UNIVERSIDADE FEDERAL DO PARANÁ
SETOR DE TECNOLOGIA
CURSO DE ENGENHARIA ELÉTRICA
ÊNFASE EM ELETRÔNICA E TELECOMUNICAÇÕES

VICTOR LUIS PADILHA MAYER

MÉTODO PARA IMPLEMENTAÇÃO DE ALGORITMOS DE OTIMIZAÇÃO EM HARDWARE

CURITIBA
2017

VICTOR LUIS PADILHA MAYER

MÉTODO PARA IMPLEMENTAÇÃO DE ALGORITMOS DE OTIMIZAÇÃO EM HARDWARE

Monografia apresentado como trabalho de conclusão de Curso da disciplina TE105 - Projeto de Graduação, do departamento de Engenharia Elétrica, Setor de Tecnologia, da Universidade Federal do Paraná, para obtenção do título de Engenheiro Eletricista.

Orientadora: Prof.ª. Dr.ª Sibilla Batista da Luz França
Coorientador: Prof. Dr. André Augusto Mariano

CURITIBA

2017

TERMO DE APROVAÇÃO

VICTOR LUIS PADILHA MAYER

MÉTODO PARA IMPLEMENTAÇÃO DE ALGORITMOS DE OTIMIZAÇÃO EM HARDWARE

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia Elétrica da Universidade Federal do Paraná como requisito à obtenção do grau de Engenheiro Eletricista, pela seguinte banca examinadora:

_____

Prof.ª. Dr.ª Sibilla Batista da Luz França
Orientadora:
Departamento de Engenharia Elétrica, UFPR

_____

Prof. Dr. Eduardo Parente Ribeiro
Departamento de Engenharia Elétrica, UFPR

_____

Prof. Me. Carlos Alexandre Gouvêa da Silva
Departamento de Engenharia Elétrica, UFPR

Curitiba, 04 de dezembro de 2017

AGRADECIMENTOS

Agradeço aos meus pais e familiares, que me apoiaram e incentivaram ao longo desses anos da graduação, sempre torcendo pelo meu sucesso.

Agradeço também aos amigos e colegas que me acompanharam durante essa trajetória, com os quais compartilhei momentos de alegria, tristeza, angustia e ansiedade. Realmente foram muitas emoções nesses últimos anos.

Agradeço ao curso de Engenharia Elétrica da Universidade Federal do Paraná, em especial aos meus orientadores Prof.ª. Sibilla Batista da Luz França e Prof. André Augusto Mariano pelo apoio e paciência que tiveram comigo durante os últimos meses.

Agradeço a Mariane Gavioli Bergamini por dispor do seu tempo corrido para me ajudar nesse trabalho.

Por fim agradeço a todos que direta ou indiretamente me ajudaram a construir o meu caminho até esse momento.

“Nós somos quem podemos ser
Sonhos que podemos ter”

Humberto Gessinger

RESUMO

A utilização de algoritmos para otimização de funções apresenta um problema em relação ao tempo gasto para realização de simulações executadas via software, principalmente quando se trata de aplicação em tempo real. Conforme a complexidade da função objetivo e do algoritmo crescem, o aumento do tempo necessário para a otimização pode tornar o algoritmo inviável para esse tipo de situação. A partir disso, a solução proposta é a utilização da FPGA (Matriz de portas programáveis em campo) para otimizar a execução e permitir que essas aplicações em tempo real sejam possíveis. Uma das vantagens que esse dispositivo fornece é o paralelismo que ele permite, com o qual é possível desenvolver múltiplos processos que funcionem de forma concorrente para reduzir o tempo total necessário pelos algoritmos. Esse trabalho então propõe um método para, a partir de um algoritmo projetado para software, obter um equivalente para ser implementado em hardware. Nesse método incluem-se as ferramentas necessárias para realizar essa conversão, considerações realizadas para essa conversão e formas de validação e verificação dos resultados obtidos. Os resultados desse trabalho demonstram que o tempo de processamento em hardware para os algoritmos testados pode ser menor que o tempo necessário para executar o algoritmo via software, concluindo que a FPGA pode ser considerada como uma forma de solucionar o problema relacionado ao tempo de execução dos algoritmos.

Palavras-chave: Implementação em hardware, algoritmos de otimização, FPGA.

ABSTRACT

The use of algorithms to function's optimization introduce a problem in relation to the time spent to do the simulations performed by software, principally when it comes to real time applications. As the complexity of the objective function and algorithm increases, the increase of the necessary time to the optimization can make the algorithm impraticable to this kind of situation. From this, the proposed solution is the use of FPGA (Field Programmable Gate Array) to optimize the performance and allows the real time applications. One of the advantages that this device provides is the parallelism that it allows, with which it is possible to develope multiple processes that work concurrently to reduce the total time nedded by the algorithm. So this work proposes a method for, starting from a algorithm developed to software, to obtain a equivalent one to be implemented in hardware. In these method are included the tools used to this conversion, the considerations made in this conversion and the validation and verification forms of the obtained results. The results of this work demonstrate that the hardware's nedded process time to the tested algorithms can be lower than the software's nedded execution time, concluding that the FPGA can be considered as a way to solve the problem related to the execution time of algorithms.

Keywords: Hardware implementation, optimization algorithm, FPGA.

LISTA DE ILUSTRAÇÕES

FIGURA 1 – INVESTIMENTO EM APLICAÇÕES UTILIZANDO FPGAs NA ÁSIA - PACÍFICO 12
FIGURA 2 – ESTRUTURA INTERNA DA FPGA..... 16
FIGURA 3 – DIAGRAMA DO ALGORITMO DA EVOLUÇÃO DIFERENCIAL.. 18
FIGURA 4 – DIAGRAMA DO ALGORITMO DE OTIMIZAÇÃO DE LEÃO 21
FIGURA 5 – LFSR DE 16 BITS..... 24
FIGURA 6 – FLUXOGRAMA DO PROJETO..... 25
FIGURA 7 – LFSR DE 14 BITS..... 30
FIGURA 8 – RESULTADOS PARA VALIDAÇÃO DO CÓDIGO GERADO PARA DE 32
FIGURA 9 – COMPARAÇÃO ENTRE RESULTADOS DO MATLAB E SIMULAÇÃO PARA DE 33
FIGURA 10 – SIMULAÇÃO TEMPORAL DO ALGORITMO DE 34
FIGURA 11 – INTEGRAÇÃO DO SIMULINK COM A FPGA..... 35
FIGURA 12 – RESULTADO DA SIMULAÇÃO UTILIZANDO O FIL PARA DE 35
FIGURA 13 – RESULTADOS PARA VALIDAÇÃO DO CÓDIGO GERADO PARA LOA 36
FIGURA 14 – COMPARAÇÃO ENTRE RESULTADOS DO MATLAB E SIMULAÇÃO PARA LOA 37
FIGURA 15 – SIMULAÇÃO TEMPORAL DO ALGORITMO LOA 39
FIGURA 16 – SIMULAÇÃO UTILIZANDO FIL PARA LOA..... 39
FIGURA 17 – INICIO DA UTILIZAÇÃO DO HDL CODER..... 45
FIGURA 18 – DEFINIÇÃO DO TIPO DE VARIÁVEIS..... 47
FIGURA 19 – EXEMPLO DE INFORMAÇÕES DE ERRO FORNECIDAS PELA FERRAMENTA..... 48
FIGURA 20 – TELA DE OPÇÕES PARA GERAÇÃO DO CÓDIGO EM HDL.. 49

LISTA DE SIGLAS

DE - Differential evolution (Evolução diferencial)

FPGA - Field programmable gate array (Matriz de portas programáveis em campo)

FIL - FPGA-in-the-loop

HCPLD - High capacity programmable logic device (Dispositivo lógico programável de alta capacidade)

HDL - Hardware description language (Linguagem de descrição de hardware)

LaSICO - Laboratório de simulação, controle e otimização de sistemas.

LFSR - Linear-feedback shift register (Registrador de deslocamento linear realimentado)

LOA - Lion optimization algorithm (Algoritmo de otimização de leão)

LUT - Look up table (Tabela lógica)

MATLAB - Matrix laboratory (Laboratório de matrizes)

PLD - Programmable logic device (Dispositivo lógico programável)

UFPR - Universidade federal do Paraná

VHDL - VHSIC hardware description language (Linguagem VHSIC de descrição de hardware)

VHSIC - Very high speed integrated circuits (Circuito integrado de velocidade muito alta)

SUMÁRIO

1. INTRODUÇÃO 11

1.1 OBJETIVO GERAL 13

1.2 OBJETIVOS ESPECÍFICOS 13

2. FUNDAMENTAÇÃO TEÓRICA 14

2.1 PLD 14

2.2 FPGA 15

2.3 FUNÇÃO OBJETIVO 17

2.4 ALGORITMO DE EVOLUÇÃO DIFERENCIAL (DE) 17

2.5 ALGORITMO DE OTIMIZAÇÃO DE LEÃO (LOA) 20

2.6 GERADOR DE NÚMEROS PSEUDO-ALEATÓRIOS 23

3. MATERIAIS E MÉTODOS 25

3.1 METODOLOGIA 25

3.2 ESTUDOS DE CASO 28

4. RESULTADOS 30

4.1 LFSR 30

4.2 ALGORITMO DE EVOLUÇÃO DIFERENCIAL 31

4.3 ALGORITMO DE OTIMIZAÇÃO DE LEÃO 36

5. CONCLUSÃO 41

REFERÊNCIAS 43

APÊNDICE 1 – UTILIZAÇÃO DO HDL *CODER* 45

1. INTRODUÇÃO

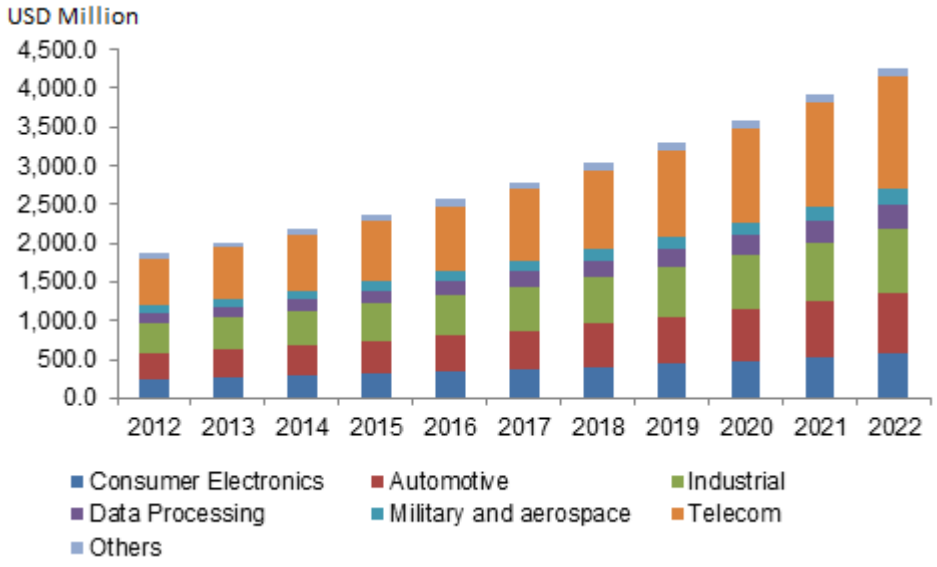
A implementação de algoritmos, independentemente de sua função ou complexidade, é atualmente feita de forma bastante natural através de dispositivos como microcontroladores e microprocessadores. Esses dispositivos são utilizados em diversas aplicações, onde são normalmente programados de modo sequencial [TOSCANI; OLIVEIRA; CARISSIMI, 2003].

Essa forma de programação apresenta um problema em determinadas aplicações. Conforme a complexidade dos algoritmos aumenta, o número de instruções necessárias para o algoritmo também tende a aumentar. Esse maior número de instruções necessita de período de tempo maior para serem processadas. Quando se deseja aplicações em tempo real, o fato do tempo de processamento ser muito longo pode inviabilizar a utilização de determinado algoritmo. Uma solução possível é a programação concorrente, onde aproveita-se do paralelismo para uma execução mais rápida. Com esse paralelismo, é possível realizar operações de maneira simultânea, o que faz com que o tempo total de execução do algoritmo seja menor.

A partir disso, esse trabalho propõem a utilização de FPGAs (*Field Programmable Gate Array*) para a implementação de algoritmos, visando aplicações em tempo real. Quando se utiliza esses dispositivos, deve-se sempre considerar sua resposta atrelada a hardware, onde podem ser projetados diversos circuitos independentes para operar de forma concorrente, o que pode otimizar o tempo total do processo.

Além disso, as FPGAs vêm ganhando espaço no mercado atual, principalmente nas áreas de aplicação mais críticas, onde se deseja muitas vezes respostas em tempo real. Um grande exemplo disso é a área de telecomunicações, onde existe um grande fluxo de dados sendo processado a todo momento e esses dados necessitam de uma resposta quase imediata. A Figura 1 apresenta esse crescimento, demonstrando o estado atual e previsão de crescimento no uso desses dispositivos no mercado da Ásia-Pacífico.

FIGURA 1 – INVESTIMENTO EM APLICAÇÕES UTILIZANDO FPGAs NA ÁSIA - PACÍFICO



FONTE: GLOBAL MARKET INSIGHTS, 2016.

A crescente utilização desses dispositivos se deve muito ao fato que nos últimos anos houve um significativo aumento no número de elementos lógicos disponíveis nas FPGAs, juntamente com uma redução no seu preço [ALBERTINI, 2015].

Esse investimento verificado no mercado asiático é uma tendência em outras partes do mundo também. O uso das FPGAs vem crescendo nos últimos anos e deve ser cada vez mais incorporado o mercado, ganhando cada vez mais destaque. Conhecer esses dispositivos possibilita uma infinidade de novas aplicações e o desenvolvimento de novas soluções, trazendo um grande avanço na forma com que processos são realizados.

Esse trabalho propõe a conversão de algoritmos de otimização desenvolvidos em MATLAB (*Matrix Laboratory*) para a implementação em hardware, criando uma metodologia para auxiliar na conversão desses e outros algoritmos. Por utilizarem conjuntos de dados grandes, a implementação em hardware se mostra interessante para esses algoritmos devido ao fato de poder paralelizar operações que utilizem esses conjuntos, otimizando o tempo total do processo.

1.1 OBJETIVO GERAL

Este trabalho tem como objetivo desenvolver um método para a implementação de um algoritmo de otimização em uma FPGA e torná-lo o mais eficiente possível em questão de tempo de processamento e utilização de hardware.

1.2 OBJETIVOS ESPECÍFICOS

Para o desenvolvimento desse trabalho, foram definidos alguns objetivos específicos, a fim de dividir suas etapas. Esses objetivos são:

- Analisar o algoritmo em MATLAB e criar um arquivo equivalente que seja compatível com a extensão HDL *Coder*.
- Configurar o HDL *Coder* a fim de obter um arquivo no formato .vhd, sintetizável e implementável.
- Realizar simulações comportamentais para verificar a funcionalidade e a validade do código.
- Realizar as simulações temporais para obter o tempo necessário para executar o algoritmo.
- Utilizar a ferramenta FPGA-*in-the-loop* para verificar o comportamento em hardware.
- Gerar um arquivo no formato .bit, que possa ser descarregado na FPGA.

2. FUNDAMENTAÇÃO TEÓRICA

Essa seção tem como objetivo abordar alguns conceitos que serão utilizados durante o trabalho. É extremamente importante que os conceitos aqui explicados sejam compreendidos, pois todo e desenvolvimento e a análise dos resultados é feita baseada nessas informações.

Primeiramente, será introduzido o conceito de PLDs (*Programmable Logic Device*), um grupo de dispositivos que inclui as FPGAs, que serão abordadas de maneira mais aprofundada em seguida. Após isso, será explicado o conceito de função objetivo, conceito que será utilizado nos tópicos seguintes. Os próximos tópicos abordados serão a respeito dos algoritmos estudados, os quais servirão como base para o desenvolvimento do método para implementação em hardware. Por fim será analisada uma forma de gerar sequências de números pseudoaleatórios.

2.1 PLD

Os dispositivos lógicos programáveis foram introduzidos pela Texas Instruments, em 1970. Como o próprio nome já propõe, a ideia por trás desses dispositivos era de desenvolver circuitos integrados cuja função lógica fosse definida pelo usuário através de programação. Em outras palavras, seriam produzidos *chips* sem função predefinida, dando liberdade ao programador de desenvolver seu próprio circuito integrado.

Diferentemente de um microprocessador, que pode ser programado a fim de executar um programa, mas que possui um hardware fixo construído em torno da unidade central de processamento, a programação dos PLDs se dá em nível de hardware, onde o dispositivo é programado de forma que o circuito interno seja configurado para atender a uma determinada especificação [DAVIES, 2008].

Os primeiros PLDs trouxeram grandes inovações na época, principalmente quando comparados aos dispositivos discretos utilizados. A

redução do tamanho das placas de circuitos impressos foi uma das vantagens desses dispositivos, assim como uma redução no consumo de energia e menores perdas de potência.

Ao longo dos anos, diversos novos tipos de PLDs foram sendo criados, cada uma trazendo novidades acompanhadas de um aumento na capacidade e complexidade dos dispositivos. Entre esses novos tipos, estão as FPGAs.

2.2 FPGA

Em 1985 a Xilinx lançava comercialmente as primeiras FPGAs. Esses dispositivos eram mais sofisticados do que os demais PLDs lançados até então, tendo como alvo projetos complexos que exigiam elevado desempenho, sendo denominadas como HCPLDs (*High Capacity Programmable Logic Device*).

Sua arquitetura interna composta por uma matriz de blocos lógicos permite que as FPGAs tenham uma alta capacidade lógica, grandes quantidades de entradas/saídas e serem capazes de operar com elevadas frequências de *clock*. Essas caraterísticas, aliada com o paralelismo que a ferramenta permite, fazem com que a FPGA seja uma boa opção em diversas aplicações, inclusive quando se deseja a resposta em tempo real.

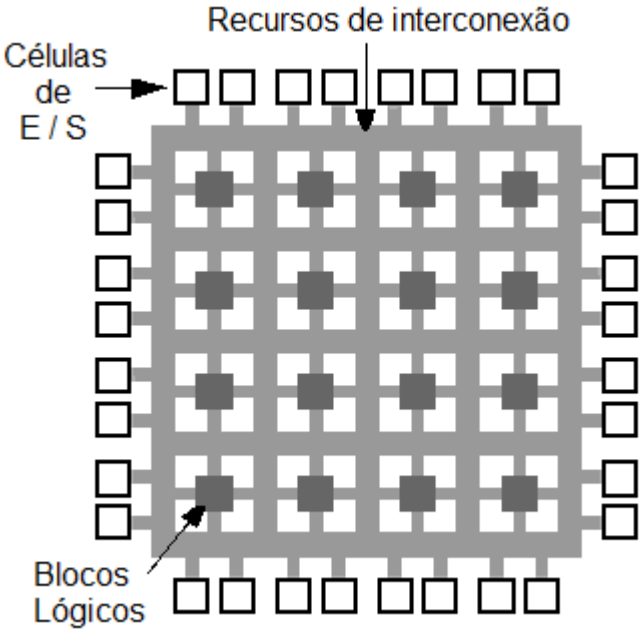
A Figura 2 demonstra como é a matriz interna da FPGA. Nessa matriz existem três elementos principais que definem o comportamento do dispositivo. Esses elementos são: blocos lógicos, as células de entrada/saída e os recursos de interconexão.

Os blocos lógicos são responsáveis por receber as funções lógicas necessárias para a aplicação. Esses blocos são compostos por algumas funções lógicas simples (tipicamente LUTs (*Look-up tables*) e *Flip-flops*, podendo incluir outras funções dependendo do modelo) e irão conter pequenos pedaços do circuito que se deseja implementar. As células de entrada/saída são responsáveis por fornecer os estímulos do circuito e também são os elementos em que são verificadas as saídas do mesmo. Os recursos de interconexão são

responsáveis por conectar os blocos lógicos entre si. Juntos esses elementos formam a matriz lógica [CASTRO, 2011].

O grande potencial da FPGA encontra-se justamente nessa matriz, onde um bloco lógico isolado não apresenta grande capacidade, mas ao interconectar milhares deles as possibilidades de aplicações aumentam significativamente.

FIGURA 2 – ESTRUTURA INTERNA DA FPGA



FONTE: CASTRO, 2011

Outro ponto importante desses dispositivos é linguagem utilizada para programação. Diferente de um microprocessador onde um programa está sendo executado, a FPGA se trata de um hardware, então a linguagem utilizada será uma HDL (*Hardware Description Language*). Essa HDL descreve como o hardware deve operar, descrevendo o comportamento esperado do circuito, suas entradas e saídas.

Uma etapa bastante importante antes de partir para a implementação é a síntese do código HDL. Nessa etapa o código é convertido em um modelo de circuito a ser implementado, definindo um *netlist* com todos os elementos lógicos que compõem o projeto. A partir desse *netlist* são atribuídos aos blocos lógicos

as funções adequadas necessárias para compor o projeto, juntamente com as conexões necessárias para esses blocos e as entradas/saídas a serem utilizadas, definindo um modelo para o circuito desejado. Com isso pode-se partir para a etapa de implementação, onde o modelo é convertido em um circuito digital que pode ser descarregado na FPGA.

2.3 FUNÇÃO OBJETIVO

A avaliação da aptidão dos indivíduos é feita através da análise da função objetivo, função de custo ou função de aptidão (*fitness*). Segundo Bergamini (2017b), a função objetivo trata da minimização ou maximização do problema estudado, podendo ser representada pela fórmula apresentada na equação 2.1. Essa equação representa a função custo, a qual trata-se do erro de predição da função. Nessa equação os valores $(x_1, x_2, \dots, x_{Np})$ representam os indivíduos de uma população de tamanho Np, avaliados pela função f. Essa população representa um conjunto de indivíduos que fornecem valores para a função objetivo e serão parâmetros de entrada para os algoritmos estudados.

$$\min \mathrm{f}(X_i) = f(x_1, x_2, \dots, x_{Np}) \tag{2.1}$$

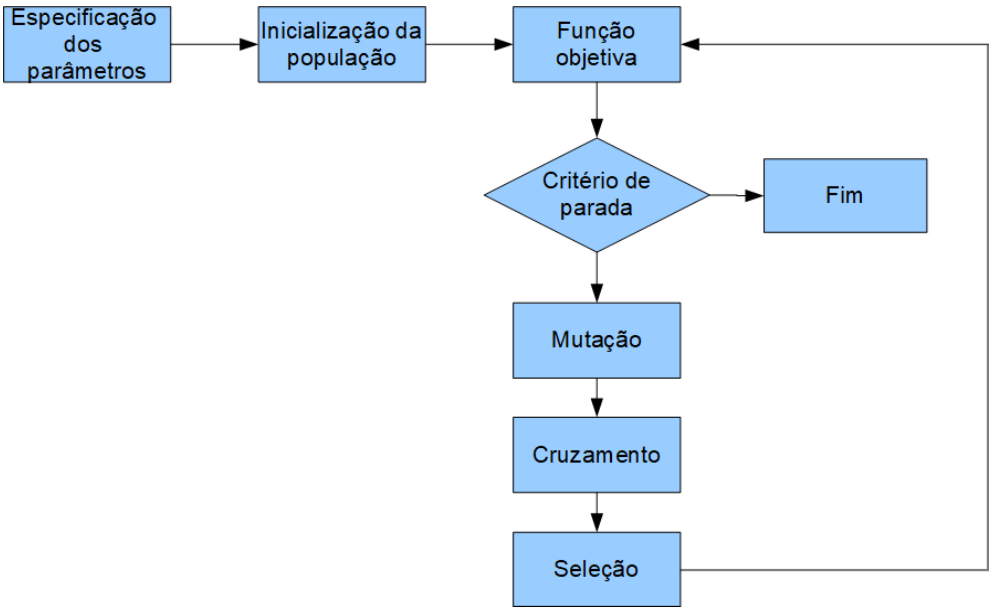
2.4 ALGORITMO DE EVOLUÇÃO DIFERENCIAL (DE)

O primeiro algoritmo a ser estudado é o algoritmo de Evolução Diferencial, uma meta-heurística introduzida por Storn e Price (1995). Esse é um algoritmo evolutivo baseado em população, onde os indivíduos dessa população fornecem os valores de entrada da função. A otimização é feita através de mutações, onde os indivíduos dessa população têm seus valores alterados, a fim de obter novos indivíduos que possam possivelmente forneçam melhores resultados para a função objetivo.

A Figura 3 mostra o diagrama desse algoritmo. Primeiramente, são definidos alguns parâmetros de entrada, que estarão diretamente relacionados ao comportamento do algoritmo. Entre esses parâmetros podemos citar: a taxa

de mutação, a probabilidade de cruzamento da população, o tamanho dessa população, os limites que as variáveis podem atingir, entre outros parâmetros [BERGAMINE, 2017a].

FIGURA 3 – DIAGRAMA DO ALGORITMO DA EVOLUÇÃO DIFERENCIAL



FONTE: AUTOR, 2017

Em seguida é feita a inicialização da população, representada na figura 3 por um bloco com o mesmo nome. Essa inicialização é feita aleatoriamente, de forma a tentar abranger ao máximo o espaço de busca da função. A população é definida a partir de um tamanho pré-determinado, dentro de um espaço de busca com um valor máximo e um valor mínimo. Isso pode ser visto na equação 2.2, onde Pop_L e Pop_U são os limites inferior e superior que a população pode adquirir, enquanto o índice M representa um indivíduo de uma população de tamanho N.

$$Pop_L \leq Pop(M) \leq Pop_U \tag{2.2}$$

Uma vez inicializada a população, é possível verificar se ela atende aos critérios de parada. Conforme a figura 3 indica, ao aplicar a população gerada na função objetivo, verifica-se se essa possui indivíduos que atendam aos critérios de parada. Caso isso ocorra, o algoritmo terá encontrado o seu ponto

ótimo e poderá ser encerrado, caso contrário, é necessária a etapa da mutação para gerar novas soluções.

Na etapa da mutação, novos indivíduos são gerados a fim de expandir o espaço de busca da função. É possível utilizar diversas estratégias para determinar a mutação dos indivíduos. Uma dessas estratégias, utilizada no trabalho de Bergamini (2017b, p. 46), pode ser vista na equação 2.3. Os índices α, β e γ representam índices para os indivíduos aleatórios da população, enquanto F é o chamado fator de mutação, uma constante que irá afetar o valor da população mutante, chamada de $Pop_m(M)$.

$$Pop_m(M) = Pop(\alpha) + F * (Pop(\beta) - Pop(\gamma)) \tag{2.3}$$

A etapa seguinte, conforme a figura 3, é o cruzamento. Nessa etapa são combinados em uma nova solução os valores da etapa da mutação com os valores pré-determinados, gerando uma nova população chamada de $Pop_C(M)$. Para isso é avaliado o parâmetro da taxa de Cruzamento (Cr), definida no início do algoritmo. Compara-se esse parâmetro com uma variável aleatória, chamada aqui de $Rand(M)$, no intervalo entre zero e um para a tomada de decisão. Conforme a equação 2.4 demonstra, se o valor de $Rand(M)$ for menor ou igual ao parâmetro Cr, o indivíduo da população mutante passa a fazer parte da nova solução, caso contrário, é mantido o valor da população original [BERGAMINI, 2017b].

$$Pop_C(M) = \begin{cases} Pop_m(M) \;\; se\; Rand(M) \leq Cr \\ Pop(M) \;\; se\; Rand(M) > Cr \end{cases} \tag{2.4}$$

Por fim vem a etapa da seleção. Nessa etapa são comparados os valores da função objetivo utilizando a população inicial ($Pop(M)$) e os valores da função objetivo com a população gerada na etapa de cruzamento ($Pop_C(M)$). No caso do algoritmo ser de minimização de funções, a seleção irá comparar as duas populações e escolher os indivíduos que resultem no menor valor para a função objetivo (f), substituindo-os na população para a próxima geração. Essa etapa está representada através da equação 2.5. Com isso será gerada uma nova população ($Pop_S(M)$), com possíveis melhores valores para minimização da função.

$$Pop_S(M) = \begin{cases} Pop_C(M) \;\; se\; f\big(Pop_C(M)\big) \leq \; f\big(Pop(M)\big) \\ Pop(M) \;\; se\; f\big(Pop_C(M)\big) > \; f\big(Pop(M)\big) \end{cases} \tag{2.5}$$

Com essa nova população, aplica-se na função objetivo e verifica-se se os critérios de parada do algoritmo são satisfeitos. Caso isso ocorra, o algoritmo chega ao fim, caso contrário, são refeitos os passos da mutação, cruzamento e seleção até atingir esses critérios.

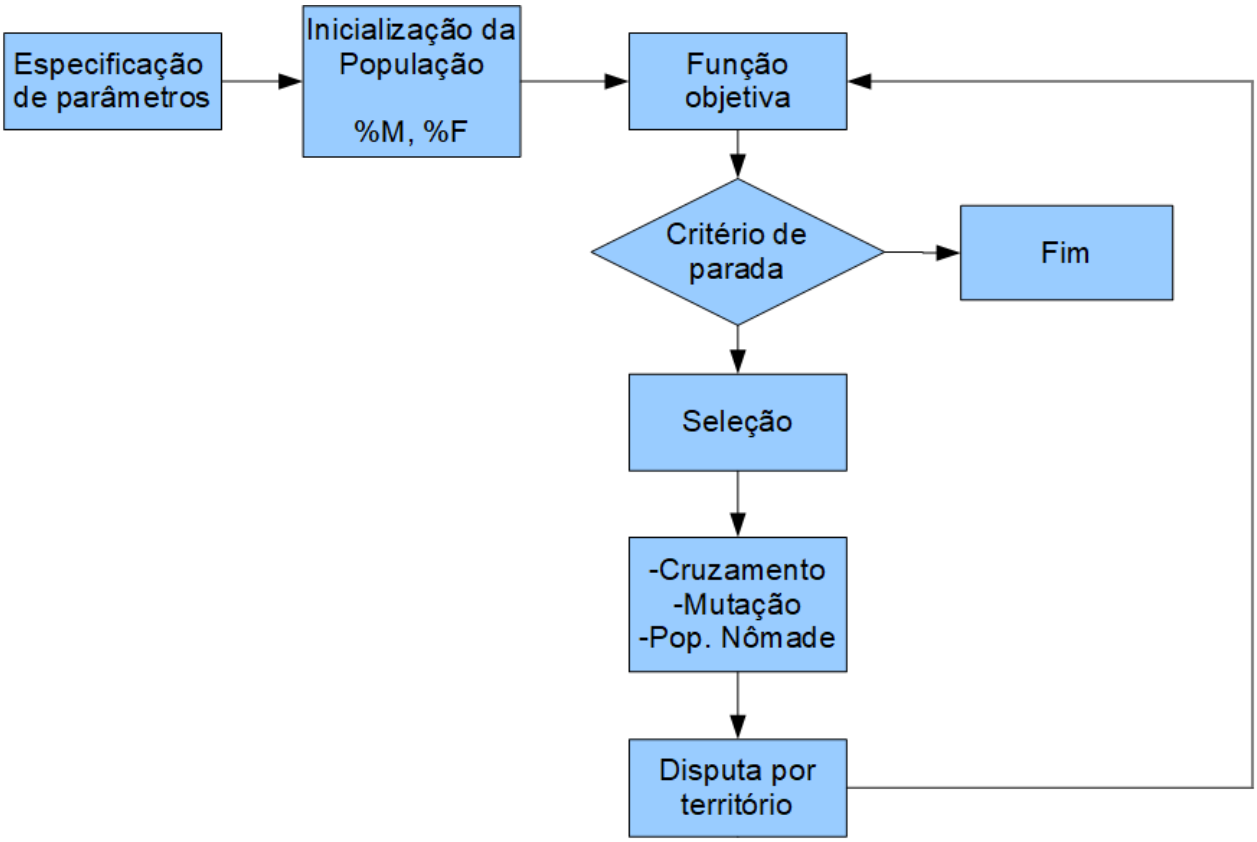
2.5 ALGORITMO DE OTIMIZAÇÃO DE LEÃO (LOA)

Esse algoritmo, descrito por Rajakumar (2012) e Wang et al. (2012), assim como a evolução diferencial, também se trata de um algoritmo evolutivo baseado em população que possui inspiração na natureza. Os leões são animais selvagens, cujo comportamento e organização social são bem definidos, onde o cooperativismo e a força dos leões são fatores de extrema importância [BERGAMINI, 2017b]. A partir desses conceitos foi desenvolvido o algoritmo que pode ser utilizado para resolver problemas de otimização.

Esse algoritmo introduz alguns novos conceitos que não estavam presentes na Evolução Diferencial. É importante ressaltar também que existem duas abordagens diferentes para esse algoritmo: A disputa por território, descrita por Rajakumar (2012) e a abordagem da caça, descrita no trabalho de Wang et al. (2012). Nesse trabalho a abordagem escolhida foi a da disputa por território, por fornecer as melhores soluções ótimas [BERGAMINI, 2017b].

A figura 4 mostra o diagrama do Algoritmo de Otimização de Leão. Primeiramente, são escolhidos os parâmetros de entrada do algoritmo, entre eles a taxa de mutação (F), taxa de cruzamento (Cr) , dimensão do problema, entre outros parâmetros necessários ao longo do algoritmo.

FIGURA 4 – DIAGRAMA DO ALGORITMO DE OTIMIZAÇÃO DE LEÃO



FONTE: AUTOR, 2017

O primeiro dos novos conceitos presentes nesse algoritmo é a divisão da população. Conforme a figura 4 demonstra, ocorre uma divisão da população inicial entre machos e fêmeas. Em um grupo de leões a maioria dos indivíduos é composto por leoas, onde cerca de 70% a 90% da população se enquadra. A porcentagem restante entre 10% e 30% será composta por leões [BERGAMINI, 2017b].

Esses indivíduos da população são avaliados através da função objetivo, onde o leão e a leoa com a melhor aptidão, que no caso da minimização são aqueles que fornecem os menores valores da função objetivo, são selecionados para gerar filhotes para a próxima geração.

Após a seleção dos indivíduos, a próxima etapa conforme a figura 4 é o cruzamento. Nessa etapa ocorre a troca de informações entre os pais, que

geram novos indivíduos para a população. Os leões selecionados na etapa anterior geram filhotes que poderão no futuro fazer parte do bando principal.

Conforme demostra no fluxograma da figura 4, esses novos indivíduos passam pela etapa da mutação, onde um ou mais genes do filhote podem ser modificados a partir da taxa de mutação. Essa mutação é definida através de uma ou mais alteração na cadeia de bits que representa o valor do filho, onde o bit que originalmente receberia o valor um, passa a receber o valor zero, e vice-versa. Esses novos valores substituem os valores da população anterior, e após isso parte-se para a etapa da disputa de território. Os genes dos filhos passam a ser representados por uma cadeia de bits, ondem os resultados do cruzamento e mutação podem ser descritos da seguinte maneira:

Resultado do cruzamento:

Filho1 = 1000001100 = 0,5125;

Filho2 = 1110001110 = 1,8891;

Resultado da mutação:

Filho1 = 100**1**001100 = 0,5742;

Filho2 = 111**1**001110 = 0,9473.

Para a disputa de território, é gerada uma nova divisão da população, chamada de nômade, de até 10% da população atual. Essa população é composta por indivíduos masculinos que foram expulsos do bando, mas que voltam para disputar o antigo lar [BERGAMINI, 2017b].

Durante a disputa de território são selecionados os melhores filhotes do bando, que irão competir com os leões da população nômade. Os filhotes que não oferecerem melhores soluções são excluídos do grupo, podendo ser mortos ou expulsos do bando.

Quando os filhotes atingem a maturidade, esses podem disputar território com o leão dominante do bando, e caso um dos filhotes tenha uma força maior que a do atual rei, ele se tornará o novo leão governante [BERGAMINI, 2017b].

Após a disputa por território uma nova população é gerada, a qual pode ser aplicada na função objetivo. Verifica-se se essa população atende aos critérios de parada do algoritmo, podendo ou não chegar ao fim do algoritmo.

2.6 GERADOR DE NÚMEROS PSEUDO-ALEATÓRIOS

Algo indispensável nos algoritmos estudados nesse trabalho é a presença da aleatoriedade em diversos pontos dos códigos. O MATLAB conta com funções prontas para gerar os valores aleatórios necessários para que o algoritmo trabalhe de forma adequada. Isso facilita o trabalho dos desenvolvedores da ferramenta, que não precisam se preocupar com os detalhes dessas funções, só fornecer as entradas adequadas.

Quando deseja-se fazer a conversão do código em MATLAB para o VHDL, um problema surge em virtude dessa simplificação existente no MATLAB. As ferramentas de desenvolvimento de PLDs não fornecem funções prontas para gerar números aleatórios, sendo necessário desenvolver manualmente uma função que seja implementável em hardware e que possa cumprir com essa função.

Uma das saídas para isso é o LFSR (*Linear-feedback shift register*), que consiste num registrador de deslocamento de n bits, capaz de gerar sequências pseudoaleatórias, contendo $2^n - 1$ valores (onde apenas o valor 0 não é possível), que irão se repetir a cada 2^n estados [HEALY, 2017].

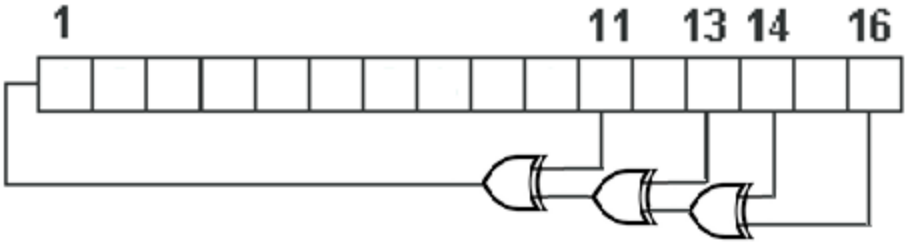
O modo mais comum de implementar esse gerador pseudoaleatório é utilizando registradores conectados em série e realimentadas através de portas lógicas “Ou-exclusivo” (XOR), de acordo com um polinômio de enésima ordem. Esse polinômio varia de acordo com o número de bits desejados para o gerador, mostrado de forma genérica na equação 2.6, onde os termos 1 e x^n estão sempre presentes no polinômio, e os coeficientes c_n assumem apenas os valores zero e um [HEALY, 2017].

$$LFSR(n) = 1 + c_1x^1 + c_2x^2 + \cdots + c_nx^n \qquad (2.6)$$

Em termos de circuito, esse polinômio representa quais registradores estão realizando as operações XOR para a realimentação. Para simplificar a explicação, será utilizado um LFSR de 16 bits, cujo polinômio é representado pela equação 2.7. Um LSFR pode ser descrito por mais de um polinômio, o que faz com que diferentes circuitos sejam possíveis para o mesmo resultado. No caso do polinômio escolhido para esse exemplo, seu circuito equivalente pode ser visto na figura 5, onde se percebe que os elementos cujos coeficientes c_n são diferentes de zero realizam sucessivas operações XOR entre si, para ao fim realimentar o primeiro registrador.

$$LFSR(16) = x^{16} + x^{14} + x^{13} + x^{11} + 1 \tag{2.7}$$

FIGURA 5 – LFSR DE 16 BITS

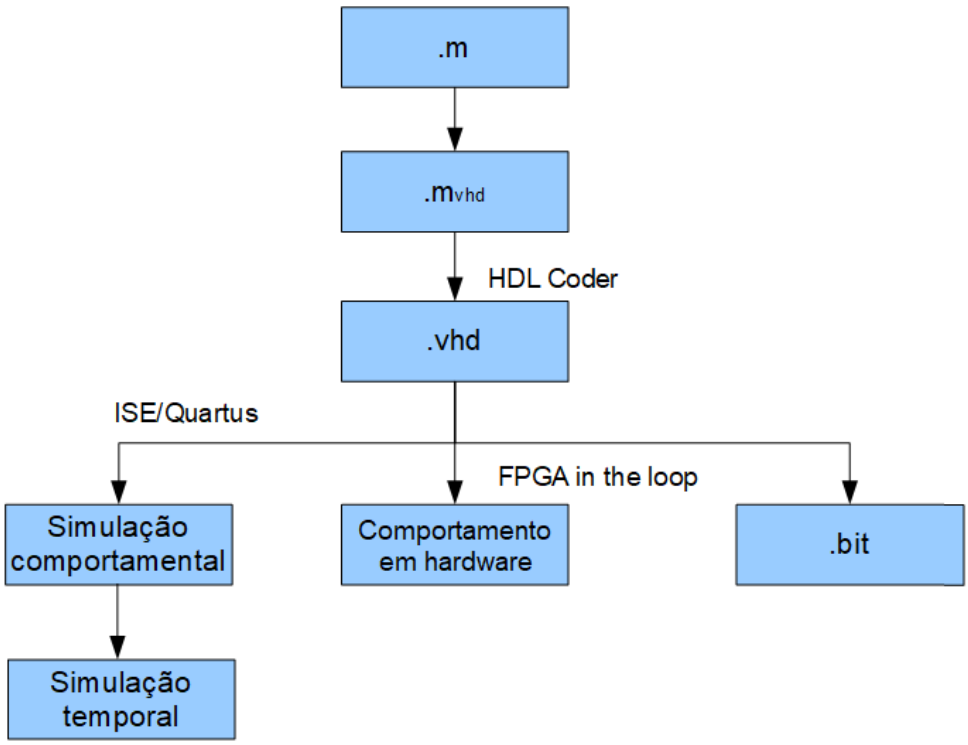


FONTE: AUTOR, 2017.

3. MATERIAIS E MÉTODOS

Nessa seção é apresentada a metodologia utilizada para o desenvolvimento desse trabalho, juntamente com as ferramentas utilizadas. A partir dessa metodologia foi desenvolvido um fluxo de trabalho, apresentado através da figura 6, e que será explicado no decorrer desse capitulo.

FIGURA 6 – FLUXOGRAMA DO PROJETO



FONTE: AUTOR, 2017

3.1 METODOLOGIA

A primeira ferramenta utilizada nesse trabalho foi o software MATLAB, desenvolvido pela MathWorks Inc. Esse software, que já é amplamente conhecido e difundido no ambiente acadêmico, trata-se de um software voltado para cálculos numérico de alta performance, capaz de trabalhar com uma precisão bastante elevada, utilizando uma linguagem simplificada.

Os algoritmos utilizados nesse trabalho foram desenvolvidos e validados em MATLAB pelo grupo LaSICO (Laboratório de Simulação, Controle e Otimização de Sistemas) da UFPR, pela Eng.ª Mariane Gavioli Bergamini, sob orientação do Prof. Dr. Gideon Villar Leandro. O primeiro passo do trabalho foi verificar o comportamento desses algoritmos na própria ferramenta, para compreender como funcionavam. Essa etapa de análise é representada no fluxograma da figura 6 pelo bloco .m (Formato de arquivos do MATLAB).

Após essa análise realizada no próprio MATLAB, desejou-se obter uma nova versão desse código em linguagem de descrição de hardware. Para isso utilizou-se uma extensão do MATLAB chamada HDL *Coder*, que auxilia com essa tradução do código.

Antes de partir para o uso dessa extensão foram necessárias algumas adaptações no código original no MATLAB, para que fosse possível utilizar o HDL *Coder*. Diferentes algoritmos necessitam de diferentes modificações, que podem ser verificadas através do manual do usuário da ferramenta. No caso dos Algoritmos de DE e LOA, os pontos adaptados foram:

- Números Aleatórios: O Algoritmo implementado originalmente em MATLAB contou com a geração de números aleatórios através de funções pré-programadas. Essas funções não são convertidas pelo HDL *Coder*.
- Formato de dados do tipo real: Os tipos de dados utilizados para armazenar as variáveis utilizadas nos algoritmos definidos como ponto flutuante puderam até ser convertidos pela ferramenta, porém o código gerado não é sintetizável.
- Matrizes: Deve-se tomar cuidado ao utilizar dados em forma de matrizes, pois existiram situações onde esse tipo de formato não pode ser utilizados, como por exemplo nas entradas e no *test bench*.
- *Loops*: São suportados pelo HDL *Coder* apenas loops com limite estático.
- Função: Todo código em MATLAB para ser aceito pelo HDL *Coder* deve estar na forma de uma função, para que o circuito correspondente possua as entradas e saídas definidas.

- *Test bench*: Juntamente com o código que desejou-se traduzir, foi criado um outro arquivo externo para fornecer as entradas da função, que serão necessárias durante a tradução.

Felizmente o HDL *Coder* facilita algumas dessas adaptações, automatizando a formatação de alguns tipos de dados e implementando o ponto fixo. Entretanto, as adaptações em relação aos números aleatórios e as matrizes utilizadas durante o algoritmo foram realizadas manualmente. Para traduzir o código do MATLAB foram inicialmente retiradas as funções aleatórias do código, tornando-as constantes. Essas aleatoriedades foram implementadas posteriormente de forma manual no código em HDL. Já as matrizes foram divididas em vetores e os loops foram delimitados com valores fixos. Com essas alterações foi criada uma nova versão do código do MATLAB, que no fluxograma da figura 6 é representado pelo bloco .m$_{vhd}$.

Com isso parte-se para a geração do arquivo .vhd, representado no fluxograma da figura 6 pelo bloco com mesmo nome. Essa extensão de arquivo é gerada para códigos na linguagem de descrição de hardware VHDL (VHSIC *Hardware Description Language*), que podem ser utilizados na FPGA. A etapa de geração desse arquivo é explicada de maneira mais aprofundada no Apêndice 1 desse trabalho. Com esse arquivo pode ser realizada a síntese do projeto e a simulação comportamental, utilizando algum software de *design* de dispositivos lógicos programáveis, como o Xilinx ISE, para FPGAs fabricadas pela Xilinx, ou o Altera Quartus, para FPGAs fabricadas pela Altera, juntamente com o Modelsim.

Foi escolhida para esse projeto a utilização da placa de desenvolvimento da Altera DE2-115, que conta com 114.480 elementos lógicos, 528 entradas/saídas, três osciladores de 50MHz para fornecer o *clock* de entrada, entre outros elementos. O número desses e outros elementos da placa devem ser respeitados para que o circuito projetado não ultrapasse os limites do hardware. Ao sintetizar um projeto utilizando tanto as ferramentas da Xilinx quanto da Altera, é possível obter o número de elementos utilizados pelo circuito,

podendo verificar se a capacidade do dispositivo é suficiente para o circuito descrito.

Esse código gerado necessitou ser validado, para isso foi realizada a simulação comportamental do código. Nessa simulação não foram considerados os atrasos do dispositivo. Essa etapa está representada no fluxograma da figura 6 pelo bloco cujo nome é “Simulação comportamental”.

Após validar o algoritmo com a simulação comportamental, as considerações feitas em relação as funções aleatórias do algoritmo foram corrigidas. Para isso foi criada uma função para gerar sequências pseudoaleatórias, através do LFSR. Essa função foi inserida nos pontos do código onde foram retiradas as funções de aleatoriedade. Após isso, foram realizadas simulações temporais para verificar a resposta do circuito considerando os atrasos de propagação do dispositivo.

Em seguida utiliza-se uma outra extensão do MATLAB chamada de FPGA-*in-the-loop*, conforme a figura 6. Essa extensão utiliza o próprio hardware da FPGA para realizar o processamento. Essa metodologia permite criar um *loop* entre o computador, que pôde fornecer entradas e verificar as saídas do circuito, e a FPGA. Com isso, pôde-se verificar se o comportamento em hardware corresponde as simulações.

Por fim, após verificadas todas essas simulações, pôde-se realizar o mapeamento dos pinos da FPGA e gerar o arquivo no formato .bit, que é o formato de arquivo que pôde ser descarregado na FPGA, encerrando o fluxo.

3.2 ESTUDOS DE CASO

Algo que ainda não foi apresentado até o momento nesse trabalho foi a função objetivo utilizada pelos algoritmos. Essa função será o critério de avaliação dos resultados obtidos pelos algoritmos e a partir dela será verificado o valor da aptidão dos indivíduos encontrados. A função avaliada nesse trabalho é a chamada função de Matyas, definida pela equação 3.1. Essa função se trata de uma função de teste, cujo mínimo global se encontra no ponto (0,0) e possui

o valor de zero. Essa função simples de duas variáveis serviu como base para desenvolver o método descrito nesse trabalho. Tipicamente por se tratar de uma função de teste, seu espaço de busca é limitado entre [-10,10] [SURJANOVIC; BINGHAM, 2017].

$$f(x, y) = 0.26(x^2 + y^2) - 0.48(xy) \tag{3.1}$$

Como o mínimo dessa função assume o valor de zero, qualquer valor obtido como resposta diferente de zero pode ser considerado o próprio erro. O objetivo então é encontrar valores para os parâmetros (x, y) dentro do intervalo de busca da função, que façam o resultado tender a zero.

Tanto para o algoritmo da Evolução Diferencial quanto para o Algoritmo de Otimização de Leão foram feitas as seguintes considerações para criar o código em MATLAB adaptado que será convertido pelo HDL *Coder* (será chamado de código principal a partir de agora):

- Números Aleatórios: Os pontos do código onde havia a necessidade de utilizar a geração de números aleatórios foram substituídos por vetores constantes, gerados de forma aleatória. Esse formato foi escolhido pois permite criar conjuntos de variáveis do tamanho dos *loops*, e a cada iteração o valor da variável era mudado de acordo com a posição no vetor. Por exemplo, para as taxas de mutação e de cruzamento, que são atualizadas para receber valores aleatórios N vezes (de acordo com o tamanho da população), foram criados vetores de tamanho N, que recebiam um conjunto de N valores fixos dentro do intervalo dessas taxas. A cada iteração do *loop* o valor recebido pelas variáveis correspondia ao valor da posição N do vetor. Com isso, quando era realizada a conversão do código utilizando o HDL *Coder*, não haviam declarações no algoritmo que não fossem executadas e convertidas.
- Matrizes: As variáveis que utilizavam matrizes foram adaptadas em vetores, e os pontos do código que utilizavam essas matrizes foram adaptados para utilizar esses vetores. Um exemplo dessa adaptação foi a matriz da população inicial, que tinha a dimensão (30,2) foi convertida em dois vetores de tamanho 30.

4. RESULTADOS

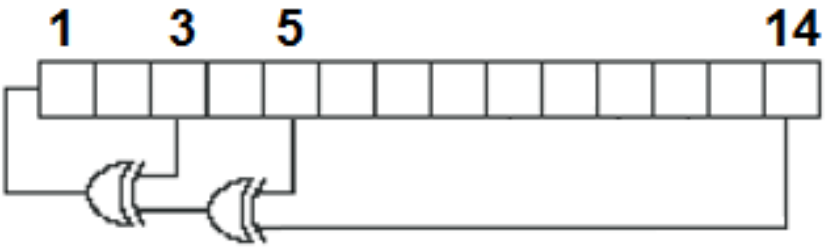
Nessa seção serão apresentados os resultados obtidos durante o desenvolvimento do trabalho. Serão analisados alguns resultados parciais para verificar o funcionamento das etapas desenvolvidas e para fazer as validações necessárias. Após isso, os resultados obtidos para a implementação em hardware serão comparados com os resultados esperados das simulações no MATLAB.

4.1 LFSR

Após definir o número de bits necessários para representar as variáveis do projeto (conforme demostrado no Apêndice 1), foi desenvolvida uma função correspondente ao circuito do LFSR, para gerar as sequências binárias pseudoaleatórias desse projeto. O polinômio utilizado para isso foi retirado do manual de recomendações da Xilinx e está sendo apresentado na equação 4.1, enquanto que o circuito equivalente é apresentado na figura 7.

$$LFSR(n) = 1 + x^3 + x^5 + x^{14} \tag{4.1}$$

FIGURA 7 – LFSR DE 14 BITS.



FONTE: AUTOR, 2017

O resultado desse circuito é uma contagem em sequência pseudoaleatória de difícil previsão. Nessa contagem o valor armazenado nos registradores é alterado a cada ciclo de *clock*, de forma que após 2^n (onde n foi

definido como 14 conforme o Apêndice 1) ciclos o primeiro valor da contagem se repita. Considerando essas informações, os valores da sequência para esse circuito só se repetem a cada 16384 ciclos de *clock*, após todas as opções para o contador de 14 bits terem sido obtidas apenas uma vez. Isso garante que a probabilidade de obter qualquer valor nessa sequência é igual a dos demais valores, garantindo que o contador não está viciado.

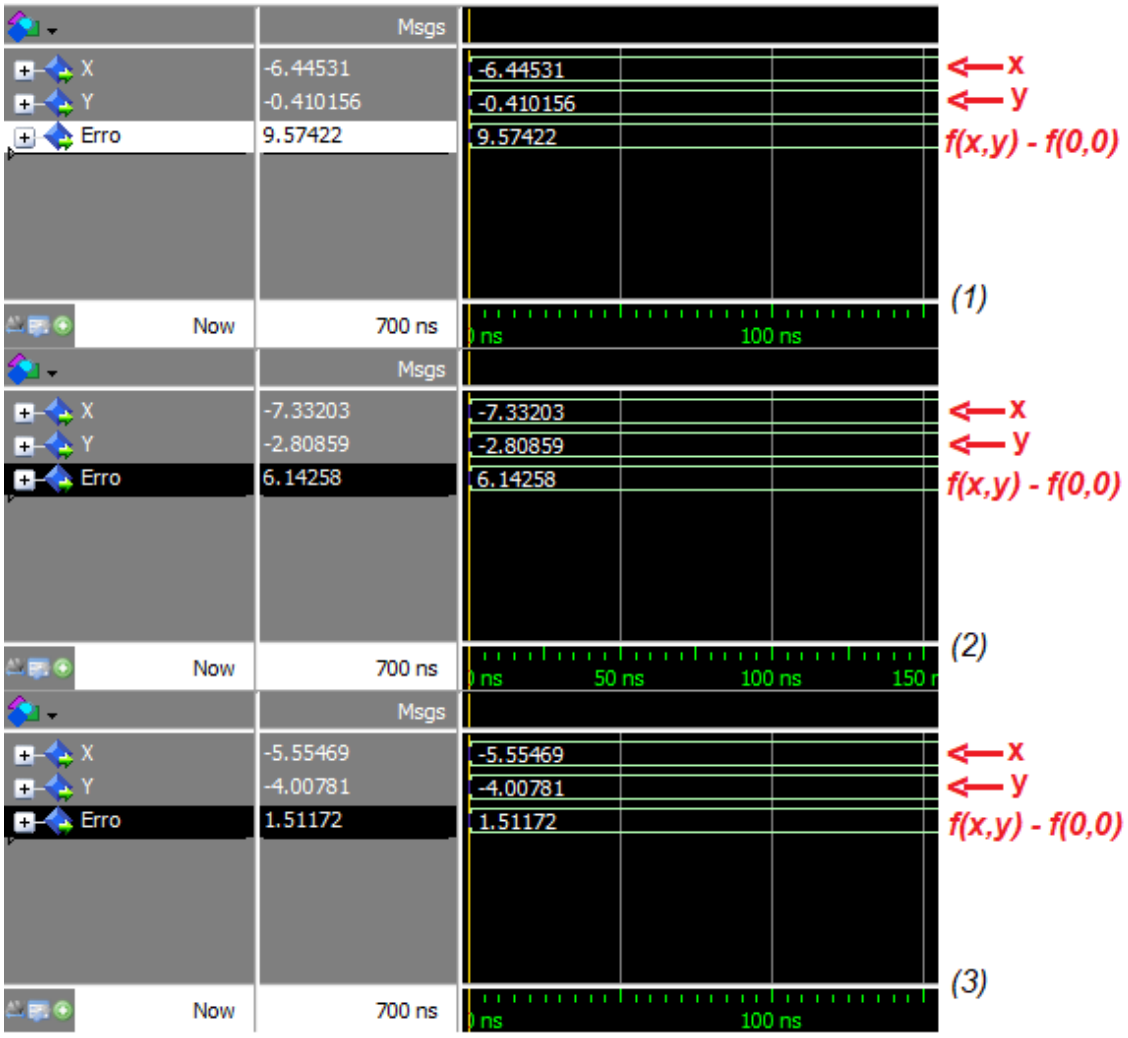
4.2 ALGORITMO DE EVOLUÇÃO DIFERENCIAL

Feita a conversão do código do MATLAB para o VHDL, foi necessário validar se o código gerado no HDL *Coder* corresponde ao código original. Como o algoritmo possui em muitos pontos a presença da aleatoriedade, o modo encontrado para validar o código foi tornar constante, tanto no MATLAB quanto no código em VHDL, esses elementos, para em sequência realizar a comparação entre os resultados esperados em ambas as ferramentas.

Utilizando o software de *design* da Altera para fazer a síntese do código, juntamente com o software Modelsim para fazer a simulação comportamental desse código, foi obtida a figura 8, onde foram apresentados alguns resultados obtidos através de simulações, nos quais não foi considerado os atrasos de propagação do dispositivo. A figura 8 apresenta três resultados de simulações independentes, obtidos através de diferentes populações. Os sinais X e Y da figura representam os valores dos melhores indivíduos obtidos para otimizar a função de Matyas ($f(x, y)$), enquanto que o erro é a diferença entre o valor obtido por esses indivíduos e o valor mínimo da função no ponto (0,0).

Nessa primeira simulação a informação mais relevante a ser analisada é se os valores de resposta obtidos com determinados parâmetros são similares ao valor esperado utilizando o MATLAB nas mesmas condições. Fixado valores como a população, as taxas de cruzamento e mutação, é possível verificar se ocorre a convergência entre os resultados obtidos.

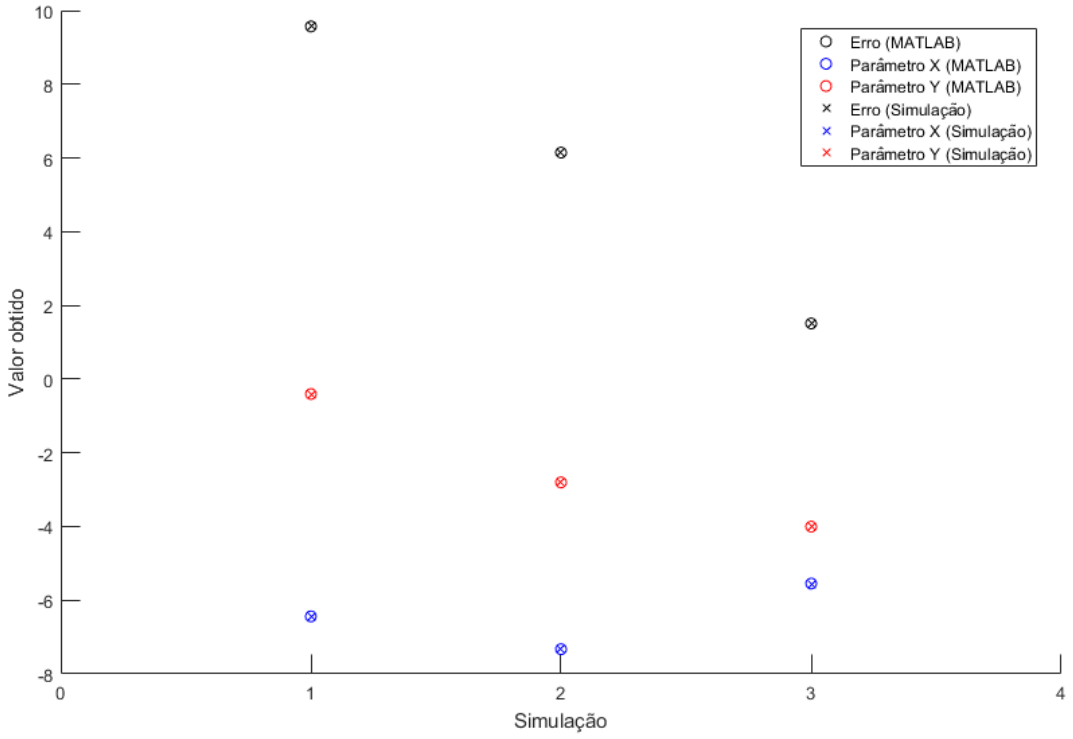
FIGURA 8 – RESULTADOS PARA VALIDAÇÃO DO CÓDIGO GERADO PARA DE



FONTE: AUTOR, 2017

Com esses resultados, é possível realizar uma comparação com os valores obtidos através do MATLAB, aplicando as mesmas condições. Essas comparações são apresentadas na figura 9, onde os círculos no gráfico são os resultados obtidos através do MATLAB e os “x” representam os valores obtidos através das simulações comportamentais realizadas anteriormente. Pode se verificar que os resultados obtidos são praticamente iguais aos esperados, validando o código gerado pelo HDL *Coder*.

FIGURA 9 – COMPARAÇÃO ENTRE RESULTADOS DO MATLAB E SIMULAÇÃO PARA DE



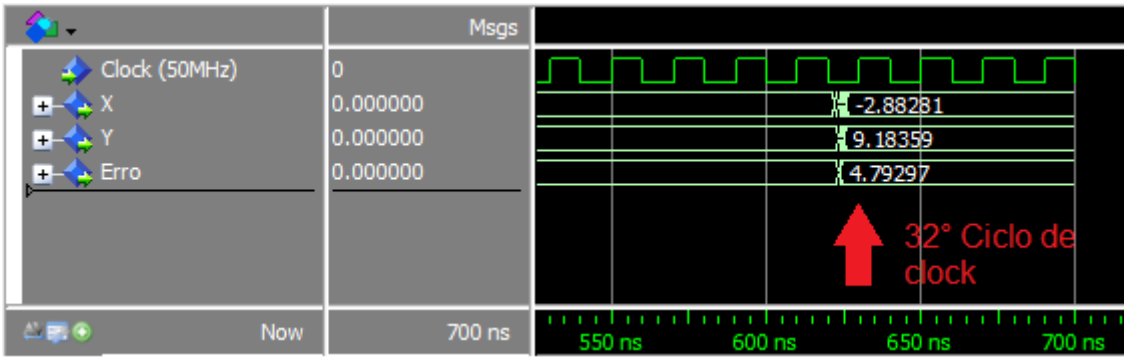
FONTE: AUTOR, 2017

Pequenas diferenças entre os valores obtidos no MATLAB e na simulação comportamental podem ser verificadas em função da quantidade de bits utilizadas para representar os dados. Utilizar dados do tipo *double*, como o MATLAB faz, permitiu uma precisão numérica entre 15 e 16 dígitos, enquanto que utilizando os 14 bits definidos para as saídas essa precisão foi reduzida para no máximo 4 dígitos.

Após a validação, a próxima etapa é inserir as aleatoriedades removidas do algoritmo, utilizando o LFSR para realizar a contagem pseudoaleatória. A partir de agora também será verificado o comportamento esperado no circuito em relação ao tempo. Para isso é utilizada a simulação temporal do circuito, que permite analisar a resposta do circuito considerando os atrasos de propagação do dispositivo. Essa simulação fornece uma resposta mais completa do circuito e permite verificar o número de ciclos de *clock* necessários para obter a resposta do algoritmo.

O resultado dessa simulação temporal pode ser visto na figura 10. Nessa figura percebe-se que são necessários cerca de 32 ciclos de *clock* para que o algoritmo gere todos os valores aleatórios e forneça as respostas obtidas. Considerando o valor do *clock* do *kit* de desenvolvimento de 50 MHz, o tempo estimado para obter a resposta é de cerca de 640 ns.

FIGURA 10 – SIMULAÇÃO TEMPORAL DO ALGORITMO DE

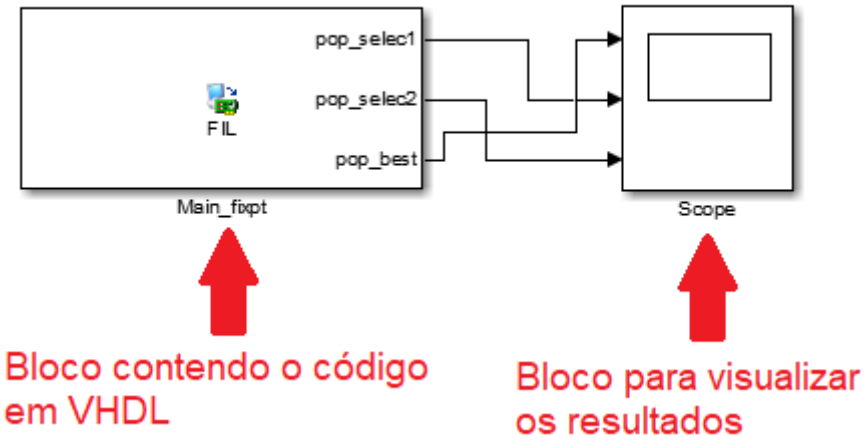


FONTE: AUTOR, 2017

Utilizando a extensão do MATLAB chamada de FPGA-*in-the-loop* (FIL), foi possível fazer a verificação do comportamento do algoritmo utilizando o hardware da FPGA. Nesse projeto foi utilizada a integração possível entre o Simulink e a FPGA, onde um bloco contendo o código em VHDL gerado é utilizado para ser descarregado na FPGA através do Simulink.

A partir da interface do Simulink, foi possível utilizar outros blocos da ferramenta juntamente com o bloco gerado contendo o código em VHDL, como pode ser visto na figura 11, onde é utilizado o bloco “Scope” para verificar as saídas da FPGA.

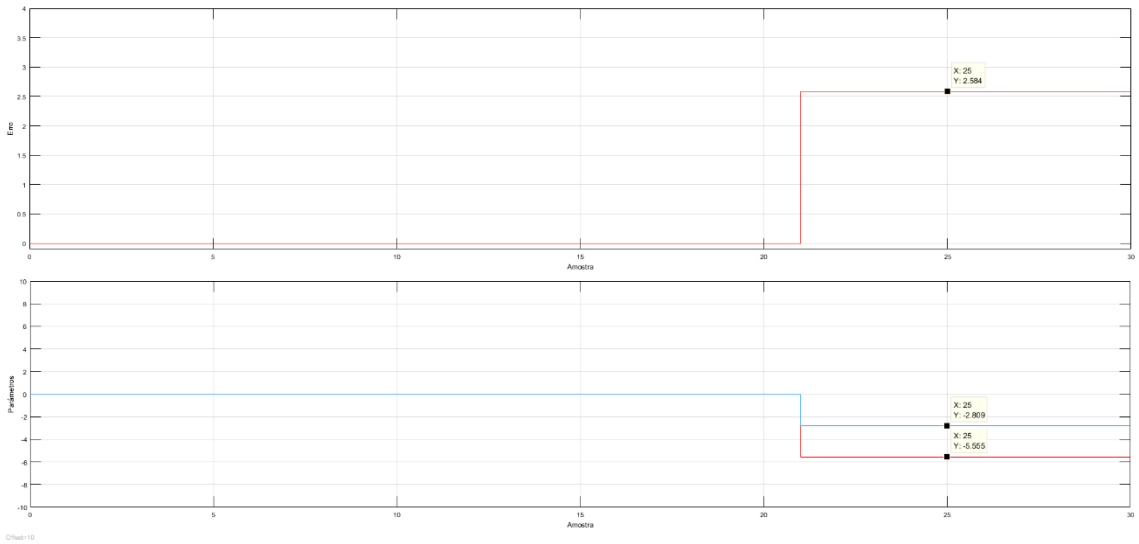
FIGURA 11 – INTEGRAÇÃO DO SIMULINK COM A FPGA



FONTE: AUTOR, 2017

A figura 12 mostra a utilização do FIL para o DE, onde o algoritmo foi executado utilizando o hardware da FPGA e os resultados obtidos pelo Simulink. O resultado apresentado na figura 12 é a simulação utilizando o FIL, e o resultado obtido para os parâmetros x e y da função objetivo definida na equação 3.1 recebem os valores de -2.809 e -5.555 respectivamente, enquanto que o erro encontrado para esses valores tem o valor de 2.584.

FIGURA 12 – RESULTADO DA SIMULAÇÃO UTILIZANDO O FIL PARA DE



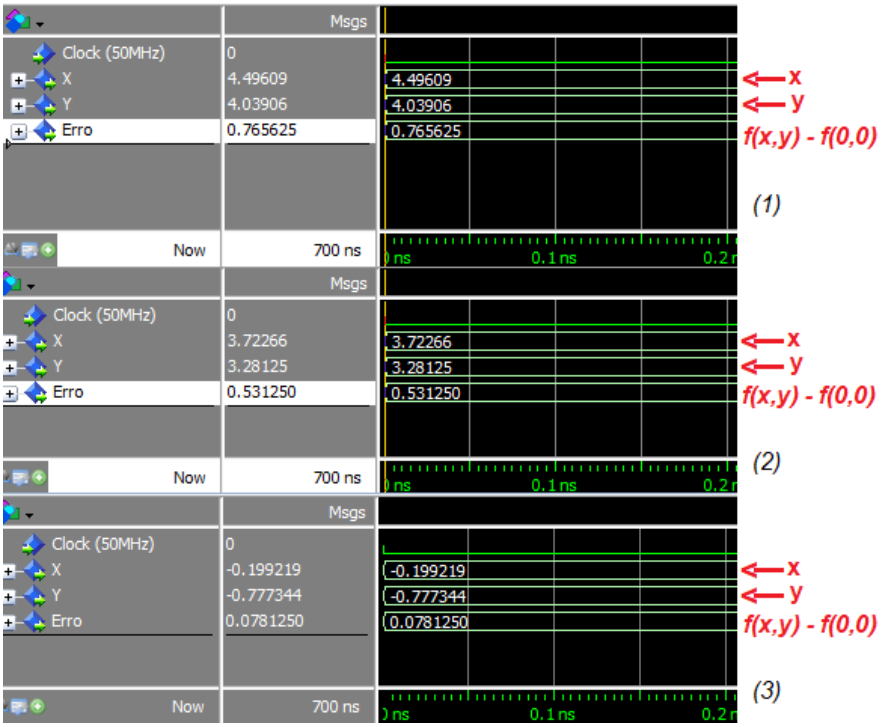
FONTE: AUTOR, 2017

Com essas etapas do fluxo de projeto concluídas, é necessário repeti-las para o Algoritmo de Otimização de Leão, apresentado na próxima seção.

4.3 ALGORITMO DE OTIMIZAÇÃO DE LEÃO

Assim como no algoritmo de Evolução Diferencial, antes de realizar a conversão do código em MATLAB para o VHDL foram necessárias algumas adaptações para permitir o uso da ferramenta. Essas adaptações são feitas apenas para a conversão do algoritmo, sendo corrigidas posteriormente. A validação do código gerado é realizada da mesma forma feita para o algoritmo anterior, onde os valores das funções aleatórias foram fixados, permitindo a comparação de resultados entre o MATLAB e o Modelsim. Foram realizadas diversas simulações para validar o algoritmo, mas na figura 13 são mostrados apenas 3 desses resultados. Mais uma vez se trata de uma simulação comportamental, então será analisado nessa etapa apenas a convergência desses resultados com o MATLAB.

FIGURA 13 – RESULTADOS PARA VALIDAÇÃO DO CÓDIGO GERADO PARA LOA

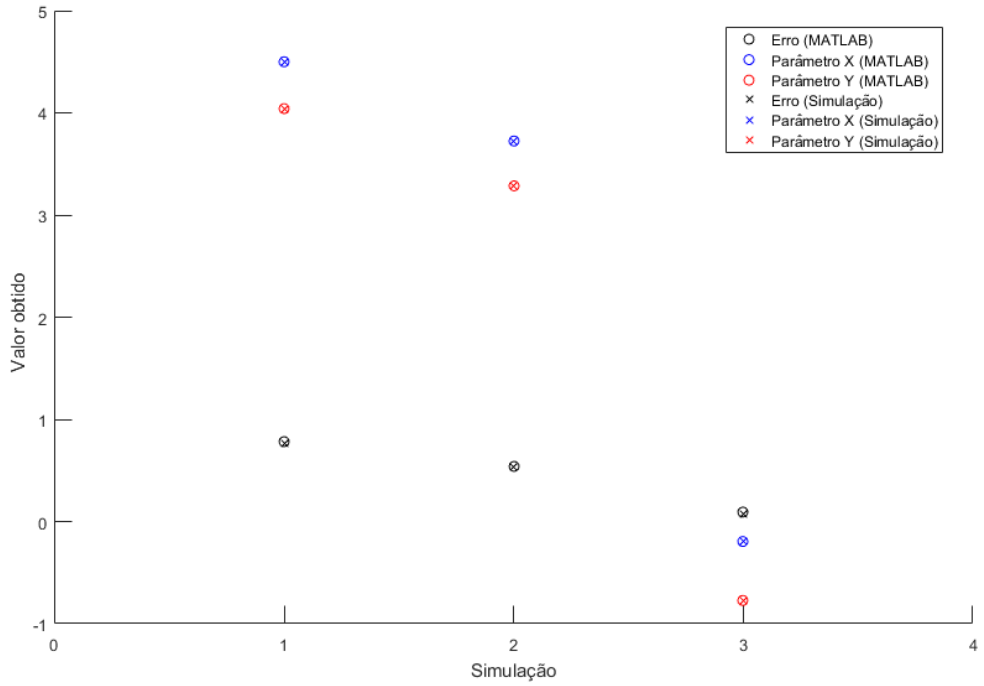


FONTE: AUTOR, 2017

Os sinais X e Y da figura 13 representam os melhores parâmetros encontrados para a minimização da função de Matyas ($f(x,y)$), enquanto que o erro é a diferença calculada da função de Matyas utilizando esses parâmetros em relação ao valor mínimo que ela pode assumir, que ocorre no ponto (0,0).

Com esses resultados, pode-se comparar os resultados obtidos com os esperados no MATLAB, aplicando as mesmas condições em ambos os códigos. O gráfico da figura 14 mostra a diferença entre os valores simulados apresentados na figura 13 e os valores esperados obtidos através do MATLAB, onde verifica-se a convergência dos resultados. Nesse gráfico os resultados marcados como círculos representam a resposta esperada pelo MATLAB, enquanto que os “x” são as respostas obtidas através das simulações. Comparando tanto os parâmetros da função de Matyas encontrados quanto o valor do erro obtido a partir desses parâmetros, percebe-se que os resultados obtidos são bastante próximos dos esperados, validando o código gerado.

FIGURA 14 – COMPARAÇÃO ENTRE RESULTADOS DO MATLAB E SIMULAÇÃO PARA LOA



FONTE: AUTOR, 2017

A tabela 1 apresenta os valores numéricos referentes as simulações realizadas tanto no MATLAB quanto através da simulação utilizando o Modelsim.

TABELA 1 – TABELA CONTENDO OS RESULTADOS DAS SIMULAÇÕES COMPORTAMENTAIS PARA LOA

	Simulação 1			Simulação 2			Simulação 3		
	x	**y**	**Erro**	**x**	**y**	**Erro**	**x**	**y**	**Erro**
MATLAB	4.5	4.041	0.782	3.724	3.286	0.539	-0.196	-0.772	0.092
Modelsim	4.496	4.039	0.766	3.723	3.282	0.531	-0.199	-0.773	0.078

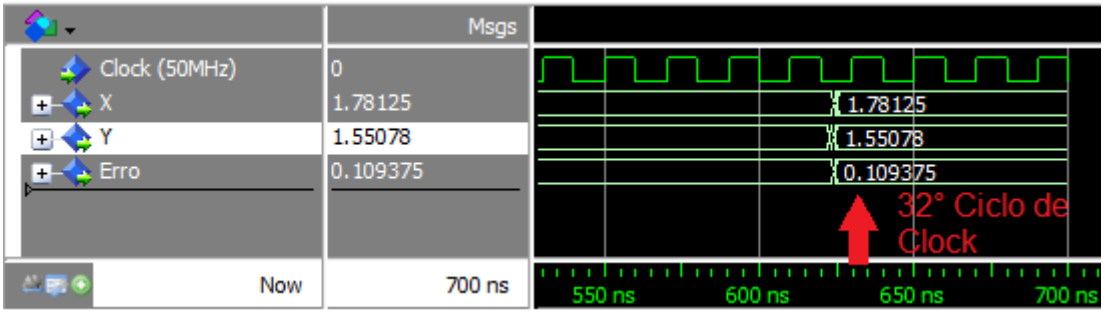
FONTE: AUTOR, 2017

Finalizada a simulação comportamental para validação do circuito e implementadas as variáveis aleatórias no código, deseja-se obter as informações a respeito do tempo de execução do algoritmo com as aleatoriedades do código implementadas.

Entretanto, durante a síntese percebeu-se que, ao tentar aplicar valores aleatórios para as variáveis responsáveis pela seleção, ocorria um erro devido ao limite de memória da máquina utilizada ter sido alcançado. Essa limitação não permitia a síntese do algoritmo, inviabilizando a aplicação. Essa situação foi contornada considerando o critério de seleção fixo. O resultado disso é uma convergência mais lenta para encontrar os pontos ótimos da função objetivo. Os resultados daqui para frente consideram o critério de seleção fixo.

Utilizando a simulação temporal, é possível realizar a estimação do tempo necessário para o algoritmo. A figura 15 mostra a resposta obtida ao longo do tempo para o circuito projetado. As saídas finais do circuito são obtidas após aproximadamente 32 ciclos de *clock*, o que corresponde a aproximadamente 640 ns, definindo a frequência como sendo 50 MHz.

FIGURA 15 – SIMULAÇÃO TEMPORAL DO ALGORITMO LOA

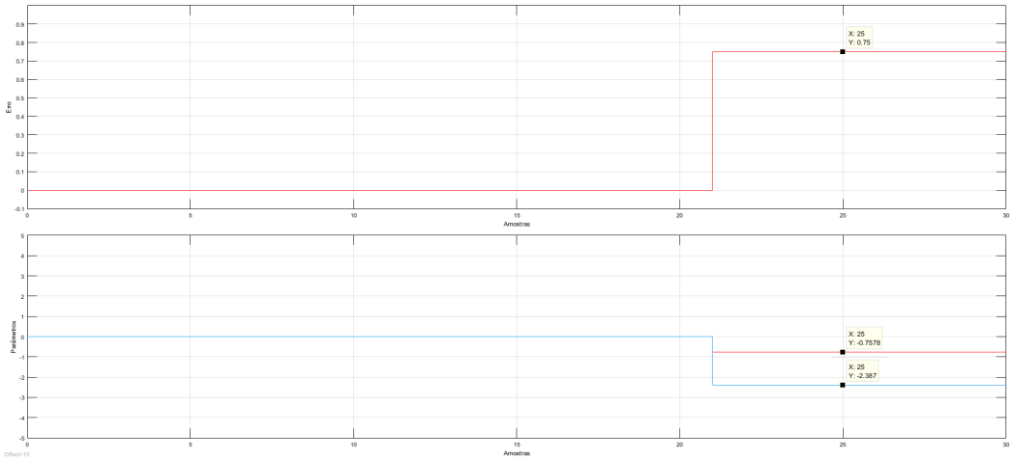


FONTE: AUTOR, 2017

Os resultados com essas simulações, utilizando o software de *design* Quartus juntamente com o Modelsim, são emulações do computador com o objetivo de simular o comportamento da FPGA, mas sem de fato utilizar o hardware da mesma. A extensão FPGA-*in-the-loop* permite utilizar o hardware da FPGA para realizar a execução do algoritmo, enquanto que o computador apenas recolhe as informações de saída.

A próxima etapa do fluxo de projeto é a utilização dessa extensão, onde o resultado obtido com essa ferramenta para o LOA pode ser visto na figura 16. O comportamento em hardware corresponde as simulações, e o valor obtido para o erro nesse caso foi de 0.75, onde os parâmetros x e y da função objetivo definida na equação 3.1 recebem os valores de -2.387 e -0.762 respectivamente.

FIGURA 16 – SIMULAÇÃO UTILIZANDO FIL PARA LOA



FONTE: AUTOR, 2017

Para uma análise mais abrangente dos resultados obtidos, foram realizadas vinte execuções dos algoritmos de DE e LOA, utilizando o mesmo número de iterações (40) e o mesmo tamanho para as populações (30). Após essas simulações foi calculado o valor médio do erro obtido, e comparado com o valor do erro médio obtido através MATLAB. A Tabela 2 mostra os valores de erro mínimos e máximos encontrado durante as execuções tanto no MATLAB quanto utilizando no Modelsim. Nessa tabela ainda estão presentes os valores calculados para o erro médio e o tempo médio necessário para uma execução. Os tempos de execução no MATLAB foram obtidos utilizando dois processadores Intel Core i5 diferentes: o *6400 de 2.7 GHz e **5200U de 2.2 GHz.

TABELA 2 – RESULTADOS MÉDIOS

Algoritmo	***Erro mínimo obtido***	***Erro máximo obtido***	***Erro médio***	***Tempo Médio***
DE (MATLAB)	0.301	27.13	7.111	*4.1 – **7.5 ms
DE (Simulado)	0.547	25.316	8.268	640 ns
LOA (MATLAB)	0.011	0.726	0.187	*5.1 – **8.3 ms
LOA (Simulado)	0.063	0.891	0.286	640 ns

FONTE: AUTOR, 2017

5. CONCLUSÃO

O objetivo desse trabalho foi verificar se era possível implementar em hardware um algoritmo otimização, desenvolvendo um método que possa ser aplicado para diferentes tipos de algoritmos. Para isso, aliou-se várias ferramentas, que permitiram gerar um código em HDL a partir de um outro em uma linguagem de programação.

A geração desses códigos em HDL já se provou uma tarefa que exige bastante atenção. Apesar das ferramentas auxiliarem muito a conversão, muitas considerações devem ser tomadas para isso. Analisar o problema e perceber quais considerações devem ser feitas para determinado código muitas vezes não é uma tarefa fácil, sendo necessário realizar diversas tentativas e propor diferentes soluções, até alcançar o resultado desejado.

Apesar de não ter sido feita a implementação de fato, as diversas simulações realizadas sugerem que a implementação é possível e recomendada quando desejado um baixo tempo de processamento. As simulações temporais realizadas verificaram que o circuito gerado fornece respostas para os algoritmos de forma rápida, enquanto que a utilização do FPGA-*in-the-loop* permitiu verificar o comportamento dos algoritmos em hardware.

Os resultados obtidos utilizando o método proposto nesse trabalho são bastante satisfatórios, considerando as condições de contorno utilizadas. Com eles é possível perceber que o comportamento do circuito desenvolvido é bastante similar aos resultados obtidos em software. O erro médio verificado utilizando o MATLAB é inferior ao erro estimado para o código em VHDL, o que é algo esperado afinal a precisão com que o MATLAB pode trabalhar com seus dados é superior. Em compensação o tempo estimado para a execução em hardware se mostrou inferior na implementação em hardware. Enquanto que nas simulações em software o tempo necessário para as execuções dos algoritmos está na ordem de milissegundos, a resposta em hardware é obtida na ordem de nanosegundos. Essa resposta só é possível devido ao paralelismo que a FPGA permite.

Fica claro que a FPGA pode ser considerada uma boa opção para implementação de algoritmos similares ao de otimização, e que dependendo da complexidade da função objetivo permite até a aplicação em tempo real.

Uma das dificuldades encontradas ao longo do trabalho foi a falta de bibliografias referente a utilização das ferramentas propostas. São poucos os exemplos práticos disponíveis para serem utilizados como referência. Esse trabalho contribui um pouco a mais com o tema, fornecendo uma referência para quem desejar realizar uma abordagem mais prática.

Os próximos estudos incluem a implementação de algoritmos com um maior número de variáveis, para verificar a viabilidade da aplicação em tempo real para funções objetivo de grau mais elevado. Também seria interessante implementar um sistema colaborativo, utilizando mais de uma meta-heurística nos algoritmos.

REFERÊNCIAS

TOSCANI, Simão S.; OLIVEIRA, R. S.; CARISSIMI, A. S. Sistemas Operacionais e Programação Concorrente. 1. ed. [S.l.: s.n.], 2003. 247 p. v. 1. Disponível em: <https://www.inf.pucrs.br/~ramos/sop_progconctext.doc>. Acesso em: 13 nov. 2017.

ALBERTINI, Bruno de Carvalho. A vez da FPGA. O Setor Elétrico**,** [s. L], v. 1, n. 110, p.182-183, 1 mar. 2015. Mensal. Disponível em: <http://www.ieee.org.br/wp-content/uploads/2014/05/Ed110_EspacoIEEE.pdf>. Acesso em: 19 nov. 2017.

GLOBAL MARKET INSIGHTS (Estados Unidos). Field Programmable Gate Array (FPGA) Market Size By Application. 2016. Disponível em: <https://www.gminsights.com/industry-analysis/field-programmable-gate-array-fpga-market-size>. Acesso em: 28 set. 2017.

DAVIES, John. Embedded Eletronic Systems and Microcontrollers: Larger Systems. In: DAVIES, John. MSP430 Microcontroller Basics. USA: Newnes, 2008. cap. 1, p. 4-5. v. 1. Disponível em: <https://books.google.com.br/books?id=qVjhtNYvGGAC&pg=PA5&lpg=PA5&dq=microcontroller+and+fpga+fixed+hardware&source=bl&ots=C9j47vmeS7&sig=7hw2O9tm2au6znJ5hD6b0dVVxIs&hl=pt-BR&sa=X&ved=0ahUKEwjf7-r2lNPXAhUBTJAKHaylAE8Q6AEIazAJ#v=onepage&q=microcontroller%20and%20fpga%20fixed%20hardware&f=false>. Acesso em: 21 nov. 2017.

CASTRO, F.S.; ESKINAZI, R. Design ofan 8051 Microcontroller in a FPGA with a Reconfigurable Instruction Set. Tecnologus**,** Curitiba, v. 5, n. 6, p.1-3, 30 dez. 2011. Disponível em: <http://www.unibratec.edu.br/tecnologus/wp-content/uploads/2012/08/tecnologus_edicao_06_artigo_05.pdf>. Acesso em: 14 out. 2017.

BERGAMINI, Mariane Gavioli; LEANDRO, Gideon Villar. Algoritmo Genético Aplicado em Identificação de Sistemas de Múltiplas Entradas e Múltiplas Saídas. Revista Eletrônica Multidisciplinar: FACEAR, Curitiba, v. 6, n. 1, p.1-10, abr. 2017a. Semestral. <http://www.revista.facear.edu.br/artigo/$/algoritmo-genetico-aplicado-em-identificacao-de-sistemas-de-multiplas-entradas-e-multiplas-saidas-1>. Acesso em: 23 set. 2017.

STORN, R.; PRICE, K. Differential Evolution – A Simple and efficient adaptive schemefor global optimization over continuous spaces. International Computer Science Institute, Berkley, 1995.

RAJAKUMAR, B. R. The Lion’s Algorithm: A New Nature-Inspired Search Algorithm. **Procedia Technology**, v. 6, p. 126–135, 2012. B.R. Rajakumar.

WANG, B.; JIN, X. P.; CHENG, B. Lion pride optimizer: An optimization algorithm inspired by lion pride behavior. **Science China Information Sciences**, v. 55, n. 10, p. 2369–2389, 2012.

HEALY, Dan. Understanding Linear Feedback Shift Registers – The Easy Way. Disponível em: <http://www.yikes.com/~ptolemy/lfsr_web/index.htm>. Acesso em: 07 nov. 2017.

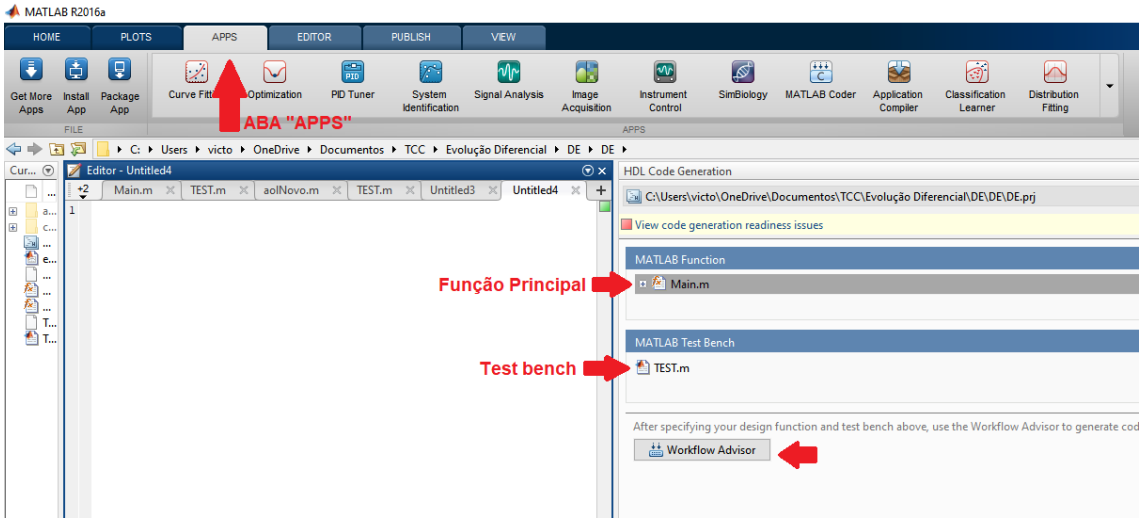
SURJANOVIC, Sonja; BINGHAM, Derek. Matyas Function. Disponível em: <https://www.sfu.ca/~ssurjano/matya.html>. Acesso em: 15 ago. 2017.

BERGAMINI, Mariane Gavioli. Estimação de parâmetros de sistemas não-lineares utilizando um Sistema colaborativo de metaheurísticas**.** 2017b. 72 f. Dissertação (Mestrado) - Curso de Engenharia Elétrica, Tecnologia, Universidade Federal do Paraná, Curitiba, 2017.

APÊNDICE 1 – UTILIZAÇÃO DO HDL *CODER*

A extensão do MATLAB HDL *Coder* pode ser muito útil para realizar a conversão de códigos do MATLAB para uma das linguagens de descrição de hardware disponíveis (VHDL ou Verilog). A primeira etapa ao utilizar essa extensão é selecionar a ferramenta na aba “APPS” do MATLAB e criar um novo projeto. Nesse projeto deverá ser definido o arquivo principal e o *test bench* utilizado. O arquivo principal é a função que se deseja converter em um circuito, enquanto que o *test bench* é um arquivo externo que fornecerá as entradas para essa função. O botão “*Workflow Advisor*” iniciará a ferramenta. Essas etapas podem ser vistas na figura 17.

FIGURA 17 – INICIO DA UTILIZAÇÃO DO HDL *CODER*



FONTE: AUTOR, 2017

Após iniciar a ferramenta, a primeira tela que irá surgir será para definir os tipos de entradas. Deve-se selecionar a opção de ponto-fixo, pois a conversão com ponto flutuante não é sintetizável. Em seguida é necessário apertar o botão “*Run*”, para chegar até a opção “*Fixed-Point Conversion*”, marcada na figura 18 pelo número 1. Aqui serão convertidos os pontos flutuantes em pontos fixos e serão definidos o número de bits necessários para representar as variáveis do projeto. Essa etapa é muito importante para o projeto, pois definir tamanhos

incorretos para as variáveis podem gerar aumentos significativos no erro das saídas ou uma utilização de hardware muito alta.

A primeira estimativa pode ser feita selecionado a opção "*Log data from histogram*" e pressionando "*Run Simulation*" (marcada pelo número 2 na figura 18), o que habilita uma ferramenta para auxiliar na verificação dos tamanhos propostos para as variáveis. Essa ferramenta está marcada com o número 3 na figura 18, e permite visualizar os *status* de cobertura das variáveis. Após rodar a simulação, o campo "*Proposed Types"* da aba "*Variables"* será preenchido com os tipos propostos para as variáveis, baseados nessa simulação utilizando o *test bench*.

Esses tipos propostos através da simulação podem ser modificados manualmente, o que pode acabar sendo necessário caso o *test bench* não cubra todas as condições possíveis, ou se a simulação propor tamanhos de variáveis maiores do que o necessário. Os tamanhos das variáveis definidos nessa etapa serão implementados em hardware, portanto deve-se ter cuidado com o número de bits definidos para elas, para que os limites do hardware não sejam ultrapassados.

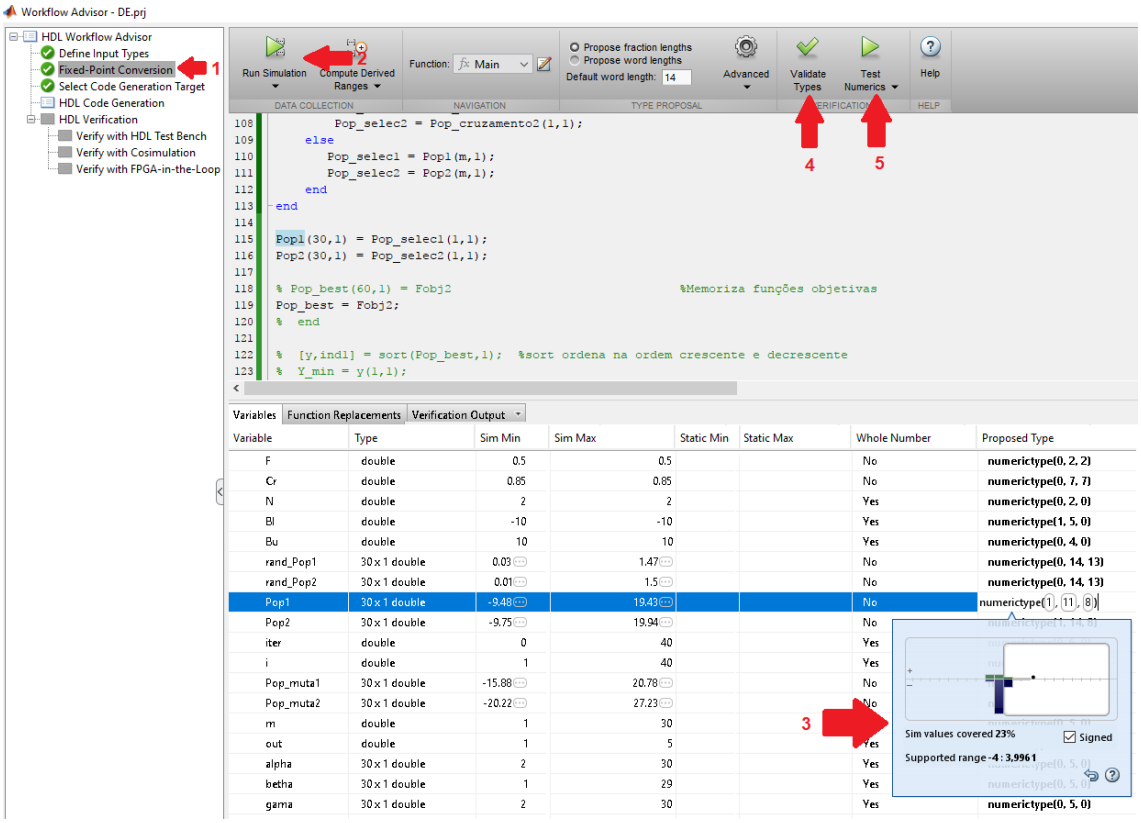
Nessa etapa é definido, por exemplo, o tamanho das variáveis (população, taxa de cruzamento, taxa de mutação) sendo representadas por 14 bits. Esse valor foi obtido realizando testes numéricos que serão explicados em seguida, de forma que esse resultado foi encontrado empiricamente, onde nesses casos, o erro gerado definindo as variáveis com esse tamanho era baixo e não era utilizado um número muito elevado de bits para isso.

Uma vez definidos os tipos das variáveis, é necessário fazer a validação dessas (marcado pelo número 4 na figura 18), para então partir para a geração do código em HDL. Existe ainda uma etapa opcional para testar a resposta do código com os tipos de dados definidos. Selecionado a opção "*Log inputs e outputs for comparison plots*" no botão "*Test Numerics"* (na figura 18 marcada pelo número 5) é possível obter valiosas informações sobre a resposta da função, utilizados os tipos de dados definidos.

Essa opção permite verificar as diferenças esperadas entre o código original com ponto flutuante e o código com ponto fixo. Um exemplo desses

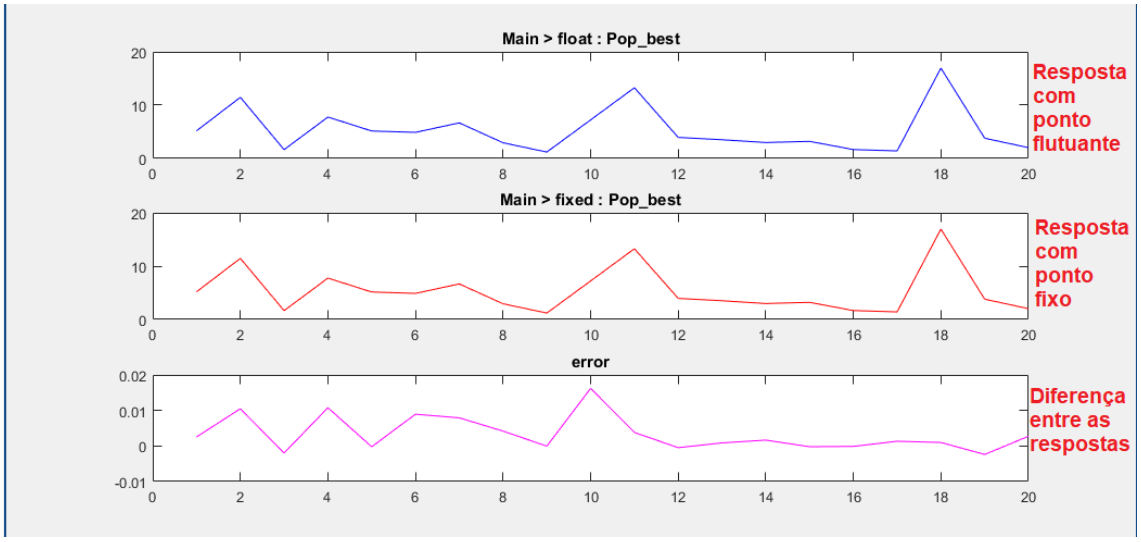
resultados pode ser visto na figura 19, onde as saídas com ponto flutuante e fixo são comparadas e o erro dessa conversão é fornecido para as saídas. Com essa ferramenta é possível realizar modificações nos tipos de dados até encontrar uma resposta que utilize um número razoável de bits e que gere um erro pequeno nas saídas.

FIGURA 18 – DEFINIÇÃO DO TIPO DE VARIÁVEIS



FONTE: AUTOR, 2017

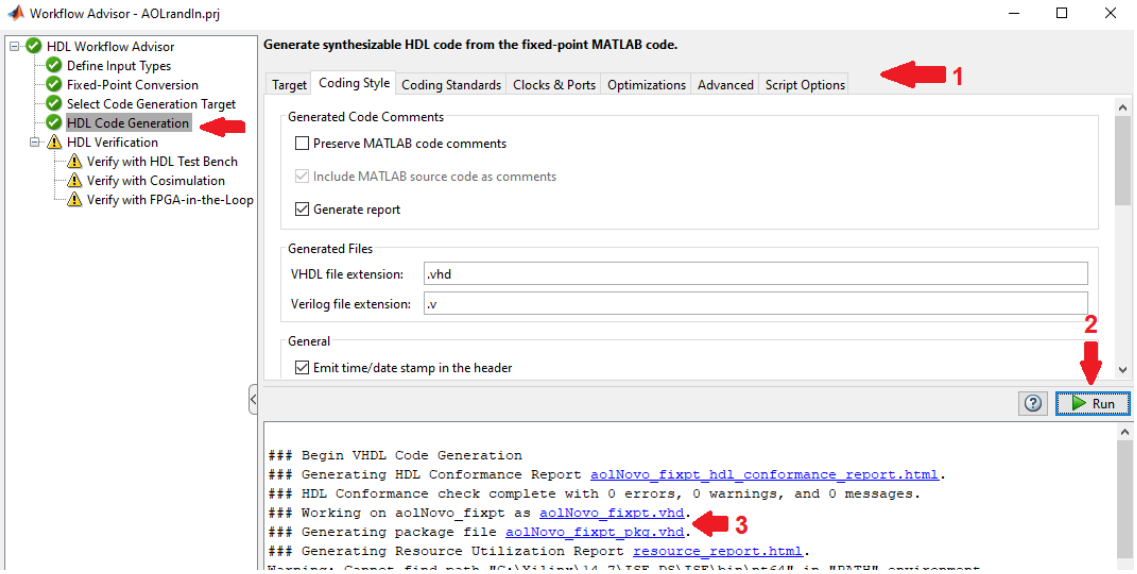
FIGURA 19 – EXEMPLO DE INFORMAÇÕES DE ERRO FORNECIDAS PELA FERRAMENTA



FONTE: AUTOR, 2017

Após definido o ponto fixo e o tipo das variáveis, pode se gerar o código na HDL desejada. Em "*HDL Code Generation*" é possível escolher entre várias opções para a geração do código (mostradas na figura 20 pelo número 1). Essas opções são definidas conforme as necessidades do código. Após configuradas as opções, clica-se no botão "*Run*" (marcado com número 2 na figura 20) para obter a conversão do código. Se todas as configurações tiverem sido realizadas com sucesso, uma mensagem informando que o código juntamente com os *packages* necessários foram gerados com sucesso (mostrado pelo número 3 da figura 20).

FIGURA 20 – TELA DE OPÇÕES PARA GERAÇÃO DO CÓDIGO EM HDL



FONTE: AUTOR, 2017