoardScope* não se preocupem com controladores, barramentos e tipos de barramento (PCI, ISA e outros). [15]

7. Ferramentas Utilizadas

7.1. Hardware

Conforme descrito no capítulo 4, o modelo da FPGA utilizado neste projeto foi o *Spartan-3E*, do fabricante XILINX, disponível no pacote *Spartan-3E Starter Kit*.

7.1. Software

Xilinx® *Integrated Software Environment* (ISE®) Project Navigator – Software que permite a realização das seis etapas de programação e projeto de uma FPGA: definição de comportamento da FPGA, geração de uma *netlist*, processo de roteamento, validação do mapeamento, geração do arquivo binário e configuração da FPGA.

ModelSim XE III/Starter 6.4b – Simulador de códigos VHDL.

8. Implementação

8.1. Desenvolvimento Estático

Para a melhor compreensão do funcionamento de uma FPGA, projetos estáticos (ou não evolutivos) foram implementados. Dentre os projetos realizados na plataforma reconfigurável, o somador de dois bits foi selecionado para servir de base para testes futuros envolvendo EHW. A escolha deste projeto se justifica pela necessidade de avaliação de uma operação simples durante o desenvolvimento evolucionário e pelo fato de não se tratar de um operador lógico básico, como um AND ou um OR.

O projeto foi desenvolvido, seguindo as seis etapas de programação e projeto de uma FPGA. A definição de comportamento da FPGA, foi realizada através da criação dos códigos em VHDL, "add.vhd" e "testbench.vhd". O primeiro possui a entidade e arquitetura para execução da soma de dois números de dois bitst, enquanto o segundo possui a rotina de testes para simulação.

```
1  ----------------------------------------------------------------------------------
2  ------------------------------ SOMADOR DE 2 BITS ---------------------------------
3  -- ENG1133 - PROJ GRAD EM ENG COMPUTACAO II
4  -- Aluno: Rogério Cortez B. L. Póvoa
5  -- Orientador: Marco Aurélio M. Pacheco
6  ----------------------------------------------------------------------------------
7  library IEEE;
8  use IEEE.STD_LOGIC_1164.ALL;
9  use IEEE.numeric_std.all;
10
11 -- Entidade do Programa
12 entity add is
13   generic(N: integer := 2);  -- define o número de bits da soma
14   port(
15     entrada0, entrada1: in std_logic_vector(N-1 downto 0);
16     carry: out std_logic;
17     soma: out std_logic_vector(N-1 downto 0)
18     );
19 end add;
20 -- Fim da Entidade
21
22 --Arquitetura ArchAdd
23 architecture ArchAdd of add is
24   signal entradaExtendida0, entradaExtendida1, somaExtendida: unsigned (N downto 0);
25 begin
26   entradaExtendida0 <= unsigned('0' & entrada0);
27   entradaExtendida1 <= unsigned('0' & entrada1);
28   somaExtendida <= entradaExtendida0 + entradaExtendida1;
29   soma <= std_logic_vector(somaExtendida(N-1 downto 0));
30   carry <= somaExtendida(N);
31 end ArchAdd;
32 -- Fim da Arquitetura ArchAdd
33
```

FIGURA 8.1. Código em VHDL do somador de dois bits (add.vhd).

```
1  ---------------------------------------------------------------------------------
2  ---------------------------- TEST BENCH DO SOMADOR -------------------------------
3  -- ENG1133 - PROJ GRAD EM ENG COMPUTACAO II
4  -- Aluno: Rogério Cortez B. L. Póvoa
5  -- Orientador: Marco Aurélio M. Pacheco
6  ---------------------------------------------------------------------------------
7  library IEEE;
8  use IEEE.STD_LOGIC_1164.ALL;
9
10 -- Entidade do Programa
11 entity testbench is
12 end testbench;
13 -- Fim da Entidade
14
15 --Arquitetura ArchTestBench
16 architecture ArchTestBench of testbench is
17   signal testIn0, testIn1, testOut: std_logic_vector(1 downto 0);
18   signal carryOut: std_logic;
19 begin
20   uut: entity work.add(ArchAdd)
21   port map (entrada0 => testIn0, entrada1 => testIn1, carry => carryOut, soma => testOut);
22   process
23   begin
24   -- primeiro teste
25   testIn0 <= "00";
26   testIn1 <= "00";
27   --wait for 200 ns;
28   -- segundo teste
29   testIn0 <= "00";
30   testIn1 <= "01";
31   wait for 200 ns;
32   -- terceiro teste
33   testIn0 <= "00";
34   testIn1 <= "10";
35   wait for 200 ns;
```

FIGURA 8.2.a. Código em VHDL do testbench (testbench.vhd).

```
36     -- quarto teste
37     testIn0 <= "00";
38     testIn1 <= "11";
39     wait for 200 ns;
40     -- quinto teste
41     testIn0 <= "01";
42     testIn1 <= "00";
43     wait for 200 ns;
44     -- sexto teste
45     testIn0 <= "01";
46     testIn1 <= "01";
47     wait for 200 ns;
48     -- sétimo teste
49     testIn0 <= "01";
50     testIn1 <= "10";
51     wait for 200 ns;
52     -- oitavo teste
53     testIn0 <= "01";
54     testIn1 <= "11";
55     wait for 200 ns;
56     -- nono teste
57     testIn0 <= "10";
58     testIn1 <= "00";
59     wait for 200 ns;
60     -- décimo teste
61     testIn0 <= "10";
62     testIn1 <= "01";
63     wait for 200 ns;
64     -- décimo primeiro teste
65     testIn0 <= "10";
66     testIn1 <= "10";
67     wait for 200 ns;
68     -- décimo segundo teste
69     testIn0 <= "10";
70     testIn1 <= "11";
71     wait for 200 ns;
```

FIGURA 8.2.b. Código em VHDL do testbench (testbench.vhd).

```
72      -- décimo terceiro teste
73      testIn0 <= "11";
74      testIn1 <= "00";
75      wait for 200 ns;
76      -- décimo quarto teste
77      testIn0 <= "11";
78      testIn1 <= "01";
79      wait for 200 ns;
80      -- décimo quinto teste
81      testIn0 <= "11";
82      testIn1 <= "10";
83      wait for 200 ns;
84      -- décimo sexto teste
85      testIn0 <= "11";
86      testIn1 <= "11";
87      wait for 200 ns;
88      -- Fim do Teste
89      assert false
90        report "Fim da Simulação"
91       severity failure;
92      end process;
93
94  end ArchTestBench;
95  -- Fim da Arquitetura ArchTestBench
```

FIGURA 8.2.c. Código em VHDL do *testbench* (testbench.vhd).

Após a criação dos códigos, através do *ModelSim*, os mesmos foram testados e simulados, conforme o resultado abaixo.

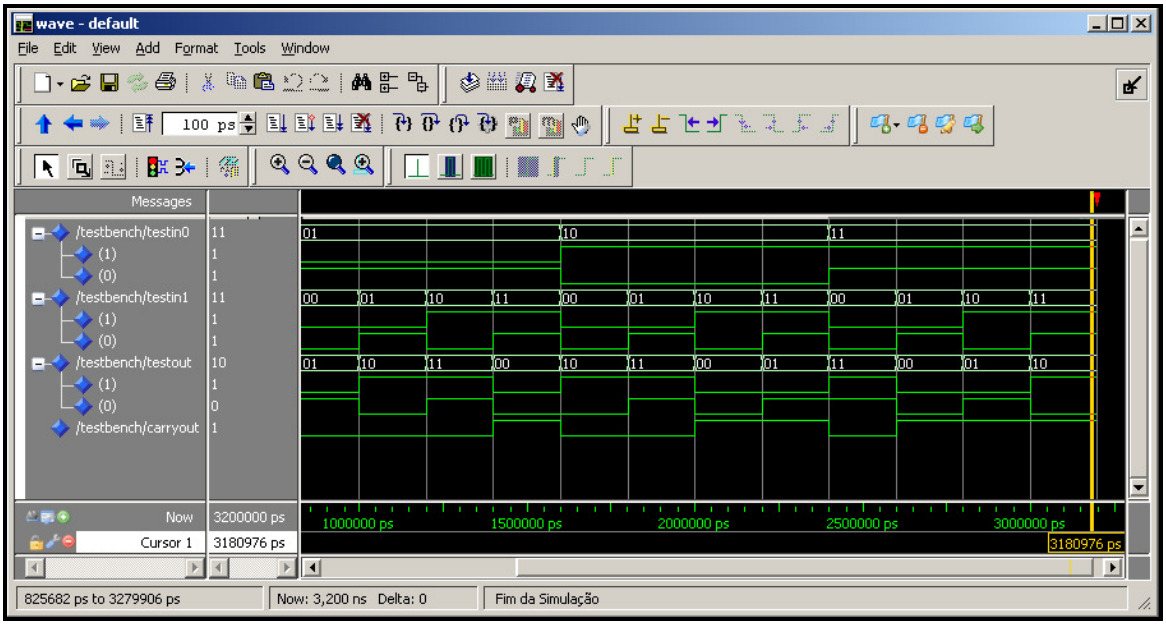

FIGURA 8.3. Simulação do somador de dois bits.

Com os códigos gerados e simulados, foram gerados dois esquemas do circuito lógico para verificação do mesmo afim de avaliar entradas, saídas e dispositivos intermediários do circuito proposto pelo *Xilinx® Integrated Software Environment (ISE®) Project Navigator*.

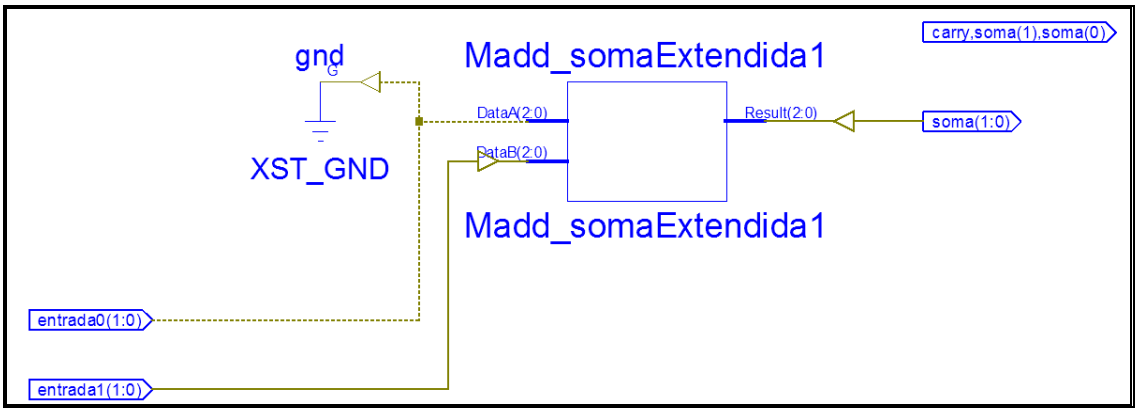

FIGURA 8.4.a. Esquema do circuito lógico.

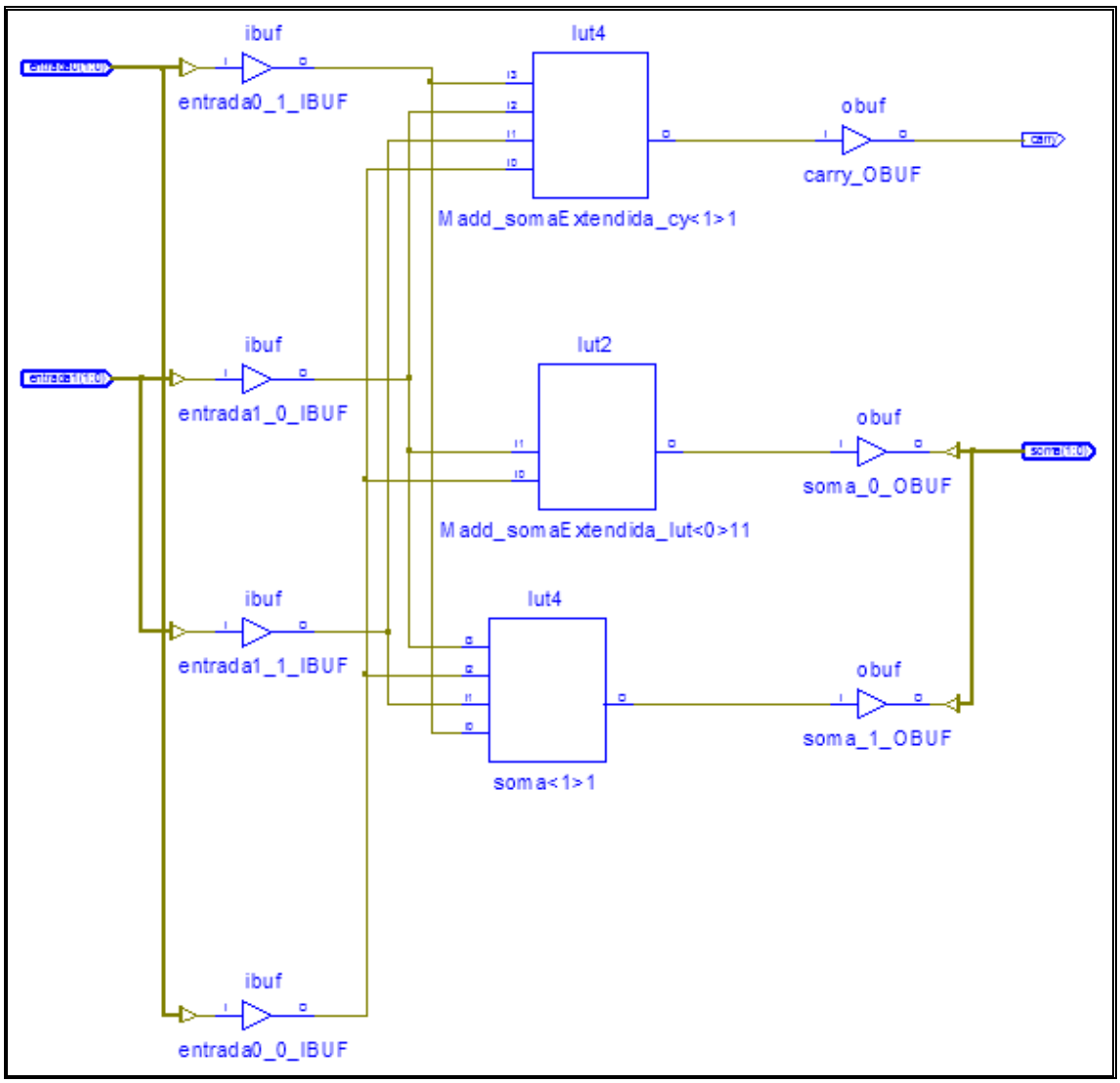

FIGURA 8.4.b. Esquema do circuito lógico.

Em seguida a *netlist* foi gerada descrevendo as portas de entrada e saída do circuito. O processo de roteamento e mapeamento foram realizados pelo próprio software, gerando assim o *bitstream* para configuração da FPGA.

8.2. Desenvolvimento Evolucionário

Esta etapa do projeto não foi implementada devido as dificuldades encontradas para a definição da solução para o desenvolvimento de projetos de Hardware Evolucionário utilizando FPGA's. Em contra partida uma pesquisa sobre soluções existentes nesta área foi realizada.

Como já descrito na definição de solução das atividades realizadas, capítulo 6, a solução mais adequada para este projeto foi evoluir arquivos binários, gerados a partir da criação de um código VHDL e do mapeamento da FPGA. Esta evolução é feita através de uma API chamada JBits.

Este projeto propõe a trabalhos futuros a utilização desta API para a evolução de *bitstrems* e para auxiliar estes trabalhos realiza uma descrição sobre o JBits no capítulo 7. É necessário lembrar que para a utilização desta interface, o usuário deverá conhecer a arquitetura das FPGA's a serem utilizadas e avaliar se as classes em Java oferecidas atendem a arquitetura da plataforma reconfigurável. Caso não sejam utilizados modelos de FPGA's que o fabricante XILINX garante o funcionamento do JBits, deverão ser criadas classes que atendam as diferenças do modelo da FPGA.

Conclusão

Neste projeto foram realizadas pesquisas para a utilização de FPGA's em projetos de Hardware Evolucionário, avaliando projetos e trabalhos existentes, além de ferramentas disponíveis. Durante a pesquisa fez necessária a implementação de projetos estáticos de FPGA, para adquirir conhecimentos sobre a plataforma reconfigurável e verificar as condições para se utilizar a computação evolucionária. Estudos sobre Algoritmos Genéticos foram importantes para o aprendizado dos conceitos de *Evolvable Hardware*. A definição da utilização da ferramenta JBits exigiu um estudo sobre esta interface, onde percebeu-se as limitações do software. Com o fim do prazo do trabalho e os conhecimentos adquiridos, uma proposta para continuação do mesmo foi realizada de forma a orientar trabalhos futuros nesta área.

Referências Bibliográficas

1 – SANTINI, C. C. **DESENVOLVIMENTO DE UMA PLATAFORMA RECONFIGURÁVEL ANALÓGICA PARA A EVOLUÇÃO INTRÍSICA DE CIRCUITOS**. Tese de Mestrado, Departamento de Engenharia Elétrica, Pontifícia Universidade Católica, Rio de Janeiro, Brasil, 2001.

2 – ZEBULUM, R. S. **SÍNTESE DE CIRCUITOS ELETRÔNICOS POR COMPUTAÇÃO EVOLUTIVA**. Tese de Mestrado, Departamento de Engenharia Elétrica, Pontifícia Universidade Católica, Rio de Janeiro, Brasil, PP. 1-2,1999.

3 – RIDLEY, M. **EVOLUTION**: Second Edition, Blackwell Science, EUA, 1996.

4 – BACK, T.; FOGEL, D. B.; MICHALEWICZ, Z. **HANDBOOK OF EVOLUTIONARY COMPUTATION.** Institute of Physics Publishing, Bristol, NY, EUA 1997. 1.1.

5 – MICHALEWICZ, Z. **GENETIC ALGORITHMS + DATA STRUCTURES = EVOLUTION PROGRAMS**: 2nd, extended ed. Springer-Verlag New York, Inc., New York, NY, USA, 1994. 1.1, 3.1.4, 4.1, 4.1.2.

6 – KOZA, J. R. **GENETIC PROGRAMMING: ON THE PROGRAMMING OF COMPUTERS BY MEANS OF NATURAL SELECTION**. MIT Press, 1992. 1.1, 4.2.2.

7 – STORN, R.; PRICE, K. **DIFFERENTIAL EVOLUTION – A SIMPLE AND EFFICIENT ADAPTIVE SCHEME FOR GLOBAL OPTIMIZATION OVER OVER CONTINUAOUS SPACES**, Technical Report TR-95-012, 1995. 1.1, 3.1.3.

8 – REYNOLDS, R. G. **AN INTRODUCTION TO CULTURAL ALGORITHMS**: Proceeding of the 3rd annual conference on evolutionary programming, p. 131–139, 1994. 1.1, 2.6.

9 – REYNOLDS, R. G., PENG, B. **CULTURAL ALGORITHMS: MODELING OF HOW CULTURES LEARN TO SOLVE PROBLEMS**: Proccedings of the 18th IEEE international conference on toolswith artificial intelligence (ICTAI 2004), 2004. 1.1, 2.6.1, 2.6.1.

10 –YAO, X.; LIU, Y.; LIN, G. M. **EVOLUTIONARY PROGRAMMING MADE FASTER**, IEEE Trans. Evolutionary Computation, 3:82–102, 1999. 1.1, 4.1.2.

11 – KEYMEULEN, D.; ZEBULUM, R.; DUONG, V.; GUO, X.; FERGUSON, I.; STOICA, A. **HIGH TEMPERATURE EXPERIMENTS FOR CIRCUIT SELF-RECOVERY**. Aceito para publicação na Conferencia de Computação Genética e Evolutiva (Genetic and Evolutionary Computation Conference – GECCO), Seattle, WA, EUA, Junho, 2004.

12 – STOICA, A.; ZEBULUM, R.; FERGUSON,M.I.; KEYMEULEN, D.; DUONG, V; DAUD , T; GUO, X. **EVOLUTIONARY CONFERATION OF FIELD PROGRAMMABLE ANALOG DEVICES**, conferência especial IEEE Aerospace Conference, Big Sky, MT, EUA, Março, 2003.

13 – CHU, P. P. **FPGA PROTOTYPING BY VHDL EXAMPLES: XINLNX SPARTAN-3 VERSION**. Cleveland State Universitty: John Wiley & Sons, 2008.

14 – GOLDBERG, D. **GENETIC ALGORITHMS IN SEARCH, OPTIMIZATION AND MACHINE LEARNING**, Addison-Wesley 1989.

15 – GUCCIONE, S. A.; LEVI, D. **XILINX BITSTREAM INTERFACE: A JAVA-BASED INTERFACE TO FPGA HARDWARE**, Bellingham, WA, EUA, 1998.