ELGEN: FERRAMENTA PARA GERAÇÃO DE CIRCUITOS COMBINATÓRIOS E SEQUENCIAIS PARA BENCHMARK

LEANDRO MAIA SILVA

ELGEN: FERRAMENTA PARA GERAÇÃO DE CIRCUITOS COMBINATÓRIOS E SEQUENCIAIS PARA BENCHMARK

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Antônio Otávio Fernandes

Belo Horizonte

Março de 2011

Silva, Leandro Maia
S586e Elgen: ferramenta para geração de circuitos combinatórios e sequenciais para Benchmark / Leandro Maia Silva. — Belo Horizonte, 2011
xxii, 75 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

Orientador: Antônio Otávio Fernandes

1. Computação - Teses. 2. Circuitos integrados - Teses. I. Orientador. II. Título.

CDU 519.6*17(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

elGen: ferramenta para geração de circuitos combinatórios e sequenciais para benchmark

LEANDRO MAIA SILVA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. ANTÔNIO OTÁVIO FERNANDES - Orientador
Departamento de Ciência da Computação - UFMG

PROF. JOSÉ AUGUSTO MIRANDA NACIF
Departamento de Informática - UFV

PROF. ROMANELLI LODRON ZUIM
Departamento de Ciência da Computação-PUC

PROF. SÉRGIO VALE AGUIAR CAMPOS
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 29 de março de 2011.

Dedico este trabalho a minha mãe que, com seu amor incondicional, nunca me deixou perder o foco ou desviar dos bons caminhos.

Agradecimentos

- A Deus pelo sopro da vida.
- A minha mãe Rosa por tudo. Não há espaço suficiente nessa dissertação para agradecê-la por tudo que ela merece.
- Ao Prof. Dr. Antônio Otávio Fernandes, orientador desta dissertação, por todo empenho, sabedoria, compreensão e por indicar as melhores escolhas quando parecia não haver opções.
- Ao Prof. Dr. Fabrício Vivas Andrade, pela amizade e pela oportunidade de trabalharmos juntos em sua pesquisa, essenciais para a existência deste trabalho.
- Aos meus amigos do LECOM, pela paciência que vocês têm comigo e pela força que vocês me dão todos os dias que trabalhamos juntos.
- A minha irmã Thaís, pela força na revisão e correção do texto.
- Aos membros da banca, por terem aceitado prontamente ao convite para participar dessa importante etapa da minha vida.
- Aos meus familiares, por aceitarem as desculpas quando faltei aos encontros enquanto trabalhava neste projeto.
- A todas as pessoas que contribuiram direta, ou indiretamente, para o desenvolvimento desta pesquisa.

Resumo

Conjuntos de circuitos para *benchmark* são mecanismos muito importantes como guias na seleção de ferramentas em Automação de Projetos Eletrônicos (EDA). Dada a vasta diversidade de estudos no campo da automação de projetos eletrônicos, e grande variedade de ferramentas comerciais em cada área, há uma crescente necessidade de novos conjuntos de circuitos para serem utilizados como *benchmarks*. Ainda, como a escala de integração dos circuitos tem crescido rapidamente com o passar dos anos, os *benchmarks* tornam-se rapidamente inadequados/obsoletos.

Existem basicamente duas principais classificações para os conjuntos de circuitos para *benchmark* em EDA, de acordo com o tipo do projeto: Circuitos de projetos industriais, e circuitos de projetos sintéticos. Ambas as classes, isoladamente, não são suficientes para suprir a demanda de *benchmarks*. Uma abordagem alternativa consiste na mistura entre as duas classes, resultando na geração de circuitos sintéticos baseados em projetos frequentemente utilizados nas indústrias de semicondutores.

Neste trabalho, propomos a definição e construção de uma ferramenta de software para geração de circuitos lógico-aritméticos de *hardware* baseados em modelos e utilizados como conjuntos de circuitos para *benchmarks* de ferramentas em EDA. Para geração dos circuitos, é utilizada a abordagem mista entre as classes de circuitos baseados em projetos industriais e sintéticos.

Palavras-chave: *Benchmark*, gerador de circuitos, síntese lógica, linguagem, Automação de Projetos Eletrônicos.

Abstract

Sets of benchmark circuits are very important mechanisms as guides in the selection of tools in electronic design automation (EDA). Given the wide diversity of studies in the field of electronic design automation, and a wide variety of commercial tools in each area, there is an increasing need for additional circuitry to be used as benchmarks. Still, as the scale of integration of circuits has grown rapidly over the years, benchmarks quickly become inadequate/obsolete.

There are basically two main sets of ratings for the benchmark circuits in EDA, according to the type of project: Circuits of industrial projects and projects of synthetic circuits. Both classes alone are not sufficient to supply demand of benchmarks. An alternative approach is to mix between the two classes, resulting in the generation of synthetic circuits based designs often used in semiconductor industries.

We describe the definition and construction of a software tool for generating logic-arithmetic hardware circuits based in models and used as benchmarks for circuitry in EDA tools. In order to generate the circuit, is used the mixed approach between the classes of circuits based on synthetic and industrial projects.

Keywords: benchmark, circuit generator, logic synthesis, language, Eletronic Design Automation.

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Sumário

Capítulo 1

Introdução

1.1 Objetivos e contextualização

Esta dissertação de mestrado trata da definição e construção de uma ferramenta de *software* para geração de circuitos lógico-aritméticos de *hardware* baseados em modelos e utilizados como conjuntos de circuitos para *benchmarks* de ferramentas de Automação de Projetos Eletrônicos (EDA, *Electronic Design Automation*). O seu objetivo é permitir que seja possível ao usuário construir o modelo de um circuito, gerar um circuito como uma instância específica dele, visualizar o circuito gerado e sintetizá-lo.

O modelo de um circuito pode ser entendido como um molde ou um conjunto de regras utilizados para a construção do circuito. Por se tratar de um *software*, o circuito não será construído efetivamente, mas descrito através de uma Linguagem de Descrição de *Hardware* (HDL, *Hardware Description Language*). Uma instância de um modelo corresponde a descrição do circuito gerado a partir dele, ou seja, o circuito representado através de uma HDL.

Benchmarking é o processo de comparar o desempenho de uma ou mais entidades sob um conjunto de aspectos utilizando como base algumas referências. *Benchmark* consiste, portanto, em descrever quais serão as referências utilizadas durante o processo de comparação.

Conjuntos de circuitos para *benchmark* são mecanismos muito importantes como guias na seleção de ferramentas em EDA. Dada a vasta diversidade de estudos no campo da automação de projetos eletrônicos, e grande variedade de ferramentas comerciais em cada área, há uma crescente necessidade de novos conjuntos de circuitos para serem utilizados como *benchmarks*. Ainda, como a escala de integração dos circuitos tem crescido rapidamente com o passar dos anos, os *benchmarks* tornam-se rapidamente inadequados [Andrade et al., 2008].

Um conjunto de circuitos em EDA é uma coleção de circuitos em um formato comum, os quais podem ser utilizados para avaliar algoritmos e ferramentas em um determinado domínio de um problema [III, 2000]. Em EDA, os mais comuns domínios de problemas que utilizam conjuntos de circuitos como *benchmarks* são: verificação formal (verificação de equivalência combinatória e sequencial, e *Model Checking*), simulação de falhas, síntese lógica, mapeamento tecnológico e análise de testabilidade [Andrade et al., 2008], heurísticas de otimização em Diagrama de Decisão Binária (BDD, *Binary Decision Diagram*) e Geração Automática de Padrões de Teste (ATPG, *Automatic Test Pattern Generation*) [Grout, 2001].

Existem basicamente duas principais classificações para os conjuntos de circuitos para *benchmark* em EDA, de acordo com o tipo do projeto: circuitos de projetos industriais, e circuitos de projetos sintéticos. Os primeiros têm sido a mais popular escolha nas últimas duas décadas de várias áreas em EDA, apesar disso, ainda se mostram inadequados sob três principais pontos de vista: o primeiro e mais importante está relacionado às indústrias de circuitos integrados, que não disponibilizam projetos atualizados devido ao seu alto valor de Propriedade Intelectual agregado. O segundo refere-se ao limitado número de conjuntos de circuitos publicados, justificado pelo mesmo motivo do aspecto anterior. O terceiro relaciona-se ao fato de que apenas circuitos de tamanhos e formatos específicos são disponibilizados. Notadamente, apesar de importantes, os circuitos de projetos industriais não são suficientes para suprir a vasta demanda de circuitos para serem utilizados como *benchmarks* para ferramentas em EDA [Andrade et al., 2008].

A classe formada por circuitos de projetos sintéticos surge como alternativa para suplantar as limitações dos circuitos de projetos industriais. Um conjunto de circuitos sintéticos para *benchmark* é gerado por uma ferramenta dada certas características desejadas. Apesar de essa abordagem ser adequada em muitas situações, algumas vezes é vantajoso saber exatamente qual é a função do circuito utilizado como *benchmark*[Kunes & Danek, 2003]. Outra desvantagem dessa classe trata da parte dos conjuntos de circuitos sintéticos para *benchmark* gerada aleatoriamente, culminando em circuitos muito distantes daqueles realmente aplicados nos projetos industriais, portanto, inadequados para avaliação de muitas ferramentas.

Uma abordagem alternativa consiste na mistura entre as duas classes, resultando na geração de circuitos sintéticos baseados em projetos frequentemente utilizados nas indústrias de semicondutores [Bakalis et al., 2001] [Homma et al., 2006]. O maior benefício oferecido por este tipo de abordagem é que os circuitos criados não são tão artificiais como aqueles gerados aleatoriamente, nem tão específicos e limitados como aqueles fornecidos pela classe de projetos industriais para *benchmark*. A ferramenta

apresentada neste documento, entitulada *elGen*, utiliza essa abordagem para geração dos seus circuitos.

Um dos fatores que limita a utilização desses conjuntos como *benchmark* é a limitação de tamanho dos circuitos que os compõem, conforme visto anteriormente. Alguns circuitos possuem comportamento similar, diferenciando-se apenas quanto ao número de *bits* sobre os quais operam. Um bom exemplo seria dois circuitos aritméticos que realizam a função de multiplicar, um recebe como entrada dois números inteiros de oito *bits* e gera como resultado um inteiro de dezesseis *bits*, e outro recebe como entrada dois números inteiros de nove *bits* e gera como resultado um inteiro de dezoito *bits*. A diferença entre os dois circuitos é apenas o tamanho das entradas e das saídas. Apesar de a diferença, de modo geral, ser apenas um *bit* entre as entradas e dois *bits* entre as saídas, internamente esses circuitos podem chegar a ter diferença de tamanho de duas vezes, dependendo do algoritmo de multiplicação utilizado. Essa simples, mas essencial diferença pode implicar na seleção de uma ferramenta em EDA em detrimento a outra, caso uma delas não seja capaz de realizar a tarefa sobre o circuito maior, por exemplo.

Se o comportamento de dois circuitos é essencialmente o mesmo, diferindo apenas quanto à quantidade de *bits* sobre os quais são realizadas as operações, então existe uma descrição comportamental do circuito que independe de seu tamanho, e essa descrição pode ser representada através de um modelo. Neste documento um modelo de circuito corresponde à descrição das regras de ligação entre seus componentes, independentemente de seu tamanho. Isso permite a geração de um circuito de tamanho específico apenas instanciando o modelo com o tamanho desejado. Observe que a descrição do modelo poderia ser feita em diversos níveis de abstração, mas o nível aceito pela ferramenta desenvolvida no presente trabalho está muito próximo do RTL (*Register Transfer Level*) (ver capítulo 3).

Várias HDLs permitem que sejam criados módulos parametrizáveis, similares ao conceito de modelo descrito anteriormente. Dois exemplos vastamente difundidos na literatura são o *generic*, do VHDL [vhd, 1994], e o *parameter*, do Verilog HDL [ver, 2006]. Infelizmente, nem todos os formatos de descrição de circuitos dispõe desse recurso, a exemplo do *BENCH* [F.Brglez & Fujiwara, 1985]. Em uma situação hipotética, se quiséssemos dez circuitos multiplicadores de tamanhos diferentes para utilizarmos como *benchmark*, todos descritos em *BENCH*, teríamos que escrever dez arquivos diferentes, ou utilizar *scripts* para automatizar essa tarefa. Neste trabalho apresentamos a ferramenta *elGen*, que utiliza a linguagem *elLen*, a qual permite a descrição desses modelos. Trata-se de uma linguagem bastante simples que será detalhada no capítulo 2.

Uma importante consequência da grande variedade de ferramentas em EDA existentes, em conjunto com a falta de padronização entre elas, é a diversidade de formatos de entradas existentes. Apesar de os formatos de descrição dos circuitos utilizados por essas ferramentas serem padronizados, nem todas as ferramentas aceitam o mesmo subconjunto de formatos. Esse fato vai exatamente ao encontro do mesmo problema apresentado quando discutido a respeito dos módulos parametrizáveis, pois temos uma quantidade grande de arquivos diferentes descrevendo o mesmo circuito. O fato de duas ferramentas em EDA não possuirem um formato de entrada comum implica na descrição de um mesmo circuito, no mínimo, duas vezes. Observe que esse problema tem crescimento vertiginoso quando trabalhamos com um grande número de ferramentas, circuitos e/ou formatos diferentes. A linguagem *elLen* permite a ferramenta gerar circuitos em tamanhos e formatos diversos utilizando uma única modelagem.

A figura 1.1 ilustra a utilização da *elLen*, em conjunto com a ferramenta *elGen*, resultando em um circuito descrito em quatro HDLs diferentes, entradas de três hipotéticas ferramentas em EDA. Observe que para o exemplo, caso o circuito não tivesse sido modelado utilizando *elLen*, seria necessária a descrição do mesmo, no mínimo, duas vezes.

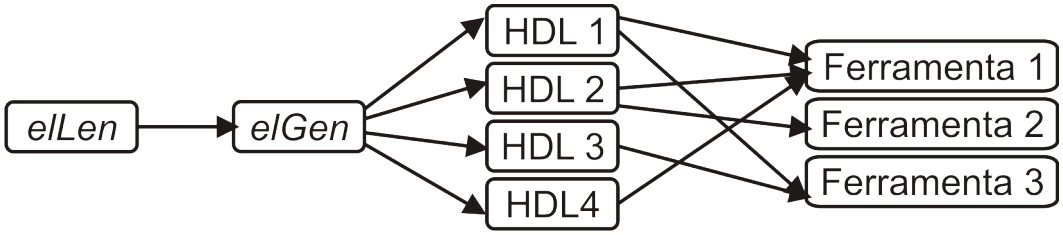

Figura 1.1. *elLen* + *elGen* produzindo circuitos para ferramentas em EDA

Quando discutido as duas principais classes nos quais os *benchmarks* em EDA podem ser classificados, foi mencionado que algumas vezes é vantajoso saber exatamente qual é função desempenhada pelo circuito, mais do que isso, algumas vezes é desejado que o circuito gerado seja factível de ser transformado em um circuito físico. O processo de transformação da descrição comportamental do circuito em sua implementação em termos de portas lógicas é chamado de síntese, mas nem toda descrição de um circuito pode ser considerada sintetizável. Um dos principais fatores que impede o processo de síntese é o uso de elementos de descrição que não fazem parte do subconjunto sintetizável da HDL [Group, 1996] [Gordon, 1997]. Apesar de algumas ferramentas em EDA aceitarem como entrada circuitos descritos utilizando um superconjunto do conjunto sintetizável, esses circuitos não podem efetivamente se tornar circuitos reais. Como os conjuntos de circuitos de projetos industriais são sintetizáveis,

por construção, é necessário que a abordagem de gerar conjuntos de circuitos sintéticos baseados em projetos industriais também gere somente circuitos sintetizáveis. Esse importante fator é considerado pois a ferramenta proposta só gera circuitos utilizando o respectivo subconjunto sintetizável das HDLs.

As ferramentas em EDA utilizam como entrada, em sua maioria, descrições textuais dos circuitos, que, apesar de funcionais, tornam enfadonha a tarefa de analisar as ligações entre os componentes dos circuitos quando feita por seres humanos. A *elGen* oferece um visualizador simples dos circuitos gerados utilizando o mesmo número de níveis de detalhamento usado durante a definição do modelo do circuito. Assim, se o circuito foi definido, por exemplo, com três níveis de modularização, será possível visualizar o circuito e suas conexões em cada um dos três níveis.

Uma vez apresentados os principais pontos abordados durante o processo de definição e construção da ferramenta, será realizada uma compilação dos trabalhos existentes relacionados à geração de circuitos para *benchmark*. O capítulo 1 aborda ainda a motivação para o desenvolvimento do presente trabalho, bem como a estrutura geral da ferramenta e breve resumo dos capítulos subsequentes.

1.2 Trabalhos relacionados

Existem basicamente duas principais classificações para os conjuntos de circuitos para *benchmark* em EDA: Os baseados em projetos industriais e os baseados em projetos sintéticos.

Os *benchmarks* do primeiro grupo tiveram seu auge de utilização nas décadas de 80 e 90, e seus principais representantes são ISCAS 85 [F.Brglez & Fujiwara, 1985] e ISCAS 89 [Brglez et al., 1989] (ISCAS, *IEEE International Symposium on Circuits and Systems*). Apenas dez circuitos combinatórios compunham o pacote de circuitos para *benchmark* do ISCAS 85 (c432 *: 27-channel interrupt controller;* c499/c1355 *: 32-bit SEC circuit;* c880 *: 8-bit ALU;* c1908 *: 16-bit SEC/DED circuit;* c2670 *: 12-bit ALU and controller;* c3540 *: 8-bit ALU;* c5315 *: 9-bit ALU;* c6288 *: 16x16 multiplier;* c7552 *: 32-bit adder/comparator*). Dentre eles, o menor circuito apresenta 160 portas lógicas (c432) e o maior 3.512 (c7552), sendo que dos dez circuitos, quatro deles não possuem nem mil portas lógicas. Observe que uma unidade lógico-aritmética de 9 *bits*, ou um multiplicador 16x16, são circuitos bastante defasados. O pacote de circuitos do ISCAS 89 veio com objetivo de estender o ISCAS 85 em tamanho e complexidade, acrescentando 31 circuitos sequenciais, todos sincronizados por um *clock* e utilizando como elementos de memória apenas *flip-flops D*. O menor dos circuitos é formado por

dez portas lógicas e três *flip-flops D* (circuito s27) e o maior por 22.179 portas lógicas e 1.636 *flip-flops D* (circuito s38417). Irrefutavelmente, ISCAS 89 cumpriu seu objetivo e, junto com ISCAS 85, foi bastante utilizado durante as duas décadas. Porém, hoje, comparados às ferramentas em EDA atuais, são considerados obsoletos.

Com objetivo de suprir as limitações apresentadas pelos *benchmarks* do ISCAS, surge em 1999 outro conjunto de *benchmarks*, denominado ITC 99 (ITC, *International Test Conference*) [Davidson, 1999]. Esse conjunto é composto por 31 circuitos, divididos em quatro grupos. O primeiro grupo consiste de cinco circuitos baseados em projetos de Circuitos Integrados de Aplicação Específica (ASIC, *Application-Specific Integrated Circuit*) e outros Circuitos Integrados (CIs). O segundo é composto por versões RTL de quatro circuitos do ISCAS (s208, s289, s344 e s349), criados pela Universidade de Michigan. O terceiro grupo, formado por um único circuito, foi extraído de um projeto industrial e tornou-se conhecido por causar problemas para ATPGs industriais, sendo, portanto, considerado interessante para pertencer ao conjunto. O quarto e mais extenso grupo é formado por 22 circuitos e ficou conhecido como *benchmark* de Politecnico di Torino.

O *benchmark* de Politecnico di Torino possui seus circuitos sequenciais descritos em RTL, variando de 45 portas lógicas e 5 *flip-flops* (b01) até 98.726 portas lógicas e 6.618 *flip-flops* (b19). Alguns dos maiores circuitos são composições de vários outros menores, todos conectados. Todos são considerados simples por serem sincronizados por um único *clock*, não possuírem memória interna, nem componente do tipo *tri-states*. Esse conjunto é considerado bastante útil para testes de algoritmos de ATPG em grandes projetos. A principal contribuição deste *benchmark* está no aumento em tamanho e complexidade se comparado ao ISCAS. No entanto, até o momento de sua publicação (Março de 1999), vários de seus circuitos não haviam sido validados, portanto, sem garantias de corretude.

Uma nova geração de *benchmarks* do ISCAS foi proposta em 2004 e é conhecida como Miroslav Velev's SAT Benchmarks ou Velev Benchmarks [Velev, 2004]. O conjunto é formado por 28 circuitos derivados de projetos de processadores. O maior circuito disponível nesse *benchmark* possui mais de 700.000 portas lógicas, ultrapassando sete vezes o tamanho do maior circuito disponível no *benchmark* de Politecnico di Torino. O foco principal desse *benchmark* é a utilização em verificação formal, mais precisamente em resolvedores de problemas de satisfabilidade booleana (SAT, *Satisfiability*, simplificação de *Boolean satisfiability Problem*). Apesar de cumprir bem o seu papel para a área de SAT, esse *benchmark* deixa a desejar quanto ao formato de distribuição do mesmo, disponível quase na totalidade em forma normal conjuntiva (CNF, *Conjunctive Normal Form*), de pouca utilidade em áreas como verificação de

equivalência.

Assim como o *benchmark* de Politecnico di Torino, vários conjuntos de circuitos utilizados como *benchmarks* são fornecidos por sítios localizados na Internet. Dois grandes representantes são *OpenCores*[openCores.org, 2005] e *Free Model Foundry* [freemodelfoundry.com, 1995]. Esses sítios representam comunidades de projetistas de CIs e disponibilizam uma coleção de núcleos digitais gratuitamente, os quais são submetidos por usuários registrados, e avaliados/verificados por usuários da própria comunidade. Apesar do esforço na avaliação dos núcleos, a maioria deles não é verificada através de um processo formal, e sim por meio do fluxo de utilização, notificação e correção dos erros. Consequentemente, não há garantias de os núcleos estarem livres de erros, nem que realmente correspondem a especificação do núcleo.

Com objetivo de avaliar os processos de síntese, otimização e verificação de circuitos integrados, surge um novo *benchmark* conhecido como *International Workshop on Logic and Synthesis* 2005 (IWLS 2005) [Albrecht, 2005]. Apresentado durante o referido *workshop* e mantido até a presente data, o IWLS 2005 é composto de 84 projetos, com 185.000 registradores e 900.000 portas lógicas, coletados de diferentes sítios na Internet. Esse *benchmark* é composto por 26 projetos do *OpenCores*, quatro do grupo de pesquisa da Aeroflex Gaisler AB [AB, 2005], três da Faraday Technology Corporation [Corporation, 2005], 21 do *benchmark* ITC 99 e 30 dos *benchmarks* do ISCAS. Apesar de possuir o maior número de circuitos disponíveis até o ano de seu surgimento, o IWLS 2005 possui utilização limitada devido à grande quantidade de circuitos providos pelo *OpenCores*, não tendo se tornado tão popular como os demais citados.

Um método diferente para geração de *benchmarks* baseados em projetos industriais é proposta em Kunes & Danek [2003]. O método consiste na transformação de um projeto existente em outro, realizado através do mapeamento de elementos em outros equivalentes. Por exemplo, a substituição de uma porta lógica por um multiplexador.

Embora os *benchmarks* baseados em projetos industriais tenham sido a principal escolha de muitas áreas em EDA, eles sozinhos não são suficientes para atender a demanda crescente por novos circuitos. Assim os *benchmarks* sintéticos surgem como forma de suprir suas limitações.

Uma das primeiras utilizações satisfatórias de *benchmarks* sintéticos foi na área de roteamento de FPGAs (*Field Programmable Gate Array*)[Darnauer & Dai, 1996]. Posteriormente, outros *benchmarks* sintéticos foram propostos por autores como Hutton et al. [2002], Pistorius et al. [1999] e Verplaetse et al. [2002].

O grande problema dos *benchmarks* sintéticos está no fato de que, em sua maioria, são gerados de maneira aleatória, ou atendendo a casos bastante específicos. Essa

abordagem leva a criação de circuitos muito distantes daqueles efetivamente utilizados nas indústrias de semicondutores. A utilização desses circuitos gerados para avaliação de ferramentas e/ou algoritmos pode levar a resultados pouco confiáveis ou inaplicáveis em circuitos reais, uma vez que as características avaliadas não estão presentes em circuitos de fato.

Uma alternativa para geração de *benchmarks* consiste na utilização de *softwares* geradores de núcleos funcionais, ou seja, circuitos sintéticos baseados em projetos industriais. Esses geram descrições de circuitos com funções específicas (somadores, multiplicadores etc), que então são utilizados como *benchmarks*. A principal vantagem desse tipo de geração está no extenso número de circuitos disponíveis. Dois importantes representantes são Arithmetic Module Generator [Homma et al., 2006] e Eudoxus [Bakalis et al., 2001]. O primeiro oferece mais de 20 tipos de circuitos diferentes, sendo a maior parte constituída de somadores. O segundo fornece 32 tipos de circuitos, divididos em quatro categorias, cuja principal possui 19 tipos de somadores e multiplicadores paralelos. Uma grave deficiência apresentada nesses *softwares* relaciona-se a indisponibilidade de alguns núcleos básicos, como comparadores e paridade. Outra deficiência, presente também em vários *benchmarks*, refere-se ao fato de que os circuitos são disponibilizados em poucos formatos - ambos geradores fornecem os circuitos apenas em Verilog HDL e VHDL. E por fim, os gerados mencionados possuem foco em reuso de seus núcleos, ou seja, a geração de *benchmarks* não representa seu objetivo principal.

O estado da arte em geradores de núcleos funcionais utiliza a mesma abordagem das duas últimas ferramentas apresentadas, mas seu foco é a geração de circuitos para *benchmarks*. A ferramenta BenCGen [Andrade et al., 2008] [Andrade, 2008] é a mais completa em número e tipos de circuitos gerados e também em formatos nos quais esses circuitos são disponibilizados. A maior deficiência da ferramenta é a ausência de circuitos sequenciais na lista de tipos gerados, sendo disponibilizados apenas circuitos combinatórios.

1.3 Motivação

Problemas em EDA estão cada dia mais complexos, principalmente devido ao rápido aumento da escala de integração dos CIs. Como consequência desse crescimento em complexidade, as técnicas utilizadas para resolver os problemas em EDA tornam-se rapidamente obsoletas. Dessa forma, novas técnicas precisam ser pesquisadas e testadas. Para que essas novas técnicas passem por avaliação, são necessários novos conjuntos de circuitos para *benchmark*.

O esforço para melhoria e ampliação destes deve ser uma tarefa contínua, uma vez que a demanda por novos circuitos, maiores e mais complexos, tem crescido com o passar dos anos sem perspectiva de declínio, mas, como vimos, nem *benchmarks* baseados em projetos industriais, nem os sintéticos, suprem sozinhos a demanda.

A principal motivação para a construção da ferramenta *elGen* é possibilitar que uma quantidade quase ilimitada de circuitos, os quais podem ser gerados em diferentes tamanhos e formatos a fim de sua utilização como *benchmarks* de ferramentas em EDA. A definição de uma linguagem para descrição dos modelos dos circuitos a serem gerados faz com que a ferramenta seja expansível sem a necessidade de qualquer modificação em seu código-fonte.

Outra motivação é tornar disponível para o meio acadêmico a ferramenta de forma que cada usuário possa adaptá-la ao seu uso diário, permitindo que ele mesmo modele os circuitos que necessite.

Por fim, a ferramenta *elGen* objetiva gerar somente circuitos sintetizáveis, uma vez que são implementados algoritmos de síntese de circuitos combinatórios e sequenciais pela ferramenta. Todo circuito criado por *elGen* é validado através desses algoritmos de síntese e, portanto, são gerados apenas circuitos sintetizáveis.

1.4 Estrutura da ferramenta

Nesta seção será descrita de maneira simplificada a estrutura da ferramenta *elGen*. A figura 1.2 representa a estrutura da ferramenta dividida em três grupos básicos: O corpo, suas entradas e suas saídas. As entradas são representadas por três blocos cujas setas estão na direção do corpo da ferramenta, e as saídas, de forma semelhante, apresentam três blocos com as setas saindo da ferramenta.

A primeira entrada esperada pela ferramenta é a descrição do modelo do circuito, utilizada como base para geração do circuito desejado. O modelo deve ser descrito através da linguagem *elLen*, criada para descrição dos modelos aceitos pela ferramenta aqui apresentada. A segunda entrada identifica o tamanho do circuito que será gerado. Conforme será visto no capítulo 2, o circuito é gerado a partir da descrição do modelo e do tamanho desejado. Por último, a terceira entrada seleciona o formato no qual o circuito será gerado. Os formatos disponíveis são BENCH, Verilog HDL, VHDL, BLIF [Berkeley, 2005] e EQN.

O corpo da ferramenta pode ser dividido em três grandes níveis. No primeiro tem-se o processador da linguagem *elLen*, cuja função é receber o modelo do circuito e construir a representação dos módulos que o compõe, deixando-os prontos para serem

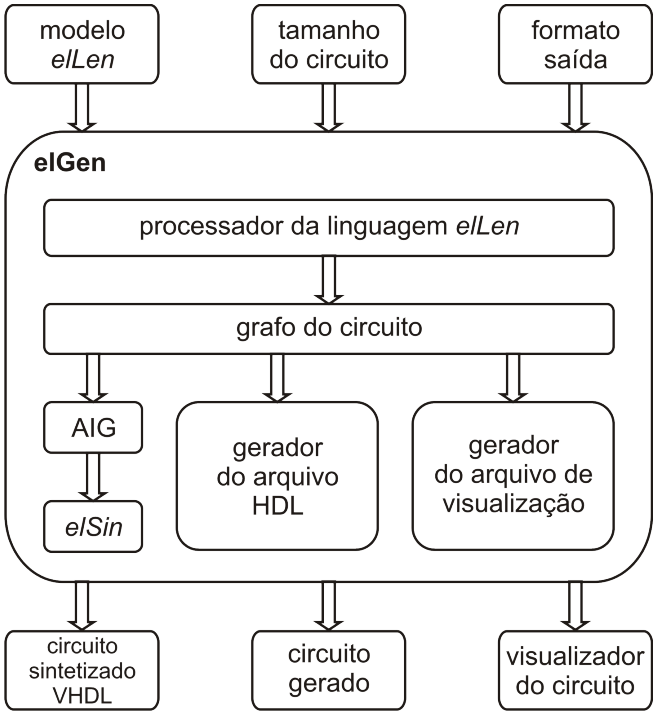

Figura 1.2. Estrutura da ferramenta elGen

processados. O segundo nível corresponde à representação do circuito e recebe os módulos processados no nível anterior. Soma-se a ele o tamanho estabelecido para gerar a representação correspondente ao circuito. O terceiro e maior nível, engloba todo o processamento da representação do circuito para executar as três tarefas da ferramenta: síntese, geração e visualização do circuito. A síntese é realizada através da transformação da representação em seu respectivo AIG (*And-Inverter Graph*), que é a entrada para o algoritmo de síntese dessa ferramenta, denominado *elSim* (ver capítulo 3). A geração do circuito consiste na descrição do circuito através da HDL desejada. E a visualização acontece por meio do processamento de um arquivo de texto simples com todos os componentes do circuito gerado.

Finalmente, a ferramenta também gera três saídas para o usuário; são eles circuito sintetizado VHDL [1], circuito gerado e visualização do circuito. A primeira saída corresponde ao circuito resultante do processo de síntese totalmente descrito em VHDL e contendo apenas portas lógicas AND e NOT, além de todos os elementos de memória serem representados por uma transição sincronizada por um *clock*. A segunda saída fornece a descrição do circuito na linguagem desejada. Já o visualizador do circuito, última saída, é uma interface gráfica bastante simples que permite ao usuário visualizar

[1] A escolha de VHDL como linguagem de saída do circuito sintetizado deve-se a sua grande aceitação como entrada por várias ferramentas.

o circuito gerado. Em geral, é mais simples entender as ligações do circuito gráfica do que textualmente.

É importante ressaltar que o usuário da *elGen* não precisará compreender o funcionamento interno da ferramenta, cabendo-lhe apenas fornecer as entradas adequadas e utilizar as saídas entregues.

1.5 Organização da dissertação

Os demais conteúdos desta dissertação foram organizados de modo a facilitar a leitura, a busca e o entendimento por parte do leitor. O capítulo 2 apresenta a linguagem *elLen*, que é utilizada para descrição dos modelos dos circuitos, e também como uma das entradas da ferramenta. O capítulo 3 apresenta o algoritmo de síntese *elSin*, utilizado para sintetizar e validar os circuitos gerados a partir do modelo descrito com *elLen*. O capítulo 4 descreve a ferramenta do presente trabalho, a *elGen*, e detalha aspectos como a entrada do modelo do circuito, seu processamento e sua representação; além do processo de síntese, geração e visualização dos circuito efetivamente criado. No mesmo capítulo é realizada uma comparação qualitativa com as ferramentas e conjuntos de circuitos para *benchmarks*. Em conclusão, o capítulo 5 apresenta as considerações finais deste trabalho e algumas orientações para guiar sua extensão em trabalhos futuros.

Capítulo 2

A linguagem elLen

2.1 Introdução

A utilização de uma HDL para descrever o circuito a ser produzido é muito importante por vários aspectos. Dentre os quais podemos citar a facilidade para simulação do circuito que, em muitos casos, reduz a quantidade de falhas existentes; a possibilidade de prototipação rápida, permitindo assim testes mais próximos do circuito final; além da redução considerável do custo de produção, uma vez que, em geral, o valor da confecção do circuito efetivo tende a ser mais elevado.

Diversas linguagens bastante eficientes para descrever circuitos combinatórios e sequenciais já foram propostas. Entre elas, podemos citar Verilog HDL, VHDL, BLIF, BENCH. Além de sua importância, as HDLs oferecem várias vantagens, tais como a grande quantidade de ferramentas em EDA que aceitam como entrada circuitos descritos através delas, e a vasta documentação existente. Apesar de as linguagens citadas possuírem uma descrição formal e algumas delas serem regulamentadas por conselhos, as ferramentas em EDA não possuem qualquer tipo de padronização quanto ao subconjunto de linguagens por elas utilizadas. Essa falta de padrão acarreta alguns problemas graves para os seus usuários.

Um grande problema é o número de vezes que um mesmo circuito precisa ser descrito caso não haja uma linguagem aceita na interseção do conjunto de linguagens utilizadas pelas ferramentas. Além de gerar trabalho que poderia ser evitado, a questão pode resultar em descrições diferentes do circuito, uma vez que elementos presentes em uma linguagem não necessariamente existem em outras, e sua utilização implica na tradução destes elementos em outros equivalentes. Outro problema está relacionado à falha humana, pois aumentar o número de vezes que um circuito é descrito implica em aumentar também a possibilidade de erro na descrição do mesmo.

Outro grave problema está diretamente relacionado ao primeiro e diz respeito à qualificação profissional. A grande quantidade de HDLs implica em tempo de aprendizado para cada uma delas. E por fim temos a impossibilidade, para alguns casos, de transformar um circuito descrito inicialmente em determinada linguagem em outra diferente, sem perda de informação. Por exemplo, um circuito descrito inicialmente em BENCH perde a informação de modularização dos componentes do circuito, e ao transforma-lo em Verilog HDL, torna-se impossível a recuperação da topologia dos módulos utilizados.

A partir disso, este capitulo apresenta a linguagem *elLen* cujo objetivo é possibilitar ao usuário a descrição de um modelo que possa criar circuitos em diversos tamanhos e formatos, reduzindo assim a necessidade de descrever circuitos similares várias vezes e em diferentes linguagens. É importante ressaltar que a linguagem tem como princípio fornecer um mecanismo bastante simplificado para descrição de modelos, e não substituir as HDLs já existentes ou se tornar padrão de entrada das ferramentas em EDA. A linguagem *elLen* se diferencia das HDLs em dois aspectos: primeiramente sua criação destina-se a descrição do modelo, e não do circuito; e em segundo lugar, trata-se do meio de entrada da ferramenta apresentada neste trabalho, a *elGen*.

2.2 Fundamentação teórica

Nesta seção será apresentada uma compilação de vários conceitos acerca de grafos e linguagens formais, todos necessários para o completo entendimento deste trabalho. Todas as informações contidas nesta seção foram extraídas dos livros *Introdução aos Fundamentos da Computação*[Vieira, 2006] e *Graph Teory*[Agnarsson & Greenlaw, 2006].

Em sua obra, Vieira (2006) apresenta um estudo geral, sobre em máquinas de estados finitos, autômatos de pilha e máquinas de Turing, para tratar o problema da decidibilidade em computação. Conjuntos, grafos, indução matemática, entre outros, também são apresentados através de conceitos. Enquanto Geir (2006) traz em seu livro um estudo bastante completo a respeito da teoria de grafos, abordando desde a modelagem e aplicação, até os algoritmos utilizados para resolver diversos problemas e várias aplicações em grafos.

2.2.1 Fundamentos de grafos

Nesta subseção será apresentada uma série de conceitos básicos sobre teoria dos grafos, que será utilizada como fundamentação teórica para os capítulos restantes.

Teoria dos grafos é um ramo da matemática cujo objetivo é estudar a estrutura utilizada para modelar relações entre pares de objetos de certa coleção, denominada grafo. Um grafo no contexto deste documento refere-se a uma coleção de vértices (também chamado de nodos) e arestas que conectam um par de vértices.

Um grafo G é um par (V,A), sendo V um conjunto de vértices e A um conjunto de arestas. Um grafo é dito dirigido se A é formado por pares ordenados de vértices pertencentes a V. Um grafo é dito não dirigido se A é formado por pares não ordenados de vértices pertencentes a V.

Na representação gráfica de um grafo utilizada nesta subseção, o vértice será representado por um círculo; a aresta, por uma linha unindo as representações de dois vértices, *origem* e *destino*. No caso de um grafo dirigido, existirá uma seta na extremidade da linha mais próxima ao vértice *destino*. As figuras 2.1 e 2.2 apresentam exemplos de representações gráficas de um grafo dirigido e não dirigido, respectivamente.

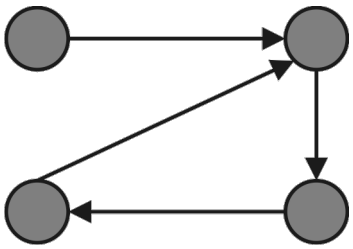

Figura 2.1. Exemplo de grafo dirigido

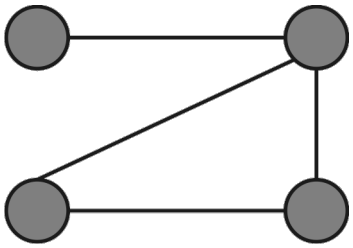

Figura 2.2. Exemplo de grafo não dirigido

Os dois tipos de grafos explicados anteriormente representam tipos básicos de grafos. Podem existir grafos mistos, que contêm ambos os tipos de arestas, e também grafos que contêm não apenas um conjunto de arestas mas um multiconjunto [1].

Um grafo é dito rotulado se existem rótulos associados a suas arestas e/ou vértices. Um grafo rotulado é uma tripla (V,A,R), sendo V um conjunto de vértices, $A \subseteq V$ x R x V é um conjunto de arestas rotuladas, e R é um conjunto de rótulos.

[1] Um multiconjunto é um conjunto que admite repetições de elementos. Por exemplo, {a} $\neq$ {a,a} $\neq$ {a,a,a}

A representação gráfica é muito similar a de um grafo não rotulado. A diferença é que, para cada aresta *(a,r,b)*, coloca-se o rótulo r adjacente à linha que representa a aresta *(a,b)* ou internamente ao vértice. As figuras 2.3 e 2.4 mostram exemplos de representações gráficas de um grafo dirigido rotulado e não dirigido rotulado, respectivamente.

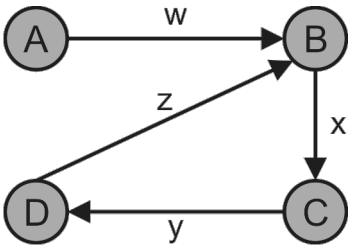

Figura 2.3. Exemplo de grafo dirigido rotulado

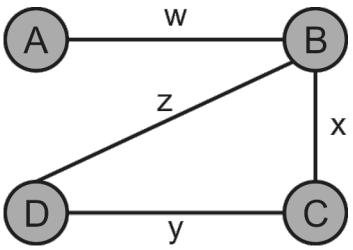

Figura 2.4. Exemplo de grafo não dirigido rotulado

Em um grafo dirigido, um *caminho* de tamanho n de um vértice a para um vértice b, é formado por uma sequência de vértices e arestas $v_0a_1v_1a_2v_2...v_{n-1}a_nv_n$, tal que $v_0 = a$, $v_n = b$, e a aresta a_i sai do vértice v_{i-1} e incide no vértice v_i. Em um grafo não dirigido, a representação do caminho pode ser realizada apenas através dos vértices, presumindo a existência das arestas entre os vértices. Essa observação é particularmente importante uma vez que, para os grafos dirigidos, a existência de um caminho de a para b não implica na existência de uma caminho de b para a. Um caminho é dito *caminho nulo* se $n = 0$, ou seja, se $v_0 = a = b$. Um *caminho fechado* é um caminho não nulo em que os vértices inicial e final são os mesmos, isto é, $v_0 = v_n$. Um *ciclo* é caminho fechado onde $v_i \neq v_j$ para todo $0 \leq i < j \leq n$, exceto para $i = 0$ e $j = n$, simplificadamente, é um caminho fechado sem vértices e arestas repetidos. Um *laço* é um ciclo de tamanho 1. Um *caminho simples* é um caminho sem vértices repetidos.

Um grafo é dito grafo *acíclico* se não existe qualquer ciclo no mesmo. Analogamente, um grafo é dito *cíclico* se existe um ou mais ciclos no mesmo. Um grafo *conexo* é um grafo não dirigido onde existe um caminho de v_i para v_j para todo $0 \leq i < j$

$\leq n$; e um grafo *fortemente conexo* é um grafo dirigido onde existe um caminho de v_i para v_j para todo $i \neq j$.

2.2.2 Fundamentos de linguagens formais

Uma linguagem é todo e qualquer sistema que serve de meio de comunicação de conceitos, idéias, significados, pensamentos etc. A linguagem natural é um conceito formulado pela filosofia da linguagem e pela linguística para se referir às linguagens desenvolvidas pelo ser humano como instrumento de comunicação. A linguagem formal, ao contrário das naturais, é apreendida por meio de um conjunto de regras gramaticais muito bem definido, ao qual é dado o nome sintaxe. O objetivo desta subseção é apresentar um conjunto de conceitos relacionados às linguagens formais.

Dois pré-requisitos são necessários para que haja uma linguagem formal: Sintaxe bem definida e semântica precisa. Pelo primeiro deles é possível saber se uma dada sentença pertence, ou não, a determinada linguagem; já pelo segundo, determina-se o correto entendimento da sentença, eliminando ambiguidades. No caso específico deste trabalho, trataremos apenas o que diz respeito à sintaxe em linguagens formais, uma vez que seu estudo é suficiente para a realização do projeto aqui apresentado. Para efeito de simplificação, o termo linguagem será utilizado como sinônimo de linguagem formal, a menos que dito o contrário.

Toda linguagem contém um *alfabeto* (será utilizado a letra grega sigma maiúsculo, Σ, para representá-lo) que pode ser definido como um conjunto finito não vazio de elementos chamados *símbolos*. Uma *palavra* em Σ é uma sequência finita de símbolos pertencentes a Σ. O tamanho de uma palavra p, $|p|$, é o número de símbolos que a compõem. Uma palavra de zero símbolo é chamada de *palavra vazia* e será representada neste documento por λ (*lambda*, letra grega minúscula), ou seja, $|\lambda| = 0$.

Seja s um símbolo qualquer, a notação s^n, em que $n \in \mathbb{N}$, será utilizada para representar uma palavra constituída de n símbolos s em sequência. Duas notações especiais são s^*, que representa uma sequência de 0 ou mais símbolos, e s^+, que representa 1 ou mais símbolos.

Uma linguagem L definida sob Σ é um conjunto de palavras em Σ. Sendo Σ^* o conjunto de todas as palavras possíveis de serem escritas em Σ, logo toda linguagem em Σ é um subconjunto de Σ^*.

Uma gramática G é definida como uma quádrupla (V,Σ,R,P), tal que V é um conjunto finito de elementos denominados *variáveis*; Σ é um *alfabeto*, sendo $V \cap \Sigma = \emptyset$; $R \subseteq (V \cup \Sigma)^+$ x $(V \cup \Sigma)^*$ é um conjunto finito de pares ordenados $(u \rightarrow v)$ chamados *regras*; e $P \in V$ é uma variável conhecida como *variável de partida*.

Uma linguagem *L(G)* constitui o conjunto de todas as sentenças [2] que podem ser geradas através da gramática. Assim, a gramática representa o formalismo da definição da linguagem. Uma sentença é dita pertencente à linguagem se, a partir da varíavel de partida, é possível aplicar sucessivas regras e chegar a uma sentença. Esse processo é conhecido como *derivação*.

A derivação consiste em substituir sistematicamente um conjunto de símbolos existentes em uma forma sentencial [3] por outro conjunto de símbolos, definidas pelas regras em *R*. O processo é demonstrado através do exemplo abaixo.

Seja $G_1(V_1,\Sigma_1,R_1,P_1)$, tal que $V_1 = \{I,A,B\}$; $\Sigma_1 = \{0,1\}$; $R_1 = \{$ 1:(I $\rightarrow$ A), 2:(A $\rightarrow$ 0A), 3:(A $\rightarrow$ B), 4:(B $\rightarrow$ 1B), 5:(B $\rightarrow \lambda$) $\}$ [4] ; e P_1 = I. A sentença 00 pertence a $L(G_1)$ pois aplicando as regras 1,2,2,3,5, nesta ordem, obtem-se a sentença. O mesmo vale para a sentença 01 aplicando as regras 1,3,4,5. A partir da definição de G_1, não é difícil demonstrar que $L(G_1) = 0^*1^*$.

2.3 A linguagem elLen

O objetivo da linguagem *elLen* é possibilitar ao usuário uma forma simples de descrever circuitos similares através de um modelo. A utilização de modelos reduz consideravelmente o tempo para geração de circuitos cujo comportamento é similar, variando apenas o número de *bits* sobre os quais operam.

Um exemplo bastante simples de circuito que pode ser facilmente modelado é o aritmético somador de dois números inteiros. A figura 2.5 representa graficamente dois circuitos chamados de *Ripple Carry Adder* (RCA). Simplificadamente, o RCA recebe dois números inteiros codificados em binário e realiza a soma dos algarismos partindo do menos significativo para o mais significativo, e a cada excesso, vai um para a próxima soma - semelhante a operação de somar de maneira similar ao aprendido durante as aulas de aritmética básica. A diferença entre os circuitos é simplesmente a quantidade de *bits*, mas o modelo de construção é exatamente o mesmo.

Cabe breve explicação sobre a diferença sutil entre descrever um modelo de um circuito e descrever um circuito efetivamente. Ainda utilizando como exemplo o somador, é possível descrever um modelo que some dois números de uma quantidade de *bits* arbitrária, uma vez que o procedimento para somar é sempre o mesmo, mas não é possível descrever um circuito que some dois números de uma quantidade de *bits*

[2] Sentença: Palavra constituída apenas de terminais, ou seja, sem variáveis.

[3] Forma sentencial: Palavra constituída de variáveis e/ou terminais

[4] Duas regras que possuem o mesmo lado esquerdo $u \rightarrow v$ e $u \rightarrow v'$ podem ser escritas simplificadamente por $u \rightarrow v \mid v'$

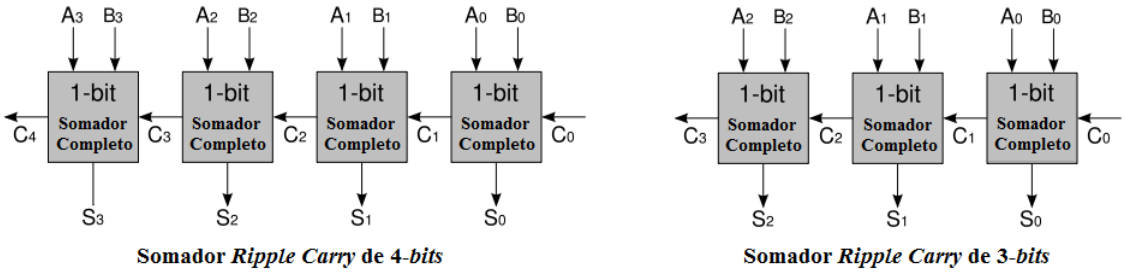

Figura 2.5. Dois *RCAs* com diferentes números de *bits*

arbitrária, isso porque o *hardware* é composto de componentes físicos, que não permite a criação de um circuito que opere sobre uma quantidade qualquer de *bits*, sem que haja um teto.

2.3.1 Estrutura da linguagem

A linguagem *elLen* foi estruturada de maneira a tornar bastante simples a descrição dos modelos. O número de regras existentes é pequeno e, apesar de fornecer certa abstração para o usuário, está bem próxima do nível RTL.

Todo modelo deve ser descrito através de módulos, que são identificados pela palavra-chave *module* seguido por nome que deve ser único em todo o arquivo do modelo. O nome do módulo deve respeitar a regra de ser iniciado por uma letra e conter apenas letras, números e o caractere sublinhado (_). Toda a informação contida entre as palavras-chave *module* e *endmodule* são consideradas pertencentes aos módulos. O trecho a seguir exemplifica a criação de um módulo chamado *modulo_exemplo*.

```
module modulo_exemplo
... corpo do módulo
endmodule
```

O corpo do módulo é dividido em quatro partes: parâmetros, entradas, saídas e comandos; dispostos exatamente nesta ordem. Nenhuma das quatro partes é obrigatória, apesar de um módulo sem nenhuma entrada e saída, e comando não possuir utilidade prática.

A função dos parâmetros é definir toda informação cujo valor varia de uma instância para outra. Os parâmetros são identificados através do uso da palavra-chave *parameter*, dispostos em forma de lista e separados por vírgula, respeitando a mesma regra de nomeação utilizada pelo módulo, porém finalizado com o caractere ponto-e-vírgula.

A função das entradas e saídas é definir a interface de conexão entre os módulos, não sendo possível ligar um módulo ao outro senão através de suas interfaces. As entradas e saídas são identificadas pelas palavras-chave *input* e *output*, respectivamente, e seguem a mesma regra de nomeação dos parâmetros. As entradas e saídas podem ser definidas como conjuntos de sinais de comunicação, explicitado através do uso de colchetes, e pode conter uma expressão matemática simples[5] em função dos parâmetros previamente listados. O exemplo a seguir complementa o módulo *modulo_ exemplo*. Nele foi adicionando o parâmetro *n*, definindo a existência de duas entradas chamadas *valA* e *valB*, cada uma composta por *n* sinais, e uma saída chamada *resul* com *n+1* sinais.

```
module modulo_exemplo
  parameter n;
  input valA[n], valB[n];
  output resul[n+1];
  ... lista de comandos
endmodule
```

Os comandos constituem a parte mais elaborada da linguagem, pois são responsáveis por definir as regras de conexão entre os módulos que compõe o modelo. De uma maneira simplificada, os comandos podem ser definidos em duas classes: instâncias e ligações.

As instâncias correspondem a criação dos módulos utilizados como componentes de outros módulos. A instância é um importante mecanismo de modularização dos modelos, aumentando sua reutilização e facilitando a legibilidade da descrição dos mesmos. Uma instância é caracterizada pela utilização da palavra-chave *instance* seguida do nome do módulo a ser instanciado e os valores dos parâmetros dos módulos entre parênteses. É importante que os valores estejam listados exatamente na mesma ordem em que foram definidos dentro do modelo do módulo. Assim como as entradas e saídas, os valores dos parâmetros aceitam expressões matemáticas simples. Toda instância segue a mesma regra de nomeação já apresentada e contém um identificador. Complementamos o exemplo do módulo *modulo_ exemplo* com a instância de um módulo *subModulo*. O módulo *subModulo* ainda não foi apresentado, mas por enquanto é necessário saber que ele não possui parâmetros.

```
module modulo_exemplo
```

[5]Expressão Matemática Simples: Expressão matemática em função dos parâmetros e contendo apenas as quatro operações aritméticas básicas (adição, subtração, multiplicação e divisão).

```
  parameter n;
  input valA[n], valB[n];
  output resul[n+1];

  sb := instance subModulo();
endmodule
```

Um tipo de instância especial é o uso de portas lógicas básicas [6]. Toda porta lógica básica pode ser instanciada utilizando apenas seu respectivo nome, sem necessidade de criação de um módulo para cada uma delas. Uma peculiaridade é que todos os sinais de entrada das portas lógicas básicas devem ser passados no momento de sua instanciação, e não através de ligação, como nos módulos. Similarmente, a saída de uma porta lógica básica também deve ser conectada a um sinal no momento de sua instanciação. O exemplo a seguir demonstra a utilização das portas lógicas básicas para a criação do módulo *subModulo*, utilizado no exemplo anterior.

```
module subModulo
  input a,b,cIn;
  output s, cOut;
  x1 := xor(a,b);
  s := xor(x1,cIn);
  a1 := and(x1,cIn);
  a2 := and(a,b);
  cOut := or(a1,a2);
endmodule
```

As ligações, usadas nos módulos, correspondem a conexões entre as entradas e saídas dos componentes. Toda ligação é realizada, obrigatoriamente, através da conexão de um par de sinais. Um sinal é identificado por nome único, exatamente como os módulos, parâmetros, entradas e saídas, mas que se utilizado exclusivamente dentro do módulo, não precisa ser previamente declarado. Conforme mencionado anteriormente, entradas e saídas também são sinais e, portanto, obedecem as mesmas regras de conexão a seguir.

Toda ligação é definida como um par *destino* := *origem*, exatamente como a conexão de dois vértices em um grafo direcionado. A componente *destino* pode ser um sinal auxiliar do módulo, a entrada de um módulo instanciado, a saída do módulo

[6] Portas lógicas básicas: *AND*, *OR*, *NOT*, *BUFFER*, *NAND*, *NOR*, *XOR*, *XNOR*.

sendo descrito, ou a entrada de um elemento de memória associado a outro sinal. A componente *origem* pode ser um sinal auxiliar ao módulo, a saída de um módulo instanciado, a entrada do módulo sendo descrito, ou a saída de um elemento de memória associado a outro sinal. Sinais de entrada ou saída devem ser acessados através do nome da instância do módulo, um ponto final, seguido do nome da entrada ou saída. Para sinais compostos, deve-se utilizar o índice do sinal a ser acessado. Valores de índices podem conter expressões matemáticas simples. Elementos de memória são definidos através do uso de um sinal de apóstrofe seguido da palavra chave *reg*.

Completaremos o *modulo_exemplo* com quatro ligações: dois sinais de entrada do módulo foram conectados as entradas da instância do *subModulo*, a saída *s* foi ligada a um elemento de memória, e a saída do elemento de memória associado a saída do *modulo_exemplo*.

```
module modulo_exemplo
  parameter n;
  input valA[n], valB[n];
  output resul[n+1];

  sb := instance subModulo();
  sb.a := valA[0];
  sb.b := valB[0];
  mem'reg := sb.s;
  resul[0] := mem'reg;
endmodule
```

A linguagem *elLen* apresenta três elementos de indexação em função do parâmetro. Esses elementos são importantes, uma vez que a definição de um modelo, em geral, é realizada para um número arbitrário de *bits*, e *a priori* não se sabe o valor do parâmetro.

O primeiro elemento é o próprio identificador do parâmetro, cujo significado está relacionada a *para todos os valores de*. Portanto, ao indexar um conjunto de sinais com o próprio parâmetro, estamos dizendo que a ligação vale para todos os valores do parâmetro, de zero até o número estabelecido na instância. Nos exemplos anteriores, se o *modulo_exemplo* fosse instanciado com o valor cinco para o parâmetro *n*, isso implicaria em indexar com todos os valores de zero a quatro (cinco valores diferentes).

O segundo elemento é o valor do parâmetro recebido durante a instanciação do módulo. Essa indexação é realizada através do símbolo tralha (#). Dessa forma,

ao indexar utilizando #parâmetro estamos dizendo qual é o valor do parâmetro na instância. Ainda com o exemplo da instância de valor cinco, indexar com *#n* implica indexar com o valor cinco.

O terceiro elemento é o valor atual em uma ligação com a semântica *para todos os valores de*. Esse elemento só pode ser utilizado na componente *origem*, se, e somente se, o primeiro elemento de indexação foi utilizado na componente *destino*. Essa indexação é realizada através do símbolo cifrão ($). Uma ligação do tipo *destino*[*n*] := *origem*[*$n*] conecta cada sinal de destino ao seu respectivo valor de origem, definido pelo valor do índice.

Auxiliar de indexação é a limitação da faixa de valores do parâmetro. Esse auxiliar só pode ser utilizado com o primeiro elemento de indexação (*para todos os valores de*), uma vez que os demais correspondem a valores de instâncias e não a faixas de valores. O auxiliar é utilizado para definir a faixa de valores do parâmetro entre os símbolos de abre chaves ({) e fecha chaves (}), separados por vírgula. Se utilizarmos o mesmo exemplo anterior, uma ligação do tipo *destino*[*n*{1,*#n-1*}] := *origem*[*$n*] conecta cada sinal de destino ao seu respectivo valor de origem, definido pelo índice, mas apenas para os valores de *n* entre *1* e *n-1*.

O exemplo apresentado no decorrer desta seção corresponde a parte do modelo do RCA apresentado na seção 2.3. O modelo completo pode ser visto a seguir.

```
module rippleCarryAdder
  parameter n;
  input valA[n], valB[n];
  output resul[n+1];

  sc[n] := instance somadorCompleto();

  resulMem[n]'reg := sc[$n].s;
  resulMem[#n]'reg := sc[#n-1].cOut;

  sc[n].a := valA[$n];
  sc[n].b := valB[$n];
  sc[n{1,#n-1}].cIn := sc[$n-1].cOut;
  sc[0].cIn := LOW;

  resul[n] := resulMem[$n]'reg;
  resul[#n] := resulMem[#n]'reg;
```

```
endmodule

module somadorCompleto
  input a,b,cIn;
  output s, cOut;
  x1 := xor(a,b);
  s := xor(x1,cIn);
  a1 := and(x1,cIn);
  a2 := and(a,b);
  cOut := or(a1,a2);
endmodule
```

2.3.2 Definição da gramática

A subseção 2.3.1 apresentou uma descrição informal para a estrutura da linguagem *elLen*. Essa descrição foi efetuada com objetivo de auxiliar o leitor quanto à forma de utilização da linguagem sem preocupação com a gramática. No entanto, como a linguagem *elLen* é formal, ela deve ser apresentada segundo suas regras específicas.

No decorrer desta subseção, a gramática detalhada de *elLen* será apresentada sem, no entanto, descrever o processo de construção da mesma. Caso necessite, o leitor poderá retornar a subseção 2.2.2 e rever a definição de gramática para melhor entendimento do conteúdo a seguir.

Como foi dito, uma linguagem *L(G)* é definida como um conjunto de sentenças que podem ser geradas a partir da gramática *G*. Vimos ainda que uma gramática *G* é uma quádrupla (V,Σ,R,P). Antes de apresentarmos os conjunto de regras da gramática da linguagem *elLen*, iremos definir o alfabeto Σ.

A um agrupamento de caracteres daremos o nome de *token*, o qual corresponde a um terminal do alfabeto Σ e será utilizado a partir de agora como sinônimo. Esse agrupamento é necessário para simplificar a gramática e porque a semântica do terminal só importa para a gramática, sendo o valor necessário apenas na transformação entre o modelo e um grafo de módulos (ver seção 2.4).

A tabela 2.1 apresenta a lista de terminais e seus respectivos *tokens*. Os terminais serão identificados na gramática com nomes sublinhados, facilitando a identificação quando misturados às variáveis.

A tabela 2.2 apresentada a gramática de *elLem*. Todas as palavras não sublinhadas pertencem ao conjunto V de variáveis da linguagem. A variável de partida P é

Terminal	***Token***
pontoVirgula	Caractere ponto-e-vírgula (;)
virgula	Caractere vírgula (,)
atrib	Caractere dois pontos seguindo de igual (:=)
ponto	Caractere ponto final (.)
adicao	Caractere mais (+)
subtracao	Caractere menos (-)
multiplicacao	Caractere asterisco (*)
divisao	Caractere barra (/)
colcOpen	Caractere abre colchete ([ )
colcClose	Caractere fecha colchete ( ])
parOpen	Caractere abre parênteses (( )
parClose	Caractere fecha parênteses ( ))
chaveOpen	Caractere abre chave ({)
chaveClose	Caractere fecha chave (})
moduleKey	Palavra *module*
endmoduleKey	Palavra *endmodule*
parameterKey	Palavra *parameter*
inputKey	Palavra *input*
outputKey	Palavra *output*
lowKey	Palavra *LOW*
highKey	Palavra *HIGH*
andKey	Palavra *and*
orKey	Palavra *or*
notKey	Palavra *not*
bufferKey	Palavra *buffer*
nandKey	Palavra *nand*
norKey	Palavra *nor*
xorKey	Palavra *xor*
xnorKey	Palavra *xnor*
regKey	Caractere apóstrofo seguido da palavra *reg* ('reg)
numero	Qualquer número inteiro sem zeros à esquerda
identificador	Qualquer palavra começada por uma letra seguida por letras e/ou números e/ou sublinhados

Tabela 2.1. Tabela de terminas e *tokens*

representada pela variável *inicio*. Utilizaremos como forma de apresentação das regras o mesmo esquema visto na subseção 2.2.2.

Como pôde ser visto, a gramática da linguagem *elLen* é bastante simples. Essa simplicidade auxilia não só o processamento, mas também o aprendizado da linguagem por parte do usuário.

inicio →	modulos
modulos →	modulos modulo | modulo
modulo →	moduleKey nomeValido parametros entradas saidas corpo endmoduleKey
parametros →	parameterKey listaParametros pontoVirgula |
listaParametros →	listaParametros virgula nomeValido | nomeValido
entradas →	inputKey listaEntradas pontoVirgula |
listaEntradas →	listaEntradas virgula nomeValido | nomeValido
saidas →	outputKey listaSaidas pontoVirgula |
listaSaidas →	listaSaidas virgula nomeValido | nomeValido
corpo →	listaComandos
listaComandos →	listaComandos comando pontoVirgula |
comando →	acesso atrib acesso pontoVirgula | acesso atrib constante pontoVirgula | acesso atrib portasBasicas parOpen listaAcesso parClose pontoVirgula
listaAcesso →	listaAcesso virgula acesso | acesso
acesso →	nomeValido indice interface
interface →	ponto nomeValido indice |
indice →	colcOpen expressao colcClose reg |
reg →	regKey |
portasBasicas →	andKey | orKey | notKey | bufferKey | nandKey | norKey | xorKey | xnorKey
nomeValido →	identificador
elemento →	identificador | numero
constante →	lowKey | highKey
expressao →	elemento | parOpen expressao parClose | expressao adicao expressao | expressao subtracao expressao | expressao multiplicacao expressao | expressao divisao expressao

Tabela 2.2. Gramática da linguagem *elLen*

2.4 De modelo a circuito

A definição de um modelo é bastante importante durante a etapa de geração dos circuitos porque permite a definição de um conjunto de circuitos que possuem o mesmo processo de construção. Apesar da importância do modelo, ele por si só não representa um circuito e, portanto, não possui utilidade para as ferramentas em EDA, as quais precisam de circuitos efetivamente descritos através de uma de suas linguagens de entrada.

A instanciação de um modelo constitui o passo intermediário entre o modelo e o circuito gerado. Ao instanciar um modelo, o primeiro passo é definir valores para parâmetros (lembre-se que as ligações são realizadas utilizando os valores dos parâmetros).

Definido os valores para os parâmetros, um grafo é criado para representar a instância do modelo. Esse grafo é muito importante para o processo de geração do circuito em outras linguagens (ver capítulo 4.5) e também para o processo de síntese do circuito (ver capítulo 3).

Ao instanciar um módulo são criados vértices para todos os seus sinais de entrada e saída. Essa etapa é muito importante, pois só pode haver comunicação entre os módulos através dos sinais de entrada e saída. Cada módulo instanciado dentro de outro módulo é visto como uma caixa preta, ou seja, não se conhece o conteúdo, apenas sua interface (entradas e saídas). Dois tipos de vértices são criados durante esse processo: *vértices io* e *vértices módulos*. Para cada *vértice io* que representa uma entrada é criada uma aresta partindo dele e incidindo no *vértice módulo*, e para cada *vértice io* que representa uma saída é criada uma aresta partindo do *vértice módulo* e incidindo sobre ele. A figura 2.6 apresenta um grafo instanciando um módulo com três entradas e três saídas. Nele, os *vértices io* são graficamente representados por triângulos, e os *vértices módulos* por retângulos.

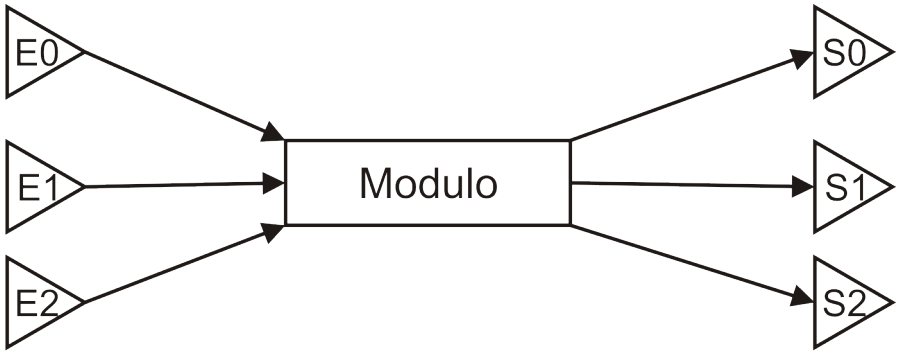

Figura 2.6. Grafo de um módulo com 3 entradas e 3 saídas

Cada comando processado corresponde a criação de uma aresta entre dois vértices, o vértice do sinal de origem e o do sinal de destino. O vértice origem já deverá existir, ora por corresponder a uma entrada ou saída de um módulo, ora por ser destino de um comando que já foi processado. Caso o vértice de destino não exista, ele é gerado, para posterior criação da aresta. Três tipos de vértices são possíveis ao processar um comando que não seja de instância de módulo: *vértice sinal*, *vértice porta* e *vértice reg*. Um *vértice sinal* é criado cada vez que um sinal interno ao módulo é processado, e é representado por um losângulo. A criação de um *vértice porta* se dá toda vez que uma porta básica é instanciada, sendo ilustrado por um círculo. Um *vértice reg* surge a cada referência feita a um elemento de memória de um sinal, e é simbolizado por um quadrado. A figura 2.7 mostra o processamento de um comando que utiliza os três

tipos de vértices.

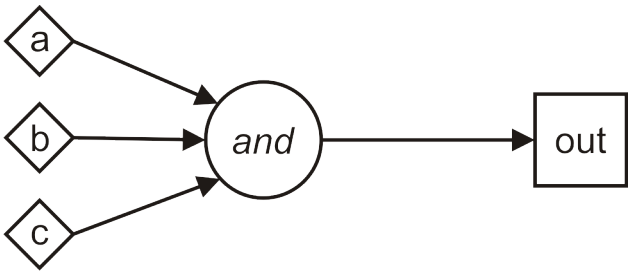

Figura 2.7. Grafo gerado pelo comando *out'reg := and(a,b,c)*

Uma instância é considerada válida se para toda entrada existe um caminho que leve à saída; e se a saída possuir um valor constante, ou puder ser alcançada a partir de uma entrada. A figura 2.8 demonstra a criação do grafo representando uma instância válida do módulo *rippleCarryAdder* com parâmetro $n = 3$.

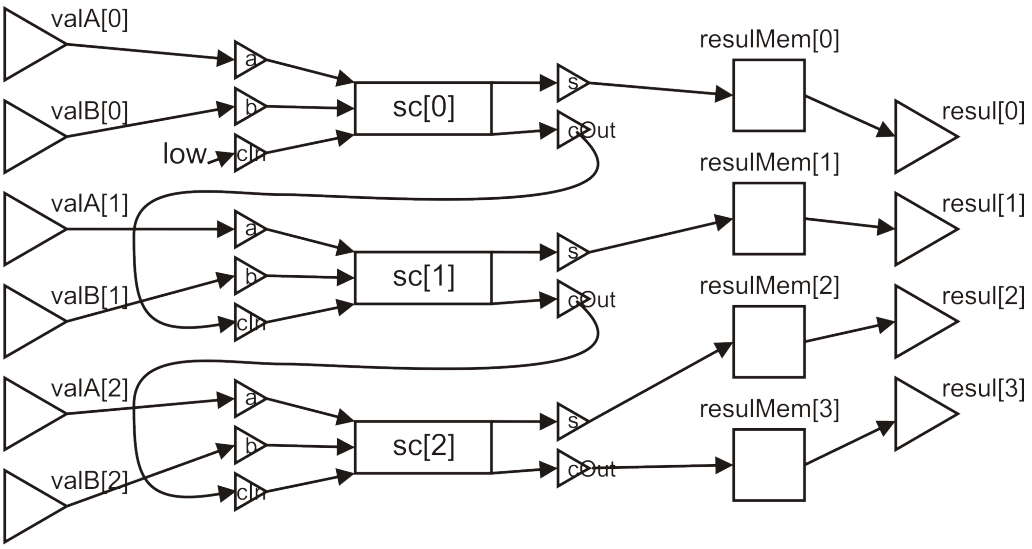

Figura 2.8. Grafo da instância do RCA com $n = 3$

Como observado, o módulo *somadorCompleto* foi representado como uma caixa preta. Essa organização permite que módulo a módulo possa ser validado. Após a validação de todos, os *vértices módulos* são substituídos pelo grafo do módulo em questão, formando assim um único grafo com todos os vértices, nenhum do tipo *vértice módulo*.

É importante o pleno entendimento do processo de construção do grafo da instância a partir do modelo. Os grafos construídos nesta etapa formam a base para o processo de síntese, geração e visualização dos circuitos.

Capítulo 3

A síntese elSin

3.1 Introdução

Síntese lógica é o nome dado ao processo de produção automática de componentes lógicos, e diz respeito à abstração, representação, manipulação, análise e otimização de circuitos lógicos. A síntese lógica desempenha um papel fundamental na automação de projetos eletrônicos (EDA), sendo duas técnicas amplamente utilizadas em outros campo, a exemplo da verificação formal e síntese de *software*.

Desde o primeiro circuito integrado (CI), inventado em 1958 por Jack Kilby, iniciou-se a busca por processos de miniaturização e aumento da densidade de transistores em um único CI. Gordon Moore, em 1965, anunciou uma tendência que mais tarde ficou conhecida como *Lei de Moore* [Moore, 1965], sendo a qual o número de transistores em um único CI crescia exponencialmente, dobrando aproximadamente a cada dois anos. Essa tendência persiste até os dias de hoje, com fracos indícios de queda.

Diante desse notável crescimento (inclusive em complexidade) dos CIs e de seus projetos, surtiu a necessidade de maximizar a automação dos processos e técnicas utilizadas na produção dos circuitos integrados. Hierarquização e abstração são típicas em EDA, claramente observadas respectivamente 1)na abordagem de dividir para conquistar, e 2) na descrição dos circuitos em diferentes níveis, como nível comportamental (*behavior level*); RTL; nível de portas lógicas (*Gate Level*); nível de transistores (*Transistor Level*); entre outros. A síntese lógica é o processo aplicado na transição de RTL para nível de portas lógicas ou transistores, e é altamente automatizado.

A base para a síntese lógica está na interseção da álgebra com a lógica, criada por George Boole em 1848, e entitulada *álgebra booleana* [Boole, 1848]. Uma vez que o presente trabalho diz respeito a circuitos digitais, será utilizada a *álgebra booleana de*

dois elementos[1] .

Todo circuito combinatório pode ser descrito através de uma fórmula em dois níveis, chamada de soma de produtos. A utilização de PLAs (*Programmable Logic Arrays*) para implementação de lógicas de controle é baseada neste fato. Todo circuito sequencial é constituído de duas partes: circuito combinatório e elementos de memória. Os elementos de memória são responsáveis por armazenar o estado atual do circuito, sendo que este pode ser modelado através de um grafo (para grafo, ver subseção 2.2.2) cujos vértices representam os possíveis estados, e as arestas as possíveis transições entre estados. Todo circuito sequencial implementa uma máquina de estado finito (FSM, *Finite State Machine*), e um circuito combinatório pode ser visto como um circuito sequencial de um único estado, sem memória.

Muitos estudos importantes na área de síntese lógica foram publicados na década de 80, dentre eles destacam-se a retemporização de circuitos sequenciais síncronos, mapeamento tecnológico, ROBDD (*Reduced Ordered Binary Decision Diagrams*) e verificação de equivalência sequencial. As duas maiores ferramentas de síntese lógica desta época foram ESPRESSO [Rudell & Sangiovanni-Vincentelli, 1987] e MIS [Brayton et al., 1987]. Nos anos 90, os desafios da síntese lógica consistiam em resolver os problemas de consumo de energia, testabilidade e a utilização de novas tecnologias, como os FPGAs. A ferramenta que mais se destacou foi SIS [Sentovich et al., 1992], uma evolução da MIS. Após o ano 2000, os principais desafios passaram a ser escalabilidade e verificabilidade, cujas ferramentas mais conhecidas na literatura são MVSIS [Brayton et al., 2002] e ABC [Synthesis & Group, 2005].

Os avanços da síntese lógica alavancaram a indústria em EDA. Uma das primeiras aplicações comerciais, desenvolvida pela empresa Synopsys, permitia o remapeamento de um projeto que utilizava uma biblioteca de células padrão em outra, evitando assim a necessidade de reprojeto do CI. Empresas grandes como a IBM já demonstraram a utilidade de síntese lógica através de milhares de CIs projetados. Atualmente a síntese lógica é aplicada em vários tipos de projetos, a exemplo dos CIs de aplicação específica (*ASIC*, *Application Specific IC*), onde é amplamente utilizada; e em projetos *full custom*, onde sua aplicação é reduzida.

Neste capítulo é descrito o processo de síntese lógica executado pela ferramenta *elGen*, denominado *elSin*. É importante frisar que o objetivo da ferramenta não é competir com as ferramentas de síntese existentes, mas apenas utilizar esse processo como forma de validação dos circuitos gerados. A síntese lógica apresentada neste trabalho é uma etapa anterior ao mapeamento tecnológico e à síntese física.

[1] Álgebra booleana de dois elementos: Álgebra booleana cujo domínio dos elementos da álgebra é composto de dois elementos apenas, em geral representados por 1 e 0, ou verdadeiro e falso.

3.2 Fundamentação teórica

Nesta seção será apresentada uma série de conceitos relacionados ao processo de síntese lógica. Esses conceitos são importantes para o entendimento da etapa de síntese executada pela ferramenta. Algumas informações contidas nesta seção foram extraídas do livro texto *Logic Synthesis and Verification Algorithms* [Hachtel & Somenzi, 1996].

Em sua obra, Hachtel & Somenzi [1996] combina e integra modernas técnicas de desenvolvimento em síntese lógica e verificação formal com tradicionais conceitos de comutação e teoria de autômatos finitos. O livro também provê uma boa referência a respeito de álgebra booleana e matemática discreta.

3.2.1 Representação de funções booleanas

Grande parte da eficiência dos algoritmos está relacionada à estrutura de dados utilizada, e que algoritmos similares podem possuir ordem de complexidade diferentes apenas em função da estrutura de dados. Os dados a serem processados durante a síntese lógica consistem, basicamente, de funções booleanas. Como representar as funções de maneira compacta e como processá-las de maneira eficiente são desafios a serem vencidos. No decorrer desta subseção serão apresentadas algumas das mais comuns representações utilizadas: Tabela da Verdade, Soma de Produtos, Produto de Somas, Diagrama de Decisão Binária, AIG.

Tabela da Verdade: O mapeamento de uma função booleana pode ser enumerado através de uma tabela da verdade, figura 3.1, onde cada variável assinalada possui um valor correspondente listado.

Tabelas da verdade são representações canônicas de funções booleanas, assim, duas funções booleanas são equivalentes se, e somente se, elas possuem a mesma tabela da verdade. Canonicidade é uma propriedade importante, útil em aplicações de síntese lógica e verificação, pode ser usada, por exemplo, para detectar funções ou subfunções idênticas.

Na prática, tabelas da verdade são representações eficientes apenas para funções com poucas variáveis, uma vez que o número de entradas na tabela é determinado pela função 2^n, onde n é o número de variáveis. Essa representação torna-se inviável uma vez que, geralmente, o número de variáveis é grande.

Soma de Produtos (SOP, *Sum of Products*): SOP, ou forma normal disjuntiva (DNF, *Disjunctive Normal Form*), é uma forma especial de fórmulas booleanas consistida de disjunções (somas) de conjunções de literais (produto dos termos). Essa

a	b	c	$f(a,b,c)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Figura 3.1. Exemplo de representação através de tabela da verdade

representação, figura 3.2, corresponde a um circuito em dois níveis, o primeiro contendo apenas portas lógicas AND, e o segundo apenas uma porta lógica OR.

Toda fórmula booleana pode ser reescrita através de uma soma de produtos, mas essa representação não é canônica. De fato, expressar uma função booleana da forma SOP mais concisa é um problema da classe *NP-Completo*.

$$f(a,b,c) = \neg a \neg bc + \neg ab \neg c + a \neg b \neg c + a \neg bc + ab \neg c$$

Figura 3.2. Exemplo de representação através de *SOP*

Produto de Somas (POS, *Product of Sums*): POS, ou forma normal conjuntiva (CNF), é uma forma especial de fórmulas booleanas consistida de conjunções (produtos) de disjunções de literais (soma dos termos). A fórmula booleana representada em POS, figura 3.3, pode ser obtida através do complemento de SOP, o mesmo acontece com a fórmula em SOP que pode ser obtida através do complemento de POS.

Problemas de satisfabilidade (SAT) sobre fórmulas CNF são um dos mais importantes problemas em ciência da computação.

$$f(a,b,c) = (a + b + c).(a + \neg b + \neg c).(\neg a + \neg b + \neg c)$$

Figura 3.3. Exemplo de representação através de *POS*

Diagrama de Decisão Binária (BDD): BDDs são representações de funções booleanas através de um grafo acíclico direcionado, constituído de dois tipos de vértices, classificados como de decisão ou terminas. Cada vértice de decisão é rotulado por uma variável booleana e possui dois filhos chamados filho-0 e filho-1. Uma aresta que liga um vértice a seu filho representa um assinalamento à variável, onde a tracejada corresponde ao assinalamento 0 e liga o vértice ao filho-0. A aresta contínua representa o assinalamento 1 e liga o vértice ao seu filho-1.

Os BDDs foram propostos por Lee [Lee, 1959] e desenvolvidos por Akers [Akers, 1978], e mais tarde ganharam a forma canônica por Bryant [Bryant, 1986], através da forma ordenada e reduzida, conhecida como ROBDD.

Formalmente, um BDD pode ser definido da seguinte maneira:
- Um vértice terminal v possui um atributo cujo valor dado pela função $value(v) \in \{0, 1\}$.
- Um vértice não terminal v possui um atributo representando o índice do nível no BDD, $index(v) \in \{1, ..., n\}$, onde n é o número de níveis; e dois filhos: filho-0, denotado $else(v) \in V$, e filho-1, denotado $then(v) \in V$.
- Se $index(v) = i$, então x_i é chamada de variável de decisão para o vértice v.

Cada vértice v em um BDD corresponde a uma função $f[v]$ definida recursivamente assim:
1. Se o vértice v é um terminal,
(a) Se $value(v) = 1$, então $f[v] = 1$.
(b) Se $value(v) = 0$, então $f[v] = 0$.
2. Se o vértice v é um não terminal,
$f[v](x_1, ..., x_n) = \neg x_i.f[else(v)](x_1, ..., x_n) + x_i.f[then(v)](x_1, ..., x_n)$

A figura 3.4 apresenta um exemplo de BDD em sua forma original, correspondente à tabela da verdade apresentada na figura 3.1.

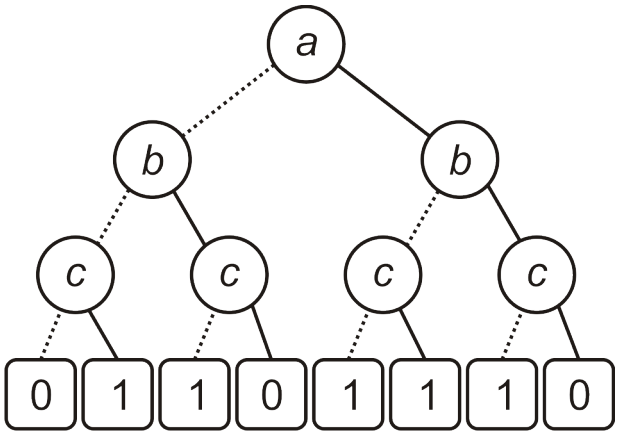

Figura 3.4. Exemplo de representação através de BDD original

Um BDD é dito *ordenado* (OBDD) se os vértices em todos os caminhos da raiz a um terminal possuem a mesma ordenação de variáveis. Dois OBDDs D_1 e D_2 são considerados isomorfos se existe uma função bijetora $fb(v)$ dos vértices de D_1 nos vértices de D_2, tal que $\forall v \in D_1$, $value(v) = value(fb(v))$, se v é um terminal; $index(v) = index(fb(v))$, $then(v) = then(fb(v))$, e $else(v) = else(fb(v))$, se v é um não terminal. Um BDD é *reduzido* (ROBDD) caso as três condições seguintes sejam

atendidas: se dois vértices terminais com mesmo valor são fundidos; se dois vértices não terminais u e v com a mesma variável de decisão, $else(u) = else(v)$, $then(u) = then(v)$, são fundidos; e se um vértice não terminal v com $then(v) = else(v)$ é removido e as arestas a ele incidentes são redirecionadas a seus filhos. A figura 3.5 apresenta o ROBDD do BDD apresentado na figura 3.4.

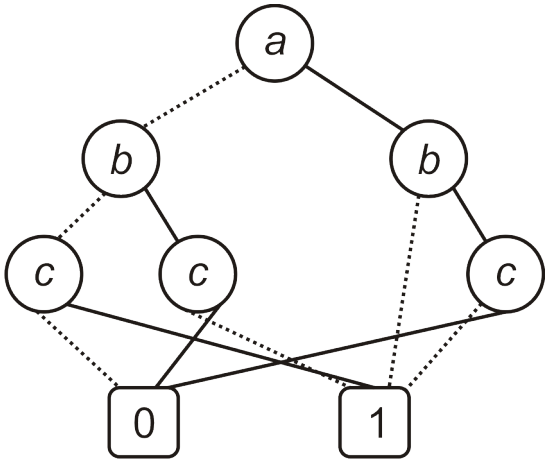

Figura 3.5. Exemplo de representação através de ROBDD

BDDs são estruturas muito úteis e compactas, mas sofrem com o problema da ordenação das variáveis. Note que ordens diferentes de variáveis podem resultar em BDDs com diferentes números de vértices. Apesar de existirem técnicas avançadas de escolha da ordenação das variáveis, não é possível determinar a organização ótima, pois se trata de um problema *NP-Completo* [Grumberg et al., 2003][Zhang et al., 2002]. A figura 3.6 apresenta um ROBDD diferente para a mesma função booleana apresentada, variando apenas a sua ordenação.

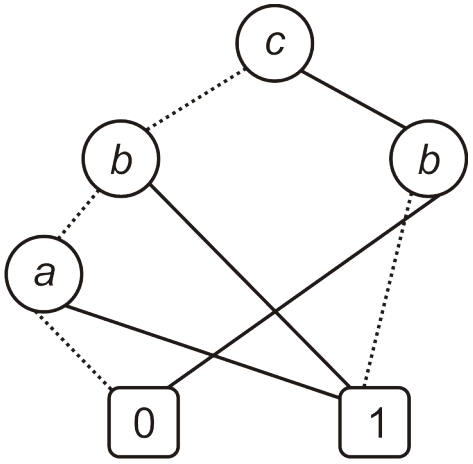

Figura 3.6. ROBDDs com ordenação $c < b < a$

And-Inverter Graph(AIG): Um AIG é um grafo acíclico direcionado *G=(V,E)* onde cada vértice $v \in V$ representa uma porta AND de duas entradas, e cada aresta $a \in E \subseteq V \text{x} V$ a conexão entre as portas. Inversores são representados como arestas marcadas por um ponto. Uma vez que as operações de conjunção e negação ($\wedge$ e $\neg$) são funcionalmente completas, qualquer função booleana pode ser representada por um AIG. A figura 3.7 ilustra o AIG correspondente à mesma função apresentada anteriormente.

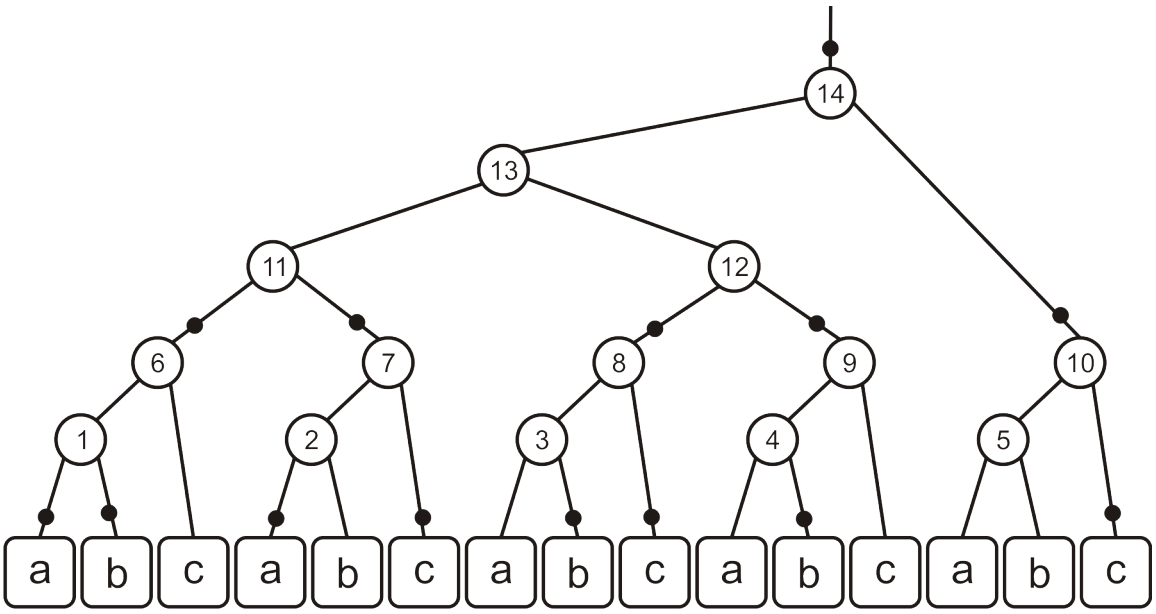

Figura 3.7. Exemplo de representação através de AIG

A estrutura simplificada de um AIG permite uma rápida e barata estrutura de *hash* em seus vértices. Dois vértices com a mesma entrada e nas mesmas condições de complementação são fundidos. O *hash* em AIGs é semelhante ao processo de fusão dos vértices em ROBDD, no entanto, sem a propriedade de possuir uma representação canônica. A figura 3.8 mostra a versão com *hash* do AIG apresentado na figura 3.7.

As três operações básicas (conjunção, disjunção e complemento) podem ser realizadas com complexidade constante. Para a operação de conjunção entre dois AIGs, basta adicionar um vértice. A operação de disjunção é feita exatamente como a conjunção, exceto pela adição de marcas nas arestas de entradas e saída do vértice adicionado, em complexidade constante. O complemento é realizado simplesmente adicionando marcas as arestas não marcadas, ou removendo as marcas das arestas marcadas, de maneira constante. Uma variação do AIG para circuitos sequenciais consiste de uma marca adicional à aresta, representando um elemento de memória.

Um formato binário chamado AIGER [Biere, 2007] foi proposto para permitir a

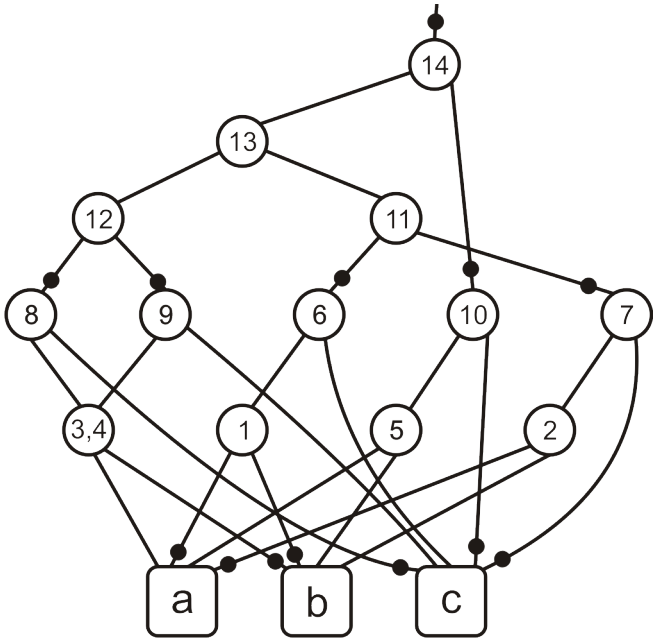

Figura 3.8. AIG com estrutura de *hash*

representação compacta de AIGs em arquivos e memória. A representação é realizada utilizando apenas três *bytes* por vértice, e por isso tornou-se a representação padrão para problemas com base em circuitos nas competições de resolvedores SAT.

3.2.2 Síntese combinatória e sequencial

O processo de síntese lógica pode ser dividido em duas etapas: Síntese combinatória e síntese sequencial. A primeira consiste em, dada uma fórmula booleana *f*, encontrar uma fórmula *f'* funcionalmente equivalente que seja mais otimizada. É importante perceber que os critérios para otimização podem variar de uma síntese para outra, e estão diretamente ligados ao mapeamento tecnológico. Já a segunda é caracterizada pela codificação do diagrama de estados do circuito, eliminação dos estados redundantes ou inalcançáveis, e retemporização. A síntese sequencial também está ligada ao mapeamento tecnológico, uma vez que as funções de excitação dos elementos de memória dependem do tipo de elemento escolhido.

É importante relembrar que a ferramenta apresentada neste trabalho não realiza qualquer tipo de mapeamento tecnológico, portanto, toda a síntese lógica consiste em reduzir o número de portas lógicas AND e NOT utilizadas, e aumentar a frequência máxima na qual o sistema pode operar, ou seja, reduzir o caminho crítico do circuito.

A síntese combinatória é baseada nos teoremas da álgebra booleana [Ronald J. Tocci, 2007], que são aplicados sistematicamente até que não seja mais possível

simplificar a função. Considere a função utilizada como exemplo na subseção 3.2.1, $f(a,b,c) = \neg a\neg bc + \neg ab\neg c + a\neg b\neg c + a\neg bc + ab\neg c$. A figura 3.9 apresenta a aplicação dos teoremas até encontrar a forma mais simplificada da função.

$f(a,b,c) = \neg a\neg bc + \neg ab\neg c + \underline{a\neg b\neg c} + \underline{a\neg bc} + ab\neg c \longrightarrow Teorema : xy + x\neg y = x$
$f(a,b,c) = \neg a\neg bc + \underline{\neg ab\neg c} + a\neg b + \underline{ab\neg c} \longrightarrow Teorema : xy + x\neg y = x$
$f(a,b,c) = \underline{\neg a\neg bc} + b\neg c + \underline{a\neg b} \longrightarrow Teorema : x + \neg xy = x + y$
$f(a,b,c) = \neg bc + b\neg c + a\neg b$

Figura 3.9. Aplicação de teoremas booleanos para simplificação de função

Observe que existe um trabalho de agrupamento das variáveis da função de forma a possibilitar a aplicação dos teoremas. Na seção 3.3 veremos que esse trabalho é simplificado através da utilização de AIG com *hash*, pois os teoremas podem ser aplicados a cada dois níveis de portas lógicas.

Outro tipo de simplificação que pode ser aplicada e não é contemplada pelos teoremas da álgebra booleana é a transformação de uma porta seguida de um inversor em sua porta negada equivalente (ex. AND seguida de NOT = NAND, NOR seguida de NOT = OR, etc.). Outra simplificação não contemplada determina a utilização de disjunção exclusiva ($\oplus$), ou seja, a substituição de duas AND e uma OR por uma única XOR, expressa através da relação $\neg ab + a\neg b = a \oplus b$.

Um notável trabalho de síntese é a transformação da função em uma SOP para ser gravada em PLAs [Fleisher & Maissel, 1975]. Outro trabalho de extrema importância foi EXPRESSO [Rudell & Sangiovanni-Vincentelli, 1987], baseado nos conceitos de cubo e primo implicante utilizados no método de simplificação Quine-McCluskey [McCluskey, 1956]. O trabalho SCHERZO [Coudert, 1995] é basicamente o EXPRESSO, porém utilizando BDDs como estrutura de representação das funções. SCHERZO superou EXPRESSO em mais de duas ordens de magnitude apenas pela utilização de BDDs, frisando a importância da representação apresentada na seção 3.2.1.

A síntese sequencial realizada por *elSin* está ligada exclusivamente à retemporização, portanto, as demais características dessa etapa não serão abordadas. O processo de retemporização consiste em manipular os elementos de memória presentes no circuito de forma a minimizar o tempo gasto na propagação do sinal no maior caminho existente no circuito, denominado *caminho crítico* - esse assunto será abordado em mais detalhes na subseção 3.3.3.

Vários importantes trabalhos na área de síntese sequencial foram produzidos, como exemplo, destacam-se Pan & Lin [1998] com um método de retemporização fortemente ligado ao mapeamento tecnológico para FPGAs, Bjesse & Borälv [2004] que apresentou uma forma de compressão de um AIG para ser utilizado em verifica-

ção formal, e Pan [1997] executando retemporização com valores reais, em oposição a utilização tradicional de valores inteiros.

3.3 A síntese *elSin*

Vimos na seção 2.4 que uma etapa muito importante da ferramenta *elGen* é a construção do grafo representando a instância do modelo do circuito descrito através da linguagem *elLen*. Esse grafo representa a base para três etapas da ferramenta: síntese lógica, geração dos circuitos e visualização do circuito gerado.

O foco desta seção será apresentar o processo de síntese lógica executado, *elSin*, a partir do grafo do circuito. Esse processo tem como objetivo validar os circuitos gerados, e não obter a melhor representação possível. Todas as operações são baseadas nos teoremas da álgebra booleana e no processo de retemporização (*retiming*).

A síntese *elSim* é dividida em três etapas: transformação do grafo do circuito em um *AIG*, síntese combinatória baseada na álgebra booleana, e retemporização do circuito com a movimentação e criação dos elementos de memória presentes.

3.3.1 Criação do AIG

Durante a subseção 3.2.1 foram apresentadas diversas formas de representação para as funções booleanas. Uma das estruturas mais simples e eficientes mencionada é o *AIG*, e por esse motivo foi escolhida como estrutura de representação para a síntese *elSin*.

A primeira etapa consiste na definição das variáveis da função do circuito. Todas as entradas do circuito e todas as saídas dos elementos de memória são consideradas variáveis da função. A diferença entre os dois tipos de variáveis está no momento em que são criadas: as que representam entradas do circuito são definidas previamente, enquanto as oriundas de elementos de memória surgem dinamicamente à medida que o grafo vai sendo percorrido.

A segunda etapa consiste em caminhar no grafo do circuito e substituir cada um dos *vértices io*, *vértices porta, vértices reg* e *vértices sinal* por elementos equivalentes no *AIG*.

Um *vértice io* possui três destinos, podendo 1) já ter sido transformado em variável, caso seja entrada do módulo principal (figura 3.10-a); 2) corresponder ao resultado de uma função, caso seja saída do módulo principal, e portanto não precisa ser representado (figura 3.10-b); ou 3) corresponder a uma entrada ou saída de um módulo interno, e é eliminado fazendo com que as arestas incidente cheguem diretamente ao

seu destino (figura 3.10-c). Para efeito de distinção, os vértices com fundo colorido representam vértices do grafo do circuito, e os em branco, vértices do AIG.

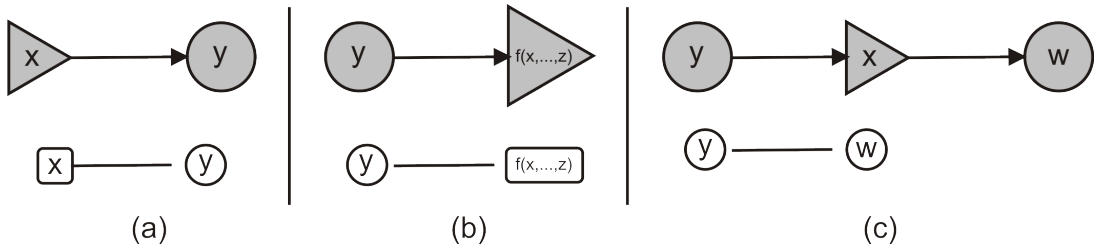

Figura 3.10. As três possíveis transformações de um *vértice io*

Um *vértice sinal* é sempre eliminado, e assim como o *vértice io* de um módulo interno, as arestas que chegam até ele passam a incidir diretamente.

Um *vértice reg* é dividido e transformado em variável. Todos os vértices destino passam a receber no lugar do *vértice reg* uma nova variável criada com sua eliminação. A aresta incidente agora é o resultado da função. A figura 3.11 apresenta um exemplo de transformação de um *vértice reg*.

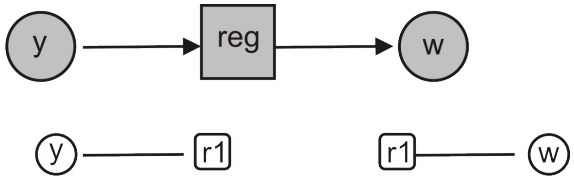

Figura 3.11. Transformação de um *vértice reg* em uma nova variável

Sabe-se que o conjunto dos operadores $\{\neg, \wedge\}$ é completo, ou seja, é possível escrever qualquer fórmula booleana utilizando apenas esses operadores. A última etapa é caracterizada pela substituição de todos os *vértices porta* por representações equivalentes utilizando portas lógicas AND de apenas duas entradas, e portas NOT. Perceba o expressivo crescimento em número de vértices, uma vez que a representação do grafo do circuito permitia portas lógicas com qualquer número de entradas, e também porque algumas portas lógicas necessitam de mais de uma porta AND para representar sua função. É importante lembrar que devido à estrutura de *hash* aplicada na construção do *AIG*, parte do crescimento vertiginoso dos vértices é compensada pelo compartilhamento de alguns deles. A figura 3.12 demonstra a transformação de um *vértice porta* representando uma porta lógica NOR de três entradas.

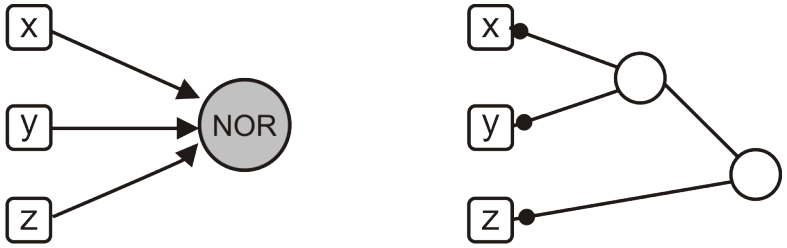

Figura 3.12. Transformação de um *vértice porta* NOR de três entradas

3.3.2 Síntese combinatória

Por meio da subseção 3.2.2, sabe-se que a base para a síntese combinatória é formada por transformações descritas através dos teoremas da álgebra booleana. A maior parte dos teoremas é aplicada a funções, ou parte delas, cujo nível de portas lógicas é, em geral, dois, contendo poucas variáveis. Essa informação é particularmente importante quando aplicados sobre AIGs, pois em um nível uma operação é executada com dois operandos, consequentemente, em dois níveis são executadas três operações com quatro operandos.

É bastante importante a compreensão do processo de *hashing* no AIG, pois será através dele que as variáveis serão identificadas. Duas variáveis a e b representam o resultado de uma mesma função se $hash(a) = hash(b)$. Isso significa que não é possível uma mesma função possuir dois valores de *hash* diferentes, o que torna a condição necessária, mas não suficiente, pois pode existir $hash(a) = hash(b)$, sendo $a \neq b$.

O primeiro passo para a síntese *elSin* está na definição dos vértices que irão compor o conjunto inicial para caminhar no AIG. Os vértices desse conjunto correspondem àqueles gerados durante a criação do grafo (ver seção 3.3.1) e que correspondiam às variáveis da função representada, caracterizados por vértices sem arestas incidentes, chamados *vértices folha*. Essa etapa é importante uma vez que *elSin* realiza o caminhamento no AIG por níveis, ou seja, nenhum vértice do nível n é visitado até que todos de *n-1* sejam. Essa forma de caminhar possui a importante propriedade de não permitir que um vértice seja visitado antes dos dois incidentes.

O próximo passo consiste da aplicação de subprocessos de síntese à medida que é realizado o caminhamento no AIG. Um dos subprocessos é caracterizado pela **propagação de constantes**, que elimina um vértice quando alguma das seguintes situações acontece: $a + 0$, $a + 1$, $a.0$ ou $a.1$. A figura 3.13 ilustra as quatro situações e o resultado da eliminação. Vale ressaltar que a aplicação desse subprocesso, vértice eliminado em um nível n, faz com que o seu sucessor, ou seja, nível *n+1*, passe a partencer ao

n, e assim sucessivamente; sendo, portanto, processado também antes de avançar no caminhamento.

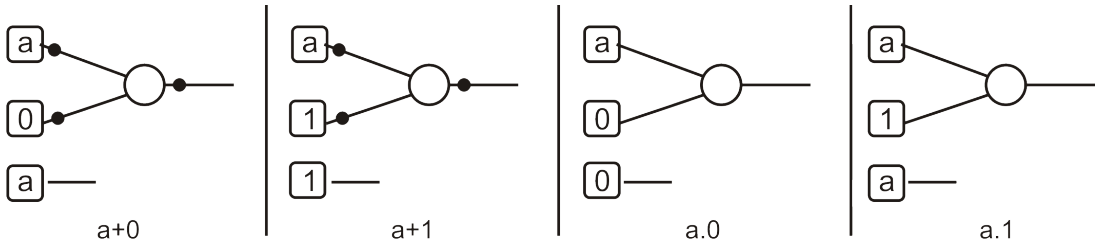

Figura 3.13. Simplificação por propagação de constantes

Outro subprocesso é chamado de **reordenação de variáveis** e possui a tarefa de diminuir a quantidade de representações diferentes para uma mesma função. Considere a função hipotética $f(a, b, c, d) = a.b.c.d$. A mesma função pode ser representada de 24 maneiras diferentes, apenas trocando a ordem das variáveis. Seja $left(v)$ e $right(v)$ duas funções tais que, $\forall v \in V$, retornam a função presente na primeira e segunda entrada de v, respectivamente. A reordenação das variáveis é feita de forma que $\forall v \in V$, $hash(left(v)) < hash(right(v))$. Observe que para a função f, são necessários, no mínimo, dois níveis de vértices em um AIG, mas toda ela pode ser representada por uma única porta lógica AND. É importante perceber que como a ordenação é realizada nível a nível, é possível que a representação final não seja a mais otimizada. A figura 3.14 apresenta a reordenação de variáveis para a função f hipotética.

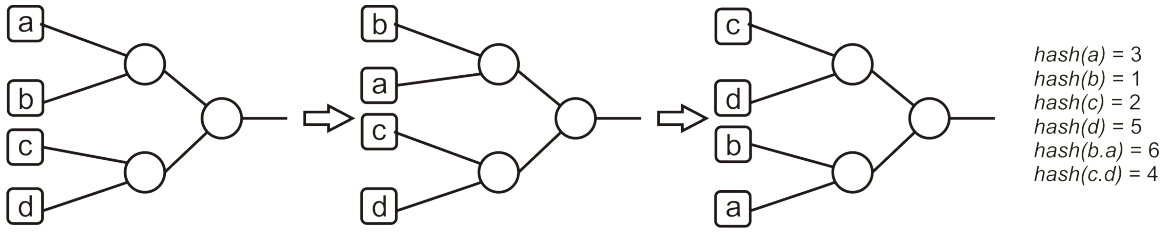

Figura 3.14. Exemplo de reordenação de variáveis para $f(a, b, c, d) = a.b.c.d$

O último subprocesso, entitulado **aplicação de teoremas**, representa a utilização direta dos teoremas da álgebra booleana em, no máximo, dois níveis. Os subprocessos já apresentados também correspondem a aplicações de teoremas, mas foram tratados separadamente por serem muito simples (propagação de constantes) ou não produzirem simplificação (reordenação de variáveis). Uma consequência da reordenação de variáveis para aplicação dos teoremas é uma visão menos ambígua das funções, e, portanto, mais fácil de serem identificadas como descrição, ou não, a mesma função. Os teoremas cuja porta lógica OR está presente, são aplicados à operação de negação

nas duas arestas incidentes e na que representa o resultado da função; uma vez que uma porta OR corresponde à uma porta AND com todas as entradas e saída negadas. A figura 3.15 mostra a aplicação de dois teoremas e os respectivos resultados.

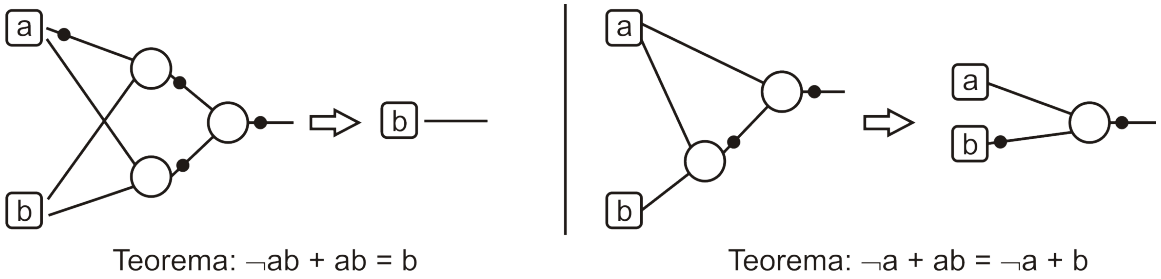

Figura 3.15. Aplicação de dois teoremas booleanos

Apesar de todos subprocessos serem independentes, a aplicação sequencial deles, mostrada nesse texto, apresentou melhores resultados e, portanto, *elSin* os executam exatamente nessa ordem. Observe que quanto mais vezes todo o processo é executado, melhores poderão ser os os resultados.

3.3.3 Síntese sequencial

Assim como a síntese combinatória, várias métricas podem ser utilizadas para definir as diretivas utilizadas durante o processo de síntese sequencial. Todos os processos apresentados na subseção 3.3.2 possuem como único objetivo minimizar o número de portas lógicas AND e NOT utilizadas para descrever o circuito. O foco da síntese sequencial realizada por *elSin* é minimizar o caminho crítico existente no circuito, aumentando assim a frequência máxima de operação do mesmo. O processo de variar a frequência através da manipulação de elementos de memória possui o nome de *retemporização*.

Sabe-se que o caminho crítico de um circuito é aquele cuja variação dos sinais na entrada demora mais tempo até ser propagado e estabilizado no circuito, gerando um resultado que possa ser confiavelmente lido, sem risco de alteração. Simplificadamente, a soma do tempo de atraso de cada elemento existente no caminho define o tempo de propagação. Para um circuito representado através de AIG só existem dois tipos de portas lógicas, AND e NOT, e arbitrariamente foi determinado tempos de atraso $2T$ e T , respectivamente, durante a análise do caminho crítico.

Para encontrar esse caminho, executa-se o mesmo procedimento de caminhamento realizado durante a síntese combinatória, mas a cada nível avançado no grafo, o vértice é marcado com o valor de atraso máximo até o momento. O caminho máximo é encontrado seguindo o caminho inverso dos vértices que representam os resultados das

funções, passando sempre por aqueles com maior valor acumulado. O menor caminho é encontrado de maneira similar, mas passando sempre por aqueles com menor valor acumulado. Determinar o caminho máximo e mínimo é importante, pois eles correspondem ao ponto de partida e destino do algoritmo – qualquer alteração que aumente o caminho máximo é descartada, e também não é possível encontrar um caminho melhor que o menor. O objetivo da síntese *elSin* é tornar o atraso em todos os caminhos o mais próximo possível do menor caminho. A figura 3.16 ilustra o procedimento descrito, e destaca o maior (pontilhado) e menor (tracejado) caminho encontrado.

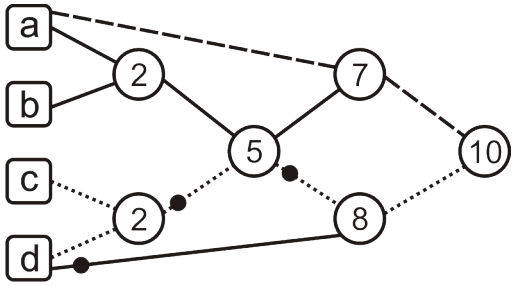

Figura 3.16. Procedimento para encontrar o caminho crítico

Uma aresta é dita *candidata a retemporização* se as funções de entrada do seu vértice de origem produzem resultados em uma mesma quantidade de *clocks*, ou seja, se o número de elementos de memória presentes no caminho da entrada até o vértice é igual. Todas as arestas candidatas representam as possíveis localizações para onde os elementos de memória podem ser movidos sem alteração no comportamento do circuito. Um elemento de memória pode ter sua posição deslocada em todo caminho cujas arestas são todas candidatas a retemporização e que, 1) se em direção às entradas, não ultrapasse nenhum vértice que contenha mais de uma saída; e 2) se em direção às saídas, não ultrapasse um vértice cuja outra entrada tenha resultado com tempo inferior. A movimentação do elemento de memória, obedecendo às restrições, faz com que as funções de saída de cada vértice continuem produzindo resultado em um mesmo *clock*. Caso o elemento siga por caminho onde uma aresta, ou mais, seja candidata, as entradas do vértice produzirão resultados em tempos diferentes dos originais, podendo alterar o resultado final. A figura 3.17 mostra as arestas candidatas (tracejado) e aquelas nas quais efetivamente podem acontecer a retemporização (pontilhado). As marcas perpendiculares as arestas representam os elementos de memória, apenas para visualização, pois no AIG os elementos de memória são transformados em variáveis de entrada e saída, como já foi visto durante a criação do AIG (ver seção 3.3.1).

A movimentação do elemento de memória é realizada sempre de maneira que não seja criado um caminho maior que o existente, que o atraso dos caminhos seja próximo

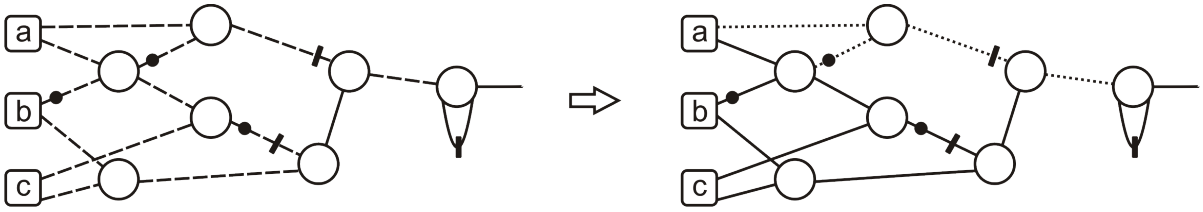

Figura 3.17. Arestas candidatas e efetivas para retemporização

da média de atrasos dos caminhos alterados, ou até que atinja a última aresta do possível caminho de movimentação. A avaliação do possível deslocamento do elemento é realizada facilmente, pois a cada movimentação, basta incrementar ou decrementar o tempo de propagação da porta no atraso acumulado, registrado no vértice.

Além da movimentação, também é possível adicionar elementos de memória de forma a dividir caminhos críticos em outros menores. O acréscimo é realizado quando o tempo no caminho crítico é maior que o tempo do segundo maior somado ao menor. Se isso acontece, é possível dividi-los de tal forma que nenhum dos caminhos é menor que o menor caminho, nem maior que o segundo maior. Caso a divisão resultasse em caminhos menores que o menor, então existiria um caminho maior e, portanto, não interferiria na frequência; e se caminhos maiores fossem criados, então a frequência seria reduzida e não atingiria o objetivo da síntese. Ao acrescentar um elemento de memória, todas as funções cujo resultado era produzido com *n clocks*, agora serão produzidas com *n+1 clocks*. Diante disto, todas as funções que produziam resultados com os mesmos *n clocks*, mas que não faziam parte do caminho dividido recebe um elemento de memória extra, fazendo com que os resultados continuem sendo produzidos com a mesma diferença de tempo existente anteriormente. A figura 3.18 exemplifica a divisão de um caminho crítico com as características apresentadas. A adição de elementos de memória só é realizada quando não é mais possível realizar as operações de movimentação.

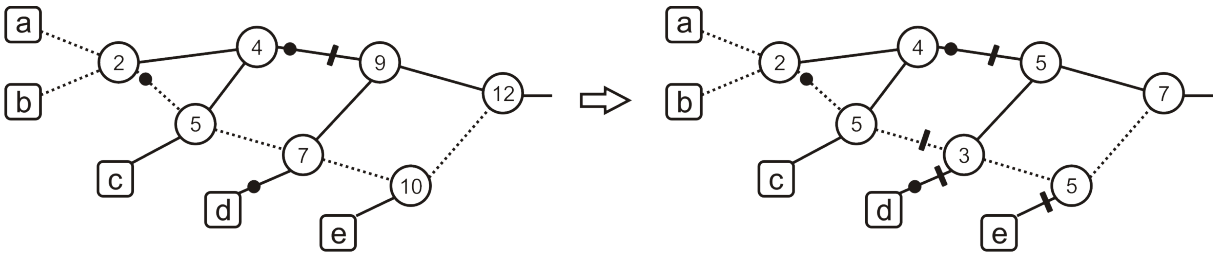

Figura 3.18. Divisão do caminho crítico de *12T* para *7T*

Ao contrário da síntese combinatória, a sequencial não apresenta melhorias se realizada mais de uma vez. Esse fato é resultado da visão global existente na síntese

sequencial, pois é possível determinar sempre o caminho crítico analisando os vértices imediatamente anteriores aos elementos de memória, uma vez que eles possuem registrado o valor acumulado. Já a síntese combinatória possui visão local, operando sempre com um conjunto de vértices de dois níveis apenas.

A síntese *elSin* consiste, portanto, na construção do AIG a partir do grafo do circuito, na execução dos três processos de síntese combinatória e nos dois de síntese sequencial - realizados exatamente nessa ordem.

Capítulo 4

A ferramenta elGen e resultados

4.1 Introdução

A intensa atividade de pesquisa em Automação de Projetos Eletrônicos (EDA) faz surgir várias ferramentas para auxiliarem o processo de automatização das tarefas. Como consequência, novos conjuntos de circuitos para serem utilizados como *benchmark*s tornam-se necessários, uma vez que os conjuntos existentes rapidamente ficam obsoletos. Baseado no tipo de projeto utilizado para criação dos circuitos, destacam-se duas classes principais de conjuntos para *benchmark*: projetos industriais e projetos sintéticos. O primeiro possui como fonte de circuitos os projetos utilizados na indústria de semicondutores, já o segundo é formado de circuitos gerados a partir da exploração de características específicas, ou, muitas vezes, aleatoriamente.

Os *benchmarks* formados por circuitos de projetos industriais carregam consigo o peso da propriedade intelectual, e por isso estão frequentemente desatualizados, uma vez que as empresas não costumam divulgar informações de seus projetos mais recentes. De outro lado, os projetos sintéticos raramente possuem informações sigilosas, mas não correspondem a circuitos utilizados comercialmente e, portanto, avaliações realizadas com esse tipo de circuito, podem levar a resultados muito diferentes de quando aplicados a circuitos integrados reais.

Uma abordagem mista para geração de circuitos para *benchmarks* consiste em criá-los sinteticamente utilizando como base projetos industriais, que resultam em conjuntos pouco limitados se comparados aos industriais e muito mais próximos da realidade frente aos sintéticos. As ferramentas que utilizam essa abordagem para geração de seus circuitos são conhecidas como geradoras de núcleos funcionais, ou em Inglês, *Cores Generators*.

A ferramenta definida neste trabalho utiliza a abordagem mista para a geração

dos conjuntos para *benchmarks*. A ferramenta em questão, e parte do título do trabalho, foi nomeada de *elGen*, acrônimo para *El Generator* - O Gerador em Espanhol.

elGen é um gerador de núcleos funcionais baseado em modelos descritos através de *elLen*, linguagem de entrada da ferramenta. Os modelos são tratados, instanciados em tamanho e formato especificados pelo usuário. Uma das partes da ferramenta consiste em realizar a síntese lógica do circuito gerado como forma de validação, uma vez que um dos objetivos da ferramenta é gerar apenas aqueles sintetizáveis. Essa etapa produz um circuito equivalente, sintetizado, contendo apenas portas lógicas AND e NOT, e transições de estado sincronizadas por um único sinal de *clock*. A outra etapa permite ao usuário visualizar o circuito criado através de uma interface gráfica bastante simples.

No decorrer deste capítulo são apresentados detalhes de funcionamento interno da ferramenta, o visualizador do circuito e modo de operação, alguns detalhes de implementação da ferramenta, bem como a comparação com os *benchmarks* existentes e análise dos resultados.

4.2 A ferramenta elGen

Os três primeiros capítulos trataram de conceitos e procedimentos necessários para a construção da ferramenta *elGen*. Nesta seção será apresentado o funcionamento interno da ferramenta, descrita através de uma abordagem *top-down*, isto é, partindo do geral para o específico. Antes de iniciar a descrição da ferramenta, é importante saber como executá-la apropriadamente. Os algoritmos da ferramenta e as respectivas explicações serão apresentados posteriormente.

4.2.1 Sintaxe de linha de comando

O primeiro passo para execução da ferramenta é a criação de um arquivo com o modelo do circuito. Para fim de exemplificação, considere a existência do arquivo *rca.ell* modelando um *Ripple Carry Adder* (apresentado na seção 2.3.1). A ferramenta é executada com a seguinte sintaxe:

```
Geral: elGen arquivoModelo modelo [--p parâmetros] --f formato [-s] [-sa] saída
Exemplo: elGen rca.ell rippleCarryAdder --p 10 --f verilog --sa rca10
```

O *arquivoModelo* espera o caminho para o arquivo com a descrição do modelo do circuito e escrito utilizando a linguagem *elLen*, no exemplo, *rca.ell*. A opção *modelo* especifica o nome do modelo principal, ou seja, o módulo topo do circuito. No exemplo, o arquivo possui dois módulos, rippleCarryAdder e somadorCompleto, sendo o primeiro

escolhido como principal. A opção *–p parâmetros* define os parâmetros de instância do modelo, no caso 10, informando que o valor do parâmetro n deve ser substituído pelo valor 10. Não são aceitos valores fracionários, e os parâmetros deverão ser passados em forma de lista, separados por vírgulas, na mesma ordem em que foram declarados no modelo. Observe que a opção –p é opcional, pois um modelo pode ser descrito sem parâmetros. A opção *–f formato* define o formato de saída do circuito gerado e estão disponíveis: bench, verilog, vhdl, eqn e blif. As opções *–s* e *–sa* são específicas para síntese, sendo –s para executar o procedimento de síntese, e –sa para executar o procedimento de síntese com adição de elementos de memória (ver seção 3.3). A opção –sa ativa automaticamente a síntese, mesmo que –s não tenha sido utilizada.

A *saída* define o prefixo para os arquivos de saída da ferramenta. Dois arquivos são gerados: o circuito gerado e o arquivo de visualização. O circuito gerado possui o nome definido por saída seguindo pela extensão específica de cada formato: .bench (BENCH), .v (VERILOG HDL), .vhd (VHDL), .eqn (EQN) e .blif (BLIF). Já o arquivo de visualização possui o nome definido por saída seguido pela extensão .elv (elGen Visualizador). Caso alguma das opções de síntese tenha sido habilitada, um arquivo com nome definido por saída seguido de .els.vhd (elGen Síntese + VHDL) é criado. Para o exemplo, serão criados os arquivos *rca10.v*, *rca10.elv* e *rca10.els.vhd*.

Durante a execução da ferramenta podem surgir erros, alguns dos quais impedem a continuação do procedimento de geração, e outros não. A descrição de cada um deles é apresentada a seguir:

- Erro 1 – Erro ao abrir o arquivo do modelo: o caminho especificado para o modelo está incorreto ou restrições de acesso impedem a abertura do arquivo. O usuário deve fornecer o caminho correto e obter as permissões de acesso adequadas.

- Erro 2 – Erro de sintaxe: o arquivo com a descrição do modelo não está bem formado. Isso significa que o arquivo viola a sintaxe da linguagem *elLen* e portanto não pode ser processado corretamente. O usuário deve verificar a descrição do modelo.

- Erro 3 – Erro de instância: o nome do módulo principal não foi encontrado na lista de módulos existentes no modelo ou os parâmetros fornecidos não correspondem com os esperados pelo módulo. O usuário deve verificar se especificou o nome corretamente ou não esqueceu algum parâmetro.

- Erro 4 – Erro de verificação: o modelo foi instanciado corretamente, mas não passou no processo de verificação. Isso significa que existe um caminho de uma ou mais entradas que não leva a uma saída, ou que uma ou mais saídas não são

alcançadas a partir de alguma entrada. O usuário deve verificar a descrição do modelo. O procedimento de geração do circuito não é interrompido.

- Erro 5 – Erro de alocação de memória: a instância do modelo é muito grande e não pode ser alocada na memória. Esse erro acontece durante a criação do grafo do circuito ou do AIG, quando o sistema operacional nega o pedido de alocação dinâmica de memória. O usuário deve escolher uma instância menor do modelo.

4.2.2 Algoritmos de elGen

Entendido como é realizada a execução da ferramenta, o próximo passo é compreender seu funcionamento interno. O algoritmo 4.1 apresenta a sequência de execução das principais rotinas realizadas pela ferramenta.

A função *processaModelo* é responsável por ler o arquivo com a descrição do modelo e criar um lista de módulos, para que estes possam ser instanciados posteriormente. Nessa função é realizado o processo de análise do arquivo e aplicação da sintaxe da linguagem *elLen*. Na seção 4.4 são apresentados detalhes desta etapa, mas neste ponto é importante saber que a cada identificação de um módulo completo (redução da regra de definição de um módulo) é criado um objeto contendo seu identificador, a lista de parâmetros, entradas, saídas e também lista de comandos. A função *instanciaModelo* é responsável por avaliar todas as expressões matemáticas simples pendentes por falta de valores dos parâmetros e executar os comandos existentes na lista de comandos do módulo. Conforme seção 2.4, a cada comando executado são criados vértices no grafo, que representa o circuito gerado. Também é nessa função que é realizada a análise da instância, conferindo se todas as saídas são alcançáveis e se todas as entradas chegam a alguma saída.

A rotina *geraCircuito* é responsável por realizar o caminhamento no grafo do circuito e substituir cada vértice e aresta por elementos equivalentes na HDL escolhida para que o circuito seja gerado.

As funções de síntese somente são executadas se desejado pelo usuário da ferramenta. Caso essa condição seja verdadeira, o primeiro passo é transformar o grafo do circuito em um AIG. Essa tarefa é realizada pela função *transformaEmAIG*. Essa transformação é necessária pois, conforme seção 3.3, a estrutura de dados utilizada para representar o circuito e realizar as rotinas de síntese foi o AIG. As funções *sinteseCombinatoria*, *sinteseSequencialMov* e *sinteseSequencialAdd* serão apresentadas posteriormente.

A função *geraSintetizado* é responsável por caminhar no AIG e mapear os vértices AND e as arestas marcadas por inversores e/ou elementos de memória em portas lógicas

```
1  Algoritmo: elGen
   Data: Modelo, tamanho da instância, formato do circuito, e outros
         parâmetros.
   Result: Circuito gerado
2  elGen(arquivoModelo,moduloPrincipal,parametros,formato) begin
       // Processa o arquivo com o modelo e obtém a lista de modelos
3      listaModulos ← processaModelo(arquivoModelo);
       // Cria o grafo do circuito baseado nos parâmetros
4      gCircuito ←
       instanciaModelo(listaModulos,moduloPrincipal,parametros);
       // Gera o arquivo com a descrição do circuito na HDL
          escolhida
5      arquivoCircuito ← geraCircuito(gCircuito,formato);
6      if sintetizar then
           // Cria o AIG baseado no grafo do circuito
7          aig ← transformaEmAIG(gCircuito);
           // Realiza a síntese combinatória e sequencial baseada
              apenas na movimentação dos elementos de memória
8          aig ← sinteseCombinatoria(aig);
9          aig ← sinteseSequencialMov(aig);
10         if sinteseSeqAdd then
               // Adição de elementos de memória
11             aig ← sinteseSequencialAdd(aig);
12         end
           // Gera o arquivo com o circuito sintetizado
13         arquivoSintetizado ← geraSintetizado(aig);
14     end
       // Gera o arquivo de visualização do circuito
15     arquivoVisualizacao ← geraVisualizacao(gCircuito);
16 end
```

Algoritmo 4.1: Algoritmo principal da ferramenta *elGen*

AND ou NOT e/ou transições de sinais sincronizadas por um sinal de *clock*, todos descritos em VHDL. A função *geraVisualizacao* caminha no grafo do circuito e cria o arquivo de visualização, para ser apresentado pelo visualizador (seção 4.3).

O algoritmo 4.2 descreve a rotina de síntese combinatória, realizada por *propagacaoConstantes*, *reordenaVariaveis* e *aplicaTeoremas*, executadas a cada vértice à medida em que é realizado o caminhamento no AIG.

A função *propagacaoConstantes*, esquematizada no algoritmo 4.3, tenta realizar o primeiro processo de simplificação apresentado na seção 3.3, que consiste na substituição de três vértices (sendo um deles representação de uma constante) por um único equivalente. O primeiro teste feito pela função verifica se algum dos vértices de origem

```
1  Algoritmo: sinteseCombinatoria
   Data: AIG do circuito
   Result: AIG simplificado, resultado da síntese combinatória
2  sinteseCombinatoria(aig) begin
       // Desmarca todos os vértices como visitados
3      aig ← caminhaDesmarcaVertices(aig);
       // Percorre todo AIG por largura
4      while (prox ← proximoVertice(aig)) ≠ NULO do
           // Faz propagação de constante
5          aig ← propagacaoConstantes(prox,aig);
           // Faz reordenação de variáveis
6          aig ← reordenaVariaveis(prox,aig);
           // Aplica os teoremas booleanos
7          aig ← aplicaTeoremas(prox,aig);
8      end
9      return aig ;
10 end
```

Algoritmo 4.2: Algoritmo de síntese combinatória

do vértice atual é uma constante, caso não seja, não há como realizar nenhuma simplificação. A rotina *aplicaPropagacao* realiza o teste de qual das quatro possibilidades de simplificação pode ser aplicada, ilustrada na figura 3.13, e aplica a adequada. Note que caso uma das funções seja constante é sempre possível simplificá-la.

```
1  Algoritmo: Propagação de Constantes
   Data: Vértice atual e AIG do circuito
   Result: AIG simplificado, resultado do processo de síntese combinatória
2  propagacaoConstantes(vertice,aig) begin
3      antecessorEsq ← left(vertice);
4      antecessorDir ← right(vertice);
5      if não constante(antecessorDir) e não constante(antecessorEsq) then
           // Não há como propagar constantes
6          return aig;
7      end
8      aig ← aplicaPropagacao(antecessorDir,antecessorEsq,aig);
9      return aig;
10 end
```

Algoritmo 4.3: Algoritmo de síntese combinatória - Propapação de constantes

A função *reordenaVariaveis*, algoritmo 4.4, simplesmente tenta diminuir a quantidade de representações diferentes para uma mesma função, garantindo que o resultado do *hash* aplicado à função de origem da esquerda seja sempre menor que a da direita.

A rotina *defineAntecessores* simplesmente troca a ordem dos vértices, pois o segundo parâmetro espera o vértice esquerdo e o terceiro o vértice direito, e ela é chamada com os vértices trocados.

```
1 Algoritmo: Reordenação de variáveis
  Data: Vértice atual e AIG do circuito
  Result: AIG simplificado, resultado do processo de síntese combinatória
2 reordenaVariaveis(vertice,aig) begin
      // Obtem os vértices antessores ao vértice atual
3     antecessorEsq ← left(vertice);
4     antecessorDir ← right(vertice);
5     if hash(antecessorEsq) < hash(antecessorDir) then
          // Troca os vértices
6         aig ← defineAntecessores(vertice,antecessorDir,antecessorEsq);
7     end
8     return aig;
9 end
```

Algoritmo 4.4: Algoritmo de síntese combinatória - Reordenação de variáveis

A última função da síntese combinatória é *aplicaTeoremas*, algoritmo 4.5, e é responsável por testar e, em caso de possibilidade, aplicar o teorema que simplifica a função. Como não é possível realizar nenhuma simplificação se todas as quatro variáveis de entrada forem diferentes, esse é o primeiro teste realizado. Não é garantido de ser possível aplicar algum teorema, mas caso seja possível, ele é aplicado e os demais não são testados. Assim como o primeiro processo, os testes com portas lógicas OR são realizados invertendo as entradas e saídas do vértice, transformando a porta AND em uma OR.

O próximo passo é a realização da síntese sequencial. O primeiro processo consiste em movimentar os elementos de memória de forma a reduzir os maiores caminhos de dados do circuito. Esse procedimento tem como objetivo aumentar a frequência máxima na qual o circuito pode operar. Como a síntese realizada pela ferramenta não está ligada ao mapeamento tecnológico, não são utilizados tempos reais de propagação dos sinais pelas portas lógicas, apenas a relação de 2 para 1 entre as portas AND e NOT. O algoritmo 4.6 descreve o primeiro processo.

Conforme é visto na seção 4.4, cada vértice do AIG possui um atributo responsável por armazenar o custo para alcançá-lo, ou seja, o maior caminho de uma entrada até aquele ponto. A etapa inicial do algoritmo é obter a lista de elementos de memória, facilmente obtida, pois durante a criação do AIG, cada elemento de memória encontrado é transformado em uma variável de entrada da função do circuito. A todo o momento

```
1  Algoritmo: Aplicação de teoremas
   Data: Vértice atual e AIG do circuito
   Result: AIG simplificado, resultado do processo de síntese combinatória
2  aplicaTeoremas(vertice,aig) begin
3      antecessorEsq ← left(vertice);
4      antecessorDir ← right(vertice);
       // Se nenhuma variável é igual, então não há como simplificar
5      if (hash(left(antecessorEsq)) ≠ hash(right(antecessorEsq))) e
       (hash(left(antecessorEsq)) ≠ hash(left(antecessorDir))) e
       (hash(left(antecessorEsq)) ≠ hash(right(antecessorDir))) e
       (hash(right(antecessorEsq)) ≠ hash(left(antecessorDir))) e
       (hash(right(antecessorEsq)) ≠ hash(right(antecessorDir))) e
       (hash(left(antecessorDir)) ≠ hash(right(antecessorDir))) then
6          return aig;
7      end
       // Testa os teoremas
8      foreach t em listaTeoremas do
9          viavel ← testaTeorema(t,vertice);
10         if viavel then
11             aig ← aplicaTeorema(t,vertice,aig);
12             return aig;
13         end
14     end
       // Se chegou até aqui é porque não aplicou nenhum teorema
15     return aig;
16 end
```

Algoritmo 4.5: Algoritmo de síntese combinatória - Aplicação de teoremas

é possível identificar onde está cada elemento de memória e a qual variável ele se refere. De posse dessa lista, é realizada a ordenação desses elementos, de forma que a aresta destino que contém o vértice de maior custo seja a primeira da lista. A movimentação é realizada sempre para a esquerda, ou seja, em direção as entradas. A rotina *testaMovimentoEsquerda* é responsável por avaliar se a movimentação é viável, ou seja, se o elemento está deslocando apenas em arestas candidatas a retemporização, e se o caminho a direita não ficou maior que caminho crítico atual. Caso a movimentação seja viável, o elemento é removido da aresta atual e acrescentado às duas arestas incidentes ao vértice atual.

Nem sempre o maior caminho permite uma retemporização inicial, pois nem sempre ele possui arestas candidatas, sendo necessário retemporizar outros caminhos ligados a ele primeiro. Esse fato não se torna um problema, pois a cada retemporização, uma nova lista é obtida e ordenada, ou seja, o maior caminho aparecerá novamente nas

```
1  Algoritmo: Movimentação de elementos de memória
   Data: AIG do circuito
   Result: AIG retemporizado, resultado do processo de síntese sequencial
2  sinteseSequencialMov(aig) begin
3      continua ← Verdadeiro;
4      while continua do
           // Obtem a lista de elementos de memória
5          listaRegs ← obtemListaRegs(aig);
           // Ordena descendentemente por tamanho do caminho
6          listaRegs ← ordenaListaMaiorMenor(listaRegs);
7          foreach v em listaRegs do
8              melhorou ← testaMovimentacaoEsquerda(v,aig);
9              if melhorou then
                   // Move para esquerda
10                 aig ← removeElementoMemoria(v,aig);
11                 aig ← adicionaElementoMemoria(left(v),aig);
12                 aig ← adicionaElementoMemoria(right(v),aig);
13                 continua While;
14             end
               // Não melhorou, então tenta o próximo
15             if v = ultimo(listaRegs) then
16                 continua ← Falso;
17             end
18         end
19     end
20     return aig;
21 end
```

Algoritmo 4.6: Algoritmo de síntese sequencial - Movimentação

outras iterações do algoritmo. O processo inteiro termina quando não é mais possível movimentar os elementos.

O último processo de síntese sequencial é responsável por dividir o caminho crítico. Suponha que após a movimentação ainda exista um caminho que é maior que o segundo maior caminho somado ao menor. O algoritmo 4.7 apresenta o processo de divisão desse caminho crítico.

Será visto na seção 4.4 que cada vértice possui a informação de quantos elementos de memória existem desde as entradas do circuito até o vértice, ou seja, em quantos *clocks* a saída da função representada pelo vértice estará correta. Essa informação é crucial para a realização desse processo de síntese. Ao dividir um caminho crítico em dois, é acrescentado um elemento de memória a ele. Ao realizar esse procedimento, o resultado que estaria disponível corretamente em n *clocks* passa a estar disponível em

```
1  Algoritmo: Adição de elementos de memória
   Data: AIG do circuito
   Result: AIG retemporizado, resultado do processo de síntese sequencial
2  sinteseSequencialAdd(aig) begin
3      continua ← Verdadeiro;
4      while continua do
           // Obtem a lista de elementos de memória
5          listaRegs ← obtemListaRegs(aig);
           // Ordena descendentemente por tamanho do caminho
6          listaRegs ← ordenaListaMaiorMenor(listaRegs);
           // Verifica se é possível quebrar caminho crítico
7          if tamanho(ultimo(listaRegs)) > (tamanho(penultimo(listaRegs) +
           tamanho(primeiro(listaRegs)))) then
               // Quebra o caminho crítico
8              v ← encontraPontoQuebra(ultimo(listaRegs));
9              tempoResultado ← obtemNClocks(v);
10             aig ← adicionaElementoMemoria(left(v),aig);
11             aig ← adicionaElementoMemoria(right(v),aig);
               // Crio elementos em todos vértices no caminho com
                  resultado no mesmo clock
12             aig ← incrementaClockAtual(v,aig);
13             continua ← Verdadeiro;
14             continua While;
15         end
16         continua ← Falso;
17     end
18     return aig;
19 end
```

Algoritmo 4.7: Algoritmo de síntese sequencial - Criação

n+1 clocks. Para evitar que as funções operem sobre dados inválidos, são crescentados elementos de memória em todas as arestas ligadas aos vértices que fazem parte do caminho a partir do ponto de divisão e que seus vértices anteriores não tenham sido visitados ainda. Isso garante que o dado está disponível para as funções no caminho 1 *clock* depois, garantido o correto resultado.

É importante perceber que o resultado desse último processo, ao contrário da movimentação, altera o número total de *clocks* no qual o resultado efetivo do circuito estará disponível. Por esse motivo, a realização desse processo de síntese sequencial é opcional, de acordo com o deseja do usuário. Após realização de todas as etapas de síntese tem-se um AIG simplificado, disponível para ser caminhado e gerado o arquivo VHDL.

4.3 Visualizador do circuito

Um dos objetivos do trabalho apresentado é permitir ao usuário visualizar o circuito gerado pela ferramenta, fornecendo assim uma representação gráfica do mesmo. Para tal, *elGen* vem acompanhado de um visualizador bastante simples, mas que permite visualização gráfica dos sinais e componentes existentes no circuito. A figura 4.1 apresenta a interface do visualizador, sem nenhum circuito aberto e com as principais áreas assinaladas.

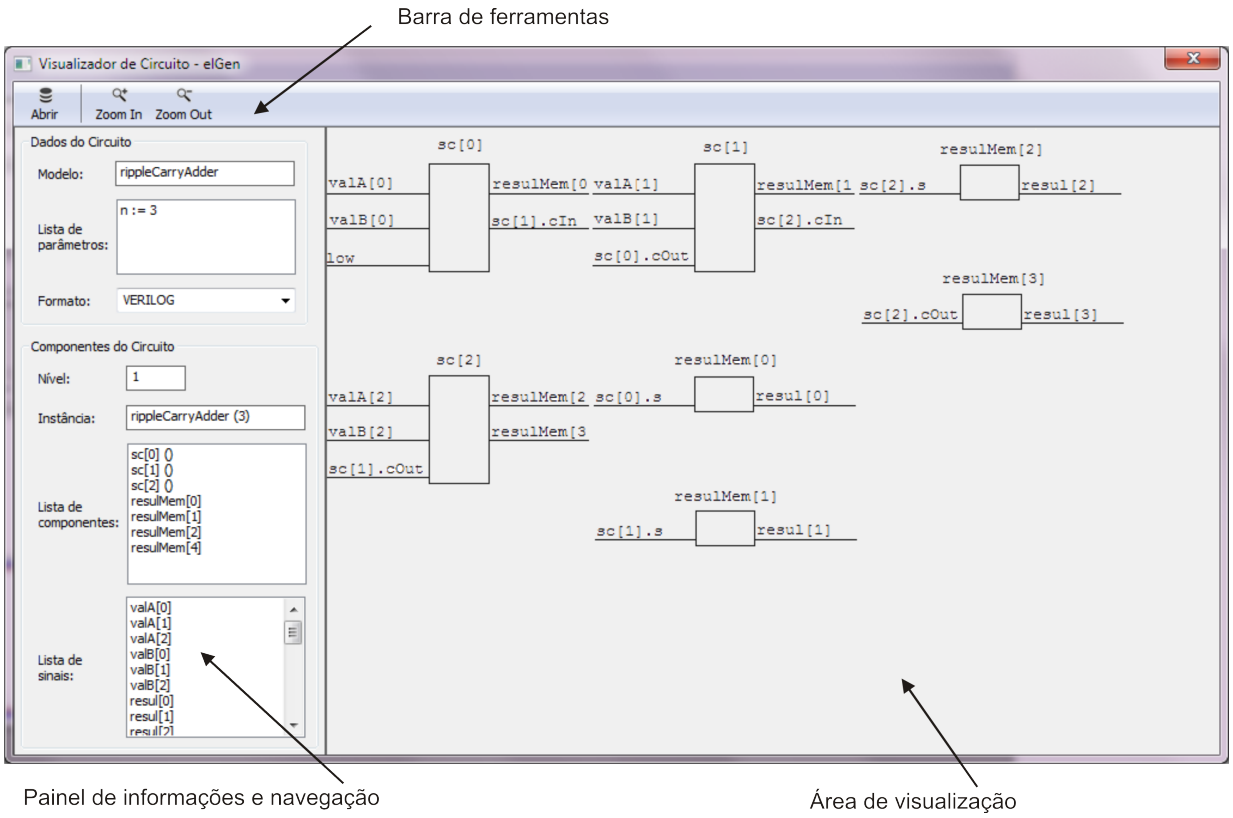

Figura 4.1. Tela do visualizador do circuito gerado

A **barra de ferramentas** possui botões de acesso rápido às funcionalidades de abrir o circuito a ser visualizado (Abrir), entrar em um componente (Zoom In) ou sair de um componente (Zoom Out). A opção *Zoom In* só está disponível quando o componente selecionado possui um modelo associado, ou seja, ele é uma instância de um submodelo. A opção de *Zoom Out* está sempre disponível, desde que não esteja no nível mais alto dos componentes, o primeiro nível, instância do modelo principal.

O **painel de informações e navegação** possui duas sub-áreas: *Dados do circuito* e *Componentes do circuito*. Na sub-área de dados do circuito são fornecidas as informações do modelo principal que está sendo visualizado, a lista de parâmetros e

seus respectivos valores utilizados na instância do modelo e o formato no qual o circuito foi gerado. Todas as informações dessa sub-área são carregadas ao abrir o arquivo de visualização e não podem ser alteradas. Na sub-área de componentes do circuito estão listados os componentes e os sinais pertencentes à instância atualmente selecionada. Além dessas informações, o nome da instância e o nível de visualização também são fornecidos. Todo componente da lista de componentes pode ser visualizado internamente, desde que o mesmo não seja uma porta lógica básica. Quando um componente é selecionado, ele ganha foco na visualização e sua cor é alterada para vermelho. O mesmo acontece com um sinal, mas a cor é alterada para azul.

A **área de visualização** é a região onde os componentes são desenhados para serem visualizados graficamente pelo usuário. Todo componente, seja porta lógica básica ou instância de um modelo, possui a forma geométrica de um retângulo. Na face esquerda são desenhados todos os sinais de entrada do componente, e na direita, todos os sinais de saída, dispostos exatamente na mesma ordem na qual foram declarados. Os componentes possuem dimensões que variam conforme o número de entradas e saídas, sendo que a largura é sempre de 50 *pixels*, e a distância entre as entradas, ou saídas, é de 20 *pixels*, e há sempre uma margem de 5 *pixels* entre a face superior e o primeiro sinal, e 5 *pixels* entre a face inferior e o último sinal. Cada sinal ligado é identificado com até 10 caracteres utilizando fonte *Courier New* 10, de forma a manter a largura total do componente em 210 *pixels*. O nome da instância aparece com a mesma fonte, mas limitado a 20 caracteres, sublinhado e centralizado, distante 10 *pixels* da face superior. Todos os componentes são desenhados do canto superior esquerdo para o canto inferior direito, mantendo um espaçamento vertical e horizontal de 40 *pixels* entre eles, tentando manter a área total de visualização mais próxima possível de um quadrado.

4.4 Detalhes de implementação

A ferramenta *elGen* foi implementada utilizando a linguagem de programação C++ e o paradigma de programação orientada a objetos. Essa linguagem foi escolhida devido a familiaridade do autor com a mesma, por existirem compiladores para os sistemas operacionais mais comuns e pela integração com a biblioteca gráfica utilizada para a construção do visualizador.

O algoritmo 4.1 apresenta a função *processaModulo*, cujo objetivo é analisar o arquivo de entrada e retornar a lista de módulos existentes. A leitura do arquivo é auxiliada por duas ferramentas: Lex [Lesk & Schmidt, 1978] e Yacc [Johnson, 1979].

Ambas são utilizadas em conjunto para realizar a varredura do arquivo com a descrição do modelo do circuito, escrito através de *elLen*. O objetivo de utilizar essas ferramentas é facilitar o trabalho de agrupamento de caracteres em *tokens* e o casamento destes com a sintaxe da linguagem. O processamento do modelo é realizado durante a redução[1] , onde a lista dos parâmetros, entradas, saídas e comandos, bem como o identificador do módulo já foram coletados.

Um módulo é armazenado através de um objeto da classe *CModulo* e possui os seguintes membros: identificador, lista de parâmetros, lista de entradas, lista de saídas e lista de comandos. Ao reduzir uma regra de parâmetros, entradas ou saídas, ele é adicionado a sua respectiva lista. As entradas e saídas são compostas por um par ordenado (identificador, expressão matemática), uma vez que seu valor efetivo depende do valor do parâmetro. A lista de comandos também é composta por um par ordenado (sinal destino, sinal origem) representados os pares de sinais participantes da ligação. Os sinais, por sua vez, também são pares ordenados (identificador, expressão matemática). A última redução a acontecer é aquela que identifica o módulo, e neste ponto o módulo recebe o seu identificador e é adicionado à lista de módulos, que será utilizada durante as instanciações. Toda expressão matemática é um objeto da classe *CExpressaoMatematica* que contém um expressão. A avaliação é realizada após o fornecimento dos valores das variáveis da expressão, ou seja, após os valores dos parâmetros da instância do módulo.

A função *instanciaModelo* consiste na criação do grafo do circuito baseado no modelo e os parâmetros utilizados para a instância. O grafo é construído do módulo principal, aquele que é primeiramente declarado no arquivo do modelo, para os sub-módulos, que são instâncias do módulo principal e de outros módulos. Os valores dos parâmetros para a instância foram especificados ao chamar a ferramenta e devem corresponder exatamente ao número de parâmetros do módulo principal, e os valores são preenchidos na mesma ordem com que foram passados. A primeira instância corresponde à instância do módulo principal, que, por construção, sempre está na lista de módulos. Os valores dos parâmetros são preenchidos e todas as expressões matemáticas que aguardavam esses valores podem ser avaliadas. Fornecidos os valores e as expressões avaliadas, os comandos são processados um a um e transformados em vértices e arestas do grafo, conforme visto na seção 2.4.

O AIG é criado através da função *transformaEmAIG*. Essa função recebe o grafo do circuito e transforma-o em um AIG. Por definição, um AIG é um grafo direcionado, mas por razões de otimização no tempo de execução dos algoritmos de síntese, ele é

[1] Redução: Processo realizado durante o casamento de um conjunto de símbolos com uma regra gramatical.

implementado como um grafo direcionado, mas existe uma aresta auxiliar, responsável por permitir o caminhamento no sentido inverso para atualização dos valores de alguns atributos dos vértices. Um AIG é um objeto da classe *CAIG* e possui os seguintes membros principais: *tabela hash* e *lista de variáveis*. A primeira é responsável por armazenar os vértices do AIG, todos objetos da classe *CNodoAIG*, e a segunda é responsável por armazenar os apontadores para os vértices que representam as variáveis da função. A existência dessa lista permite aos algoritmos de síntese a rápida recuperação dos elementos de memória existentes no circuito.

Cada objeto da classe *CNodoAIG* possui alguns atributos: *arestaDireita*, *arestaEsquerda*, *lista de arestas de saída*, *valorAcumulado*, *nClocksResultado*. Os três primeiros são responsáveis por armazenar os apontadores para as arestas ligadas ao vértice; o *valorAcumulado* possui a informação do atraso existente até o vértice, ou seja, a soma dos tempos das portas lógicas atravessadas até a posição atual no circuito; e o *nClocksResultado* contém a informação de quantos elementos de memória existem no caminho desde a entrada até o vértice atual.

O visualizador também foi implementado utilizando C++ como linguagem de programação, e a biblioteca wxWidgets 2.8.11 [Smart, 1992] como biblioteca gráfica. A escolha dessa biblioteca foi feita devido a disponibilidade de implementação da mesma para Linux, Windows e Mac OS. Isso permite que a ferramenta possa ser compilada em cada um dos sistemas operacionais mencionados, nem necessidade de realização de porte na camada de interface.

4.5 Resultados

Nesta seção são apresentados os resultados obtidos utilizando a ferramenta *elGen* para geração dos circuitos e a comparação com os *benchmarks* existentes. Todos os experimentos foram realizados em um computador com processador Core 2 Duo 3,0 GHz com 2GB de memória RAM e sistema operacional Linux Ubuntu 10.4 de 32 *bits*.

A tabela 4.1 apresenta os circuitos combinatórios gerados pela ferramenta, classe do circuito e número de *bits* máximo no qual foi possível gerar o circuito. Para obtenção do número de *bits* máximo, foi criado um *script* com a tarefa de rodar automaticamente a ferramenta para cada um dos modelos, incrementando em 1 o número de bits, e parando quando um erro de qualquer tipo aconteça. Todos os erros ocorridos foram o erro 5 (Erro de alocação de memória). Os circuitos gerados também foram sintetizados com a opção –s da ferramenta.

A tabela 4.2 é semelhante a tabela 4.1, mas apresenta os resultados para os

Nome do circuito	Classe	Núm. Circuitos
Ripple Carry Adder	Somador	100.093
Ripple Carry Adder/Subtractor	Somador/Subtrador	100.093
Carry Lookahead	Somador	480
Carry Skip Adder	Somador	98.945
Carry Select Adder	Somador	98.833
Carry Save Adder	Somador	101.164
Carry Lookahead Block	Somador	99.436
Dadda Tree	Multiplicador	501
Reduced Tree	Multiplicador	486
Wallace Tree	Multiplicador	628
Array	Multiplicador	699
Carry-Save	Multiplicador	100.304
Non Restoring Celular Array Divider	Divisor	904
Restoring Celular Array Divider	Divisor	803
CarryLookahead Array Divider	Divisor	608
Barrel Shifter	Deslocador	5
Decoder	Básico	124.569
Encoder	Básico	123.453
Priority Encoder	Básico	111.145
Multiplexer	Básico	100.345
Demultiplexer	Básico	100.345
Parity Circuit	Básico	109.447
Magnitude Comparator	Básico	87.102

Tabela 4.1. Tabela de circuitos combinatórios gerados

circuitos sequenciais gerados pela ferramenta. A obtenção dos valores foi realizada da mesma maneira.

Com objetivo de comparar os circuitos gerados por elGen com aqueles gerados por BenCGen e Eudoxus, as tabelas 4.3 e 4.4 apresentam as mesmas informações daquelas apresentadas anteriormente.

Alguns circuitos gerados por Eudoxus não estão disponíveis em toda a faixa de valores de *bits*. O gerador Arithmetic Module Generator não fornece possibilidade de verificação dos tamanhos máximos de circuitos gerados, isso porque para cada circuito solicitado, um formulário no sítio tem que ser preenchido, inviabilizando o levantamento dessa informação.

É importante perceber que Eudoxus apresenta uma quantidade muito grande de tipos de circuitos, 31 no total, e também a maior diversidade de classes de circuitos. Tanto elGen quanto BenCGen geram apenas circuitos aritméticos da classe de somadores, subtradores, multiplicadores e divisores, enquanto Eudoxus , além deles, cria raizes quadradras, raizes e operações de complemento de 1 e 2.

Nome do circuito	Classe	Núm. Circuitos
ADD-AND-SHIFT	Multiplicador	23.341
Indirect Multiplication	Multiplicador	12.730
Roberson Signed Number Multiplication	Multiplicador	7.991
Booth Algorithm	Multiplicador	5.102
Canonical Multiplier Recording	Multiplicador	1.031
Subtract-And-Shift	Divisor	19.345
Binary Restoring Division	Divisor	18.042
Binary Non-restoring Division	Divisor	10.085
High-Radix Division	Divisor	2.504
SRT Division	Divisor	2.403
Robertson High-Radix Division	Divisor	1.302
Convergence Division	Divisor	1.405

Tabela 4.2. Tabela de circuitos sequenciais gerados

Embora Eudoxus possua a maior diversidade de circuitos, o número efetivo de circuitos gerados, 15.688, está muito aquém dos 1.511.504 gerados por BenCGen e 1.565.669 por elGen. Outra aspecto muito importante é que a diferença entre o número de circuitos de BenCGen e elGen é de menos de 4%, e que para os circuitos gerados por ambas as ferramentas, BenCGen é capaz de gerar uma quantidade maior de circuitos. A diferença existente na capacidade de geração dos circuitos é devido ao modo de produção deles. BenCGen possui o algoritmo para cada circuito implementado diretamente na ferramenta, sem necessidade de pré-processamento, enquanto elGen precisa ler o modelo, processá-lo, criar sua representação em grafo, para então gerar o circuito. Esse trabalho adicional implica em maior tempo e gasto de memória.

Além da informação de tipos e tamanhos dos circuitos gerados, é importante uma comparação qualitativa dos formatos nos quais cada um dos *benchmarks* disponibiliza seus circuitos. A tabela 4.5 apresenta resumidamente essa comparação. Nessa tabela também foram inclusos os conjuntos de *benchmarks* não originados de geradores de núcleos funcionais.

4.6 Análise dos resultados

De acordo com os resultados apresentados na seção 4.5, podem ser realizadas quatro análises a respeito da ferramenta *elGen* em comparação com as demais existentes. A primeira refere-se à quantidade de circuitos disponibilizados. Todos os *benchmarks* não originados de ferramentas de geração de núcleos funcionais, por mais que possuam uma grande quantidade de circuitos, são consideravelmente menores. O maior conjunto é

Nome do circuito	Classe	Núm. Circuitos
Array	Multiplicador	704
Carry LookAhead	Multiplicador	512
Reduced Tree	Multiplicador	704
Dadda Tree	Multiplicador	704
Wallace Tree	Multiplicador	704
Ripple Carry	Somador	102.400
Carry LookAhead	Somador	1.408
Block Carry LookAhead	Somador	102.400
Carry Save	Somador	102.400
Carry Skip	Somador	102.400
Carry Select	Somador	102.400
Ripple Carry	Somador/Subtrador	102.400
Barrel Shifter	Shifter	16
Non Restoring	Divisor	1.280
Restoring	Divisor	1.024
Carry LookAhead	Divisor	704
Multiplexador	Básico	131.072
Demultiplexador	Básico	131.072
Decodificador Binário	Básico	131.072
Codificador Binário	Básico	131.072
Codificador de Prioridade	Básico	131.072
Comparador de Magniture	Básico	102.400
Circuito de Paridade	Básico	131.072
Unidade Lógica e Aritmética	Aritmético	512

Tabela 4.3. Tabela de circuitos gerados por BenCGen

oferecido por IWLS 2005, com 84 circuitos. Todos os geradores, considerando apenas um único tipo de circuito, são capazes de criar um conjunto consideravelmente maior. Se comparados apenas a classe dos geradores, a quantidade de circuitos é bastante grande: BenCGen é capaz de gerar 1.511.504 circuitos diferentes [Andrade, 2008], Eudoxus 15.688 circuitos [Bakalis et al., 2001], e *elGen* 1.565.669 circuitos. O gerador Arithmetic Module Generator não possui um método de geração instantânea disponível para o usuário, sendo necessário preencher um formulário no sítio e solicitar um circuito específico, dificultando a contagem dos circuitos possíveis de serem gerados.

A segunda análise diz respeito ao formato no qual os circuitos são disponibilizados. Foi apresentado nas seções 1.1 e 1.2 que uma das maiores deficiências dos *benchmarks* disponíveis trata-se da quantidade limitada de formatos nos quais os circuitos são disponibilizados. A tabela 4.5 apresenta de maneira compacta os *benchmarks* e os formatos nos quais o usuário pode encontrar o circuito descrito. A grande maioria deles fornece os circuitos em dois ou três formatos, sendo alguns disponíveis em apenas

Nome do circuito	Classe	Núm. Circuitos
Ripple Carry Adder/Subtracter	Somador/Subtrador	1.024
Group Carry Look-Ahead Adder/Subtracter	Somador/Subtrador	1.024
Ripple Carry Adder	Somador	1.024
Ripple Carry Subtracter	Subtrador	1.024
Group Carry Look-Ahead Adder	Somador	1.024
Group Carry Look-Ahead Subtracter	Subtrador	1.024
Multilevel Carry Look-Ahead Adder	Somador	256
Ladner-Fischer Parallel Prefix Adder	Somador	1.024
Kogge-Stone Parallel Prefix Adder	Somador	1.024
Carry Propagate Array Multiplier	Multiplicador	256
Carry Save Array Multiplier	Multiplicador	256
Trisection Pezaris Array Multiplier	Multiplicador	256
Baugh-Wooley Array Multiplier	Multiplicador	256
Modified Booth Multiplier	Multiplicador	256
Restoring Cellular Array Divider	Divisor	64
Non-Restoring Cellular Array Divider	Divisor	64
Non-Restoring Fractional Square Rooter	Raiz quadrada	256
Non-Restoring Fractional Squarer	Raiz	256
Multiplexer Based Shifter	Deslocador	256
Unsigned Serial/Parallel Multiplier	Multiplicador	256
2s complement Serial/Parallel Multiplier	Multiplicador	256
Serial Adder	Somador	1.024
Serial Column Adder	Somador	1.024
Serial 1s complement	Complemento	1.024
Serial 2s complement	Complemento	1.024
Serial Multiplier	Multiplicador	1.024
Serial squarer	Raiz	1.024
Linear Feedback Shift Registers (LFSRs)	Gerador padrão	256
Linear Hybrid Cellular Automata (LHCA)	Gerador padrão	256
Accumulator - Based	Gerador padrão	1.024
Rotate Carry Accumulator	Gerador padrão	1.024
Multiple Input Shift Registers (MISRs)	Analisador resposta	256

Tabela 4.4. Tabela de circuitos gerados por Eudoxus

um único formato. Apesar de alguns disponibilizarem os circuitos em mais de um formato, nem todos os circuitos estão disponíveis em todos esses formatos, com exceção do gerados, que fornecem em todos os formatos anunciados. Comparando *elGen* com BenCGen, ambos possuem relação n x m, qualquer um dos n circuitos em qualquer um dos m formatos, mas apenas *elGen* disponibiliza os circuitos em VHDL. O formato CNF é encontrado apenas em Velev, e nenhum dos geradores oferta os circuitos nesse formato.

benchmark	BENCH	BLIF	VERILOG	VHDL	EQN	CNF	Outros
ISCAS 85	√	⊗	⊗	⊗	⊗	⊗	⊗
ISCAS 89	√	⊗	⊗	⊗	⊗	⊗	⊗
Politecnico di Torino	⊗	⊗	⊗	√	⊗	⊗	⊗
ITC 99	⊗	⊗	√	⊗	⊗	⊗	⋆
Velev	⋆	⋆	⊗	⊗	⊗	√	⊗
IWLS 2005	⊗	⊗	√	⊗	⊗	⊗	√
OpenCores	⊗	⊗	⋆	⋆	⊗	⊗	⋆
Free Model Foundry	⊗	⊗	⋆	⋆	⊗	⊗	⋆
Arithmetic Module Generator	⊗	⊗	√	√	⊗	⊗	⊗
Eudoxus	⊗	⊗	√	√	⊗	⊗	⊗
BenCGen	√	√	√	⊗	√	⊗	⊗
elGen	√	√	√	√	√	⊗	⊗

L egenda: ⊗ Nenhum - √ Todos - ⋆ Alguns

Tabela 4.5. Formato de distribuição dos circuitos por *benchmark*

A terceira trata da corretude dos circuitos. A análise desse fator é uma tarefa extremamente complexa, uma vez que estar correto diz respeito a representar exatamente sua especificação, e para a maior parte dos circuitos, a especificação não se encontra disponível. Muitos *benchmarks* foram verificados através do uso, e como não foram encontrados erros, são considerados corretos. Aqueles disponibilizados em sítios da Internet passam por processo contínuo de modificação e testes, e são considerados corretos até que se encontre um erro.

BenCGen utiliza um processo de verificação formal para os circuitos gerados, através de resolvedores SAT. Alguns circuitos gerados de cada tipo foram comparados a outros garantidamente corretos, chamados de *gold reference.* Como não foram encontradas diferenças funcionais, ou seja, os circuitos gerados e suas respectivas *gold referentes* implementam a mesma função booleana, eles são considerados corretos. Observe que por mais formal que seja a verificação, não há garantias de todos os circuitos estarem corretos, pois não há referências garantidamente corretas para todos. Já *elGen* apresenta uma forma diferente de garantir a corretude de seus circuitos. Um circuito é correto se representa exatamente sua especificação. A especificação do circuito é uma tripla (modelo, parâmetros, formato), onde o modelo é descrito através da linguagem *elLen*, os parâmetros representam os valores para a instância do modelo, e o formato corresponde à uma HDL. Já foi mostrado que todo comando presente no modelo é processado, e que existe um mapeamento direto do grafo do circuito para a HDL, portanto, se a ferramenta está correta, o circuito estará correto se implementa sua especificação. A etapa de transformação de uma especificação em alto nível para um modelo não é

contemplada, logo, se o modelo está corretamente descrito através de *elLen* e o circuito é gerado a partir do modelo, o circuito é considerado correto.

A quarta análise, assim como a terceira, trata de um aspecto de qualidade, nomeado extensibilidade. Esse fator, no contexto de conjunto de circuitos para *benchmark*, refere-se à capacidade de incrementar, ou estender, o conjunto. Para todos os *benchmarks* é sempre possível aumentar o conjunto, bastando acrescentar mais um circuito, portanto, é útil analisar esse fator apenas para os geradores de núcleos funcionais. Todos os geradores são extensíveis modificando o código-fonte, e *elGen* fornece uma alternativa de extensão através do acréscimo de um novo modelo. Os geradores possuem uma quantidade finita de tipos de circuitos disponíveis, e para cada tipo de circuito um tamanho máximo de *bits* em suas entradas e saídas. Qualquer novo tipo de circuito a ser disponibilizado precisa ser acrescentado através da modificação do código-fonte do gerador. Assim como os demais geradores, *elGen* também possui um limite superior de circuitos para cada tipo, mas os tipos de circuitos não são limitados, pois cada um é descrito como um modelo, e a descrição do modelo é uma das entradas da ferramenta, ou seja, para incrementar o número de tipos de circuitos disponibilizados basta adicionar um novo modelo, sem necessidade de alterar o código-fonte da ferramenta.

Em análises comparativas entre os *benchmarks*, *elGen* possui alguns diferenciais não encontrados em nenhum outro. A primeira, já citada, diz respeito à possibilidade de incremento no número de tipos de circuitos gerados sem necessidade de alteração do código-fonte. Isso é muito importante pois permite ao usuário da ferramenta adaptá-la ao seu dia-a-dia, modelando e gerando novos circuitos conforme a necessidade, sem que seja necessário alterar o programa ou solicitar a alteração aos autores. Um segundo diferencial está no fato de *elGen* realizar em uma de suas etapas a síntese lógica do circuito gerado. A síntese realizada não tem como objetivo competir com diversas ferramentas de síntese existentes, mas garantir ao usuário que o circuito gerado é sintetizável, pois existe uma transformação polinomial do circuito para aquele sintetizado. Por fim, se comparada apenas com a BenCGen, que é a ferramenta no estado da arte em geradores de circuitos para *benchmark*, *elGen* possibilita a geração de circuitos sequenciais e acrescenta um novo formado de disponibilização dos circuitos, VHDL.

Capítulo 5

Conclusões e trabalhos futuros

5.1 Conclusões

O presente trabalho apresentou três importantes contribuições para o problema de geração de circuitos para serem utilizados como *benchmarks* por ferramentas em EDA. A primeira contribuição foi a disponibilização de uma ferramenta para geração de grandes quantidades de circuitos, tanto combinatórios, quanto sequenciais. Como mencionado nos demais capítulos, a demanda crescente por novos conjuntos de *benchmarks* incentiva a produção de novos geradores de núcleos funcionais, uma vez que os circuitos gerados possui características de projetos industriais mas não são limitados como eles.

Relacionada à primeira contribuição temos um maior conjunto de circuitos disponíveis para *benchmarks*, ampliando o conjunto utilizado para testes nas ferramentas. O gerador do estado da arte, BenCGen, não permite a geração de circuitos sequenciais, e os mais conhecidos geradores, Eudoxus e Arithmetic Module Generator, fornecem poucos tipos de circuitos.

Ainda relacionada a mesma contribuição, temos uma nova forma de geração de circuitos – baseada em modelos. Essa abordagem permite a extensibilidade da ferramenta sem a necessidade de alteração do código-fonte. Um fator importante na qualidade do *software* é o grau de extensibilidade do mesmo. A extensão da ferramenta, ampliando a quantidade de tipos de circuitos gerados é facilmente obtida apenas adicionando um novo modelo de circuito aos modelos já existentes.

Outra contribuição é a pesquisa e análise dos principais *benchmarks* utilizados na área de automação de projetos eletrônicos e verificação formal, apresentado suas características, contribuições e deficiências.

A segunda contribuição diz respeito à utilização de uma metodologia para verificação dos circuitos gerados. BenCGen utilizou para verificação de seus circuitos

referências garantidamente corretas, e resolvedores de SAT para verificar a equivalência entre os circuitos gerados e as referências. Apesar do esforço, duas deficiências existem nesse tipo de verificação: não existem referências corretas para todos os circuitos gerados, logo, não há garantias de corretude de todos os circuitos gerados. Os demais geradores não apresentaram qualquer tipo de técnica de verificação dos circuitos. elGen, por sua vez, utiliza como processo de validação dos circuitos algoritmos de síntese lógica, garantindo que os circuitos gerados são sintetizáveis – premissa de circuitos industriais. BenCGen também utiliza algoritmos de síntese para verificação dos circuitos gerados, mas como parte do trabalho de verificação de equivalência, e não como função da ferramenta [Andrade, 2008]. Quando a verificação do modelo, elGen supõe que o circuito foi modelado corretamente. Qualquer erro na definição do modelo é considerado uma falha na tradução do tipo do circuito para a descrição do modelo, etapa independente da ferramenta.

A terceira contribuição, diretamente ligada à primeira, diz respeito à possibilidade de o usuário da ferramenta adaptá-la ao seu uso diário. A decisão de tornar o modelo uma das entradas da ferramenta, além de facilitar a extensão, permite que o usuário modele os tipos de circuitos desejados como *benchmarks* sem necessidade de solicitar ao autor da ferramenta qualquer tipo de modificação.

Uma outra contribuição ligada a definição do modelo, foi a criação de uma nova linguagem para descrição dos modelos, *elLen*, com principal objetivo de ser bastante simples, reduzindo assim o tempo necessário para aprendizado. Outra contribuição é permitir ao usuário descrever o circuito em uma única linguagem e gerá-la em cinco outras.

5.2 Trabalhos futuros

As sugestões de trabalhos futuros apresentados nesta seção correspondem a continuações do presente trabalho ou melhorias a serem realizadas na ferramenta, apresentadas em ordem crescente de relevância, segundo critérios do autor:

- Redução da quantidade de memória utilizada – apesar de não ter sido detectado nenhum *vazamento de memória*, a memória utilizada pela ferramenta é muito grande. Um dos principais motivos para esse gasto excessivo é a existência simultânea de duas representações do circuito, grafo do circuito e AIG, cada uma delas utilizada em etapas diferentes da ferramenta. Uma possível melhoria seria aplicar todos os procedimentos sobre uma única representação, ou que ambas

as representações não existissem simultaneamente, ou seja, que uma não fosse criada a partir da outra, como é feito atualmente.

- Aplicação de mais iterações na síntese combinatória – A síntese combinatória realizada pela ferramenta possui uma única iteração. No entanto, uma vez que as simplificações realizadas pela aplicação dos teoremas booleanos fornecem apenas melhorias locais, uma melhor síntese poderia ser realizada se o processo fosse repetido algumas vezes. É importante perceber que após algumas iterações, o ganho realizado com a simplificação não vale o esforço de todo o processo. A síntese combinatória poderia ser parametrizada com valores de números de iterações ou ganhos na redução em número de portas lógicas.

- Melhoria dos algoritmos de síntese – Apesar de a ferramenta possuir como tarefa principal a geração de circuitos, uma de suas tarefas é realizar a síntese. Todos os processos de síntese realizados são bastante simples e possuem como objetivo apenas mostrar que o circuito gerado é sintetizável, como os projetos industriais. Outros técnicas de síntese podem ser aplicadas diretamente sobre um AIG. O mapeamento tecnológico também permite que a síntese realizada seja específica para um dispositivo, permitindo assim que condições especiais disponibilizadas pelo FPGA alvo sejam utilizadas, ao invés de apenas portas lógicas básicas. Mesmo que não seja utilizado o mapeamento tecnológico, a parametrização dos tempos para as portas lógicas AND e NOT poderia contribuir com resultados mais próximos dos tempos reais se aplicados à um FPGA.

- Melhoria da linguagem – A linguagem *elLen* foi criada com objetivo de ser bastante simples. Essa simplicidade dificulta algumas construções, por exemplo, em dois níveis de *loop*. Atualmente, algumas construções tem que ser realizadas criando um módulo para representar um nível, e a maninulação da instância desse módulo representando o outro nível. Outra melhoria seria permitir inclusões de arquivos, de forma que cada módulo pudesse ser armazenado em arquivo separado. Essa melhoria traria ganhos em reusabilidade e legibilidade dos modelos.

- Melhoria do processo de análise do modelo – Atualmente, ao ser violada alguma das regras da gramática da linguagem *elLen*, a ferramenta aponta a existência de erro ao usuário, porém, sem detalhes. Seria interessante que o usuário tivesse informação do local do erro e possíveis causas do mesmo.

- Adição de novos formatos – A ferramenta atualmente disponibiliza os circuitos criados em cinco formatos: VERILOG HDL, VHDL, BENCH, EQN e BLIF.

Apesar de esse conjunto de HDLs ser aceito pela maioria das ferramentas, alguns formatos de descrição poderiam ser disponibilizados, por exemplo, OpenAccess e CNF, presentes em IWLS 2005 e Velev, respectivamente.

- Melhoria da ferramenta de visualização – O objetivo inicial da ferramenta de visualização é facilitar ao usuário a conferência do circuito gerado. Apesar de cumprir com seu objetivo, o visualizador é muito simples e fornece poucas possibilidades de interação com o usuário. Uma possível melhoria seria permitir a movimentação dos componentes desenhados na tela, possibilitnado agrupá-los conforme necessidade. Outra possível melhoria consiste na distribuição dos componentes na tela segundo outros critérios que não localidade. Como os componentes são desenhados a partir do arquivo de visualização, e esse é criado conforme o caminhar do grafo do circuito, os componentes ligados diretamente tendem a permanecer próximos no desenho. É possível uma organização por referência, sendo destacados aqueles acessados por mais componentes, por exemplo.

- Verificação dos modelos – Por construção, todo modelo que está de acordo com a sintaxe da linguagem *elLen* está correto, no entanto, ele pode não representar corretamente o tipo de circuito que está sendo modelado. Uma extensão considerável seria a criação de um processo de verificação do modelo, e não do circuito gerado. Essa abordagem faria com que todo circuito criado tivesse validação automática, assumindo que a ferramenta gera o circuito corretamente a partir do modelo.

- Criação de novos modelos – Apesar de a ferramenta em sua versão inicial possuir mais modelos de circuitos do que qualquer outro *benchmark* ou ferramenta existente, a criação de novos modelos amplia mais a utilização da ferramenta para geração de circuitos para a serem utilizados como *benchmarks* em ferramentas de Automação de Projetos Eletrônicos.

Referências Bibliográficas

(1994). Ieee standard vhdl language reference manual. *ANSI/IEEE Std 1076-1993*.

(2006). Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*.

AB, A. G. (2005). Aeroflex gaisler ab – http://www.gaisler.com – community website, not an academic paper.

Agnarsson, G. & Greenlaw, R. (2006). *Graph Theory: Modeling, Applications, and Algorithms*. Prentice Hall; 1 edition (October 2, 2006).

Akers, S. B. (1978). Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516.

Albrecht, C. (2005). International workshop on logic and synthesis 2005 benchmarks.

Andrade, F. V. (2008). Contribuições para o problema de verificação de equivalência combinacional – tese 798, departamento de ciência da computação - universidade federal de minas gerais.

Andrade, F. V.; Silva, L. M. & Fernandes, A. O. (2008). Bencgen: a digital circuit generation tool for benchmarks. In *SBCCI '08: Proceedings of the 21st annual symposium on Integrated circuits and system design*, pp. 164--169, New York, NY, USA. ACM.

Bakalis, D.; Adaos, K.; Alexiou, G.; Nikolos, D. & Lymperopoulos, D. (2001). Eudoxus: A www-based generator of reusable arithmetic cores. In *IEEE International Workshop on Rapid System Prototyping*, pp. 182–187. IEEE Computer Society.

Berkeley (2005). Berkeley logic interchange format (blif) – http://www.cs.uic.edu/ jlillis/courses/cs594/spring05/blif.pdf – resume of blif, not an academic paper.

Biere, A. (2007). The aiger and-inverter graph (aig) format.

Bjesse, P. & Borälv, A. (2004). Dag-aware circuit compression for formal verification. In *ICCAD*, pp. 42–49. IEEE Computer Society / ACM.

Boole, G. (1848). The calculus of logic. *The Cambridge and Dublin Mathematical Journal*, 3.

Brayton, R. K.; Gao, M.; Jiang, J.-H. R.; Jiang, Y.; Li, Y.; Mishchenko, A.; Sinha, S. & Villa, T. (2002). Optimization of multi-valued multi-level networks. In *ISMVL*, pp. 168–. IEEE Computer Society.

Brayton, R. K.; Rudell, R. L.; Sangiovanni-Vincentelli, A. L. & Wang, A. R. (1987). Mis: A multiple-level logic optimization system. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(6):1062–1081.

Brglez, F.; Bryan, D. & Kosminski, K. (1989). Combinational profiles of sequential benchmark circuits. In *International Symposiumon Circuits and Systems*, pp. 1929–1934.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677--691.

Corporation, F. T. (2005). Faraday technology corporation – http://www.faraday-tech.com – community website, not an academic paper.

Coudert, O. (1995). Doing two-level logic minimization 100 times faster. In *SODA*, pp. 112–121.

Darnauer, J. & Dai, W. W.-M. (1996). A method for generating random circuits and its application to routability measurement. In *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, FPGA '96, pp. 66--72, New York, NY, USA. ACM.

Davidson, S. (1999). Itc'99 benchmark circuits - preliminary results. In *ITC*, p. 1125.

F.Brglez & Fujiwara, H. (1985). A neutral net list of 10 combinational benchmark circuits and a target translator in fortran. In *International Symposiumon Circuits and Systems*, pp. 695–698.

Fleisher, H. & Maissel, L. I. (1975). An introduction to array logic. *IBM Journal of Research and Development*, 19(2):98–109.

freemodelfoundry.com (1995). Free model foundry (fmf) – http://www.freemodelfoundry.com/ – community website, not an academic paper.

Gordon, M. (1997). Synthesizable verilog - syntax and semantics / university of cambridge computer laboratory.

Group, S. (1996). Standard for synthesizing from vhdl language at the register transfer level.

Grout, S. (2001). Working group in advanced eda benchmark datasets – http://www.vhdl.org/benchmrk/ – community website, not an academic paper.

Grumberg, O.; Livne, S. & Markovitch, S. (2003). Learning to order BDD variables in verification. *Journal of Artificial Intelligence Research*, 18:83–116.

Hachtel, G. D. & Somenzi, F. (1996). *Logic Synthesis and Verification Algorithms*. Springer; 1 edition (June 30, 1996).

Homma, N.; Watanabe, Y.; Aoki, T. & Higuchi, T. (2006). Formal design of arithmetic circuits based on arithmetic description language. *IEICE Transactions*, 89-A(12):3500–3509.

Hutton, M. D.; Rose, J. & Corneil, D. G. (2002). Automatic generation of synthetic sequential benchmark circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(8):928–940.

III, J. E. H. (2000). Overview of popular benchmark sets. *IEEE Design & Test of Computers*, 17(3):15–17.

Johnson, S. C. (1979). Yacc: Yet another compiler-compiler – http://dinosaur.compilertools.net/ – community website, not an academic paper.

Kunes, M. & Danek, M. (2003). Invar-a new approach to eda benchmark generation. In *Microelectronics, 2003. ICM 2003. Proceedings of the 15th International Conference on*, pp. 50 – 53.

Lee, C. Y. (1959). Representation of switching circuits by binary-decision programs. Technical report, Bell Systems Technical.

Lesk, M. E. & Schmidt, E. E. (1978). Lex - a lexical analyzer generator – http://dinosaur.compilertools.net/ – community website, not an academic paper.

McCluskey, E. J. (1956). Minimization of Boolean functions. *The Bell System Technical Journal*, 35(5):1417--1444.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

openCores.org (2005). Opencores.org – http://opencores.org/ – community website, not an academic paper.

Pan, P. (1997). Continuous retiming: Algorithms and applications. In *ICCD*, pp. 116–121.

Pan, P. & Lin, C.-C. (1998). A new retiming-based technology mapping algorithm for lut-based fpgas. In *FPGA*, pp. 35–42.

Pistorius, J.; Legai, E. & Minoux, M. (1999). Generation of very large circuits to benchmark the partitioning of fpga. In *ISPD*, pp. 67–73.

Ronald J. Tocci, Neal S. Widmer, G. L. M. (2007). *Sistemas Digitais - princípios e aplicações*. Prentice-Hall, 10a edição.

Rudell, R. L. & Sangiovanni-Vincentelli, A. L. (1987). Multiple-valued minimization for pla optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(5):727–750.

Sentovich, E. M.; Singh, K. J.; Lavagno, L.; Moon, C.; Murgai, R.; Saldanha, A.; Savoj, H.; Stephan, P. R.; Brayton, R. K. & Sangiovanni-vincentelli, A. (1992). Sis: A system for sequential circuit synthesis. Technical report.

Smart, J. (1992). wxwidgets : Cross-plataform gui library – http://www.wxwidgets.org – community website, not an academic paper.

Synthesis, B. L. & Group, V. (2005). Abc: A system for sequential synthesis and verification.

Velev, M. N. (2004). A new generation of iscas benchmarks from formal verification of high-level microprocessors. In *ISCAS (5)*, pp. 213–216.

Verplaetse, P.; Stroobandt, D. & Campenhout, J. V. (2002). Synthetic benchmark circuits for timing-driven physical design applications. In *in Proc. International Conference on VLSI*, pp. 31--37. CSREA Press.

Vieira, N. J. (2006). *Introdução aos Fundamentos da Computação - Linguagens e Máquinas*. Editora Cengage Learning.

Zhang, L.; Lin, Z. & Lv, Z. (2002). How to group variables for reducing bdd. In *Communications, Circuits and Systems and West Sino Expositions, IEEE 2002 International Conference on*.