Ivan Ferrucio Reche da Silva Filgueiras

OTIMIZAÇÃO DE CIRCUITOS CMOS POR ALGORITMO GENÉTICO

Trabalho de Conclusão de Curso apresentado à Escola de Engenharia de São Carlos, da Universidade de São Paulo

Curso de Engenharia de Computação

ORIENTADOR: João Navarro Soares Jr.

São Carlos
2010

Ficha catalográfica preparada pela Seção de Tratamento da Informação do Serviço de Biblioteca – EESC/USP

Filgueiras, Ivan Ferrucio Reche da Silva
Otimização de Circuitos CMOS por Algoritmo Genético / Ivan Ferrucio Reche da Silva Filgueiras ; orientador João Navarro Soares Jr. – São Carlos, 2010.

Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação) – Escola de Engenharia de São Carlos da Universidade de São Paulo, 2010.

1. Circuitos integrados MOS. 2. Fontes de corrente. 3. Inteligência artificial.

4. Algoritmos genéticos. 5. Otimização. I. Título.

FOLHA DE APROVAÇÃO

Nome: Ivan Ferrucio Reche da Silva

Título: "Otimização de Circuitos CMOS por Algoritmo Genético"

Trabalho de Conclusão de Curso defendido e aprovado em 17/ 6 /2010,

com NOTA 10,0 (dez, zero), pela comissão julgadora:

Prof. Associado Amílcar Careli César - SEL/EESC/USP

Prof. Dr. Evandro Luís Linhari Rodrigues - SSC/ICMC/USP

Prof. Dr. Evandro Luís Linhari Rodrigues
Coordenador pela EESC/USP do
Curso de Engenharia de Computação

DEDICATÓRIA

Dedico este trabalho aos meus pais Walkiria e Dionísio, por terem arcado com a difícil tarefa de garantir a minha educação desde jovem.

Dedico também à minha namorada Márcia, que sempre foi compreensiva e amorosa, mesmo nos momentos onde estive ausente para dedicar-me ao trabalho.

Por fim, dedico ao meu orientador João, que acreditou no meu potencial de realizar um trabalho desafiador.

AGRADECIMENTOS

Agradeço a todos os professores que participaram da minha formação e possibilitaram que eu tivesse a base teórica necessária para realizar este trabalho.

Agradeço também à comunidade de software livre, que trabalha arduamente para fornecer ferramentas de qualidade para todos, como as utilizadas ao longo deste trabalho (GNU/Linux, Gnucap, OpenOffice e PLT-Scheme).

Resumo do Projeto de Formatura apresentado à EESC-USP como parte dos requisitos necessários para a obtenção da conclusão do curso de Engenharia de Computação

OTIMIZAÇÃO DE CIRCUITOS CMOS POR ALGORITMO GENÉTICO

Ivan Ferrucio Reche da Silva Filgueiras

05/2010

Orientador: Prof. Dr. João Navarro Soares Jr.

Áreas de Concentração: Microeletrônica, inteligência artificial.

Palavras chave: Circuitos integrados MOS, fontes de corrente, inteligência artificial, algoritmos genéticos, otimização.

RESUMO

O dimensionamento de transistores para blocos eletrônicos, digitais ou analógicos não é uma tarefa simples e várias técnicas são empregadas para tal. Normalmente, projetistas trabalham com relações simplificadas para transistores e obtêm equações para os blocos. A partir destas equações, são determinadas dimensões que, posteriormente, são ajustadas por meio de simulações. Tal caminho é demorado, pouco prático e exige, para o sucesso, experiência do projetista. Um caminho alternativo para o dimensionamento de transistores é o desenvolvimento de programas de auxílio a projeto. Os métodos empregados em tais programas são os mais variados, mas os resultados, principalmente para circuitos analógicos, deixam a desejar e dependem muito da adequação dos modelos aplicados. O objetivo deste trabalho é estudar a aplicabilidade de algoritmos genéticos no projeto de circuitos. Para isto, será desenvolvido um programa para dimensionar alguns blocos de circuitos CMOS (*Complementary Metal Oxide Semiconductor*), aplicando algoritmos genéticos e conhecimento mínimo sobre o domínio da aplicação.

Abstract of the Undergraduate Project presented to EESC-USP as a partial fulfilment of the requirements to conclude Computer Engineering course

OPTIMIZATION OF CMOS CIRCUITS BY GENETIC ALGORITHMS

Ivan Ferrucio Reche da Silva Filgueiras

05/2010

Advisor: Prof. Dr. João Navarro Soares Jr.

Concentration Areas: Microelectronics, artificial intelligence.

Keywords: MOS integrated circuits, current sources, artificial intelligence, genetic algorithms, optimization.

ABSTRACT

The sizing of transistors for electronic blocks, digital or analogical, is not a simple task and a variety of techniques are employed to do so. Designers usually work with simplified relations for transistors and obtain equations for those blocks. From these equations, the dimensions are determined which, afterwards, are adjusted by simulations. This process is time-consuming and not practical and it demands a lot of designer expertise. A possible alternative way for the sizing of transistors is the development of computer programs to help the design. The methods employed by such programs are diverse, but the results, especially for analogical circuits, are usually disappointing and very affected by the used models. The goal of this work is to study the applicability of genetic algorithms in circuit design. For this it will be developed a software for sizing some CMOS (Complementary Metal Oxide Semiconductor) circuit blocks, employing genetic algorithms and the minimum amount of knowledge in the application domain.

Índice de figuras

Índice de tabelas

Sumário

1 Introdução

O projeto de circuitos integrados CMOS (*Complementary Metal Oxide Semiconductor*) se faz através da aplicação de relações simplificadas que descrevem o funcionamento dos transistores construído com a tecnologia. Através destas equações obtém-se relações que servem de ponto de partida. A partir dai, um projetista experiente pode aprofundar os modelos utilizados e, consequentemente, melhorar os resultados utilizando simulações. Este processo é lento e caro.

Surge, então, uma demanda por ferramentas que possam auxiliar projetistas sem o perfil de especialista a projetar circuitos otimizados e com o mínimo de conhecimento e no intervalo mais curto possível.

A tecnologia CMOS passou, a partir dos anos 80, a ser dominante na fabricação de circuitos integrados devido às vantagens sem igual que ela oferece: alto nível de integração, baixo consumo de potência e simplicidade de projeto. Nos últimos anos, 75% dos circuitos semicondutores (tanto em quantidade como em valor) foi produzido em CMOS, fato que adiciona outra vantagem à tecnologia: redução de custos devido à escala de produção. Este quadro não deve se alterar nos próximos anos [IT05] e é um fator motivador para a realização do trabalho nesta área.

1.1 Objetivo

O objetivo deste trabalho é analisar o uso de algoritmos genéticos para projetar circuitos CMOS. Desta forma, introduz-se uma mudança de paradigma na confecção de circuitos eletrônicos: soluções para este domínio de problemas não são projetadas, e sim procuradas em um espaço de busca.

1.2 Escolha do Algoritmo Genético

O espaço de busca do projeto de um circuito CMOS simples já pode ser suficiente para uma explosão da complexidade computacional, tornando inviável a análise de cada uma das possíveis combinações de parâmetros envolvidos no problema.

Necessita-se, então, de técnicas agressivas de busca em domínios vastos. Os algoritmos genéticos são conhecidos por seu ótimo desempenho em diversas aplicações, além de sua flexibilidade. Com mudanças nos parâmetros do método, é possível otimizar o desempenho da própria busca e conseguir resultados satisfatórios em tempo reduzido (se comparados ao tempo de projeto ou à busca aleatória).

A ideia de utilizar um algoritmo genético para esta otimização decorre também da característica de projetos de circuitos eletrônicos: constituem-se basicamente de um conjunto de elementos, conectados entre si, que são dotados de atributos (parâmetros). Sendo assim, é trivial modelar circuitos como indivíduos possuidores de DNA, sendo este o fulcro da técnica. Portanto, a

abordagem por algoritmos genéticos parece ser adequada ao problema de projeto.

1.3 Escolha da fonte de corrente como circuito a ser otimizado

As técnicas de otimização por algoritmo genético utilizadas neste trabalho serão aplicadas em um circuito de fonte de corrente voltado para aplicações de microeletrônica.

Esta escolha foi feito pela importância deste circuito. Diversas implementações clássicas de circuitos analógicos utilizam fontes de corrente. Aumentar a eficiência delas afeta diretamente a qualidade final do circuito para qual ela está sendo utilizada.

Além disso, foi considerada também a simplicidade de simular uma fonte deste tipo. A simulação é *DC*, e isso faz com que os simuladores operem consideravelmente mais rápido. Como as simulações são utilizadas constantemente na otimização por algoritmo genético empregada neste trabalho, o desempenho do simulador em termos de tempo é um fator importante a ser considerado.

1.3.1 Tecnologia MOS utilizada

Foram escolhidos os transistores típicos da tecnologia CMOS 0,35um da *foundry* da AMS (*Austria Micro-Systems*) [AMS10]. É fornecido para estes transistores um modelo BSIM3v3 (*Berkeley Short-channel IGFET Mode*) para ser usado em simuladores do tipo SPICE (mais detalhes na seção Simulador de circuitos eletrônicos, página 30).

1.4 Estrutura do trabalho

No capítulo 2 será apresentada uma revisão bibliográfica sucinta, de forma a contextualizar o tema. No capítulo 3, as escolhas das ferramentas utilizadas no desenvolvimento do trabalho serão discutidas com detalhes. No capítulo 4, será explicada a modelagem do problema para a aplicação de algoritmos genéticos. No capítulo 5, o modo como o algoritmo genético foi configurado para a aplicação no problema alvo deste trabalho será descrito com maiores detalhes. No capítulo 6, os resultados obtidos em ambiente computacional serão analisados. No capítulo 7, a conclusão da análise dos resultados será apresentada. Por fim, o capítulo 9 relaciona as referências bibliográficas.

2 Revisão dos conceitos abordados

Este trabalho é baseado na aplicação de técnicas de inteligência artificial para a otimização do dimensões de transistores em circuitos CMOS. A técnica escolhida foi algoritmo genético. As ferramentas utilizadas estão descritas neste capítulo.

2.1 Algoritmos Genéticos

Um algoritmo genético é uma técnica de inteligência artificial para realizar buscas por soluções ótimas ou satisfatórias em diversas aplicações. Esta busca baseia-se em heurísticas para tentar convergir rapidamente a um ponto desejado. Porém, no caso do algoritmo genético, estas heurísticas diferem de outras técnicas por serem baseadas no processo da evolução dos seres vivos, conforme observado por Darwin e refinado por cientistas da área ao longo dos anos. Este processo de evolução simulado encontra rapidamente máximos (ou mínimos) locais do domínio de busca, ao mesmo tempo em que não fica estagnada no primeiro máximo (ou mínimo) local que encontrar.

Quando o problema em questão oferece diversas soluções, mas a determinação da ótima não é trivial, temos um candidato forte a otimização por técnicas desta natureza.

Para modelar um problema para este tipo de busca é necessário modelar as possíveis soluções do problema em uma estrutura semelhante a um DNA, composta por cromossomos que são capazes de guardar informações e serem usadas na geração indivíduos. Cada indivíduo é uma possível solução.

Inicialmente uma população inicial é gerada, com cada indivíduo codificado pelo seu DNA. Esta população é submetida a um processo de seleção natural, onde os indivíduos mais adaptados são selecionados em detrimento dos menos adaptados.

Esta seleção ocorre por meio de uma função de aptidão (*fitness function*), que avalia o quão satisfatória é cada uma das soluções, ou seja, cada um dos indivíduos representados por seus DNA. Com esta elite, ocorre a geração de uma nova população através de acasalamentos e mutação.

O processo seleção-geração é repetido até que um determinado critério de parada seja atendido e se obtenha uma população final. O problema deve ser satisfatoriamente solucionado por um ou mais indivíduos presente nesta população final.

2.1.1 Indivíduo

O indivíduo, como já mencionado, é um candidato à solução do problema proposto. Por exemplo: se o problema visado é encontrar um polinômio, de maneira que este passe por um conjunto de pontos definidos, um indivíduo seria qualquer um dos polinômios pertencentes ao

domínio de busca e este pode ser representado pelo conjunto de valores de coeficientes que o descrevem.

O DNA representa a codificação de cada uma das solução e contém a informação mínima necessária para que uma função geradora possa produzir um novo indivíduo (solução). A figura 1 mostra a relação indivíduo e DNA.

Figura 1: Exemplo de indivíduos e seus respectivos DNAs

2.1.2 População e geração

Uma população é composta por um conjunto de indivíduos. Uma geração é similar à uma população, pois também é composta por um conjunto de indivíduos, mas o termo geração normalmente representa uma iteração específica na execução de uma otimização por algoritmo genético.

Por exemplo: suponha uma busca por algoritmo genético que gerou dez gerações. Cada uma destas é uma população diferente, composta por diversos indivíduos. O termo geração agrega uma noção de iteratividade importante para a compreensão do problema.

2.1.3 Função de Aptidão

Uma parte vital do processo de busca por algoritmo genético é a definição de uma boa função de aptidão. Através da sua aplicação, os indivíduos serão julgados como mais ou menos adaptados, permitindo o avanço das novas gerações de populações em direção a máximos (ou mínimos) locais.

A equação 1 mostra uma representação matemática deste processo,

$$f(i) = s, \forall\, i \in D_i, s \in I_S \tag{1}$$

Onde *f* é a função de aptidão; *i* é um indivíduo; *s* é a pontuação deste indivíduo, que representa a sua aptidão; D_i é o domínio de todos os indivíduos possíveis; I_S é o intervalo que abrange as possíveis pontuações que serão utilizadas para ordenar os indivíduos entre os mais e menos adaptados.

Porém, o valor obtido com a aplicação desta função nem sempre é utilizado diretamente na

aplicação do algoritmo genético. É comum utilizar a técnica de normalização, onde a pontuação de cada indivíduo é dividida pela soma da pontuação de todos antes de ser utilizada. Isto permite que características, como a intensidade da seleção (descrita mais adiante), possam ser comparadas com outras implementações de algoritmos genéticos.

Neste trabalho, a definição desta função foi um grande desafio, e será detalhada na seção Avaliação da população, na página 43.

2.1.4 Seleção

A seleção é uma função que imita, no ambiente virtual da modelagem, o processo da seleção natural. Uma população de indivíduos é classificada pela função de aptidão e, através de seu resultado, escolherá uma elite da população. Isto decorre, geralmente, da associação de cada indivíduo com uma pontuação.

Com estes números em mãos, um mecanismo de seleção é aplicado para decidir quais indivíduos serão elitizados e quais serão descartados. Este mecanismo pode ser caracterizado pela intensidade ou pressão de seleção. Apesar deste conceito ser interpretado de formas diferentes por diversos autores, adotarei ao longo do trabalho a abordagem que define a intensidade da seleção como a variação causada na aptidão média de uma população após a aplicação do critério de seleção.

A definição formal da intensidade do teste, baseada na definição de [ZEB02], é dada pela equação 2,

$$I = \frac{M^{*} - M}{\sigma} \tag{2}$$

Onde I é a intensidade do teste; M é a média de aptidão da população atual; M^{*} é a média de aptidão esperada após a seleção; σ é o desvio padrão dos valores de aptidão dos indivíduos da população antes da seleção.

As seleções mais comuns, citadas em literaturas especializadas, são: proporcional, torneio, truncada, classificação linear e classificação exponencial.

O critério escolhido neste trabalho, como será mostrado em seções subsequentes, foi o de seleção truncada. Posteriormente, neste capítulo, este tema será aprofundado.

Seleção Truncada

A seleção ocorre pelo truncamento dos indivíduos mais adaptados da população atual baseado em um limiar T (*threshold*) fixo e predeterminado. Os T indivíduos com maior aptidão são selecionados, ocorrendo a exclusão dos demais. Pode ser demonstrado que a pressão

correspondente deste método é dada pela equação 3,

$$I = \frac{1}{T} \cdot \frac{1}{\sqrt{(2\pi)}} \cdot e^{-\frac{fc^2}{2}} \quad (3)$$

Onde *I* é a intensidade da seleção; f_c é a aptidão mais baixa dos indivíduos selecionados.

Observando a equação, vemos que a intensidade da seleção diminuirá conforme afrouxarmos (aumentarmos) o limiar *T*, conforme o previsto.

Este resultado será utilizado posteriormente no planejamento da estratégia a ser adotada.

2.1.5 Acasalamento

Acasalamento é o processo de geração de novos indivíduos a partir de dois ou mais já existentes (pais). Usualmente, utiliza-se a técnica de cruzamento (*crossover*) para este fim. Esta técnica consiste em misturar o DNA dos pais, como ocorre na natureza, e gerar indivíduos através do produto desta mistura.

Uma forma de fazer o cruzamento é definir pontos de corte no DNA e realizar a permutação das partes seccionadas para gerar a prole. Um ponto de corte é uma posição no DNA do indivíduo que será utilizada como referência para partir o mesmo em duas partes. Segundo [RAN04], gerar dois filhos a partir de dois pais e com apenas um ponto de corte é o usual.

Por exemplo, suponha dois indivíduos com DNAs cujos cromossomos são modelados como caracteres. Neste caso, uma sequência de cinco letras compõe o DNA de um indivíduo. A figura 2 representa graficamente este conceito.

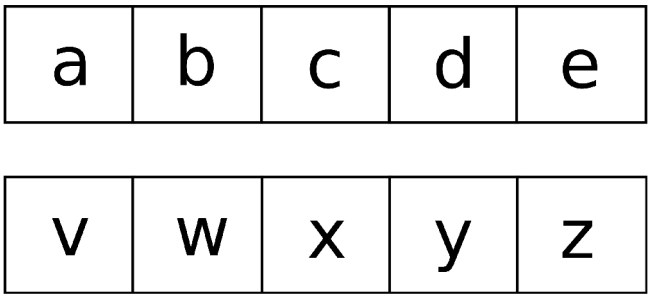

Figura 2: Exemplo de DNA de indivíduos

Caso adotemos um ponto de corte entre o segundo e o terceiro cromossomo, e efetuarmos o cruzamento, obteremos o DNA de dois “filhos” conforme ilustra a figura 3.

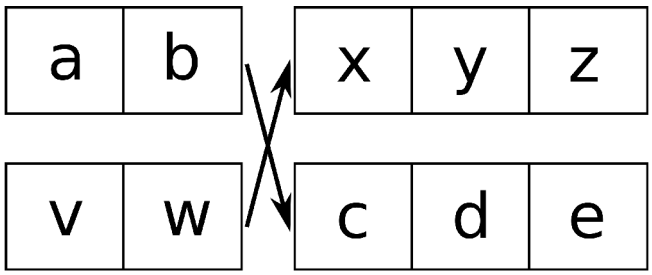

Figura 3: Cruzamento de dois indivíduos

A figura 4 mostra uma representação gráfica de um dos indivíduos gerados por uma aplicação de *crossover* (um ponto de corte aplicado entre o segundo e o terceiro digito).

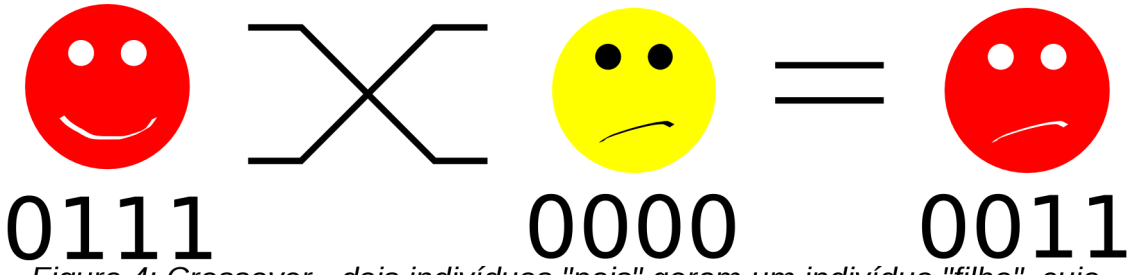

Figura 4: Crossover - dois indivíduos "pais" geram um indivíduo "filho", cujo DNA é uma mistura dos DNAs dos pais

2.1.6 Mutação

É o processo de modificar aleatoriamente o DNA de um indivíduo. Geralmente, é o último passo no processo de geração de uma nova população.

Segundo observações de [ZEB02], algoritmos genéticos que utilizam apenas seleção e acasalamento convergem muito rápido, mas acabam atingindo apenas máximos (ou mínimos) locais.

A operação de mutação permite ao algoritmo um comportamento exploratório mais agressivo no processo de busca, reciclando o material genético ao encontrar indivíduos mais adaptados que jamais seriam gerados na população atual por acasalamento. Por exemplo, pode-se observar o exemplo da figura 4: é impossível gerar um indivíduo com o cromossomo mais à esquerda com o valor *1*, pois nenhum dos pais possui esta característica genética.

A figura 5 representa um universo de busca onde ocorrem diversos máximos locais de amplitudes diferentes. Neste exemplo, um algoritmo genético sem a operação de mutação poderia ficar restrito a encontrar uma solução na região do ponto vermelho, e não encontraria uma possível melhor solução tal qual aquela representada pelo ponto azul.

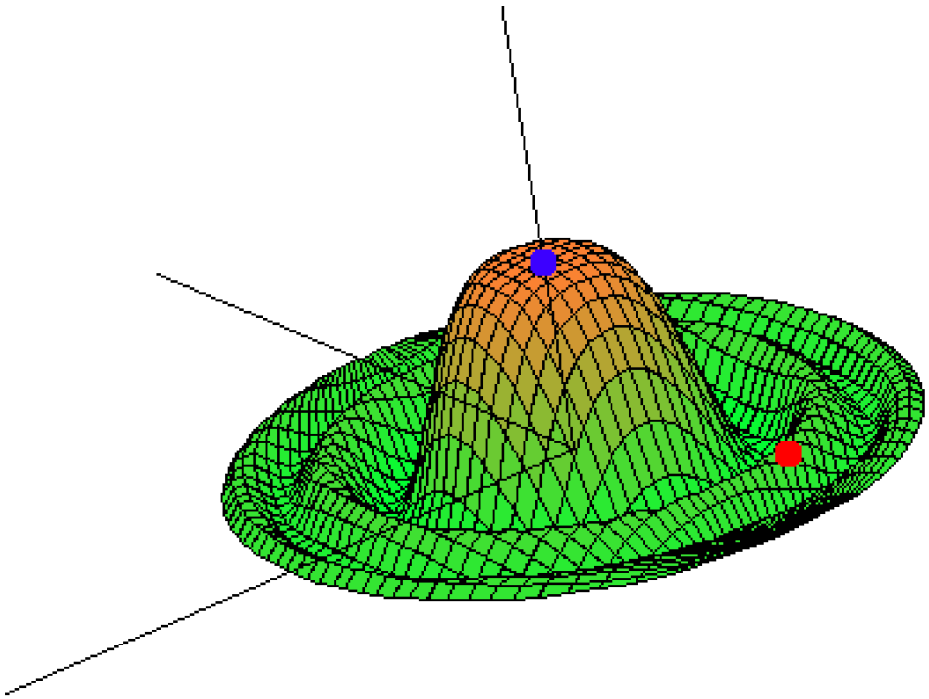

Figura 5: Máximos locais de amplitudes diferentes (o ponto vermelho representa um máximo local e o ponto azul representa outro máximo local, de amplitude maior do que o anterior)

Normalmente, a operação de mutação seleciona aleatoriamente um conjunto de cromossomos de um indivíduo e o modifica de acordo com uma função geradora de cromossomos. Dado um conjunto com *N* indivíduos, representando a geração atual, seleciona-se um número N_m para sofrer mutação. A equação 4 modela formalmente a mutação de uma população.

$$N_m = N \cdot \alpha \tag{4}$$

Onde α é a proporção de mutação.

Escolhidos os indivíduos para sofrerem mutação, um conjunto n_c de cromossomos é selecionado, também aleatoriamente, e estes cromossomos são modificados pela função geradora. A figura 6 ilustra o resultado da mutação em um indivíduo.

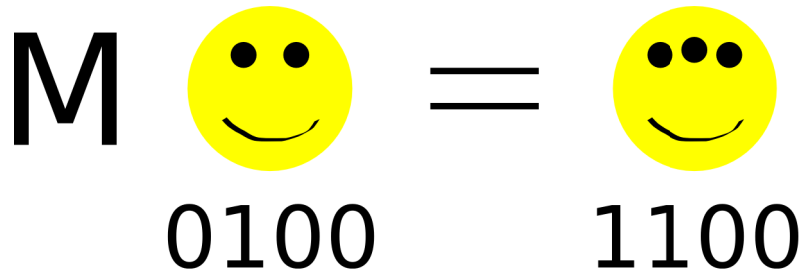

Figura 6: Operação de mutação - um indivíduo novo, que não poderia ser obtido por crossover dos pais dos exemplos anteriores, é gerado

2.1.7 Heurística

É um adjetivo para técnicas baseadas em experiência (geralmente conhecimento de um especialista) para ajudar a resolver problemas.

O objetivo da heurística é diminuir o domínio de busca do problema, de forma a eliminar (sem a necessidade de avaliação computacional) indivíduos que são pouco interessantes.

Um exemplo de heurística é evitar que dimensões de um transistor possam assumir valores impraticáveis, como metros. Um simulador elétrico, SPICE por exemplo, até permite a simulação de tal dispositivo, mas os resultados obtidos seriam inúteis do ponto de vista prático.

2.2 Fonte de corrente

Um desafio de aplicações de microeletrônica é o desenvolvimento de fontes de corrente que funcionam com baixas tensões de alimentação, além de terem alta eficiência e estabilidade na região de operação.

A fonte de corrente que serviu de base para todas as outras utilizadas neste trabalho, mostrada na seção seguinte, foi inspirada nas aplicações utilizadas em [SIL08] e [RAZ03].

2.2.1 Topologia da fonte

A fonte de corrente base consiste de dois transistores do tipo PMOS em configuração de espelho de corrente, e de dois transistores NMOS em configuração de espelho de corrente degenerado. A degeneração se dá pelo uso da resistência, que permite fixar o valor da corrente de dreno no transistor M4. A figura 7 ilustra a topologia da fonte. A corrente de dreno do transistor M4 pode ser facilmente copiada para outras porções de um circuito que possua esta fonte.

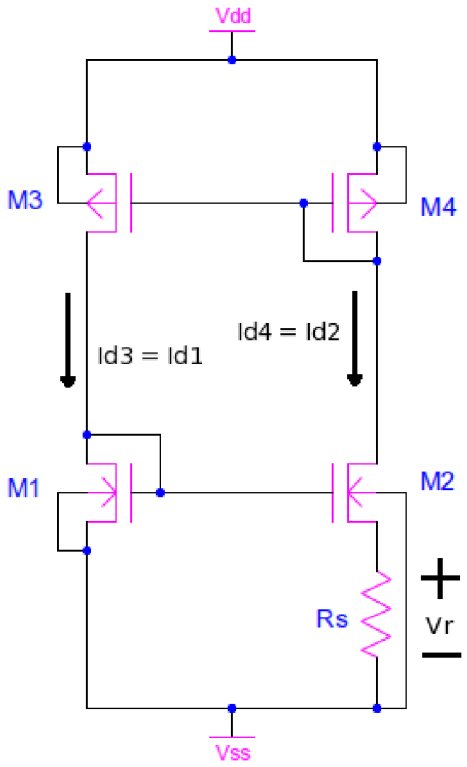

Figura 7: Topologia da fonte de corrente base com pontos de interesse em destaque

Neste circuito, a corrente de dreno em M4 será espelhada para o circuito de carga que consumirá a corrente fornecida. Uma boa forma de obter este valor é determinando a tensão no resistor R_s.

Vamos considerar aqui o caso em que os transistores NMOS operam em inversão fraca. Nesta situação, a corrente gerada, como veremos, é diretamente proporcional à temperatura do circuito, o que é muito útil para a construção de fontes de tensão compensadas com a temperatura [SIL08]. Os transistores PMOS, por outro lado, podem operar tanto em forte como em fraca inversão. A tabela 1 mostra as equações para a corrente do dreno dos MOS para ambos os casos.

Tabela 1: Equações de corrente para ambos os modos de operação de um MOS

Região de operação	Equação da corrente de dreno
Fraca inversão	$I_D = \frac{W}{L} \cdot I_{D0} \cdot e^{\frac{V_G}{n \cdot V_T}} \cdot \left(e^{\frac{-V_S}{V_T}} - e^{\frac{-V_D}{V_T}} \right)$
Forte inversão (região linear)	$I_D = \mu \cdot C_{ox} \cdot \frac{W}{L} \cdot \left(V_{GS} - V_T - \frac{V_{DS}}{2} \right) \cdot V_{DS}$
Forte inversão (região de saturação)	$I_D = \frac{1}{2} \cdot \mu \cdot C_{ox} \cdot \frac{W}{L} \cdot \left(V_{GS} - V_T \right)^2$

Onde I_D é a corrente do dreno; I_{D0} é uma corrente característica da tecnologia de fabricação; n é o fator *slope*; W é a largura do canal do transistor; L é o comprimento do canal do transistor; V_G é a tensão de porta-substrato; V_D é a tensão dreno-substrato; V_S é a tensão fonte-substrato; V_T é a tensão térmica; μ é a constante de mobilidade dos elétrons ou lacunas (dependendo do tipo do transistor); C_{ox} é o valor da capacitância do óxido de porta por unidade de área.

Portanto, obtém-se as correntes de dreno dos transistores por meio das equações 5 e 6,

$$I_{D2} = I_{D4} = I_{D0} \cdot \left(\frac{W_4}{L_4} \right) \cdot e^{\left(\frac{V_{G4}}{n \cdot V_T} - \frac{V_{S4}}{V_T} \right)} = I_{D0} \cdot S_4 \cdot e^{\left(\frac{V_{G4}}{n \cdot V_T} - \frac{V_{S4}}{V_T} \right)} \tag{5}$$

$$I_{D1} = I_{D3} = I_{D0} \cdot \left(\frac{W_3}{L_3} \right) \cdot e^{\left(\frac{V_{G3}}{n \cdot V_T} \right)} = I_{D0} \cdot S_3 \cdot e^{\left(\frac{V_{G3}}{n \cdot V_T} \right)} \tag{6}$$

Onde S é a relação W sobre L dos transistores; W_i é a largura do canal do transistor i; L_i é o comprimento do canal do transistor i; a tensão V_{Gi} é a tensão porta-substrato do transistor i; a tensão V_{Si} é a tensão fonte-substrato do transistor i.

Nota-se que a tensão fonte-substrato do transistor M2 é igual à queda de tensão sobre o resistor R_S, ou seja, $V_{S3} = V_R$. Supondo que a tensão $V_{DS} >> V_T$, onde V_{DS} é a tensão dreno-subtrato do transistor, para os transistores M1 e M2, obtém-se a relação da equação 7,

$$\frac{I_{D3}}{I_{D4}} = \frac{I_{D1}}{I_{D2}} = \frac{S_1 \cdot I_{D0} \cdot e^{\left(\frac{V_{G1}}{n \cdot V_T} \right)}}{S_2 \cdot I_{D0} \cdot e^{\left(\frac{V_{G2}}{n \cdot V_T} - \frac{V_R}{V_T} \right)}} = \frac{S_1}{S_2} \cdot e^{\left(\frac{V_{G1}}{n \cdot V_T} - \frac{V_{G2}}{n \cdot V_T} + \frac{V_R}{V_T} \right)} = \frac{S_1}{S_2} \cdot e^{\left(\frac{V_R}{V_T} \right)} \tag{7}$$

A relação entre o valor W/L do transistor M4 e do M3 é dada pela equação 8,

$$\frac{W_3}{L_3} = M \cdot \frac{W_4}{L_4} \tag{8}$$

Caso os transistores PMOS M3 e M4 sejam iguais, exceto pela relação *W/L* dos mesmos, obtém-se a relação entre suas correntes de dreno na equação 9,

$$I_{D4} = \frac{I_{D3}}{M} \tag{9}$$

Substituindo o resultado da equação 7 em 9, obtém-se a equação 10,

$$\frac{S_1}{S_2} \cdot e^{\left(\frac{V_R}{V_T}\right)} = M \tag{10}$$

A partir da equação 10, podemos obter a equação 11, que modela o comportamento da tensão V_R,

$$V_R = V_T \left[\ln\left(\frac{S_2}{S_1} \cdot \frac{1}{M} \right) \right] \tag{11}$$

Nota-se que a tensão é proporcional a V_T independe da tensão de alimentação. Finalmente, a corrente I_{D4} é dada por meio da equação 12,

$$I_{D4} = \frac{V_R}{R_S} = \frac{V_T \cdot \ln\left(\frac{S_2}{S_1} \cdot \frac{1}{M} \right)}{R_S} \tag{12}$$

Por esta análise simplificada, o valor da corrente fornecida por esta fonte independe da tensão de alimentação. Na prática (através de simulações), percebe-se que o comportamento acaba sendo diferente do esperado, ocorrendo forte dependência. Isto se deve ao efeito da modulação de canal nos transistores que não foi considerada nas relações.

Os circuitos de topologias mais complexas apresentados posteriormente neste trabalho visam atenuar ou até eliminar este problema. Eles apresentam um número maior de transistores operando em configurações *cascode*. Porém, quanto maior o número de transistores empregados no circuito, mais difícil é dimensioná-los de forma a obter o melhor desempenho possível da fonte.

A figura 8 mostra gráfico da corrente de dreno no transistor M4 em função da tensão de alimentação. Ele foi obtido através de simulação de um circuito desta topologia, com os seguintes parâmetros: *W4 = 0,4µm*, *W3 = 0,8µm*, *L3 = L4 = 2,0µm*, *W1 = W2 = 55µm*, *L1 = L2 = 1,0µm*, R_S *= 21500Ω*. A simulação foi feita com o simulador Gnucap, modelo 49 do BSIM3v3 (semelhante ao

HSpice®, com os parâmetros fornecidos pela *AMS*). Mais detalhes sobre as ferramentas de simulação no capítulo 3 , página 30.

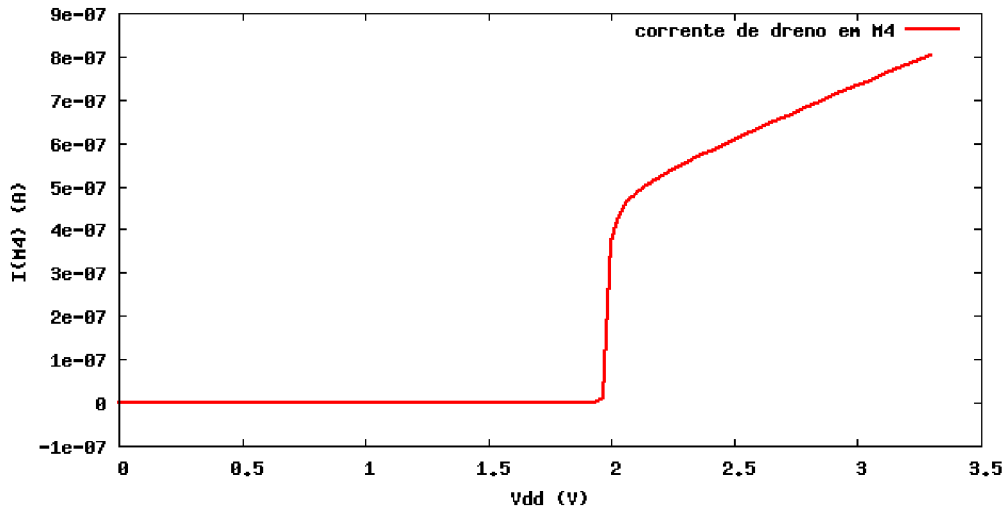

Figura 8: Curva de corrente da fonte base

No gráfico da figura 8, a corrente fornecida pela fonte aparece no eixo *y* em função da variação na tensão de alimentação, mapeada no eixo *x*.

Esta demonstração foi baseada no trabalho [SIL08].

3 Ferramentas utilizadas

Este capítulo descreverá as ferramentas utilizadas para atingir os objetivos propostos no trabalho.

3.1 Simulador de circuitos eletrônicos

Simuladores da família SPICE (*Simulation Program with Integrated Circuit Emphasis*) são os mais utilizados para a simulação de circuitos eletrônicos analógicos. Exemplos de simuladores: HSpice®, da empresa Synopsys®; ELDO®, da empresa Mentor Graphics®; Gnucap e NGSpice, sendo estes de código livre e desenvolvidos pela comunidade.

Neste trabalho, foi utilizado o simulador Gnucap devido à facilidade de instalação e integração do mesmo com o *software* desenvolvido.

3.1.1 Linguagem

A linguagem utilizada pelos simuladores da família SPICE varia entre as implementações. Não existe um padrão da indústria, mas a maioria delas implementa a sintaxe do antigo *software* Spice3.

Abaixo consta um exemplo de código SPICE. Ele representa um filtro passa-baixa simples, apresentado em [ECC10].

```
* LPFILTER.CIR - SIMPLE RC LOW-PASS FILTER
VS 1 0 AC 1 SIN(0VOFF 1VPEAK 2KHZ)
R1 1 2 1K
C1 2 0 0.032UF

.AC DEC 5 10 10MEG
.TRAN 5US 500US

.PRINT AC VM(2) VP(2)
.PRINT TRAN V(1) V(2)

.PROBE
.END
```

Algumas regras da linguagem são:

- a primeira linha é sempre de comentário;
- o primeiro caractere de cada linha indica a função: “*” indica comentário; “.” indica linha de comando; “M” indica especificação de um transistor MOS; “C” indica especificação de um capacitor; “R” indica a especificação de um resistor; “V” indica especificação de uma fonte de tensão.
- o arquivo deve terminar com “.end” ;

- não ha distinção entre letras maiúsculas ou minúsculas.

Informações sobre o significado dos termos utilizados neste trecho podem ser encontrados em [QUA93].

3.2 Ambiente de desenvolvimento

Implementações que utilizam inteligência artificial são geralmente bastante simbólicas. Portanto, uma linguagem de programação com uma abordagem desta natureza é indicada para uma modelagem mais adequada do problema.

Por este motivo, a linguagem escolhida para este trabalho foi *Scheme:* um dialeto minimalista de *LISP* (*List Processing*). Esta linguagem possui diversas ferramentas desenvolvidas pela comunidade de *software* livre, tornando confortável o ato de programar aplicações com ênfase em símbolos. A implementação utilizada foi a *PLT-Scheme*.

A representação voltada aos símbolos facilita a forma de pensar do programador, que pode operar com estruturas não numéricas, como *"bananas"* ou *"laranjas"*. No contexto deste trabalho, esta habilidade foi utilizada para modelar os componentes de circuitos e as ideias abstratas do funcionamento de um algoritmo genético. Isto fez com que a implementação da otimização seja bastante genérica, e pode ser aplicada a outros trabalhos com pouco esforço.

Mais informações sobre esta linguagem e suas ferramentas podem ser encontradas em [DYB09].

4 Modelagem do Problema

Para aplicar otimização por algoritmos genéticos é necessário modelar o problema no universo desta técnica.

4.1 Dimensões de um transistor CMOS

A otimização deste trabalho é aplicada nas dimensões dos transistores CMOS de um circuito integrado e na variação do valor de uma resistência.

As dimensões escolhidas foram a largura *W* e o comprimento *L* do canal do transistor, conforme ilustra a figura 9.

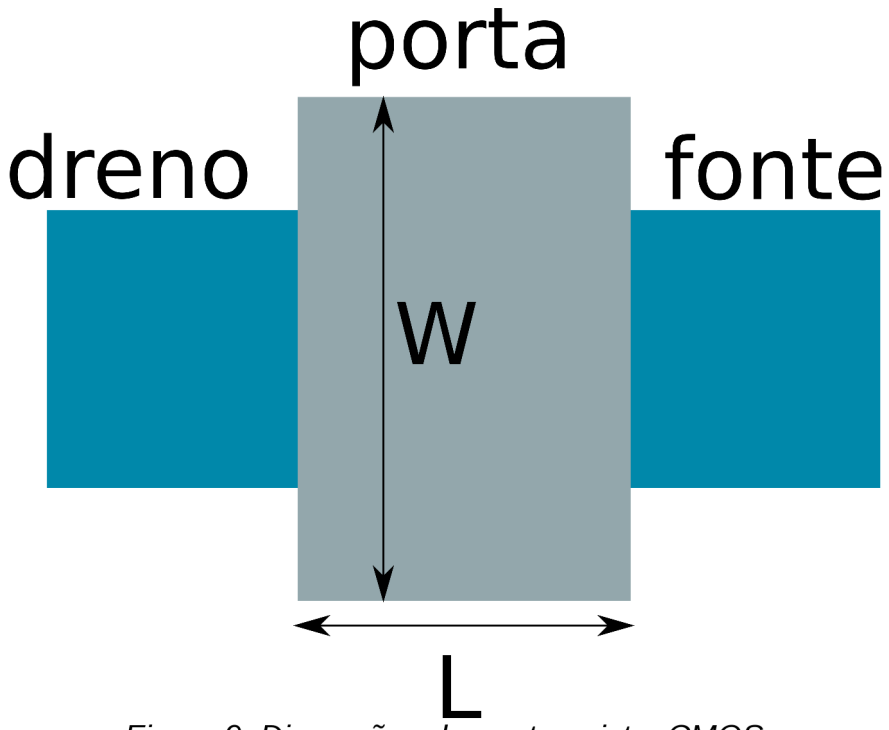

Figura 9: Dimensões de um transistor CMOS

Estas dimensões afetam diretamente o comportamento deste dispositivo.

4.2 Circuitos como indivíduos

Neste trabalho, um circuito é representado como o arquivo SPICE que o modela (uma fonte de corrente, por exemplo). Na proposta aqui desenvolvida, estes circuitos serão de topologia fixa, ou seja, os elementos que compõe os circuitos não serão removidos ou substituídos. Eles terão os seus parâmetros modificados.

Como apenas os parâmetros são modificados, um conjunto de valores para estes representará um indivíduo e será o próprio DNA dele. A partir deste DNA serão gerados tanto arquivos para o simulador SPICE, que o modelará eletricamente, (mais informações na página 25) como novos indivíduos.

Nos exemplos aqui utilizados (fontes de corrente), o objetivo será otimizar o dimensionamento dos transistores CMOS presentes nestes circuitos. Os simuladores SPICE possuem parâmetros específicos para a representação destas dimensões em seus modelos de componentes.

4.3 Fontes de corrente

Um bom exemplo de uma classe de problemas de projeto que necessitam de experiência por parte do projetista é a confecção de uma fonte de corrente para o contexto da microeletrônica.

Além da topologia, o dimensionamento dos transistores CMOS utilizados causa forte influência na qualidade da fonte. Uma escolha ruim pode torná-la menos eficiente e, possivelmente, inútil.

A escolha de boas dimensões depende de modelos complexos. O casamento das dimensões dos transistores busca melhorar o desempenho da fonte, de forma que ela funcione com tensões de alimentação menores e forneça correntes mais constantes (independem de variações na alimentação) independentemente de possíveis variações de componentes na fabricações.

Além disso, existem limitações do processo de fabricação que restringem a escolha das dimensões.

Neste trabalho, quatro fontes de correntes tiveram as dimensões de seus transistores otimizadas. Elas foram batizadas, para uso interno neste documento, como “alpha”, “beta”, “gamma” e “delta”.

O DNA de um indivíduo de fonte é caracterizado pelos valores das larguras e comprimentos de canal dos seus transistores e valor do resistor.

4.3.1 Fonte Alpha

A fonte *alpha* é a topologia mais simples dentre as consideradas. Ela utiliza quatro transistores (dois NMOS e dois PMOS) e gera uma corrente de referência no canal do transistor M4, conforme ilustra a figura 10. O equacionamento desta fonte foi visto anteriormente.

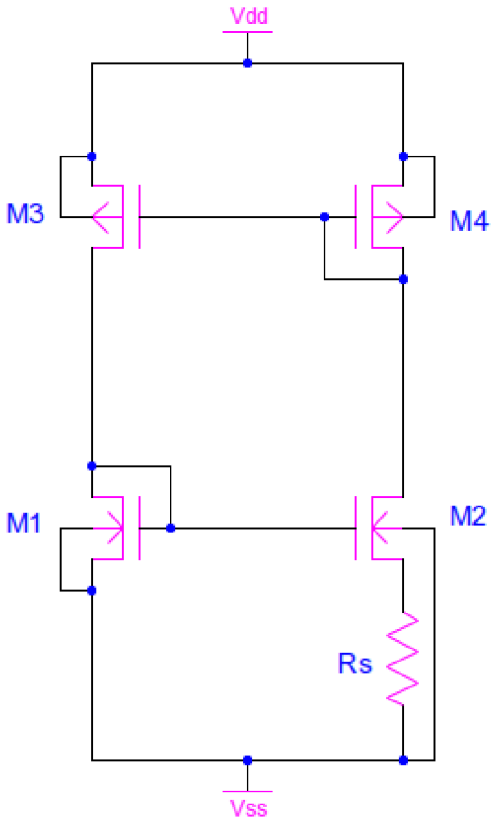

Figura 10: Fonte de corrente "Alpha"

Um indivíduo desta fonte foi modelado com um DNA de cinco cromossomos (conforme a modelagem previamente especificada). As relações entre os *Ws* dos transistores NMOS e entre os *Ws* dos PMOS foram mantidos fixas. Os parâmetros contidos no modelo de cinco cromossomos são

$$\mathbf{L_n \mid W_n \mid L_p \mid W_p \mid R_S}$$

Onde L_n representa o comprimento dos transistores NMOS; W_n representa a largura dos transistores NMOS; L_p representa o comprimento dos transistores PMOS; W_p representa a largura dos transistores PMOS; R_S representa o valor da resistência.

Neste caso, a heurística é considerar o *L* dos transistores NMOS iguais entre si e o *L* dos transistores PMOS iguais entres si. Estas restrições reduzem o número de indivíduos investigados, afastando aqueles sabidamente inadequados à solução do problema (mais detalhes sobre heurísticas na página 25). A relação *X* entre W_{p3}/L_{p3} e W_{p4}/L_{p4} foi utilizada com valor 2. A relação *Y* entre W_{n1}/L_{p1} e W_{n2}/L_{p2} foi utilizada com valor 1.

Numerando estes cromossomos de 1 a 5, da esquerda para a direita, pode-se construir um indivíduo a partir deste DNA conforme o código SPICE a seguir.

```
* COGA generated circuit - <INDIVÍDUO>.cir
M4 Vdd 2 2 Vdd MODP L=<CROMOSSOMO 3> W=<CROMOSSOMO 4>
```

```
M3 1 2 Vdd Vdd MODP L=<CROMOSSOMO 3> W=<CROMOSSOMO 4> M=2
M2 2 1 3 Vss MODN L=<CROMOSSOMO 1> W=<CROMOSSOMO 2>
M1 Vss 1 1 Vss MODN L=<CROMOSSOMO 1> W=<CROMOSSOMO 2>
R Vss 3 <CROMOSSOMO 5>

Vgnd Vss 0 0.0V
Vpow Vdd 0 3.3V

.MODEL MODP PMOS LEVEL=49 <PARÂMETROS>
.MODEL MODN NMOS LEVEL=49 <PARÂMETROS>

.PRINT DC I(M4)
.DC Vpow 0V 3.3V 0.01V > <INDIVÍDUO>.out
.END
```

Os termos <CROMOSSOMO i>, onde *i* varia de 1 a 5, correspondem aos campos que serão modificados. O termo <INDIVÍDUO> representa o nome do indivíduo, na forma "*ln-wn-lp-wp-rs"*.

O termo <PARÂMETROS> indica os parâmetros de simulação para os transistores da tecnologia *AMS 0,35µm*, fornecidos pela *AMS*.

4.3.2 Fonte Beta

A fonte *beta* possui algumas melhorias com relação à fonte *alpha*, como maior estabilidade ao custo de tensões de alimentação maiores. Ela utiliza seis transistores (três NMOS e três PMOS) e gera uma corrente de referência no canal do transistor M6, conforme ilustra a figura 11.

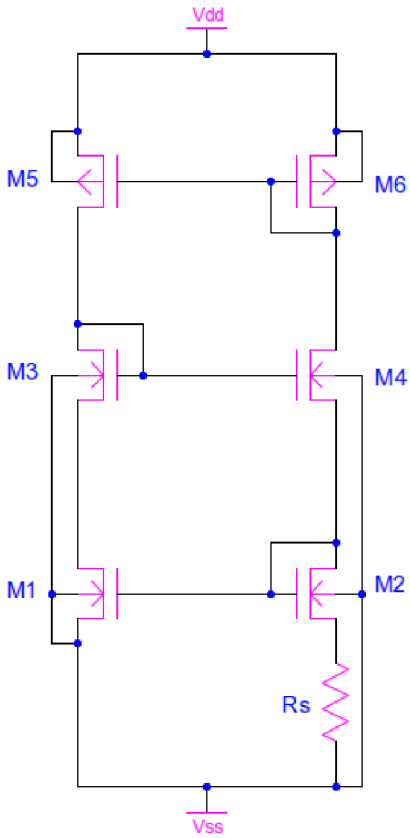

Figura 11: Fonte de corrente "Beta"

Um indivíduo desta fonte foi modelado com um DNA de cinco cromossomos (conforme a modelagem previamente especificada).

De forma análoga à fonte *alpha*, a representação de cinco cromossomos é

$$\mathbf{L_n \mid W_n \mid L_p \mid W_p \mid R_S}$$

Onde L_n representa o comprimento dos transistores NMOS; W_n representa a largura dos transistores NMOS; L_p representa o comprimento dos transistores PMOS; W_p representa a largura dos transistores PMOS; R_S representa o valor da resistência.

Neste caso, a heurística é considerar $L_{n1}=L_{n2}=L_{n3}=L_{n4}$, $W_{n4}=W_{n3}=W_{n2}=W_{n1}$ (mais detalhes sobre heurísticas na página 25).

A relação X entre W_{p5}/L_{p5} e W_{p6}/L_{p6} foi utilizada com valor 2. A relação Y entre W_{n1}/L_{p1} e W_{n2}/L_{p2} foi utilizada com valor 1.

Numerando estes cromossomos de 1 a 5, da esquerda para a direita, pode-se construir um indivíduo a partir deste DNA conforme o código SPICE a seguir.

```
" COGA generated circuit - <INDIVÍDUO>.cir
M1 Vss 1 2 Vss MODN L=<CROMOSSOM 1> W=<CROMOSSOM 2>
M2 1 1 3 Vss MODN L=<CROMOSSOM 1> W=<CROMOSSOM 2>
M3 2 4 4 Vss MODN L=<CROMOSSOM 1> W=<CROMOSSOM 2>
```

```
M4 5 4 1 Vss MODN L=<CROMOSSOM 1> W=<CROMOSSOM 2>
M5 4 5 Vdd Vdd MODP L=<CROMOSSOM 3> W=<CROMOSSOM 4> M=2
M6 Vdd 5 5 Vdd MODP L=<CROMOSSOM 3> W=<CROMOSSOM 4>
Rs Vss 3 <CROMOSSOM 5>

Vgnd Vss 0 0.0V
Vpow Vdd 0 3.3V

.MODEL MODP PMOS LEVEL=49 <PARÂMETROS>
.MODEL MODN NMOS LEVEL=49 <PARÂMETROS>

.PRINT DC I(M6)
.DC Vpow 0V 3.3V 0.01V > <INDIVÍDUO>.out
.END
```

Os termos <CROMOSSOMO i>, onde *i* varia de 1 a 5, correspondem aos campos que serão modificados. O termo <INDIVÍDUO> representa o nome do indivíduo, na forma "*ln-wn-lp-wp-rs"*.

O termo <PARÂMETROS> indica os parâmetros de simulação para os transistores da tecnologia *AMS 0,35µm*, fornecidos pela *AMS*.

4.3.3 Fonte Gamma

A fonte *gamma* possui algumas melhorias com relação à fonte *beta*, visando desempenho similar, mas com tensão de alimentação menor. Esta topologia utiliza um circuito de polarização adicional, transistores M7 e M8, que polariza os transistores cascode M3 e M4 e garante uma correta tensão no dreno dos transistores M1 e M2. Ela utiliza oito transistores (quatro NMOS e quatro PMOS) e gera uma corrente de referência no canal do transistor M6, conforme ilustra a figura 12.

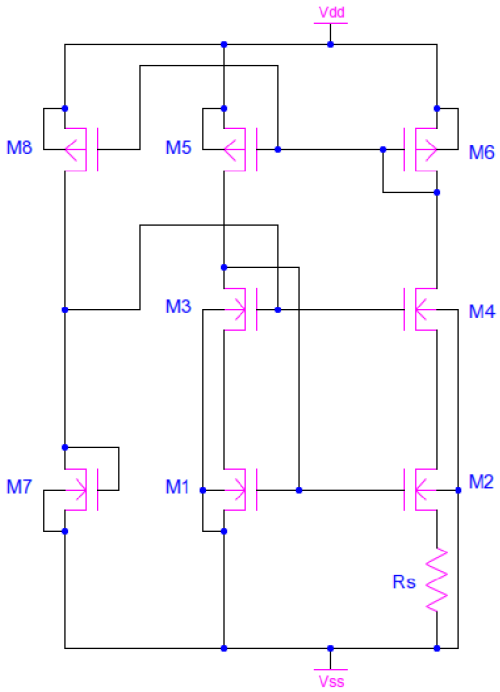

Figura 12: Fonte de corrente "Gamma"

Um indivíduo desta fonte foi modelado com um DNA de dez cromossomos (conforme a modelagem previamente especificada).

Os parâmetros contidos nos dez cromossomos são

$$\mathbf{L_{n\text{-}1\text{-}2} \mid L_{n\text{-}3\text{-}4} \mid W_{n\text{-}1\text{-}3} \mid W_{n\text{-}2\text{-}4} \mid L_{p\text{-}5\text{-}6\text{-}8} \mid W_{p\text{-}5\text{-}6} \mid W_{p\text{-}8} \mid L_{n\text{-}7} \mid W_{n\text{-}7} \mid R_S}$$

Onde $L_{n\text{-}i}$ representa o comprimento dos transistores NMOS de número i; $W_{n\text{-}i}$ representa a largura dos transistores NMOS de número i; $L_{p\text{-}i}$ representa o comprimento dos transistores PMOS de número i; $W_{p\text{-}i}$ representa a largura dos transistores PMOS de número i; R_S representa o valor da resistência.

Neste caso, a heurística é considerar $L_{n1}=L_{n2}$, $L_{n3}=L_{n4}$, $W_{n1}=W_{n3}$, $W_{n2}=W_{n4}$, $L_{p5}=L_{p6}=L_{p8}$ e $W_{p5}=W_{p6}$.

A relação X entre W_{p5}/L_{p5} e W_{p6}/L_{p6} foi utilizada com valor 2. A relação Y entre W_{n1}/L_{p1} e W_{n2}/L_{p2} foi utilizada com valor 1.

Numerando estes cromossomos de 1 a 10, da esquerda para a direita, pode-se construir um indivíduo a partir deste DNA conforme o código SPICE a seguir.

```
" COGA generated circuit - <INDIVÍDUO>.cir
M1 Vss 1 2 Vss MODN L=<CROMOSSOMO 1> W=<CROMOSSOMO 3>
M2 3 1 4 Vss MODN L=<CROMOSSOMO 1> W=<CROMOSSOMO 4>
M3 2 6 1 Vss MODN L=<CROMOSSOMO 2> W=<CROMOSSOMO 3>
M4 5 6 3 Vss MODN L=<CROMOSSOMO 2> W=<CROMOSSOMO 4>
M5 1 5 Vdd Vdd MODP L=<CROMOSSOMO 5> W=<CROMOSSOMO 6> M=2
M6 Vdd 5 5 Vdd MODP L=<CROMOSSOMO 5> W=<CROMOSSOMO 6>
```

```
M7 Vss 6 6 Vss MODN L=<CROMOSSOMO 8> W=<CROMOSSOMO 9>
M8 6 5 Vdd Vdd MODP L=<CROMOSSOMO 5> W=<CROMOSSOMO 7>
Rs Vss 4 <CROMOSSOMO 10>

Vgnd Vss 0 0.0V
Vpow Vdd 0 3.3V

.MODEL MODP PMOS LEVEL=49 <PARÂMETROS>
.MODEL MODN NMOS LEVEL=49 <PARÂMETROS>

.PRINT DC I(M6)
.DC Vpow 0V 3.3V 0.01V > <INDIVÍDUO>.out
.END
```

Os termos <CROMOSSOMO i>, onde *i* varia de 1 a 10, correspondem aos campos que serão modificados. O termo <INDIVÍDUO> representa o nome do indivíduo, na forma "*ln12-ln34-wn13-wn24-lp568-wp56-wp8-ln7-wn7-rs*".

O termo <PARÂMETROS> indica os parâmetros de simulação para os transistores da tecnologia *AMS 0,35µm*, fornecidos pela *AMS*.

4.3.4 Fonte Delta

A fonte *delta* possui algumas melhorias com relação à fonte *gamma*, e esta possui desempenho similar, mas com uma tensão de alimentação menor ainda. Esta topologia utiliza dois circuitos de polarização para atingir tal objetivo. Ela utiliza dez transistores (cinco NMOS e cinco PMOS) e gera uma corrente de referência no canal do transistor M6, conforme ilustra a figura 13.

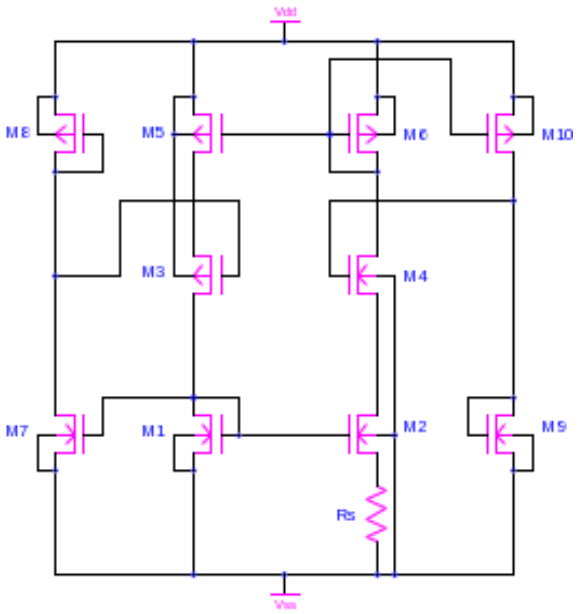

Figura 13: Fonte de corrente "Delta"

Um indivíduo desta fonte foi modelado com um DNA de quinze cromossomos (conforme a modelagem previamente especificada).

Os parâmetros contidos nos quinze cromossomos são mostrados abaixo.

$L_{n-1-2-7}$ | L_{n-4} | W_{n-1} | W_{n-2-4} | W_{n-7} | L_{p-3} | $L_{p-5-6-10}$ | W_{p-5-3} | W_{p-6} | W_{p-10} | L_{n-8} | W_{p-8} | L_{n-9} | W_{n-9} | R_S

Onde L_{n-i} representa o comprimento dos transistores NMOS de número *i*; W_{n-i} representa a largura dos transistores NMOS de número *i*; L_{p-i} representa o comprimento dos transistores PMOS de número *i*; W_{p-i} representa a largura dos transistores PMOS de número *i*; R_S representa o valor da resistência.

Neste caso, a heurística é considerar $L_{n1}=L_{n2}=L_{n7}$, $W_{n2}=W_{n4}$, $L_{p5}=L_{p6}=L_{p10}$ e $W_{p5}=W_{p3}$.

A relação *X* entre W_{p5}/L_{p5} e W_{p6}/L_{p6} foi utilizada com valor 2. A relação *Y* entre W_{n1}/L_{p1} e W_{n2}/L_{p2} foi utilizada com valor 1.

Numerando estes cromossomos de 1 a 15, da esquerda para a direita, pode-se construir um indivíduo a partir deste DNA conforme o código SPICE a seguir.

```
" COGA generated circuit - <INDIVÍDUO>.cir
M1 Vss 1 1 Vss MODN L=<CROMOSSOMO 1> W=<CROMOSSOMO 3>
M10 Vdd 5 6 Vdd MODP L=<CROMOSSOMO 7> W=<CROMOSSOMO 10>
M2 2 1 3 Vss MODN L=<CROMOSSOMO 1> W=<CROMOSSOMO 4>
M3 1 7 4 Vdd MODP L=<CROMOSSOMO 6> W=<CROMOSSOMO 8>
M4 5 6 2 Vss MODN L=<CROMOSSOMO 2> W=<CROMOSSOMO 4>
M5 4 5 Vdd Vdd MODP L=<CROMOSSOMO 7> W=<CROMOSSOMO 8> M=4
M6 Vdd 5 5 Vdd MODP L=<CROMOSSOMO 7> W=<CROMOSSOMO 9>
M7 Vss 1 7 Vss MODN L=<CROMOSSOMO 1> W=<CROMOSSOMO 5>
M8 7 7 Vdd Vdd MODP L=<CROMOSSOMO 11> W=<CROMOSSOMO 12>
M9 6 6 Vss Vss MODN L=<CROMOSSOMO 13> W=<CROMOSSOMO 14>
Rs Vss 3 <CROMOSSOMO 15>

Vgnd Vss 0 0.0V
Vpow Vdd 0 3.3V

.MODEL MODP PMOS LEVEL=49 <PARÂMETROS>
.MODEL MODN NMOS LEVEL=49 <PARÂMETROS>

.PRINT DC I(M6)
.DC Vpow 0V 3.3V 0.01V > <INDIVÍDUO>.out
.END
```

Os termos <CROMOSSOMO i>, onde *i* varia de 1 a 15, correspondem aos campos que serão modificados. O termo <INDIVÍDUO> representa o nome do indivíduo, na forma “ln127-ln4-wn1-wn24-wn7-lp3-lp5610-wp53-wp6-wp10-ln8-wp8-ln9-wn9-rs”.

O termo <PARÂMETROS> indica os parâmetros de simulação para os transistores da tecnologia *AMS 0,35µm*, fornecidos pela *AMS*.

5 Otimização por Algoritmo Genético

No contexto deste trabalho, otimizar um circuito significa buscar combinações de seus parâmetros relevantes (dimensão dos transistores CMOS) de modo que o seu desempenho melhore. A definição de um bom desempenho varia para cada aplicação.

Para o caso de nossa fonte de corrente, bom desempenho é funcionar com baixa tensão de alimentação ao mesmo tempo que a corrente dependa pouco deste fator (estável). O algoritmo genético deve ser capaz de projetar uma fonte que forneça uma corrente especificada pelo projetista.

5.1 Execução preliminar

Antes de começar o processo de otimização das fontes, foi necessário realizar execuções iniciais no caso mais simples (fonte de topologia Alpha) para tomar algumas decisões baseadas no comportamento observado.

5.1.1 Escolha dos pesos

O primeiro fator foi a escolha dos pesos w_1 e w_2 (detalhes sobre o seu uso na página 43). Os pesos utilizados não são normalizados, e foi feito um esforço para mantê-los em ordens de grandeza próximas, mas de forma que um tenha prioridade sobre o outro.

A diferença de *7* ordens de grandeza em $w_1=1$ e $w_2=-1.10^7$ garantiu, nas execuções observadas, uma diferença de duas ordens de grandeza em relação ao peso w_2. Ou seja, foi dada prioridade para que a fonte gerada respeite mais a especificação do valor da corrente a ser fornecida do que a capacidade de operar com tensões de alimentação menores.

O valor negativo do peso w_2 se dá pela forma como o valor da corrente é avaliado, detalhado na seção Estratégia de otimização (página 42).

5.1.2 Intensidade da seleção

O cálculo da intensidade da seleção quantifica a intensidade da variação de uma geração para a outra. Através de uma média deste número em algumas poucas execuções preliminares, foi possível tomar decisões com relação ao método de seleção e ao tamanho das populações.

A tabela 2 mostra uma relação dos métodos de seleção (discutidos na página 45), combinados com diferentes tamanhos de população, e a respectiva média da intensidade da seleção para 10 execuções. As pontuações dos indivíduos de cada execução foram normalizadas apenas para o cálculo da intensidade, pois a normalização torna a execução mais lenta sem fornecer benefícios práticos no caso da otimização.

Tabela 2: Relação dos métodos de seleção com o tamanho da população e a intensidade da seleção

Método de seleção	Tamanho da população	Média da Intensidade da Seleção
Truncada convencional	10	0,13
Truncada convencional	20	0,07
Truncada convencional	30	0,04
Truncada convencional	40	0,03
Truncada modificada	10	0,19
Truncada modificada	20	0,10
Truncada modificada	30	0,07
Truncada modificada	40	0,05

Os valores da intensidade da seleção vão diminuindo conforme o aumento do tamanho da população. Isto ocorre por tomarmos uma elite de *T* indivíduos proporcional ao tamanho da população (explicado com mais detalhes na página 45).

Por este motivo, o tamanho de população 40 apresentou uma intensidade muito baixa e foi descartado. Dentre os dois métodos de seleção selecionados, o da seleção truncada modificada apresentou maior intensidade e será selecionado como forma de buscar soluções de forma mais agressiva.

5.2 Estratégia de otimização

A estratégia utilizada neste trabalho instancia uma técnica de algoritmo genético para esta aplicação, seguindo sugestões de [ZEB02] e [RAN04].

5.2.1 População inicial

O primeiro passo é a geração de uma população inicial de indivíduos. Para gerar um indivíduo desta população, é utilizada uma função geradora de parâmetros. Cada parâmetro é gerado aleatoriamente, dentro de um intervalo considerado razoável do ponto de vista da aplicação. No caso de dimensões dos transistores, este razoável seriam valores de *L* variando de 0,35µm a 5,0µm e valores de *W* variando de 2µm a 100µm. Para o valor da resistência, eles variam de 5000 a 45000Ω.

O número de indivíduos na população afeta diretamente o desempenho da busca na seleção utilizada (mais detalhes sobre a seleção adotada na página 45). Este impacto pode ser analisado por meio da intensidade do teste (mais detalhes sobre a intensidade do teste na página 21).

5.2.2 Avaliação da população

A cada iteração os indivíduos da população são avaliados pela função de aptidão (detalhes teóricos na página 20). Esta função atribuirá uma pontuação para cada um dos indivíduos, de forma que o mais apto é o de maior pontuação.

Após esta classificação, a população é ordenada do indivíduo mais apto ao menos como preparação para a etapa de seleção. A implementação desta função pode ser bastante complexa, visto que o *software* capaz de avaliar um indivíduo deve agregar o conhecimento específico do domínio da aplicação empregada, sendo que esta geralmente é realizada por um ser humano. Existem estudos dedicados somente à criação destas funções. O estudo [BAR02] dedica-se a otimizar a criação destas funções por meio de modificações técnicas nas implementações, como a diminuição de condicionais *if-then* em cascata (*if* dentro de *if*), e sugere que podem ser obtidos ganhos grandes com este tipo de cuidado. Já o estudo [SAN02] utiliza algoritmos genéticos para obter funções de aptidão para uso em outros algoritmos genéticos.

As próximas seções descrevem as peculiaridades da função de aptidão aplicada para cada um dos circuitos de interesse.

Função de aptidão utilizada

Conforme descrito anteriormente, uma boa fonte de corrente funciona com baixa tensão de alimentação e fornece uma corrente pouco dependente de variações no valor desta mesma tensão. A função de aptidão irá obter dois números de uma curva I_{out} *x* V_{DD} para uso no cálculo da pontuação de cada indivíduo. Chamá-los-ei de n_1 e n_2.

O primeiro é relacionado ao valor de V_{DD} para o qual a corrente atinge um valor fora de um intervalo onde ela é considerada estável. Por exemplo: se, com alimentação máxima (*3,3V*), a fonte fornece uma corrente *I*, e o fator de tolerância de variação ζ for *10%*, então este número será o primeiro valor de V_{DD}, do maior para o menor, no qual a corrente saiu do intervalo *[I + 10%.I ; I – 10%.I]*. As equações 13 e 14 expressam este comportamento,

$$I(V_{DD_{INSTAVEL}}) \in [I_{REF} + (I_{REF} \cdot \Gamma), I_{REF} - (I_{REF} \cdot \Gamma)], tal\ que\ I_{REF} = I(V_{DD_{MAX}}) \quad (13)$$

$$n_1 = V_{DD_{INSTÁVEL}} - V_{DD_{MAX}} \quad (14)$$

Onde $V_{DD\ INSTÁVEL}$ é o valor de V_{DD} para o qual a corrente sai do limiar de estabilidade; $V_{DD\ MAX}$ é o valor máximo de V_{DD} na simulação, definido como *3,3V*.

O segundo número é o módulo da diferença entre a corrente no ponto de tensão máxima de alimentação $V_{DD\ MAX}$ e a corrente escolhida pelo projetista a ser fornecida pela fonte. Seu

comportamento pode ser expresso pela equação 15,

$$n_2 = \| I(V_{DD_{MAX}}) - I_{ALVO} \| \quad (15)$$

Onde $I(V_{DD\ MAX})$ é a corrente no ponto de maior tensão de alimentação; I_{ALVO} é a corrente definida pelo projetista.

Por fim, o valor de aptidão de um indivíduo consiste na soma de n_1 e n_2 após a aplicação de pesos de ajuste de ordem de grandeza e sinal, para que ambos os parâmetros afetem a escolha do melhor indivíduo e que façam isso de forma positiva ou negativa. Chamando estes pesos de w_1 e w_2, a função final de aptidão é dada pela equação 16,

$$f(ind) = w_1 \cdot n_1 + w_2 \cdot n_2 \quad (16)$$

Ao contrário do que foi descrito na seção Função de Aptidão (página 20), este valor não será normalizado para melhorar a velocidade de execução do algoritmo. A normalização será utilizada apenas pontualmente para calcular a intensidade de seleção, quando necessário.

Os valores definidos para os pesos foram $w_1=1$ e $w_2=-1.10^7$ (ajustado empiricamente e que resultam em fontes mais estáveis).

Neste trabalho, definimos como uma boa fonte de corrente uma fonte que respeite sua especificação de corrente e trabalhe com tensões de alimentação o menor possíveis. A figura 14 apresenta um gráfico de um exemplo de fonte de corrente útil por esta perspectiva.

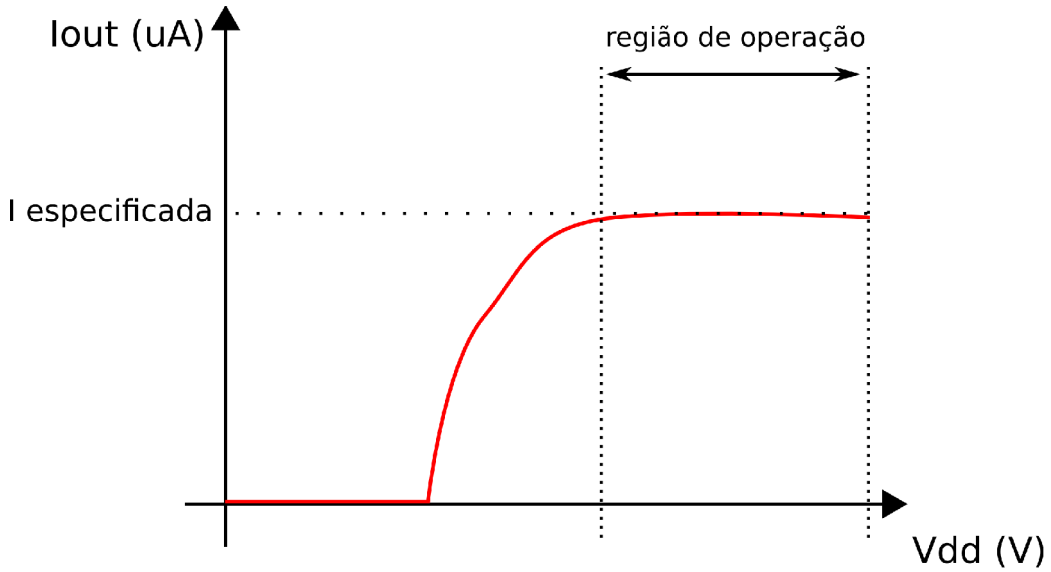

Figura 14: Exemplo gráfico de uma boa fonte de corrente

Nota-se, neste gráfico, que a corrente na região de operação é estável (independe de variações

no *Vdd*). Este é o tipo de curva de corrente por tensão de alimentação que desejamos encontrar em nossas fontes otimizadas e é isso que a função de aptidão busca quantificar. Porém, como a função de aptidão será utilizada durante a execução, ela deve ser simples para consumir pouco processamento computacional e permitir execuções mais rápidas. A função de aptidão descrita nesta seção apresenta boa classificação de curvas desse tipo, ao mesmo tempo em que seu desempenho é satisfatório.

5.2.3 Seleção

A seleção foi feita pelo método de truncamento, mas de forma modificada. Normalmente, a seleção por truncamento escolhe os *T* melhores indivíduos de uma população para serem a elite. Esta será a responsável por gerar a prole que comporá a geração seguinte. Chamaremos este método de seleção truncada convencional.

Porém, neste trabalho, este método foi um pouco modificado de forma a aumentar a agressividade do algoritmo (que converge mais rapidamente). Esta mudança pode ser evidenciada através do cálculo da intensidade da seleção, detalhado na seção Intensidade da seleção (página 41).

A mudança feita foi a seguinte: ao invés de selecionar os *T* melhores indivíduos, seleciona-se os *T - 1* melhores e adiciona-se à elite o pior indivíduo da geração atual. Este método será chamado de seleção truncada modificada.

Com esta modificação, conforme explicitado na seção Intensidade da seleção (página 41), a variedade do material genético entre duas gerações consecutivas é maior, e o algoritmo mostrou-se mais eficiente (menor tempo de convergência) desta forma. Os resultados preliminares encontrados com esta configuração foram satisfatórios.

5.2.4 Criação da prole

O número de indivíduos da população é fixo e especificado antes do início da busca. A cada geração, o primeiro passo para a criação da geração seguinte é a escolha de uma elite de indivíduos da geração atual.

Através desta elite, são criados os indivíduos restantes para manter o tamanho da população de cada geração fixo. Por exemplo: se a elite de uma população de 20 indivíduos é composta por 4, então estes 4 serão utilizados para gerar os 16 restantes (prole), de forma a obtermos novamente uma população de 20 indivíduos.

A prole é gerada através da criação de pares de indivíduos a partir de pares de pais. Os pais são escolhidos aleatoriamente do conjunto da elite, e criam os filhos por *crossover*. Caso o número de indivíduos da população seja ímpar, na criação do último par, um dos filhos é descartado.

Tendo a prole sido gerada, ela sofrerá mutação. Conforme mostrado na seção Mutação (página 23), cada indivíduo tem probabilidade *α* de ser selecionado para mutação, sendo *α* a taxa de mutação. Por exemplo: se a taxa de mutação for 50%, e a prole for composta por 16 indivíduos, então, em média 8 indivíduos sofrerão mutação.

A função de corte para uso no *crossover* e a função de mutação utilizadas em nosso trabalho serão detalhadas a seguir.

Função de corte

Conforme visto no capítulo Revisão dos conceitos abordados, seção 2.1.5 (página 22), a função de corte é responsável por determinar entre quais cromossomos ocorrerá o seccionamento do DNA dos pais para que haja a geração dos filhos (utilizamos apenas um ponto de corte neste trabalho).

A função escolhe aleatoriamente um ponto de corte entre dois cromossomos, variando da posição entre o primeiro e o segundo até a posição entre o penúltimo e o último.

Função de mutação

Conforme visto no capítulo Revisão dos conceitos abordados, seção 2.1.6 (página 23), a função de mutação é responsável por decidir qual cromossomo será modificado e como será feita esta modificação.

O cromossomo a ser modificado pela mutação é escolhido aleatoriamente (apenas um cromossomo é modificado na mutação).

Ele é modificado de acordo com as restrições impostas para cada execução, explicadas no capítulo 6.

6 Análise dos Resultados Obtidos

A avaliação de uma população, no contexto deste trabalho, é computacionalmente cara. Para realizar esta tarefa, é necessária a simulação de cada um dos indivíduos (circuitos), além do tratamento dos extensos arquivos de saída gerados como produto do simulador.

Para cada fonte a ser otimizada, serão feitas cinco execuções completas do algoritmo genético para três tamanhos de população: 10, 20 e 30 indivíduos. A população inicial é sempre criada com parâmetros aleatórios, com restrições baseadas nos critérios de projeto previamente estabelecidos (mais detalhes na página 42). A tabela 3 apresenta estes critérios, válidos para todas as fontes.

Tabela 3: Critérios de restrição da fonte Alpha

Critério	Restrição Imposta
Primeiro critério	*L* varia de 0,35µm a 5,0µm
Segundo critério	*W* varia de 1,0µm a 100µm
Terceiro critério	Rs varia de 5kΩs a 50kΩs
Quatro critério	X fixado em 2
Quinto critério	Y fixado em 1

6.1 Resultados de otimização da fonte Alpha

As otimizações foram executadas com populações de 10, 20 e 30 indivíduos. O fator tolerância de variação ζ foi fixado em *5%*. O limiar (*threshold*) *T* depende do tamanho da população e foi estabelecido como *33,3%* de uma geração. A taxa de mutação α foi escolhida como *50%*.

É útil ressaltar que apenas a prole da elite de uma geração pode sofrer mutação (mais detalhes na página 23). Apesar da taxa de mutação ser alta para os padrões de algoritmos genéticos, apenas um cromossomo de cada indivíduo que sofre mutação é modificado, equilibrando o impacto desta operação com as modificações do acasalamento.

Para justificar esta afirmação, suponha uma população de 30 indivíduos, cada um com DNA de 10 cromossomos, uma elite com 10 elementos, acasalamento por crossover com um ponto de corte e taxa de mutação de 50% com apenas um cromossomo alterado por mutação. Vamos comparar o número de cromossomos modificador pelo acasalamento e pela mutação de uma geração para outra. No acasalamento, são gerados 20 indivíduos e 5 cromossomos de cada um serão modificados, de forma que, ao todo, 100 cromossomos serão modificados nesta geração pelo acasalamento. Por outro lado, na mutação, são escolhidos 10 indivíduos e apenas 1 cromossomo de cada um será modificado, de forma que, ao todo, apenas 10 cromossomos serão modificados nesta geração pela mutação. Com isso, a mutação estará modificando em média 10

vezes menos cromossomos que a operação de acasalamento, razoável para este trabalho.

Para testar o algoritmo, especificou-se uma fonte com corrente de saída de *0,5µA*, com tolerância de variação de *±0,03µA*. Especificou-se também que a fonte deve operar com boa estabilidade com tensão de alimentação inferior a *2,1V*. Caso estas especificações não sejam atingidas em trinta gerações, a otimização será encerrada.

A tabela 4 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Alpha com população de 10 indivíduos.

Tabela 4: Indivíduos encontrados para a fonte Alpha com população de tamanho 10

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln (µm)	4,6	4,86	4,3	4,92	4,96
Wn (µm)	91,8	92,2	44,7	29,4	51,3
Lp (µm)	4,86	4,68	4,87	4,58	4,76
Wp (µm)	64,9	2,0	72,7	91,0	46,2
Rs (Ωs)	44424	43537	45924	46967	45954

A tabela 5 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Alpha com população de 20 indivíduos.

Tabela 5: Indivíduos encontrados para a fonte Alpha com população de tamanho 20

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln (µm)	4,8	4,96	4,96	4,77	4,84
Wn (µm)	38,8	83,5	90,4	61,7	38,0
Lp (µm)	4,62	4,72	4,48	4,93	4,27
Wp (µm)	88,9	6,8	50,7	66,9	88,9
Rs (Ωs)	47104	44906	44666	45006	47082

A tabela 6 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Alpha com população de 30 indivíduos.

Tabela 6: Indivíduos encontrados para a fonte Alpha com população de tamanho 30

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln (µm)	4,5	4,71	4,81	4,99	4,81
Wn (µm)	30,6	77,7	90,0	99,5	97,2
Lp (µm)	4,7	4,8	4,55	4,9	4,78
Wp (µm)	49,9	95,7	56,7	1,3	73,1
Rs (Ωs)	48500	45040	44554	42987	44476

A tabela 7 mostra os resultados das otimizações com os três tamanhos de população e cinco execuções completas para a fonte Alpha.

Tabela 7: Resultados da otimização da fonte Alpha

Tamanho da população	**Média do nº de gerações (cinco execuções)**	**Média do tempo gasto (s) (cinco execuções)**	**Média da pontuação (cinco execuções)**
10	23,6	102	22498
20	15,2	111	22597
30	7,6	95,6	22658

Obs.: Execuções em um computador com: processador Intel(R) Core(TM)2 Duo CPU E6750, 2.66GHz; 4GB de memória RAM; sistema operacional Gentoo GNU/Linux 64 bits; interpretador mzscheme 4.2.5. O ambiente continha apenas uma carga mínima necessária, como o ambiente gráfico X-Windows e processos do sistema.

Pelos dados apresentados, nota-se que as fontes otimizadas possuem valores de R_s, L_p e L_n altos. Nota-se uma tendência da diminuição do tempo total da otimização conforme a população fica mais numerosa. Este comportamento é interessante, pois apesar de aumentar o número de simulações por geração com o aumento do tamanho da população, o tempo final de execução foi menor por conta do menor número de gerações necessárias para atingir a solução. No caso desta topologia de fonte, a maior diversidade de indivíduos nas populações mais numerosas acelerou a otimização da fonte. Porém, devido ao baixo número de execuções, estes dados são pouco significativos do ponto de vista estatístico.

Nem todas as fontes atingiram a especificação traçada inicialmente. Para a população de 10 indivíduos, as fontes otimizadas 4 e 5 atingiram a especificação. Para a população de 20 indivíduos, as fontes otimizadas 1, 2, 4 e 5 atingiram a especificação. Para a população de 30 indivíduos, as fontes otimizadas 1, 2, 4 e 5 atingiram a especificação. As outras alcançaram o limite de 30 gerações fora da especificação, mas apresentaram uma boa curva também e chegaram bem próximas dos valores desejados.

As figuras 15, 16 e 17 apresentam os gráficos das curvas de corrente pela tensão de alimentação da fonte *alpha* obtidas com as otimizações feitas para os três tamanhos de

população.

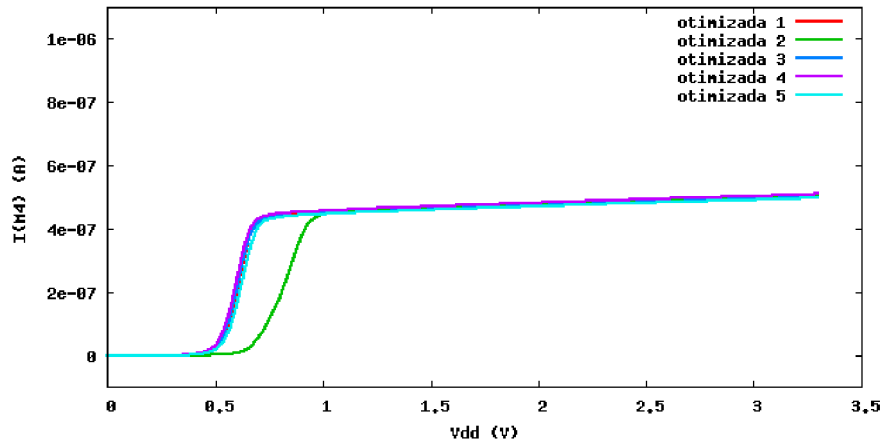

Figura 15: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Alpha otimizadas pelo algoritmo genético para população de 10 indivíduos

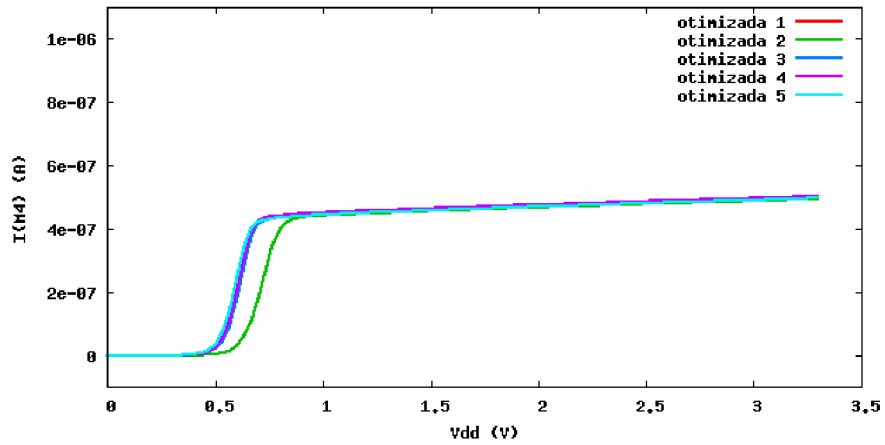

Figura 16: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Alpha otimizadas pelo algoritmo genético para população de 20 indivíduos

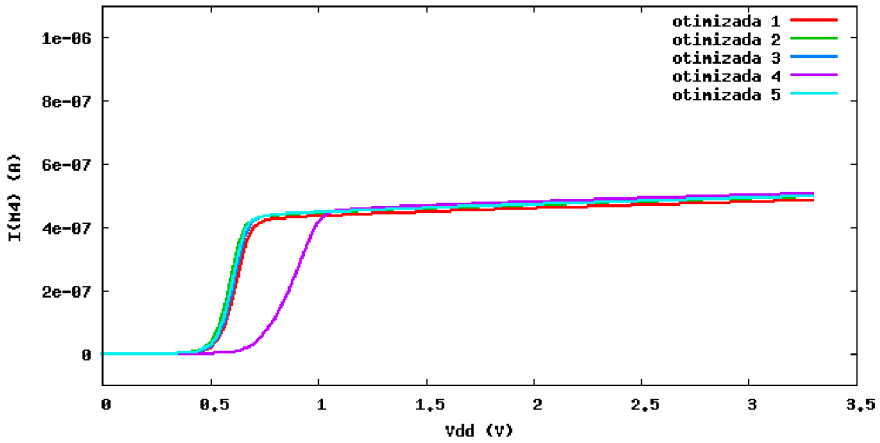

Figura 17: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Alpha otimizadas pelo algoritmo genético para população de 30 indivíduos

Dos gráficos, observa-se que todas as fontes otimizadas, em todos os casos, são utilizáveis em circuitos eletrônicos.

Esta topologia batizada de *alpha* possui limitações que não podem ser superadas apenas com o mero ajuste de parâmetros. As fontes analisadas nas seções seguintes possuem topologias mais robustas e são projetadas para oferecer uma corrente mais estável do que a vista nesta fonte.

6.2 Resultados de otimização da fonte Beta

As otimizações foram executadas com populações de 10, 20 e 30 indivíduos. O fator tolerância de variação ζ foi fixado em 2%. O limiar (*threshold*) T depende do tamanho da população e foi estabelecido como *33,3%* de uma geração. A taxa de mutação α foi escolhida como *50%*.

Para testar o algoritmo, especificou-se uma fonte com corrente de saída de $0{,}5\mu A$, com tolerância de variação de $\pm 0{,}03\mu A$. Por esta topologia ser mais robusta do que a *Alpha*, a tensão de alimentação mínima para o qual a fonte deve operar com boa estabilidade foi forçada para um valor mais baixo, especificada como $1{,}8V$. Caso estas especificações não sejam atingidas em trinta gerações, a otimização será encerrada.

A tabela 8 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Beta com população de 10 indivíduos.

Tabela 8: Indivíduos encontrados para a fonte Beta com população de tamanho 10

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln (µm)	1,09	1,46	2,71	4,8	3,86
Wn (µm)	61,7	82,6	43,5	42,6	49,5
Lp (µm)	4,69	4,8	4,9	4,96	4,89
Wp (µm)	94,0	82,8	99,0	99,7	76,5
Rs (Ωs)	39792	39502	41579	43103	43880

A tabela 9 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Beta com população de 20 indivíduos.

Tabela 9: Indivíduos encontrados para a fonte Beta com população de tamanho 20

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln (µm)	4,62	4,33	2,34	0,93	3,74
Wn (µm)	54,6	99,9	12,6	17,8	95,3
Lp (µm)	4,95	4,44	4,89	4,82	4,82
Wp (µm)	98,8	95,3	66,5	69,3	45,1
Rs (Ωs)	42493	40887	44802	42694	39389

A tabela 10 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Beta com população de 30 indivíduos.

Tabela 10: Indivíduos encontrados para a fonte Beta com população de tamanho 30

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln (µm)	1,75	3,74	3,15	0,72	1,49
Wn (µm)	63,7	61,4	76,1	36,7	14,8
Lp (µm)	4,42	4,62	4,73	4,75	4,92
Wp (µm)	86,7	92,0	58,3	68,0	94,8
Rs (Ωs)	40840	41620	41047	39782	42533

A tabela 11 mostra os resultados das otimizações com os três tamanhos de população e cinco execuções completas para a fonte Beta.

Tabela 11: Resultados da otimização da fonte Beta

Tamanho da população	Média do nº de gerações (cinco execuções)	Média do tempo gasto (s) (cinco execuções)	Média da pontuação (cinco execuções)
10	14,4	77	14838
20	14,4	129	14234
30	8,2	112	14734

Obs.: Execuções em um computador com: processador Intel(R) Core(TM)2 Duo CPU E6750, 2.66GHz; 4GB de memória RAM; sistema operacional Gentoo GNU/Linux 64 bits; interpretador mzscheme 4.2.5. O ambiente continha apenas uma carga mínima necessária, como o ambiente gráfico X-Windows e processos do sistema.

Pelos dados apresentados, nota-se que as fontes otimizadas possuem valores de R_s, L_p e W_p altos. Diferentemente da topologia *Alpha*, nota-se um equilíbrio no tempo de execução das otimizações. Apesar do número de parâmetros ser igual ao da fonte *Alpha*, a topologia *Beta* é mais robusta e, por conta disso, existem mais soluções satisfatórias no domínio de busca. Como consequência, as execuções convergiram mais rápido do que com a fonte *Alpha*. Devido ao baixo número de execuções, estes dados são pouco significativos do ponto de vista estatístico.

Nem todas as fontes atingiram a especificação traçada inicialmente. Para a população de 10 indivíduos, todas as fontes atingiram a especificação. Para a população de 20 indivíduos, apenas as fontes otimizadas 1, 2, 4 e 5 atingiram a especificação. Para a população de 30 indivíduos, apenas as fontes otimizadas 1, 2, 3 e 4 atingiram a especificação. As outras alcançaram o limite de 30 gerações fora da especificação, mas apresentaram uma boa curva também e chegaram bem próximas dos valores desejados. Nota-se que mais fontes otimizadas atingiram a especificação com esta topologia por conta da maior robustez desta fonte.

As figuras 20, 19 e 18 apresentam os gráficos das curvas de corrente pela tensão de alimentação da fonte *Beta* obtidas com as otimizações feitas para os três tamanhos de população.

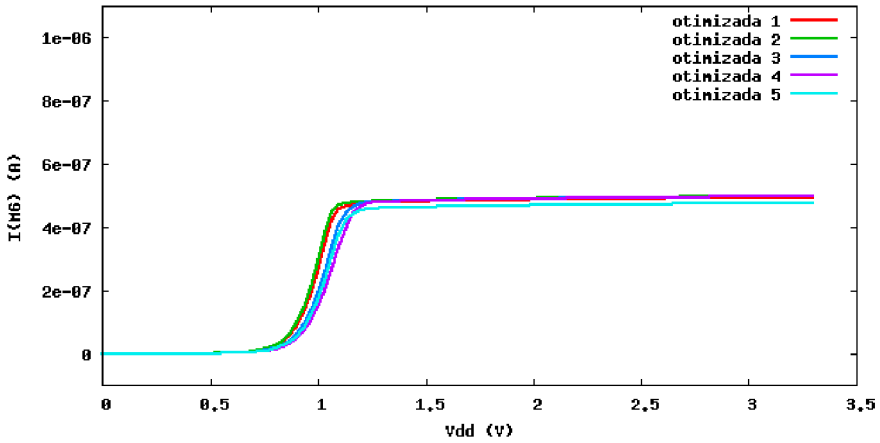

Figura 18: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Beta otimizadas pelo algoritmo genético para população de 10 indivíduos

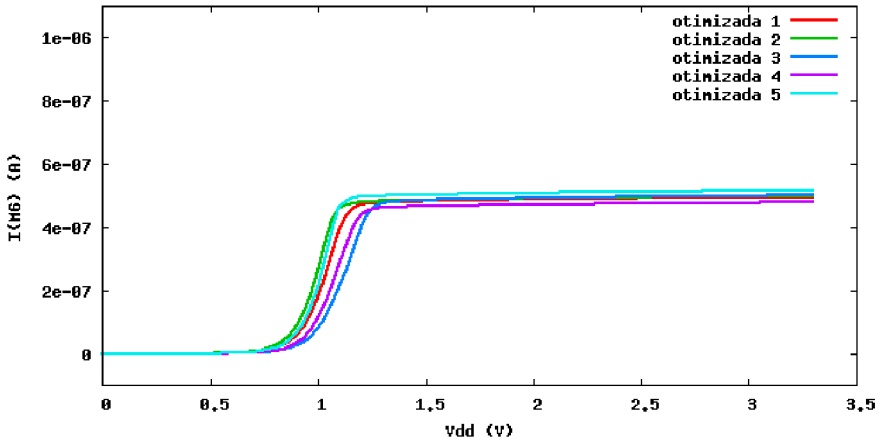

Figura 19: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Beta otimizadas pelo algoritmo genético para população de 20 indivíduos

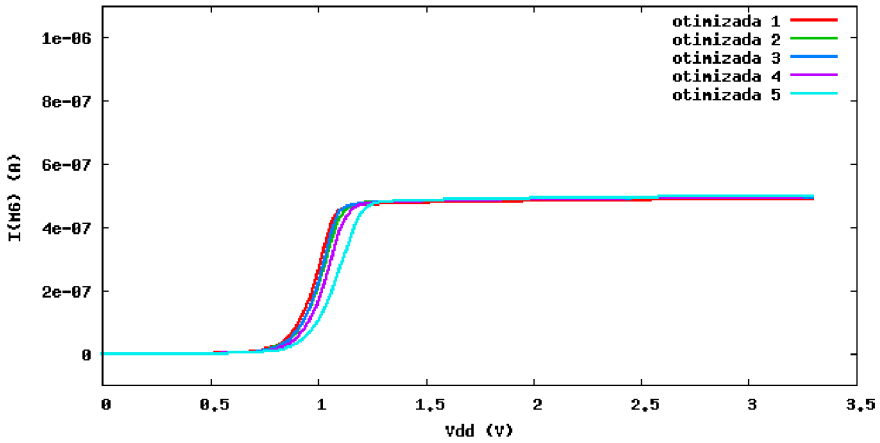

Figura 20: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Beta otimizadas pelo algoritmo genético para população de 30 indivíduos

Dos gráficos, observa-se que as fontes otimizadas desta topologia possuem maior estabilidade do que a fonte Alpha, ao custo de tensões de alimentação um pouco maiores para o início da região de operação. Em todos os casos, as fontes são utilizáveis em circuitos eletrônicos. Aquelas que não cumpriram a especificação obtiveram comportamento próximo ao das que cumpriram.

As topologias seguintes visam obter correntes tão estáveis quanto a Beta, mas operam com tensões de alimentação menores.

6.3 Resultados de otimização da fonte Gamma

As otimizações foram executadas com populações de 10, 20 e 30 indivíduos. O fator tolerância de variação ζ foi fixado em 2%. O limiar (*threshold*) T depende do tamanho da população e foi estabelecido como *33,3%* de uma geração. A taxa de mutação α foi escolhida como *50%*.

Para testar o algoritmo, especificou-se uma fonte com corrente de saída de $0{,}5\mu A$, com tolerância de variação de $\pm 0{,}03\mu A$. *A* tensão de alimentação para o qual a fonte deve operar com boa estabilidade foi especificada como $1{,}8V$, assim como a fonte *Beta*. Caso estas especificações não sejam atingidas em trinta gerações, a otimização será encerrada.

A tabela 12 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Gamma com população de 10 indivíduos.

Tabela 12: Indivíduos encontrados para a fonte Gamma com população de tamanho 10

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln-1-2 (µm)	0,54	1,13	0,74	0,58	3,65
Ln-3-4 (µm)	2,42	2,4	4,88	1,49	2,85
Wn-1-3 (µm)	42,6	73,2	31,2	72,6	81,7
Wn-2-4 (µm)	65,2	78,9	47,9	76,1	96,2
Lp-5-6-8 (µm)	4,77	4,68	4,82	3,3	3,75
Wp-5-6 (µm)	9,7	10,9	43,5	6,3	6,3
Wp-8 (µm)	84,3	71,5	82,9	93,7	54,8
Ln-7 (µm)	4,55	1,19	4,14	1,76	4,11
Wn-7 (µm)	8,5	5,4	2,6	14,5	63,8
Rs (Ωs)	446527	43047	48567	42806	49698

A tabela 13 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Gamma com população de 20 indivíduos.

Tabela 13: Indivíduos encontrados para a fonte Gamma com população de tamanho 20

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln-1-2 (µm)	2,9	0,89	1,46	0,36	0,65
Ln-3-4 (µm)	3,41	3,44	2,82	2,28	4,05
Wn-1-3 (µm)	77,6	73,3	39,8	53,4	66,8
Wn-2-4 (µm)	89,0	99,1	54,5	56,9	77,3
Lp-5-6-8 (µm)	4,28	4,74	4,85	4,79	4,74
Wp-5-6 (µm)	18,7	23,3	10,0	9,4	16,2
Wp-8 (µm)	42,3	93,3	96,1	57,9	89,5
Ln-7 (µm)	3,55	4,04	3,25	3,64	4,94
Wn-7 (µm)	7,1	5,4	9,6	16,2	18,2
Rs (Ωs)	47337	47460	47160	41998	47553

A tabela 14 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Gamma com população de 30 indivíduos.

Tabela 14: Indivíduos encontrados para a fonte Gamma com população de tamanho 30

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln-1-2 (µm)	1,06	0,89	0,72	0,56	1,08
Ln-3-4 (µm)	1,84	2,76	2,41	4,34	2,35
Wn-1-3 (µm)	51,7	41,3	32,5	64,4	58,8
Wn-2-4 (µm)	59,3	63,9	53,6	92,9	95,1
Lp-5-6-8 (µm)	4,43	4,79	4,94	4,95	4,9
Wp-5-6 (µm)	11,7	29,6	16,6	15,5	43,2
Wp-8 (µm)	70,8	96,8	96,6	71,5	94,7
Ln-7 (µm)	4,66	4,1	4,66	4,63	3,08
Wn-7 (µm)	11,6	3,6	16,7	5,6	3,9
Rs (Ωs)	46454	48371	49523	49706	49982

A tabela 15 mostra os resultados das otimizações com os três tamanhos de população e cinco execuções completas para a fonte Gamma.

Tabela 15: Resultados da otimização da fonte Gamma

Tamanho da população	**Média do nº de gerações (cinco execuções)**	**Média do tempo gasto (s) (cinco execuções)**	**Média da pontuação (cinco execuções)**
10	15	72	18918
20	14,8	122	20165
30	25,8	330	21906

Obs.: Execuções em um computador com: processador Intel(R) Core(TM)2 Duo CPU E6750, 2.66GHz; 4GB de memória RAM; sistema operacional Gentoo GNU/Linux 64 bits; interpretador mzscheme 4.2.5. O ambiente continha apenas uma carga mínima necessária, como o ambiente gráfico X-Windows e processos do sistema.

Pelos dados apresentados, nota-se que as fontes otimizadas possuem valores de R_s altos. Diferentemente da topologia Beta, nota-se que o aumento do tamanho da população acarretou uma maior média do número de gerações necessárias para obter a solução. Uma maior diversidade de indivíduos por geração (tamanho da população maior) não acelerou o processo de otimização. Isto pode ser consequência da inadequação da função de aptidão, fazendo com que fontes que não atinjam a especificação obtenham uma alta de pontuação (conforme evidenciamos pela coluna da média de pontos da tabela 15). Mais estudos são necessários para obter conclusões mais precisas. Novamente, devido ao baixo número de execuções, estes dados são pouco significativos do ponto de vista estatístico.

Nem todas as fontes atingiram a especificação traçada inicialmente. Para a população de 10 indivíduos, apenas as fontes otimizadas 2, 4 e 5 atingiram a especificação. Para a população de 20 indivíduos, apenas as fontes otimizadas 1, 4 e 5 atingiram a especificação. Para a população

de 30 indivíduos, apenas a fonte otimizada 1 atingiu a especificação. As outras alcançaram o limite de 30 gerações fora da especificação e não se aproximaram da especificação. Nota-se que o número de fontes que atingiram a especificação foi inferior ao das topologias *Alpha* e *Beta*. Possivelmente, isto ocorreu por causa do aumento do domínio de busca (maior número de parâmetros), que pode ter tornado o limite de 30 gerações muito precoce para estas execuções.

As figuras 23, 22 e 21 apresentam os gráficos das curvas de corrente pela tensão de alimentação da fonte *Gamma* obtidas com as otimizações feitas para os três tamanhos de população.

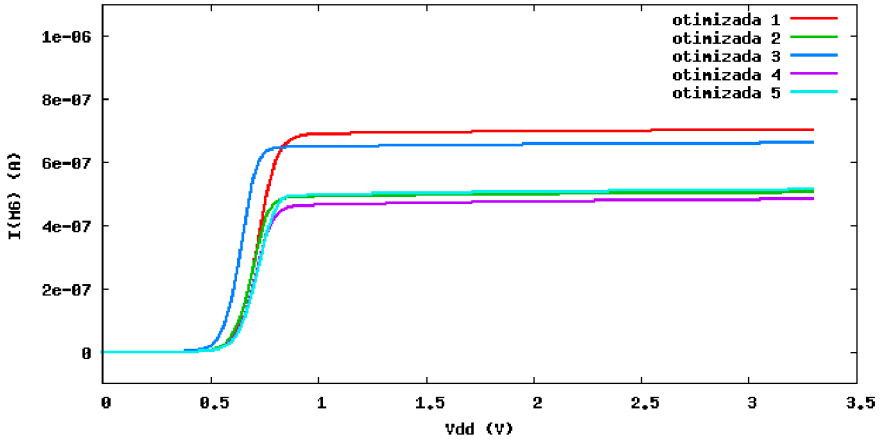

Figura 21: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Gamma otimizadas pelo algoritmo genético para população de 10 indivíduos

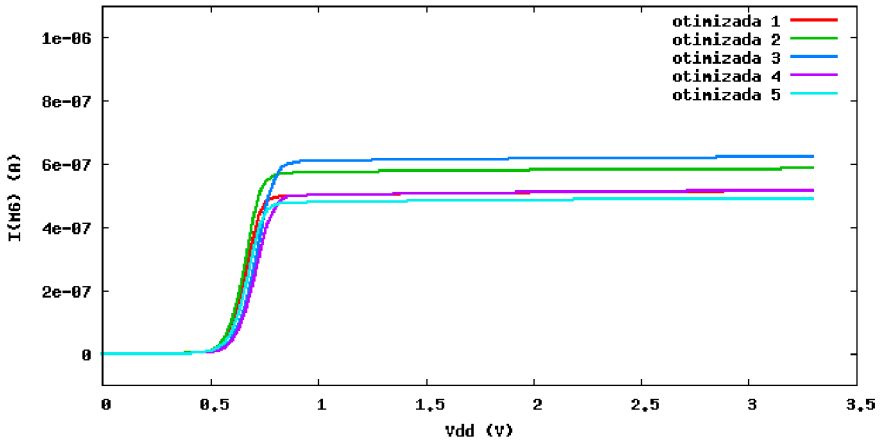

Figura 22: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Gamma otimizadas pelo algoritmo genético para população de 20 indivíduos

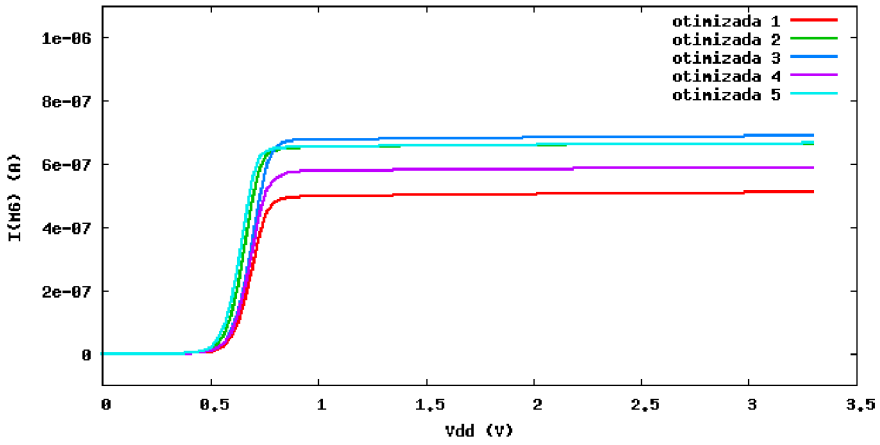

Figura 23: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Gamma otimizadas pelo algoritmo genético para população de 30 indivíduos

Dos gráficos, observa-se que as fontes otimizadas desta topologia possuem boa estabilidade, operam com tensões de alimentação menores do que a fonte *Beta*, mas um número considerável de fontes não cumpriu a especificação inicial de corrente.

A última topologia foi projetada para obter desempenho melhor do que a *Gamma* em todos as especificações definidas no início do capítulo.

6.4 Resultados de otimização da fonte Delta

As otimizações foram executadas com populações de 10, 20 e 30 indivíduos. O fator tolerância de variação ζ foi fixado em 2%. O limiar (*threshold*) *T* depende do tamanho da população e foi estabelecido como *33,3%* de uma geração. A taxa de mutação α foi escolhida como *50%*.

Para testar o algoritmo, especificou-se uma fonte com corrente de saída de *0,5µA*, com tolerância de variação de *±0,03µA*. *A* tensão de alimentação para o qual a fonte deve operar com boa estabilidade foi especificada como 1,8*V*, assim como as fontes *Beta* e *Gamma*. Caso estas especificações não sejam atingidas em trinta gerações, a otimização será encerrada.

A tabela 16 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Delta com população de 10 indivíduos.

Tabela 16: Indivíduos encontrados para a fonte Delta com população de tamanho 10

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln-1-2-7 (µm)	0,7	4,7	4,0	11,8	1,4
Ln-4 (µm)	1,0	0,2	1,2	0,5	2,1
Wn-1 (µm)	1,4	62,5	0,9	10,9	34,1
Wn-2-4 (µm)	37,1	95,5	11,7	90,0	56,6
Wn-7 (µm)	28,7	58,3	46,9	8,9	91,2
Lp-3 (µm)	2,9	1,4	4,9	4,2	1,0
Lp-5-6-10 (µm)	2,2	3,4	2,7	1,4	1,1
Wp-5-3 (µm)	84,2	67,5	25,6	65,3	11,1
Wp-6 (µm)	27,9	28,0	7,7	59,4	8,5
Wp-10 (µm)	47,1	77,7	21,9	44,3	95,0
Ln-8 (µm)	3,7	3,3	1,2	1,5	5,0
Wp-8 (µm)	0,4	98,0	36,0	52,6	21,9
Ln-9 (µm)	0,8	0,6	0,5	2,6	4,0
Wn-9 (µm)	68,4	22,5	37,9	14,9	39,4
Rs (Ωs)	10227	47158	14746	29867	37452

A tabela 17 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Delta com população de 20 indivíduos.

Tabela 17: Indivíduos encontrados para a fonte Delta com população de tamanho 20

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln-1-2-7 (µm)	1,1	3,6	1,7	0,2	3,5
Ln-4 (µm)	2,4	4,1	0,2	1,5	4,9
Wn-1 (µm)	7,7	26,6	20,0	12,6	9,9
Wn-2-4 (µm)	86,7	32,1	84,8	98,2	64,4
Wn-7 (µm)	11,7	53,0	33,9	59,7	60,8
Lp-3 (µm)	0,3	4,2	1,4	2,6	3,5
Lp-5-6-10 (µm)	2,3	1,5	2,0	4,6	1,8
Wp-5-3 (µm)	19,3	99,5	44,0	30,7	71,5
Wp-6 (µm)	81,2	34,2	32,8	86,1	44,1
Wp-10 (µm)	63,5	58,5	89,7	33,8	53,2
Ln-8 (µm)	3,2	0,8	1,0	2,8	0,8
Wp-8 (µm)	5,9	90,6	93,8	24,9	88,9
Ln-9 (µm)	3,4	3,4	3,2	4,5	5,0
Wn-9 (µm)	2,9	4,5	42,8	5,0	29,4
Rs (Ωs)	45502	29630	40797	46203	48801

A tabela 18 mostra os indivíduos encontrados através da otimização por algoritmo genético para a fonte Delta com população de 30 indivíduos.

Tabela 18: Indivíduos encontrados para a fonte Delta com população de tamanho 30

	Otimizada 1	**Otimizada 2**	**Otimizada 3**	**Otimizada 4**	**Otimizada 5**
Ln-1-2-7 (µm)	2,4	2,9	1,2	1,3	2,7
Ln-4 (µm)	2,2	2,6	1,9	5,0	0,2
Wn-1 (µm)	49,3	39,5	50,5	19,5	49,7
Wn-2-4 (µm)	81,2	71,7	98,9	97,6	54,1
Wn-7 (µm)	88,1	63,2	94,7	61,1	52,9
Lp-3 (µm)	1,1	0,9	1,3	3,5	0,9
Lp-5-6-10 (µm)	3,8	4,1	1,7	5,0	1,3
Wp-5-3 (µm)	91,7	66,8	73,4	38,1	46,1
Wp-6 (µm)	55,8	46,8	51,7	14,6	18,8
Wp-10 (µm)	91,2	24,5	83,1	92,0	7,8
Ln-8 (µm)	2,3	3,8	4,8	5,0	4,5
Wp-8 (µm)	30,1	37,8	60,7	44,8	20,8
Ln-9 (µm)	1,0	2,7	4,8	1,6	3,8
Wn-9 (µm)	7,4	1,7	13,3	21,7	93,1
Rs (Ωs)	48241	47763	49339	49406	49514

A tabela 19 mostra os resultados das otimizações com os três tamanhos de população e cinco execuções completas para a fonte Delta.

Tabela 19: Resultados da otimização da fonte Delta

Tamanho da população	**Média do nº de gerações (cinco execuções)**	**Média do tempo gasto (s) (cinco execuções)**	**Média da pontuação (cinco execuções)**
10	27,4	145	1480
20	23,8	197	8939
30	23	314	22803

Obs.: Execuções em um computador com: processador Intel(R) Core(TM)2 Duo CPU E6750, 2.66GHz; 4GB de memória RAM; sistema operacional Gentoo GNU/Linux 64 bits; interpretador mzscheme 4.2.5. O ambiente continha apenas uma carga mínima necessária, como o ambiente gráfico X-Windows e processos do sistema.

Pelos dados apresentados, nota-se que atingir a solução satisfatória tornou-se uma tarefa mais mais custosa com o aumento do número de parâmetros (a fonte *Delta* possui DNA com 15 cromossomos). O número da geração máxima das execuções (no caso, 30) mostrou-se baixo, visto que as execuções com tamanhos de população menores tiveram dificuldade para convergir (média de gerações mais alta). Novamente, devido ao baixo número de execuções, estes dados são pouco significativos do ponto de vista estatístico.

Nem todas as fontes atingiram a especificação traçada inicialmente. Para a população de 10 indivíduos, apenas a fonte otimizada 5 atingiu a especificação. Para a população de 20 indivíduos, apenas as fontes otimizadas 1 e 4 atingiram a especificação. Para a população de 30 indivíduos, apenas as fontes otimizadas 1, 2 e 3 atingiram a especificação. As outras alcançaram o limite de 30 gerações fora da especificação e não se aproximaram da especificação. Nota-se que o número de fontes que atingiram a especificação foi inferior ao das topologias *Alpha*, *Beta* e *Gamma*. Possivelmente isto ocorreu por causa do aumento do domínio de busca (maior número de parâmetros), que pode ter tornado o limite de 30 gerações muito precoce para estas execuções.

As figuras 26, 25 e 24 apresentam os gráficos das curvas de corrente pela tensão de alimentação da fonte *Delta* obtidas com as otimizações feitas para os três tamanhos de população.

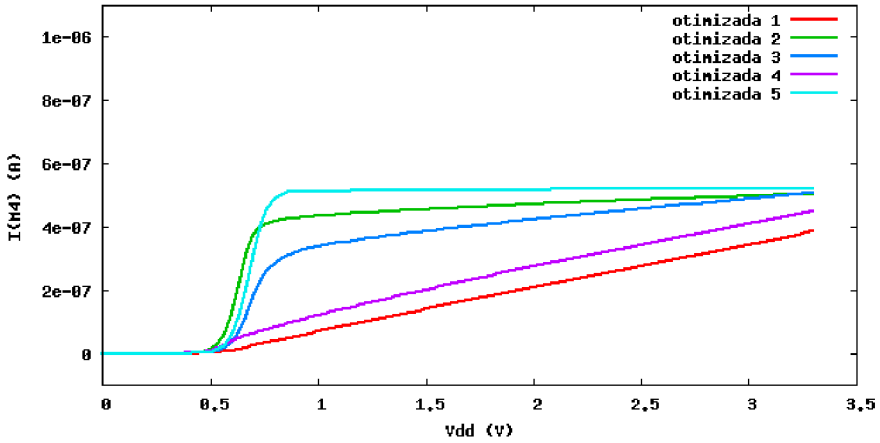

Figura 24: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Delta otimizadas pelo algoritmo genético para população de 10 indivíduos

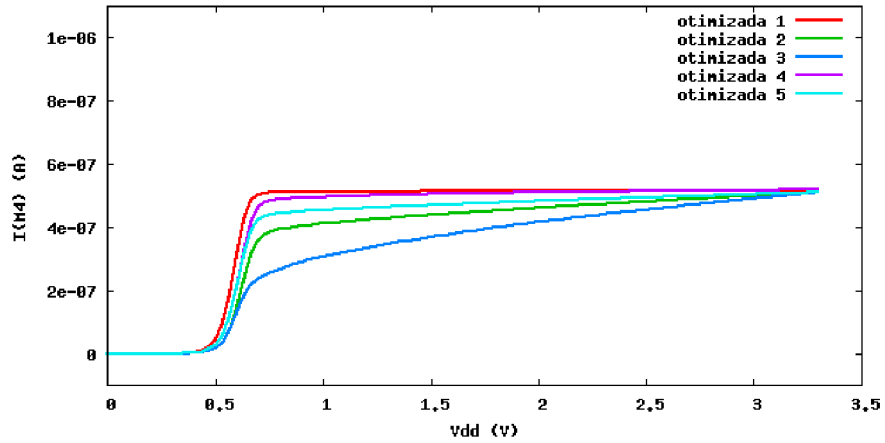

Figura 25: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Delta otimizadas pelo algoritmo genético para população de 20 indivíduos

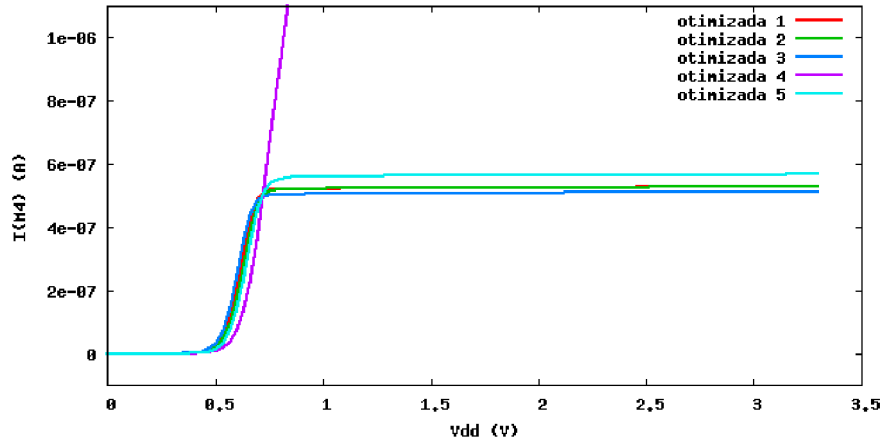

Figura 26: Curva da corrente fornecida pela tensão de alimentação das fontes com topologia Delta otimizadas pelo algoritmo genético para população de 30 indivíduos

Dos gráficos, observa-se que as fontes otimizadas desta topologia possuem boa estabilidade, operam com tensões de alimentação menores do que a fonte *Beta*, mas um número considerável de fontes não cumpriu a especificação inicial de corrente. No caso das execuções com 30 indivíduos, houve uma fonte que ficou muito distante da especificação. Por a população inicial ser

construída de maneira aleatória, é possível que esta fonte tenha sido bastante desfavorecida na primeira geração. As fontes que respeitaram a especificação mostraram-se parecidas com as fontes obtidas com a topologia *Gamma*.

Os gráficos enfatizam a conclusão de que 30 gerações é um número baixo para este DNA de 15 cromossomos convergir. Para fundamentar mais esta hipótese, fizemos uma execução extra da topologia *Delta*, com 30 indivíduos, ininterruptamente até atingir 60 gerações.

O indivíduo gerado por esta execução é apresentado na tabela 20.

Tabela 20: Otimização da fonte de corrente Delta, com 60 gerações e população de tamanho 30

	Otimizada após 60 gerações
Ln-1-2-7 (µm)	3,3
Ln-4 (µm)	2,4
Wn-1 (µm)	45,5
Wn-2-4 (µm)	91,9
Wn-7 (µm)	68,6
Lp-3 (µm)	2,4
Lp-5-6-10 (µm)	1,2
Wp-5-3 (µm)	55,4
Wp-6 (µm)	67,4
Wp-10 (µm)	99,7
Ln-8 (µm)	0,8
Wp-8 (µm)	6,7
Ln-9 (µm)	4,2
Wn-9 (µm)	6,1
Rs (Ωs)	23631

A figura 27 mostra o gráfico desta execução.

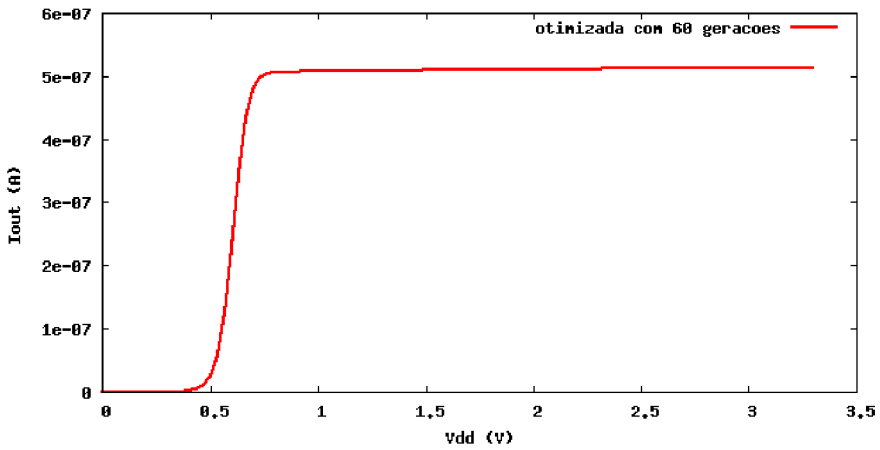

Figura 27: Execução da fonte Delta com 60 gerações e 30 indivíduos

Nota-se que a fonte obtida é de boa qualidade, baseado nos critérios descritos no capítulo Otimização por Algoritmo Genético (página 41). Além de ter a região de operação iniciando-se com valores baixos de V_{DD}, a corrente fornecida é bastante estável e próxima da especificada (*0,5μA*). Portanto, a hipótese exposta anteriormente de que 30 gerações não são um número suficiente para obter boas otimizações com DNAs de números de cromossomos elevados é razoável.

7 Conclusão

Neste trabalho foi montada uma configuração de algoritmo genético para otimizar parâmetros de circuitos eletrônicos. Foram definidos os cromossomos dos indivíduos (possível solução), uma função de aptidão, uma função de seleção e uma função de mutação para buscar boas fontes de corrente (operar com baixa tensão de alimentação e fornece corrente estável). Feito isso, foi implementado todo o software necessário para integrar o simulador com a otimização. Testes com o programa e com a fonte mostraram que é possível encontrar uma boa configuração destes parâmetros através de algoritmos genéticos, utilizando conhecimento mínimo do domínio do problema.

Conforme observou-se no capítulo 6 (Análise dos Resultados Obtidos), o objetivo proposto foi atingido: otimizar circuitos integrados (em particular, fontes de corrente) através de algoritmos genéticos. Os resultados foram motivadores, com a aplicação de um algoritmo genético convencional conseguindo resultados práticos em uma classe de problemas complexos.

Este trabalho abre portas para trabalhos onde técnicas mais avançadas de algoritmos genéticos poderão ser aplicadas para resolver problemas mais desafiadores do universo de projetos de circuitos eletrônicos. Ferramentas deste tipo poderão, em um futuro próximo, reduzir o custo e o tempo de implementação de projetos de *hardware* em geral.

A seguir são apresentadas possíveis continuações do projeto:

- estudar se as otimizações funcionam com um número maior de restrições, como gerar fontes com as menores dimensões possíveis de transistores, forçar os transistores em fraca ou forte inversão;
- integrar outros simuladores comerciais mais confiáveis;
- aplicar técnicas mais sofisticadas de algoritmos genéticos;
- a implementação de uma ferramenta gráfica, de interface amigável, para que projetistas experientes possam utilizar esta ferramenta, sem a necessidade de conhecimentos aprofundados do ambiente de programação utilizado;
- testar outros circuitos, como amplificadores operacionais e fontes de tensão bandgap, que exigem diferentes tipos de simulação (tais como transiente, AC, com a temperatura, etc);
- outros tipos de simulação, como transiente e AC, também podem obter ser estudados com a técnica desenvolvida.

8 Apêndice A – Código Fonte

8.1 spice-integration/elements.ss

```
;; COGA - Spice utility procedures
#lang scheme/base

(require scheme/mpair)
(require srfi/1)
(require srfi/13)

(provide (all-defined-out))

;; Generic elements procedures
(define (element->string element)
  (let ([type (car element)]
    [name (cadr element)]
    [rest (cddr element)])
    (string-append
     (param->string type)
     (param->string name) " "
     (string-join (map param->string rest)))))

(define (model-command->string model)
  (let ([name (cadr model)]
    [device (caddr model)]
    [params (cdddr model)])
    (string-append
     ".MODEL " (param->string name) " "
     (param->string device) " "
     (string-join (map param->string params)))))

(define (content->string contents)
  (fold (lambda (x seed)
      (string-append (any->string x) " " seed)) "" contents))

(define (param->string param)
  (let ([name (mcar param)]
    [content (mcdr param)])
    (cond [(and (not (list? param))
        (not (mpair? param)))
       (error "error: param must be a pair or a list")]
      [(type? param) (type->string param)]
      [(name? param) (name->string param)]
      [(nodes? param) (nodes->string param)]
      [(value? param) (value->string param)]
      [(model? param) (model->string param)]
      [(device? param) (device->string param)]
      [else
       (string-append (any->string name) "="
            (if (list? content)
              (content->string content)
              (any->string content)))])))

(define (params->string params)
  (fold (lambda (x seed)
      (string-append (param->string x) " " seed)) "" params))

(define (type? param)
  (let ([type (mcar param)]
    [content (mcdr param)])
    (cond [(not (mpair? param)) #f]
      [(not (eq? type 'TYPE)) #f]
      [else #t])))

(define (name? param)
  (let ([name (mcar param)]
    [content (mcdr param)])
    (cond [(not (mpair? param)) #f]
      [(not (eq? name 'NAME)) #f]
      [else #t])))

(define (nodes? param)
```

```
  (let ([nodes (mcar param)]
    [content (mcdr param)])
    (cond [(not (list? content)) #f]
      [(not (eq? nodes 'NODES)) #f]
      [else #t])))

(define (value? param)
  (let ([name (mcar param)]
    [content (mcdr param)])
    (cond [(not (mpair? param)) #f]
      [(not (eq? name 'VALUE)) #f]
      [else #t])))

(define (model? param)
  (let ([name (mcar param)]
    [content (mcdr param)])
    (cond [(not (mpair? param)) #f]
      [(not (eq? name 'MODEL)) #f]
      [else #t])))

(define (device? param)
  (let ([name (mcar param)]
    [content (mcdr param)])
    (cond [(not (mpair? param)) #f]
      [(not (eq? name 'DEVICE)) #f]
      [else #t])))

(define (type->string param)
  (let ([content (mcdr param)])
    (any->string content)))

(define (name->string param)
  (let ([content (mcdr param)])
    (any->string content)))

(define (nodes->string param)
  (let ([content (mcdr param)])
    (string-join
     (map any->string content))))

(define (value->string param)
  (let ([content (mcdr param)])
    (any->string content)))

(define (model->string param)
  (let ([content (mcdr param)])
    (any->string content)))

(define (device->string param)
  (let ([content (mcdr param)])
    (any->string content)))

(define (any->string any)
  (cond [(number? any) (number->string any)]
    [(symbol? any) (symbol->string any)]
    [(char? any) (make-string 1 any)]
    [(string? any) any]
    [else (error any)]))

;; Element constructors
(define (make-param name value)
  (mcons name value))

(define (make-voltage-source name . nodes-and-value)
  (let ([nodes (drop-right nodes-and-value 1)]
    [value (car (take-right nodes-and-value 1))])
    (cond [(< (length nodes-and-value) 3) (error "error: voltage source must have a name, two nodes and a value")]
      [(not (string? name)) (error "error: voltage-source's name must be a string")]
      [(list? value) (error "error: value must be an atom")]
      [else (list (make-param 'TYPE 'V)
            (make-param 'NAME name)
            (make-param 'NODES nodes)
            (make-param 'VALUE value))])))

(define (make-current-source name . nodes-and-value)
  (let ([nodes (drop-right nodes-and-value 1)]
```

```
    [value (car (take-right nodes-and-value 1))])
    (cond [(< (length nodes-and-value) 3) (error "error: current source must have a name, two nodes and a value")]
     [(not (string? name)) (error "error: current-source's name must be a string")]
     [(list? value) (error "error: value must be an atom")]
     [else (list (make-param 'TYPE 'I)
                 (make-param 'NAME name)
                 (make-param 'NODES nodes)
                 (make-param 'VALUE value))])))

(define (make-resistor name . nodes-and-value)
  (let ([nodes (drop-right nodes-and-value 1)]
    [value (car (take-right nodes-and-value 1))])
    (cond [(< (length nodes-and-value) 3) (error "error: resistor must have a name, two nodes and a value")]
     [(not (string? name)) (error "error: resistor's name must be a string")]
     [(list? value) (error "error: value must be an atom")]
     [else (list (make-param 'TYPE 'R)
                 (make-param 'NAME name)
                 (make-param 'NODES nodes)
                 (make-param 'VALUE value))])))

(define (make-capacitor name . nodes-and-value)
  (let ([nodes (drop-right nodes-and-value 1)]
    [value (car (take-right nodes-and-value 1))])
    (cond [(< (length nodes-and-value) 3) (error "error: capacitor must have a name, two nodes and a value")]
     [(not (string? name)) (error "error: capacitor's name must be a string")]
     [(list? value) (error "error: value must be an atom")]
     [else (list (make-param 'TYPE 'C)
                 (make-param 'NAME name)
                 (make-param 'NODES nodes)
                 (make-param 'VALUE value))])))

(define (make-inductor name . nodes-and-value)
  (let ([nodes (drop-right nodes-and-value 1)]
    [value (car (take-right nodes-and-value 1))])
    (cond [(< (length nodes-and-value) 3) (error "error: inductor must have a name, two nodes and a value")]
     [(not (string? name)) (error "error: inductor's name must be a string")]
     [(list? value) (error "error: value must be an atom")]
     [else (list (make-param 'TYPE 'I)
                 (make-param 'NAME name)
                 (make-param 'NODES nodes)
                 (make-param 'VALUE value))])))

(define (make-nmos name . nodes-and-model-and-params)
  (let* ([lst nodes-and-model-and-params]
    [size (length lst)]
    [pos-first-param (list-index (lambda (x) (mpair? x)) lst)]
    [nodes (if (eq? pos-first-param #f)
               (drop-right lst 1)
               (drop-right (take lst pos-first-param) 1))]
    [model (if (eq? pos-first-param #f)
               (car (take-right lst 1))
               (car (take-right (take lst pos-first-param) 1)))]
    [params (if (eq? pos-first-param #f)
                '()
                (drop lst pos-first-param))])
   (cond [(< (length nodes-and-model-and-params) 5) (error "error: NMOS must have a name, four nodes and zero or more
parameters")]
     [(not (string? name)) (error "error: nmos's name must be a string")]
     [(not (symbol? model)) (error "error: nmos's model must be a symbol")]
     [else (append (list (make-param 'TYPE 'M)
                         (make-param 'NAME name)
                         (make-param 'NODES nodes)
                         (make-param 'MODEL model))
                   params)])))

(define (make-pmos name . nodes-and-model-and-params)
  (let* ([lst nodes-and-model-and-params]
    [size (length lst)]
    [pos-first-param (list-index (lambda (x) (mpair? x)) lst)]
    [nodes (if (eq? pos-first-param #f)
               (drop-right lst 1)
               (drop-right (take lst pos-first-param) 1))]
    [model (if (eq? pos-first-param #f)
               (car (take-right lst 1))
               (car (take-right (take lst pos-first-param) 1)))]
    [params (if (eq? pos-first-param #f)
```

```
          '()
          (drop lst pos-first-param))])
   (cond [(< (length nodes-and-model-and-params) 5) (error "error: PMOS must have a name, four nodes and zero or more
parameters")]
     [(not (string? name)) (error "error: pmos's name must be a string")]
     [(not (symbol? model)) (error "error: pmos's model must be a symbol")]
     [else (append (list (make-param 'TYPE 'M)
                         (make-param 'NAME name)
                         (make-param 'NODES nodes)
                         (make-param 'MODEL model))
                   params)])))

(define (make-model-command name device . params)
  (cond [(not (string? name)) (error "error: model's name must be a string")]
    [(not (symbol? device)) (error "error: model's deice must be a symbol")]
    [else (append (list (make-param 'TYPE 'COMMAND)
                        (make-param 'NAME name)
                        (make-param 'DEVICE device))
                  params)]))

;; Element accessors
(define (get-param name element)
  (if (not (symbol? name))
      (error "error: a parameter's index must be a symbol")
      (let ([index (list-index (lambda (x) (eq? name (mcar x)))
                               element)])
    (if (not (number? index))
        void
        (list-ref element
                  index)))))

(define (set-param! name newvalue element)
  (if (not (symbol? name))
      (error "error: a parameter's index must be a symbol")
      (let ([index (list-index (lambda (x) (eq? name (mcar x)))
                               element)])
    (if (not (number? index))
        void
        (set-mcdr! (list-ref element
                             index) newvalue)))))

;; Analysis related
(define (make-circuit . elements)
  elements)

(define (circuit->string circuit)
  (fold (lambda (x seed) (string-append (element->string x) "\n" seed)) "" circuit))

(define (command->string command)
  (let ([name (mcar command)]
    [model (mcar (mcdr command))]
    [type (mcar (mcdr (mcdr command)))]
    [rest (mcdr (mcdr (mcdr command)))])
    (cond [(not (string? name)) (error "error: command name must be a string")]
      [(not (symbol? model)) (error "error: command model must be a symbol")]
      [(not (symbol? type)) (error "error: command type must be a symbol")]
      [else
       (string-append
        "." (param->string name) " "
        (param->string model) " "
        (param->string type) " "
        (string-join (map param->string rest)))])))
```

8.2 spice-integration/analysis.ss

```
#lang scheme/base

(require scheme/list)
(require (file "../spice-integration/elements.ss"))

(provide (all-defined-out))

(define (make-analysis type label start stop step)
  (list (make-param 'TYPE 'COMMAND)
    (make-param 'NAME type)
    (make-param 'LABEL label)
```

```
   (make-param 'START start)
   (make-param 'STOP stop)
   (make-param 'STEP step)))

(define (analysis->string analysis)
  (let ([type (mcdr (first analysis))]
    [name (mcdr (second analysis))]
    [label (mcdr (third analysis))]
    [start (mcdr (fourth analysis))]
    [stop (mcdr (fifth analysis))]
    [step (mcdr (sixth analysis))])
    (if (not (eq? type 'COMMAND))
    (error "analysis must be a command")
    (string-append "."
            (any->string name) " "
            (any->string label) " "
            (any->string start) " "
            (any->string stop) " "
            (any->string step)))))
```

8.3 spice-integration/simulation.ss

```
#lang scheme/base

(require (file "../spice-integration/elements.ss"))
(require (file "../spice-integration/analysis.ss"))
(require (file "../spice-integration/print.ss"))

(require (planet neil/csv:1:5))
(require scheme/system)
(require srfi/1)

(provide (all-defined-out))

;; Circuit simulation internal modelling
(define (make-simulation name circuit models analysis watches)
  (if (not (string? name))
      (error "error: simulation's name must be a string")
      (list (make-param 'TYPE 'SIMULATION)
        (make-param 'NAME name)
        (make-param 'CIRCUIT circuit)
        (make-param 'MODELS models)
        (make-param 'ANALYSIS analysis)
        (make-param 'WATCHES watches))))

(define (simulation->gnucap-cir simu)
  (let* ([name (mcdr (get-param 'NAME simu))]
    [circuit (mcdr (get-param 'CIRCUIT simu))]
    [models (mcdr (get-param 'MODELS simu))]
    [analysis (mcdr (get-param 'ANALYSIS simu))]
    [watches (mcdr (get-param 'WATCHES simu))]
    [print-command (make-print-command (mcdr (get-param 'NAME analysis)) watches)])
    (string-append "\" COGA generated circuit - " name ".cir\n"
          (circuit->string circuit) "\n"
          (fold (lambda (x seed)
              (string-append (model-command->string x) "\n" seed)) "" models)
          "\n"
          (print-command->string print-command) "\n"
          (analysis->string analysis) " > "
          name ".out\n"
          ".END\n")))

(define (simulation->ngspice-cir simu)
  (let* ([name (mcdr (get-param 'NAME simu))]
    [circuit (mcdr (get-param 'CIRCUIT simu))]
    [models (mcdr (get-param 'MODELS simu))]
    [analysis (mcdr (get-param 'ANALYSIS simu))]
    [watches (mcdr (get-param 'WATCHES simu))]
    [print-command (make-print-command (mcdr (get-param 'NAME analysis)) watches)])
    (string-append "\" COGA generated circuit - " name ".cir\n"
          (circuit->string circuit) "\n"
          (fold (lambda (x seed)
              (string-append (model-command->string x) "\n" seed)) "" models)
          "\n"
          (print-command->string print-command) "\n"
          (analysis->string analysis) "\n"
```

```
        ".END\n")))


(define (run-gnucap-simulation simulation)
  (let* ([simu-name (mcdr (get-param 'NAME simulation))]
    [spice-file (string-append simu-name ".cir")]
    [spice-port (open-output-file spice-file
                  #:exists 'replace)]
    [simu-string (simulation->gnucap-cir simulation)]
    [spice-cir-ops (begin (display simu-string spice-port)
             (close-output-port spice-port))]
    [gnucap-run (system (string-append "gnucap -b " spice-file "> /dev/null"))]
    [input-file (string-append simu-name ".out")]
    [input-port (open-input-file input-file)]
    [results (read-string 9999999 input-port)]
    [input-port-ops (close-input-port input-port)]
    [results (begin (read-string 9999999 (open-input-file (string-append simu-name
                                ".out")))])
   (read-simulation-output results)))

(define (run-ngspice-simulation simulation)
  (let* ([simu-name (mcdr (get-param 'NAME simulation))]
    [spice-file (string-append simu-name ".cir")]
    [output-file (string-append simu-name ".out")]
    [spice-port (open-output-file spice-file
                  #:exists 'replace)]
    [simu-string (simulation->ngspice-cir simulation)]
    [port-ops (list (display simu-string spice-port)
          (close-output-port spice-port))]
    [gnucap-run (system (string-append "ngspice -b " spice-file " -r " output-file))]
    [results (read-string 9999999 (open-input-file (string-append simu-name
                                ".out")))])
    (read-simulation-output results)))

(define (read-simulation-output string)
  (let* ([make-my-csv-reader
     (make-csv-reader-maker
      '((separator-chars #\space)
        (strip-leading-whitespace?  . #t)
        (strip-trailing-whitespace? . #t)
        (whitespace-char #\space)))]
    [ascii-list (csv->list (make-my-csv-reader string))])
   (map (lambda (row)
      (filter (lambda (x)
          (not (equal? "" x)))
        row))
    ascii-list)))

(define (get-results-row column index results)
  (let* ([header (car results)]
    [num-of-cols (length header)]
    [liststr (string->list index)]
    [real-index (if (eq? (list-ref liststr (- (length liststr) 1)) #\.)
          (string-append index "0")
          index)])
   (if (> column num-of-cols)
   (error "error: target column is bigger than number of columns")
   (find (lambda (x)
      (equal? (list-ref x column)
        (any->string index)))
      results))))
```

8.4 spice-integration/print.ss

```
#lang scheme/base

(require srfi/1)
(require (file "../spice-integration/elements.ss"))

(provide (all-defined-out))

(define (make-print-command analisys watches)
  (append (list (make-param 'TYPE 'COMMAND)
      (make-param 'ANALISYS analisys))
      watches))
```

```
(define (print-command->string print-command)
  (let ([type (mcdr (first print-command))]
    [analisys (mcdr (second print-command))]
    [watches (cddr print-command)])
    (if (not (eq? type 'COMMAND))
    (error "print command must be of COMMAND type")
    (string-append ".PRINT "
             (any->string analisys) " "
             (fold (lambda (x seed)
               (string-append (watch->string x) " " seed)) "" watches)))))

(define (make-watch nature point) ;point can be a node, an element's label or an index
  (if (not (or (eq? nature 'VOLTAGE)
           (eq? nature 'CURRENT)))
      (error "nature must be VOLTAGE or CURRENT")
      (list (make-param 'TYPE 'WATCH)
        (make-param 'NATURE nature)
        (make-param 'POINT point))))

(define (watch->string watch)
  (let ([type (mcdr (first watch))]
    [nature (mcdr (second watch))]
    [point (mcdr (third watch))])
    (if (not (eq? type 'WATCH))
    (error "watch must be of WATCH type")
    (string-append
     (cond [(eq? nature 'VOLTAGE) "V"]
           [(eq? nature 'CURRENT) "I"])
     "(" (any->string point) ")"))))
```

8.5 spice-integration/hspice-models.ss

```
#lang scheme/base

(require (file "../spice-integration/elements.ss"))

(define (make-hspice-modn)
(make-model-command "MODN" 'NMOS (make-param 'LEVEL 49)
; ----------------------------------------------------------------------
;;;;;;;;;;;;;;;;;;;;;;;;;; SIMULATION PARAMETERS ;;;;;;;;;;;;;;;;;;;;;;;;;;
; ----------------------------------------------------------------------
; format    : HSPICE
; model     : MOS BSIM3v3
; process   : CS[ADFI]
; extracted : CSA C61417; 1998-10; ese(487)
; doc#      : 9933016 REV_N/C
; created   : 1999-01-12
; ----------------------------------------------------------------------
;                       TYPICAL MEAN CONDITION
; ----------------------------------------------------------------------
;       ;;; Flags ;;;
(make-param 'MOBMOD  1.000e+00) (make-param 'CAPMOD 2.000e+00)
;       ;;; Threshold voltage related model parameters ;;;
(make-param 'K1      6.044e-01)
(make-param 'K2      2.945e-03) (make-param 'K3      -1.72e+00) (make-param 'K3B    6.325e-01 )
(make-param 'NCH     2.310e+17) (make-param 'VTH0    4.655e-01)
(make-param 'VOFF    -5.72e-02) (make-param 'DVT0    2.227e+01) (make-param 'DVT1   1.051e+00 )
(make-param 'DVT2    3.393e-03) (make-param 'KETA    -6.21e-04)
(make-param 'PSCBE1  2.756e+08) (make-param 'PSCBE2  9.645e-06)
(make-param 'DVT0W   0.000e+00) (make-param 'DVT1W   0.000e+00) (make-param 'DVT2W  0.000e+00 )
;       ;;; Mobility related model parameters ;;;
(make-param 'UA      1.000e-12) (make-param 'UB      1.723e-18) (make-param 'UC     5.756e-11 )
(make-param 'U0      4.035e+02)
;       ;;; Subthreshold related parameters ;;;
(make-param 'DSUB    5.000e-01) (make-param 'ETA0    3.085e-02) (make-param 'ETAB   -3.95e-02 )
(make-param 'NFACTOR 1.119e-01)
;       ;;; Saturation related parameters ;;;
(make-param 'EM      4.100e+07) (make-param 'PCLM    6.831e-01)
(make-param 'PDIBLC1 1.076e-01) (make-param 'PDIBLC2 1.453e-03) (make-param 'DROUT  5.000e-01 )
(make-param 'A0      2.208e+00) (make-param 'A1      0.000e+00) (make-param 'A2     1.000e+00 )
(make-param 'PVAG    0.000e+00) (make-param 'VSAT    1.178e+05) (make-param 'AGS    2.490e-01 )
(make-param 'B0      -1.76e-08) (make-param 'B1      0.000e+00) (make-param 'DELTA  1.000e-02 )
(make-param 'PDIBLCB 2.583e-01)
;       ;;; Geometry modulation related parameters ;;;
(make-param 'W0      1.184e-07) (make-param 'DLC     8.285e-09)
(make-param 'DWC     2.676e-08) (make-param 'DWB     0.000e+00) (make-param 'DWG    0.000e+00 )
```

```
(make-param 'LL      0.000e+00) (make-param 'LW      0.000e+00) (make-param 'LWL    0.000e+00 )
(make-param 'LLN     1.000e+00) (make-param 'LWN     1.000e+00) (make-param 'WL     0.000e+00 )
(make-param 'WW      0.000e+00) (make-param 'WWL     0.000e+00) (make-param 'WLN    1.000e+00 )
(make-param 'WWN     1.000e+00)
;       ;;; Temperature effect parameters ;;;
(make-param 'AT      3.300e+04) (make-param 'UTE     -1.80e+00)
(make-param 'KT1     -3.30e-01) (make-param 'KT2     2.200e-02) (make-param 'KT1L   0.000e+00 )
(make-param 'UA1     0.000e+00) (make-param 'UB1     0.000e+00) (make-param 'UC1    0.000e+00 )
(make-param 'PRT     0.000e+00)
;       ;;; Overlap capacitance related and dynamic model parameters   ;;;
(make-param 'CGDO    2.100e-10) (make-param 'CGSO    2.100e-10) (make-param 'CGBO   1.100e-10 )
(make-param 'CGDL    0.000e+00) (make-param 'CGSL    0.000e+00) (make-param 'CKAPPA 6.000e-01 )
(make-param 'CF      0.000e+00) (make-param 'ELM     5.000e+00)
(make-param 'XPART   1.000e+00) (make-param 'CLC     1.000e-15) (make-param 'CLE    6.000e-01 )
;       ;;; Parasitic resistance and capacitance related model parameters ;;;
(make-param 'RDSW    6.043e+02)
(make-param 'CDSC    0.000e+00) (make-param 'CDSCB   0.000e+00) (make-param 'CDSCD  8.448e-05 )
(make-param 'PRWB    0.000e+00) (make-param 'PRWG    0.000e+00) (make-param 'CIT    1.000e-03 )
;       ;;; Process and parameters extraction related model parameters ;;;
(make-param 'TOX     7.700e-09) (make-param 'NGATE   0.000e+00)
(make-param 'NLX     1.918e-07)
(make-param 'XL      5.000e-08) (make-param 'XW      0.000e+00)
;       ;;; Substrate current related model parameters ;;;
(make-param 'ALPHA0  0.000e+00) (make-param 'BETA0   3.000e+01)
;       ;;; Noise effect related model parameters ;;;
(make-param 'AF      1.400e+00) (make-param 'KF      2.810e-27) (make-param 'EF     1.000e+00 )
(make-param 'NOIA    1.000e+20) (make-param 'NOIB    5.000e+04) (make-param 'NOIC   -1.40e-12 )
(make-param 'NLEV    0)
;       ;;; Common extrinsic model parameters ;;;
(make-param 'ACM     2)
(make-param 'RD      0.000e+00) (make-param 'RS      0.000e+00) (make-param 'RSH    8.200e+01 )
(make-param 'RDC     0.000e+00) (make-param 'RSC     0.000e+00)
(make-param 'LINT    8.285e-09) (make-param ' WINT    2.676e-08)
(make-param 'LDIF    0.000e+00) (make-param 'HDIF    6.000e-07) (make-param 'WMLT   1.000e+00 )
(make-param 'LMLT    1.000e+00) (make-param 'XJ      3.000e-07)
(make-param 'JS      2.000e-05) (make-param 'JSW     0.000e+00) (make-param 'IS     0.000e+00 )
(make-param 'N       1.000e+00) (make-param 'NDS     1000.)     (make-param 'VNDS   -1.000e+00 )
(make-param 'CBD     0.000e+00) (make-param 'CBS     0.000e+00) (make-param 'CJ     9.300e-04 )
(make-param 'CJSW    2.800e-10) (make-param 'FC      0.000e+00)
(make-param 'MJ      3.100e-01) (make-param 'MJSW    1.900e-01) (make-param 'TT     0.000e+00 )
(make-param 'PB      6.900e-01) (make-param 'PHP     9.400e-01)))
; ---------------------------------------------------------------------

(define (make-hspice-modp)
(make-model-command "MODP" 'PMOS (make-param 'LEVEL 49)
; ---------------------------------------------------------------------
;;;;;;;;;;;;;;;;;;;;;;;;;;; SIMULATION PARAMETERS ;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ---------------------------------------------------------------------
; format    : HSPICE
; model     : MOS BSIM3v3
; process   : CS[ADFI]
; extracted : CSA C61417; 1998-10; ese(487)
; doc#      : 9933016 REV_N/C
; created   : 1999-01-12
; ---------------------------------------------------------------------
;                       TYPICAL MEAN CONDITION
; ---------------------------------------------------------------------
;
;       ;;; Flags ;;;
(make-param 'MOBMOD  1.000e+00) (make-param 'CAPMOD 2.000e+00)
;       ;;; Threshold voltage related model parameters ;;;
(make-param 'K1      5.675e-01)
(make-param 'K2      -4.39e-02) (make-param 'K3      4.540e+00) (make-param 'K3B    -8.52e-01)
(make-param 'NCH     1.032e+17) (make-param 'VTH0   -6.17e-01)
(make-param 'VOFF    -1.13e-01) (make-param 'DVT0    1.482e+00) (make-param 'DVT1   3.884e-01)
(make-param 'DVT2    -1.15e-02) (make-param 'KETA   -2.56e-02)
(make-param 'PSCBE1  1.000e+09) (make-param 'PSCBE2 1.000e-08)
(make-param 'DVT0W   0.000e+00) (make-param 'DVT1W   0.000e+00) (make-param 'DVT2W  0.000e+00)
;       ;;; Mobility related model parameters ;;;
(make-param 'UA      2.120e-10) (make-param 'UB      8.290e-19) (make-param 'UC     -5.28e-11)
(make-param 'U0      1.296e+02)
;       ;;; Subthreshold related parameters ;;;
(make-param 'DSUB    5.000e-01) (make-param 'ETA0    2.293e-01) (make-param 'ETAB   -3.92e-03)
(make-param 'NFACTOR 8.237e-01)
;       ;;; Saturation related parameters ;;;
(make-param 'EM      4.100e+07) (make-param 'PCLM   2.979e+00)
```

```
(make-param 'PDIBLC1 3.310e-02) (make-param 'PDIBLC2 1.000e-09) (make-param 'DROUT  5.000e-01)
(make-param 'A0      1.423e+00) (make-param 'A1      0.000e+00) (make-param 'A2     1.000e+00)
(make-param 'PVAG    0.000e+00) (make-param 'VSAT    2.000e+05) (make-param 'AGS    3.482e-01)
(make-param 'B0      2.719e-07) (make-param 'B1      0.000e+00) (make-param 'DELTA  1.000e-02)
(make-param 'PDIBLCB -1.78e-02)
;       ;;; Geometry modulation related parameters ;;;
(make-param 'W0      4.894e-08) (make-param 'DLC    -5.64e-08)
(make-param 'DWC     3.845e-08) (make-param 'DWB     0.000e+00) (make-param 'DWG    0.000e+00)
(make-param 'LL      0.000e+00) (make-param 'LW      0.000e+00) (make-param 'LWL    0.000e+00)
(make-param 'LLN     1.000e+00) (make-param 'LWN     1.000e+00) (make-param 'WL     0.000e+00)
(make-param 'WW      0.000e+00) (make-param 'WWL     0.000e+00) (make-param 'WLN    1.000e+00)
(make-param 'WWN     1.000e+00)
;       ;;; Temperature effect parameters ;;;
(make-param 'AT      3.300e+04) (make-param 'UTE    -1.35e+00)
(make-param 'KT1    -5.70e-01) (make-param 'KT2     2.200e-02) (make-param 'KT1L   0.000e+00)
(make-param 'UA1     0.000e+00) (make-param 'UB1     0.000e+00) (make-param 'UC1    0.000e+00)
(make-param 'PRT     0.000e+00)
;       ;;; Overlap capacitance related and dynamic model parameters   ;;;
(make-param 'CGDO    2.100e-10) (make-param 'CGSO    2.100e-10) (make-param 'CGBO   1.100e-10)
(make-param 'CGDL    0.000e+00) (make-param 'CGSL    0.000e+00) (make-param 'CKAPPA 6.000e-01)
(make-param 'CF      0.000e+00) (make-param 'ELM     5.000e+00)
(make-param 'XPART   1.000e+00) (make-param 'CLC     1.000e-15) (make-param 'CLE    6.000e-01)
;       ;;; Parasitic resistance and capacitance related model parameters ;;;
(make-param 'RDSW    1.853e+03)
(make-param 'CDSC    6.994e-04) (make-param 'CDSCB   2.943e-04) (make-param 'CDSCD  1.970e-04)
(make-param 'PRWB    0.000e+00) (make-param 'PRWG    0.000e+00) (make-param 'CIT    1.173e-04)
;       ;;; Process and parameters extraction related model parameters ;;;
(make-param 'TOX     7.700e-09) (make-param 'NGATE  0.000e+00)
(make-param 'NLX     1.770e-07)
(make-param 'XL      5.000e-08) (make-param 'XW      0.000e+00)
;       ;;; Substrate current related model parameters ;;;
(make-param 'ALPHA0  0.000e+00) (make-param 'BETA0  3.000e+01)
;       ;;; Noise effect related model parameters ;;;
(make-param 'AF      1.290e+00) (make-param 'KF      1.090e-27) (make-param 'EF     1.000e+00)
(make-param 'NOIA    1.000e+20) (make-param 'NOIB    5.000e+04) (make-param 'NOIC   -1.40e-12)
(make-param 'NLEV    0)
;       ;;; Common extrinsic model parameters ;;;
(make-param 'ACM     2)
(make-param 'RD      0.000e+00) (make-param 'RS      0.000e+00) (make-param 'RSH    1.560e+02)
(make-param 'RDC     0.000e+00) (make-param 'RSC     0.000e+00)
(make-param 'LINT    -5.64e-08) (make-param ' WINT   3.845e-08)
(make-param 'LDIF    0.000e+00) (make-param 'HDIF    6.000e-07) (make-param 'WMLT   1.000e+00)
(make-param 'LMLT    1.000e+00) (make-param 'XJ      3.000e-07)
(make-param 'JS      2.000e-05) (make-param 'JSW     0.000e+00) (make-param 'IS     0.000e+00)
(make-param 'N       1.000e+00) (make-param 'NDS     1000.)     (make-param 'VNDS   -1.000e+00)
(make-param 'CBD     0.000e+00) (make-param 'CBS     0.000e+00) (make-param 'CJ     1.420e-03)
(make-param 'CJSW    3.800e-10) (make-param 'FC      0.000e+00)
(make-param 'MJ      5.500e-01) (make-param 'MJSW    3.900e-01) (make-param 'TT     0.000e+00)
(make-param 'PB      1.020e+00) (make-param 'PHP     9.400e-01)))
; ----------------------------------------------------------------------
```

8.6 ga/common.ss

```
#lang scheme/base

(require srfi/1)

(provide (all-defined-out))

;; Genetic algorithms structures constructors
(define (make-dna . chromosomes)
  chromosomes)

(define (make-individual dna)
  dna)

(define (make-evaluated-individual ind score)
  (cons ind score))

(define (make-population number-of-individuals generation-procedure)
  (if (eq? number-of-individuals 1)
      (list (generation-procedure))
      (append (list (generation-procedure))
          (make-population (- number-of-individuals 1) generation-procedure))))

(define (make-generation selected-individuals cut-procedure mutation-procedure mutation-ratio pop-size)
```

```
  (define (will-mutate?)
    (zero? (random mutation-ratio)))
  (define (generation-iter parent-candidates offspring iters-left)
    (let* ([num-parents (length parent-candidates)]
       [parent-1 (list-ref parent-candidates (random num-parents))]
       [parent-2 (list-ref parent-candidates (random num-parents))]
       [parent-1-size (length parent-1)]
       [parent-2-size (length parent-2)]
       [parent-size (if (< parent-1-size parent-2-size)
              parent-1-size
              parent-2-size)]
       [crossover-offspring (crossover parent-1 parent-2 cut-procedure)]
       [new-offspring (if (will-mutate?)
                (list (mutate (first crossover-offspring) mutation-procedure)
                  (mutate (second crossover-offspring) mutation-procedure))
                crossover-offspring)])
      (cond [(< iters-left 0)
         (error "error: number of iterations left must be grater than zero")]
        [(= iters-left 0) new-offspring]
        [else (append offspring (generation-iter parent-candidates new-offspring (- iters-left 1)))])))
  (let* ([size (length selected-individuals)]
     [pop-minus-selected (- pop-size size)]
     [num-iters (if (= (remainder pop-minus-selected 2) 1)
          (+ (quotient pop-minus-selected 2) 1)
          (quotient pop-minus-selected 2))]
    [selected-without-score (map car selected-individuals)])
    (if (< pop-minus-selected 0)
    (error "error: new generation size must be greater than number of parents, since parents are included also")
    (let ([new-generation (append selected-without-score (generation-iter selected-without-score '() num-iters))])
      (if (even? num-iters)
        new-generation
        (drop-right new-generation 1))))))


;; Basic GA operations
(define (crossover ind1 ind2 cut-procedure)
  (let ([size (length ind1)]
    [size-other (length ind2)])
    (if (not (eq? size size-other))
    (error "error: number of chromosomes must be equal in this procedure")
    (let* ([cut (cut-procedure size)]
         [new-ind1 (append (drop-right ind1 cut) (take-right ind2 cut))]
         [new-ind2 (append (drop-right ind2 cut) (take-right ind1 cut))])
      (list new-ind1 new-ind2)))))

(define (mutate ind mutation-procedure)
  (let* ([size (length ind)]
     [chromosome (random size)])
    (if (not (procedure? mutation-procedure))
    (error "error: mutation-procedure must be a procedure")
    (append (take ind chromosome)
       (list (mutation-procedure chromosome)) ;the mutation depends on the chromosome
       (drop ind (+ chromosome 1))))))


;; Selection procedure
;; This procedure shall receive a population
;; and return a group of selected individuals
;; based on their fitness and the selection procedure.
;; The number of selected individuals will vary according
;; to the selection procedure.
(define (selection population selection-procedure sort-criteria fitness-procedure)
  (let ([fittests (fitness-procedure population)])
    (selection-procedure fittests sort-criteria)))

(define (score ind)
  (cdr ind))

;; Utility methods for GA operators
(define (random-cut size)
  (+ (random (- size 1)) 1))

(define (random-mutation-test dont-care)
  (random 10))

(define (best-first-sort-criteria x y)
```

```
  (> (score x) (score y)))

(define (worst-first-sort-criteria x y)
  (< (score x) (score y)))

(define (truncated-selection fittests sort-criteria)
  (let* ([sorted (sort fittests sort-criteria)]
     [elite-divider 3]
     [threshold (quotient (length fittests) elite-divider)])
    (take sorted threshold)))

(define (truncated-selection-with-dumb-individual fittests sort-criteria)
  (let* ([sorted (sort fittests sort-criteria)]
     [threshold (quotient (length fittests) 5)])
    (append (take sorted (- threshold 1)) (list (last sorted)))))

(define (truncated-selection-intensity ind pop-size)
  (let ([T pop-size]
    [sqrt-1/2pi 0.3989422804014327]
    [fc (cdr ind)])
    (* (/ 1 T) sqrt-1/2pi (exp (- (/ (* fc fc) 2))))))

(define (ga-optimization pop bests individual-string-conversion-procedure good-enough-procedure fitness-procedure cut-procedure
mutation-procedure generation)
  (let* ([size-of-pop (length pop)]
     [current-generation (+ generation 1)]
     [good-enough? good-enough-procedure]
     [fittests (selection pop truncated-selection best-first-sort-criteria fitness-procedure)]
     [new-generation (make-generation fittests cut-procedure mutation-procedure size-of-pop)]
     [new-fittests (fitness-procedure new-generation)]
     [sort-criteria (lambda (x y)
            (> (cdr x) (cdr y)))]
     [best-individual (car (take (sort new-fittests sort-criteria) 1))]
     [best-string (string-append (number->string current-generation)
                  " " (individual-string-conversion-procedure (car best-individual))
                  " " (number->string (cdr best-individual))
                  "\n")])
    (if (good-enough? current-generation) ;TODO generalize better than this
    (string-append bests best-string)
    (begin (display (string-append (number->string current-generation) " "))
         (newline)
         (ga-optimization new-generation
            (string-append bests best-string)
            individual-string-conversion-procedure
            good-enough-procedure
            fitness-procedure
            cut-procedure
            mutation-procedure
            current-generation)))))
```

8.7 ga/current-source-alpha.ss

```
#lang scheme/base

(require (file "../ga/common.ss"))
(require (file "../spice-integration/elements.ss"))
(require (file "../spice-integration/analysis.ss"))
(require (file "../spice-integration/print.ss"))
(require (file "../spice-integration/simulation.ss"))
(require (file "../spice-integration/hspice-models.ss"))

(require srfi/1)

(define (make-cs-alpha-individual ln wn lp wp rs)
  (make-dna ln wn lp wp rs))

(define (cs-alpha-individual->string dna)
  (fold (lambda (x seed)
     (string-append seed "-" x)) (car dna) (cdr dna)))

(define (cs-alpha-individual->filename dna)
  (fold (lambda (x seed)
     (string-append seed "-" x)) (car dna) (cdr dna)))

(define (cs-alpha-chromosome-generator chromosome)
  (cond [(or (= chromosome 0)
```

```
      (= chromosome 2)) (string-append (number->string (/ (+ (random 465) 35) 100.0)) "u")]
   [(or (= chromosome 1)
      (= chromosome 3)) (string-append (number->string (/ (+ (random 990) 10) 10.0)) "u")]
   [(= chromosome 4) (number->string (+ 5000 (random 45000)))]
   [else (error "error: chromosome to mutate doesn't exist")]))

(define (cs-alpha-cut size)
  (random-cut size))

(define (cs-alpha-mut chromosome-position)
  (cs-alpha-chromosome-generator chromosome-position))

(define (make-cs-alpha-mut-ratio)
  (let* ([ratio 50]
    [divider (/ 100 ratio)])
   divider))

(define (make-cs-alpha-tolerance-delta)
  (let* ([delta 5]
    [divider (/ 100 delta)])
   divider))

(define (cs-alpha-good-enough generation values)
  (let* ([dc (car values)]
    [current (cdr values)]
    [junk (begin (display "dc: ")
           (display dc)
           (display "   current: ")
           (display current)
           (newline))])
   (if (or (>= generation 30)
      (and (< dc 2.1)
       (and (< current .53e-6)
           (> current .47e-6))))
      #t #f)))

(define (cs-alpha-individual->circuit individual)
  (let* ([ln (first individual)]
    [wn (second individual)]
    [lp (third individual)]
    [wp (fourth individual)]
    [r (fifth individual)]
    [vpow (make-voltage-source "pow" 'Vdd 0 '3.3V)]
    [vgnd (make-voltage-source "gnd" 'Vss 0 '0.0V)]
    [rs (make-resistor "" 'Vss '3 r)]
    [m1 (make-nmos "1" 'Vss 1 1 'Vss 'MODN (make-param 'L ln) (make-param 'W wn))]
    [m2 (make-nmos "2" 2 1 3 'Vss 'MODN (make-param 'L ln) (make-param 'W wn))]
    [m3 (make-pmos "3" 1 2 'Vdd 'Vdd 'MODP (make-param 'L lp) (make-param 'W wp) (make-param 'M 2))]
    [m4 (make-pmos "4" 'Vdd 2 2 'Vdd 'MODP (make-param 'L lp) (make-param 'W wp))])
   (make-circuit vpow vgnd rs m1 m2 m3 m4)))

(define (cs-alpha-dc-sweep-value tuple)
  (string->number (list-ref tuple 0)))

(define (cs-alpha-output tuple)
  (string->number (list-ref tuple 1)))

(define (score-and-values-cs-alpha-simulation-data results)
  (define (next-iter results-left mean delta min-value last-dc)
    (let* ([value (cs-alpha-output (last results-left))]
      [dc (cs-alpha-dc-sweep-value (last results-left))])
     (cond [(or (> value (+ mean delta))
       (< value (- mean delta))) (list (- last-dc dc) ;to calculate score
                 (cons dc mean))] ;to calculate stop point
      [(equal? dc 0) last-dc]
      [else (next-iter (drop-right results-left 1)
            mean
            delta
            min-value
            last-dc)])))
  (let* ([last-dc (cs-alpha-dc-sweep-value (last results))]
    [last-value (cs-alpha-output (last results))]
    [tolerance-delta (/ last-value 20)]
    [target-current 0.5e-6]
    [weight-1 -1e7]
    [weight-2 1]
```

```
    [current-value-score (abs (- last-value target-current))]
    [stability-calculus-product (next-iter (drop-right results 1) last-value tolerance-delta target-current last-dc)]
    [stability-score (car stability-calculus-product)]
    [stability-values (cdr stability-calculus-product)]
    [candidate-score (+ (* current-value-score weight-1) ;score
                        (* stability-score weight-2))])
   (list candidate-score stability-values)))

(define (evaluate-cs-alpha-individual models dc watches individual)
  (let* ([circuit (cs-alpha-individual->circuit individual)]
    [simulation (make-simulation (cs-alpha-individual->filename individual) circuit models dc watches)]
    [results (cdr (run-gnucap-simulation simulation))])
   (score-and-values-cs-alpha-simulation-data results)))

(define (make-evaluated-cs-alpha-individual individual score-and-values)
  (let ([score (car score-and-values)])
   (make-evaluated-individual individual score)))

(define (cs-alpha-fitness population)
  (let* ([modn (make-hspice-modn)]
    [modp (make-hspice-modp)]
    [models (list modn modp)]
    [dc (make-analysis "DC" "Vpow" '0V '3.3V '0.01V)]
    [watches (list (make-watch 'CURRENT 'M4))]
    [evaluated-individuals (map (lambda (ind) (make-evaluated-cs-alpha-individual ind (evaluate-cs-alpha-individual models dc watches
ind))) population)])
   evaluated-individuals))

(define (make-cs-alpha-population n)
  (make-population n (lambda ()
            (make-cs-alpha-individual (cs-alpha-chromosome-generator 0)
                    (cs-alpha-chromosome-generator 1)
                    (cs-alpha-chromosome-generator 2)
                    (cs-alpha-chromosome-generator 3)
                    (cs-alpha-chromosome-generator 4)))))

(define (ga-cs-alpha-optimization pop bests good-enough-procedure fitness-procedure cut-procedure mutation-procedure mutation-ratio
generation)
  (let* ([size-of-pop (length pop)]
    [current-generation (+ generation 1)]
    [good-enough? good-enough-procedure]
    [fittests (selection pop truncated-selection-with-dumb-individual best-first-sort-criteria fitness-procedure)]
    [new-generation (make-generation fittests cut-procedure mutation-procedure mutation-ratio size-of-pop)]
    [new-fittests (fitness-procedure new-generation)]
    [best-individual (car (take (sort new-fittests best-first-sort-criteria) 1))]
    [worst-individual (car (take (sort new-fittests worst-first-sort-criteria) 1))]
    [selection-intensity (truncated-selection-intensity worst-individual (length fittests))]
    [best-string (string-append "generation "(number->string current-generation)
                    " | champion: " (cs-alpha-individual->string (car best-individual))
                    " | score: " (number->string (cdr best-individual))
                    " | selection intensity: " (number->string selection-intensity)
                    "\n")]
    [modn (make-hspice-modn)] ;workaround to obtain data for my paper
    [modp (make-hspice-modn)]
    [models (list modn modp)]
    [dc (make-analysis "DC" "Vpow" '0V '3.3V '0.01V)]
    [watches (list (make-watch 'CURRENT 'M4))]
    [best-values (caadr (evaluate-cs-alpha-individual models dc watches (car best-individual)))])
   (begin (display (string-append "Generation number " (number->string current-generation) " done..."))
     (newline)
     (if (good-enough? current-generation best-values)
        (string-append bests best-string)
        (ga-cs-alpha-optimization new-generation
                (string-append bests best-string)
                good-enough-procedure
                fitness-procedure
                cut-procedure
                mutation-procedure
                mutation-ratio
                current-generation)))))


(define (do-ga-cs-alpha-optimization)
  (let ([out (open-output-file "optimization.txt"
                #:exists 'replace)]
   [pop (make-cs-alpha-population 20)])
```

```
    (begin
      (display "COGA - optimizing Alpha current source circuit") (newline)
      (display "Time spent: ")
      (time (display
        (ga-cs-alpha-optimization pop
                  "Coefficients aproximation optimization:\n"
                  cs-alpha-good-enough
                  cs-alpha-fitness
                  cs-alpha-cut
                  cs-alpha-mut
                  (make-cs-alpha-mut-ratio)
                  0)
        out))
      (display "Optimization finished") (newline)
      (close-output-port out))))

(do-ga-cs-alpha-optimization)
```

8.8 ga/current-source-beta.ss

```
#lang scheme/base

(require (file "../ga/common.ss"))
(require (file "../spice-integration/elements.ss"))
(require (file "../spice-integration/analysis.ss"))
(require (file "../spice-integration/print.ss"))
(require (file "../spice-integration/simulation.ss"))
(require (file "../spice-integration/hspice-models.ss"))

(require srfi/1)

(define (make-cs-beta-individual ln wn lp wp rs)
  (make-dna ln wn lp wp rs))

(define (cs-beta-individual->string dna)
  (fold (lambda (x seed)
     (string-append seed "-" x)) (car dna) (cdr dna)))

(define (cs-beta-individual->filename dna)
  (fold (lambda (x seed)
     (string-append seed "-" x)) (car dna) (cdr dna)))

(define (cs-beta-chromosome-generator chromosome)
  (cond [(or (= chromosome 0)
        (= chromosome 2)) (string-append (number->string (/ (+ (random 465) 35) 100.0)) "u")]
    [(or (= chromosome 1)
        (= chromosome 3)) (string-append (number->string (/ (+ (random 990) 10) 10.0)) "u")]
    [(= chromosome 4) (number->string (+ 5000 (random 45000)))]
    [else (error "error: chromosome to mutate doesn't exist")]))

(define (cs-beta-cut size)
  (random-cut size))

(define (cs-beta-mut chromosome-position)
  (cs-beta-chromosome-generator chromosome-position))

(define (make-cs-beta-mut-ratio)
  (let* ([ratio 50]
     [divider (/ 100 ratio)])
    divider))

(define (make-cs-beta-tolerance-delta)
  (let* ([delta 5]
     [divider (/ 100 delta)])
    divider))

(define (cs-beta-good-enough generation values)
  (let* ([dc (car values)]
     [current (cdr values)]
     [junk (begin (display "dc: ")
            (display dc)
            (display "   current: ")
            (display current)
            (newline))])
    (if (or (>= generation 30)
        (and (< dc 1.8)
```

```
       (and (< current .53e-6)
            (> current .47e-6))))
      #t #f)))

(define (cs-beta-individual->circuit individual)
  (let* ([ln (first individual)]
    [wn (second individual)]
    [lp (third individual)]
    [wp (fourth individual)]
    [r (fifth individual)]
    [vpow (make-voltage-source "pow" 'Vdd 0 '3.3V)]
    [vgnd (make-voltage-source "gnd" 'Vss 0 '0.0V)]
    [rs (make-resistor "" 'Vss '3 r)]
    [m1 (make-nmos "1" 'Vss 1 2 'Vss 'MODN (make-param 'L ln) (make-param 'W wn))]
    [m2 (make-nmos "2" 1 1 3 'Vss 'MODN (make-param 'L ln) (make-param 'W wn))]
    [m3 (make-nmos "3" 2 4 4 'Vss 'MODN (make-param 'L ln) (make-param 'W wn))]
    [m4 (make-nmos "4" 5 4 1 'Vss 'MODN (make-param 'L ln) (make-param 'W wn))]
    [m5 (make-pmos "5" 4 5 'Vdd 'Vdd 'MODP (make-param 'L lp) (make-param 'W wp) (make-param 'M 2))]
    [m6 (make-pmos "6" 'Vdd 5 5 'Vdd 'MODP (make-param 'L lp) (make-param 'W wp))])
   (make-circuit vpow vgnd rs m1 m2 m3 m4 m5 m6)))

(define (cs-beta-dc-sweep-value tuple)
  (string->number (list-ref tuple 0)))

(define (cs-beta-output tuple)
  (string->number (list-ref tuple 1)))

(define (score-and-values-cs-beta-simulation-data results)
  (define (next-iter results-left mean delta min-value last-dc)
    (let* ([value (cs-beta-output (last results-left))]
      [dc (cs-beta-dc-sweep-value (last results-left))])
     (cond [(or (> value (+ mean delta))
       (< value (- mean delta))) (list (- last-dc dc) ;to calculate score
                 (cons dc mean))] ;to calculate stop point
       [(equal? dc 0) last-dc]
       [else (next-iter (drop-right results-left 1)
             mean
             delta
             min-value
             last-dc)])))
  (let* ([last-dc (cs-beta-dc-sweep-value (last results))]
    [last-value (cs-beta-output (last results))]
    [tolerance-delta (/ last-value 50)]
    [target-current 0.5e-6]
    [weight-1 -1e7]
    [weight-2 1]
    [current-value-score (abs (- last-value target-current))]
    [stability-calculus-product (next-iter (drop-right results 1) last-value tolerance-delta target-current last-dc)]
    [stability-score (car stability-calculus-product)]
    [stability-values (cdr stability-calculus-product)]
    [candidate-score (+ (* current-value-score weight-1) ;score
             (* stability-score weight-2))])
   (list candidate-score stability-values)))

(define (evaluate-cs-beta-individual models dc watches individual)
  (let* ([circuit (cs-beta-individual->circuit individual)]
    [simulation (make-simulation (cs-beta-individual->filename individual) circuit models dc watches)]
    [results (cdr (run-gnucap-simulation simulation))])
   (score-and-values-cs-beta-simulation-data results)))

(define (make-evaluated-cs-beta-individual individual score-and-values)
  (let ([score (car score-and-values)])
   (make-evaluated-individual individual score)))

(define (cs-beta-fitness population)
  (let* ([modn (make-hspice-modn)]
    [modp (make-hspice-modp)]
    [models (list modn modp)]
    [dc (make-analysis "DC" "Vpow" '0V '3.3V '0.01V)]
    [watches (list (make-watch 'CURRENT 'M6))]
    [evaluated-individuals (map (lambda (ind) (make-evaluated-cs-beta-individual ind (evaluate-cs-beta-individual models dc watches
ind))) population)])
   evaluated-individuals))

(define (make-cs-beta-population n)
  (make-population n (lambda ()
```

```
            (make-cs-beta-individual (cs-beta-chromosome-generator 0)
                  (cs-beta-chromosome-generator 1)
                  (cs-beta-chromosome-generator 2)
                  (cs-beta-chromosome-generator 3)
                  (cs-beta-chromosome-generator 4)))))

(define (ga-cs-beta-optimization pop bests good-enough-procedure fitness-procedure cut-procedure mutation-procedure mutation-ratio
generation)
  (let* ([size-of-pop (length pop)]
    [current-generation (+ generation 1)]
    [good-enough? good-enough-procedure]
    [fittests (selection pop truncated-selection-with-dumb-individual best-first-sort-criteria fitness-procedure)]
    [new-generation (make-generation fittests cut-procedure mutation-procedure mutation-ratio size-of-pop)]
    [new-fittests (fitness-procedure new-generation)]
    [sort-criteria (lambda (x y)
          (> (cdr x) (cdr y)))]
    [best-individual (car (take (sort new-fittests sort-criteria) 1))]
    [worst-individual (car (take (sort new-fittests worst-first-sort-criteria) 1))]
    [selection-intensity (truncated-selection-intensity worst-individual (length fittests))]
    [best-string (string-append "generation "(number->string current-generation)
                " | champion: " (cs-beta-individual->string (car best-individual))
                " | score: " (number->string (cdr best-individual))
                " | selection intensity: " (number->string selection-intensity)
                "\n")]
    [modn (make-hspice-modn)]
    [modp (make-hspice-modp)]
    [models (list modn modp)]
    [dc (make-analysis "DC" "Vpow" '0V '3.3V '0.01V)]
    [watches (list (make-watch 'CURRENT 'M6))]
    [best-values (caadr (evaluate-cs-beta-individual models dc watches (car best-individual)))])
   (begin (display (string-append "Generation number " (number->string current-generation) " done..."))
     (newline)
     (if (good-enough? current-generation best-values)
        (string-append bests best-string)
        (ga-cs-beta-optimization new-generation
              (string-append bests best-string)
              good-enough-procedure
              fitness-procedure
              cut-procedure
              mutation-procedure
              mutation-ratio
              current-generation)))))


(define (do-ga-cs-beta-optimization)
  (let ([out (open-output-file "optimization.txt"
              #:exists 'replace)]
    [pop (make-cs-beta-population 30)])
    (begin
      (display "COGA - optimizing Beta current source circuit") (newline)
      (display "Time spent: ")
      (time (display
        (ga-cs-beta-optimization pop
              "Coefficients aproximation optimization:\n"
              cs-beta-good-enough
              cs-beta-fitness
              cs-beta-cut
              cs-beta-mut
              (make-cs-beta-mut-ratio)
              0)
        out))
      (display "Optimization finished") (newline)
      (close-output-port out))))

(do-ga-cs-beta-optimization)
```

8.9 ga/current-source-gamma.ss

```
#lang scheme/base

(require (file "../ga/common.ss"))
(require (file "../spice-integration/elements.ss"))
(require (file "../spice-integration/analysis.ss"))
(require (file "../spice-integration/print.ss"))
(require (file "../spice-integration/simulation.ss"))
(require (file "../spice-integration/hspice-models.ss"))
```

```
(require srfi/1)

(define (make-cs-gamma-individual ln-1-2 ln-3-4 wn-1-3 wn-2-4 lp-5-6-8 wp-5-6 wp-8 ln-7 wn-7 rs)
  (make-dna ln-1-2 ln-3-4 wn-1-3 wn-2-4 lp-5-6-8 wp-5-6 wp-8 ln-7 wn-7 rs))

(define (cs-gamma-individual->string dna)
  (fold (lambda (x seed)
     (string-append seed "-" x)) (car dna) (cdr dna)))

(define (cs-gamma-individual->filename dna)
  (fold (lambda (x seed)
     (string-append seed "-" x)) (car dna) (cdr dna)))

(define (cs-gamma-chromosome-generator chromosome)
  (cond [(or (= chromosome 0)
        (= chromosome 1)
        (= chromosome 4)
        (= chromosome 7)) (string-append (number->string (/ (+ (random 465) 35) 100.0)) "u")]
    [(or (= chromosome 2)
        (= chromosome 3)
        (= chromosome 5)
        (= chromosome 6)
        (= chromosome 8)) (string-append (number->string (/ (+ (random 990) 10) 10.0)) "u")]
    [(= chromosome 9) (number->string (+ 5000 (random 45000)))]
    [else (error "error: chromosome to mutate doesn't exist")]))

(define (cs-gamma-cut size)
  (random-cut size))

(define (cs-gamma-mut chromosome-position)
  (cs-gamma-chromosome-generator chromosome-position))

(define (make-cs-gamma-mut-ratio)
  (let* ([ratio 50]
     [divider (/ 100 ratio)])
    divider))

(define (make-cs-gamma-tolerance-delta)
  (let* ([delta 5]
     [divider (/ 100 delta)])
    divider))

(define (cs-gamma-good-enough generation values)
  (let* ([dc (car values)]
     [current (cdr values)]
     [junk (begin (display "dc: ")
             (display dc)
             (display "   current: ")
             (display current)
             (newline))])
    (if (or (>= generation 30)
        (and (< dc 1.8)
         (and (< current .53e-6)
             (> current .47e-6))))
        #t #f)))

(define (cs-gamma-individual->circuit individual)
  (let* ([ln-1-2 (first individual)]
     [ln-3-4 (second individual)]
     [wn-1-3 (third individual)]
     [wn-2-4 (fourth individual)]
     [lp-5-6-8 (fifth individual)]
     [wp-5-6 (sixth individual)]
     [wp-8 (seventh individual)]
     [ln-7 (eighth individual)]
     [wn-7 (ninth individual)]
     [r (tenth individual)]
     [vpow (make-voltage-source "pow" 'Vdd 0 '3.3V)]
     [vgnd (make-voltage-source "gnd" 'Vss 0 '0.0V)]
     [rs (make-resistor "" 'Vss '4 r)]
     [m1 (make-nmos "1" 'Vss 1 2 'Vss 'MODN (make-param 'L ln-1-2) (make-param 'W wn-1-3))]
     [m2 (make-nmos "2" 3 1 4 'Vss 'MODN (make-param 'L ln-1-2) (make-param 'W wn-2-4))]
     [m3 (make-nmos "3" 2 6 1 'Vss 'MODN (make-param 'L ln-3-4) (make-param 'W wn-1-3))]
     [m4 (make-nmos "4" 5 6 3 'Vss 'MODN (make-param 'L ln-3-4) (make-param 'W wn-2-4))]
     [m5 (make-pmos "5" 1 5 'Vdd 'Vdd 'MODP (make-param 'L lp-5-6-8) (make-param 'W wp-5-6) (make-param 'M 2))]
```

```
    [m6 (make-pmos "6" 'Vdd 5 5 'Vdd 'MODP (make-param 'L lp-5-6-8) (make-param 'W wp-5-6))]
    [m7 (make-nmos "7" 'Vss 6 6 'Vss 'MODN (make-param 'L ln-7) (make-param 'W wn-7))]
    [m8 (make-pmos "8" 6 5 'Vdd 'Vdd 'MODP (make-param 'L lp-5-6-8) (make-param 'W wp-8))])
   (make-circuit vpow vgnd rs m1 m2 m3 m4 m5 m6 m7 m8)))

(define (cs-gamma-dc-sweep-value tuple)
  (string->number (list-ref tuple 0)))

(define (cs-gamma-output tuple)
  (string->number (list-ref tuple 1)))

(define (score-and-values-cs-gamma-simulation-data results)
  (define (next-iter results-left mean delta min-value last-dc)
    (let* ([value (cs-gamma-output (last results-left))]
      [dc (cs-gamma-dc-sweep-value (last results-left))])
     (cond [(or (> value (+ mean delta))
       (< value (- mean delta))) (list (- last-dc dc) ;to calculate score
                (cons dc mean))] ;to calculate stop point
       [(equal? dc 0) last-dc]
       [else (next-iter (drop-right results-left 1)
             mean
             delta
             min-value
             last-dc)])))
  (let* ([last-dc (cs-gamma-dc-sweep-value (last results))]
    [last-value (cs-gamma-output (last results))]
    [tolerance-delta (/ last-value 50)]
    [target-current 0.5e-6]
    [weight-1 -1e7]
    [weight-2 1]
    [current-value-score (abs (- last-value target-current))]
    [stability-calculus-product (next-iter (drop-right results 1) last-value tolerance-delta target-current last-dc)]
    [stability-score (car stability-calculus-product)]
    [stability-values (cdr stability-calculus-product)]
    [candidate-score (+ (* current-value-score weight-1) ;score
             (* stability-score weight-2))])
   (list candidate-score stability-values)))

(define (evaluate-cs-gamma-individual models dc watches individual)
  (let* ([circuit (cs-gamma-individual->circuit individual)]
    [simulation (make-simulation (cs-gamma-individual->filename individual) circuit models dc watches)]
    [results (cdr (run-gnucap-simulation simulation))])
   (score-and-values-cs-gamma-simulation-data results)))

(define (make-evaluated-cs-gamma-individual individual score-and-values)
  (let ([score (car score-and-values)])
   (make-evaluated-individual individual score)))

(define (cs-gamma-fitness population)
  (let* ([modn (make-hspice-modn)]
    [modp (make-hspice-modp)]
    [models (list modn modp)]
    [dc (make-analysis "DC" "Vpow" '0V '3.3V '0.01V)]
    [watches (list (make-watch 'CURRENT 'M6))]
    [evaluated-individuals (map (lambda (ind) (make-evaluated-cs-gamma-individual ind (evaluate-cs-gamma-individual models dc
watches ind))) population)])
   evaluated-individuals))

(define (make-cs-gamma-population n)
  (make-population n (lambda ()
           (make-cs-gamma-individual (cs-gamma-chromosome-generator 0)
                (cs-gamma-chromosome-generator 1)
                (cs-gamma-chromosome-generator 2)
                (cs-gamma-chromosome-generator 3)
                (cs-gamma-chromosome-generator 4)
                (cs-gamma-chromosome-generator 5)
                (cs-gamma-chromosome-generator 6)
                (cs-gamma-chromosome-generator 7)
                (cs-gamma-chromosome-generator 8)
                (cs-gamma-chromosome-generator 9)))))

(define (ga-cs-gamma-optimization pop bests good-enough-procedure fitness-procedure cut-procedure mutation-procedure mutation-
ratio generation)
  (let* ([size-of-pop (length pop)]
    [current-generation (+ generation 1)]
    [good-enough? good-enough-procedure]
```

```
    [fittests (selection pop truncated-selection-with-dumb-individual best-first-sort-criteria fitness-procedure)]
    [new-generation (make-generation fittests cut-procedure mutation-procedure mutation-ratio size-of-pop)]
    [new-fittests (fitness-procedure new-generation)]
    [sort-criteria (lambda (x y)
            (> (cdr x) (cdr y)))]
    [best-individual (car (take (sort new-fittests sort-criteria) 1))]
    [worst-individual (car (take (sort new-fittests worst-first-sort-criteria) 1))]
    [selection-intensity (truncated-selection-intensity worst-individual (length fittests))]
    [best-string (string-append "generation "(number->string current-generation)
                  " | champion: " (cs-gamma-individual->string (car best-individual))
                  " | score: " (number->string (cdr best-individual))
                  " | selection intensity: " (number->string selection-intensity)
                  "\n")]
    [modn (make-hspice-modn)] ;workaround to obtain data for my paper
    [modp (make-hspice-modp)]
    [models (list modn modp)]
    [dc (make-analysis "DC" "Vpow" '0V '3.3V '0.01V)]
    [watches (list (make-watch 'CURRENT 'M6))]
    [best-values (caadr (evaluate-cs-gamma-individual models dc watches (car best-individual)))])
   (begin (display (string-append "Generation number " (number->string current-generation) " done..."))
     (newline)
     (if (good-enough? current-generation best-values)
         (string-append bests best-string)
         (ga-cs-gamma-optimization new-generation
                 (string-append bests best-string)
                 good-enough-procedure
                 fitness-procedure
                 cut-procedure
                 mutation-procedure
                 mutation-ratio
                 current-generation)))))


(define (do-ga-cs-gamma-optimization)
  (let ([out (open-output-file "optimization.txt"
               #:exists 'replace)]
    [pop (make-cs-gamma-population 30)])
    (begin
      (display "COGA - optimizing Gamma current source circuit") (newline)
      (display "Time spent: ")
      (time (display
        (ga-cs-gamma-optimization pop
                  "Coefficients aproximation optimization:\n"
                  cs-gamma-good-enough
                  cs-gamma-fitness
                  cs-gamma-cut
                  cs-gamma-mut
                  (make-cs-gamma-mut-ratio)
                  0)
        out))
      (display "Optimization finished") (newline)
      (close-output-port out))))

(do-ga-cs-gamma-optimization)
```

8.10 ga/current-source-delta.ss

```
#lang scheme/base

(require (file "../ga/common.ss"))
(require (file "../spice-integration/elements.ss"))
(require (file "../spice-integration/analysis.ss"))
(require (file "../spice-integration/print.ss"))
(require (file "../spice-integration/simulation.ss"))
(require (file "../spice-integration/hspice-models.ss"))

(require srfi/1)

(define (make-cs-delta-individual ln-1-2-7 ln-4 wn-1 wn-2-4 wn-7 lp-3 lp-5-6-10 wp-5-3 wp-6 wp-10 ln-8 wp-8 ln-9 wn-9 rs)
  (make-dna ln-1-2-7 ln-4 wn-1 wn-2-4 wn-7 lp-3 lp-5-6-10 wp-5-3 wp-6 wp-10 ln-8 wp-8 ln-9 wn-9 rs))

(define (cs-delta-individual->string dna)
  (fold (lambda (x seed)
     (string-append seed "-" x)) (car dna) (cdr dna)))

(define (cs-delta-individual->filename dna)
```

```
  (fold (lambda (x seed)
     (string-append seed "-" x)) (car dna) (cdr dna)))

(define (cs-delta-chromosome-generator chromosome)
  (cond  [(or (= chromosome 0)
         (= chromosome 1)
         (= chromosome 5)
         (= chromosome 6)
         (= chromosome 10)
         (= chromosome 12)) (string-append (number->string (/ (+ (random 50) 1) 10.0)) "u")]
    [(or (= chromosome 2)
         (= chromosome 3)
         (= chromosome 4)
         (= chromosome 7)
         (= chromosome 8)
         (= chromosome 9)
         (= chromosome 11)
         (= chromosome 13)) (string-append (number->string (/ (+ (random 1000) 1) 10.0)) "u")]
    [(= chromosome 14) (number->string (+ 5000 (random 45000)))]
    [else (error (string-append "error: chromosome to mutate doesn't exist -> " (number->string chromosome)))]))

(define (cs-delta-cut size)
  (random-cut size))

(define (cs-delta-mut chromosome-position)
  (cs-delta-chromosome-generator chromosome-position))

(define (make-cs-delta-mut-ratio)
  (let* ([ratio 50]
    [divider (/ 100 ratio)])
   divider))

(define (make-cs-delta-tolerance-delta)
  (let* ([delta 5]
    [divider (/ 100 delta)])
   divider))

(define (cs-delta-good-enough generation values)
  (let* ([dc (car values)]
    [current (cdr values)]
    [junk (begin (display "dc: ")
           (display dc)
           (display "   current: ")
           (display current)
           (newline))])
   (if (or (>= generation 30)
      (and (< dc 1.8)
       (and (< current .53e-6)
           (> current .47e-6))))
   #t #f)))

(define (cs-delta-individual->circuit individual)
  (let* ([ln-1-2-7 (first individual)]
    [ln-4 (second individual)]
    [wn-1 (third individual)]
    [wn-2-4 (fourth individual)]
    [wn-7 (fifth individual)]
    [lp-3 (sixth individual)]
    [lp-5-6-10 (seventh individual)]
    [wp-5-3 (eighth individual)]
    [wp-6 (ninth individual)]
    [wp-10 (tenth individual)]
    [ln-8 (list-ref individual 10)]
    [wp-8 (list-ref individual 11)]
    [ln-9 (list-ref individual 12)]
    [wn-9 (list-ref individual 13)]
    [r (list-ref individual 14)]
    [vpow (make-voltage-source "pow" 'Vdd 0 '3.3V)]
    [vgnd (make-voltage-source "gnd" 'Vss 0 '0.0V)]
    [rs (make-resistor "" 'Vss '3 r)]
    [m1 (make-nmos "1" 'Vss 1 1 'Vss 'MODN (make-param 'L ln-1-2-7) (make-param 'W wn-1))]
    [m2 (make-nmos "2" 2 1 3 'Vss 'MODN (make-param 'L ln-1-2-7) (make-param 'W wn-2-4))]
    [m3 (make-pmos "3" 1 7 4 'Vdd 'MODP (make-param 'L lp-3) (make-param 'W wp-5-3))]
    [m4 (make-pmos "4" 5 6 2 'Vss 'MODN (make-param 'L ln-4) (make-param 'W wn-2-4))]
    [m5 (make-pmos "5" 4 5 'Vdd 'Vdd 'MODP (make-param 'L lp-5-6-10) (make-param 'W wp-5-3))]
    [m6 (make-pmos "6" 'Vdd 5 5 'Vdd 'MODP (make-param 'L lp-5-6-10) (make-param 'W wp-6))]
```

```
    [m7 (make-pmos "7" 'Vss 1 7 'Vss 'MODN (make-param 'L ln-1-2-7) (make-param 'W wn-7) (make-param 'M 2))]
    [m8 (make-pmos "8" 7 7 'Vdd 'Vdd 'MODP (make-param 'L ln-8) (make-param 'W wp-8))]
    [m9 (make-pmos "9" 6 6 'Vss 'Vss 'MODN (make-param 'L ln-9) (make-param 'W wn-9))]
    [m10 (make-pmos "10" 'Vdd 5 6 'Vdd 'MODP (make-param 'L lp-5-6-10) (make-param 'W wp-10))])
   (make-circuit vpow vgnd rs m1 m2 m3 m4 m5 m6 m7 m8 m9 m10)))

(define (cs-delta-dc-sweep-value tuple)
  (string->number (list-ref tuple 0)))

(define (cs-delta-output tuple)
  (string->number (list-ref tuple 1)))

(define (score-and-values-cs-delta-simulation-data results)
  (define (next-iter results-left mean delta min-value last-dc)
    (let* ([value (cs-delta-output (last results-left))]
      [dc (cs-delta-dc-sweep-value (last results-left))])
     (cond [(or (> value (+ mean delta))
       (< value (- mean delta))
       (< value min-value)) (list (- last-dc dc) ;to calculate score
                (cons dc mean))] ;to calculate stop point
      [(equal? dc 0) last-dc]
      [else (next-iter (drop-right results-left 1)
            mean
            delta
            min-value
            last-dc)])))
  (let* ([last-dc (cs-delta-dc-sweep-value (last results))]
    [last-value (cs-delta-output (last results))]
    [tolerance-delta (/ last-value 50)]
    [target-current 0.5e-6]
    [weight-1 -1e7]
    [weight-2 1]
    [current-value-score (abs (- last-value target-current))]
    [stability-calculus-product (next-iter (drop-right results 1) last-value tolerance-delta target-current last-dc)]
    [stability-score (car stability-calculus-product)]
    [stability-values (cdr stability-calculus-product)]
    [candidate-score (+ (* current-value-score weight-1) ;score
            (* stability-score weight-2))])
   (list candidate-score stability-values)))

(define (evaluate-cs-delta-individual models dc watches individual)
  (let* ([circuit (cs-delta-individual->circuit individual)]
    [simulation (make-simulation (cs-delta-individual->filename individual) circuit models dc watches)]
    [results (cdr (run-gnucap-simulation simulation))])
   (score-and-values-cs-delta-simulation-data results)))

(define (make-evaluated-cs-delta-individual individual score-and-values)
  (let ([score (car score-and-values)])
   (make-evaluated-individual individual score)))

(define (cs-delta-fitness population)
  (let* ([modn (make-hspice-modn)]
    [modp (make-hspice-modp)]
    [models (list modn modp)]
    [dc (make-analysis "DC" "Vpow" '0V '3.3V '0.01V)]
    [watches (list (make-watch 'CURRENT 'M6))]
    [evaluated-individuals (map (lambda (ind) (make-evaluated-cs-delta-individual ind (evaluate-cs-delta-individual models dc watches
ind))) population)])
   evaluated-individuals))

  (define (make-cs-delta-population n)
   (make-population n (lambda ()
        (make-cs-delta-individual (cs-delta-chromosome-generator 0)
                (cs-delta-chromosome-generator 1)
                (cs-delta-chromosome-generator 2)
                (cs-delta-chromosome-generator 3)
                (cs-delta-chromosome-generator 4)
                (cs-delta-chromosome-generator 5)
                (cs-delta-chromosome-generator 6)
                (cs-delta-chromosome-generator 7)
                (cs-delta-chromosome-generator 8)
                (cs-delta-chromosome-generator 9)
                (cs-delta-chromosome-generator 10)
                (cs-delta-chromosome-generator 11)
                (cs-delta-chromosome-generator 12)
                (cs-delta-chromosome-generator 13)
```

```
                (cs-delta-chromosome-generator 14)))))

  (define (ga-cs-delta-optimization pop bests good-enough-procedure fitness-procedure cut-procedure mutation-procedure mutation-
ratio generation)
    (let* ([size-of-pop (length pop)]
      [current-generation (+ generation 1)]
      [good-enough? good-enough-procedure]
      [fittests (selection pop truncated-selection-with-dumb-individual best-first-sort-criteria fitness-procedure)]
      [new-generation (make-generation fittests cut-procedure mutation-procedure mutation-ratio size-of-pop)]
      [new-fittests (fitness-procedure new-generation)]
      [sort-criteria (lambda (x y)
            (> (cdr x) (cdr y)))]
      [best-individual (car (take (sort new-fittests sort-criteria) 1))]
      [worst-individual (car (take (sort new-fittests worst-first-sort-criteria) 1))]
      [selection-intensity (truncated-selection-intensity worst-individual (length fittests))]
      [best-string (string-append "generation "(number->string current-generation)
                  " | champion: " (cs-delta-individual->string (car best-individual))
                  " | score: " (number->string (cdr best-individual))
                  " | selection intensity: " (number->string selection-intensity)
                  "\n")]
      [modn (make-hspice-modn)]
      [modp (make-hspice-modn)]
      [models (list modn modp)]
      [dc (make-analysis "DC" "Vpow" '0V '3.3V '0.01V)]
      [watches (list (make-watch 'CURRENT 'M6))]
      [best-values (caadr (evaluate-cs-delta-individual models dc watches (car best-individual)))])
     (begin (display (string-append "Generation number " (number->string current-generation) " done..."))
       (newline)
       (if (good-enough? current-generation best-values)
       (string-append bests best-string)
       (ga-cs-delta-optimization new-generation
                  (string-append bests best-string)
                  good-enough-procedure
                  fitness-procedure
                  cut-procedure
                  mutation-procedure
                  mutation-ratio
                  current-generation)))))


  (define (do-ga-cs-delta-optimization)
    (let ([out (open-output-file "optimization.txt"
              #:exists 'replace)]
     [pop (make-cs-delta-population 30)])
     (begin
    (display "COGA - optimizing Delta current source circuit") (newline)
    (display "Time spent: ")
    (time (display
          (ga-cs-delta-optimization pop
                "Coefficients aproximation optimization:\n"
                cs-delta-good-enough
                cs-delta-fitness
                cs-delta-cut
                cs-delta-mut
                (make-cs-delta-mut-ratio)
                0)
          out))
    (display "Optimization finished") (newline)
    (close-output-port out))))

  (do-ga-cs-delta-optimization)
```

9 Referências

[AMS10] AMS, 0,35 um CMOS process technology. 2010. http://www.austriamicrosystems.com/05foundry/indexc35.htm, acesso em Maio, 2010

[BAR02] BARESEL, A; et al, Fitness function design to im-prove evolutionary structural testing. Genetic andEvolutionary Computation Conference (GECCO 2002) , p. 1329–1336, Nova Iorque, 2002. Morgan Kaufmann.

[DYB09] DYBVIG, R.K., The Scheme programming language. Quarta edição, MIT Press, Massachusetts, 2009.

[ECC10] eCircuit Center, Spice basic. 2010. http://www.ecircuitcenter.com/Basics.htm, acesso em Maio/2010

[IT05] ITRS, International technology roadmap for semiconductors. 2005. http://www.itrs.net/reports.html, acesso em Maio/2010

[QUA93] QUARLES, T; et al, Spice3 version user's manual. Manual de usuário. 1993.

[RAN04] HAUPT, R.L., HAUPT, S.E., Pratical genetic algorithms. Segunda edição, Wiley-Interscience, Malden, 2004.

[RAZ03] RAZAVI, B., Design of analog CMOS integrated circuits. Primeira edição, McGraw-Hill Science, Columbus, 2003.

[SAN02] SANO, Y, KITA, H., Optimization of noisy fitness functionsby means of genetic algorithms using history of search with test ofestimation. Congress on EvolutionaryComputation CEC2002 (2002) , p. 360–365, Nova Jersey, 2002. IEEE Press.

[SIL08] SILVA, E.S.C., Projeto de fontes de referência de baixa tensão em tecnologia CMOS. 2008. Trabalho de Conclusão de Curso - Departamento de Engenharia Elétrica, Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2008.

[ZEB02] ZEBULUM, R.C.; et al, Evolutionary electronics. Primeira edição, CRC Press, Washington D.C., 2002.