UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DAINF - DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ANDRÉ LUIZ LEMOS GUERRA

IMPLEMENTANDO EM FPGA UM ALGORITMO GENÉTICO PARA A BUSCA DE SOLUÇÕES PARA O PROBLEMA DO CAIXEIRO VIAJANTE

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2021

ANDRÉ LUIZ LEMOS GUERRA

IMPLEMENTANDO EM FPGA UM ALGORITMO GENÉTICO PARA A BUSCA DE SOLUÇÕES PARA O PROBLEMA DO CAIXEIRO VIAJANTE

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação da Universidade Tecnológica Federal do Paraná, como requisito parcial para a obtenção do título de Bacharel.

Orientador: Professor MSc. André Macário Barros
Universidade Tecnológica Federal do Paraná

PATO BRANCO
2021

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Pato Branco
Curso de Engenharia de Computação

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

Às 20 horas e 20 minutos do dia 18 de maio de 2021, reuniu-se de forma online a banca examinadora composta pelos professores André Macário Barros (orientador), Marco Antônio de Castro Barbosa e Dalcimar Casanova para avaliar o trabalho de conclusão de curso com o título **Implementando em FPGA um Algoritmo Genético para a busca de soluções para o Problema do Caixeiro Viajante**, do aluno **André Luiz Lemos Guerra**, matrícula 1809725, do curso de Engenharia de Computação. Após a apresentação o aluno foi arguido pela banca examinadora. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

_____
André Macário Barros
Orientador (UTFPR)

_____
Prof. Dalcimar Casanova
(UTFPR)

_____
Prof. Marco Antonio de Castro Barbosa
(UTFPR)

_____
Prof. Pablo Gauterio Cavalcanti
Coordenador de TCC

_____
Prof. Ives Renê Venturini Pola
Coordenador do Curso de
Engenharia de Computação

AGRADECIMENTOS

Agradeço ao meu orientador André Macário, por me inspirar a realizar um projeto deste tamanho e por todo o suporte e apoio que foi dado durante a realização deste trabalho.

Ao professor Marco Barbosa, por apresentar e instigar meu amor pelo mundo da otimização e das meta-heurísticas.

À todos os meus professores, do ensino superior, médio e fundamental que foram de grande importância para que eu tivesse esse amor pelo conhecimento e pelo estudo.

À toda minha família, que colaborou direta e indiretamente nesta jornada, em especial meus pais, Maria e Antônio, que muitas vezes deram mais do que poderiam para garantir que eu pudesse crescer; ao meu irmão e irmãs, que estiveram ao meu lado todos esses anos; à minha madrinha, pelo apoio nesta jornada educacional e emocional; à minha avó Maria, que foi de grande importância na minha construção pessoal.

À minha namorada Vanessa, que me ajudou a seguir em frente nos momentos difíceis e tornou meus dias cinzas, um pouco mais ensolarados.

Aos colegas de faculdade, de diversos cursos, que me marcaram durante meu caminho; em especial ao meu grande amigo André Alessi, por tornar esta experiência mais agradável e me ajudar nos piores momentos; ao Gustavo e ao Natanael que faziam meus dias mais divertidos e não me deixavam perder o foco do que era importante; aos meus colegas de classe que tornaram essa busca pelo conhecimento ainda melhor. Além das muitas outras amizades que fiz e que com certeza lembrarei com carinho.

Por fim agradeço aos meus amigos de fora da faculdade, que marcaram a minha vida, sejam os do colégio; os meus amigos de jogos; até mesmo meus amigos do trabalho; todos vocês tem um espaço muito grande no meu coração.

São as perguntas que não podemos responder que mais nos ensinam. Eles nos ensinam como pensar. Se você der uma resposta a um homem, tudo o que ele ganha é um pequeno fato. Mas faça uma pergunta e ele buscará suas próprias respostas. (ROTHFUSS, Patrick, 2011).

RESUMO

GUERRA, André Luiz Lemos. Implementando em FPGA um algoritmo genético para a busca de soluções para o problema do caixeiro viajante. 2021. 34 f. Trabalho de Conclusão de Curso – Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Pato Branco, 2021.

O Problema do Caixeiro Viajante que busca encontrar a rota de menor custo entre diversas cidades de um conjunto, passando por cada uma delas apenas uma vez e retornando à cidade inicial. Existem diversas aplicações para este problema tais como planejamento de processos, manufatura celular, gerenciamento de rotas, estrutura de matrizes e até mesmo montagem de placas de circuito impresso. Encontrar uma solução exata é custoso em tempo e processamento, para isso existem métodos aproximativos. Contudo algumas aplicações do PCV tem uma necessidade de respostas em questão de microssegundos. Nessas situações o paralelismo é uma forte opção para aceleração. Dentro dos métodos aproximativos a meta-heurística Algoritmo Genético se destaca, por sua capacidade de paralelização.
Neste trabalho foi implementada uma arquitetura reconfigurável em FPGA, que representa a meta-heurística algoritmo genético, capaz de buscar soluções para o problema do caixeiro viajante. A implementação ocorreu em um microcontrolador combinado a blocos lógicos reconfiguráveis e foi realizada em um *kit* FPGA *Spartan 3E*. Por fim, foram realizados experimentos que comprovaram um crescimento linear do tempo de execução conforme se aumentou o número de cidades do problema.
Palavras-chave: Algoritmos Genéticos. Problema do Caixeiro Viajante. FPGA. Otimização.

ABSTRACT

GUERRA, André Luiz Lemos. Implementing in FPGA a genetic algorithm to search for solutions to the traveling salesman problem. 2021. 34 f. Trabalho de Conclusão de Curso – Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Pato Branco, 2021.

The Traveling Salesman Problem that seeks to find the lowest cost route between several cities in a set, going through each one only once and returning to the starting city. There are several applications for this problem such as process planning, cell manufacturing, route management, matrix structure and even the assembly of printed circuit boards. Finding an exact solution is costly in time and processing, for which there are approximate methods. However, some PCV applications have a microsecond response requirement. In these cases, parallelism is a strong option for acceleration. Within the approximate methods, the metaheuristic Genetic Algorithm stands out for its parallelization capacity.
In this work, a reconfigurable architecture in FPGA was implemented, which represents the meta-heuristic genetic algorithm, capable of seeking solutions to the traveling salesman problem. The implementation took place in a microcontroller combined with reconfigurable logic blocks and was carried out in a kit FPGA Spartan 3E. Finally, experiments were conducted which proved a linear increase runtime as the number of cities in the problem increased.

Keywords: Genetic Algorithms. Traveling Salesman Problem. FPGA. Optimization.

LISTA DE FIGURAS

Figura 1 – Representação Cromossomial de um Indivíduo 7
Figura 2 – Diagrama de fluxo de um algoritmo genético 8
Figura 3 – Diagrama de máquina de estados para algoritmo do mínimo divisor comum 11
Figura 4 – Modelo simplificado de FSM (para *hardware*) 11
Figura 5 – Modelo de Implementação do Sistema Proposto 14
Figura 6 – Modelo de Implementação dos Blocos Lógicos 16
Figura 7 – Descrição Visual da Operação de Mutação 18
Figura 8 – Descrição Visual da Operação de Reprodução 19
Figura 9 – Interface do XPS 21
Figura 10 – Interface do SDK 22
Figura 11 – Gráfico de Média do Tempo de Execução (em μs) por Número de Cidades 26

LISTA DE QUADROS

Quadro 1 – Algoritmo Rodando na Interface do Tera Term 24
Quadro 2 – Gráfico de Média do Tempo de Execução (em μs) por Número de Cidades 25

LISTA DE ABREVIATURAS E SIGLAS

FPGA	Field Prgrammable Gate Array
FSM	Finite State Machine
PCV	Problema do Caixeiro Viajante
PSO	Particle Swarm Optimization
VHDL	VHSIC Hardware Description Language
VHISC	Very High-Speed Integrated Circuit
XPS	Xilinx Platform Studio
SDK	Xilinx Software Development Kit

SUMÁRIO

1 – INTRODUÇÃO 1
1.1 OBJETIVO GERAL 2
1.2 OBJETIVOS ESPECÍFICOS 3
1.3 ORGANIZAÇÃO DO TRABALHO 3

2 – FUNDAMENTAÇÃO TEÓRICA 4
2.1 COMPLEXIDADE COMPUTACIONAL 4
2.1.1 PROBLEMA DO CAIXEIRO VIAJANTE 5
2.2 HEURÍSTICAS E META-HEURÍSTICAS 6
2.2.1 ALGORITMOS GENÉTICOS 6
2.2.1.1 MODELAGEM 6
2.2.1.2 ALGORITMO 7
2.2.1.3 POPULAÇÃO INICIAL 8
2.2.1.4 OPERADORES GENÉTICOS 8
2.2.1.5 PARALELISMO EM ALGORITMO GENÉTICO 9
2.3 ACELERAÇÃO DE CÓDIGOS EM FPGAS 9
2.3.1 FPGA 9
2.3.2 VHDL 9
2.3.3 CONVERSÃO DE CÓDIGO 10
2.3.4 CALCULO DE ACELERAÇÃO 12
2.4 TRABALHOS RELACIONADOS 12

3 – MATERIAIS E MÉTODOS 14
3.1 MATERIAIS 14
3.2 SISTEMA IMPLEMENTADO 14
3.2.1 SISTEMA PARALELO 15

4 – DESCRIÇÃO DA IMPLEMENTAÇÃO 18
4.1 IMPLEMENTAÇÃO VHDL 18
4.1.1 MUTAÇÃO 18
4.1.2 REPRODUÇÃO 19
4.1.3 XOROSHIRO128+ 20
4.2 COMPONENTES DO MICROBLAZE 20
4.2.1 XPS Timer 20
4.2.2 RS232 DCE 20
4.3 INTEGRAÇÃO COM O MICROBLAZE 20

4.4 CÓDIGO EM C 21
4.4.1 GERAÇÃO INICIAL 22
4.4.2 REPRODUÇÃO 23
4.4.3 MUTAÇÃO 23
4.4.4 CALCULO DE FITNESS 23
4.4.5 CONDIÇÃO DE PARADA 23
4.5 EXECUÇÃO 24
4.6 EXPERIMENTOS COMPUTACIONAIS 25
4.7 DISCUSSÕES 26

5 – CONCLUSÃO 27
5.1 PRINCIPAIS DIFICULDADES 27
5.2 TRABALHOS FUTUROS 27

Referências 28

Apêndices 31

APÊNDICE A – Código VHDL de Mutação 32

APÊNDICE B – Código VHDL de Reprodução 33

1 INTRODUÇÃO

Na era da tecnologia, a busca pela minimização de custos e maximização de lucros é constante. Uma empresa de transportes marítimos que busca encontrar a menor rota que passe por todos os portos de uma região ou uma empresa de entregas que deseja consumir o mínimo de combustível possível ao entregar seus produtos aos clientes, são exemplos práticos de problemas de otimização. Trazendo essas informações para o mundo computacional, pode se dizer que essas empresas buscam uma solução para um problema já bastante conhecido.

O Problema do Caixeiro Viajante (PCV) do inglês *Traveling Salesman Problem* consiste em encontrar a rota de menor custo entre diversas cidades de um conjunto, passando por cada uma delas apenas uma vez e retornando para à cidade inicial. O PCV é um dos problemas de otimização mais estudados na área de pesquisa operacional. Apesar de sua definição ser relativamente simples, o PCV, ainda representa um dos maiores desafios da pesquisa operacional (LAPORTE, 1992).

Por se tratar de um problema de otimização, suas aplicações não são limitadas à busca de rotas. São encontrados usos para o PCV em problemas de escalonamento, como por exemplo os que acontecem nos sistemas operacionais dos computadores atuais, manufatura celular, em que há um agrupamento de máquinas em sequencia sob o controle de uma célula central e até mesmo no projeto de design de placas de circuito impresso (PUNNEN, 2007).

Contudo, a busca pela melhor solução não é trivial, uma vez que a quantidade de possíveis soluções cresce de maneira fatorial em relação ao número de cidades ou portos que se deseja visitar.

Segundo Garey e Johnson (1979) ao fazer uso da técnica de força bruta, isto é, verificar todas as soluções possíveis, um computador que demore 1 microssegundo para processar uma operação, com um conjunto de 10 cidades demorará pouco mais de 1 segundo para encontrar a resposta, porém para 40 cidades este número facilmente ultrapassa os 3855 séculos.

Devido a esta realidade, o uso de técnicas e algoritmos com tempo computacional polinomial mas que não garantem a solução ótima do problema, tem se tornado cada vez mais comum. Estes algoritmos são denominados heurísticos ou aproximativos (GOLDBARG, 2005).

Dentro das heurísticas existe um subgrupo, as meta-heurísticas. Estas representam uma família de técnicas de otimização aproximativa, que proveem soluções "aceitáveis" e em tempo polinomial, para problemas difíceis ou complexos, como o PCV (TALBI, 2009). Por aceitável se entende uma solução que não seja a melhor possível mas muito próximo desta, para um contexto comercial o aceitável, representa um solução que não proveem o máximo de lucros possível mas oferece uma melhora significativa.

Uma destas meta-heurísticas são os Algoritmos Genéticos, cujas técnicas de busca de repostas são baseadas em teorias evolucionistas tais como a seleção, a reprodução e a mutação (LINDEN, 2012).

Embora as meta-heurísticas busquem reduzir ou eliminar os problemas das buscas heurísticas, essas podem consumir um grande valor de tempo (OCHI et al., 2002). Isso se deve à natureza do funcionamento das meta-heurísticas, que se baseia em técnicas de tentativa e erro com apoio de inteligencia computacional.

Para um problema como a minimização de rotas, uma empresa de logística, não teria problemas em aguardar algumas horas por uma solução. Porém algumas das aplicações do PCV, como a de fabricação de eletrônicos de precisão, exigem um tempo de resposta na casa dos microssegundos. A aceleração proveniente da paralelização é uma possível resposta para alguns desses casos.

As duas principais etapas do Algoritmo Genético, a mutação e a reprodução, são independentes entre si e também entre as soluções em uma população, possibilitando serem implementadas de forma paralela. Implementações paralelas permitem solucionar problemas maiores ou encontrar soluções melhores, com respeito a formas sequenciais, devido à partição do espaço de buscas (possíveis soluções do problema) e há mais possibilidades de intensificação e diversificação de busca. Isso torna o paralelismo não apenas uma forma de reduzir o tempo de execução de algoritmos de busca e meta-heurísticas como também melhorar sua eficiência e robustez (CUNG et al., 2002).

O paralelismo pode ser implementado de diversas formas. Em *software* é possível serem criados diferentes processos em um mesmo programa, cada um realizando uma tarefa diferente, mas sua capacidade de escalonamento é limitada. Enquanto que em *hardware* é possível serem projetadas arquiteturas capazes de realizar a mesma natureza das operações com uma quantidade muito maior de parcelas em paralelo.

São diversas as opções de escolha para paralelismo em *hardware*, tais como um processador com diversos núcleos, uma placa gráfica com múltiplos núcleos cuda ou um *Field Programmable Gate Array* (FPGA), que nada mais é, que um arranjo de portas lógicas programáveis. Devido a natureza da arquitetura dos FPGA, a implementação paralela não é limitada pelo número de núcleos.

Arquiteturas em *hardware* podem ser inicialmente implementadas usando linguagens de descrição de *hardware* como a *VHSIC Hardware Description Language* (VHDL). Esse tipo de linguagem permite que o circuito seja sintetizado e simulado antes de qualquer implementação física. Depois de escrito e simulado, o código pode ser usado para implementação física do circuito em um *chip* do tipo FPGA (PEDRONI, 2010).

1.1 OBJETIVO GERAL

Desenvolver e implementar em uma arquitetura de *hardware* em FPGA da meta-heurística algoritmo genético fazendo-se uso dos princípios do paralelismo para buscar soluções para o Problema do Caixeiro Viajante.

1.2 OBJETIVOS ESPECÍFICOS

- Projetar uma arquitetura paralela para a meta-heurística algoritmo genético em VHDL.
- Implementar a arquitetura proposta em um kit de desenvolvimento FPGA.

1.3 ORGANIZAÇÃO DO TRABALHO

No capítulo 1 é realizada a apresentação do problema, o objetivo geral e os objetivos específicos do trabalho. No capítulo 2 são apresentados todos os aspectos teóricos que se julgam necessários para o entendimento deste trabalho. O capítulo 3 apresenta os materiais e a metodologia que foram utilizados para realizar o objetivo proposto. Na seção 4 é apresentada a implementação, execução da arquitetura e também os experimentos realizados. Por fim a seção 5 apresenta as conclusões sobre o trabalho, dificuldades encontradas e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os conteúdos considerados essenciais para a realização da solução proposta. Na Seção 2.1 as questões como complexidade e intratabilidade são apresentadas, além da formulação e apresentação do PCV e também uma breve apresentação sobre heurísticas e meta-heurísticas. A Seção 2.2.1 apresenta informações sobre os Algoritmos Genéticos. Na Seção 2.3 é realizada a apresentação dos conceitos de FPGA e VHDL além de um método de transformação de código. Por fim na Seção 2.4 são apresentados e comparados alguns trabalhos relacionados ao tema.

2.1 COMPLEXIDADE COMPUTACIONAL

Para descrever melhor o problema se faz necessário uma breve compreensão de complexidade de algoritmos e intratabilidade.

A função de complexidade de tempo de um algoritmo determina o tempo requerido, para cada possível tamanho de entrada, o maior tempo necessário para que o algoritmo resolva uma instância do problema daquele tamanho. Diferentes algoritmos possuem diferentes funções de complexidade de tempo, algumas podem ser classificadas como eficientes ou não eficientes dependendo da situação.

Na teoria de computação existe uma divisão entre funções de complexidade de tempo que são os algoritmos de tempo polinomial e os algoritmos de tempo exponencial.

Um algoritmo de tempo polinomial é definido por $O(p(n))$ em que p representa uma função polinomial e n representa o tamanho da entrada do problema. Deve ser dito também que essa definição inclui certas complexidades de tempo não polinomiais como n^{logn} que não são tratadas como exponenciais. Qualquer algoritmo cuja função de complexidade de tempo não se encaixe nesta notação é considerado um algoritmo de tempo exponencial (GAREY; JOHNSON, 1979).

Geralmente os algoritmos de tempo exponencial são buscas exaustivas pela resposta enquanto os algoritmos de tempo polinomial são possíveis quando se possui um entendimento mais profundo do problema. Existe um consenso de que um problema não é dito "resolvido" até que um algoritmo de tempo polinomial seja conhecido. Com isso em mente pode se definir um problema intratável como um problema que é tão difícil que nenhum algoritmo de tempo polinomial consiga resolver (GAREY; JOHNSON, 1979).

Um problema é dito NP-Completo quando sua complexidade é não polinomial e é possível a redução em tempo polinomial a outro problema NP-Completo ou ao problema de satisfatibilidade booleana dito o "primeiro" problema NP-Completo e que é verificável em tempo polinomial (COOK, 1971).

2.1.1 PROBLEMA DO CAIXEIRO VIAJANTE

> "O problema do caixeiro viajante é um problema de otimização associado ao problema da determinação de caminhos hamiltonianos em um grafo qualquer. O objetivo do PCV é encontrar, em um grafo [...] o caminho hamiltoniano de menor custo." (GOLDBARG, 2005, p.331).

Um caminho é dito hamiltoniano quando se passa por todos os vértices de um grafo apenas uma vez, e ao fim se retorna ao grafo inicial. Realizando permutações para encontrar o número de soluções possíveis se encontra a complexidade do problema $O(n!)$.

O problema é considerado intratável por Garey e Johnson (1979) e foi classificado NP-Completo por Karp (1975). Uma das formulações mais aceitas na literatura é a formulação de Dantzig-Fulkerson-Johnson (DFJ) presente em Dantzig, Fulkerson e Johnson (1954) que formula o PCV como um grafo $G = (N,A)$, em que N são os vértices que representam as cidades e A são as arestas que representam as rotas, dadas as seguintes restrições:

$$z = \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} x_{ij} \tag{1}$$

$$sujeito\ a:$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in N \tag{2}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in N \tag{3}$$

$$\sum_{i,j \in S}^{n} x_{ij} \leq |S| - 1 \quad \forall S \subset N \tag{4}$$

$$x_{ij} \in \{0,1\} \quad \forall i,j \in N \tag{5}$$

Busca-se minimizar o valor de Z. O valor de $c_{i,j}$ representa a distância entre duas cidades. A variável x_{ij} pode assumir os valores 0 e 1, sendo 1 quando o arco(i,j)$\in A$ for escolhido como parte da solução e 0 quando não fizer parte, as variáveis i e j representam a cidade de origem e a cidade de termino do arco. S é um subgrafo de G, e $|S|$ representa o número de vértices desse subgrafo. Nesta formulação se assume implicitamente que x_{ij} não existe e que existem $n(n-1)$ variáveis inteiras e $O(2^n)$ restrições.

> "Esta formulação destaca um importante aspecto do PCV que é sua natureza combinatória. Pela formulação fica claro que solucionar um PCV é determinar uma certa permutação legal de custo mínimo" (GOLDBARG, 2005, p.333).

Dada a intratabilidade do PCV são aplicados métodos aproximativos como os que são mostrados na próxima seção.

2.2 HEURÍSTICAS E META-HEURÍSTICAS

Com base no conteúdo da seção 2.1.1, os métodos aproximativos das heurísticas e meta-heurísticas apresentam uma boa alternativa para o problema do caixeiro viajante.

A ideia de métodos aproximados, que são fáceis de se usar mas não garantem a melhor solução não é nova. O nome heurística deriva da palavra grega *heuriskien* que significa descobrir. Atualmente o termo é usado para descrever um método que, baseado em experiência ou julgamento, parece entregar uma boa solução, mas que não tem garantia de ser a ótima (FOULDS, 1984).

Uma meta-heurística pode ser definida como uma estratégia de busca, não específica para um problema, que tenta de maneira eficiente explorar o espaço de soluções. São algoritmos aproximativos que incorporam mecanismos para evitar cair em máximos ou mínimos locais (BECCENERI, 2008).

Na literatura são encontradas diversas meta-heurísticas, como o Recozimento Simulado, Colonia de Formigas, Busca Tabu dentre elas o Algoritmo Genético.

2.2.1 ALGORITMOS GENÉTICOS

Algoritmos Genéticos são parte de um grupo de algoritmos chamados evolucionários que servem como ferramenta para buscas e otimizações. Eles não precisam de informações sobre o espaço de busca, apenas uma função objetivo ou valor de *fitness* relacionada à uma solução. Essa característica torna este processo de otimização, robusto e muito atrativo para soluções de larga escala em sistemas não lineares (COSTA et al., 2007).

> "Os algoritmos evolucionários funcionam mantendo uma população de estruturas, denominadas indivíduos ou cromossomos, operando sobre esta de forma semelhante à evolução das espécies. A essas estruturas são aplicados os chamados operadores genéticos, como recombinação e mutação, entre outros. Cada indivíduo recebe uma avaliação que é uma quantificação da sua qualidade como solução do problema em questão. Com base nesta avaliação são aplicados os operadores genéticos de forma a simular a sobrevivência do mais apto. Os operadores genéticos consistem em aproximações computacionais de fenômenos vistos na natureza, como a reprodução sexuada, a mutação genética e outros [...]" (LINDEN, 2012, p.44).

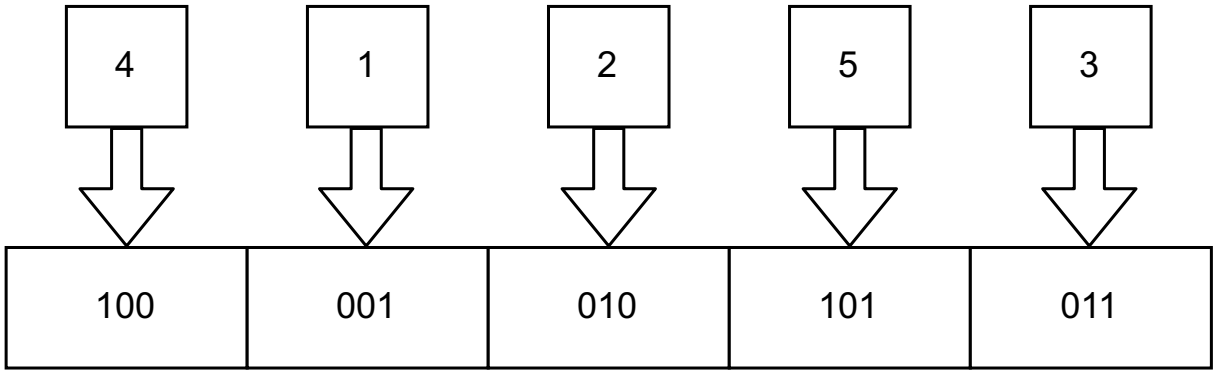
2.2.1.1 MODELAGEM

Um dos pontos mais importantes no algoritmo genético é a representação da informação. Quanto mais adequada ao problema, melhor serão as respostas encontradas (LINDEN, 2012).

Para o caso do problema do caixeiro viajante a solução deve ser uma rota que passe por todas as cidades. Se existem 5 cidades uma solução válida seria começar pela cidade 4 e então passar pelas cidades 1,2,5 e por fim a cidade 3.

A Figura 1 apresenta a representação cromossomial da solução apresentada no paragrafo anterior. Os número das cidades é representado em codificação binária e a posição representa a ordem em que as cidades serão visitadas.

Figura 1 – Representação Cromossomial de um Indivíduo



Fonte: Autoria Própria.

Esta representação pode ser denominada *string* de *bits*, "espaços" que podem assumir apenas valores de 0 ou 1. Fazendo um comparativo com a biologia, um cromossomo pode ser representado pelo código genético de uma solução, o genótipo é a forma em que a *string* está estruturada e por fim o fenótipo representa a interação do código genético com o ambiente.

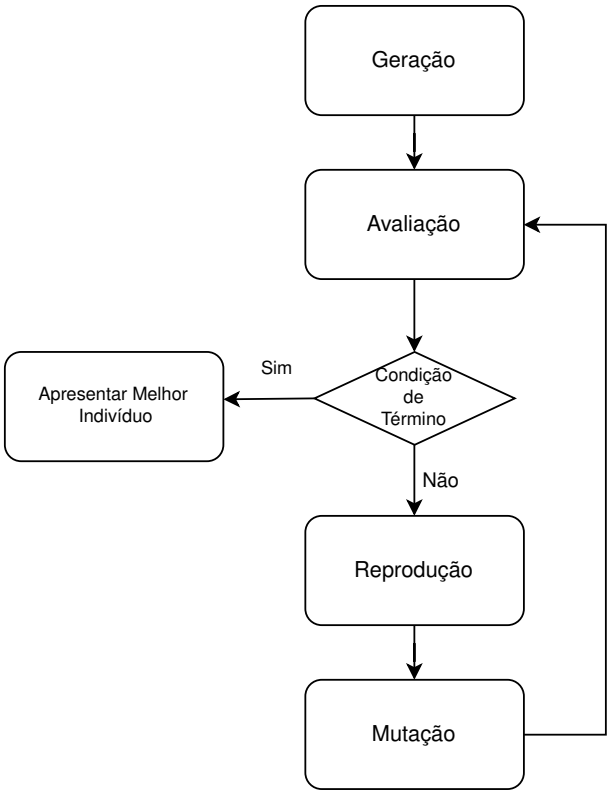
2.2.1.2 ALGORITMO

O algoritmo genético começa com uma população de indivíduos que podem ser representados por *strings* de *bits* e subsequentemente produz populações sucessivas. As mecânicas de um algoritmo genético são surpreendentemente simples, envolvendo nada mais complexo do que copiar *strings* e realizar permutações entre *strings* (GOLDBERG, 1989).

A Figura 2 apresenta o diagrama de fluxo de um algoritmo genético. Gerar população é a primeira operação a ser realizada e não acontece mais de uma vez por execução do algoritmo. Logo após, o primeiro cálculo de *fitness* é realizada a verificação da condição de parada, caso não seja atendida a condição o algoritmo prossegue.

Os operadores genéticos agem nos indivíduos selecionados, esta seleção pode ser baseada no valor de *fitness* como na reprodução ou aleatória que ocorre na mutação. Alguns operadores atuam em dois indivíduos ao mesmo tempo como a reprodução e outros em apenas um como a mutação. A população é reavaliada e os indivíduos com baixa aptidão ou muito antigos são removidos. Por fim a condição de parada é verificada novamente, a condição pode ser um número máximo de iterações, um número máximo de gerações sem mudança na melhor reposta ou tempo total de execução do código.

Figura 2 – Diagrama de fluxo de um algoritmo genético



Fonte: Autoria Própria.

2.2.1.3 POPULAÇÃO INICIAL

> "A inicialização da população, na maioria dos trabalhos feitos na área, é feita da forma mais simples possível, fazendo-se uma escolha aleatória independente para cada indivíduo da população inicial. A lei das probabilidades sugere que teremos uma distribuição que cobre praticamente todo o espaço de soluções, mas isso não pode ser garantido, pois a população tem tamanho finito [...]" (LINDEN, 2012, p.44).

2.2.1.4 OPERADORES GENÉTICOS

A reprodução é um processo em que o material genético de alguns indivíduos é copiado de acordo seu valor de *fitness*, copiar o material genético de acordo com seus valores aumenta as chances de obter melhores resultados nas populações sucessoras. Este operador é uma versão artificial da seleção natural de Darwin (GOLDBERG, 1989).

A seleção simula o mecanismo de seleção natural que atua na natureza, os indivíduos mais aptos sobrevivem mais e conseguem gerar mais filhos enquanto os menos aptos sobrevivem por menos tempo e consequentemente geram menos descendentes (LINDEN, 2012).

A mutação é necessária porque, mesmo que a seleção e reprodução realizem uma busca efetiva, esses operadores podem ocasionalmente se tornar repetitivos e perder algum

material genético com ótimo potencial de solução (GOLDBERG, 1989).

2.2.1.5 PARALELISMO EM ALGORITMO GENÉTICO

A programação paralela nada mais é que um conjunto de processos trabalhando simultaneamente para resolver um dado problema. Logo o paralelismo pode ser dito como uma decomposição da carga de trabalho computacional e distribuição das tarefas. Uma das estratégias mais simples de paralelismo geralmente se utiliza do modelo mestre-escravo, em que um programa "mestre" executa sozinho um controle sequencial do algoritmo e despacha as tarefas de maior custo computacional para os "escravos" realizarem.(GENDREAU; POTVIN, 2010)

Em comparação com o Algoritmo Genético pode relacionar os blocos de mutação e reprodução aos "escravos" e o sistema de controle e armazenamento da população ao "mestre". Devido a natureza intrínseca dos Algoritmos Genéticos poderem funcionar de maneira paralela, essas são ótimas candidatas para a aceleração em *hardware*.

2.3 ACELERAÇÃO DE CÓDIGOS EM FPGAS

Para explicar melhor como é realizada a aceleração em *hardware* do algoritmo genético é necessária uma breve explicação de alguns conceitos como FPGA e VHDL.

2.3.1 FPGA

Um *field programmable gate array* (FPGA) é um dispositivo lógico programável que possui um vetor bidimensional de células lógicas genéricas e chaves programáveis. Uma célula lógica pode ser configurada para realizar uma função simples e as chaves podes ser configuradas para realizar interconexões entre as células lógicas (CHU, 2008). A configuração dos blocos lógicos implementados em FPGA é realizada utilizando a linguagem de descrição de *hardware* VHDL.

2.3.2 VHDL

VHDL é a sigla para VHISC (*very high-speed integrated circuit*) *hardware description language*. Criada para descrever e modelar sistema digitais em vários níveis e é uma linguagem extremamente complexa (CHU, 2008).

O código escrito em VHDL descreve o comportamento ou a estrutura de um circuito eletrônico, que pode ser transformado em um circuito físico inferido por um compilador. Suas principais aplicações incluem a síntese de circuitos digitais em FPGA e o *layout* de circuitos integrados. VHDL permite a síntese de circuitos e também sua simulação. A primeira se trata de transformar o código fonte em uma estrutura de hardware que implementa a funcionalidade desejada enquanto que a segunda se refere a um procedimento para verificar e garantir que a

implementação realmente efetue a funcionalidade desejada pelo circuito sintetizado (PEDRONI, 2010).

2.3.3 CONVERSÃO DE CÓDIGO

Levando em consideração um código de computador em linguagem C que implementa o algoritmo genético de maneira sequencial, pode se realizar a conversão deste código para uma estrutura de *hardware*.

A conversão de um código para *hardware* não é direta, é necessário um entendimento do algoritmo e das limitações e funcionamento do hardware para se obter uma conversão eficiente. Um método é a conversão para máquina de estados finitos (*finite state machine* - FSM) (PEDRONI, 2010).

A abordagem do método da máquina de estados finitos é muito útil quando o funcionamento do circuito pode ser descrito por meio de uma lista definida e não muito longa, contendo todos os estados de um sistema, junto com as condições necessárias para o sistema efetuar as trocas de estado, assim como os valores que devem ser gerados de saída em cada estado (PEDRONI, 2010).

O algoritmo 1 apresenta um pseudocódigo para encontrar o valor do mínimo divisor comum de dois números a e b, os valores de a e b são inseridos a partir de das entradas a_en e b_en. Enquanto o valor de a for diferente do valor de b o é executada uma verificação de qual dos dois valores é maior, se a for maior seu novo valor é $a - b$, se b for maior o novo valor de b é $b - a$. No momento em que a e b tiverem o mesmo valor, o laço enquanto termina e a variável de saída mdc recebe o valor de a. Por fim a variável fim recebe o valor 1 e o programa se encerra.

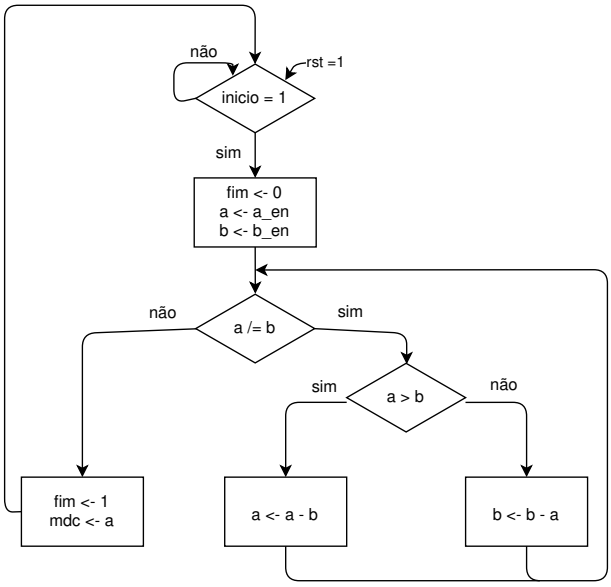
Algoritmo 1: Fonte: Estendido de (D'AMORE, 2012, p.258)

Entrada: valor do número $a(a\_en)$, valor do número $b(b\_en)$
Saída: valor do mínimo divisor comum(mdc)
enquanto *inicio* $= 1$ **faça**
    $fim \leftarrow 0$
    $a \leftarrow a\_en$
    $b \leftarrow b\_en$
    enquanto $a \neq b$ **faça**
        se $a > b$ **então**
            $a \leftarrow a - b$
        senão
            $b \leftarrow b - a$
        fim
    fim
    $mdc \leftarrow a$
    $fim \leftarrow 1$
fim

A figura 3 apresenta um modelo de máquina de estados proposto por D'Amore (2012)

para o algoritmo 1. A entrada rst é responsável por dar partida ao sistema, é feita a verificação do valor de início e caso o valor seja 1 o estado se altera. São realizadas as atribuições $fim = 0$, $a = a\_en$, $b = b\_en$ e então é realizada verificação de condição. Caso o valor de a seja diferente de b é realizada uma nova verificação e os valores das variáveis são alterados, quando os valores forem iguais a variável fim é colocada em 1 e o valor de retorno de a é atribuído a mdc.

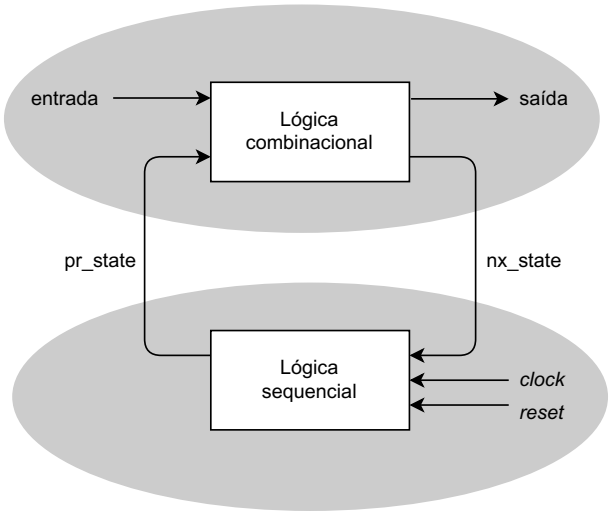
Figura 3 – Diagrama de máquina de estados para algoritmo do mínimo divisor comum



Fonte: Estendido de D'Amore (2012, p.263)

Considerando *hardware*, na Figura 4 é ilustrado um modelo simples para uma FSM. A seção inferior representa a lógica sequencial, que é dependente de *clock* e a seção superior representa a lógica combinacional, que é independente de *clock*.

Figura 4 – Modelo simplificado de FSM (para *hardware*)



Fonte: Estendido de Pedroni (2010, p.350).

O valor armazenado no presente momento representam o sinal *pr_state* (*present state*), enquanto que os valores que serão armazenados na próxima transição do clock representam o sinal *nx_state* (*next state*). A seção superior (lógica combinacional) é responsável por processar o sinal de *pr_state* e da entrada do circuito, e responde com *nx_state* e a saída do sistema (PEDRONI, 2010).

2.3.4 CALCULO DE ACELERAÇÃO

Para que se possa verificar a diferença de tempo entre implementações em *software* e implementações em *hardware*, foi necessária a criação de uma métrica, o *speedup*. A Lei de Amdahl define o calculo máximo de *speedup* esperado em um sistema implementado quando apenas uma fração desse sistema é implementado de maneira paralela (AMDAHL, 1967).

O calculo de *speedup*, é apresentado na Equação 6. A variável p representa como o número de processadores operando na execução da tarefa, enquanto a variável s se refere à porcentagem da tarefa que foi paralelizada.

$$Speedup = \frac{1}{(1-p) + \frac{p}{s}} \tag{6}$$

Por se tratar de uma tecnologia que não utiliza núcleos e sim blocos lógicos, para o calculo de *speedup* em FPGAs, a Lei de Amdahl pode ser generalizada conforme a Equação 7. Em que $Tseq$ representa o tempo de execução do programa de maneira sequencial, enquanto $Tpar$ se refere ao tempo de execução do programa com a aceleração proveniente do paralelismo (STALLINGS, 2010).

$$Speedup = \frac{Tseq}{Tpar} \tag{7}$$

2.4 TRABALHOS RELACIONADOS

Em Huang (2014), o autor apresenta uma implementação paralela em FPGA da meta-heurística *Particle Swarm Optimization* (PSO) e sua aplicação no planejamento de percurso global para robôs autônomos navegando em ambientes estruturados com obstáculos. Segundo o autor a implementação paralela permitia um maior controle sobre a diversidade da população e inibia a convergência prematura em comparação com a implementação convencional do PSO. Este trabalho também implementou a meta-heurística paralela em FPGA apenas com outro foco de aplicação.

No trabalho apresentado por Jaen-Cuellar et al. (2015), os autores propõem e implementam uma arquitetura baseada em FPGA da meta-heurística Algoritmos Genéticos com conceito de micro-população para buscar otimização de parâmetros em controle. Este trabalho fez uso do algoritmo genético porém sem a restrição de população e com um foco em outro problema.

Os autores Nedjah e Mourelle (2014), propõem e descrevem uma arquitetura completamente implementável em *hardware* para o algoritmo genético sem problema específico. O trabalho descreve que os cálculos de *fitness* são realizados em *hardware* com o auxilio de redes neurais para o cálculo dos pesos. Segundo os autores a implementação completa em *hardware* possui desempenho extremamente melhor que a implementação puramente em *software* e é muito mais rápida que a implementação híbrida (*hardware* e *software*). Este trabalho também implementou a meta-heurística Algoritmo Genético em *hardware* (FPGA) contudo a implementação foi realizada com propósito específico e com o auxilio de um microprocessador *softcore* que integra os componentes lógicos.

3 MATERIAIS E MÉTODOS

Neste capítulo são descritos os materiais e métodos necessários para implementação de uma arquitetura para FPGA que represente a meta-heurística algoritmo genético com suas etapas implementadas em linguagem de descrição de *hardware*

3.1 MATERIAIS

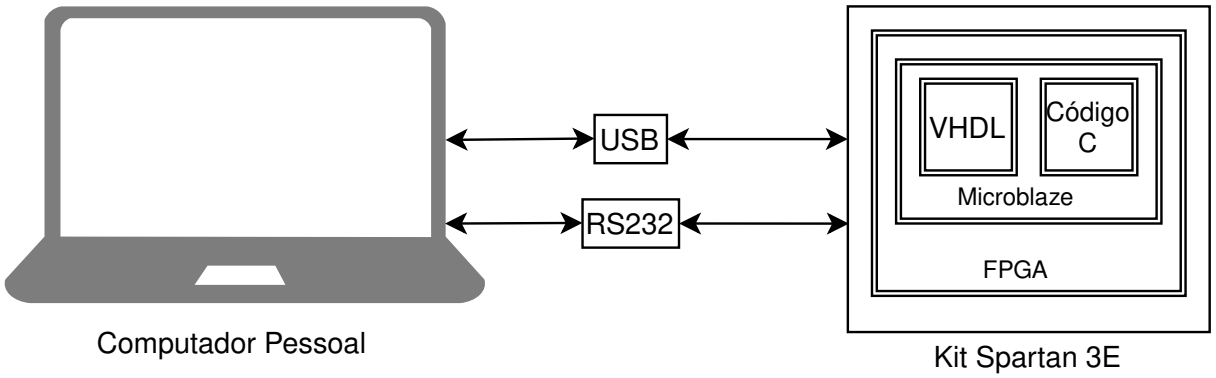
Foram utilizados os seguintes materiais:

- Kit FPGA *Spartan 3E Starter Kit* da *Xilinx*
- Cabo USB para serial RS232 e cabo USB para JTAG
- Ferramenta de síntese e análise ISE Design Suit 14.7
- *Software* de design de *hardware Xilinx Project Studio* (XPS)
- Ambiente de desenvolvimento *Xilinx Software Development Kit* (SDK)
- Emulador de terminal Tera Term
- Computador pessoal
- Biblioteca de instâncias TSPLIB

3.2 SISTEMA IMPLEMENTADO

O modelo geral da implementação é ilustrado na figura 5. Na figura os dois elementos principais são o computador pessoal e a *Spartan 3E Starter Kit* da *Digilent* (XILINX, 2011), a conexão entre estes é realizada por dois cabos, um USB para RS232, o outro USB para JTAG.

Figura 5 – Modelo de Implementação do Sistema Proposto



Fonte: Autoria Própria.

No *kit*, há um quadrado representando o FPGA, dentro deste há um outro que representa o *Microblaze* (XILINX, 2013b), um microprocessador *softcore* que pode ser instanciado dentro do *kit* FPGA para uso geral. O *Microblaze* recebe em sua implementação um código na linguagem C (na figura o quadrado "Código C"), que descreve sua rotina de funções e

sua interação com os blocos lógicos implementados em VHDL (descritos na figura como o quadrado "VHDL"). Uma das conexões de dados do *Microblaze* com o computador é realizada por um cabo USB-Serial RS232, contudo para que esses dados sejam lidos e interpretados corretamente no computador, é necessário um programa de emulador de terminal, o Tera Term (T.TERANISHI, 2004).

No computador, o Ise Design Suit 14.7 (XILINX, 2009), foi o *software* utilizado para o desenvolvimento dos blocos lógicos VHDL, esta ferramenta oferece todo o suporte para a escrita do código VHDL, implementação e a realização das simulações e testes.

Uma vez que os blocos estejam funcionando da maneira esperada, estes são utilizados no *software Xilinx Project Studio* (XILINX, 2013a), é pela interface do XPS que é realizada a descrição das conexões entre os componentes disponíveis dentro da FPGA, os blocos lógicos e o *Microblaze*. Realizada esta descrição, um arquivo *bitstream* é carregado, por meio do cabo USB-JTAG, no *kit Spartan 3E* para que o *hardware* descrito seja "montado" internamente.

Também no XPS é realizada a exportação de um arquivo de descrição do *hardware*, um "*framework*", contendo todos os *drivers* necessários para a implementação de um código em linguagem C que será carregado no *Microblaze*. O *framework* quando utilizado no *Xilinx Software Development Kit* (XILINX, 2013a), permite ao usuário ter acesso aos registradores de todos os componentes que foram instanciados e conectados, com essa lista de endereços, é possível implementar um código em C que pode fazer uso das funções dos blocos lógicos implementados.

As instâncias de teste para realização das implementações serão retiradas de (REINELT, 2000), um acervo de instâncias do PCV disponibilizada *online* de maneira gratuita.

3.2.1 SISTEMA PARALELO

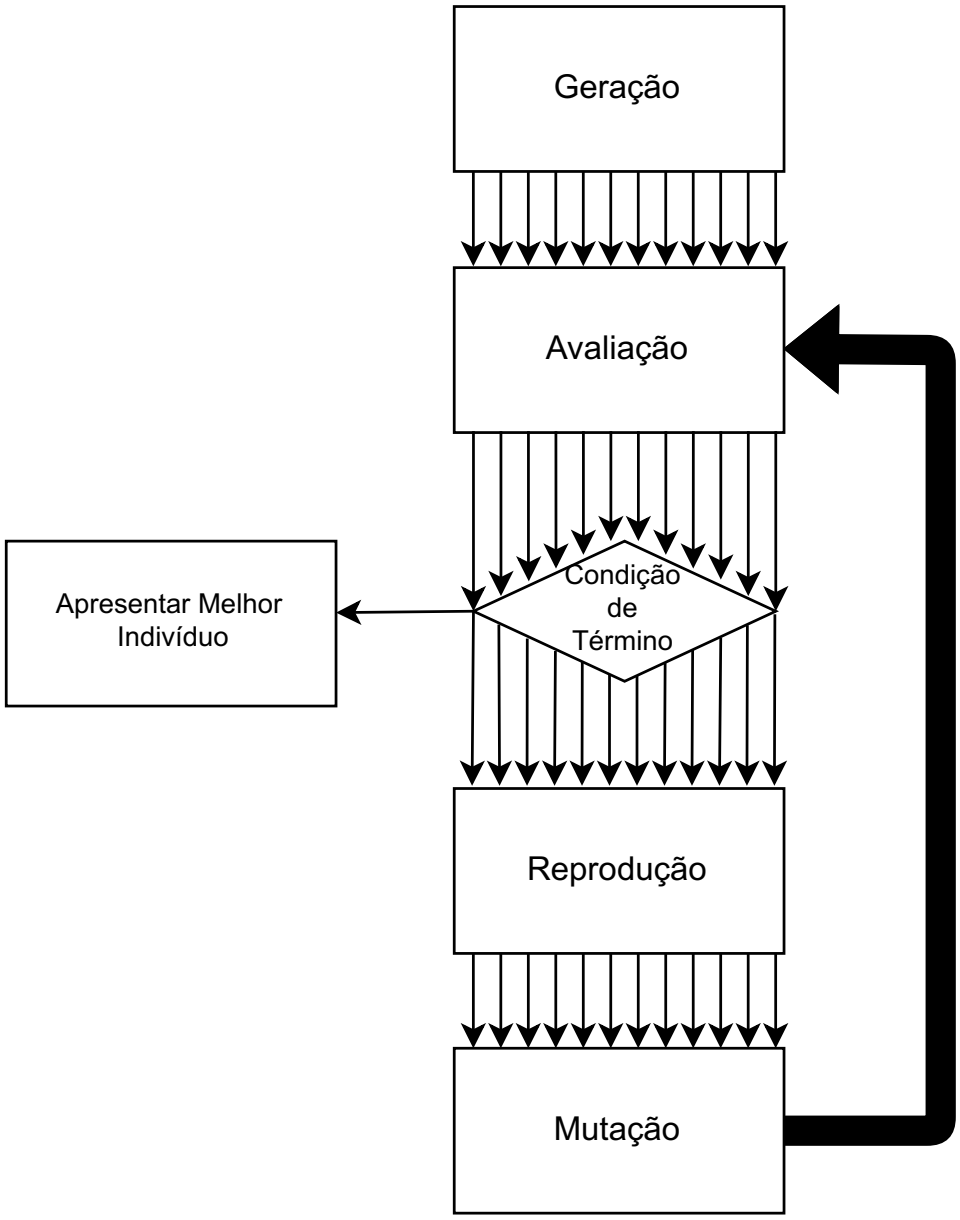
Foram implementados blocos lógicos cada um responsável pela operacionalização do algoritmo genético que buscará soluções do PCV. Tais blocos são regidos em seu funcionamento pelo mesmo microprocessador *softcore* apresentado na subseção 3.1, o *MicroBlaze*, entretanto este componente será responsável pelo especificamente pelo gerenciamento, cabendo aos blocos lógicos o processamento referente ao algoritmo genético.

Na figura 6 é mostrado o modelo para implementação. O conjunto de setas pretas que saem do bloco de geração e se conectam a avaliação, condição de término, reprodução, mutação e reavaliação em sequência, representam os diversos indivíduos presentes na população atual, seu formato é descrito na seção 2.2.1.1. A grande seta preta representa a nova população que é enviada para o bloco de verificação de condição. Foram omitidos na imagem sinais auxiliares como *clock*, sinal de partida, sinal de encerramento, sinal de término de operação entre outros.

Em um instante t_0 o bloco lógico de geração de população terá gerado um número n de indivíduos com características aleatórias, estes indivíduos são então enviados para um bloco lógico de avaliação. Após um tempo, no instante t_1, o bloco de avaliação "devolve" os

indivíduos com seus valores de *fitness* e no tempo t_2 é finalizada a verificação de término.

Figura 6 – Modelo de Implementação dos Blocos Lógicos



Fonte: Autoria Própria.

Caso o término de execução não aconteça os indivíduos são enviados para o bloco reprodução, responsável por entregar novos indivíduos gerados a partir da combinação de indivíduos da antiga população, o fator de reprodução é definido pelo parâmetro p_{crs}, a nova população é enviada ao bloco de mutação no instante t_3.

No bloco de mutação alguns indivíduos são selecionados de maneira aleatória para receberem modificações em seu código genético, o fator de mutação é definido pelo parâmetro p_{mut} que pode ser alterado em cada execução do algoritmo a fim de se melhorar o tempo de resposta. No instante t_4 a população que já passou pela reprodução e pela mutação é enviada novamente ao bloco de avaliação.

Por fim o bloco avaliação faz novamente o calculo de *fitness* dos indivíduos da nova população e remove os indivíduos menos aptos. Novamente o processo de verificação de término é realizado e caso os parâmetros de conclusão sejam atingidos o programa é encerrado e é informado ao usuário a melhor resposta encontrada (indivíduo com a melhor *fitness*). Caso não a população é enviada para o bloco de reprodução e o ciclo recomeça.

4 DESCRIÇÃO DA IMPLEMENTAÇÃO

Neste capítulo é descrita a implementação realizada, os componentes VHDL desenvolvidos e também os componentes já disponíveis no *Microblaze* utilizados na implementação da arquitetura para FPGA que representa o Algoritmo Genético no *kit Spartan 3E*.

4.1 IMPLEMENTAÇÃO VHDL

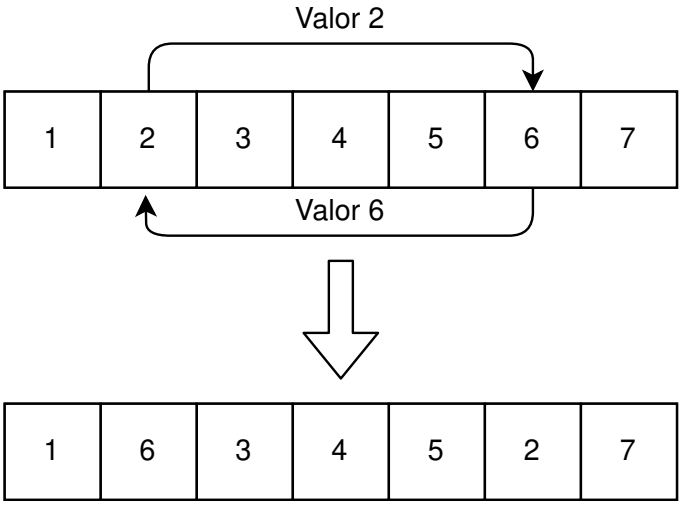
Nesta seção são descritos os blocos VHDL desenvolvidos para a implementação, na Figura 6 eles são representados pelo quadrado VHDL, presente dentro do *Microblaze*. Os blocos Mutação e Reprodução se referem aos operadores do Algoritmo Genético enquanto o bloco *Xoroshiro128+* (BLACKMAN; VIGNA, 2019) se refere à um gerador de números pseudo-aleatórios. Todos possuem em comum um sinal de *clock* e de *reset*. Os códigos em VHDL dos blocos de Mutação e Reprodução podem ser encontrados nos Apêndices A e B respectivamente.

4.1.1 MUTAÇÃO

O bloco VHDL responsável pela mutação, tem como valores de entrada uma solução, dois valores aleatórios diferentes que representam duas "posições" na solução e tem como valores de saída um sinal de fim de operação e uma *string* solução.

Ao sinal de *reset* a solução na entrada é carregada no bloco e o sinal de fim de operação é definido com '0'. Quando o valor de *reset* se torna '0' o número que se encontra nas posições recebidas de entrada é armazenado e após alguns ciclos de *clock*, os valores são trocados de posição. Por fim o sinal de fim de operação é definido com '1'.

Figura 7 – Descrição Visual da Operação de Mutação



Fonte: Autoria Própria.

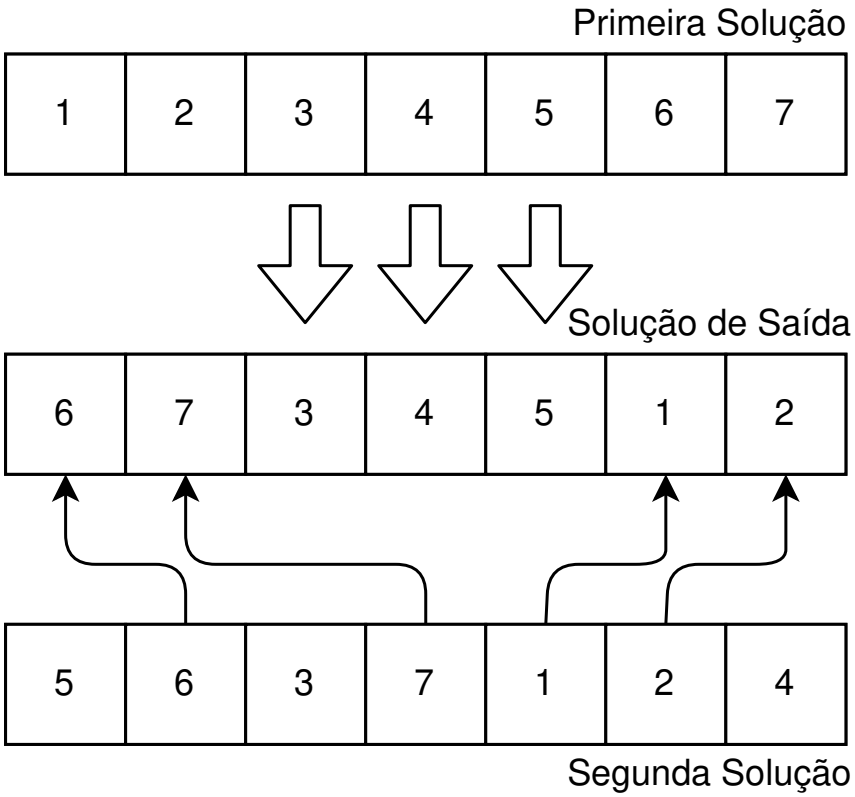
A Figura 7 ilustra a alteração em uma solução realizada pelo bloco de mutação. As setas pretas representam a troca de posição dos valores e a seta branca a mudança da solução. Os valores de entrada são a solução [1,2,3,4,5,6,7] e posições de troca 2 e 6. A solução de saída é [1,6,3,4,5,2,7].

4.1.2 REPRODUÇÃO

O bloco de reprodução recebe como valores de entrada as duas soluções que oferecem seu "código genético" e a posição de corte. Suas saídas são o a solução proveniente do cruzamento e o sinal de fim de operação.

No sinal de *reset* a solução de saída recebe na posição de corte o valor da cidade que estiver na posição de corte da primeira solução de entrada, o mesmo se repete para os dois valores seguintes a posição de corte. Os valores da segunda solução são então colocados, da direita para esquerda, na solução de saída. Se o valor já estiver presente ele é ignorado e se verifica o próximo valor da segunda solução, se a posição já foi preenchida pela primeira solução então verifica-se a próxima posição. Ao fim do preenchimento da solução final, o sinal de fim de operação é definido como '1'.

Figura 8 – Descrição Visual da Operação de Reprodução



Fonte: Autoria Própria.

A Figura 8 demonstra a operação realizada pelo bloco de reprodução.As setas brancas representam os valores recebidos das posições determinadas pelo número de corte enquanto as

setas pretas representam os valores recebidos pela segunda solução. Os valores de entrada são a primeira solução [1,2,3,4,5,6,7], a segunda solução [5,6,3,7,1,2,4] e o valor de corte '3'. A solução de saída gerada é [6,7,3,4,5,1,2].

4.1.3 XOROSHIRO128+

Para obtenção de números aleatórios a nível de *hardware* a escolha realizada foi a utilização do algoritmo *Xoroshiro128+*. O código VHDL utilizado neste trabalho foi previamente implementado em (RANTWIJK, 2016).

Xoroshiro128+ é um gerador de números pseudo-aleatórios de 128 *bits*, possível de ser implementado em *hardware* e que recebe seu nome baseado nas funções *xor, rotate, shift, rotate* que são utilizadas (BLACKMAN; VIGNA, 2019).

Para o funcionamento o bloco necessita apenas de uma 'semente', um valor inicial diferente de zero, e após alguns ciclos de *clock* é capaz de gerar novos números aleatórios a cada ciclo de *clock*. O valor de semente pode ser alterado no código VHDL na hora da geração do bloco.

4.2 COMPONENTES DO MICROBLAZE

Além dos componentes VHDL desenvolvidos, foram necessários alguns componentes já disponíveis pela *Xilinx* dentro do Xilinx Platform Studio, estes são descritos nesta seção. Estes componentes podem ser instanciados na hora da geração, e na Figura 6 está presente dentro do quadrado *Microblaze*.

4.2.1 XPS Timer

Módulo de *timer* de 32 bits, este componente permite que em uma chamada de função se capture um valor de tempo correspondente ao número de ciclos de *clock* ocorridos desde sua inicialização. Nesta implementação foi utilizado para medir o número de ciclos de *clock* e consequentemente o tempo de execução do algoritmo.

4.2.2 RS232 DCE

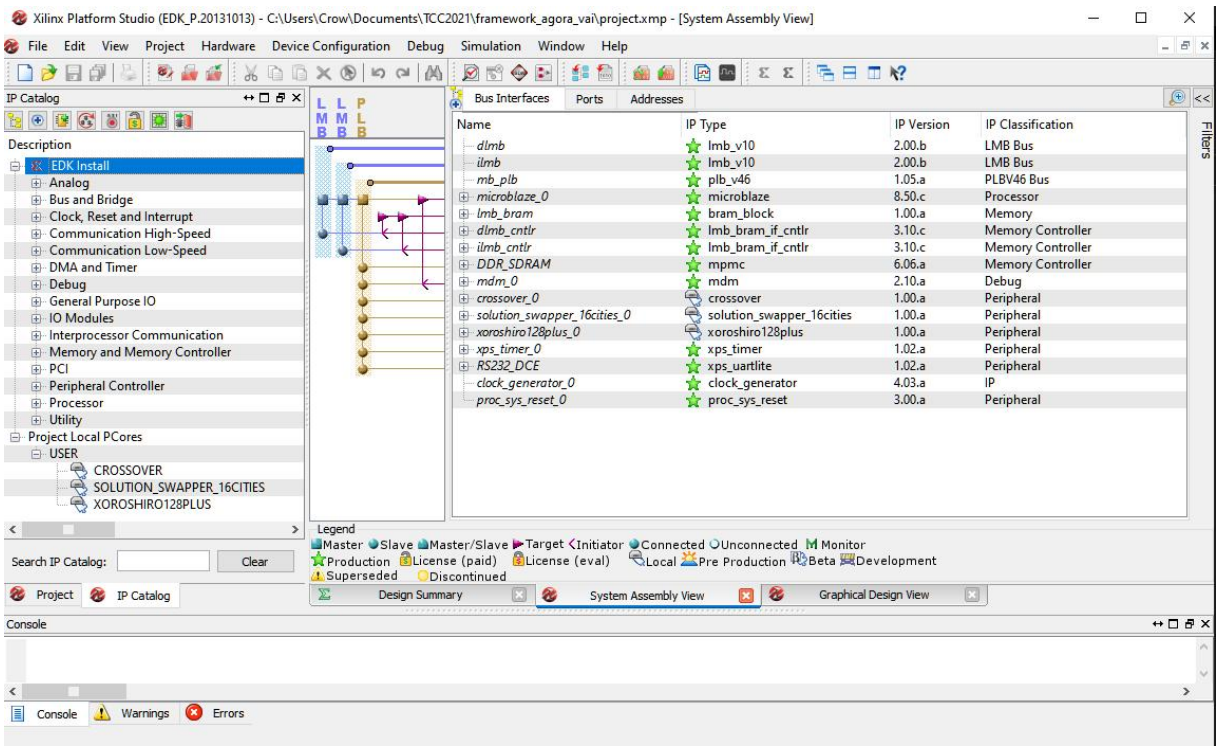
O módulo RS232 DCE é o responsável por realizar a conexão serial entre o *Microblaze* e o computador, por ele é possível enviar informações para um hiper-terminal que esteja conectado a porta serial. Neste trabalho foi utilizado para fazer a conexão do algoritmo com o usuário.

4.3 INTEGRAÇÃO COM O MICROBLAZE

A integração do *Microblaze* com os blocos VHDL desenvolvidos e apresentados na seção anterior, foi realizada no *software Xilinx Platform Studio*. Após os componentes necessários

terem sido selecionados e atribuídos endereços de acesso aos registradores, o XPS permite a geração de um arquivo de *bistream* que configura os blocos lógicos da FPGA de acordo com o *hardware* descrito. O arquivo pode ser então enviado ao *Kit Spartan 3E* e assim, a configuração dos periféricos e os blocos VHDL descritos estarão prontos para receber o código da aplicação.

Figura 9 – Interface do XPS



Fonte: Autoria Própria.

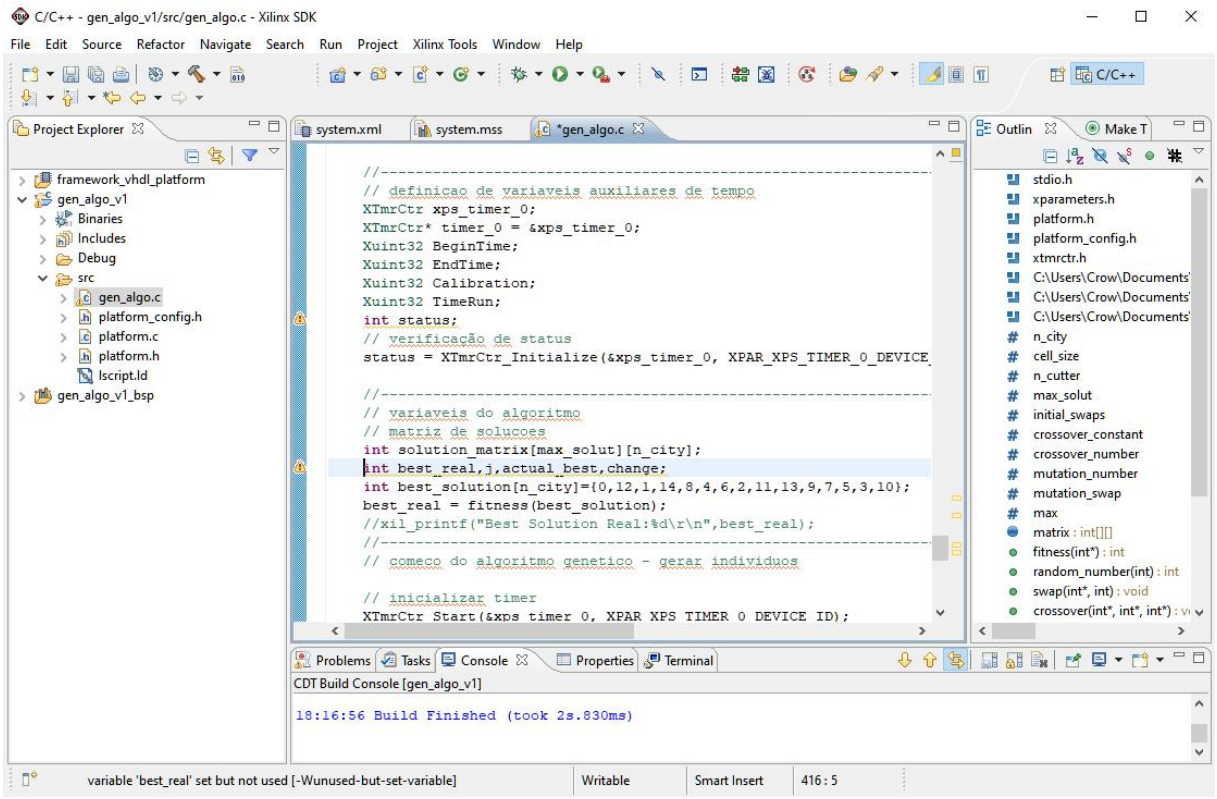
A Figura 9 mostra a interface do *software* XPS, é possível ver os periféricos instanciados, os blocos VHDL gerados pelo usuário, e os barramentos em que estão conectados. LMB se refere à *Local Memory Bus* que é o barramento responsável por acessar a memória disponível, enquanto o PLB significa *Processor Local Bus* e é o barramento que realiza a conexão dos blocos lógicos e dos periféricos ao *Microblaze*.

4.4 CÓDIGO EM C

O código na linguagem de programação C que atua no *Microblaze* foi escrito no *software Xilinx Software Development Kit*, sua integração com o XPS permite o uso de bibliotecas com funções para realizar acesso aos endereços de registradores designados nos blocos VHDL e nos periféricos.

A Figura 10 apresenta a interface do SDK, no lado esquerdo, existem 3 projetos, o primeiro *framework_vhdl_platform* é o *framework* que descreve o *hardware* implementado, o segundo *gen_algo_v1* é o projeto em que foi desenvolvido o código em C e por fim o terceiro *gen_algo_v1_bsp* é um *board support package* que possui todos os *drivers* necessários para a

Figura 10 – Interface do SDK



Fonte: Autoria Própria.

aplicação. Ao centro o arquivo *gen_algo.c* está aberto e mostra parte de seu código. Ao lado direto apresentam-se algumas das variáveis utilizadas.

A base do código é a função *main*, nela são declaradas as variáveis de solução e as variáveis auxiliares. A função *main* é responsável por chamar as funções que interagem com os blocos VHDL, controlar a condição de parada, declarar e acessar a matriz de soluções e também enviar à tela os resultados das operações.

O controle das funções de *timer* também é realizado pela *main*, ao fim da atuação da função geração inicial, o *timer* é iniciado e ao fim da ultima iteração o valor do *timer* é capturado. Este valor é divido por 50, uma constante que obtida do número de *clocks* por segundo, e obtém-se o valor de tempo em *us* da execução.

4.4.1 GERAÇÃO INICIAL

A primeira função a ser 'chamada' pela *main* é a função que gera as soluções iniciais. Esta função começa com uma solução predefinida, que pode ser alterada no código, e então para gerar mais soluções é realizado um número N de mutações na solução anterior. Gerando novas soluções até que o limite populacional seja preenchido. Este tipo de geração garante uma aleatoriedade inicial satisfatória caso o número N seja alto o suficiente e também garante que todas as soluções são válidas.

4.4.2 REPRODUÇÃO

Para que seja realizada a reprodução, um par de soluções deve ser selecionada, essa seleção é executada pelo método da roleta. Atribui-se uma chance a todas as soluções baseada no seu valor de *fitness* e então um número aleatório define qual são as duas soluções que participam desta etapa. Por se tratar de um problema de minimização um menor valor de *fitness* representa uma melhor resposta, foi necessário utilizar o inverso do valor de *fitness* para a chance de ser escolhido na roleta.

Selecionados os pais, um indivíduo deve ser selecionado para que seu espaço seja ocupado pelo novo individuo que será gerado. Esta seleção também é realizada pelo método da roleta, porém não há necessidade de se obter o valor de *fitness* inverso, já que a chance de ser escolhido neste caso deve ser proporcional ao valor de *fitness*.

Definidos estes indivíduos, a função então envia seus valores para o bloco VHDL de reprodução e espera o sinal de fim de operação, com o valor em '1' a leitura nos registradores do bloco é feita e obtém-se a solução nova, que deve ocupar o espaço da antiga na matriz de soluções.

4.4.3 MUTAÇÃO

A seleção da mutação é realizada de maneira aleatória, um individuo é escolhido independente de seu valor de *fitness* para sofrer uma mutação. A função recebe dois valores aleatórios e a solução escolhida e envia os estes valores para o bloco VHDL de mutação. Ao sinal de fim de operação, a solução já alterada é lida dos registradores do bloco. A intensidade e a periodicidade da atuação da função de mutação pode é determinada por parâmetros no código.

4.4.4 CALCULO DE FITNESS

Neste trabalho a modelagem das instancias do problema foi realizada por meio de uma matriz de distâncias, em que o valor em sua posição X/Y refletia na distancia entre as cidades X e Y. O calculo do valor de *fitness* é um somatório baseado nesta matriz de distâncias, compara-se a cidade em cada posição com a a cidade da próxima posição e então é soma-se cada uma das distancias.

4.4.5 CONDIÇÃO DE PARADA

A condição de parada implementada foi a de número de iterações do código. A função *main* é responsável por esse controle e o número de iterações executadas é definido por um parâmetro na hora da compilação do código. Ao termino da execução, a melhor solução é apresentada ao usuário com seu valor de *fitness*.

4.5 EXECUÇÃO

O Quadro 1 apresenta a sequencia de mensagens recebidas pelo cabo serial durante a execução do algoritmo para um problema com 15 instâncias e população de 8 soluções.

Quadro 1 – Algoritmo Rodando na Interface do Tera Term

```
Genetic Algorithm Starting

Generating Initial Solution
Initial Solutions:
Solution 0:0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Solution 1:0 1 2 3 4 5 6 9 14 12 7 8 10 13 11
Solution 2:5 1 2 9 4 13 6 10 3 12 7 8 14 0 11
Solution 3:1 14 12 2 4 9 6 10 3 13 7 8 5 0 11
Solution 4:1 14 12 9 4 6 8 10 13 7 3 2 5 0 11
Solution 5:11 14 12 1 4 5 8 3 13 7 10 6 2 0 9
Solution 6:11 0 12 1 4 5 6 3 13 7 14 2 10 8 9
Solution 7:11 14 12 1 4 5 2 8 3 7 6 0 10 13 9

Reproduction Starting
Parent Selected: 7 Parent Selected: 4
To-Remove Selected: 0
Cutter Position:11
Reproduction Done!

Mutation Starting
Mutation Selected:7
Mutation Done!

Ending Solutions:
Solution 0:1 14 12 9 4 6 8 7 3 2 5 0 10 13 11 - 566
Solution 1:0 1 2 3 4 5 6 9 14 12 7 8 10 13 11 - 732
Solution 2:5 1 2 9 4 13 6 10 3 12 7 8 14 0 11 - 667
Solution 3:1 14 12 2 4 9 6 10 3 13 7 8 5 0 11 - 668
Solution 4:1 14 12 9 4 6 8 10 13 7 3 2 5 0 11 - 660
Solution 5:11 14 12 1 4 5 8 3 13 7 10 6 2 0 9 - 691
Solution 6:11 0 12 1 4 5 6 3 13 7 14 2 10 8 9 - 710
Solution 7:11 14 12 1 2 5 10 8 3 7 6 0 4 13 9 - 678
Best Solution Found: 566

Time to run in FPGA: 48728 us
```

Fonte: Autoria Própria.

Após o inicio do programa as soluções iniciais são geradas e apresentadas na tela. São então selecionados dois indivíduos para reprodução, 7 e 4, o indivíduo a ser removido escolhido foi o 0. A posição de corte para a reprodução gerada foi a 11 e por fim a mensagem

de reprodução bem sucedida é apresentada na tela.

O indivíduo número 7 foi o escolhido para sofrer a mutação e logo abaixo a mensagem de sucesso é apresentada na tela. Após uma iteração completa de reprodução e mutação a execução se encerra, a população final é apresentada e a melhor solução é mostrada na tela. Por fim, o tempo necessário para execução total é mostrada e o programa se encerra.

4.6 EXPERIMENTOS COMPUTACIONAIS

Nesta seção são descritos os experimentos realizados com a fim de verificar a aceleração e o crescimento do tempo de execução em função do número de cidades no problema. As instancias utilizadas para cada uma das execuções foram reduções da instancia **P01** (BURKARDT, 2019), um conjunto de 15 cidades.

O Quadro 2 mostra os tempos de 10 execuções do Algoritmo Genético na FPGA, para um número de 8 a 15 cidades com a média e o desvio padrão ao fim do quadro.

Quadro 2 – Gráfico de Média do Tempo de Execução (em μs) por Número de Cidades

N° de Cidades	8	9	10	11	12	13	14	15
1ª Execução	3798	4003	4175	4353	4502	4644	4816	4983
2° Execução	3818	4084	4189	4320	4480	4671	4810	4959
3° Execução	3829	4023	4223	4381	4542	4645	4806	4946
4° Execução	3871	4057	4177	4324	4489	4681	4850	4961
5° Execução	3847	4062	4196	4352	4540	4678	4804	4992
6° Execução	3832	4051	4201	4350	4506	4666	4799	4984
7° Execução	3821	4041	4237	4365	4486	4640	4831	4970
8° Execução	3848	4007	4213	4318	4502	4666	4821	4990
9° Execução	3853	4037	4197	4373	4479	4637	4823	4953
10° Execução	3818	4059	4181	4362	4500	4683	4823	4982
Média	3830.5	4046	4196.5	4352.5	4501	4666	4818.5	4976
Desvio Padrão (±)	20.245	24.270	19.279	21.146	21.209	16.985	14.213	15.556

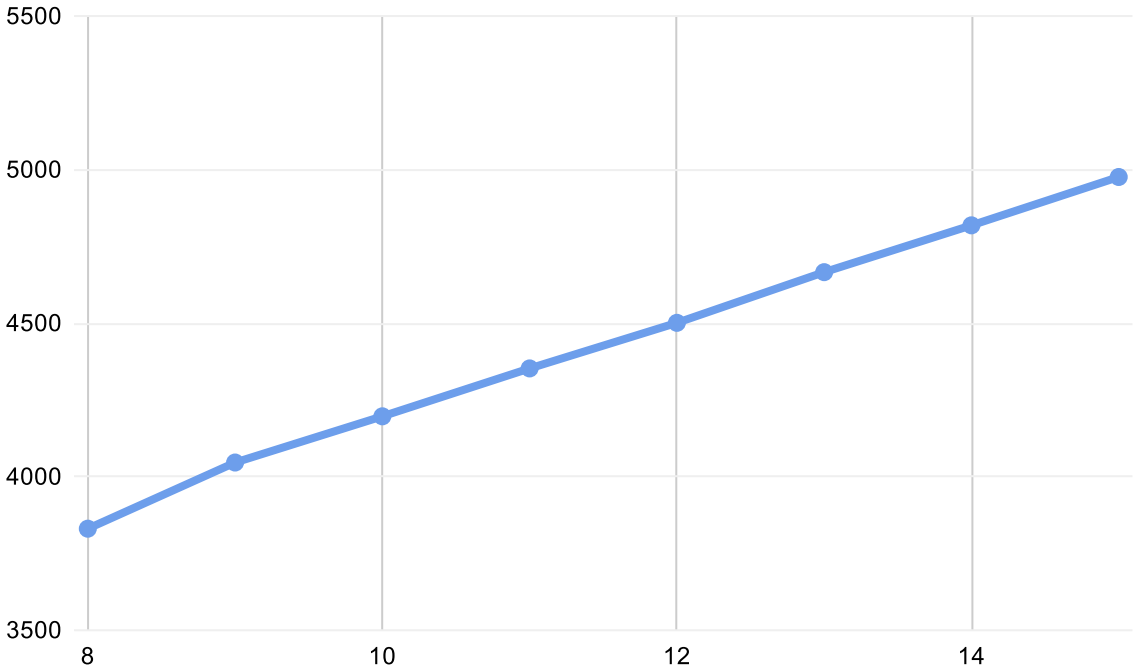
Fonte: Autoria Própria.

A Figura 11 apresenta o tempo médio das execuções para cada número de cidades. A linha azul ao centro apresenta um comportamento crescente de maneira linear, demonstrando a escalabilidade do sistema no quesito tempo de execução.

A busca pela qualidade da solução não foi um fator considerado durante estes testes, apenas o tempo de execução. Cada execução realizou 10 iterações do Algoritmo Genético com 2 reproduções e 2 mutações por iteração e uma população com 8 indivíduos.

Para fins de verificação de tempo computacional, durantes as execuções, foram retirados todos os envios de mensagens não essenciais para a tela, sendo enviados apenas os valores de tempo e de solução final encontrada.

Figura 11 – Gráfico de Média do Tempo de Execução (em μs) por Número de Cidades



Fonte: Autoria Própria.

4.7 DISCUSSÕES

Os experimentos realizados mostraram a limitação do *hardware* disponível, em muitos momentos não foi possível aumentar o número de soluções em uma população nem aumentar o número de cidades da instância sem resultar em anomalias.

Contudo para as instancias com menos de 16 cidades e com um número de soluções igual a 8, a arquitetura implementada foi capaz de encontrar resultados sem limitações. Outro fator positivo mostrado no Quadro 2 é o baixo desvio padrão, que demonstra a estabilidade em questão de tempo, durante a busca por soluções.

O gráfico apresentado na Figura 11, mostra que a curva de tempo de execução em relação ao número de cidades cresce de maneira polinomial. Este tipo de crescimento de tempo demonstra que a arquitetura implementada tem potencial para executar o Algoritmo Genético em instancias ainda maiores do que as que foram possíveis com *hardware* utilizado.

Optou-se por não realizar uma comparação entre a execução no *microblaze* e em um computador comum, devido a grande diferença na frequência de *clock* entre os dois dispositivos.

5 CONCLUSÃO

Neste trabalho foi demonstrada a implementação de uma arquitetura em FPGA que representa o Algoritmo Genético para a busca de soluções para o PCV. O uso de blocos VHDL, para a execução de etapas do Algoritmo Genético, trouxe escalabilidade ao sistema sendo possível aumentar a arquitetura para instancias de problemas ainda maiores. A implementação foi capaz de encontrar soluções válidas usando os blocos lógicos em tempo polinomial e demonstrou um crescimento do tempo de execução linear. Infelizmente a falta de componentes lógicos e memória interna, do *hardware* disponível, foi um fator limitante na seleção do tamanho das instancias de teste e do número de soluções dentro de uma população. As contribuições deste trabalho podem servir de base para implementação de outras meta-heurísticas em FPGA para diversos problemas NP-Completos.

5.1 PRINCIPAIS DIFICULDADES

A obsolescência do *hardware* disponível para uso, o *FPGA Spartan 3E Starter Kit*, resultou em uma necessidade de tempo para implementação maior do que a prevista. Por esse motivo, optou-se pela não implementação de um sistema de referencia e priorização da implementação do sistema proposto.

5.2 TRABALHOS FUTUROS

Recomenda-se para trabalhos futuros as seguintes sugestões:

- Desenvolver um sistema de referência em linguagem C, para verificação do valor de aceleração.
- Implementar a arquitetura em um *hardware* com maior capacidade computacional.
- Escalar os blocos lógicos para que possam receber e processar um número maior de soluções ao mesmo tempo.
- Desenvolver blocos lógicos referentes ao calculo de *fitness*, roleta de seleção e geração de soluções iniciais.

Referências

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: **Proceedings of the April 18-20, 1967, Spring Joint Computer Conference**. New York, NY, USA: Association for Computing Machinery, 1967. (AFIPS '67 (Spring)), p. 483–485. ISBN 9781450378956. Disponível em: <https://doi.org/10.1145/1465482.1465560>. Citado na página 12.

BECCENERI, J. C. Meta-heurísticas e otimização combinatória: Aplicações em problemas ambientais. 2008. Citado na página 6.

BLACKMAN, D.; VIGNA, S. **Scrambled Linear Pseudorandom Number Generators**. 2019. Citado 2 vezes nas páginas 18 e 20.

BURKARDT, J. **TSP Data for the Traveling Salesperson Problem**. 2019. Disponível em: <https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html>. Citado na página 25.

CHU, P. P. **FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version**. Hoboken, N.J.: John Wiley & Sons Inc, 2008. ISBN 0470185317, 9780470185315. Citado na página 9.

COOK, S. A. The complexity of theorem-proving procedures. In: **Proceedings of the Third Annual ACM Symposium on Theory of Computing**. New York, NY, USA: Association for Computing Machinery, 1971. (STOC '71), p. 151–158. ISBN 9781450374644. Disponível em: <https://doi.org/10.1145/800157.805047>. Citado na página 4.

COSTA, C. B. et al. Prior detection of genetic algorithm significant parameters: Coupling factorial design technique to genetic algorithm. **Chemical Engineering Science**, v. 62, n. 17, p. 4780 – 4801, 2007. ISSN 0009-2509. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0009250907003168>. Citado na página 6.

CUNG, V.-D. et al. Strategies for the parallel implementation of metaheuristics. In: _____. **Essays and Surveys in Metaheuristics**. Boston, MA: Springer US, 2002. p. 263–308. Disponível em: <https://doi.org/10.1007/978-1-4615-1507-4_13>. Citado na página 2.

D'AMORE, R. **VHDL: descrição e síntese de circuitos digitais**. Grupo Gen - LTC, 2012. ISBN 9788521620549. Disponível em: <https://books.google.com.br/books?id=yG1nLgEACAAJ>. Citado 2 vezes nas páginas 10 e 11.

DANTZIG, G.; FULKERSON, R.; JOHNSON, S. Solution of a large-scale traveling-salesman problem. **Journal of the Operations Research Society of America**, v. 2, n. 4, p. 393–410, 1954. Disponível em: <https://doi.org/10.1287/opre.2.4.393>. Citado na página 5.

FOULDS, L. **Combinatorial optimization for undergraduates**. SPRINGER VERLAG GMBH, 1984. (Problem Books in Mathematics). ISBN 9780387909776. Disponível em: <https://books.google.com.br/books?id=I9IZAQAAIAAJ>. Citado na página 6.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman & Co., 1979. ISBN 0716710447. Citado 3 vezes nas páginas 1, 4 e 5.

GENDREAU, M.; POTVIN, J.-Y. **Handbook of Metaheuristics**. 2nd. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441916636. Citado na página 9.

GOLDBARG, M. **Otimização combinatória e programação linear**. Rio de Janeiro: Elsevier Brasil, 2005. v. 2ed. Citado 2 vezes nas páginas 1 e 5.

GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization and Machine Learning**. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675. Citado 3 vezes nas páginas 7, 8 e 9.

HUANG, H.-C. FPGA-based parallel metaheuristic PSO algorithm and its application to global path planning for autonomous robot navigation. **Journal of Intelligent & Robotic Systems**, v. 76, n. 3, p. 475–488, Dec 2014. ISSN 1573-0409. Disponível em: <https://doi.org/10.1007/s10846-013-9884-9>. Citado na página 12.

JAEN-CUELLAR, A. Y. et al. Fpga-based embedded system architecture for micro-genetic algorithms applied to parameters optimization in motion control. **Advances in Electrical and Computer Engineering**, v. 15, n. 1, p. 23–32, 2015. Citado na página 12.

KARP, R. M. On the computational complexity of combinatorial problems. **Networks**, v. 5, n. 1, p. 45–68, 1975. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/net.1975.5.1.45>. Citado na página 5.

LAPORTE, G. The traveling salesman problem: An overview of exact and approximate algorithms. **European Journal of Operational Research**, v. 59, n. 2, p. 231 – 247, 1992. ISSN 0377-2217. Disponível em: <http://www.sciencedirect.com/science/article/pii/037722179290138Y>. Citado na página 1.

LINDEN, R. **ALGORITMOS GENÉTICOS**. 3. ed. Rio de Janeiro: CIÊNCIA MODERNA, 2012. ISBN 9788539901951. Disponível em: <https://books.google.com.br/books?id=SrKouQAACAAJ>. Citado 3 vezes nas páginas 1, 6 e 8.

NEDJAH, N.; MOURELLE, L. **Hardware for Soft Computing and Soft Computing for Hardware**. Heidelberg: Springer Publishing Company, Incorporated, 2014. Citado na página 13.

OCHI, L. S. et al. Sequential and parallel metaheuristics based on grasp and vns for solving the traveling purchaser problem. Citeseer, 2002. Citado na página 2.

PEDRONI, V. A. **Circuit Design and Simulation with VHDL**. 2. ed. London: The MIT Press, 2010. ISBN 0262014335, 9780262014335. Citado 4 vezes nas páginas 2, 10, 11 e 12.

PUNNEN, A. P. The traveling salesman problem: Applications, formulations and variations. In: **The traveling salesman problem and its variations**. [S.l.]: Springer, 2007. p. 1–28. Citado na página 1.

RANTWIJK, J. van. **Pseudo-Random Number Generators in VHDL**. 2016. Disponível em: <https://github.com/jorisvr/vhdl_prng>. Citado na página 20.

REINELT, G. **TSPLIB**. 2000. Disponível em: <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>. Citado na página 15.

STALLINGS, W. **Arquitetura e organizacao de computadores**. Sao Paulo: Pearson, 2010. Citado na página 12.

TALBI, E.-G. **Metaheuristics: From Design to Implementation**. Hoboken, Nova Jersey, EUA: John Wiley & Sons Publishing, 2009. ISBN 0470278587, 9780470278581. Citado na página 1.

T.TERANISHI. **TSPLIB**. 2004. Disponível em: <https://ttssh2.osdn.jp/index.html.en>. Citado na página 15.

XILINX. **ISE In-Depth Tutorial**. XILINX, 2009. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise11tut.pdf>. Citado na página 15.

XILINX. **Spartan-3E FPGA Starter Kit Board User Guide**. XILINX, 2011. Disponível em: <https://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf>. Citado na página 14.

XILINX. **EDK Concepts, Tools, and Techniques**. XILINX, 2013. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/edk_ctt.pdf>. Citado na página 15.

XILINX. **MicroBlaze Processor Reference Guide**. XILINX, 2013. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/mb_ref_guide.pdf>. Citado na página 14.

Apêndices

APÊNDICE A – Código VHDL de Mutação

```
entity top is
 generic(
  Cell_Size: integer := 4;
  Num_City: integer := 6);
 port(
  i_clk : in std_logic;
  i_rst : in std_logic;
  i_rand1: in integer;
  i_rand2: in integer;
  i_solution : in std_logic_vector((Num_City*Cell_Size)-1 downto 0);
  o_done : out std_logic;
  o_solution : out std_logic_vector((Num_City*Cell_Size)-1 downto 0));
end top;

architecture solution_manager_arch of top is
 signal s_permutation : std_logic_vector(1 downto 0);
 signal s_temp_a : integer range 0 to 1024;
 signal s_temp_b : integer range 0 to 1024;
 signal s_solution : std_logic_vector((Num_City*Cell_Size)-1 downto 0);
 signal s_transpose_a : std_logic_vector(Cell_Size-1 downto 0);
 signal s_transpose_b : std_logic_vector(Cell_Size-1 downto 0);
 signal s_done : std_logic;
begin
 o_done <= s_done;
 o_solution <= s_solution;
 process (i_clk,i_rst) is begin
  if rising_edge(i_clk) then
   if (i_rst = '1') then
      s_solution <= i_solution;
      s_done <= '0';
      s_permutation <= "00";
   else
    if(s_permutation = "00") then
     s_temp_a <= i_rand1;
     s_temp_b <= i_rand2;
     s_permutation <= "01";
    elsif (s_permutation = "01") then
     s_transpose_a(Cell_Size-1 downto 0) <=
     s_solution(((s_temp_a*Cell_Size)+Cell_Size-1) downto (s_temp_a*Cell_Size));
     s_transpose_b(Cell_Size-1 downto 0) <=
     s_solution(((s_temp_b*Cell_Size)+Cell_Size-1) downto (s_temp_b*Cell_Size));
     s_permutation <= "10";
    elsif (s_permutation = "10") then
     s_solution(((s_temp_a*Cell_Size)+Cell_Size-1) downto (s_temp_a*Cell_Size))
    <= s_transpose_b(Cell_Size-1 downto 0);
     s_solution(((s_temp_b*Cell_Size)+Cell_Size-1) downto (s_temp_b*Cell_Size))
    <= s_transpose_a(Cell_Size-1 downto 0);
     s_done <= '1';
     s_permutation <= "11";
    end if;
   end if;
  end if;
 end process;
end solution_manager_arch;
```

APÊNDICE B – Código VHDL de Reprodução

```
entity top is
 generic(
  Cell_Size: integer := 4;
  Num_City: integer := 16);
 port(
  i_clk : in std_logic;
  i_rst : in std_logic;
  i_cutter: in integer;
  i_size: in integer;
  i_solution1 : in std_logic_vector((Num_City*Cell_Size)-1 downto 0);
  i_solution2 : in std_logic_vector((Num_City*Cell_Size)-1 downto 0);
  o_done : out std_logic;
  o_solution : out std_logic_vector((Num_City*Cell_Size)-1 downto 0));
end top;

architecture crossover_arch of top is
 signal s_solution : std_logic_vector((Num_City*Cell_Size)-1 downto 0);
 signal s_cutter1 : integer range 0 to 31;
 signal s_cutter2 : integer range 0 to 31;
 signal s_cutter3 : integer range 0 to 31;
 signal s_present : integer range 0 to 31;
 signal step : std_logic;
 signal s_i : integer range 0 to 31;
 signal s_j : integer range 0 to 31;
 signal s_done : std_logic;
begin
 o_done <= s_done;
 o_solution <= s_solution;
 process (i_clk,i_rst) is
  begin
  if rising_edge(i_clk) then
   if (i_rst = '1') then
    s_solution <= i_solution1;
    s_done <= '0';
    s_cutter1 <= to_integer(unsigned(
    i_solution1(((i_cutter*Cell_Size)+Cell_Size-1) downto (i_cutter*Cell_Size))));
    s_cutter2 <= to_integer(unsigned(
    i_solution1((((i_cutter+1)*Cell_Size)+Cell_Size-1) downto ((i_cutter+1)*Cell_Size))));
    s_cutter3 <= to_integer(unsigned(
    i_solution1((((i_cutter+2)*Cell_Size)+Cell_Size-1) downto ((i_cutter+2)*Cell_Size))));
    s_j <= 0;
    s_i <= 0;
    step <= '0';
   else
    if (step = '0') then
     if(s_i /= i_size) then
      s_present <= to_integer(unsigned(
      i_solution2(((s_i*Cell_Size)+Cell_Size-1) downto (s_i*Cell_Size))));
     end if;
     step <= '1';
    elsif (s_j < i_size) then
     if (s_j = i_cutter) then
      s_j <= s_j + 1;
     elsif (s_j = (i_cutter + 1)) then
      s_j <= s_j + 1;
```

```
      elsif (s_j = (i_cutter + 2)) then
       s_j <= s_j + 1;
      elsif ( s_cutter1 = s_present ) then
       s_i <= s_i + 1;
      elsif ( s_cutter2 = s_present ) then
       s_i <= s_i + 1;
      elsif ( s_cutter3 = s_present ) then
       s_i <= s_i + 1;
      else
       s_solution(((s_j*Cell_Size)+Cell_Size-1) downto (s_j*Cell_Size)) <=
       std_logic_vector(to_unsigned(s_present, Cell_Size));
       s_j <= s_j + 1;
       s_i <= s_i + 1;
      end if;
      step <= '0';
     else
      s_done <= '1';
     end if;
    end if;
   end if;
  end process;
end crossover_arch;
```