Síntese de Circuitos Digitais Otimizados via Programação Genética

Sérgio Granato de Araújo, Antônio C. Mesquita, Aloysio C. P. Pedroza

[1]Departamento de Eng. Elétrica – Universidade Federal do Rio de Janeiro (UFRJ)
Caixa Postal 68.504 – 21945-970 – Rio de Janeiro – RJ – Brasil

{granato,aloysio}@gta.ufrj.br, mesquita@coe.ufrj.br

Resumo. *Este trabalho apresenta uma metodologia para o projeto de circuitos eletrônicos digitais otimizados a partir de descrições feitas em alto nível de abstração. A metodologia utiliza Programação Genética e ferramentas de síntese de alto nível para melhorar a qualidade do projeto (otimização de área). Foi utilizado um algoritmo de otimização com múltiplos objetivos, composto de dois estágios, para buscar por circuitos com a funcionalidade desejada e com restrições de área. Um experimento de projeto em FPGA de uma função que aproxima a raiz quadrada ilustra a metodologia.*

Abstract. *This paper presents a methodology for the design of optimized electronic digital systems from high abstraction level descriptions. The methodology uses Genetic Programming in addition to high-level synthesis tools to improve the design quality (area optimization). A two-stage, multiobjective optimization algorithm was used to search for circuits with the desired functionality subjected additionally to chip area constraints. Experiment with a square-root approximation function design targeted to FPGA illustrates the methodology.*

1. Introdução

Nos últimos anos, o projeto de circuitos integrados (CIs) tem objetivado atender às necessidades do mercado no menor espaço de tempo possível. Para alcançar tal objetivo faz-se necessário automatizar todo o projeto, desde sua concepção até o produto final. Esta filosofia de projeto, conhecida como *descreva-e-sintetize* [1], utiliza uma abordagem *top-down*, na qual a funcionalidade do sistema é descrita através de um modelo comportamental, simulada para verificar sua correta operação e implementada utilizando ferramentas de síntese de alto nível.

A síntese comportamental de circuitos digitais provê uma maneira efetiva de tratar a complexidade demandada pelos circuitos atuais. Ela proporciona a redução em uma ordem de magnitude do número de objetos que o projetista tem a considerar, encurtando o ciclo de projeto [2]. O tempo a mais disponível pode ser utilizado para investigar implementações alternativas utilizando ferramentas de síntese automáticas. Este procedimento é conhecido como *exploração do espaço de soluções* [3].

O processo da criação de um CI compreende várias etapas, a primeira delas consistindo na especificação da funcionalidade do sistema. Há uma variedade de modelos conceituais que podem ser utilizados para este propósito, como detalhado em

[2]. Máquinas de estados finitos (MEFs), MEFs com fluxo de dados (FSMD: *Finite-State Machine Datapath*) e redes de Petri (e suas extensões) são alguns exemplos. Linguagens de programação são modelos conceituais de interesse particular, visto que são capazes de expressar simultaneamente dados, atividade e controle [4]. Após a especificação da funcionalidade do sistema é realizada a síntese de alto nível (HLS: *High-Level Synthesis*), que transforma o modelo comportamental em um modelo estrutural no nível de transferência de registradores (RTL: *Register-transfer level*). Em seguida, são realizadas as sínteses lógica e de *layout*.

Linguagens de programação de *hardware*, mais conhecidas como linguagens de descrição de *hardware* (HDLs: *Hardware Description Languages*), tais como VHDL (*Very High-Speed Integrated Circuit (VHSIC) HDL*) [5], capturam a funcionalidade do sistema de uma forma inteligível pelo computador. VHDL é uma notação formal desenvolvida para ser utilizada em todas as fases da criação de projetos eletrônicos. Este formalismo suporta documentação, verificação, descrição em vários níveis de abstração, síntese e teste de projetos eletrônicos, sendo amplamente aceito pela comunidade de CAD eletrônico. Em contrapeso, VHDL possui semântica projetada com intuito de simulação, implicando que apenas um subconjunto de instruções VHDL possa ser sintetizado.

Embora existam atualmente muitas ferramentas de CAD disponíveis, possibilitando o projeto de circuitos eletrônicos em alto nível de abstração, o mercado ainda resiste em adotá-las. Gajski *et al.* [6] apontam duas razões para isso: baixa qualidade de resultados e ausência de interação entre o projetista e a ferramenta *durante* o processo de síntese. Especificações em alto nível de abstração são normalmente consideradas "pouco sintetizáveis" uma vez que, neste nível de abstração, muitos detalhes de implementação são mascarados (por definição) do projetista [7]. Para resolver este problema as ferramentas de síntese comportamental restringem o espaço de soluções assumindo uma arquitetura alvo na qual todas as especificações em alto nível são mapeadas. O resultado é uma solução localmente otimizada para cada projeto.

Por outro lado, as FPGAs (*Field Programmable Gate Arrays*) têm-se tornado a primeira opção em prototipagem. Estatísticas recentes indicam que, hoje em dia, mais da metade dos projetos de CIs é iniciada utilizando FPGAs [8]. As FPGAs eliminam a necessidade de percorrer as etapas convencionais de produção de um CI, reduzindo o risco financeiro e o tempo de desenvolvimento. Adicionalmente, modernas ferramentas de síntese oferecem suave migração entre as tecnologias FPGA e ASIC (*Application-Specific Integrated Circuits*), proporcionando mapeamento direto entre projetos com FPGAs e bibliotecas ASIC [9].

Este trabalho apresenta uma metodologia de projeto de circuitos digitais baseada em técnicas de *hardware* evolutivo [10]. Esta metodologia utiliza Programação Genética como paradigma de aprendizado automático para explorar o espaço de soluções na busca por estruturas funcionais otimizadas. A Programação Genética enfatiza características desejáveis tais como o uso de representações de procedimentos, gerando soluções diretamente como programas de computador. Como conseqüência, e desde que os problemas que se deseja resolver possam ser reformulados como problemas de indução de programas, ela se torna uma ferramenta mais adequada para a solução desses problemas.

O restante do artigo encontra-se organizado como se segue. A seção 2 apresenta os trabalhos relacionados com a presente linha de pesquisa. Na seção 3 são introduzidos os conceitos de Computação Evolutiva e de Programação Genética. A metodologia de projeto encontra-se detalhada na seção 4. Na seção 5 é apresentado um exemplo de emprego da metodologia. A seção 6 conclui o artigo.

2. Trabalhos Relacionados

Os algoritmos evolutivos têm sido empregados em níveis mais baixos do projeto de circuitos eletrônicos, incluindo posicionamento (*placement*) e interconexão de células [11], bem como na automação do projeto estrutural desses circuitos. Mais recentemente, os algoritmos evolutivos têm sido utilizados na busca por implementações com requisitos de qualidade. Esta seção descreve os principais trabalhos dentro deste contexto.

Trabalhos recentes [12, 13, 14, 15] apresentam estruturas evoluídas inteiramente em *software* que utilizam simulação para avaliar soluções intermediárias. Esta abordagem é conhecida como *evolução extrínseca* ou *evolução indireta*. Em [12] é apresentado um algoritmo que evolui circuitos aritméticos a partir de uma matriz de células lógicas de FPGAs, sendo capaz de redescobrir implementações convencionais para o somador de dois bits e de reduzir o número de portas lógicas utilizadas na construção do multiplicador de dois bits, se comparado às técnicas com intervenção humana. Neste trabalho, circuitos funcionais com área otimizada foram encontrados por acaso, isto é, sem recorrer a funções de avaliação com múltiplos objetivos.

Em [13] é proposta uma função com múltiplos objetivos onde restrições de área têm de ser atendidas ao lado da funcionalidade do circuito. Esta função possui dois estágios. Inicialmente são buscadas soluções 100% corretas, o que ocorre quando a tabela verdade é comprovada. Em seguida, o número de portas lógicas utilizadas no circuito é incorporado na função de aptidão. Em [14] uma função com múltiplos objetivos, similar à apresentada em [13], é abordada. A principal diferença com a anterior é o uso de uma técnica baseada em população para dividir a busca entre várias sub-populações, cada uma delas admitindo um único objetivo. Quando estes objetivos são alcançados, suas correspondentes sub-populações são combinadas, o que contribui para minimizar a divergência produzida entre os circuitos codificados e a tabela verdade.

Em [15] é apresentado um ambiente de projeto no qual restrições de performance têm de ser atendidas ao lado da funcionalidade do circuito. Neste ambiente, não apenas portas lógicas, mas blocos funcionais (por exemplo, o meio-somador) são disponibilizados na evolução. A principal contribuição deste trabalho é a idéia de utilizar um *único* programa VHDL para evoluir uma população de descrições de *hardware*, o que reduz substancialmente o tempo de compilação. Neste caso, cada indivíduo é descrito em um PROCESSO VHDL distinto.

Uma alternativa às abordagens acima mencionadas é a utilização de gramáticas formais para representar a evolução de autômatos, como no trabalho de Hemmi *et al.* [16]. Nesta abordagem, especificações de *hardware* são geradas automaticamente como programas HDL. Os autores utilizaram PG para evoluir árvores a partir de produções (regras de re-escrita na gramática HDL) para criar descrições de *hardware*. A aptidão é

avaliada através de ferramentas de simulação e a implementação é gerada somente *após* o processo de aprendizado ser concluído. Este sistema foi utilizado para gerar automaticamente um somador binário seqüencial.

3. Computação Evolutiva e Programação Genética

A Computação Evolutiva (CE) consiste de uma classe de algoritmos probabilísticos inspirados em princípios de seleção natural e variações, que se beneficiam do aumento do poder de simulação para resolver problemas complexos, problemas de controle e aplicações de descobrimento de conhecimento [17]. O Algoritmo Genético (AG) e a Programação Genética (PG) são as instâncias de CE mais utilizadas em abordagens de *hardware* evolutivo [18].

Os algoritmos evolutivos são estratégias gerais de busca sendo, portanto, classificados como *métodos fracos* de solução de problemas. Tais métodos partem de requisitos relaxados, requerendo pouco ou nenhum conhecimento sobre o domínio do problema, tendo, conseqüentemente, um amplo leque de aplicação. O problema dos métodos fracos é que o mecanismo de controle de busca pode não ser suficientemente poderoso. Em contraste, um *método forte* de solução de problemas é aquele que se aprovisiona de uma quantidade significativa de conhecimento específico do domínio do problema que pretende resolver. O problema dos métodos fortes, como por exemplo, dos sistemas especialistas, é que o conhecimento explícito torna-se um produto que determina a eficácia da técnica.

O AG [19] emula a maneira com que a natureza apura as características dos seres vivos. O *cromossomo* é o componente básico de um AG. Ele representa um ponto, chamado neste contexto de um *indivíduo*, do espaço de busca (ou solução) do problema. A aptidão é uma medida utilizada para indicar quão próximo um indivíduo está da solução. Pela Aplicação iterativa de operadores genéticos (cálculo da aptidão, seleção baseada na aptidão, reprodução, cruzamento e mutação) em uma *população* de indivíduos iniciada randomicamente, tal população evolui no sentido de gerar soluções potenciais (indivíduos apurados) para o dado problema.

A PG é derivada do AG, sendo que a principal diferença está na representação da solução. Enquanto o AG, na sua forma simples, representa soluções através de cadeias de bits de tamanho fixo, a PG codifica genótipos de tamanhos variados possibilitando codificações mais complexas. O AG encontra aplicação em sistemas de otimização onde o problema é muito complexo e não linear. A PG, por permitir que o tamanho dos cromossomos varie, é mais adequada para criar estruturas.

Koza [20] introduziu a PG canônica com genoma baseado em árvores utilizando a linguagem LISP. Desde então outros tipos de genoma têm sido propostos, como genomas lineares e gráficos [21]. Outros sistemas de PG foram desenvolvidos para criar automaticamente programas em linguagens arbitrárias (diferentes de LISP) [22]. Como alternativa as abordagens anteriores, gramáticas formais têm sido utilizadas para codificar conhecimento [23]. Em [24 – 26] programas foram criados combinando PG com gramática livre de contexto (*context-free language*). Tais sistemas ficaram conhecidos como G^3P (*Grammar-guided Genetic Programming*). G^3P provê um ambiente para criação automática de programas livres de erros em uma linguagem arbitrária. GPK (*Genetic Programming Kernel*) [25] é um sistema G^3P e foi utilizado

como núcleo da PG neste trabalho. GPK evolui programas em linguagem arbitrária desde que uma BNF (*Backus-Naur form*) seja fornecida como entrada.

4. Metodologia

A metodologia de projeto de circuitos digitais combinacionais evolui uma população de programas VHDL buscando uma solução (descrição de *hardware*) que, conjuntamente, implemente a funcionalidade desejada e satisfaça a restrição de área. Esta metodologia é mostrada na Figura 1 onde se nota dois de seus principais componentes: o núcleo do sistema de PG (GPK) e a função de avaliação. Diferentemente de outras abordagens [15, 16], a compilação e o *place-and-route* (posicionamento e interligação de células) são realizados para cada individuo da população, fornecendo ao sistema a informação exata do uso de recursos.

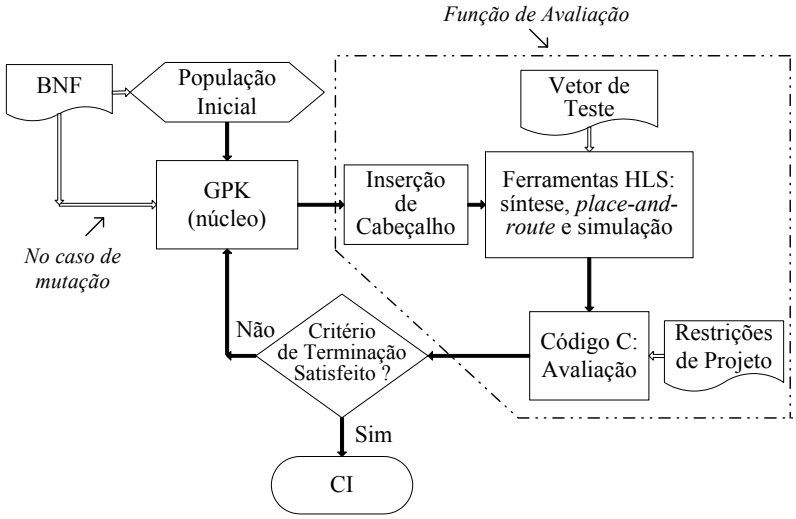


Figura 1. Metodologia de projeto de circuitos digitais.

Inicialmente, uma população de programas é criada de modo aleatório com a restrição dada por uma descrição BNF pré-definida, contendo um subconjunto da gramática VHDL e respeitando a propriedade de suficiência. Tal propriedade estabelece que as funções e operandos escolhidos pelo usuário devem ser capazes de expressar a solução do problema [20]. Este princípio também é utilizado por outros paradigmas de aprendizado automático. Na seqüência, uma função objetivo é utilizada para avaliar cada indivíduo (genoma) e atribuir um valor de aptidão a cada um deles. Se nenhum indivíduo atingir o valor esperado da função objetivo uma nova população é gerada utilizando os dois operadores genéticos primários, cruzamento (recombinação) e reprodução (clonagem), bem como o operador genético mutação. Cada um desses operadores genéticos é aplicado com uma determinada probabilidade. O sistema executa até que o critério de terminação seja satisfeito. Desde que a presente metodologia diz respeito às soluções desconhecidas, o sistema termina quando o número de gerações excede a um número máximo pré-determinado (*generational predicate*).

Para criar o código fonte VHDL, a declaração da entidade e a parte de declaração da arquitetura de um programa VHDL são mantidas fixas, pois são comuns a todos os indivíduos da população. Somente a parte de procedimentos do código VHDL será afetada pelo processo evolutivo.

O cálculo da função objetivo é um procedimento chave de um sistema de PG. As etapas envolvidas nesta operação são mostradas na Figura 2. Após a geração de uma população é necessário completar o código fonte VHDL de cada indivíduo inserindo a declaração da entidade e a parte de declaração da arquitetura. Em seguida, cada indivíduo é sintetizado utilizando a ferramenta de desenvolvimento ALTERA [27]. Para calcular efetivamente a aptidão de cada indivíduo faz-se necessário simular cada um dos casos de aptidão, isto é, casos de treinamento, e checar estes resultados com os valores desejados.

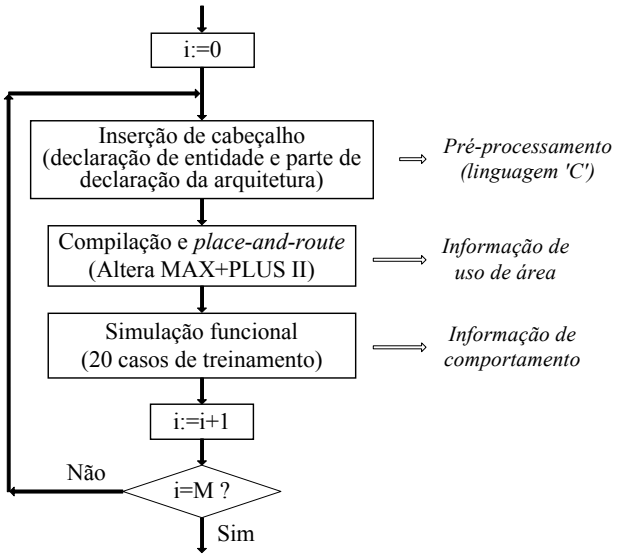


Figura 2. Bloco de Avaliação da Figura 1.

A função objetivo a ser minimizada é uma função de erro com múltiplos parâmetros, combinando *funcionalidade* e *área* em uma abordagem com dois estágios similar à adotada em [13]. Inicialmente, a especificação funcional deve ser atendida (*F1*); em seguida, o sistema busca por implementações, funcionalmente corretas, com uso de recursos otimizados (*F2*). Seja o erro médio de funcionalidade, fe_{av}, definido por:

$$fe_{av} = \frac{1}{N}\sum_{i=1}^{N} w_i \left| D_i - S_i \right| \tag{1}$$

onde D_i é o valor desejado para o caso de treinamento i, S_i é o valor obtido (simulado) para o caso de treinamento i, w_i é um fator de ponderação e N é o número de casos de treinamento. A função de aptidão F tem a forma:

$$F = \begin{cases} F_1 = fe_{av} & if\ \exists i \in \{1,\cdots,N\} \mid \left| \dfrac{D_i - S_i}{D_i} \right| > te \\ F_2 = fe_{av} * k(lu) & \text{otherwise} \end{cases} \tag{2}$$

onde *te* é o erro de especificação funcional desejado para os casos de treinamento, *lu* é o número de elementos lógicos utilizados e *k* é uma função de *lu*. A função *k(lu)* é um fator de ponderação que considera o efeito da área utilizada para implementar a funcionalidade desejada. *k(lu)* é uma função exponencial de *lu* que

privilegia indivíduos contendo as menores quantidades de elementos lógicos. A função *k(lu)* tem a forma:

$$k(lu) = ae^{b*lu} \tag{3}$$

onde *a* e *b* são constantes que devem ser definidas resolvendo o seguinte sistema:

$$\begin{aligned} k_{\min} &= ae^{b*lu_{\min}} \\ k_{\max} &= ae^{b*lu_{\max}} \end{aligned} \tag{4}$$

onde lu_{min} e lu_{max} são a mínima e a máxima quantidade de elementos lógicos e k_{min} e k_{max} são os valores mínimo e máximo de *k(lu)*, respectivamente.

5. Experimento

Sob o ponto de vista de arquitetura, o projeto de um circuito digital pode ser dividido em duas partes principais [1]: a unidade de controle, na maioria das vezes modelada por uma MEF, e o fluxo de dados (ou *datapath*), que implementa os blocos funcionais que afetam (podem alterar) os dados de entrada. O presente projeto consiste em sintetizar automaticamente um *datapath* com a aplicação de técnicas evolutivas. A função a ser implementada pelo *datapath* é avaliada através de uma função de aptidão, que leva em consideração o número de células lógicas utilizadas para implementar o circuito.

5.1. Descrição do Projeto

O presente projeto consiste em sintetizar automaticamente um *datapath* que realiza uma aproximação da raiz quadrada da soma do quadrado de dois números (*a* e *b*). A função que se deseja implementar foi obtida em [6], que utilizou a seguinte fórmula:

$$\sqrt{a^2 + b^2} \cong max((0{,}875x+0{,}5y),x) \tag{5}$$

onde $x = max(|a|,|b|)$, e $y = min(|a|,|b|)$.

Neste problema de *regressão simbólica múltipla* o objetivo é encontrar uma função com duas variáveis independentes, na forma simbólica, que tenha a aproximação desejada para um grupo de 20 conjuntos de pontos numéricos. A Figura 3 mostra a "caixa preta" do projeto do *datapath*. Nesta figura, os sinais de entrada "início" e de saída "fim" estão presentes para exercer funções de controle.

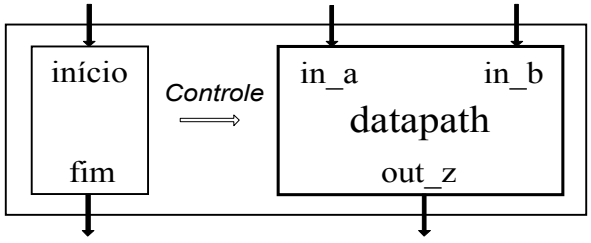


Figura 3. "Caixa preta" do projeto do *datapath*.

5.2. Definição da BNF

Uma BNF descreve estruturas admissíveis (sintaxe) na construção de uma linguagem através de 4 tuplas $\{S,N,T,P\}$, onde S denota o símbolo inicial, N denota um conjunto de símbolos não terminais, T denota um conjunto de símbolos terminais e P são as produções, isto é, regras de transformação de *string*, que mapeiam os elementos de N em T. A seguinte BNF foi definida para resolver o problema em questão:

```
S     ::= <fe>;
<fe>  ::= "PROCESS BEGIN WAIT UNTIL (clk'event and clk='1'); IF
     in_a>=in_b THEN x<=in_a; y<=in_b; ELSE y<=in_a; x<=in_b; END IF;
     t6<=" <expr> "; IF t6>=x THEN t7<=t6; ELSE t7<=x; END IF;
     out_z<=t7; done<='1'; END PROCESS;";
<expr>::= "(" <expr> ")" <op> "(" <expr> ")" | <var> |<var> "(7 downto
     " <shf_size> ")";
<op>  ::= " + " | " - ";
<var> ::= " x " | " y " | " w ";
<shf_size> ::= 1"|"2"|"3"|"4"|"5"|"6"|"7";
```

Na descrição BNF acima, os sinais *x*, *y*, *w*, *t6* e *t7* são sinais internos à arquitetura VHDL. O sinal *w*, ao qual se atribui zero, está presente para facilitar a inserção e remoção de um terminal. As operações *max* e *min* já estão inseridas na descrição BNF através da cláusula *IF*. Por fim, o problema original foi restringido a utilizar os operandos de entrada e saída como *unsigned* ao invés de inteiro, reduzindo o custo computacional.

5.3. Principais Atributos de PG

As principais características para o primeiro estágio (*F1*) do sistema de PG estão definidas na Tabela 1.

Tabela 1. Características de PG para o primeiro estágio (*F1*) do projeto do *datapath*.

Objetivo:	Criar um programa VHDL que implemente uma função de dois argumentos que satisfaça uma amostra de 20 conjuntos $\{a_i, b_i, F(a_i,b_i)\}$, onde a função alvo seja definida por $F(a,b) = sqrt(a^2 + b^2)$.
Símbolos terminais:	*x*, *y*, *w*, +, –, *7 downto* (*operador lógico deslocar à direita*), *1*, *2*, *3*, *4*, *5*, *6*, *7* (*índice do operador lógico deslocar à direita*) e a sentença entre aspas no lado direito de <*fe*>.
Símbolos não-terminais:	<*fe*>, <*expr*>, <*op*>, <*var*>, <*shf_size*>.
Casos de treinamento:	Amostra de 20 conjuntos de dados $\{a_i, b_i, F(a_i,b_i)\}$, a_i e b_i escolhidos randomicamente a partir do intervalo [0, 255] e satisfazendo a restrição $sqrt(a_i^2 + b_i^2) \leq 255$.
Aptidão:	A média, a partir de 20 casos de treinamento, do valor absoluto da diferença entre o valor da variável dependente produzida pela simulação de programas VHDL sintetizados e o valor alvo da variável dependente (Cf. equação (1)).
Método de seleção:	Proporcional à aptidão.
Parâmetros principais:	M=100, G=51, p_c=0,73 (cruzamento em um ponto), p_r=0,15, p_m= 0,12, sem elitismo.

No segundo estágio da evolução (*F2*) os parâmetros de PG permaneceram inalterados, exceto as linhas "objetivo" e "aptidão" da Tabela 1 que modificaram para incorporar a busca por implementações com recursos otimizados.

5.4. Resultados Preliminares

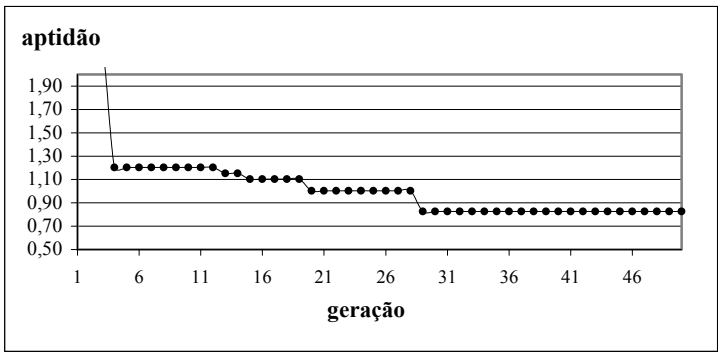
Para a realização deste experimento foi utilizado o componente FPGA EPF8282A da Altera [27] como arquitetura alvo, contendo 282 elementos lógicos. Cada elemento lógico consiste em uma tabela de *look-up*, isto é, um gerador de funções que computa qualquer função de quatro variáveis, e um registrador programável. Neste experimento, a área será medida em números de elementos lógicos. O erro funcional (*te*) foi definido em 2,5% (Cf. equações (1) e (2)). Ele estabelece o erro máximo entre a variável dependente, resultado da simulação de programas VHDL evoluídos, e o valor desejado da variável dependente, para cada um dos casos de treinamento.

Foi utilizada uma população de 100 indivíduos, que evoluiu até um máximo de 51 gerações. Este experimento utilizou N=20, w_i=1 e $k(lu)$=$0{,}25e^{(0{,}0139*lu)}$, que garante $k(lu) \in [0.5,1.0]$ para $lu \in [50,100]$ (Cf. equações (3) e (4)). A população inicial foi gerada através do método *ramped half-and-half* [20], substituindo o procedimento de inicialização do GPK criticado em [26] pela dificuldade associada em criar a primeira geração.

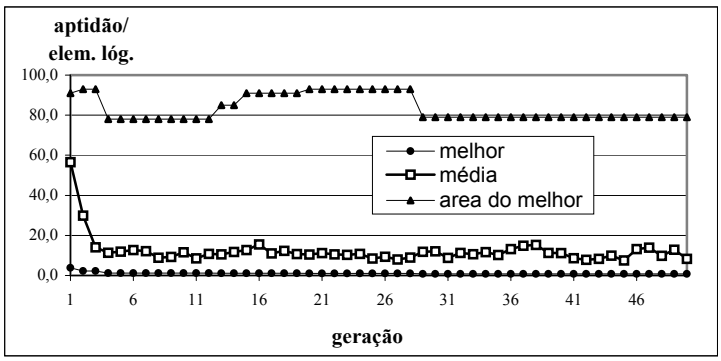
O experimento foi realizado com 10 execuções independentes. Na geração 18, de uma destas execuções, surgiu o primeiro indivíduo funcional, ou seja, um programa VHDL no qual todos os casos de treinamento atingiram a especificação funcional do erro (*te*=2.5%). A partir deste ponto, o sistema iniciou a busca por implementações otimizadas. Durante o segundo estágio da evolução vários indivíduos atenderam à especificação funcional, embora com uso não otimizado de recursos. Indivíduos funcionais com 90, 92 e 95 elementos lógicos foram encontrados. Somente na geração 29 o melhor indivíduo emergiu. Sua aptidão F foi de 0,827 e o erro médio de funcionalidade, (fe_{av}), 1,103. Este indivíduo foi sintetizado com 78 elementos lógicos. A parte da arquitetura do programa VHDL deste indivíduo é mostrada abaixo:

```
BEGIN
   w <="00000000";
   PROCESS BEGIN WAIT UNTIL (clk'event and clk='1');
     IF in_a >= in_b THEN x <= in_a;  y <= in_b;
     ELSE y <= in_a;  x <= in_b; END IF;
     t6 <=((y(7 downto 1))+((x)-(x(7 downto
            3))))+(y(7 downto 6));
     IF t6 >= x THEN t7 <= t6;
     ELSE t7 <= x; END IF;
     out_z <= t7;  done <= '1';
   END PROCESS;
```

A Figura 4(a) mostra a evolução da aptidão do melhor indivíduo. A Figura 4(b) mostra, em destaque, a evolução da área do melhor indivíduo. Nesta figura observa-se que o melhor indivíduo aumenta sua área a partir da 13ª geração para melhorar a aproximação da função. Na 18ª geração este indivíduo atinge a aproximação desejada e inicia a busca por implementações com área otimizada, o que é alcançado na geração 29. A Tabela 2 mostra um quadro comparativo entre a área utilizada e a precisão da aproximação da função para outros valores de *te*.



a)



b)

Figura 4. (a) Histograma da Aptidão e (b) da área do melhor indivíduo.

Esta tabela mostra que o indivíduo com área ótima para *te* = 25% não utilizou o sinal interno *y*, o menor entre as duas entradas. Isto indica que o sistema busca, de fato, por implementações com área otimizada, levando em conta a quantidade de recursos e o roteamento utilizados pela ferramenta de síntese.

Tabela 2. Quantidade de elementos lógicos para outros valores de *te* com respectivas descrições de *hardware*.

te	Elementos Lógicos	Descrição de *hardware*
25%	51	`x + x(7 downto 3)`
15%	58	`x + y(7 downto 2)`
12%	59	`x + y(7 downto 1)`
5%	70	`x - x(7 downto 3) + y(7 downto 1)`

Em algumas execuções notou-se que o primeiro indivíduo funcional tinha também a menor área. No entanto, isto nem sempre se comprovou, como mostrado para o caso de *te*=2,5%. Alternativamente, foi realizada uma experiência adicionando o operador matemático *multiplicação* e o operador lógico *deslocar à esquerda* no lado

direito dos símbolos não terminais <*op*> e <*expr*>, respectivamente. O sistema mostrou-se robusto, eliminando, nas primeiras gerações, indivíduos contendo estes dois operadores.

Como desvantagem a metodologia apresentada mostrou-se lenta. As ferramentas atualmente disponíveis para a compilação de programas VHDL exigem alto custo computacional, particularmente para funções *place-and-route* [28]. Uma execução para uma população de 100 indivíduos consumiu tempo médio de 3 min por geração num microcomputador com arquitetura X86 de 1GHz de relógio e 256MB de memória RAM, sob o sistema operacional Windows98. Neste caso, para obter o melhor indivíduo foram necessários (29 × 3) min = 1 h e 27 min de processamento naquela execução particular.

Para contornar este problema foi introduzido um pré-estágio a ser executado antes do início do processo principal de otimização (Figura 1). Neste pré-estágio o sistema evolui um único programa VHDL, como em [15], com objetivo único de induzir toda a população a aproximar da funcionalidade desejada. Neste pré-estágio a função *place-and-route* é eliminada. O pré-estágio termina quando pelo menos dois indivíduos funcionais surgem. Esta alteração na metodologia aumentou a performance do sistema em aproximadamente 30%.

6. Conclusão

As técnicas evolutivas, ao lado das redes neurais, se apresentaram como sendo as principais alternativas aos métodos simbólicos de aprendizado de automático [29]. Na última década as técnicas evolutivas têm transposto a barreira entre a teoria e a prática, encontrando aplicações em inúmeros problemas reais. Uma de suas principais vantagens é a simplificação que elas permitem na formulação de problemas complexos, o que eleva o nível de abstração que o projetista tem a considerar. A maior vantagem, nestes ambientes de projeto, refere-se à facilidade de contar com pessoal não especializado, com pouco ou nenhum conhecimento do domínio do problema.

Este trabalho apresentou uma metodologia baseada em Programação Genética para gerar implementações em hardware. Tal metodologia evolui uma população de programas VHDL para gerar uma solução que atenda à funcionalidade especificada, ao mesmo tempo em que busca por implementações com área reduzida. A metodologia é completamente automática no sentido de que não é necessário escrever código e apenas definir um subconjunto da gramática utilizada através de uma BNF.

Do presente trabalho pode-se concluir:

i. Não é necessário especificar exatamente os operadores básicos na BNF;
ii. A otimização de performance pode ser alcançada ao invés de (ou adicionalmente a) otimização de área, bastando para tal inserir na função objetivo parâmetros de tempo. Caso o parâmetro de performance seja utilizado conjuntamente com o parâmetro de área, pode-se utilizar a estratégia de otimização por *agregação de objetivos* [30], que obtém uma medida escalar de aptidão a partir de um vetor de objetivos;
iii. A metodologia independe da ferramenta de síntese de alto nível.

Referências

1. Gajski, D., Dutt, N., Wu, A. and Lin, S. (1992), "High Level Synthesis: Introduction to Chip and System Design", Kluver Academic Publishers.
2. Gajski, D., Zhu, J. and Dörner, R. (1997), "Special Issues in Codesign", Technical Report ICS-97-26.
3. Thompson, A. (2000) "Exploration in Design Space, Unconventional Electronics Design Through Artificial Evolution", IEEE Transactions in Evolutionary Computation.
4. Hessel, F. e Marchioro, G. (1997) "Concepção Conjunta Hardware/Software (Co-design)", XVI Jornada de Atualização em Informática, Congresso da Sociedade Brasileira de Computação, pp. 449-494.
5. IEEE Std. 1076-1993 (1993), "VHDL Language Reference Manual".
6. Gajski, D., Tadatoshi, I., Chaiyakul, V., Juan, H. and Hadley, T. (1996), "A Design Methodology and Environment for Interactive Behavioral Synthesis", Tech. Rep.. 96-29.
7. Kuusilinna, K., Hämäläinen, T. and Saarinen, J. (1999), "Practical VHDL Optimization for Timing Critical FPGA Applications", Microprocessors and Microsystems 23, pp. 459-469.
8. Landis, D. (1995), "Programmable Logic and Application Specific Integrated Circuits, Handbook of Components for Electronics", Chapter II, Vol. 1.
9. Synopsys Inc. (2001), "Synopsys Online Documentation".
10. DeGaris, H. (1993), "Evolvable Hardware: Genetic Programming of a Darwin Machine", McGraw-Hill, Artificial Neural Nets and GAs, Springer-Verlag, NY.
11. Bennett III, F., Koza, J., Yu, J. and Mydlowec, W. (2000), "Automatic Synthesis, Placement and Routing of an Amplifier Circuit by Means of Genetic Programming", Proc. of the 3rd Int. Conf. on Evolvable Systems, ICES 2000, pp. 1-10, Edinburgh, UK.
12. Miller, J., Thomson, P. and Fogarty (1997), T., "Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study", Genetic Algorithms and Evolution Strategies in Engineering and Computer Science, edited by D. Quagliarella et al., Publisher: Wiley.
13. Kalganova, T. and Miller, J. (1999), "Evolving More Efficient Digital Circuits by Allowing Circuit Layout Evolution and Multi-Objective Fitness", Proc. of the 1st NASA /DoD Workshop on Evolvable Hardware, pp. 54-63, Los Alamitos, CA, IEEE Computer Society Press.
14. Coello, C., Aguirre, A. and Buckles, B. (2000), "Evolutionary Multiobjective Design of Combinational Logic Circuits", Proc. of the 2nd NASA/DoD Work. on Evolvable Hardware, pp. 161-170, IEEE C.S., Los Alamitos, CA.
15. Hounsell, B. and Arslan, T. (2000), "A Novel Evolvable Hardware Framework for the Evolution of High Performance Digital Circuits", GECCO-2000, pp. 525-532.
16. Hemmi, H., Mizoguchi, J. and Shimohara, K. (1996), "Development and Evolution

of Hardware Behaviors", Towards Evolvable Hardware: Int. Workshop, Lausanne, edited by E. Sanchez and M. Tomassini, Springer-Verlag, LNCS 1062, pp. 250-265.

17. Rosca, J. (1997), "Hierarchical Learning with Procedural Abstractions Mechanisms", PhD Thesis, University of Rochester, New York.
18. Tomassini, M., (1996), "Evolutionary Algorithms", Towards Evolvable Hardware, Int. Workshop, Lausanne, edited by E. Sanchez and M. Tomassini, Springer-Verlag, LNCS 1062, pp. 19-47.
19. Holland, J. (1975), "Adaptation in Natural and Artificial Systems", The University of Michigan, 1st Edition.
20. Koza, J., (1992) "Genetic Programming: On the Programming of Computers by Means of Natural Selection", MIT Press.
21. Banzhaf, W., Nordin, P., Keller, R. and Francone, F. (1997), *Genetic Programming: An Introduction,* San Francisco, CA, Morgan Kaufmann and Heidelberg.
22. Keller, R. and Banzhaf, W. 1996), "Genetic Programming Using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes", Proc. of Genetic Programming 1996, pp. 116-122, MIT Press.
23. Ratle, A. and Sebag, M. (2000), "Genetic Programming and Domain Knowledge: Beyond the Limitations of Grammar-Guided Machine Discovery", PPSN.
24. Whigham, P. (1995), "Grammatically-Based Genetic Programming", Proc. of the Workshop on Genetic Programming: From Theory to Real-World Applications, pp. 33-41, Morgan Kauffmann Publishers.
25. Hörner, H. (1996), "A C++ Class Library for Genetic Programming", Release 1.0 Operating Instructions, Viena University of Economy.
26. Ryan, C. and O'Neill, M. (1998), "Grammatical Evolution: Evolving Programs for an Arbitrary Language", LNCS 1391, Proc. of the First European Workshop on Genetic Programming, pp. 83-95, Springer-Verlag.
27. Altera Inc. (2001), "Altera Digital Library Databook 2001".
28. Montana, D., Popp, R., Iyer, S. and Vidaver, G. (1998), "EvolvaWare: Genetic Programming for Optimal Design of Hardware-Based Algorithms", Genetic Programming 1998, pp. 869.
29. Luger, G., Stubblefield, W. (1998), "Artificial Intelligence: Structures and Strategies for Complex Problem Solving", Addison-Wesley, 3rd edition.
30. Zebulum, R. (1999), "Síntese de Circuitos Eletrônicos por Computação Evolutiva", Tese de Doutorado, PUC-Rio, Rio de Janeiro.