UNIVERSIDADE FEDERAL DO TOCANTINS
CAMPUS UNIVERSITÁRIO DE PALMAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO II

FRAMEWORK DE SÍNTESE DE SISTEMAS DIGITAIS EM ALTO NÍVEL DE ABSTRAÇÃO PARA EMPREGO EM FERRAMENTAS CAD

Pedro Henrique de Castro Lima

Orientador: Me. Tiago da Silva Almeida

Palmas
Maio de 2017

FRAMEWORK DE SÍNTESE DE SISTEMAS DIGITAIS EM ALTO NÍVEL DE ABSTRAÇÃO PARA EMPREGO EM FERRAMENTAS CAD

Pedro Henrique de Castro Lima

Trabalho de Conclusão de Curso II apresentado ao Curso de Ciência da Computação, CUP, da Universidade Federal do Tocantins, como parte dos requisitos necessários à obtenção do título de Bacharel em Ciência da Computação.

PALMAS, TO – BRASIL
MAIO DE 2017

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da Universidade Federal do Tocantins

L732f Lima, Pedro Henrique de Castro.

Framework de síntese de sistemas digitais em alto nível de abstração para emprego em ferramentas CAD. / Pedro Henrique de Castro Lima. – Palmas, TO, 2017.

79 f.

Monografia Graduação - Universidade Federal do Tocantins – Câmpus Universitário de Palmas - Curso de Ciências da Computação, 2017.

Orientador: Tiago da Silva Almeida

1. Síntese de Circuitos Digitais. 2. Circuitos Digitais. 3. Ferramentas CAD. 4. Framework de Síntese. I. Título

CDD 004

Elaborado pelo sistema de geração automatica de ficha catalográfica da UFT com os dados fornecidos pelo(a) autor(a).

Dedicatória

Dedico este trabalho ao meu pai Emezio Lopes Lima que sempre me incentivou, me ajudou e proporcionou a oportunidade de obter uma boa formação acadêmica e que durante toda minha existência se esforçou imensamente para que nunca me faltasse nada essencial.

Pedro Henrique de Castro Lima

Lista de Abreviações

AFD Autômato Finito Determinístico

AFN-ξ Autômato Finito Não Determinístico com Movimentos Vazios

ANTLR Another Tool For Language Recognition (Outra Ferramenta para Reconhecimento de Linguagens)

API Application Programming Interface (Interface de Programação de Aplicativos)

CAD Computer-Aided Design ou Desenho auxiliado por computador

CI Circuitos Integrados

FPGA Field Programmable Gate Arrays ou Vetores de Portas Programáveis

HDL Hardware Description Language ou Linguagem de Descrição de Hardware

IEEE Institute of Electrical and Electronics Engineers (Instituto de Engenheiros Elétricos e Eletrônicos)

JDK Java Development Kit ou Kit de Desenvolvimento em Java

JSON JavaScript Object Notation (Notação de Objetos JavaScript)

OSCI Open SystemC Initiative

RTL Register Transfer Level (Nível de Transferência de Registradores)

VHDL VHSIC Hardware Description Language (Linguagem de descrição de hardware VHSIC), Very High Speed Integrated Circuits

VLSI Very Large Scale Integration (Circuitos em Larga Escala de Integração)

Resumo

A modelagem de sistemas digitais exige que o *designer* de circuitos atente-se à adequação de um projeto aos demais domínios de abstração. Logo, o nível de dificuldade de construção de um circuito pode-se tornar um tanto quanto elevado, já que deve-se elaborar um modelo esquemático em um alto nível de abstração para depois então, à partir do diagrama construído, implementar o sistema em uma linguagem de baixo nível. De acordo com a metodologia *top down*, esse processo de mudança de um sistema digital de um nível de abstração para outro nível mais baixo é denominado síntese. Este projeto oferece um *framework* capaz de realizar a tradução de modelos esquemáticos de circuitos lógicos produzidos através da ferramenta CAD *Logisim* para implementações à nível de descrição de hardware. Os testes executados em 41 modelos de circuitos mostraram que a ferramenta é confiável e cumpre de forma eficaz o seu propósito.

Palavra-chave: Sistemas digitais. Síntese. CAD.

Abstract

The development of digital systems requires an extreme attention by the circuit designer due to the different abstraction domains that the same system could be; therefore, the difficulty of building a circuit might be higher primarily because a schematic in a high level of abstraction has to be modeled, and just then, from the constructed component, the system can be implemented into a low level language. According to the top down methodology, this process of translating a circuit from a level of abstraction to another is called synthesis. This project brings up a framework that is able to translate schematics of digital systems built on the CAD tool Logisim, into implementations at the hardware level. The set of tests applied on 41 different circuits models have shown that the tool ellaborated works and it is effective, ensuring the desired output.

Keywords: Digital Systems. Synthesis. CAD.

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

Microeletrônica é a tecnologia base para a construção de *hardware* e *software*. A explosão tecnológica aconteceu na década de 60 com os primeiros circuitos [1], mas foi no final da década de 80 que circuitos com mais de um milhão de dispositivos foram construídos de maneira efetiva, estes foram chamados de VLSI (*Very Large Scale Integration*), em português, Circuitos em Larga Escala de Integração ou circuitos microeletrônicos.

Tocci, R. (2008) faz uma boa referência a circuitos integrados que destaca a importância de seu uso:

> " Quase todos os circuitos digitais existentes nos sistemas digitais modernos são circuitos integrados (CIs). A grande variedade de circuitos de CIs lógicos disponível tornou possível a construção de sistemas digitais complexos menores e mais confiáveis do que aqueles construídos com circuitos lógicos discretos. ([2]p.12)"

Apesar desse avanço, havia um ponto chave que dificultava a construção de tais circuitos: projetos de circuitos com complexidade elevada, sendo esta caracterizada pela grande quantidade de entradas, portas e saídas utilizadas. Além do tempo para projetar ser longo, conseguir pessoas capazes de fazê-lo era difícil e caro, e o alto grau de complexidade exigia muitos componentes, o que ampliava ainda mais os custos. O grau de falhas deveria ser quase nulo para evitar excesso de gastos.

Com base nessas dificuldades passou-se a pensar na utilização de técnicas CAD (*Computer-Aided Design*) para a construção e otimização de circuitos. Na área de circuitos digitais muitas ferramentas foram desenvolvidas para este propósito, exemplos disso são as ferramentas *Circuit Maker*, *Logisim*,*Proteus*, *Multisim* e *Tina*.

1.1 Definição do Problema e Justificativa

Circuitos digitais são comumente representados à nível lógico e não à nível de pulsos elétricos, e isso diminui a quantidade de informações com as quais o projetista de sistemas digitais tem que lidar ao elaborar um sistema, podendo até mesmo fazer uso de ferramentas

visuais, tais como o *Logisim*, para a modelagem [3]. Essa representação é considerada um alto nível de abstração e através dela, podem-se construir os sistemas sem a necessidade em focar como será a estrutura final e outros detalhes da implementação.

Mesmo com as ferramentas CAD reduzindo muito o tempo gasto na produção de sistemas digitais, os diagramas esquemáticos elaborados não retratam detalhadamente o *hardware*. Esses detalhes estão em um nível de abstração mais baixo. Tais detalhes só podem ser incorporados utilizando outros tipos de linguagens. Atualmente, projetistas de circuitos utilizam uma Linguagem de Descrição de *Hardware* (HDL) para uma melhor modelagem de um circuito adicionado informações mais próximas às especificações reais.

A ferramenta utilizada foi o *Logisim* já que a mesma é gratuita e seu código é aberto. Sua facilidade de uso e sua difusão global são outros aspectos relevantes para o uso da mesma.

Tendo em vista que a ferramenta *LogiSim* não realiza a tradução de seus diagramas para linguagens HDL, há a necessidade da criação de um *framework* capaz de efetuar tal conversão que é o propósito deste trabalho: a síntese de circuitos combinacionais diagramados em um alto nível de abstração para um nível mais baixo de abstração de modo a facilitar a construção e elaboração de sistemas e circuitos digitais, e posterior gravação dos HDLs gerados em placas FPGAs [1] .

A ferramenta de síntese proposta traz em sua implementação um padrão de arquivos para a exportação da expressão booleana que rege um circuito combinacional gerado no *Logisim* de maneira que outros softwares possam utilizar o arquivo para futuras otimizações.

Ao fim, houve o acoplamento do *framework* ao *Logisim*, permitindo que esta nova versão possa ser utilizada por acadêmicos e usuários de todo o mundo.

[1]Field Programmable Gate Arrays são circuitos digitais integrados compostos de portas lógicas e a conexão entre esses blocos lógicos [4].

2 Fundamentação Teórica

Neste capítulo serão abordadas as técnicas a serem utilizadas para a realização da síntese de circuitos digitais. Uma referência a álgebra booleana e circuitos lógicos será explanada a fim de esclarecer sobre a maneira como ocorrerá a relação da tradução e refinamento de um circuito dentro do *framework*. Além disso, é preciso entender os níveis de abstração existentes ao se modelar um circuito digital e como se dará o processo de síntese que será feito através da elaboração de um compilador que partirá do projeto esquemático elaborado no *Logisim* e será traduzido para a linguagem de descrição de *hardware SystemC*.

2.1 Representação de Sistemas Digitais

As fases de concepção, análise, implementação e testes de sistemas eletrônicos são importantes para o desenvolvimento de tecnologias de qualidade. Um projeto de *hardware* ou *software* elaborado de forma coerente permite a redução significativa de tempo, custo monetário e intelectual.

Neste campo há alguns trabalhos que revolucionaram a sistematização de projetos de sistemas eletrônicos tais como o trabalho de [5], [6],[7],[8], [9], [10] e [11]. Dadas as diversas representações de um determinado sistema eletrônico, o projeto pode ser dividido em três domínios: comportamental, estrutural e físico.

O domínio comportamental diz respeito às funcionalidades de um sistema. Neste nível de abstração o sistema é tratado como uma caixa preta, ignorando as implementações internas. O foco é como o sistema irá responder dado uma série de entradas, e se as saídas representam os valores esperados.

O domínio estrutural descreve a implementação interna de um sistema. Essa descrição tem como ponto central especificar os componentes utilizados e as ligações entre eles. Um bom exemplo é a montagem de uma estrutura na ferramenta para produção de circuitos *Logisim*, onde o enfoque está no design da estrutura de um circuito digital.

Por último, o domínio físico detalha as características físicas do sistema e mais informações do ponto de vista estrutural. Nesse domínio são especificados tamanho dos

componentes, a localização física dos componentes em uma placa, e o caminho físico de cada ligação [12].

2.1.1 Níveis de Abstração

Durante a fase de elaboração, modelagem e análise de um sistema microeletrônico complexo, é essencial conhecer o conceito de nível de abstração. Um nível de abstração é um conjunto de descrições de projeto com o grau de detalhamento necessário àquela implementação [13].

Os domínios de um sistema podem ser melhores expressos em eixos e cada eixo é dividido em níveis de abstração. O eixo comportamental pode ser dividido do nível mais alto para o mais baixo em: Algoritmos, Linguagem RTL (*Register Transfer Level*) , Equações Booleanas e Equações Diferenciais. O eixo estrutural, seguindo o mesmo princípio, pode ser dividido em: Processador, Memória, Multiplexadores; Transferência de Registradores: Portas Lógicas, *Flip-Flops* e Transistores. O eixo físico pode ser dividido em: *Floorplan*, *Layout* de Células, *Netlist* e Polígonos. O *framework* produzido situa-se entre o eixo estrutural e o comportamental, pois o *Logisim* realiza descrições sobre o comportamento (Álgebra Booleana) e à forma (Portas Lógicas) de circuitos digitais.

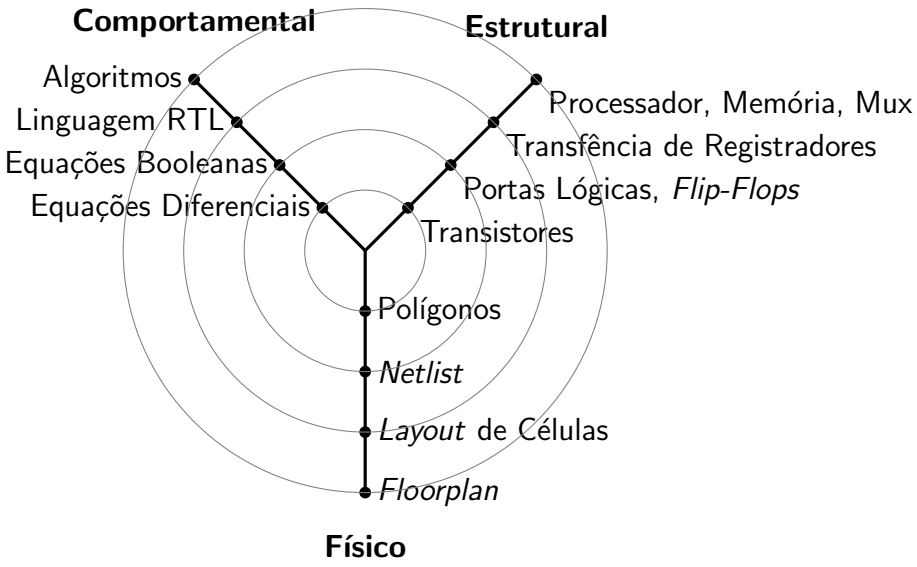
Gajski e Kuhn (1983) [5] sistematizaram os projetos de sistemas eletrônicos de acordo com o contexto da época. Assim, Riesgo, Torroja e Torre (1999) [3] atualizaram as etapas de projeto para as novas representações que surgiram ao longo dos anos, conhecido como "Diagrama Y" representado pela Figura 2.1.

O "Diagrama Y" representa os níveis de abstração do mais alto ao mais baixo, sendo denominado o processo de passar de um nível mais alto de um eixo para um nível mais baixo do mesmo eixo, Síntese. Esse processo faz parte da metodologia *top down* [5], [3].

Durante o desenvolvimento de um sistema digital, quatro níveis de abstração devem ser considerados [12]:

- Nível de Transistores
- Nível de Portas
- Nível RTL ou Transferência de Registradores
- Nível de Processador

Figura 2.1: *Diagrama Y Digital.*



Nível de Transistores

O nível mais baixo de abstração do eixo estrutural é o nível de transistores. As bases desse nível são os transistores, capacitores entre outros.

Nesse nível são destacados os pulsos elétricos como fatores de entradas dos componentes. O circuito digital é tratado como um sistema analógico onde os sinais recebidos variam no decorrer do tempo e assumem valores contínuos [12].

Nível de Portas Lógicas

O nível de portas lógicas retrata o uso de circuitos lógicos como portas *AND* e *XOR*. Nesse nível o sinal é um valor discreto variando de zero a um, avaliados como baixa e alta voltagem respectivamente.

O comportamento nesse nível é definido através da Álgebra de Boole e todos os circuitos estão no modelo de portas lógicas.

Nível RTL ou Transferência de Registradores

No nível RTL, as estruturas são definidas à partir de portas mais simples. São exemplos, somadores, comparadores e registradores. O comportamento de um RTL deve destacar o caminho da informação, por isso há o frequente uso de uma máquina de estados para descrever tal comportamento.

Nível de Processador

O nível de Processador é o nível mais alto de abstração. Destacam-se nessa camada os processadores e memórias. Associa-se o comportamento nesse nível a um código desenvolvido em uma linguagem de programação comum, ou interconexões de processadores, cache, memória e barramentos.

Conhecidos os níveis de abstração é perceptível que os domínios estão intimamente conectados entre si, sendo que cada camada de abstração tem uma representação nos três eixos. Esse conceito é um ponto importante para um melhor entendimento do processo de síntese de sistemas digitais.

2.1.2 Síntese

A síntese nada mais é que a saída de um nível de abstração para outro mais baixo. Essa mudança pode acontecer nos domínios comportamental, estrutural, ou físico, e o resultado é a descrição da estrutura em um nível mais baixo [12].

Ao analisar o Diagrama Y (Figura 2.1), o processo de síntese pode ser descrito como um movimento de saída do eixo comportamental para o estrutural ou, pode ser a mudança de um nível mais alto para um mais baixo (*top down*). Além disso, há também a extração que é uma mudança em direção às camadas superiores (*bottom up*) justificada pela utilização de componentes mais leves.

À medida que se move em direção aos níveis mais baixos, o *hardware* passa a ser mais detalhado e o resultado são estruturas a nível de portas-lógicas.

2.2 Álgebra Booleana e Circuitos Lógicos

Expressões baseadas em Álgebra Booleana são passíveis de redução através de métodos heurísticos [14]. A Figura 2.2 representa bem a relação da álgebra booleana com o projeto de circuitos, onde a saída do circuito é resultado de uma série de operações e a expressão booleana que rege a saída desse circuito (que nesse caso é somente uma) é definida por $\overline{A}C + \overline{B}D$.

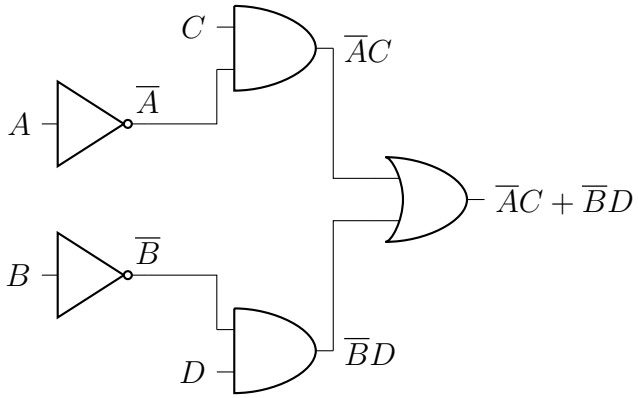


Figura 2.2: *Exemplo de circuito simples à nível de portas lógicas*

Existem algumas operações básicas na Álgebra Booleana: **E** (*AND*), **OU** (*OR*) e complemento ou inverso (*NOT*), que é a negação do símbolo ou sinal referenciado.

2.2.1 Operação OU (Adição Lógica)

A operação OU consiste em uma condição lógica "opcional", quando pelo menos uma entrada binária vale **1** a saída será **1**, caso todas as entradas sejam **0** o resultado será **0** como visto na Tabela 2.1. A Figura 2.3 é um exemplo de uma porta lógica *OR*.

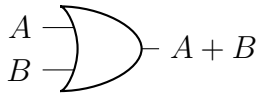


Figura 2.3: *Porta lógica OR*

Tabela 2.1: *Tabela verdade referente à operação OU*

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

2.2.2 Operação E (Multiplicação Lógica)

A operação E pode ser comparada à uma multiplicação aritmética normal contudo as únicas possibilidades são os números **0** e **1**. Considerando o número **1** como verdadeiro ou nível alto de sinal em circuitos, a saída só será alta ou verdadeira caso todas as entradas sejam **1**, em qualquer outro caso o resultado é **0**. Esse comportamento é ilustrado na Tabela 2.2.

Tabela 2.2: *Tabela verdade referente à operação E*

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

A operação **E** É uma operação que gera uma condição extremamente dependente das demais entradas.A porta lógica que a representa é a porta *AND* (em português E) vista na Figura 2.4.

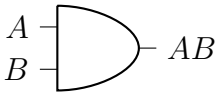


Figura 2.4: *Porta lógica AND*

2.2.3 Operação de Complemento (Negação ou Inversão)

A operação de complemento é a inversão de um sinal ou resultado. O inverso de verdadeiro é falso e o inverso de falso é verdadeiro. Da mesma maneira, pode-se afirmar que o complemento de um sinal alto ou **1** equivale à um baixo nível de sinal ou **0** e vice-versa (Tabela 2.3).

Tabela 2.3: *Tabela verdade referente à operação de complemento*

A	$\overline{A}$
0	1
1	0

O circuito associado a essa operação é denominado inversor, ou porta *NOT*. Sua ação é transformar o sinal de entrada em seu inverso, realizando a mesma função da operação de complemento. O circuito inversor é ilustrado na Figura 2.5.

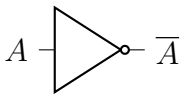


Figura 2.5: *Porta lógica NOT*

2.3 Circuitos Digitais

Tocci, R. (2008) [2] enfatiza que

> "Circuitos Digitais são projetados para produzir tensões de saída que estejam dentro dos intervalos determinados para os binários 0 e 1 [...]. Da mesma maneira, circuitos digitais são projetados para responder, de modo previsível, à tensões de entrada que estejam dentro dos intervalos definidos para 0 e 1".

A área de circuitos digitais é bastante diversificada e a construção de dispositivos exige um alto conhecimento nos tipos de circuitos existentes. Assim, os circuitos podem ser classificados em dois tipos: Circuitos Combinacionais e Sequenciais. Ambos são abordados neste trabalho contudo o foco deste projeto são os circuitos combinacionais.

2.3.1 Circuitos Combinacionais

Circuitos Combinacionais são circuitos que dependem unicamente dos valores de entrada (Figura 2.6). Eles são constituídos de portas lógicas e essas portas determinam os valores de saída que é representada por uma expressão booleana. Exemplos de circuitos combinacionais são somadores e multiplexadores.

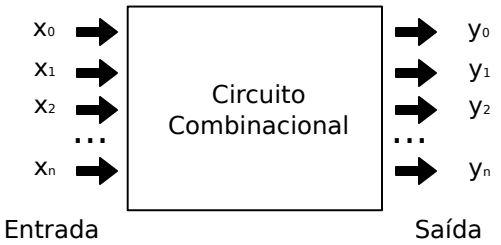


Figura 2.6: *Representação genérica do funcionamento de circuitos combinacionais*

2.3.2 Circuitos Sequenciais

Circuitos sequenciais são caracterizados por seu comportamento peculiar à respeito da saída que é influenciada por estados anteriores, ou valores armazenados (Figura 2.7). Por essa razão, a construção de expressões booleanas para esta categoria de circuito torna-se complicada já que não é possível prever o resultado de maneira genérica. Exemplos desse tipo de circuito são *latches* e *flip-flops*.

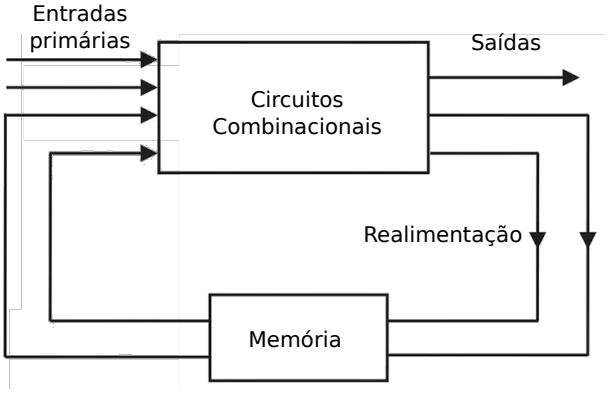


Figura 2.7: *Representação genérica do funcionamento de circuitos sequenciais*

2.4 A Ferramenta *Logisim*

A ferramenta *Logisim* foi desenvolvida pelo Dr. Carl Burch, Engenheiro de *Software* pela empresa *Google* em *Kirkland, Washington*, Estados Unidos. Seu intuito era de desenvolver uma ferramenta educacional para concepção e a simulação digital de circuitos lógicos. O último *release* do *software* foi a versão 2.7 disponibilizada em 21 de Março de 2011[15], após isso o criador deu livre acesso ao código fonte para futuras melhorias. Logo surgiram variações da ferramenta, dentre elas o *Logisim Evolution*[16] desenvolvido pelo *REDS Institute*, que faz parte do *Haute École d'Ingénierie et de Gestion du Canton de Vaud* (Escola de Engenharia em Gestão de Cantão de Vaud) em Yverdon-les-Bains, Suíça. Outro projeto derivado é a ferramenta denominada *Digital*[17], esta é uma total reformulação do *Logisim* desenvolvida pelo Professor Helmut Neemann da *Baden-Württemberg Cooperative State University Mosbach.*

O *Logisim* tornou-se popular por diversas razões sendo as principais [15]:

- criação de um módulo para o ensino de ciência da computação em geral;
- uma unidade para níveis intermediários em cursos de organização de computadores;
- a existência de mais do que um semestre inteiro em cursos mais avançados de arquiteturas de computadores.

A estruturação do *Logisim* é dedutiva, utilizando um sistema de "puxa e arrasta" no qual o usuário seleciona os componentes desejados do menu lateral, que contém diversas bibliotecas de circuitos prontos, para a área de desenho como visto na Figura 2.8 aonde uma porta lógica *AND* é inserida.

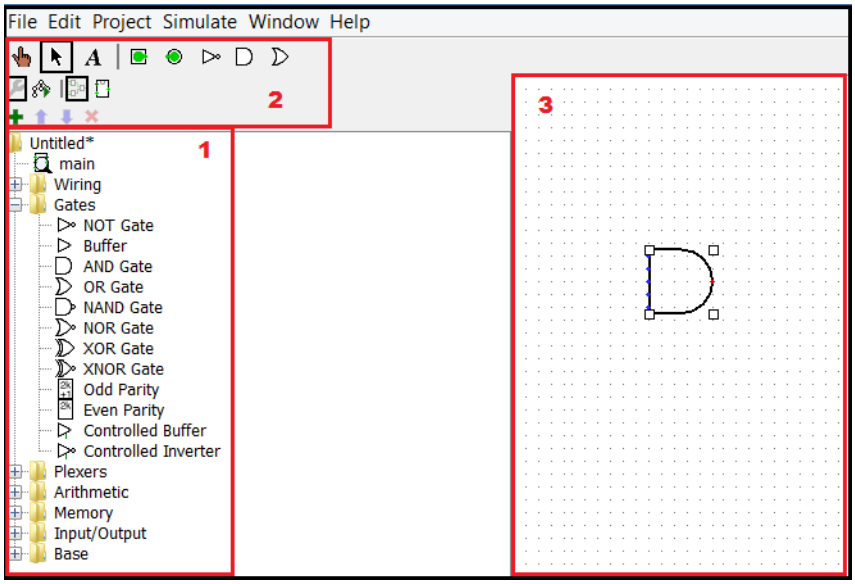


Figura 2.8: *Demonstração de adição de componentes a um projeto no Logisim*

O *Logisim* dispõe de um menu superior que dá acesso a áreas de configuração do aplicativo, simulação, linguagem, menu para seleção e armazenagem de arquivos e manual de ajuda ao usuário, e logo abaixo há o guia de ferramentas rápidas (Item 2 da Figura 2.8)

No Item 1 da Figura 2.8 é onde está localizado o menu de componentes, no qual todas as estruturas aceitas pelo *software* estão disponíveis (portas lógicas, somadores e ULAS).

Uma vez selecionado o componente, o usuário arrasta-o para a área de criação do projeto (Item 3 da Figura 2.8).

Os circuitos modelados pelo *Logisim* podem ser simulados dentro do próprio software, verificando se o comportamento do componente atende às expectativas esperadas.

Este trabalho utiliza o *Logisim Evolution*. O intuito é mesclar o *framework* ao código do *Logisim Evolution* dando uma funcionalidade extra a esse poderoso aplicativo. Um dos responsáveis pela criação dessa nova versão do *Logisim* é o Professor Yann Thoma [18]. Yann Thoma também foi um dos autores do projeto *Math2Math*[19].

Atualmente o *Logisim Evolution* está armazenado em um repositório no *GitHub* [16]. O mesmo é atualizado com frequência pelo *REDS Institute* o que motiva uma futura integração com o*framework* produzido.

Apesar de ser uma derivação do *Logisim*, tanto a estrutura do código-fonte como a interface gráfica original foram mantidas, e novas funcionalidades foram adicionadas sendo algumas dessas:

- CRONOGRAMA - permite visualizar as mudanças de sinal de um circuito através de uma visualização em um diagrama de tempo;
- INTEGRAÇÃO DE PLACAS ELETRÔNICAS - os diagramas esquemáticos podem agora ser testados em uma placa FPGA real (Ver Figura 2.9);
- EDIÇÃO DE PLACAS FPGA - novas placas FPGAs podem ser adicionadas para futuros testes;
- COMPONENTES VHDL - um novo tipo de componente no qual seu comportamento é especificado em VHDL;
- CONSOLE TCL/TK - interface entre o circuito e o usuário;
- LEDs RGB.

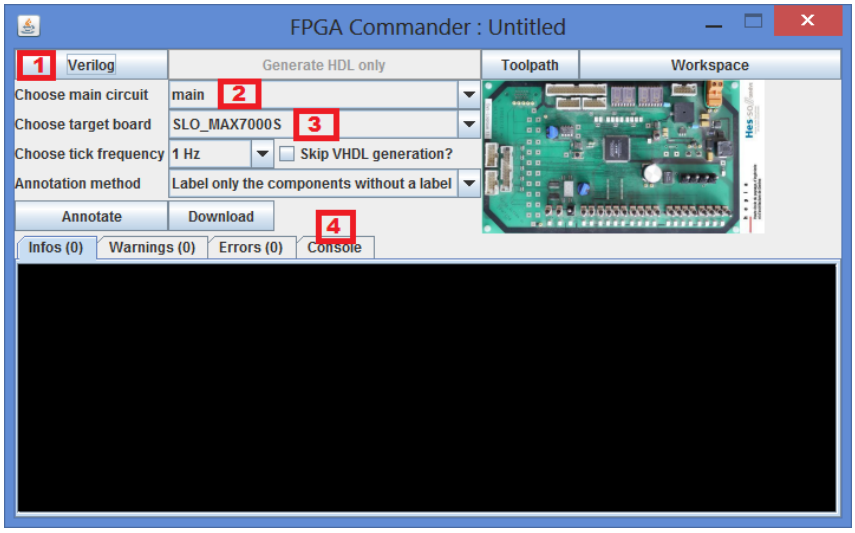


Figura 2.9: *Ferramenta para teste em uma FPGA do Logisim Evolution*

A Figura 2.9 é a janela de comandos FPGA, um recurso exclusivo do *Logisim Evolution*. Através dessa ferramenta o usuário pode obter um código em *Verilog ou VHDL* de um circuito modelado na área de desenho apenas selecionando a linguagem desejada como retratado no item 1 da Figura 2.9.

Para que a descrição aconteça da maneira esperada, o projetista deve selecionar o circuito que deseja mapear (Item 2 da Figura 2.9), que se encontra na área de projetos do *Logisim*. Feito isso, é preciso selecionar uma FPGA que esteja de acordo com as características do circuito, cabe ao projetista de sistemas digitais avaliar esses aspectos. As associações de entradas e saídas são feitas clicando no botão *Download*.

Qualquer falha ou informações extras a respeito desse processo são exibidas no console localizado no item 4 da Figura 2.9.

2.5 Conceitos de Compilação

Sistemas são dispostos em diferentes níveis de abstração como visto anteriormente. Contudo, para permitir que um *software* em um alto nível se comunique com um *hardware*, que é considerado a um baixo nível de abstração, é preciso traduzir o código original em uma linguagem que este *hardware* "compreenda". O ferramental capaz de realizar tal transformação de códigos é denominado Compilador.

O compilador possui duas etapas de mapeamento durante o processo de compilação do código original ou código-fonte: análise e síntese.

A análise é descrita como a fase em que há a quebra do código-fonte em partes constituintes sendo que uma dessas peças é regida por uma estrutura gramatical. Neste momento ocorre a montagem de uma representação intermediária do código-fonte baseada nessas estruturas gramaticais. É na análise que se define a tabela sintática do compilador usado posteriormente [20].

A fase de síntese na compilação é responsável pela construção do código final em uma plataforma alvo a partir do código intermediário proveniente da análise. Essas duas fases podem ser associadas como a relação *front end* e *back end* da construção de um *software* sendo a análise caracterizada pelo *front end* e a síntese pelo *back end* [20].

Detalhadamente um compilador é mais complexo e pode ser descrito em etapas específicas. Essas etapas são realizadas em sequência até se chegar à última etapa que realiza a criação do código final. A Figura 2.10 ilustra as etapas de compilação dentro das suas respectivas fases: análise e síntese.

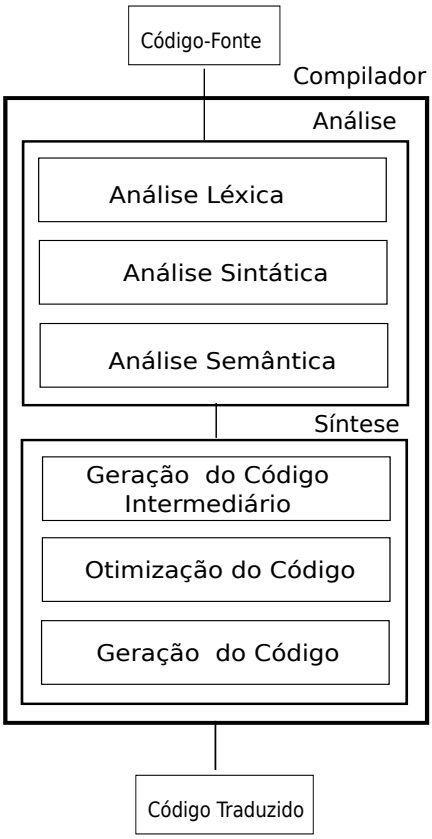


Figura 2.10: *Fases da compilação*

2.5.1 Análise Léxica

A primeira fase de um compilador é denominada Análise Léxica. Essa fase realiza a tokenização [2] do programa.

Um lexema é representado por um conjunto de símbolos com um significado ou expressões regulares. Os lexemas são informados pelo programador e associados a *tokens* na linguagem resultante [20]. Por exemplo, um *token 'INT'* é representado por um conjunto de símbolos que faz sentido para a linguagem nesse o caso o lexema regido pela expressão regular **(1+...+9)(0+...+9)*** simbolizando um conjunto de números inteiros que não inicie por zero.

O resultado é que sempre que houver ocorrência de certas cadeias de caracteres ou símbolos individuais, o analisador léxico buscará na lista de lexemas definidas pelo programador e associará *tokens* aos valores informados.

[2]Nomenclatura para o método de tradução de uma linguagem de entrada em códigos de caracteres chamados *tokens*.

Resumidamente, a análise léxica é a fase que dá significado às entradas para que estas possam ser utilizadas nas etapas posteriores.

2.5.2 Análise Sintática

Terminada a primeira fase que tratou do significado das palavras, a segunda fase fará uso dos *tokens* obtidos para uma verificação gramatical.

A estrutura típica utilizada na análise sintática é associada a uma árvore conhecida por *Árvore Sintática*, na qual um nó descreve uma operação e os filhos desse nó representam os argumentos da operação [20].

Tomando por exemplo a seguinte expressão obtida pela análise léxica:

$$< INT >< + >< INT > \tag{2.1}$$

A árvore sintática (Figura 2.11) para a Expressão 2.1 mostra os caminhos que podem ser seguidos. Caso o código siga a estrutura da árvore (primeiro operando, um operador e outro operando), a expressão informada está gramaticalmente correta.

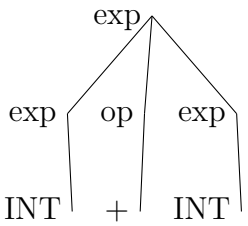


Figura 2.11: *Árvore Sintática exemplo para operação de soma*

Por exemplo, dada uma equação $2 + 2$, o primeiro número 2 é uma expressão que gera um *token INT*, obrigatoriamente após um *INT* espera-se um operador que no caso dessa árvore da Figura 2.11, deve ser um operador de soma, caso um outro valor de operador seja informado, a análise sintática informará que houve um erro gramatical.

2.5.3 Análise Semântica

A terceira fase conhecida como Análise Semântica utiliza dados provenientes da tabela sintática para a verificação do sentido do código. As etapas anteriores analisavam a

existência dos símbolos e a gramática do código-fonte. Na etapa seguinte é preciso verificar a coerência do código gerado [20].

Muitos problemas são verificados apenas através de semântica como erros de tipos de variáveis, utilização de índices fora do limite permitido de um vetor, divisão por zero, erros de alocação de memória, dentre outros.

2.5.4 Geração de Código Intermediário

No processo de tradução de um programa fonte em um código específico, um compilador pode construir uma ou mais representações intermediárias [20]. No caso deste projeto, tem-se conhecimento da produção de 2 (dois) modelos de código intermediários: um destinado à minimização e outro destinado à síntese do código final.

Duas propriedades bastante importantes que devem ser levadas em conta ao se produzir um código intermediário [20]:

- deve ser fácil de se produzir;
- deve ser fácil de se traduzir para a arquitetura alvo.

No caso deste trabalho, há um código que faz uso da notação JSON e outro utilizando um padrão de *tokens* associado às operações booleanas efetuadas por circuitos lógicos. Isso é melhor abordado na Seção 3.1.3 que trata da metodologia utilizada.

2.5.5 Otimização de Código

Antes da geração de um código final, muitas vezes opta-se pela otimização do código a fim de reduzir os custos computacionais, como quantidade de elementos utilizados, tempo de processamento e redundância de código. Com isso, o *hardware* terá menos trabalho para processar as informações e emitirá uma resposta em um espaço de tempo menor.

Geralmente, o passo de otimização faz uso de heurísticas para atingir uma solução ótima. Contudo, essa fase da compilação pode acabar se tornando a mais demorada, o que se torna justificável pelo resultado obtido: um código mais limpo e eficiente.

2.5.6 Geração de Código

A última etapa da compilação é a geração do código traduzido na linguagem desejada. Basicamente, utiliza-se o código intermediário para um mapeamento das entradas originadas do programa fonte com elementos equivalentes na linguagem alvo. Terminada essa fase, o processo de compilação tem fim e o código final pode ser testado para verificar se o mesmo comporta-se como desejado.

2.6 Linguagem de Descrição de *Hardware*

Um sistema digital pode ser descrito em diferentes níveis de abstração e de diferentes pontos de vista. Uma linguagem de Descrição de *Hardware* (HDL) deve ser capaz de modelar e descrever fielmente e precisamente um circuito em um nível de abstração desejado, independentemente se tal circuito já foi construído ou está em desenvolvimento, ou esteja no ponto de vista estrutural ou físico[12].

As características fundamentais de um circuito são definidas pelos conceitos de entidade, conectividade, concorrência e tempo. Entidade é o bloco de construção de um circuito. Descreve características a respeito de si mesma e não em relação com outras entidades. Conectividade é a associação de uma entidade com elementos externos a ela tais como entradas de dados ou outras entidades. Concorrência descreve o tipo de comportamento. Tempo define o período para a conclusão de uma operação.

Linguagens HDL cobrem as seguintes descrições:

- a níveis de Portas e RTL (*Register-transfer level* ou Nível de Transferência de Registradores);
- sistemas que estejam no eixo estrutural, comportamental e físico [12].

Muitas HDLs foram desenvolvidas no decorrer do tempo sendo as mais utilizadas globalmente: VHDL , *Verilog* e *SystemC*.

2.6.1 SystemC

O *SystemC* é um conjunto de bibliotecas em C++ construído especificamente para a produção de sistemas digitais. O *SystemC* foi definido pela OSCI (*Open SystemC Initiative*) e ratificado como um padrão IEEE $1666^{TM} - 2011$ [21].

A linguagem é feita em particular para modelagem de sistemas, avaliação e verificação das operações de blocos de implementações de *hardware* ou *software*, e para a elaboração e métrica das interações entre blocos funcionais [21].

Projetistas de sistemas utilizam uma separação de função, arquitetura, e implementação para gerenciar os problemas de produção de um circuito. O *SystemC* descreve um sistema a nível de implementação[22].

O *SystemC* realiza descrições eficientes, e sua sintaxe remete às bases da linguagem C, bastante conhecida. Como o foco da síntese de circuitos é ajudar o projetista, a utilização de uma estrutura de desenvolvimento conhecida é uma peça chave para uma maior possibilidade de êxito. Esse foi um dos motivos em se utilizar o *SystemC* neste projeto, além do fato de possuir uma biblioteca adicional para verificação do dispositivo modelado, garantido a consistência de seu uso.

Metodologia de Projeto de Sistemas Digitais Atual

A metodologia de projeto de sistemas digitais atual começa com um engenheiro de sistemas escrevendo um modelo de sistema em C ou C++ para verificar os conceitos e algoritmos a nível de sistema. Após a validação do algoritmo, as partes do modelo C/C++ a serem implementadas no *hardware* são manualmente convertidas para uma descrição em VHDL ou *Verilog* para execução a nível de *hardware* [23]. Essas etapas podem ser melhor descritas na Figura 2.12 onde o "Resto do Processo" é o mapeamento e a vinculação física em um dispositivo.

Metodologia de Projeto do SystemC

A metodologia de projeto seguida pelo *SystemC* oferece muitas vantagens em relação à metodologia tradicional para projetos a nível de sistema. A metodologia do *SystemC* é ilustrada na Figura 2.13.

Através do *SystemC* há constantes refinamentos e as melhorias são feitas por etapas. Durante o refinamento muitos *bugs*[3] podem ser detectados e corrigidos para que ocorra a síntese e o resto do processo que é o processamento para a simulação física em uma FPGA.

[3]Bugs são falhas que em compilação podem estar associados a erros léxicos, sintáticos ou semânticos

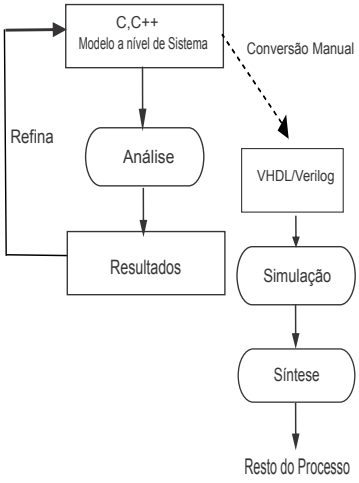


Figura 2.12: *Metodologia de projetos de sistemas utilizando tecnologias convencionais*

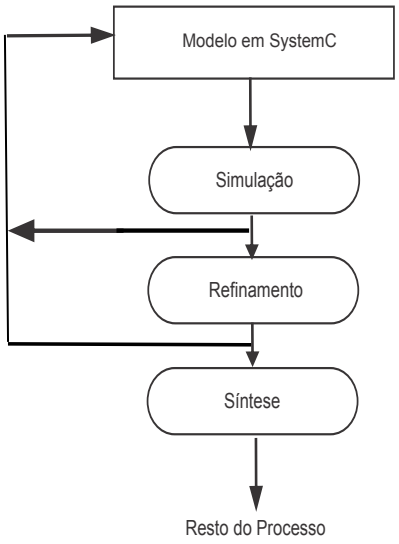


Figura 2.13: *Metodologia de projetos do SystemC*

Outra vantagem, é a utilização de uma linguagem única o que facilita para o projetista que não precisa ser um especialista em várias linguagens.

2.7 Tecnologias Utilizadas

2.7.1 JSON

JSON é um formato de texto que facilita a troca de dados entre todas as linguagens de programação. Sua sintaxe utiliza poucos símbolos que são: chaves, colchetes, dois pontos e vírgula. Essa estrutura simples é um dos aspectos que torna o JSON uma tecnologia atraente.

Muitas linguagens utilizam a notação de objetos, contudo isso não é um padrão absoluto. Logo, alguns modelos podem divergir drasticamente, dependendo da linguagem. O JSON não trabalha com objetos de fato, ao invés disso pares de informações, rótulo do objeto e valor, são armazenados. Ambos são cadeias de caracteres [24].

O JSON também dá suporte a listas de valores, que são criadas através do uso de colchetes. Todo elemento incluso no intervalo dos colchetes é um elemento da lista. O aninhamento de vetores e objetos permite a representação de estruturas de dados mais complexas como árvores [24].

Neste trabalho a notação JSON é utilizada para a modelagem de um código intermediário para uma etapa de otimização de circuitos combinacionais abordada na Seção 3.1.3.

2.7.2 ANTLR

Programas que reconhecem linguagens são chamados analisadores sintáticos. Sintaxe é um conjunto de definições que governa a gramática de uma linguagem. Dá se o nome de gramática ao conjunto de regras que estrutura uma frase. A ferramenta ANTLR transforma gramáticas em analisadores que são extremamente similares a um analisador construído por um programador experiente [25].

O ANTLR gera analisadores recursivos descendentes à partir de regras gramaticais. Contudo, existem os problemas de ambiguidade que são condicionados pela gramática de entrada. Caso haja uma recursão à esquerda, ela pode ou não ser interpretada pelo analisador. O ANTLR v4 consegue lidar com recursão à esquerda direta, porém o mesmo não pode ser afirmado quando a recursão é mais a esquerda e indireta.

O problema da ambiguidade pode ser solucionado pela adequação da gramática ofere-

cendo um único caminho ao derivar-se, contudo o ANTLR trata as questões de ambiguidade escolhendo a primeira alternativa disponível na árvore de decisão.

O ferramental ANTLR pode ser utilizado para obtenção dos *tokens* basta que o programador informe em qual dos dois estágios está sendo trabalhando.

Neste trabalho o interesse em utilizar-se o ANTLR é justificado pela diminuição de complexidade da análise sintática do código intermediário tokenizado, além da alta capacidade de adaptação da ferramenta.

3 Metodologia

O *Framework*, produto deste projeto, visa uma tradução do código-fonte proveniente de uma linguagem base. Neste caso são os esquemas de circuitos desenhados a partir de uma ferramenta CAD para uma linguagem de descrição de *hardware*. Essa tradução do programa fonte, conhecida como compilação, pode ser associada a uma mudança de nível de abstração, denominada síntese, como visto na Seção 2.1. Assim, a elaboração do projeto é descrita em fases de compilação e a etapa de obtenção da entrada utiliza o programa fonte da ferramenta *Logisim*. Na Figura 3.1, todo o processo do *framework* pode ser visualizado passo a passo.

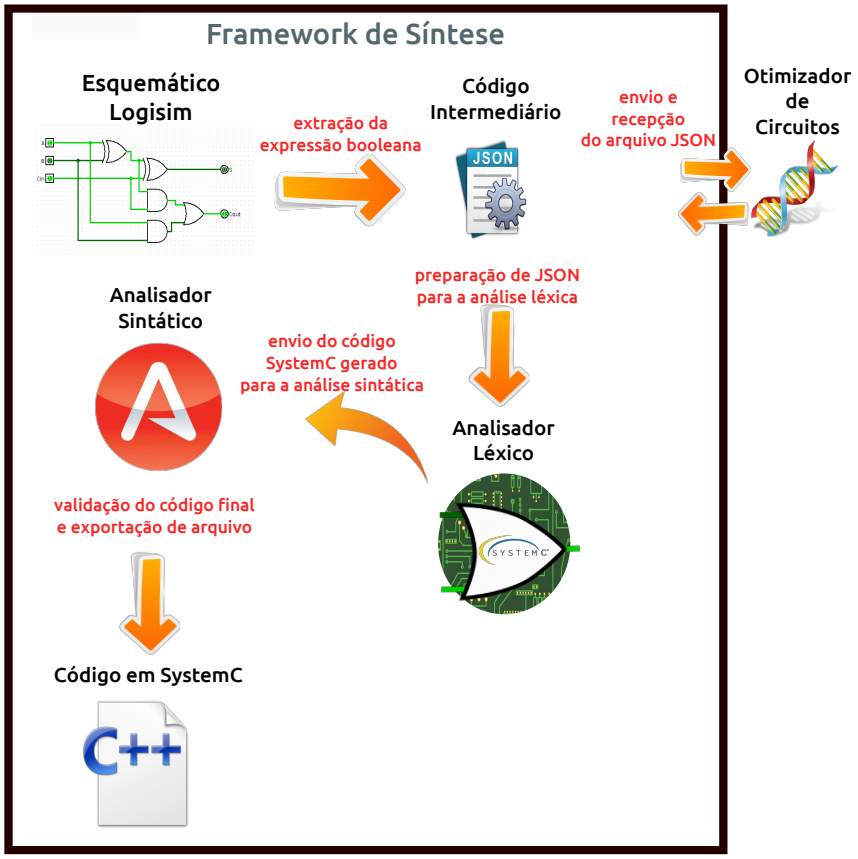


Figura 3.1: *Diagrama descritivo das etapas do framework de síntese*

A linguagem utilizada foi Java pela compatibilidade com a ferramenta *Logisim Evolution* que também foi construída em Java. O ambiente de programação foi o Eclipse 3 Neon sob a JDK 1.8.

3.1 Obtenção do código fonte

Ao modelar um circuito digital é preferível e mais cabível sair de um nível mais alto de abstração, o qual o foco é, geralmente, criar uma estrutura esquemática como as projetadas pelo *Logisim*. Não há a preocupação em ater-se no comportamento detalhado do circuito projetado, essa facilidade justifica o uso de ferramentas CAD para as etapas de elaboração e modelagem. Fica a cargo de outros *softwares*, ou do projetista de sistemas microeletrônicos detalhar as informações a respeito do componente projetado de modo que esteja próximo às especificações do mundo real, ou nível de pulsos elétricos.

Existem ferramentas já desenvolvidas e outras em andamento que se direcionam ao mesmo objetivo deste trabalho, tal como a ferramenta *Math2Math*[19]. Contudo, parte-se de um outro eixo do "Diagrama Y" (Figura 2.1), o eixo comportamental. O foco do *Math2Math* é descrever operações matemáticas básicas em VHDL utilizando a notação de ponto-flutuante tanto para arquiteturas de 32 bits como para arquiteturas de 64 bits. O código fonte foi construído em *Octave* e suporta as seguintes operações: Adição, Subtração, Multiplicação, Divisão e Raiz Quadrada, além de suportar algumas estruturas de controle: *if/then/else* e *for loop*.

A obtenção do código de origem é a primeira etapa de um compilador. O foco nesse momento é capturar um código que seja capaz de indicar a estrutura dos esquemáticos em questão de funcionamento e elementos estruturantes (portas, pinos e ligações). Foi visto na Seção 2.2 que a Álgebra Booleana descreve o funcionamento de circuitos eletrônicos, logo o objetivo da primeira fase pode ser resumido em obter a expressão booleana que representa o diagrama elaborado através do *Logisim*.

Definido o objetivo de encontrar a expressão booleana, é preciso aplicar os meios para essa extração. O *software Logisim* dispõe de códigos prontos que são capazes de realizar tal tarefa que tornam-se mais claros na seção 3.1.1.

3.1.1 Análise Combinacional

Os modelos gerados pelo *Logisim* são diagramas desenhados em uma área de construção ou área de desenho. Os componentes são inseridos nessa área e consequentemente mapeados em uma escala de coordenadas 2D (X,Y). Apesar de a modelagem de um circuito ser elaborada em formas de esquemas gráficos, há uma maneira de obter as informações

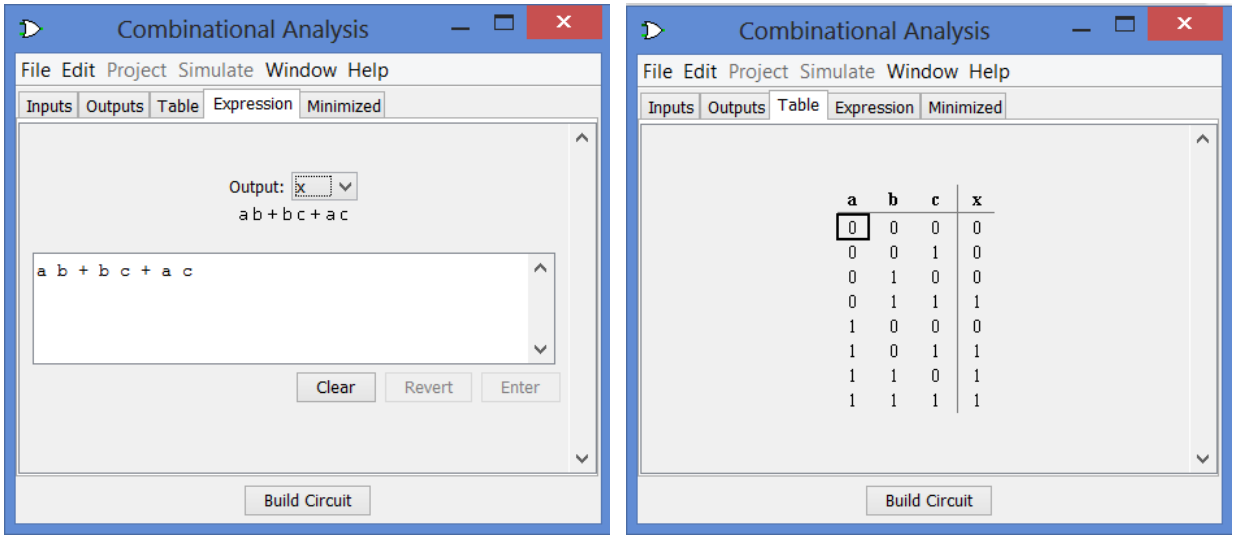
referentes a cada componente como número de entradas e saídas, faixa de *bits* e o tipo de componente (multiplexador, demultiplexador ou somador), sem a necessidade de processar os mapas gráficos da área de desenho.

O código fonte do *Logisim* possui uma classe chamada "*Analyze*". Essa classe foi feita especificamente para a etapa de análise combinacional dos circuitos construídos. Ao se invocar a função de análise combinacional de um projeto, dependendo dos tipos de circuitos utilizados, uma expressão booleana é gerada ou, caso o circuito não utilize portas lógicas diretamente, uma tabela verdade é gerada e com base nas informações dessa tabela, uma expressão booleana é montada para cada possível saída do sistema elaborado.

Na Figura 3.3 é retratado um circuito que representa a função maioria. O resultado da função maioria é o valor que mais vezes aparece ou a maioria das entradas. A expressão booleana e a tabela verdade obtidas estão ilustradas na Figura 3.2 que mostra a janela invocada pelas funções de análise. Essa janela contém informações específicas sobre processo de construção de uma Tabela Verdade de um circuito. A aba em destaque na Figura 3.2(b) foca somente na área de tabelas, já a Figura 3.2(a) retrata onde as expressões booleanas são geradas.

Existem mais abas que permitem a construção de componentes através da descrição do comportamento das entradas e saídas. Com isso, um usuário pode interagir com a *frame* de análise combinacional, vista na Figura 3.2, e pode personalizar um dispositivo eletrônico digitando ou uma expressão booleana, ou informando uma tabela verdade que descreva as ações esperadas para o circuito que se deseja obter. Ao final basta clicar no botão "*Build Circuit*".

O processo de extrair uma expressão booleana ou tabela verdade através da análise combinacional pode ser associado como uma engenharia avante (*forward engineering*) e o contrário, a obtenção do esquemático, uma engenharia reversa (*reverse engineering*).



(a) Expressão (b) Tabela Verdade

Figura 3.2: *Expressão Booleana e Tabela Verdade da Função Maioria gerada pela análise do Logisim*

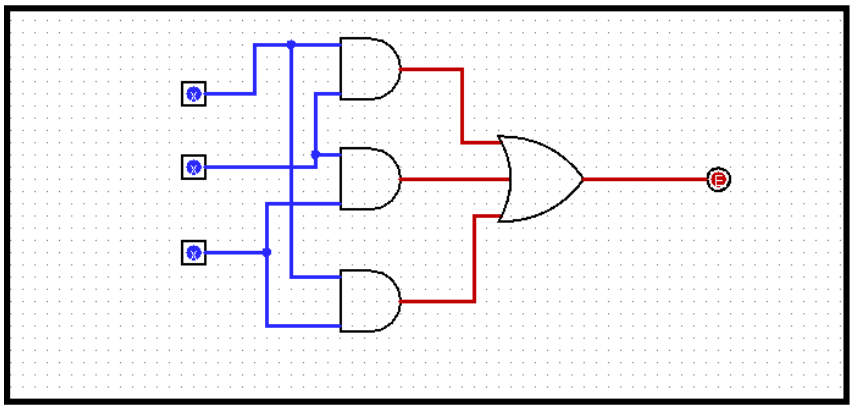


Figura 3.3: *Circuito da Função Maioria modelado no Logisim*

3.1.2 Extração da Álgebra Booleana

O próximo passo para a geração do código inicial é o armazenamento dos dados obtidos através da Análise Combinacional, em modelos que contenham as informações sobre as entradas, nome do circuito, saídas e expressões que regem esse circuito. O pacote "com.uft.logisim.entity" é a parte do *framework* de síntese encarregada por este armazenamento.

A classe "*Input*", mostrada na Figura 3.4, dispõe do nome da variável de entrada, ou o pino de entrada, enquanto a classe "*Output*"(Figura 3.4) representa um pino de saída que contém em sua estrutura o nome da mesma seguida da expressão booleana original e

a expressão minimizada pelo mapa de Karnaugh.

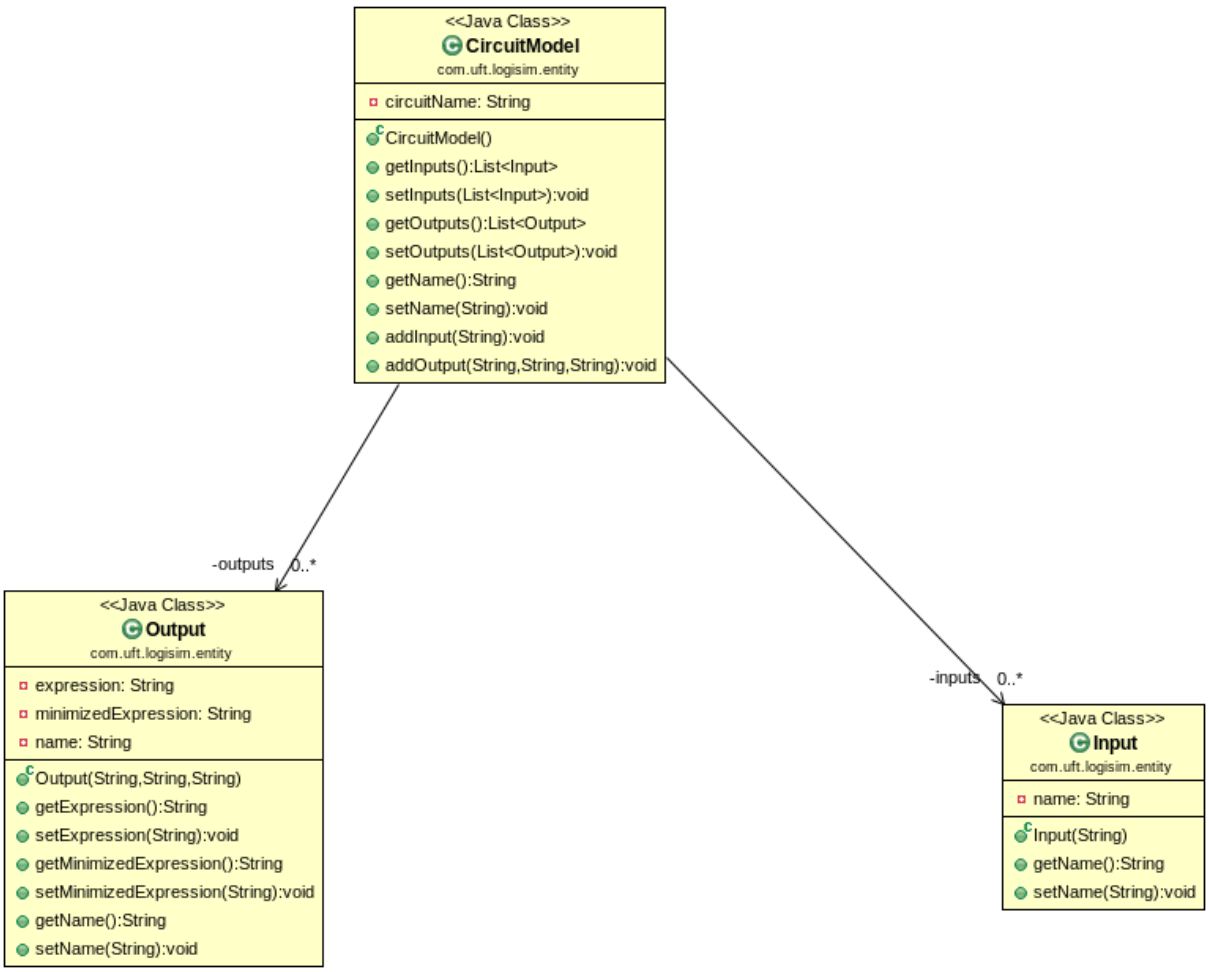


Figura 3.4: *Estruturas de modelo principais do pacote "com.uft.logisim.entity"*

Para a representação do circuito como um todo, há uma classe "*CircuitModel*", descrita na Figura 3.4, que armazena o nome real do circuito criado juntamente com a lista de entradas e saídas do mesmo na forma das estruturas *Input* e *Output*. A estrutura *CircuitModel* e as classes contidas nela foram criadas de modo a serem facilmente adaptadas ao código intermediário criado para este projeto. Todos os detalhes sobre esse código intermediário são descritos na Seção 3.1.3.

O último elemento dentro da representação de modelos é a classe "*CircuitBasic*" (Figura 3.5). Tal modelo foi definido para o auxílio do intercâmbio de dados entre as estruturas do *Logisim* e o *CircuitModel* do *framework* de síntese. O *CircuitBasic* recebe as informações do projeto do *Logisim* em utilização, o circuito desejado e uma estrutura chave-valor dos componentes entrada e saída, onde a chave é representada pelo componente, nesse caso somente pinos, e o valor é seu respectivo nome.

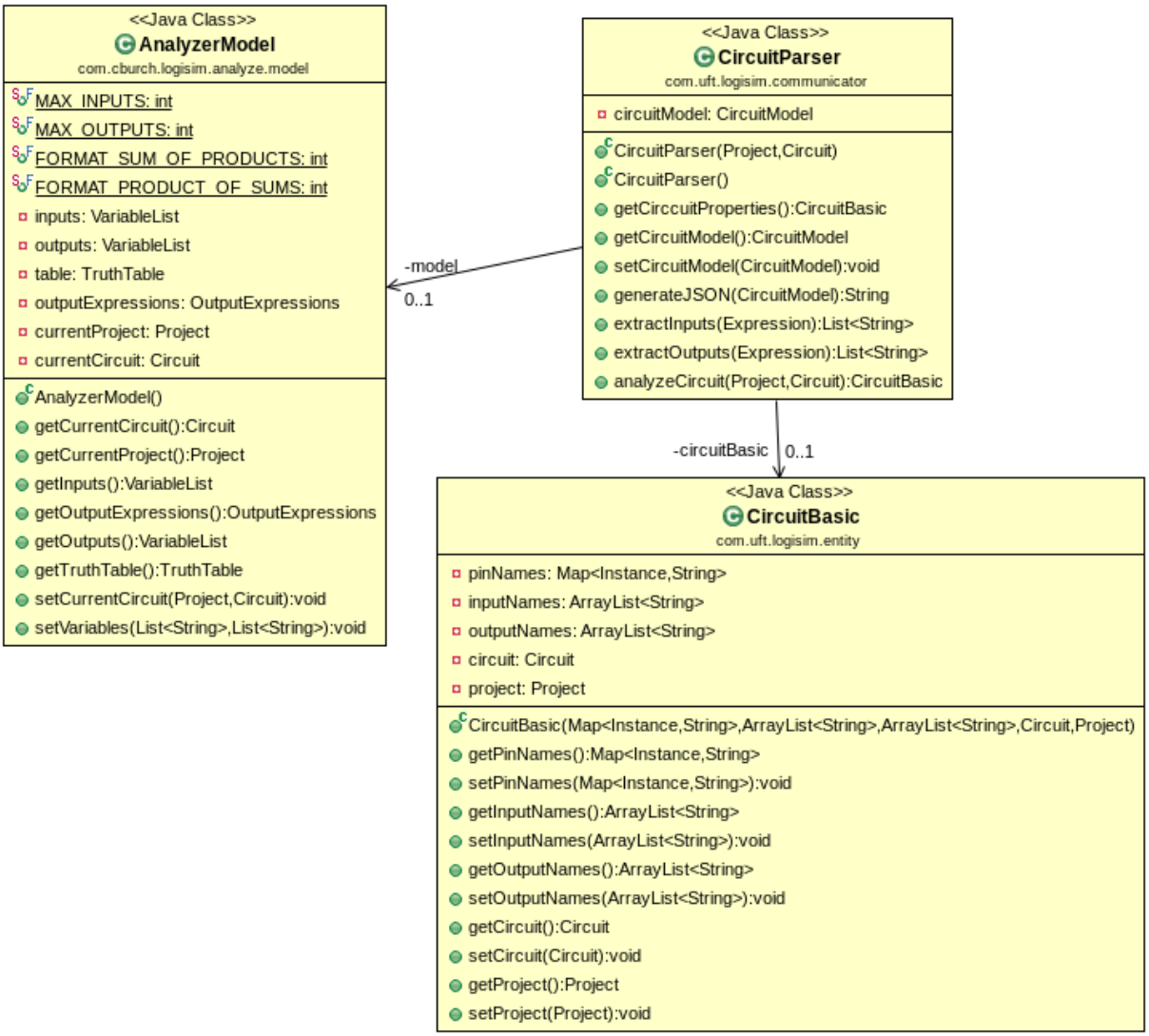


Figura 3.5: *Estrutura do modelo CircuitBasic e o intermediador CircuitParser*

A armazenagem para os modelos ocorre na classe "*CircuitParser*" referente ao pacote "com.uft.logisim.communicator". Tal classe, como observado na Figura 3.5, é considerada uma regra de negócio que intermedeia as solicitações feitas ao *Logisim* pelo *framework* de síntese. No método construtor as solicitações de análise da tabela verdade e expressão booleana do circuito em questão são realizadas através da chamada dos métodos estáticos *computeTable* e *computeExpression* da classe *Analyze* do pacote "com.cburch.logisim.circuit".

Definidas as estruturas de armazenamento e armazenadas as informações, faz-se necessária a conversão desses dados para um formato de arquivo temporário ou código intermediário que é a próxima etapa do *framework*.

3.1.3 Geração e carregamento do código intermediário

A simples transformação de um diagrama de um circuito para uma linguagem de descrição de *hardware* resolve o problema do não detalhamento dos outros domínios do circuito (estrutural, físico e comportamental). Contudo, o problema do tempo necessário para o desenvolvimento dos dispositivos eletrônicos e do número elevado de componentes descritos não são resolvidos pela tradução em si. É necessário realizar um refinamento dos elementos utilizados através de heurísticas de otimização.

De modo a permitir otimizações por meio de ferramentas externas, torna-se interessante a utilização de um código intermediário que será enviado para um minimizador e devolvido ao compilador para a realização das etapas seguintes da tradução do circuito.

Tal código intermediário baseia-se em JSON para a estruturação dos dados. Como abordado na Seção 2.7.1, o código-fonte do ferramental *Logisim* foi desenvolvido em Java o que o torna altamente compatível com o JSON.

Um modelo para o código de intercâmbio de informações foi elaborado especificamente para este projeto seguindo a estrutura da classe *CircuitModel* como referido na Seção 3.1.2. As informações são baseadas em resultados ou saídas da modelagem de dispositivos microeletrônicos. Portanto, para cada solução obtida, um objeto contendo as entradas que geraram tal resultado, as expressões referentes àquela saída e o rótulo da mesma são passadas para o minimizador como no exemplo da Listagem 3.1 a qual descreve o circuito Maioria, retratado na Figura 3.3. O código proveniente da etapa de minimização segue o mesmo formato do modelo enviado inicialmente.

```
1  {
2    "circuitName": "maioria",
3    "inputs": [
4      {
5        "name": "a"
6      },
7      {
8        "name": "b"
9      },
10     {
11       "name": "c"
12     }
13   ],
```

```
14    "outputs": [
15      {
16        "expression": "b c + a c + a b",
17        "minimizedExpression": "(a + b) (a + c) (b + c)",
18        "name": "result"
19      }
20    ]
21  }
```

Listagem 3.1: *Código JSON para a função maioria*

A geração do código intermediário é realizada dentro da classe *CircuiParser*, apresentada na Seção 3.1.2. Uma vez que tabela verdade e expressão booleana foram carregados no objeto "*AnalyzerModel*" da classe *CircuitParser* (Figura 3.5), há o preenchimento do objeto *CircuitModel*, obtendo as variáveis de entrada em uma lista de entradas e as variáveis de saída em uma lista de saídas. Em cada elemento de saída estão inclusas a expressão original e a expressão minimizada aplicada à regra do produto das somas como padronização do formato das expressões.

Uma vez que o objeto *CircuitModel* teve suas variáveis carregadas, ocorre então a geração do código intermediário no formato JSON. Essa geração é acionada através do método *generateJSON* da classe *CircuitParser*. O arquivo é gerado por meio de uma API auxiliar chamada *Gson* [26] a qual realiza conversões de objetos para o formato JSON.

A leitura de um arquivo JSON externo fica a cargo da classe "*MiddleCodeReader*" do pacote "com.uft.logisim.extract" através de um método estático para a leitura do conteúdo do arquivo e conversão do mesmo para o modelo *CircuitModel* através da API Gson [26]. Com o arquivo carregado do *framework* chega ao fim a etapa de obtenção do código inicial, o próximo passo é converter os dados obtidos (entradas e saídas) em um *template* de código que siga uma estrutura similar ao código final, que é o código em SystemC.

3.2 Criação de Template

O código inicial, que também é um código intermediário para a etapa de otimização do circuito, não está no padrão de código desejado em SystemC como pode ser visto na comparação entre a Listagem 3.2 em JSON e 3.3 em SystemC. Portanto, para que a etapa de análise léxica tenha início é preciso adequar as informações obtidas organizadas em um

template modular de um circuito.

```
{
  "circuitName": "maioria",
  "inputs": [
    {
      "name": "a"
    },
    {
      "name": "b"
    },
    {
      "name": "c"
    }
  ],
  "outputs": [
    {
      "expression": "b c + a c + a b",
      "minimizedExpression": "(a + b) (a + c) (b + c)",
      "name": "result"
    }
  ]
}
```

Listagem 3.2: *JSON que descreve um circuito de maioria de 3 entradas*

```
#include <systemc.h>

SC_MODULE (maioria){
        sc_in<sc_logic> a,b,c;
        sc_out<sc_logic> result;

        SC_CTOR (maioria){
                SC_METHOD (process);
                sensitive << a<< b<< c;

        }

        void process(){
                result = (a.read() | b.read()) &
                         (a.read() | c.read()) &
```

```
16                          (b.read() | c.read());
17          }
18 };
```

Listagem 3.3: *SystemC que descreve o circuito maioria de 3 entradas*

Analisando o código final, representado na Listagem 3.3, percebe-se que existe uma estrutura modular denominada "*SC_MODULE*" que é o módulo que envolve um circuito e suas componentes. Tal módulo é descrito por um nome, declaração de variáveis de entrada e saída caracterizadas pelo tipo lógico "*sc_logic*" um construtor denotado por "*SC_CTOR*" e um método que processa as entradas de acordo com o comportamento descrito pela expressão booleana desse circuito.

A adaptação do código em JSON para o template em questão é realizado pela classe "*CompilerCodeParser*" do pacote "com.uft.logisim.extract" através do método "*createCompilerMiddleCode*" que padroniza toda a estrutura em algo mais próximo ao código final a ponto de realizar a análise léxica, a Listagem 3.4 é o resultado do JSON do circuito da função maioria de 3 entradas após a aplicação do template.

```
1 MODULE[maioria] {
2      Inputs:a, b, c;
3      Outputs:result;
4
5      CONSTRUCTOR[maioria] {
6      METHOD[process];
7      SENSITIVE <<a<< b<< c;
8      }
9
10     VOID process[]{
11          result = [a_READ__OR_b_READ_]
12                   _AND_[a_READ__OR_c_READ_]
13                   _AND_[b_READ__OR_c_READ_];
14     }
15 };
```

Listagem 3.4: *Circuito da função maioria adaptado para análise léxica*

As variáveis de entrada e saída são mantidas em seu formato original enquanto os operadores binários são acrescidos de uma sublinha (_) em cada lado da operação. Além disso, há a substituição do símbolo do operador pelo nome da operação booleana em

letras maiúsculas. Palavras reservadas foram definidas totalmente ou iniciando, com letras maiúsculas para a fase de tokenização na análise léxica. Os demais identificadores estão em letras minúsculas. A palavra reservada READ é específica para a etapa de processamento. Tal palavra deve ser seguida de uma variável de entrada. Seu objetivo é indicar que a leitura das entradas do circuito lógico aconteçam somente no momento do processamento,

Uma vez padronizado o código-fonte, o *framework* envia o código gerado para a análise léxica que é a próxima etapa da compilação.

3.3 Análise Léxica

A análise léxica verifica se certos símbolos estão inclusos em um determinado alfabeto[20]. O analisador léxico foi elaborado à partir do conceito de Automato com Pilha ou Automato com Pilha Não-Determinístico. Essa estrutura determina se a fita de entrada se adéqua aos estados existentes no analisador. Esses possíveis estados são representados por tokens.

3.3.1 Definição de Tokens

A definição dos tokens permitidos nessa linguagem é feita na classe abstrata "*BooleanAlgebraPatterns*" do pacote "com.uft.logisim.pattern". A lista de tokens e suas respectivas expressões regulares está disposta na Tabela 3.1

O analisador léxico deste trabalho realiza duas funções:

- verificar a existência do valor de entrada no dicionário da linguagem;
- auxiliar na geração do código final através da substituição da entrada por tokens que são conhecidos pelo SystemC como pode ser observado na Tabela 3.1.

Através da análise da quantidade de tokens permitidos é perceptível que a linguagem para especificação de um componente digital é pequena, contudo o *framework* foi implementado de modo a permitir futuras expansões para um maior detalhamento e geração de códigos além da estrutura modular aqui apresentada.

A fim de permitir uma maior liberdade no trabalho da construção da linguagem, o analisador léxico foi construído por completo especificamente para o código modular em SystemC de um circuito. Sua estrutura foi dividida em fases que são: fase de conversão de expressão para a forma pós-fixa, fase de geração de autômatos finitos não determinísticos

Tabela 3.1: *Modelo da relação entre expressões regulares e tokens*

Token	Expressão Regular
IDENTIFIER	(a+...+z)(a+...+z + 0+...+9)*
[	(
]	)
{	{
}	}
;	;
<<	<<
,	,
=	=
∼	(not_)+(NOT_)
|	(_or_)+(_OR_)
&	(_and_)+(_AND_)
^	(_xor_)+(_XOR_)
.read()	_READ_
sc_in	Inputs:
sc_out	Outputs:
<sc_logic>	LOGIC
sensitive	SENSITIVE
SC_METHOD	MODULE
type	VOID
SC_MODULE	METHOD
SC_CTOR	CONSTRUCTOR

com movimentos vazios através do algoritmo de Thompson, fase de conversão do autômato gerado para um autômato finito determinístico e fase de análise da entrada por todos os autômatos do analisador.

3.3.2 Construção da linguagem

A expressão regular é um conceito formal e prático para a geração de palavras de um alfabeto [27]. Tal conceito, abrange algumas especificações de operações permitidas em sua sintaxe por meio de operadores reservados, que são "+" representando a operação binária de união, o operador "." ou, a ausência de operadores entre símbolos, representando a operação de concatenação e por último o operador unário "*" que define a operação fecho de Kleene. Os demais símbolos (exceto os símbolos '(' e ')') são considerados operandos.

As ordens de precedência dessas operações quando houver a omissão de parênteses devem obedecer as seguintes regras [27]:

- concatenação sucessiva possui prioridade sobre uma outra concatenação ou união;

- concatenação possui precedência sobre a união.

Apesar de um ser humano ser capaz de compreender as regras de precedência de forma visual, a avaliação da notação tradicional (infixa) de uma expressão não especifica à máquina quais são as ordens de precedência, logo, há a necessidade da conversão da forma tradicional para a notação pós-fixa, ou polonesa inversa para que a análise ocorra na ordem linear da expressão regular.

A classe que implementa essa transformação é a "*RegularExpressionConverter*" que faz uso de duas pilhas, uma pilha final e uma pilha de operadores. A pilha é descrita pela classe "*Stack*" definida dentro do pacote "com.uft.logisim.interpret". Os elementos que compõem *Stack* são objetos do Tipo "*Node*". Cada *Node* armazena um símbolo apenas. Além disso, há uma classe abstrata para a avaliação de precedências denominada "*RegularExpressions*" que classifica o operador em valores inteiros de acordo com o seu grau de prioridade. A relação entre os modelos e o conversor de expressões é dada pela Figura 3.6.

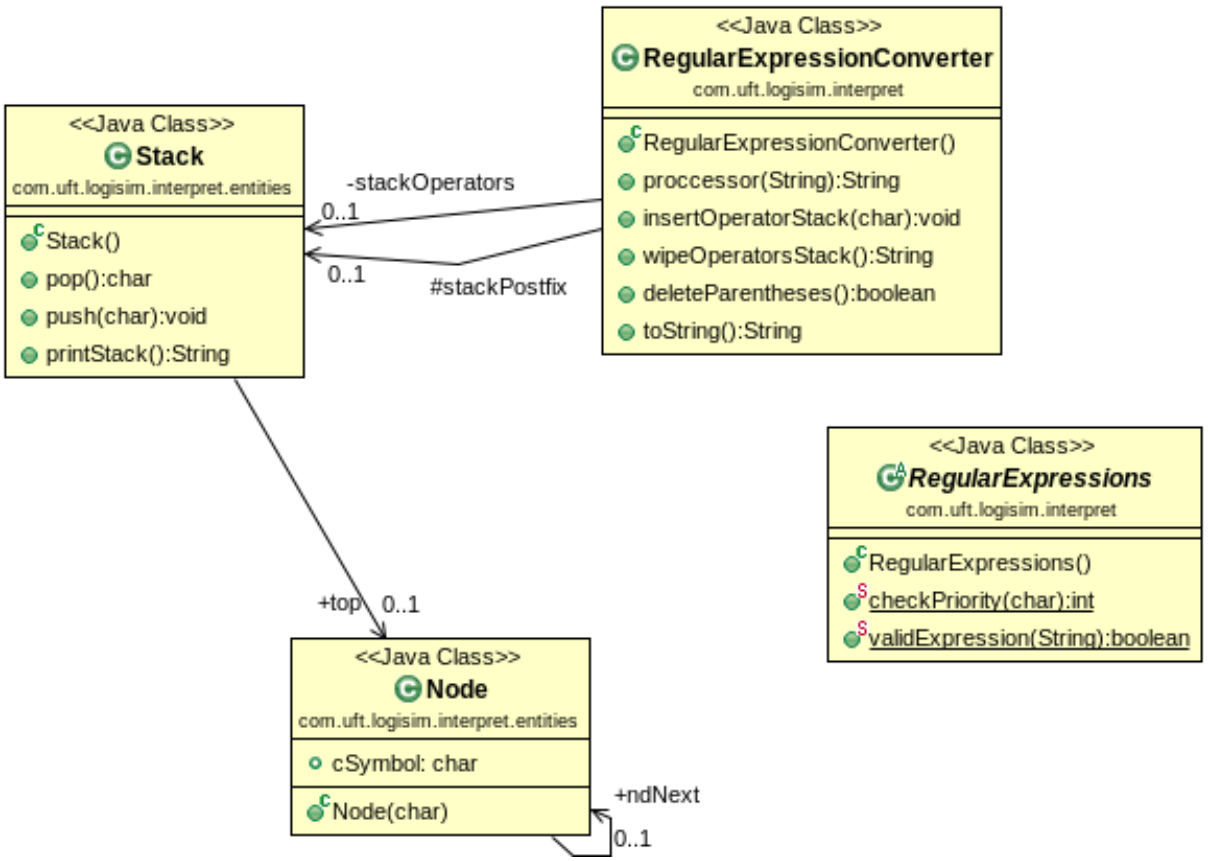


Figura 3.6: *Estrutura e relacionamento de classes para a conversão de expressões regulares*

A conversão tem início com a avaliação símbolo a símbolo da expressão de entrada. O objetivo é que a pilha de operadores fique vazia após as avaliações de precedência. Caso o programa encerre com a pilha não vazia, a expressão pós-fixa não é gerada. O funcionamento da pilha de operadores é dado pelos seguintes passos:

1. Caso o símbolo avaliado seja um operando, o mesmo deve ser empilhado na pilha final;

2. Caso o símbolo seja um operador verificam-se as seguintes condições:

   - se é um operador de fecho de Kleene empilha-se diretamente na pilha final;
   - caso seja um outro operador e sua prioridade seja maior ou igual à prioridade do topo da pilha de operadores, o topo é desempilhado e enviado para a pilha pós-fixa;

3. Se o símbolo lido seja uma abertura de parênteses, o símbolo é empilhado na pilha de operadores;

4. O último caso é se o símbolo lido é um fechamento de parênteses, nesse caso todos os elementos da pilha de operadores são desempilhados e empilhados na pilha pós-fixa até que seja encontrada uma abertura de parênteses.

A Figura 3.7 é a representação das pilhas pós-fixa e de operadores. No exemplo, o elemento em análise é o símbolo de fechamento de parênteses, assim todos os símbolos da pilha de operadores são desempilhados e transferidos para a pilha pós fixa até que se encontre uma abertura de parênteses. Após a conversão, a Expressão "$(a + c) * (bb)$" em sua forma pós-fixa equivale à "$ac + *bb..$".

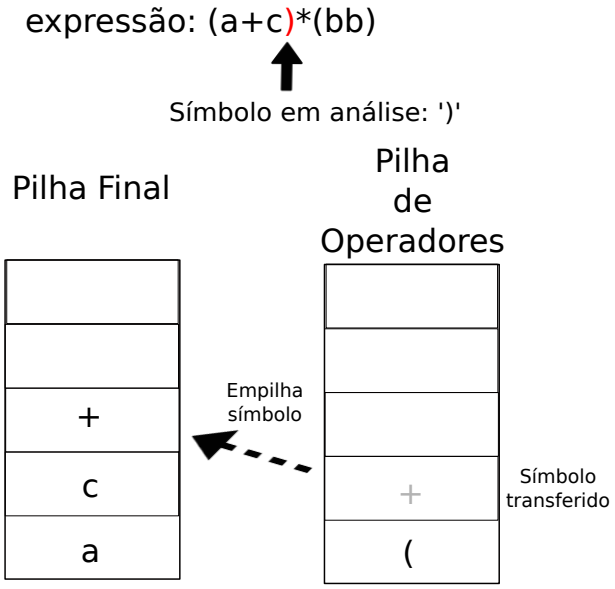


Figura 3.7: *Conversão de infixa para pós-fixa*

A fase seguinte, apresentada na Seção 3.3.3 é a criação de uma máquina de estados para reconhecer palavras que seguem as linguagens representadas por essas expressões.

3.3.3 Construção de Thompson para obtenção de AFN-ξ reconhecedor

A verificação se uma palavra está contida em um dicionário específico pode ser realizada através de uma máquina de estados finita. Tal máquina foi implementada por meio de um Autômato Finito Não Determinístico com Movimentos Vazios (AFN-ξ) pela facilidade de sua construção dada pelo Algoritmo de Thompson [28].

A máquina de estados finito é um sistema que utiliza entradas e saídas discretas, abrangendo um número finito de estados enquanto que um movimento vazio é uma transição

sem leitura de símbolo da entrada [27].

O AFN-ξ é uma composição de um alfabeto (Σ), conjunto de estados possíveis (Q), função de transição parcial (δ), estado inicial (q0) e conjunto de estados finais (F) [27]:

$$M = (\Sigma, Q, \delta, q0, F) \tag{3.1}$$

Descritas as componentes de um AFN-ξ, as mesmas devem ser representadas em forma de implementação. O modelo representativo de um estado é a classe "*State*". A transição é dada pelo modelo "*Transition*" e um Autômato é apresentado por meio da classe "*Automaton*". Uma estrutura auxiliar para o fecho-ϵ, a classe "*EpsilonClosure*", foi construída de modo a evitar sucessivas recursões e para um maior reaproveitamento de dados. O fecho-ϵ faz parte da função de transição de um AFN-ξ e são utilizados para a conversão em um Autômato Finito Determinístico (AFD) . Tais modelos estão dispostos no pacote "com.uft.logisim.automata.model" como visto na Figura 3.8.

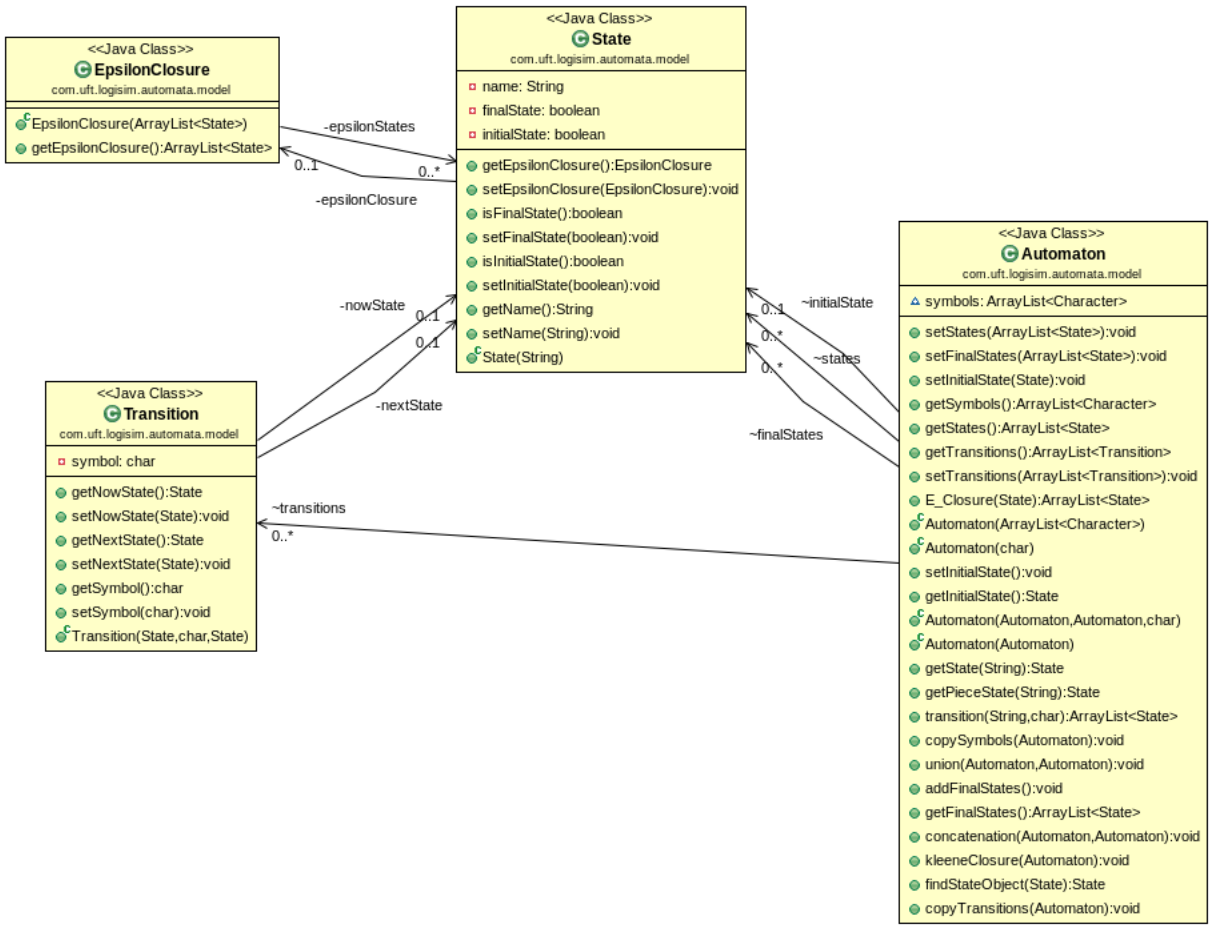
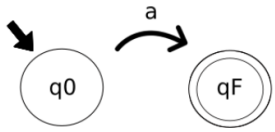


Figura 3.8: *Diagramas de classes de modelos para autômatos*

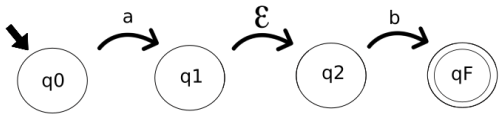
O algoritmo de Thompson consiste na avaliação linear da expressão pós-fixa. Cada operação ou operando é convertido em pequenas estruturas AFN-ξ sendo estas:

- base - o estado inicial ($q0$) possui uma transição do único símbolo desse autômato para um estado final (qF). A Figura 3.9(a) destaca a base de um símbolo “a”;
- concatenação - o estado final do primeiro autômato ($q1$) se liga ao estado inicial ($q2$) do segundo autômato por meio de um movimento vazio. No exemplo da Figura 3.9(b) a expressão avaliada é “ab”;
- união - dois estados são adicionados, um novo estado inicial ($q0$) e um novo estado final (qF). O novo estado inicial ($q0$) possui uma transição vazia para os estados iniciais do primeiro ($q1$) e segundo autômato ($q2$) e os estados finais ($q3$ e $q4$) desses respectivos autômatos, possuem uma transição vazia para o novo estado final (qF). O exemplo aplicado na Figura 3.9(c) apresenta o caso da expressão regular “a+b”;

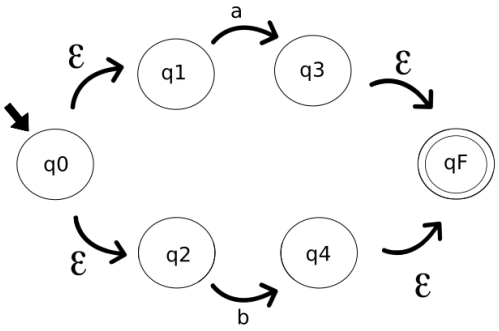
- fecho de Kleene - dois novos estados são criados. O novo estado inicial (*q0*) realiza uma transição vazia para o estado inicial (*q1*) do autômato e para o novo estado final (*qF*). O estado final do autômato base (*q2*) possui uma transição vazia para o novo estado final (*qF*) e um movimento vazio para o estado inicial do autômato base (*q1*). A Figura 3.9(d) mostra a estrutura de um autômato para a expressão "a*" [28].



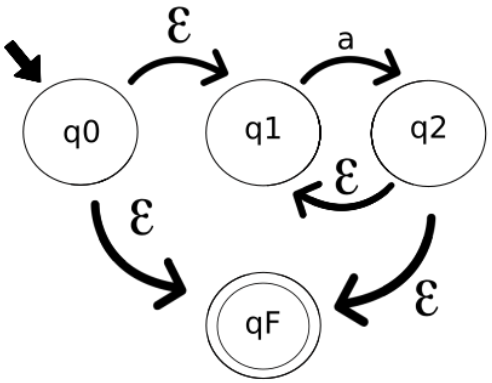
(a) Autômato base para expressão "a"



(b) Estrutura AFN-ξ da concatenação "ab"



(c) Estrutura AFN-ξ da união "a+b"



(d) Estrutura AFN-ξ fecho de Kleene da expressão "a*"

Figura 3.9: *Estruturas utilizadas na construção de Thompson*

O algoritmo de Thompson avalia uma expressão regular e termina sua execução após toda a entrada tiver sido avaliada. De acordo com o símbolo avaliado, caso esse símbolo seja um operador binário, dois autômatos são desempilhados e organizados em um só por meio das estruturas de Thompson mostradas na Figura 3.9, caso seja um operador unário apenas um autômato é desempilhado e re-estruturado e caso seja uma estrutura base, o autômato base é gerado. Os elementos resultado das operações de Thompson são incorporados a uma estrutura única que é empilhada na pilha de autômatos. O Algoritmo 1 descreve a geração do AFN-ξ final para a expressão regular de entrada. O *framework* faz uso deste algoritmo por meio da Classe "*Thompson_Algorithm*" disposta no pacote

"com.uft.logisim.automata.lexical". Ao fim, a pilha de autômatos deverá conter um único elemento que é a junção dos demais em uma única estrutura.

Algoritmo 1 Geração de AFN-ξ

```
1: while expression is not over do
2:     symbol = nextsymbol(expression);
3:     if symbol is an operand then
4:         automaton = base(symbol);
5:         push automaton to automatonStack;
6:     else
7:         if symbol is concatenation then
8:             automaton2 = automatonStack.pop();
9:             automaton1 = automatonStack.pop();
10:            automaton = concatenation(automaton1, automaton2);
11:            push automaton to automatonStack;
12:        else
13:            if symbol is union then
14:                automaton2 = automatonStack.pop();
15:                automaton1 = automatonStack.pop();
16:                automaton = union(automaton1, automaton2);
17:                push automaton to automatonStack;
18:            else
19:                automaton1 = automatonStack.pop();
20:                automaton = kleene(automaton1);
21:                push automaton to automatonStack;
22:            end if
23:        end if
24:    end if
25: end while
```

Uma vez obtido o AFN-ξ é necessário convertê-lo para um AFN e finalmente um sistema de reconhecimento que não permita ambiguidade, um AFD. A conversão de um AFN-ξ para um AFD é possível graças a equivalência entre eles. A prova para esta afirmação consiste em encontrar um AFD que realize o mesmo processamento de um AFN-ξ comportando-se de maneira esperada e chegue aos mesmos resultados [27].

3.3.4 Conversão de um AFN com Moviemntos Vazios em um AFD

A Seção 3.3.3 descreve o processo utilizado para a obtenção do AFN-ξ de uma dada linguagem que deve ser aceita pelo *framework*. Devido aos computadores atuais serem máquinas determinísticas [29], um AFN não torna-se uma boa opção para a aplicação

de um reconhecedor linguístico, pelo contrário seus caminhos alternativos podem gerar ambiguidades. Portanto, é apropriado utilizar a equivalência entre um AFN e um AFD. A prova dessa conversão é realizada através da hipótese que, partindo-se de um AFN qualquer é possível construir um AFD que realiza o mesmo processamento [27].

As transições vazias são eliminadas a fim obter um autômato menor sem movimentos vazios. Para isso, os fecho-ξ são calculados através de uma função de transição de movimentos vazios, definida pelo método "*E_Closure*" da classe *Automaton*.

Um fecho-ξ é um conjunto de estados dado pelo estado atual em união aos fecho-ξ dos estados relacionados por meio de uma transição vazia. A função é denotada pelas seguintes condições:

- FECHO-ξ(q) = q, caso não haja transição vazia;
- FECHO-ξ(q) = q $\bigcup$ ($\bigcup_{i=1}^{n} FECHO{-}\xi(\delta(p_i, \xi))$), onde $\delta(p, \xi)$ é a função de transição com um movimento vazio, p_i é o estado avaliado naquele momento, e n é o último estado da cadeia de transições vazias originada em q.

Por meio de tais regras, tem-se um fecho-ξ para cada estado do autômato que serão usados posteriormente. A conversão de um AFN para um AFD consiste na junção de estados equivalentes, ou seja estados que levam aos mesmos resultados. Portanto, um grupo de estados passa a ser representado por um único estado.

O *framework* dispõe de uma classe para a realização dessa conversão denominada "*AutomatonConversion*" contida no pacote "com.uft.logisim.automata.lexical.transformations". A relação entre as classes para a conversão, minimização e um autômato é apresentada na Figura 3.10.

Figura 3.10: *Pacote com.uft.logisim.automata.lexical.transformations e as transformações presentes*

O processo de conversão pode ser exemplificado por meio da linguagem "*ab*a*". Tal expressão gera o autômato disposto na Figura 3.11. A tabela das transições do mesmo é representada na Tabela 3.2, já com os fecho-ξ calculados, apresentados na coluna "ξ". A Tabela 3.3 mostra a tabela de transições do AFD obtido pelo algoritmo de conversão utilizado.

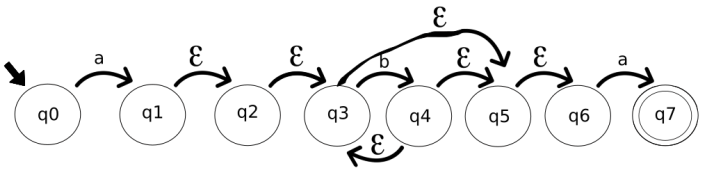


Figura 3.11: *AFN-ξ gerado pela expressão "ab*a"*

O processo de junção de estados equivalentes é um processo iterativo. O algoritmo

parte do estado inicial tomando o fecho-ξ do estado inicial sendo que o resultado do fecho desse estado, passa a ser o novo estado inicial. Tal estado é inserido de imediato à lista de estados. As iterações têm início, tomando os seguintes passos:

1. todos os novos estados do AFD são avaliados a cada iteração;
2. para cada símbolo calcula-se a união das transições dos estados q_i;
3. aplica-se então a união dos fecho-ξ sobre os estados resultantes. Essa união será considerada um novo estado e caso não esteja na lista de estados do AFD é inserido;
4. caso o resultado das uniões seja vazio, um estado de erro é criado e adicionado ao conjunto de estados do AFD. Transições vazias passam a ser transições para esse estado;
5. caso um dos estados do conjunto resultante do fecho seja final, então o novo estado será final;
6. o algoritmo deve prosseguir sua execução até que não hajam mais novos estados a serem avaliados.

Tabela 3.2: *Transições do AFN-ξ "ab*a'*

Q/$\sum$	a	b	ξ
->q0	{q1}		{q0}
q1			{q1 q2 q3 q5 q6}
q2			{q2 q3 q5 q6}
q3		{q4}	{q3}
q4			{q4 q5 q6 q3}
q5			{q5 q6}
q6	{q7}		{q6}
*q7			{q7}

Tabela 3.3: *Transições do AFD "ab*a"*

Q/$\sum$	a	b
->q0	q1q2q3q5q6	qE
q1q2q3q5q6	q7	q3q4q5q6
q3q4q5q6	q7	q3q4q5q6
*q7	qE	qE
qE	qE	qE

O autômato resultante pelo algoritmo de conversão poder ser otimizado através de técnicas que removem estados redundantes. Tal otimização é realizada pela classe "*AutomatonMinimization*" que faz parte do pacote "com.uft.logisim.automata.lexical.transformations".

Para que o algoritmo seja aplicável algumas condições devem ser respeitadas:

1. Deve ser determinístico;
2. Todos os estados existentes devem ser atingíveis ao partir-se do estado inicial;

3. Todo estado existente possui transição para todos os símbolos do alfabeto [27].

Como o AFD gerado pelo *framework* segue as condições para minimização, aplica-se o algoritmo de junção de equivalências. Uma matriz de tamanho n x n é gerada, onde n é a quantidade de estados existentes. Contudo somente o triângulo inferior da matriz é utilizado (Tabela 3.5). Para fins de exemplo o AFD da Tabela 3.3 foi utilizado e teve seus estados renomeados na Tabela 3.4.

Tabela 3.4: *Transições do AFD "ab*a" renomeados*

Q/∑	a	b
->q0	q1	qE
q1	q3	q2
q2	q3	q2
*q3	qE	qE
qE	qE	qE

Tabela 3.5: *Matriz de minimização*

q1				
q2				
*q3				
qE				
	q0	q1	q2	q3

Tabela 3.6: *Preenchimento de estados finais*

q1				
q2				
*q3	x	x	x	
qE				x
	q0	q1	q2	q3

As linhas e colunas representam estados do autômato. Toda célula que seja a junção de um estado final é marcada a princípio (Tabela 3.6). O processamento do algoritmo consiste em realizar a transição da tupla de estados, $\{q_i, q_j\}$, representada pela linha e coluna para todo símbolo do alfabeto.

O resultado das transições é uma tupla de dois estados, $\{p_i, p_j\}$, linha e coluna respectivamente. Se a célula representada por essa tupla estiver marcada, então marca-se a célula $\{q_i, q_j\}$. Caso contrario volta-se para a tupla original e avança-se para o próximo símbolo do alfabeto.

O processamento de uma linha e coluna encerra-se quando uma célula foi preenchida ou após as transições para todos os símbolos tiverem sido verificadas. O algoritmo deve continuar até que não haja preenchimento de células após uma iteração.

Células vazias representam estados equivalentes, ou seja, suas transições levam ao mesmo destino, portanto apenas esses estados se tornam um. No exemplo da Tabela 3.8 q1 e q2 são equivalentes. O AFD exemplo da linguagem "ab*a" ao ser minimizado, passa

Tabela 3.7: 1^a *Iteração*

q1	x			
q2	x			
*q3	x	x	x	
qE		x	x	x
	q0	q1	q2	q3

Tabela 3.8: 2^a *Iteração*

q1	x			
q2	x			
*q3	x	x	x	
qE	x	x	x	x
	q0	q1	q2	q3

a ter apenas quatro estados como é representado pela Tabela 3.9. A minimização reduz o trabalho do analisador léxico durante a verificação de um *Token*, uma vez que diminui a quantidade de estados passíveis de avaliação.

Tabela 3.9: *Automato minimizado da expressão "ab*a"*

Q/$\sum$	a	b
->q0	q1	qE
q1	q2	q1
*q2	qE	qE
qE	qE	qE

3.3.5 O algoritmo de análise léxica

O algoritmo de análise léxica utiliza o conceito de autômato com pilha, contudo o algoritmo implementado pelo *framework* de síntese não faz a junção entre todos os AFDs em um único como muitos algoritmos implementados. Cada autômato, que representa um token, é separado em um bloco de uma lista.

Um detalhe importante a ser ressaltado é que o token "IDENTIFIER" que representa os identificadores válidos para nomes de variáveis, no caso da linguagem em questão, não aceita caracteres especiais ou letras maiúsculas. A normalização é feita ainda na fase de geração de template, abordada na Seção 3.2.

A classe que realiza a análise léxica é rotulada como "*LexicalAnalyzer*" e está inclusa no pacote "com.uft.logisim.automata.lexical.controller". O analisador léxico faz uso de uma lista de tokens e uma lista de autômatos que estão em mesma quantidade, pois um autômato representa um token (Tabela 3.10).

Tabela 3.10: *Relação entre lista de tokens e autômatos*

	0	1	2	3
Autômatos	A1	A2	A3	A4
Tokens	SC_MODULE	=	TYPE	IDENTIFIER

O Algoritmo 2 é uma simplificação do código-fonte do *framework*. Seu critério de reconhecimento é encerrar a análise de uma dada palavra ao atingir-se um estado final. Caso a palavra não seja reconhecida por nenhum autômato um token "ERRO" é adicionado ao *Hash* de saída e o algoritmo continua sua execução. A linguagem foi criada a fim de não permitir ambiguidade como descrito na Seção 3.3.1. A tupla {palavra, token} só é inserida no *Hash* de saída se possuir tamanho maior do que a tupla anterior.

Algoritmo 2 Analisador Léxico

```
1:  while input is not over do
2:      if word is empty then
3:          word = nextword(input);
4:      end if
5:      for automaton in automatons do
6:          for symbol in word do
7:              if there is nextstate(automaton, symbol) then
8:                  recgonizedword+ = symbol;
9:              else
10:                 break the loop;
11:             end if
12:         end for
13:         if automaton.lastState is final then
14:             add (recgonizedword, automaton.token) on outputHash;
15:             automaton move to automatons.first;
16:         end if
17:     end for
18:     word = word − recgonizedword;
19: end while
```

Ao fim, têm-se um *Hash* de saída como representa a Tabela 3.11. Tal saída pode ser convertida em SystemC pela simples substituição dos tokens *IDENTIFIER* e *TYPE* por seu conteúdo original, os demais tokens permanecem os mesmos. Assim o analisador léxico faz a preparação do código final que deve passar pelo analisador sintático e finalização de sua verificação.

Tabela 3.11: *Exemplo de entrada e sua respectiva saída gerada após análise léxica*

Entrada	out	=	a	_READ_	_AND_	b	_READ_	;

Saída	IDENTIFIER	=	IDENTIFIER	.read( )	&	IDENTIFIER	.read( )	;

3.4 Análisador Sintático

A análise sintática do *framework* foi feita por meio da ferramenta ANTLR versão 4 que faz uso da tecnologia *Adaptative* LL(*) ou ALL(*). Tal ferramenta dispõe do potencial para gerar analisadores sintáticos e léxicos dada uma gramática sem recursão indireta à esquerda. A tecnologia ALL(*) realiza uma análise gramatical em tempo de execução encontrando uma sequência de reconhecimento através da navegação dentro da gramática [25].

Nessa última fase deste compilador, há o objetivo de verificar se a construção proveniente do analisador léxico está de acordo com as definições da linguagem SystemC utilizada neste projeto. O ambiente de execução do ANTLR V4 gera, de forma automática, classes para a uma re-análise léxica e sintática dispostas no pacote "com.uft.logisim.syntax.parser" como mostrado na Figura 3.12.

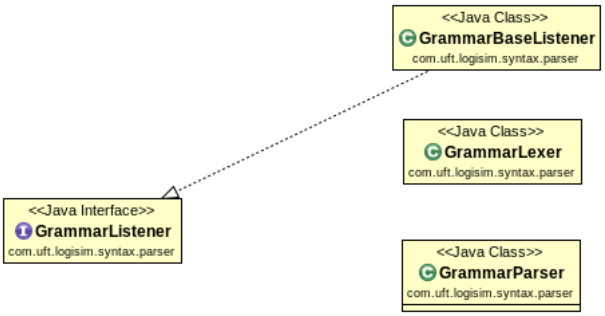


Figura 3.12: *Classes gerados pelo ANTLR*

O processamento da entrada e utilização dos analisadores gerados é feito pelo método estático "*syntaxParse*"da classe "*GrammarRunner*". Tal método cria um analisador léxico a partir da entrada e envia o resultado dos *tokens* para o analisador sintático fazendo a avaliação. Caso sejam encontrados erros, um código de erro é retornado.

A gramática da ferramenta ANTLR entende que identificadores minúsculos são regras de produção enquanto identificadores maiúsculos são tokens. Todos devem estar dispostos em um mesmo arquivo como na Listagem 3.5 e a regra de produção inicial é chamada *r* por padrão.

```
1 grammar Grammar;
2 r: module;
3 module: CODE_HEADER MODULE PARENTHESIS_OPEN IDENTIFIER PARENTHESIS_CLOSE
      CURLY_BRACES_OPEN variables constructorheader processingheader;
```

```
4 variables: inputs outputs;
5 inputs: INPUT LESS_THAN TYPE GREATER_THAN identifierslist;
6 outputs: OUTPUT LESS_THAN TYPE GREATER_THAN identifierslist;
7 identifierslist: separate | IDENTIFIER separate ;
8 separate: COMMA identifierslist| IDENTIFIER SEMICOLON;
9
10 constructorheader: CONSTRUCTOR PARENTHESIS_OPEN IDENTIFIER
      PARENTHESIS_CLOSE CURLY_BRACES_OPEN constructorbody;
11 constructorbody:  processingcall sensitiveline;
12 processingcall: METHOD_CALL PARENTHESIS_OPEN IDENTIFIER
      PARENTHESIS_CLOSE SEMICOLON;
13 sensitiveline: SENSITIVE sensitivelist;
14 sensitivelist: SENSITIVE_OPERATOR IDENTIFIER sensitivelist | SEMICOLON
      CURLY_BRACES_CLOSE;
15
16 processingheader: TYPE IDENTIFIER PARENTHESIS_OPEN PARENTHESIS_CLOSE
      CURLY_BRACES_OPEN processingbody;
17 processingbody: IDENTIFIER EQUALS operation | endprocessing  ;
18 endprocessing: SEMICOLON processingbody | SEMICOLON CURLY_BRACES_CLOSE
      CURLY_BRACES_CLOSE SEMICOLON;
19 operation: binaryoperation |unaryoperation | PARENTHESIS_OPEN operation
      PARENTHESIS_CLOSE | IDENTIFIER READ;
20 binaryoperation: IDENTIFIER READ BINARY_OPERATOR operation;
21 unaryoperation:  UNARY_OPERATOR operation;
22
23 CODE_HEADER: '#include <systemc.h>';
24 READ: '.read()';
25 BINARY_OPERATOR: '&'|'^'|'|';
26 EQUALS: '=';
27 UNARY_OPERATOR: '~';
28 METHOD_CALL: 'SC_METHOD';
29 SENSITIVE: 'sensitive';
30 SENSITIVE_OPERATOR: '<<';
31 CONSTRUCTOR: 'SC_CTOR';
32 INPUT: 'sc_in';
33 OUTPUT: 'sc_out';
34 TYPE: 'sc_logic'|'void';
35 COMMA: ',';
36 SEMICOLON: ';';
```

```
37 IDENTIFIER: ('a'..'z')('a'..'z'|'0'..'9')*;
38 WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
39 LESS_THAN: '<';
40 GREATER_THAN: '>';
41 MODULE: 'SC_MODULE';
42 PARENTHESIS_OPEN: '(';
43 PARENTHESIS_CLOSE: ')';
44 CURLY_BRACES_OPEN: '{';
45 CURLY_BRACES_CLOSE: '}';
```

Listagem 3.5: *Gramática para o SystemC do framework*

O uso de um analisador sintático por meio do ANTLR V4 proporciona uma rápida e fácil avaliação do resultado léxico, impedindo erros básicos na construção de um circuito, tais como saídas ou entradas não definidas. Demais erros como não definição de nome de pinos de entrada ou saída e uso de caracteres especiais em rótulos de variáveis são resolvidos internamente pelo próprio *Logisim* e durante a etapa geração de template (Seção 3.2). Caso a análise léxica do *framework* não associe uma dada entrada a um token da linguagem, o token ERRO é lançado para a etapa de verificação da sintaxe e a re-análise léxica não reconhece tal palavra, impedindo a análise sintática garantindo a confiabilidade dos resultados.

4 Testes e Resultados

Os testes para *benchmarking* do *framework* consistiram em analisar os variados componentes prontos disponibilizados pelo *Logisim* com diferentes quantidades de entradas e número de bits. Como o intuito é a criação da ferramenta e a sua posterior análise de comportamento comparada à entrada, o tempo de compilação não foi utilizado como uma medida de qualificação.

4.1 Interface do Framework e Incorporação ao Logisim

O padrão utilizado pelo *Logisim* para construção da interface de usuário foi a API *Swing*, já inclusa na JDK das versões mais recentes do Java. Assim, a tela de execução da ferramenta de síntese foi construída seguindo tal padronização.

A ferramenta de síntese dispõe das seguintes funcionalidades:

- geração de JSON descritor de um circuito;
- exportação de JSON gerado para arquivo;
- importação de JSON externo para geração de código final;
- síntese de circuito selecionado em SystemC;
- exportação do código final em SytemC;
- exibição de códigos gerados para visualização;
- capacidade de cancelamento de síntese.

Para acessar a ferramenta, o usuário deve navegar pelo barra de menus e ao clicar no menu Projeto, deve selecionar Ferramenta de Síntese. Uma vez iniciada a ferramenta os seguintes passos devem ser tomados para obtenção do código final:

1. seleção de um circuito (Item 1 da Figura 4.1);

2. geração do código em JSON (Item 2 da Figura 4.1);

3. início do processo de síntese (Item 3 da Figura 4.1);

4. geração do arquivo final em SystemC (Item 4 da Figura 4.1).

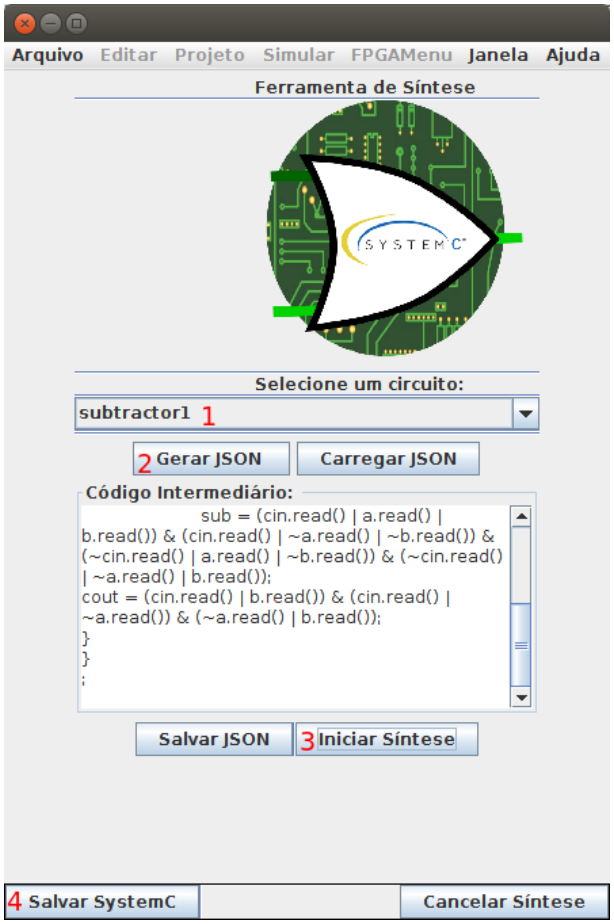


Figura 4.1: *Janela de acesso ao framework de síntese*

A primeira síntese é a mais lenta, pois o analisador léxico e sua respectiva linguagem são criados após a primeira execução. Em caso de erros o analisador sintático lança mensagens para o usuário como apresentado na Figura 4.2.

Com a ferramenta em uso, torna-se possível a simulação dos códigos gerados para verificação de seu funcionamento e se está de acordo com o esperado, a Seção 4.2 traz todo o processo para verificação de um circuito.

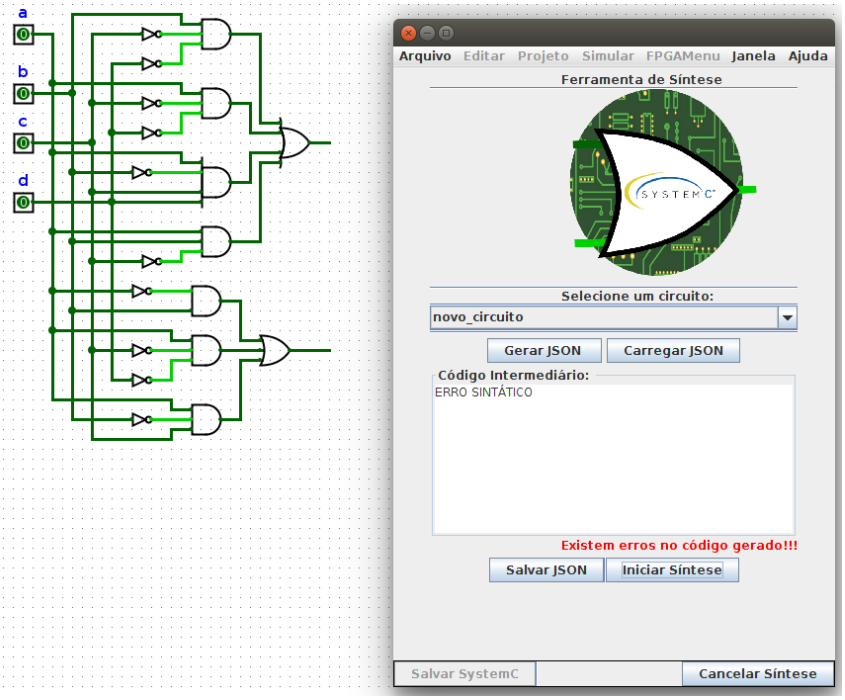


Figura 4.2: *Circuito construído com erros e sua análise pelo framework*

4.2 Simulação e Verificação de circuitos

A verificação de um circuito é feita pela comparação da tabela verdade obtida após a simulação e a tabela verdade obtida pelo *Logisim*. Para a simulação, utiliza-se o compilador *g++* pois o SystemC é um dialeto da linguagem *c++* e pode ser simulado por pura compilação sem a necessidade de outras ferramentas.

Os testes consistiram na modelagem dos esquemáticos na ferramenta *Logisim* geração e exportação do JSON, carregamento de um JSON, síntese de circuito e simulação do código gerado através da análise de toda a sua tabela verdade.

Com o objetivo de demonstrar todo o processo utilizado para os testes, foi abordado neste artigo somente o circuito Multiplexador 4x1 por ser um circuito de tamanho razoável e por ser um modelo bem conhecido na área de circuitos digitais.

4.2.1 Multiplexador 4x1

Um multiplexador é um seletor de dados. Dependendo do código de seleção ou código de controle utilizado, um Mux seleciona a entrada desejada e a envia para a saída. Seu uso é bastante conhecido em ULAs (Unidade Lógica Aritmética) e no monociclo de instruções

do MIPS (Arquitetura de Microprocessador sem estágios de pipeline), desempenhando a tarefa de seleção dos registradores utilizados para a manipulação dos dados [30].

Um Multiplexador ou Mux de 4 entradas e 1 saída utiliza 2 bits de controle. São 4 possíveis combinações desses bits de controle e cada combinação seleciona uma entrada e a envia para a saída. A Figura 4.3 retrata o circuito modelado por meio da área de desenho do *Logisim*, onde *a*, *b*, *c* e *d* são as variáveis de entrada, enquanto *ctrl1* e *ctrl2* são os bits de controle. Quando os bits de controle estão no estado "00" a entrada enviada para a saída é a variável *a*.

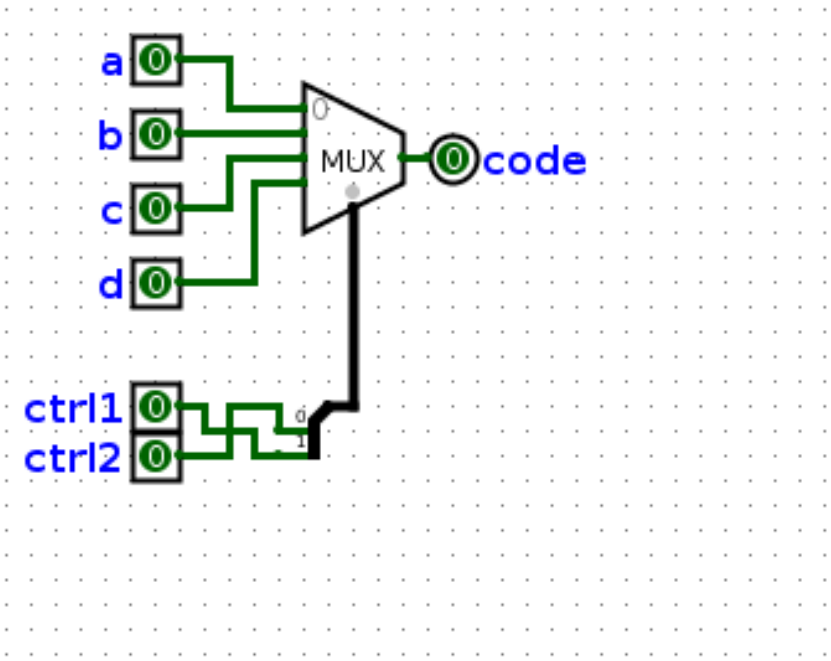


Figura 4.3: *Multiplexador 4x1 construído no Logisim*

Acessando a tela inicial da ferramenta de síntese (Figura 4.1), retoma-se o processo explanado na Seção 4.1 gerando primeiramente o código JSON do esquemático modelado, no caso do Multiplexador 4x1 o seu JSON é equivalente ao mostrado na Listagem 4.1. O resultado da conversão está de acordo com o esperado, pois o arquivo JSON conseguiu sumarizar o comportamento do circuito através da Álgebra de Boole.

```
1 {
2   "circuitName": "mux4x1",
3   "inputs": [
4     {
5       "name": "a"
6     },
```

```
7       {
8         "name": "b"
9       },
10      {
11        "name": "c"
12      },
13      {
14        "name": "d"
15      },
16      {
17        "name": "ctrl1"
18      },
19      {
20        "name": "ctrl2"
21      }
22    ],
23    "outputs": [
24      {
25        "expression": "d ctrl1 ctrl2 + c ctrl1 ~ctrl2 + b ~ctrl1 ctrl2 + a
              ~ctrl1 ~ctrl2",
26        "minimizedExpression": "(a + ctrl1 + ctrl2) (b + ctrl1 + ~ctrl2) (
              c + ~ctrl1 + ctrl2) (d + ~ctrl1 + ~ctrl2)",
27        "name": "code"
28      }
29    ]
30  }
```

Listagem 4.1: *JSON de um Multiplexador 4x1*

Após a geração do JSON é preciso testar a expressão booleana extraída a fim de verificar se o circuito apresentado no arquivo é de fato um Multiplexador 4x1, para isso foi utilizada a Análise Combinacional (Seção 3.1.1) do próprio *Logisim*. A equação extraída para a saída *Code* é dada pela expressão: $(a+ctrl1+ctrl2)(b+ctrl1+ctrl2)(c+ctrl1+ctrl2)(d+ctrl1+ctrl2)$. E ao comparar suas tabelas verdade os resultados foram compatíveis, portanto a geração do JSON e o diagrama gerado obtiveram êxito em seu teste.

Com o JSON pronto tem-se início o processo de síntese do código em SystemC. Primeiramente, antes da análise léxica, há a geração do template que resulta no código

apresentado na Listagem 4.2. É perceptível que o módulo final torna-se similar à forma desejada, o arquivo SystemC. O template não é visível ao usuário como os demais códigos obtidos.

```
1 MODULE[mux4x1] {
2  Inputs:a, b, c, d, ctrl1, ctrl2;
3  Outputs:code;
4  CONSTRUCTOR[mux4x1] {
5    METHOD[process];
6    SENSITIVE <<a<< b<< c<< d<< ctrl1<< ctrl2;
7  }
8  VOID process[]{
9          code = [a_READ__OR_ctrl1_READ__OR_ctrl2_READ_]_AND_[
               b_READ__OR_ctrl1_READ__OR_NOT_ctrl2_READ_]_AND_[
               c_READ__OR_NOT_ctrl1_READ__OR_ctrl2_READ_]_AND_[
               d_READ__OR_NOT_ctrl1_READ__OR_NOT_ctrl2_READ_];
10 }
11 };
```

Listagem 4.2: *Código de um Multiplexador 4x1 após a aplicação do template*

A análise léxica é então realizada sobre o template de entrada e a geração do código final em SystemC é executada resultando em um código final, Listagem 4.3, que deverá passar pelo analisador sintático. Como não houve tokens de erros (<ERRO>) no código gerado, assume-se que todas as palavras utilizadas estão de acordo com o alfabeto permitido. Assim, o código gerado do circuito Mux4x1 obteve êxito na avaliação léxica, resta a avaliar a sintaxe.

```
1 #include "systemc.h"
2
3 SC_MODULE (testbench)
4 {
5 SC_MODULE (mux4x1){
6          sc_in<sc_logic> a,b,c,d,ctrl1,ctrl2;
7          sc_out<sc_logic> code;
8
9          SC_CTOR (mux4x1){
10
11         SC_METHOD (process);
12         sensitive << a<< b<< c<< d<< ctrl1<< ctrl2;
```

```
13          }
14
15          void process(){
16                  code = (a.read() | ctrl1.read() | ctrl2.read()) & (b.
                        read() | ctrl1.read() |
17                   ~ctrl2.read()) & (c.read() | ~ctrl1.read() | ctrl2.read
                        ()) & (d.read() |                          ~ctrl1.read
                        () | ~ctrl2.read());
18          }
19  };
```

Listagem 4.3: *Código gerado pelo Framework para Multiplexador 4x1*

O analisador sintático verifica se é possível iterar sobre o código analisado por meio de sua gramática. Caso seja possível, o próprio ANTLR demonstra a árvore utilizada para a verificação da entrada. Caso não haja caminhos possíveis, a execução da análise termina mostrando o ponto onde o iterador não encontrou solução. A árvore do Multiplexador obteve êxito e o caminho utilizado é apresentado na Figura 4.4, portanto a geração do código final para o Mux 4x1 está de acordo com o esperado.

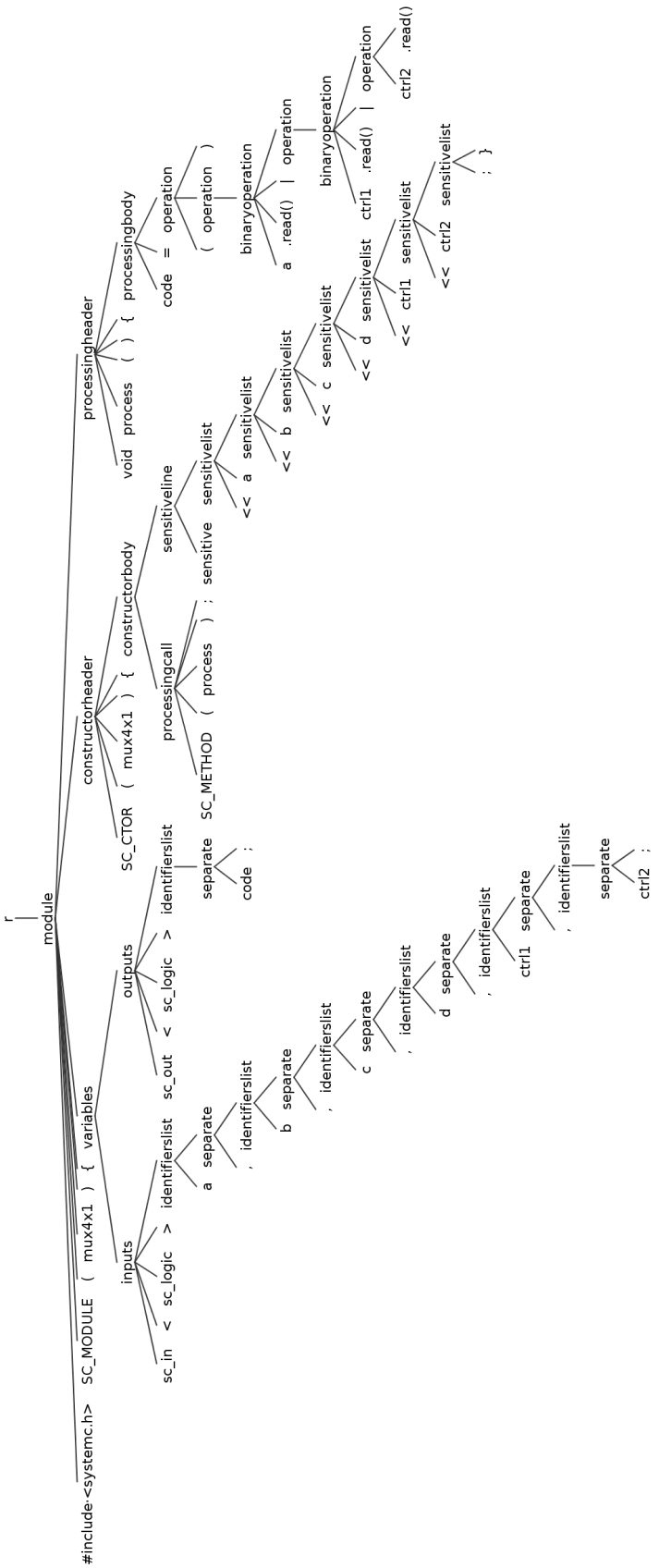


Figura 4.4: *Árvore sintática construída pelo ANTLR V4 para um multiplexador 4x1*

A simulação do módulo construído é executada através de um arquivo *testbench* que contém todas as entradas da tabela verdade, que para o exemplo do Mux 4x1, é um total de 2^6 ou 64 combinações diferentes. O arquivo de testes dispõe de um método de processamento que gera casos de testes para o circuito e exibe seu posterior resultado. Em cada caso de teste há um atraso de 15 nanosegundos como convenção, a fim de impedir que a mudança de estado das entradas ocorra de forma descontrolada. A vinculação do arquivo de testes ao módulo construído fica a cargo do arquivo principal (Listagem 4.5). Tal arquivo além de vincular as variáveis de entrada e saída, especifica o tempo total que devem durar os testes, caso hajam problemas na execução, a simulação se encerra no tempo indicado (No exemplo 1000 nano segundos). O arquivo de testes reduzido para o caso do Multiplexador 4x1 é dado na Listagem 4.4.

```
1 #include "systemc.h"
2
3 SC_MODULE (testbench)
4 {
5
6         sc_out<sc_logic> a,b,c,d,ctrl1,ctrl2;
7         sc_in<sc_logic> code;
8
9
10        SC_CTOR (testbench)
11        {
12                SC_THREAD (process);
13        }
14
15
16        void process()
17        {
18
19                a = SC_LOGIC_0;
20        b = SC_LOGIC_0;
21        c = SC_LOGIC_0;
22        d = SC_LOGIC_0;
23        ctrl1 = SC_LOGIC_0;
24        ctrl2 = SC_LOGIC_0;
25        wait (5, SC_NS);
26        wait (10, SC_NS);
```

```
27          print();
28
29          a = SC_LOGIC_0;
30          b = SC_LOGIC_0;
31          c = SC_LOGIC_0;
32          d = SC_LOGIC_0;
33          ctrl1 = SC_LOGIC_0;
34          ctrl2 = SC_LOGIC_1;
35          wait (5, SC_NS);
36          wait (10, SC_NS);
37          print();
38
39          a = SC_LOGIC_0;
40          b = SC_LOGIC_0;
41          c = SC_LOGIC_0;
42          d = SC_LOGIC_0;
43          ctrl1 = SC_LOGIC_1;
44          ctrl2 = SC_LOGIC_0;
45          wait (5, SC_NS);
46          wait (10, SC_NS);
47          print();
48
49          a = SC_LOGIC_0;
50          b = SC_LOGIC_0;
51          c = SC_LOGIC_0;
52          d = SC_LOGIC_0;
53          ctrl1 = SC_LOGIC_1;
54          ctrl2 = SC_LOGIC_1;
55          wait (5, SC_NS);
56          wait (10, SC_NS);
57          print();
58          //{...}
59
60          sc_stop();
61
62          }
63
64          void print()
65          {
```

```
66                  cout << "Inputs -> (a,b,c,d,ctrl1,ctrl2): ";
67                  cout << a.read() << b.read() << c.read() << d.read() <<
                        ctrl1.read() << ctrl2.read();
68                  cout << " Outputs -> (code): " << code.read() << endl;
69          }
70
71 };
```

Listagem 4.4: *Arquivo de testes para um Mux 4x1*

```
1 #include "mux4x1.cpp"
2 #include "mux4x1_tst.cpp"
3
4 int sc_main(int argc, char* argv[])
5 {
6          sc_signal<sc_logic> a,b,c,d,ctrl1,ctrl2, code;
7          mux4x1 mux("mux4x1");
8          mux << a << b << c<< d << ctrl1 << ctrl2 << code;
9
10         testbench test1("TestBench1");
11         test1  << a << b << c<< d << ctrl1 << ctrl2 << code;
12
13         sc_start(1000,SC_NS);
14
15         return(0);
16 }
```

Listagem 4.5: *Arquivo principal que vincula um Mux 4x1 aos casos de testes*

Após a simulação, foi observado que o código SystemC gerado pelo *framework* e o esquemático possuem comportamentos equivalentes, já que suas tabelas verdades são compatíveis, verificando-se assim a confiabilidade do resultado obtido. Tal comparação pode ser vista pelos gráficos de forma de onda das duas modelagens:*Logisim* e SystemC, Figuras 4.5 e 4.6 respectivamente.

Na comparação dos dois diagramas de forma de onda, observa-se que o diagrama gerado pelo *Logisim* não utiliza o atraso nos testes, pois seu funcionamento desconsidera todos os atrasos (Figura 4.5), contudo a simulação do SystemC exige que haja um atraso durante a realização dos testes como visto no rodapé do diagrama da Figura 4.6. Nesse caso deve-se multiplicar a medida de tempo por 15 nanosegundos que é o atraso definido.

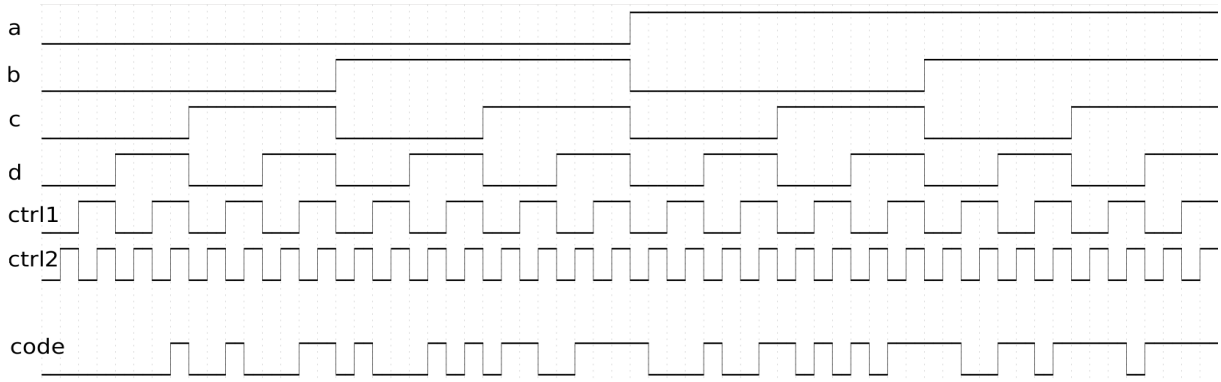


Figura 4.5: *Comportamento de um Multiplexador 4x1 após simulação no Logisim*

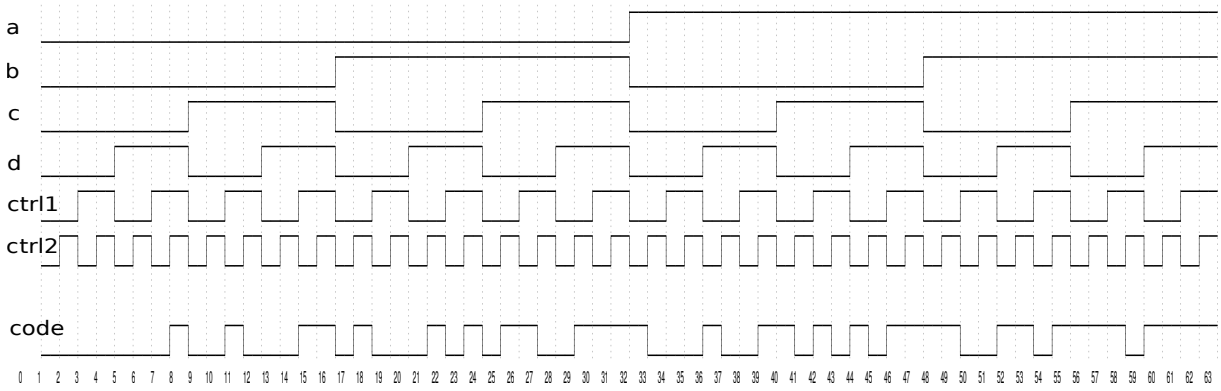


Figura 4.6: *Comportamento de um Multiplexador 4x1 após simulação no SystemC*

Por exemplo, quando os bits de de controle estão no estado *10* e a entrada *c* está em nível alto a saída *code* será alta no instante de tempo de 156 nanosegundos.

4.3 Testes

Todos os testes realizados seguiram os passos descritos anteriormente e foram executados em um computador com processador i7 3^a geração 1.84 Ghz, 16 Gb de RAM rodando sobre uma distribuição Linux Ubuntu 16.04 64 bits. A versão do SystemC utilizada foi a 2.3.1 que é versão estável mais recente.

No total foram modelados 41 circuitos diferentes e para cada um deles foram gerados: um arquivo no formato JSON, um arquivo SystemC do circuito, um arquivo de testes e um arquivo principal ou arquivo *Main*, os três últimos no formato *cpp*.

A Tabela 4.1 contém os resultados de todos os circuitos testados. Estão dispostos nessa tabela o tipo de circuito testado e suas respectivas quantidades de entradas e saídas.

A coluna *Logisim* é uma menção à produção do diagrama esquemático indicando se o mesmo foi construído e desempenha a função desejada, por exemplo se um somador de 1 bit realmente faz a soma. As colunas JSON e SystemC descrevem se a geração dos dois arquivos obteve êxito, enquanto a coluna Simulação é o resultado dos testes do código SystemC produzido após a simulação via *g++*.

Todos os circuitos testados apresentaram o comportamento esperado, não houve erros de geração dos códigos em nenhuma etapa e o compilador teve sucesso no reconhecimento das entradas de dados e as traduziu de forma equivalente. Surgiram divergências no tempo de compilação quando expressões booleanas com mais de 100 linhas foram testadas.

Tabela 4.1: *Resultados dos 41 circuitos testados*

Circuitos	Entradas	Saídas	Logisim	JSON	SystemC	Simulação
Somador 1 bit	3	2	ok	ok	ok	ok
Somador 2 bits	5	3	ok	ok	ok	ok
Somador 3 bits	7	4	ok	ok	ok	ok
Somador 4 bits	9	5	ok	ok	ok	ok
Somador 5 bits	11	6	ok	ok	ok	ok
Subtrator 1 bit	3	2	ok	ok	ok	ok
Subtrator 2 bits	5	3	ok	ok	ok	ok
Subtrator 3 bits	7	4	ok	ok	ok	ok
Subtrator 4 bits	9	5	ok	ok	ok	ok
Subtrator 5 bits	11	6	ok	ok	ok	ok
Somabits 1 bit	2	2	ok	ok	ok	ok
Somabits 2 bits	4	3	ok	ok	ok	ok
Somabits 3 bits	6	3	ok	ok	ok	ok
Somabits 4 bits	8	4	ok	ok	ok	ok
Somabits 5 bits	10	4	ok	ok	ok	ok
Comparador 1 bit	2	3	ok	ok	ok	ok
Comparador 2 bits	4	3	ok	ok	ok	ok
Comparador 3 bits	6	3	ok	ok	ok	ok
Comparador 4bits	8	3	ok	ok	ok	ok
Comparador 5 bits	10	3	ok	ok	ok	ok
Demux1x2 1 bit	2	2	ok	ok	ok	ok
Demux1x2 2 bits	3	4	ok	ok	ok	ok
Demux1x2 3 bits	4	6	ok	ok	ok	ok
Demux1x2 4 bits	5	8	ok	ok	ok	ok
Demux1x2 5 bits	6	10	ok	ok	ok	ok
Divisor 2 bits	6	4	ok	ok	ok	ok
Divisor 3 bits	9	6	ok	ok	ok	ok
Divisor 4 bits	12	8	ok	ok	ok	ok
Paridade par	2	1	ok	ok	ok	ok
Maioria	3	1	ok	ok	ok	ok
Multiplicador 2 bits	6	4	ok	ok	ok	ok
Multiplicador 3 bits	9	6	ok	ok	ok	ok
Multiplicador 4 bits	12	8	ok	ok	ok	ok
Mux2x1 1 bit	3	1	ok	ok	ok	ok
Mux2x1 2 bits	5	2	ok	ok	ok	ok
Mux2x1 3 bits	7	3	ok	ok	ok	ok
Mux2x1 4 bits	9	4	ok	ok	ok	ok
Mux2x1 5 bits	11	5	ok	ok	ok	ok
Paridade ímpar	3	1	ok	ok	ok	ok
Mux4x1	6	1	ok	ok	ok	ok
Demux1x4	3	4	ok	ok	ok	ok

5 Conclusão

Neste trabalho foi apresentado uma nova solução para síntese de circuitos combinacionais provenientes da ferramenta CAD *Logisim* em uma linguagem de descrição de *hardware* que nesse caso foi o SystemC. Todo o processo de produção desse *software*, caracterizado como *framework*, foi descrito de forma detalhada a fim de documentar e permitir melhorias futuras.

Resumidamente a primeira parte deste trabalho buscou situar o leitor sobre os conceitos de sistemas digitais e a definição de síntese na área de circuitos, as ferramentas utilizadas e o motivo da escolha do *Logisim* e do SystemC. Logo em seguida, todo o passo para a construção do *software* é explanado através de exemplos de algoritmos e técnicas utilizadas e descrição por imagens de modo a maximizar o entendimento por parte do leitor. Todas as etapas foram agrupadas em um único diagrama que representa o funcionamento completo do *framework*, a Figura 3.1.

Na segunda parte deste trabalho é modelado um circuito Multiplexador 4x1 (quatro por um) a fim de demonstrar como foram realizados os testes e sob quais medidas os resultados foram qualificados. Houve uma série de testes, totalizando 41 circuitos testados no total e todos obtiveram êxito, demonstrando a funcionalidade e confiabilidade da ferramenta.

Apesar de o *framework* ser capaz de realizar a sua funcionalidade de forma efetiva, o mesmo não é eficiente. A geração de template é um processo lento, principalmente em casos de expressões booleanas consideradas longas (mais de 100 linhas de expressão). O próximo passo seria otimizar o compilador definindo uma nova maneira de realizar o mapeamento do template ou passando a responsabilidade para o analisador léxico de forma a identificar variáveis em processamento e tokenizá-las como leitura de dados.

Além da otimização da compilação, é encorajado que o *framework* passe a gerar os casos de testes e o arquivo *Main* em conexão com o circuito obtido de forma a diminuir mais ainda o esforço por parte do *designer* de sistemas digitais. Como o escopo deste trabalho era apenas a síntese dos esquemáticos projetados na ferramenta *Logisim*, optou-se por não acoplar tais funcionalidades ao *framework* e focar somente no objtivo principal.

O objetivo final proposto por este projeto foi alcançado, com um código HDL em Sys-

temC confiável o projetista reduz o esforço e o tempo para produzir circuitos e pode então fazer a síntese a nível de *hardware* em uma FPGA compatível com o tecnologia SystemC. Seu acesso pode ser feito de forma simples em uma ferramenta gratuita permitindo que usuários de todo o mundo se interessem no campo de síntese de circuitos. Portanto, por meio dessas afirmações, conclui-se que a solução oferecida atingiu o objetivo proposto, sendo assim o objetivo deste trabalho foi alcançado.

Referências Bibliográficas

[1] MICHELI, G. D. *Synthesis and Optimization of Digital Circuits*. 1st. ed. [S.l.]: McGraw-Hill Higher Education, 1994. ISBN 0070163332.

[2] TOCCI, R. et al. *Aplicaçoes, 10a Ediçao*. [S.l.]: Sao Paulo: Pearson Prentice Hall, 2008.

[3] RIESGO, T.; TORROJA, Y.; TORRE, E. de la. Design methodologies based on hardware description languages. *IEEE Transactions on Industrial Electronics*, IEEE, USA, v. 46, n. 1, p. 3–12, 1999.

[4] MAXFIELD, C. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. 1st. ed. Newton, MA, USA: Newnes, 2004. ISBN 0750676043, 9780750676045.

[5] GAJSKI, D. D.; KUHN, R. H. Guest editors' introduction: New vlsi tools. *Computer*, IEEE, USA, v. 16, n. 12, p. 11–14, 1983.

[6] KISSION, P.; DING, H.; JERRAYA, A. *VHDL based design methodology for hierarchy and component re-use*. Brighton, Great Britain: IEEE, 1995. 470-475 p.

[7] DOBOLI, A. et al. *Behavioral Synthesis of Analog Systems Using Two-layered Design Space Exploration*. New Orleans, Louisiana, USA, pages=951-957, year=1999, publisher=ACM,: [s.n.].

[8] HANSEN, C.; BRINGMANN, O.; ROSENSTIEL, W. A vhdl reuse component model for mixed abstraction level simulation and behavioral synthesis. In: _____. USA: Springer, 2001. v. 1, p. 131–144.

[9] GANESAN, S.; VEMURI, R. *Behavioral Partitioning in the Synthesis of Mixed Analog-digital Systems*. Las Vegas, Nevada, USA: ACM, 2001. 133–138 p. Disponível em: <http://doi.acm.org/10.1145/378239.378373>.

[10] DOBOLI, A.; VEMURI, R. Behavioral modeling for high-level synthesis of analog and mixed-signal systems from vhdl-ams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, USA, v. 22, n. 11, p. 1504–1520, 2003.

[11] GIELEN, G.; MCCONAGHY, T.; EECKELAERT, T. *Performance Space Modeling for Hierarchical Synthesis of Analog Integrated Circuits*. Anaheim, California, USA: ACM, 2005. 881-886 p.

[12] CHU, L. P. *RTL Hardware Design Using VHDL:Coding for Efficiency, Portability, and Scalability*. 1st. ed. [S.l.]: Wiley-IEEE Press, 2006.

[13] GUIMARãES, R. R. do N. H. Modelagm de sistemas heterogêneos utilizando frameworks baseado em moc. *Universidade de Brasília, Faculdade de Gama*, Universidade de Brasília, Faculdade de Gama, Brasília, 2014.

[14] SHACKLEFORD, B. et al. Synthesis of minimum-cost multilevel logic networks via genetic algorithm. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, The Institute of Electronics, Information and Communication Engineers, v. 83, n. 12, p. 2528–2537, 2000.