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

RENATO ALFRED SALLA BÖHLER

LOGOSSIM: UM SIMULADOR LÓGICO WEB DE CÓDIGO ABERTO

TRABALHO DE CONCLUSÃO DE CURSO

CURITIBA
2021

RENATO ALFRED SALLA BÖHLER

LOGOSSIM: UM SIMULADOR LÓGICO WEB DE CÓDIGO ABERTO

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

Orientador: Daniel Rossato de Oliveira
Universidade Tecnológica Federal do Paraná

Coorientador: Ricardo Umbria Pedroni
Universidade Tecnológica Federal do Paraná

CURITIBA
2021

RENATO ALFRED SALLA BÖHLER

LOGOSSIM: UM SIMULADOR LÓGICO WEB DE CÓDIGO ABERTO

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 04 de agosto de 2021

DANIEL ROSSATO DE OLIVEIRA
Mestrado em Engenharia Elétrica e Informática Industrial pela Universidade Tecnológica Federal do Paraná
Universidade Tecnológica Federal do Paraná (UTFPR)

GUSTAVO BENVENUTTI BORBA
Doutorado em Engenharia Elétrica e Informática Industrial pela Universidade Tecnológica Federal do Paraná
Universidade Tecnológica Federal do Paraná (UTFPR)

ADOLFO GUSTAVO SERRA SECA NETO
Doutor em Ciências da Computação pela Universidade de São Paulo (USP)
Universidade Tecnológica Federal do Paraná (UTFPR)

CURITIBA

2021

AGRADECIMENTOS

A toda minha família, primeiramente, por sempre me apoiar e inspirar, e em especial à minha esposa por nunca deixar que eu desista dos meus sonhos.

A todos meus professores, pela orientação, dedicação que foram sempre fonte de incentivo e motivação.

A todos meus colegas, pelo companheirismo e amizade durante esta jornada.

Young man, in mathematics you don't understand things. You just get used to them. (NEUMANN, John Von, 1955).

RESUMO

BÖHLER, Renato. Logossim: um simulador lógico web de código aberto. 2021. 93 f. Trabalho de Conclusão de Curso – Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Curitiba, 2021.

Circuitos digitais formam a base de todos sistemas digitais que permeiam nossas vidas. O conhecimento em lógica digital é considerado fundamental nos campos da Informática, Engenharias e Ciência da Computação. O uso de ferramentas de simulação auxiliam estudantes em todo o mundo a compreender melhor este tópico, colocando seus conhecimentos à prova. Este trabalho apresenta o projeto Logossim, um simulador lógico de código aberto feito para a *web*, fácil de usar e facilmente extensível. Este projeto foi construído para se assemelhar ao Logisim, um aclamado *software* de simulação digital. Para isto, foram desenvolvidos um motor de simulação, um conjunto de componentes lógico digitais simuláveis e uma interface gráfica. O projeto foi desenvolvido em linguagem JavaScript, sendo React, React Diagrams e Web Workers as principais tecnologias. Como resultado, o Logossim está publicamente disponível na *internet*, podendo ser utilizado para estudos ou validações de circuitos lógico digitais por qualquer pessoa.

Palavras-chave: Simulação. Circuitos digitais. Código aberto.

ABSTRACT

BÖHLER, Renato. Logossim: open source web logic simulator. 2021. 93 f. Trabalho de Conclusão de Curso – Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Curitiba, 2021.

Digital circuits are the basis of all digital systems that permeate our lives. Knowledge in digital logic is considered fundamental in the fields of Informatics, Engineering and Computer Science. The use of simulation tools helps students around the world to better understand this topic, putting their knowledge to test. This work presents the Logossim project: an open source logic simulator created for the web, easy to use and easily extensible. This project was built to resemble Logisim, an acclaimed digital simulation software. To achieve this, a simulation engine, a set of digital logic components and a graphical interface were developed. This project was developed using JavaScript and using React, React Diagrams and Web Workers as its main technologies. As a result, Logossim is publicly available on the internet and can be used by anyone to study or to validate digital logic circuits.

Keywords: Simulation. Digital Circuits. Open source.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AJAX	*Asynchronous JavaScript and XML*
ANSI	*American National Standards Institute*
CI	*Continuous Integration*
CSS	*Cascading Style Sheets*
DOM	*Document Object Model*
HTML	*Hypertext Markup Language*
JSON	*JavaScript Object Notation*
JSX	*JavaScript XML*
LED	*Light Emitting Diode*
NPM	*Node Package Manager*
PDF	*Portable Document Format*
RAM	*Random Access Memory*
RIA	*Rich Internet Application*
ROM	*Read Only Memory*
SSD	*Seven Segments Display*
TSX	*TypeScript XML*
URL	*Uniform Resource Locator*
UUID	*Universally Unique Identifier*
XML	*Extensible Markup Language*

LISTA DE SÍMBOLOS

x	Valor flutuante
e	Valor de erro

LISTA DE ALGORITMOS

SUMÁRIO

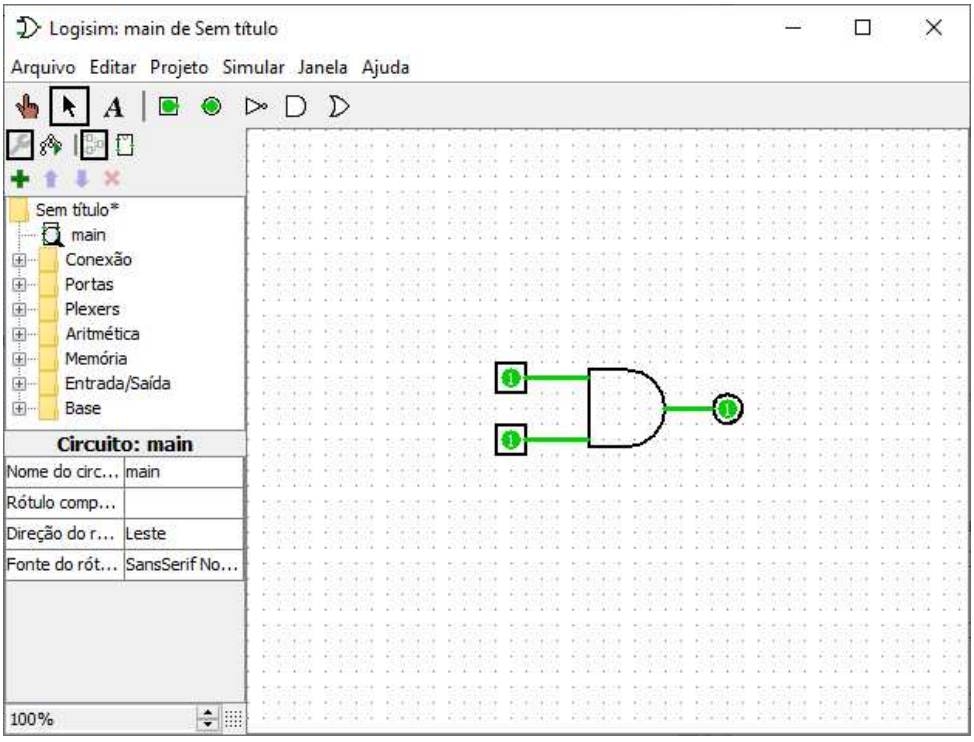
1 INTRODUÇÃO

Este capítulo possui algumas considerações iniciais, como a motivação e os objetivos da realização deste trabalho.

1.1 MOTIVAÇÃO

Este trabalho foi fortemente inspirado no programa Logisim[1], um simulador digital de circuitos lógicos utilizado como ferramenta educacional por alunos em mais de 100 instituições de ensino ao redor do mundo (BURCH, 2013). O Logisim é uma aplicação desktop desenvolvida em Java e disponível nas principais plataformas (MacOS, Windows e Linux) em código aberto. A Figura 1 exibe a interface gráfica deste simulador.

Figura 1 – Logisim sendo executado no Windows



Fonte: Autoria própria

Apesar de sua imensa popularidade, o projeto foi descontinuado em outubro de 2014[2]. Desde então, várias ramificações do Logisim surgiram, sendo o projeto mais notável o `logisim-evolution`[3], que, além de incluir diversas correções de *bugs*, também adiciona uma série de novas funcionalidades ao programa.

[1] <http://www.cburch.com/logisim/pt/index.html>
[2] <http://www.cburch.com/logisim/retire-note.html>
[3] <https://github.com/reds-heig/logisim-evolution>

No entanto, no decorrer das últimas décadas, a adoção em massa da *World Wide Web* tem alterado a maneira com que os usuários interagem com aplicações. Desde a incorporação de novas tecnologias como o AJAX as aplicações *web* tem sido cada vez mais interativas e dinâmicas, viabilizando o surgimento de *Rich Internet Applications* (RIAs): aplicações executadas num navegador *web*, que não necessitam ser instaladas na máquina local, mas ainda assim possuem as funcionalidades de uma aplicação *desktop* tradicional (ROSSI et al., 2007).

Um estudante que está aprendendo lógica digital, no entanto, pode ter dificuldades para encontrar uma aplicação *web* para simulação durante seus estudos. Boa parte das opções disponíveis são pagas ou são exageradamente complexas para alguém em nível iniciante. A proposta deste trabalho é preencher esta lacuna, disponibilizando um simulador *web* que seja simples tanto para se utilizar quanto para se desenvolver novos componentes e funcionalidades.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O projeto tem como objetivo o desenvolvimento de uma aplicação *web* para simulação de circuitos lógico-digitais.

1.2.2 Objetivos Específicos

- Desenvolver uma interface gráfica que permita ao usuário adicionar componentes e criar conexões que representem o circuito a ser simulado;
- Desenvolver um sistema de serialização e desserialização deste circuito, para permitir salvar para um arquivo e carregá-lo novamente;
- Desenvolver um motor de simulação lógica que permita a interação do usuário e exiba em tempo real o andamento da simulação;
- Desenvolver componentes de portas lógicas (and, nand, or, nor, etc.);
- Desenvolver componentes de entrada e saída (botão, switch, LED, display SSD, etc.);
- Desenvolver componentes para conexões (splitter, joiner, etc.);
- Desenvolver componentes plexers (multiplexador e demultiplexador);
- Desenvolver componentes de memória (ROM e RAM);
- Desenvolver testes unitários para todos os componentes do projeto;
- Desenvolver um tutorial guiado para novos usuários;
- Desenvolver um programa de linha de comando que permita a fácil criação de novos componentes;
- Realizar a publicação da aplicação em um endereço de internet público.

2 FUNDAMENTAÇÃO TEÓRICA

Simulação e lógica digital são dois campos de pesquisa científica extremamente frutíferos. Neste capítulo, faremos uma breve revisão de suas respectivas origens e fundamentos teóricos.

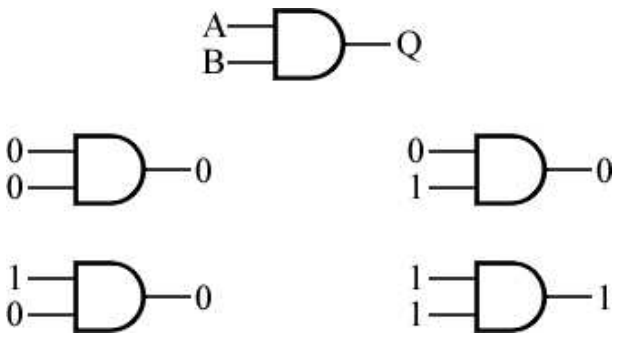
2.1 Lógica digital

O sistema de numeração binária, base para toda a lógica digital, data do início do século XVIII, quando Gottfried Leibniz publicou o artigo *"Explication de l'Arithmétique Binaire"*. Nele, Leibniz expõe a ideia de uma aritmética que usa somente os símbolos numéricos 0 e 1, mostrando exemplos de como se dão operações matemáticas básicas neste sistema (como adição, subtração, multiplicação e divisão) (LEIBNIZ, 1703).

Por meados do século XIX, George Boole desenvolveu o que hoje conhecemos por álgebra Booleana no livro intitulado *"An Investigation of the Laws of Thought"*. Nesta álgebra, declarações lógicas são descritas simbolicamente, de tal forma que seja possível resolve-las como se fossem álgebra comum. Na álgebra Booleana, existem três operações primárias, que são: conjunção ("AND"), disjunção ("OR") e negação ("NOT") (BOOLE, 1854).

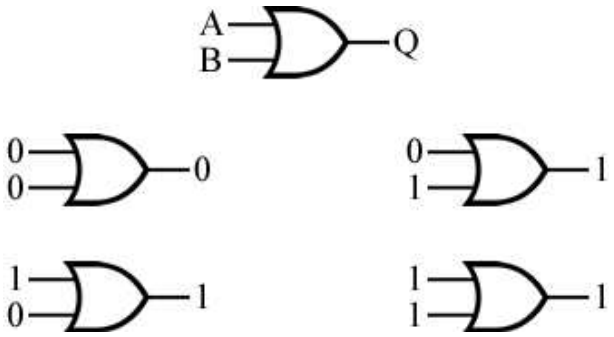
Estas três operações básicas possuem diferentes tipos de representação visual, sendo a principal definida pelo *American National Standards Institutes* (ANSI). As operações lógicas AND, OR e NOT estão representadas nas Figuras 2, 3 e 4, respectivamente, onde A e B representam entradas e Q a saída esperada para esta operação.

Figura 2 – Porta lógica AND



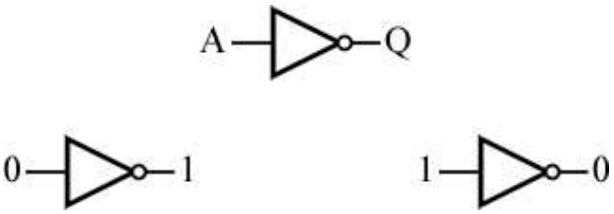
Fonte: Autoria própria

Figura 3 – Porta lógica OR



Fonte: Autoria própria

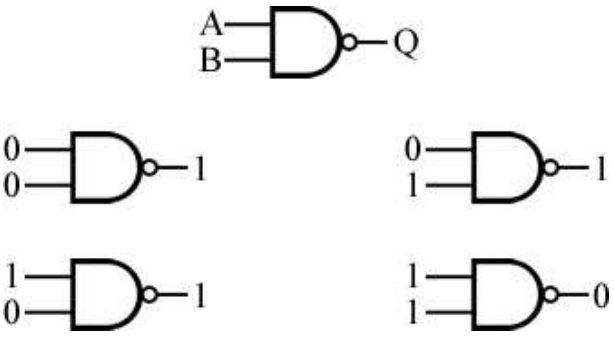
Figura 4 – Porta lógica NOT



Fonte: Autoria própria

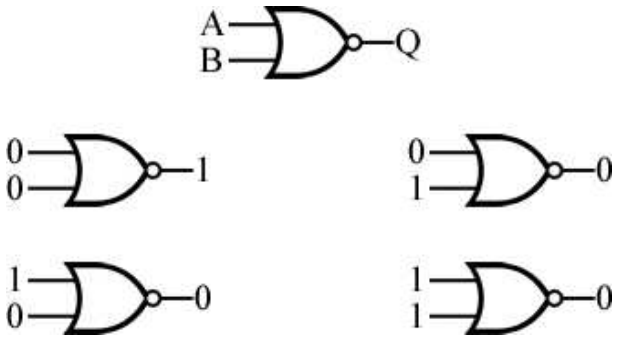
Destas três operações básicas, podem ser construídas as operações NAND (negação de AND), NOR (negação de OR), XOR ("ou exclusivo") e XNOR (negação de XOR), apresentadas nas Figuras 5, 6, 7 e 8, respectivamente.

Figura 5 – Porta lógica NAND



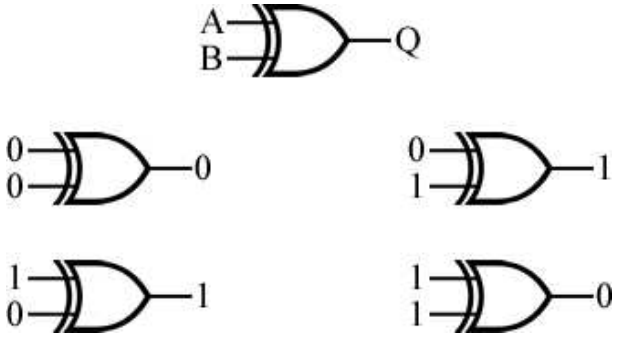
Fonte: Autoria própria

Figura 6 – Porta lógica NOR



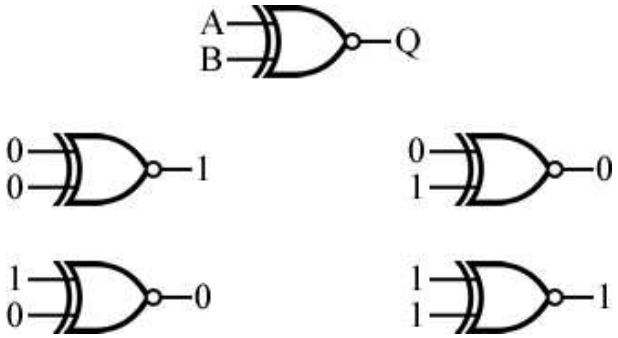
Fonte: Autoria própria

Figura 7 – Porta lógica XOR



Fonte: Autoria própria

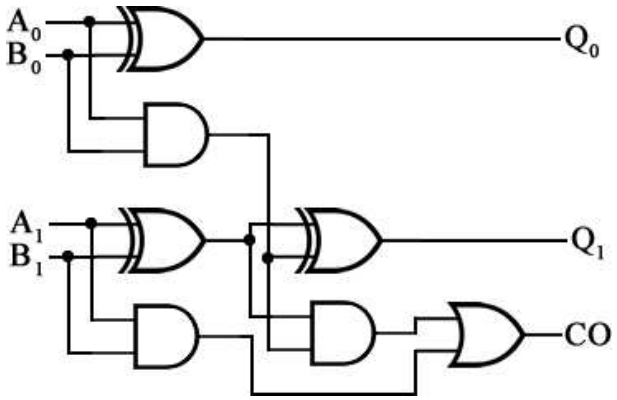
Figura 8 – Porta lógica XNOR



Fonte: Autoria própria

Estas sete operações lógicas formam um conjunto básico sobre a qual a maioria dos componentes lógicos é construído. Por exemplo: com apenas portas lógicas AND, XOR e OR, é possível construir um somador de 2 *bits* como o da Figura 9: um circuito lógico que recebe dois números de dois bits (representados por A_0 e A_1, B_0 e B_1) e retorna o valor correspondente à soma destes valores (representado por Q_0 e Q_1), bem como o valor de *carry out* (representado por CO).

Figura 9 – Somador lógico de 2 *bits*

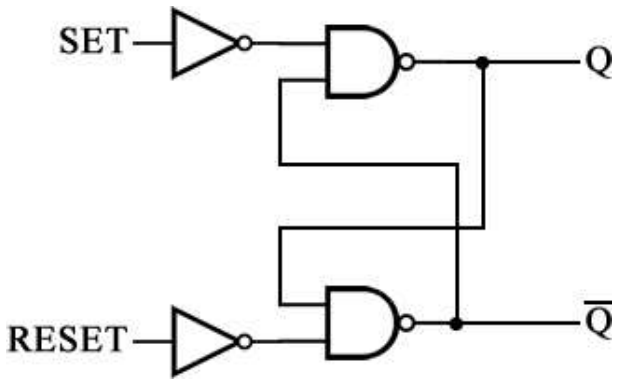


Fonte: Autoria própria

Foi só em 1938 que Claude Shannon provou, na sua tese de mestrado, que circuitos de relés poderiam ser utilizados para resolver problemas de álgebra Booleana (SHANNON, 1937). Utilizar propriedades binárias de circuitos elétricos para executar álgebra Booleana é o conceito fundamental para toda a eletrônica digital moderna.

Circuitos digitais se dividem em dois grandes grupos: combinacionais e sequenciais. Circuitos combinacionais são circuitos cuja saída depende única e exclusivamente de suas entradas (como por exemplo o somador da Figura 9). Já nos circuitos sequenciais, as saídas dependem tanto das entradas quanto de seus estados anteriores (IDOETA, 2008). Por possuírem esta propriedade, células de memória podem ser classificadas como circuitos sequenciais, como por exemplo no *latch* RS da Figura 10:

Figura 10 – Circuito *latch* RS



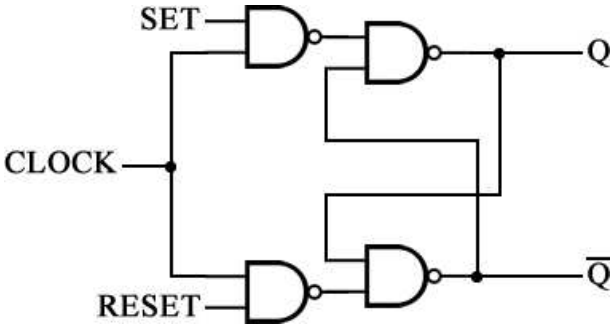
Fonte: Autoria própria

O *latch* RS possui as entradas SET e $RESET$ e tem como saídas Q e $\bar{Q}$ (a negação de Q). Este circuito possui dois elos de realimentação, onde as saídas Q e $\bar{Q}$ são injetadas juntamente com as entradas. Analisar o comportamento deste circuito pode ser um pouco contraintuitivo, porém o funcionamento é bastante simples: quando a entrada SET é acionada, Q assume o valor 1, e quando $RESET$ é acionado, Q assume o valor 0. Caso SET e $RESET$

estejam simultaneamente acionados, tanto Q quanto $\bar{Q}$ assumem o valor 1, o que torna esta combinação de entradas não permitida.

Alguns circuitos sequenciais possuem como entrada um sinal de relógio (ou *clock signal*, em inglês). Este sinal é introduzido em circuitos mais complexos com a finalidade de sincronizar as diferentes partes do circuito. Componentes que recebem um sinal de *clock* são chamados de síncronos. Um exemplo é o *latch* RS síncrono, da Figura 11:

Figura 11 – Circuito *latch* RS síncrono



Fonte: Autoria própria

Neste circuito, enquanto a entrada $CLOCK$ estiver inativa, as saídas permanecerão inalteradas (assumindo os valores do seu último estado válido) e enquanto a entrada $CLOCK$ estiver ativa, o *latch* RS síncrono terá o mesmo comportamento do *latch* RS da Figura 10.

2.2 Modelagem e simulação

Modelagem e simulação é um campo de pesquisa bastante amplo, sendo aplicado em várias áreas de conhecimento e servindo para diversos propósitos, tais como: otimização de performance, testes, educação ou entretenimento (SMITH, 1998). Modelagem consiste em encontrar uma representação física, matemática ou lógica de um sistema, entidade, fenômeno ou processo, e uma simulação é um método que implementa tal modelo no decorrer do tempo (M&SCO, 1998).

Apesar de simulações digitais serem bastante difundidas atualmente, o conceito data de antes da era digital. John von Neumann, considerado um dos pais da computação, foi também pioneiro nesta área, quando, no fim da década de 1940, concebeu a ideia de executar múltiplas repetições de um modelo para derivar comportamentos de um sistema real (SMITH, 1998). Um famoso exemplo de simulação analógica é o computador hidromecânico MONIAC, criado por Bill Phillips em 1949 para simular processos econômicos no Reino Unido (BISSELL, 2007).

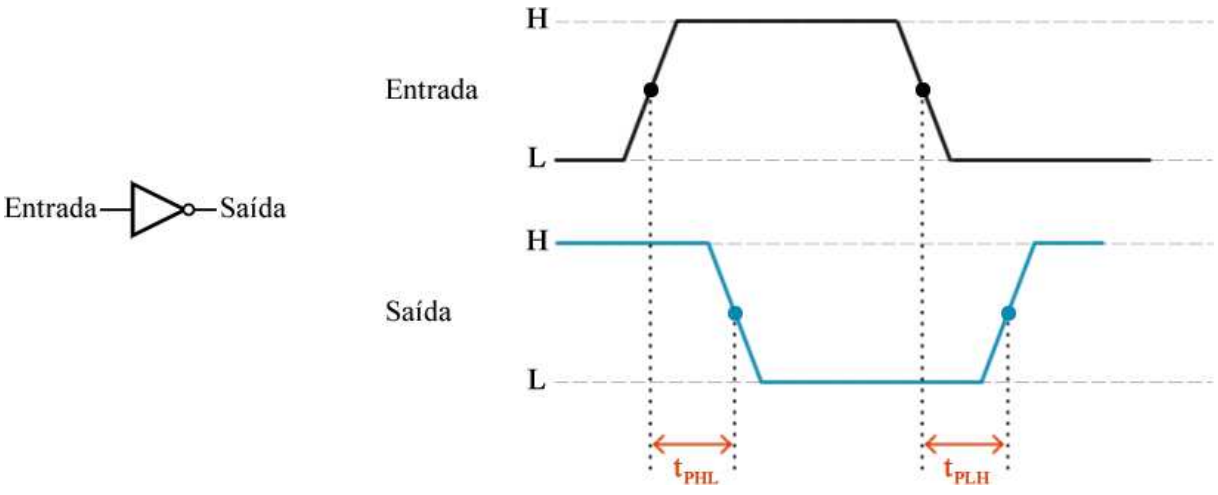
2.2.1 Simulação de circuitos digitais

Uma das principais características de um simulador de circuitos digitais é o algoritmo de escalonamento utilizado, que define uma estratégia para execução da simulação. Existem duas principais estratégias de escalonamento: *event-driven* (ou "orientado a eventos") e *compiled code* (ou "código compilado") (RIEPE et al., 1994).

Numa simulação orientada a eventos, os eventos são dinamicamente gerenciados de modo que somente a parte ativa do circuito seja simulada (MAURER, 1997). Já numa simulação de código compilado, a topologia dos componentes do circuito é estaticamente analisada e reorganizada de tal modo que, antes da execução da simulação em um determinado componente, exista a garantia de que todas suas entradas já tenham sido simuladas (WANG; TROPPER, 2006).

Outra característica importante para um simulador lógico digital se refere a como ele modela atrasos de propagação. Atraso de propagação é um parâmetro inerente a todo componente eletrônico, e é definido como o tempo que um pulso de entrada leva para ser propagado para a correspondente saída (FLOYD, 2007). A Figura 12 ilustra o tempo de atraso de propagação de subida (t_{PHL}) e de descida (t_{PLH}) de uma porta inversora.

Figura 12 – Atraso numa porta inversora



Fonte: Adaptado de Floyd (2007, p. 171)

Existem 3 principais maneiras que um simulador pode modelar atrasos de propagação: atraso zero, atraso unitário ou atraso nominal. Num simulador de atraso zero, o atraso de propagação é totalmente ignorado. Simulações de atraso unitário assumem que todos os elementos de um circuito possuem exatamente o mesmo tempo de atraso. No entanto, a realidade é que diferentes componentes podem possuir diferentes especificações de tempo de atraso, e simuladores que tentam modelar esta especificação são considerados simuladores de atraso nominal (MICZO, 2007).

Simuladores orientados a eventos possuem a vantagem de serem capazes de imple-

mentar modelos com tempos de atraso de propagação zero, unitário ou nominal, enquanto simuladores de código compilado tendem a ser mais eficientes, permitindo a execução de mais instruções num dado período de tempo (FRENCH et al., 1995).

3 DESENVOLVIMENTO

Para o desenvolvimento deste projeto, vários conceitos e tecnologias foram empregados. Este capítulo possui uma revisão sobre estes conceitos e tecnologias, bem como detalhamentos técnicos.

Durante todo o desenvolvimento do Logossim, utilizou-se a ferramenta Jira (JIRA, 2021) para registro e acompanhamento de atividades. Cada atividade cadastrada teve seu tempo de atuação medido e registrado.

3.1 Tecnologias utilizadas

Tecnologias *web* como HTML5, CSS3 e JavaScript formam uma tríade que todo desenvolvedor *web* precisa conhecer. A introdução do Node.JS e do seu gerenciador de pacotes NPM em 2009 revolucionou o ecossistema de desenvolvimento JavaScript, oferecendo a desenvolvedores *web* acesso a uma miríade de pacotes (bibliotecas, componentes e utilitários) prontos para serem usados (HOTA; PRABHU, 2014).

Nesta seção, serão apresentadas as principais bibliotecas e pacotes escolhidos, e também conceitos utilizados para o desenvolvimento do Logossim.

3.1.1 React

A decisão mais importante para o desenvolvimento de uma aplicação *web* moderna diz respeito à implementação da interface gráfica. O React, desenvolvido em 2013 pelo Facebook, é uma das principais bibliotecas para JavaScript neste quesito.

As principais alternativas ao React atualmente são os *frameworks* Angular e Vue. O Quadro 1 apresenta as principais características destas três tecnologias.

Quadro 1 – Comparativo de bibliotecas e *frameworks* alternativos ao React

	React	**Angular**	**Vue**
Linguagens	JSX ou TSX	HTML, CSS e TypeScript	HTML, JavaScript ou TypeScript e CSS
Arquitetura	Módulos, diretivas e componentes	Componentes	Diretivas e componentes
Curva de aprendizado	Média	Alta	Baixa
Autoria	Facebook	Google	Evan You
Publicado em	2013	2010	2014
Downloads semanais	8.8 milhões	2.1 milhões	2 milhões

Fonte: (WOHLGETHAN, 2018)

Comparado com as soluções concorrentes, o React é uma biblioteca pequena (BANKS; PORCELLO, 2017) e eficiente, graças ao uso da abstração conhecida como *virtual DOM* e algoritmos de *smart reconciliation* (MARDAN, 2017).

Apesar de ter sido inicialmente desenvolvido para uso na *web*, o React é agnóstico sobre em qual ambiente será renderizado, sendo possível utilizá-lo para implementar interfaces gráficas em dispositivos móveis, *desktop*, *videogames* e até mesmo em documentos PDF.

Uma das principais razões da escolha do React no uso deste projeto é a grande variedade de componentes disponibilizados pela comunidade dentro de seu ecossistema, tais como o React Diagrams e React Joyride.

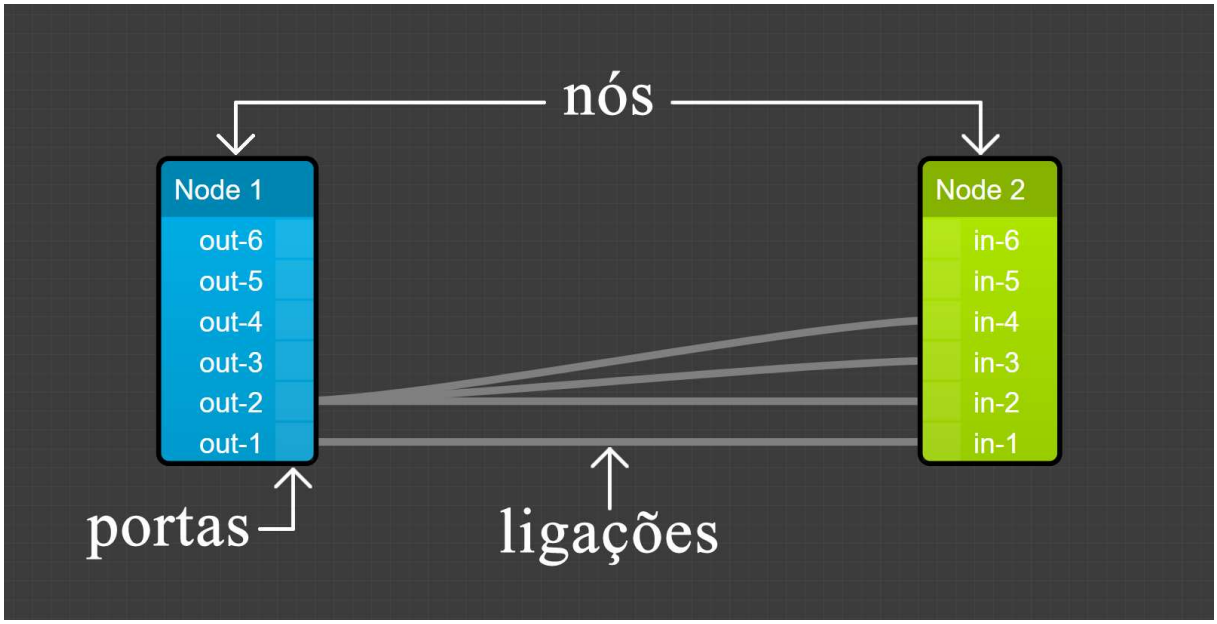
3.1.2 React Diagrams

React Diagrams (DIAGRAMS, 2021b) é uma biblioteca para criação de interfaces com diagramas interativos em React, desenvolvida majoritariamente por Dylan Vorster. Com esta biblioteca, é possível construir uma interface gráfica na qual o usuário pode adicionar, mover e conectar os nós neste diagrama, uma funcionalidade essencial para um simulador como o Logossim.

Um dos principais pontos negativos desta biblioteca é o fato da sua documentação ser praticamente inexistente, possuindo também pouquíssimos exemplos de utilização. Existem alternativas ao React Diagrams, como por exemplo a biblioteca Beautiful React Diagrams (DIAGRAMS, 2021a), que possui uma documentação mais completa, no entanto o React Diagrams permite níveis de customização muito elevados quando comparado às demais alternativas, razão pela qual ele foi utilizado.

No React Diagrams, existem 3 conceitos básicos: nós, portas e ligações (ilustrados na Figura 13). Nós são os elementos primários do diagrama, que podem ser adicionados, movidos e removidos pelo usuário. Cada nó possui zero ou mais portas, e cada porta pode estar conectada a zero ou mais portas.

Figura 13 – Conceitos da biblioteca React Diagrams



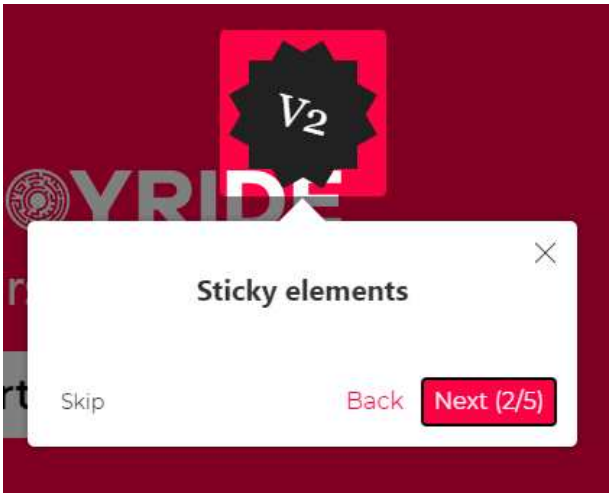
Fonte: Retirado da documentação da biblioteca

Todos aspectos de nós, portas e ligações (tais como aparência, comportamento, etc.) podem ser customizados através de opções e implementações pelo desenvolvedor que está utilizando React Diagrams. As customizações implementadas para o projeto Logossim estão descritas na Subseção 3.3.1.

3.1.3 React Joyride

Algumas aplicações utilizam-se de tutoriais guiados para diminuir a curva de aprendizado de novos usuários em suas aplicações. React Joyride (JOYRIDE, 2021) é uma biblioteca criada por Gil Barbara que facilita implementação deste conceito em React. Com ela, é possível criar tutoriais passo a passo que deem destaque a elementos específicos na tela, como exibido na Figura 14.

Figura 14 – Exemplo de funcionamento da biblioteca React Joyride



Fonte: Retirado da documentação da biblioteca (JOYRIDE, 2021)

O React Joyride foi utilizado no projeto para a implementação do tutorial introdutório guiado, descrito na Subseção 3.3.3

3.1.4 Web Workers

JavaScript é uma linguagem *single-threaded*, o que significa que ela não dá suporte a execução paralela de código. Em navegadores *web*, o código JavaScript de uma aplicação é executado na *thread* principal, também responsável por pintar a interface na tela e responder a eventos do usuário na aplicação. Por esta razão, funções JavaScript em aplicações *web* não devem demorar muito para serem executadas, caso contrário o navegador pode começar a atrasar a renderização e parar de responder a ações do usuário, efetivamente causando um travamento da aplicação.

Em alguns cenários, no entanto, aplicações *web* podem necessitar executar funções que demandam grande quantidade de tempo para serem concluídas. Para tornar isto possível, foi adicionado à especificação da linguagem o conceito de *web workers*.

Web workers possibilitam a execução de código JavaScript fora da *thread* principal, mas com acesso limitado a recursos da linguagem, de modo a evitar problemas de concorrência. Um código JavaScript que executa em um contexto de *web worker* não pode, por exemplo, acessar diretamente o objeto global `document`, que serve como ponto de entrada para o conteúdo da página atual (GREEN, 2012).

Também não é possível, dentro do contexto de um *web worker*, acessar variáveis ou executar funções que sejam do escopo da *thread* principal. Após a criação de um *web worker* pela *thread* principal, toda a comunicação entre ambos só pode ser realizada por troca de mensagens.

O Logossim utiliza-se de um *web worker* para executar o algoritmo de simulação, para evitar que a aplicação trave por completo caso o circuito sendo simulado demande uma alta

quantidade de processamento. Uma descrição mais detalhada da aplicação deste conceito é apresentada na Seção 3.6.

3.1.5 Jest

A escrita de testes automatizados é fundamental para garantir a qualidade de qualquer projeto de *software*. A existência de testes automatizados num projeto diminui a probabilidade de erros previamente corrigidos serem reinseridos no sistema, além de possuir uma escalabilidade muito maior quando comparado a testes manuais. Testes automatizados também podem servir como uma espécie de contrato, que define comportamento esperado de um sistema em cada cenário, de modo a facilitar e encorajar refatorações e melhorias de código (SALEH, 2013).

Jest (JEST, 2021) é uma das principais ferramentas para elaboração de testes automatizados em JavaScript. Criada pelo Facebook em 2014, ela fornece uma série de funções que podem ser utilizadas para descrever cenários de teste e definir expectativas sobre tais cenários.

Com Jest, também é possível gerar um relatório de cobertura de testes. Um relatório de cobertura de testes pode ser utilizado para analisar partes do código que não estão cobertas por testes ou para identificar código morto. Este relatório possui 4 tipos de cobertura distintos: cobertura de declarações (declaração/atribuição de variável, chamada/retorno de método, etc.), de ramificações (`if/else/else if` ou `for/while`), de funções e de linhas.

Esta ferramenta foi utilizada neste projeto para a escrita de testes unitários de seus componentes, descrito na Subseção 3.4.4.

3.1.6 Plop

Plop (PLOP, 2021) é uma biblioteca para JavaScript que permite o desenvolvimento de geradores de código baseados em modelos pré-definidos. A configuração desta biblioteca demanda uma implementação para definir comandos que determinarão como a geração de código deve ser parametrizada, bem como os modelos que serão utilizados para esta geração.

O criador de componentes descrito na Seção 3.8 é um exemplo do que se pode atingir com esta biblioteca: uma interface de linha de comando, destinada a ser usada por desenvolvedores do projeto, que gera código no repositório baseado na resposta dos comandos do usuário.

Ferramentas de geração de código ajudam desenvolvedores a economizar tempo despendido executando tarefas repetidas e tediosas, diminuem a curva de aprendizado para que novos desenvolvedores se integrem ao projeto e também facilitam a manutenção de um padrão de código, estrutura de pastas e arquivos.

3.1.7 Travis CI

Integração contínua (cuja sigla CI vem do inglês *Continuous Integration*) é um conjunto de práticas no desenvolvimento de *software* introduzidas por Martin Fowler em 2000 visando

acelerar o processo de desenvolvimento, mas ao mesmo tempo mantendo a qualidade do código construído (ZHAO et al., 2017). As principais práticas de integração contínua definidas por Fowler (2000) são:

- **automatizar a construção:** todos os passos para a construção de uma versão utilizável de um *software* devem ser completamente automatizados. Em outras palavras, deve ser possível construir o projeto com um só comando numa máquina que contenha o código-fonte, mesmo que ela não possua instalada todas as dependências e configurações de ambiente necessárias;
- **incluir testes automatizados na construção:** o processo de construção do *software* não deve ser composto somente pelos passos que o tornam executável, mas também pela execução de testes que garantam que a versão do *software* sendo construída não contenha regressões;
- **toda alteração deve ser construída numa máquina de integração:** para evitar que alterações aplicadas ao código-fonte do projeto introduzam problemas no *software*, toda alteração que é enviada ao repositório de um projeto deve ser construída, preferencialmente numa máquina de integração remota, para garantir que a compilação funciona e todos testes automatizados finalizam com sucesso;
- **automatizar a publicação:** o processo de publicação de uma versão do *software* deve ser totalmente automatizado.

Travis CI é uma plataforma de integração contínua que disponibiliza gratuitamente máquinas virtuais de integração para projetos de código aberto. Este projeto usou Travis CI para execução automática de testes e publicação. Maiores detalhes sobre a configuração deste serviço se encontram na Seção 3.9.

3.1.8 Repositórios monolíticos

Tradicionalmente, diferentes projetos de software, mesmo que relacionados entre si, costumam estar em repositórios de código distintos. Um repositório monolítico (também conhecido como *monorepo*), é um repositório que contém vários projetos ou pacotes de software, sendo eles relacionados entre si ou não. Este padrão estrutural é utilizado em grandes empresas como Google e Facebook, e também por uma série de projetos populares de código aberto (BRITO; TERRA; VALENTE, 2018).

Segundo Brito, Terra e Valente (2018), as principais vantagens de se utilizar esta estruturação ao invés da abordagem tradicional são:

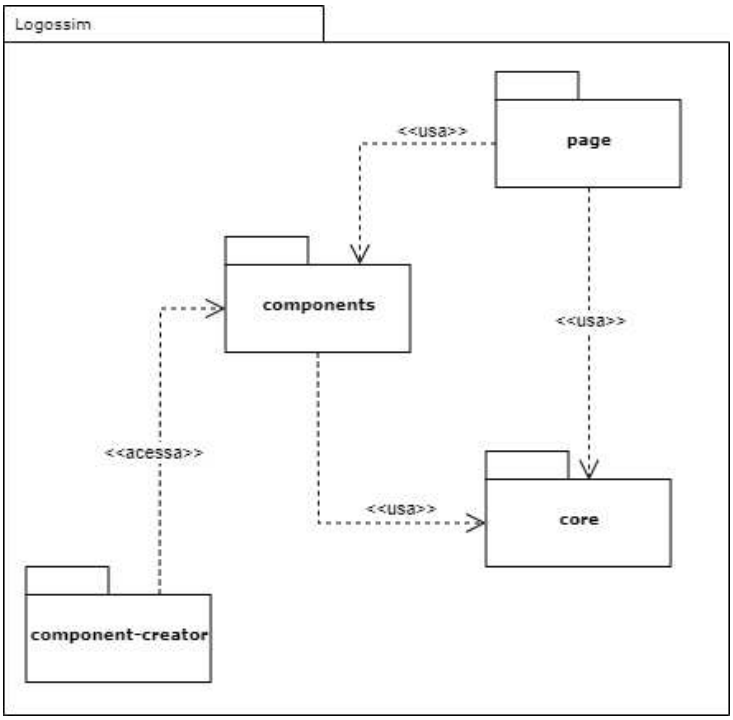
- **padronização:** ferramentas de formatação de checagem estática de código podem ser facilmente compartilhadas entre os múltiplos pacotes do repositório;
- **compartilhamento de dependências:** pacotes que vivem sobre o mesmo repositório monolítico podem ser configurados de maneira a compartilhar dependências externas, garantindo que todos os pacotes utilizam exatamente a mesma versão destas dependências;

- **facilidade para refatoração:** um repositório monolítico tem uma maior tendência a possuir códigos modulares, o que torna o processo de refatoração mais simples.

3.2 Arquitetura do projeto

Este projeto foi organizado numa estrutura de repositório monolítico, contendo 4 pacotes com finalidades específicas e que dependem entre si, conforme ilustrado na Figura 15.

Figura 15 – Visão geral dos pacotes do repositório Logossim



Fonte: Autoria própria

Para implementar esta estrutura, utilizou-se a ferramenta Yarn Workspaces. Os requisitos mínimos para executar o Logossim em modo de desenvolvimento, portanto, são a instalação dos programas Node.JS (NODE.JS, 2021) e Yarn (YARN, 2021).

Nesta seção, será detalhada a finalidade de cada um destes pacotes e sua contribuição para a construção do projeto como um todo.

3.2.1 Pacote core

O pacote `@logossim/core`[1] contém implementações essenciais para o projeto, como por exemplo:

- customizações feitas na biblioteca React Diagrams, detalhadas na seção Subseção 3.3.1;
- o componente React `Port`, utilizado para representar um pino nos componentes digitais;

[1] <https://github.com/renato-bohler/logossim/tree/master/packages/@logossim/core>

- a classe `BaseModel`, que define uma interface mínima de métodos e propriedades para a criação dos componentes digitais;
- a classe `Component`, que serve para declarar as características de um componente digital (como seu nome, descrição e configurações disponíveis);
- a implementação da execução da simulação em si.

3.2.2 Pacote components

O pacote `@logossim/components`[2] possui a implementação de todos os componentes disponíveis para uso no Logossim.

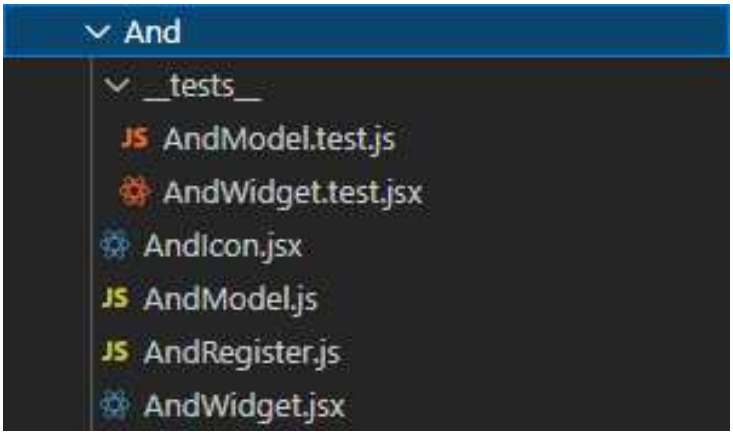
Cada componente neste pacote tem sua própria pasta, e, desconsiderando código de testes unitários, é composto por somente 4 arquivos:

- `Model`: classe que estende `BaseModel` (importado de `@logossim/core`) e define o comportamento do componente durante a simulação. Também é responsável por inicializar os pinos do componente (atribuindo um nome e os configurando);
- `Widget`: componente React que é responsável pela apresentação do componente no circuito. Ou seja, este código determina o conteúdo (HTML) e estilos (CSS) do componente, bem como o posicionamento de seus pinos de entrada e saída (utilizando `Port` de `@logossim/core`). Recebe por parâmetro a instância de `Model` correspondente, podendo acessar suas propriedades e chamar seus métodos;
- `Icon`: determina um ícone para o componente, que é exibido nos menus de adição de componente;
- `Register`: responsável por definir o nome, descrição e categoria do componente. Também é responsável por definir um conjunto de configurações que este componente pode receber.

Por exemplo, a Figura 16 apresenta a estrutura de pastas e arquivos que contém o componente que representa a porta lógica "E" (*And*).

[2] <https://github.com/renato-bohler/logossim/tree/master/packages/@logossim/components>

Figura 16 – Estrutura de pastas e arquivos do componente *And*



Fonte: Autoria própria

Conceitos e detalhes de implementação de componentes no projeto Logossim são discutidos em maior detalhe na Seção 3.4.

3.2.3 Pacote page

O pacote `@logossim/page`[3] é a aplicação *web* em si. Esta é uma aplicação React, inicializada com o utilitário de linha de comando Create React App (CREATE-REACT-APP, 2021), que implementa a interface gráfica apresentada na seção Seção 3.3.

Este pacote depende de `@logossim/core`, pois utiliza funcionalidades do React Diagrams para implementar a área de desenho (Subseção 3.3.1) e importa todos os componentes disponíveis do pacote `@logossim/components` para serem exibidos nos menus de adição de componente ao usuário (Subseção 3.3.2).

3.2.4 Pacote component-creator

O pacote `@logossim/component-creator`[4] contém o código-fonte de uma ferramenta utilitária de linha de comando que serve para gerar componentes básicos no pacote `@logossim/components`. A Seção 3.8 contém mais detalhes do funcionamento desta ferramenta.

3.3 Interface de usuário

Para a interface gráfica do Logossim, foi utilizada uma abordagem minimalista, visando evitar uma sobrecarga cognitiva ao apresentar ao usuário somente o mínimo necessário para que ele possa começar a utilizar o *software*. Toda a interface da aplicação é constituída por uma

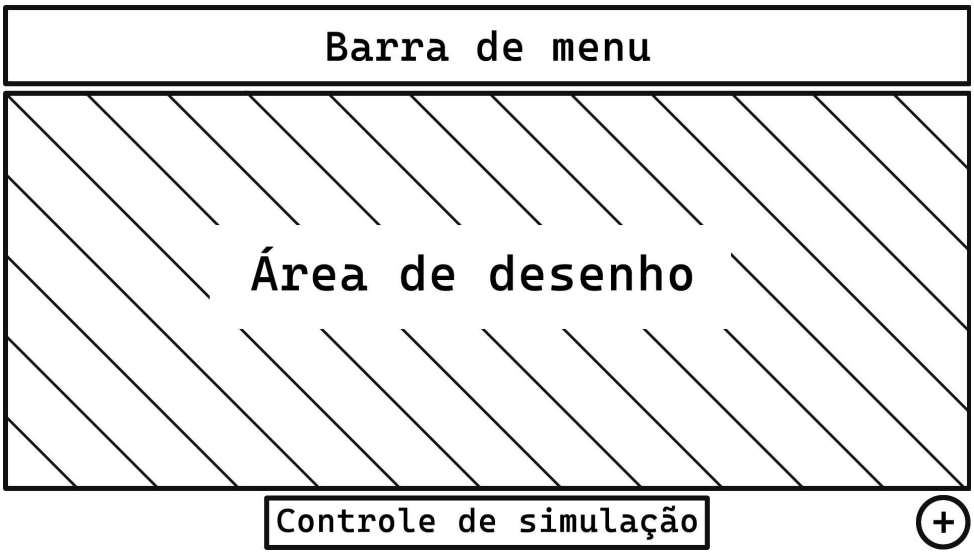
[3] <https://github.com/renato-bohler/logossim/tree/master/packages/@logossim/page>
[4] <https://github.com/renato-bohler/logossim/tree/master/packages/@logossim/component-creator>

interface gráfica principal, e todo o resto do conteúdo (como menus de adição e configuração de componentes e menu de ajuda) são exibidos em caixas de diálogo.

Antes de iniciar-se o desenvolvimento, foi desenhado um esboço da interface principal, ilustrado na Figura 17. Esta interface é composta por quatro elementos principais:

- **barra de menu:** contém botões para salvar e carregar o circuito atual, bem como botões de ajuda ao usuário;
- **área de desenho:** equivalente à área de desenho do Logisim, é onde o usuário pode construir seu circuito e visualizar o andamento da simulação;
- **controle de simulação:** contém botões para iniciar, pausar ou parar o andamento da simulação;
- **botão flutuante de adição:** representado pelo símbolo + no canto inferior direito da tela, este botão é responsável por abrir a caixa de diálogo para adição de componentes à área de desenho quando acionado.

Figura 17 – Esboço da interface principal



Fonte: Autoria própria

Neste esboço, as caixas de diálogo da aplicação seriam simplesmente janelas que flutuam centralizadas sobre a interface principal, sempre possuindo um botão para o fechamento da caixa, como exibe a Figura 18.

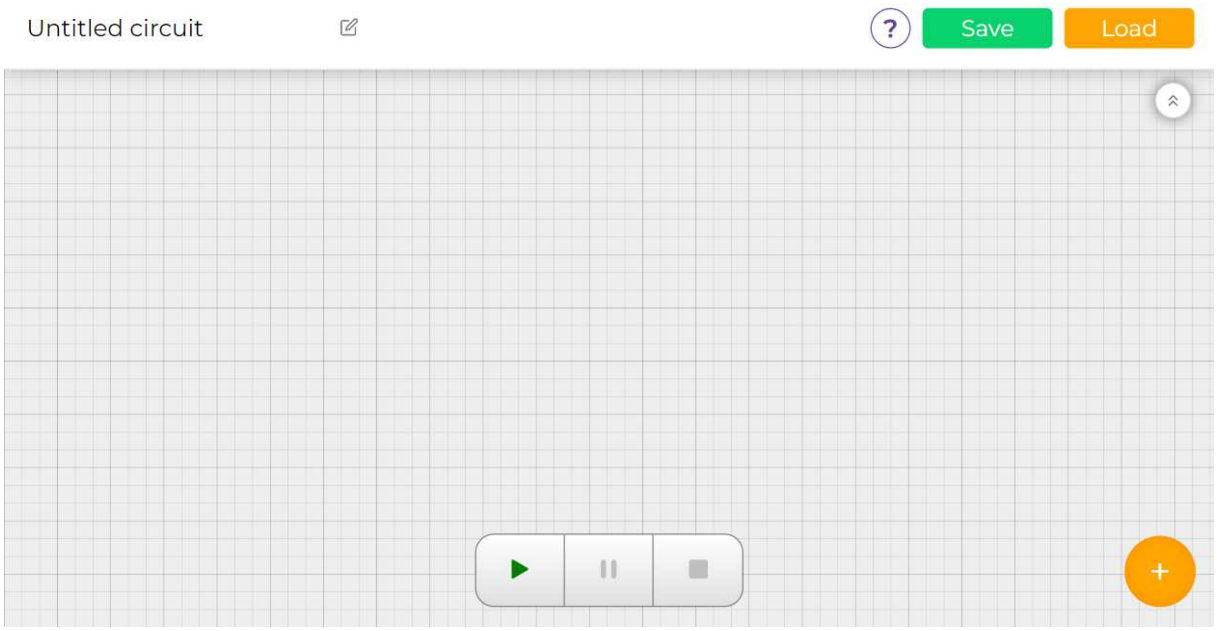
Figura 18 – Esboço de uma caixa de diálogo na aplicação

Fonte: Autoria própria

O desenvolvimento da interface gráfica foi realizado utilizando React. As seções mais simples (barra de menu, controle de simulação e botão de adição de componentes) foram as primeiras a serem desenvolvidas. A área de desenho foi inicialmente desenvolvida num repositório em separado, por questão de organização, e foi mais tarde integrada ao resto do projeto. Durante o desenvolvimento da interface, duas novas funcionalidades foram adicionadas ao escopo: um tutorial introdutório guiado e um menu de ajuda.

Para ter um alcance maior, todos os elementos textuais da interface gráfica foram escritos em inglês. A Figura 19 exibe a interface principal desenvolvida.

Figura 19 – Interface principal



Fonte: Autoria própria

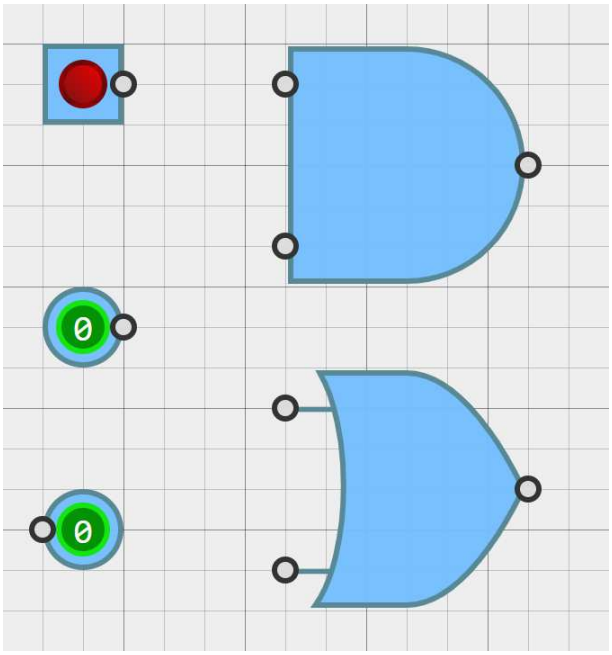
Nas próximas subseções será apresentado em maior detalhamento o desenvolvimento da área de desenho, dos menus de adição e configuração de componentes, do tutorial introdutório e dos atalhos de teclado.

3.3.1 Área de desenho

A área de desenho é o principal elemento visual do Logossim. É nela que o usuário desenha seu circuito. É também nela que o usuário interage com os componentes e visualiza a execução da simulação.

Esta área é basicamente uma grade infinita, implementada usando React Diagrams, onde é possível adicionar componentes. Todo componente no Logossim possui uma representação visual, e pode conter zero ou mais portas de entrada ou saída, conforme ilustra a Figura 20.

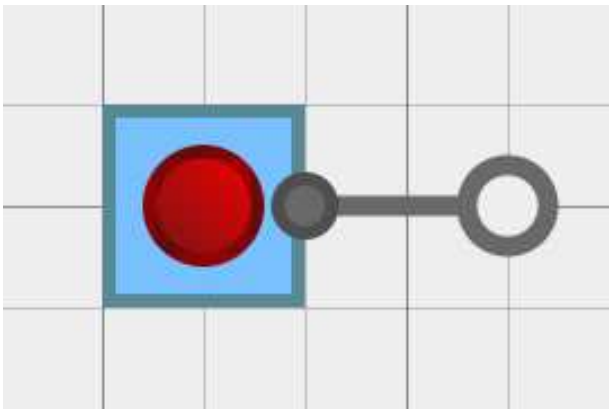
Figura 20 – Componentes na área de desenho



Fonte: Autoria própria

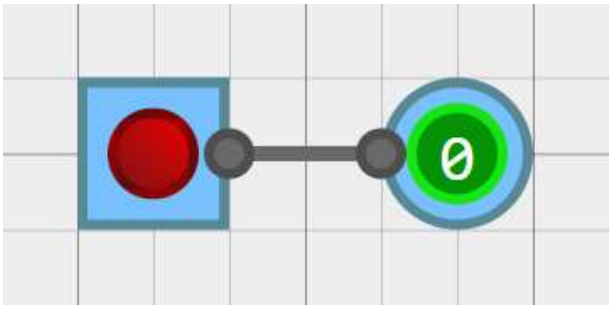
Na área de desenho, o usuário também pode criar fios a partir de qualquer porta do componente, clicando na porta e arrastando o ponteiro. A Figura 21 exibe um componente com um fio que não está conectado a outra porta. Quando o usuário solta o botão com o ponteiro posicionado sobre outra porta, é realizada a conexão entre estas duas portas, conforme exibe a Figura 22.

Figura 21 – Fio desconectado na área de desenho



Fonte: Autoria própria

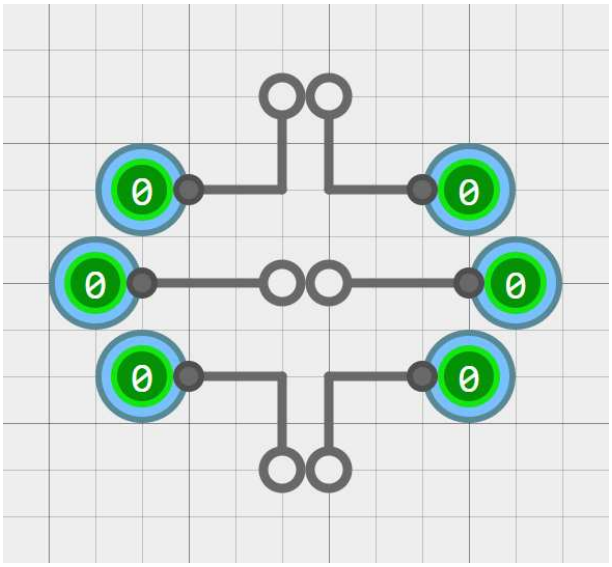
Figura 22 – Componentes conectados na área de desenho



Fonte: Autoria própria

Para que o Logossim tivesse uma experiência de uso semelhante à do Logisim, foram implementadas uma série de customizações na biblioteca React Diagrams para melhorar a criação de conexões. Uma destas customizações se refere ao formato das conexões: conforme o usuário arrasta o ponteiro, o sistema cria conexões em ângulo reto (Figura 23), ao invés de simplesmente uma reta ou uma curva suavizada.

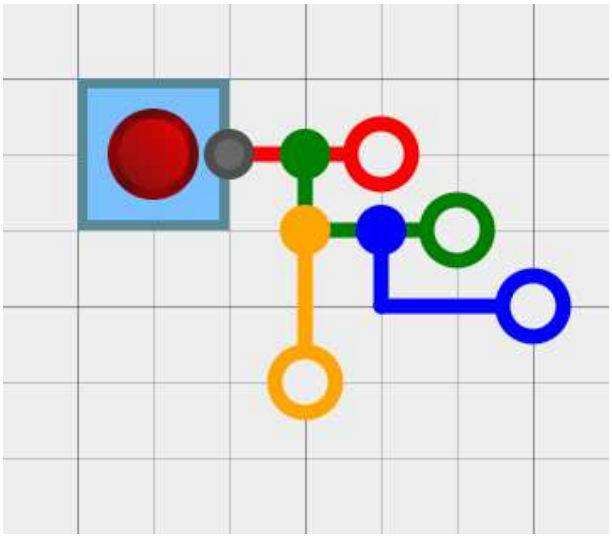
Figura 23 – Conexões em ângulo reto criadas na área de desenho



Fonte: Autoria própria

Outra customização implementada na criação de fios foi a habilidade de bifurcar fios. Esta é uma funcionalidade originalmente inexistente no React Diagrams, que dá ao usuário a habilidade de clicar em cima de um fio pré-existente e arrastá-lo para criar outro fio conectado ao original. Isto cria uma hierarquia de conexões, onde cada fio pode conter zero ou mais fios descendentes. Esta situação é ilustrada na Figura 24, onde o fio vermelho foi criado da porta de saída do componente, o verde foi criado a partir do vermelho, e os fios azul e laranja foram ambos criados a partir do verde.

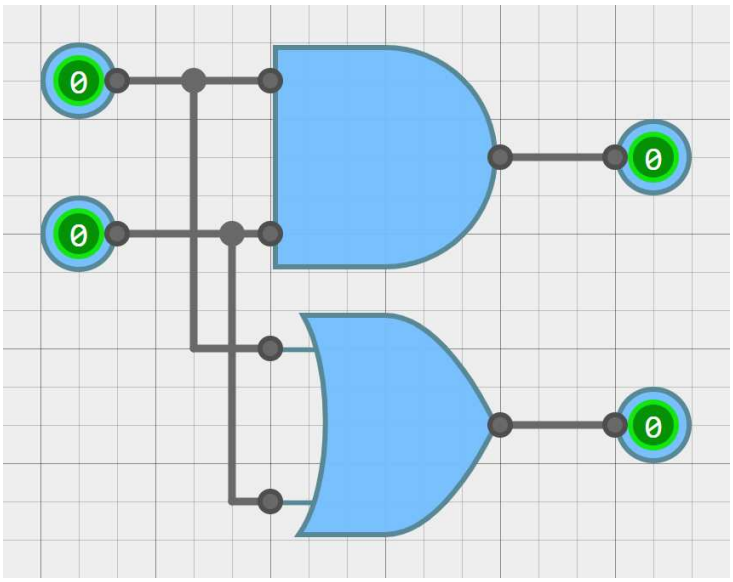
Figura 24 – Ilustração de hierarquia de fios na área de desenho



Fonte: Autoria própria

A coloração dos fios e das portas é cinza enquanto o usuário está em modo de edição, como na Figura 25.

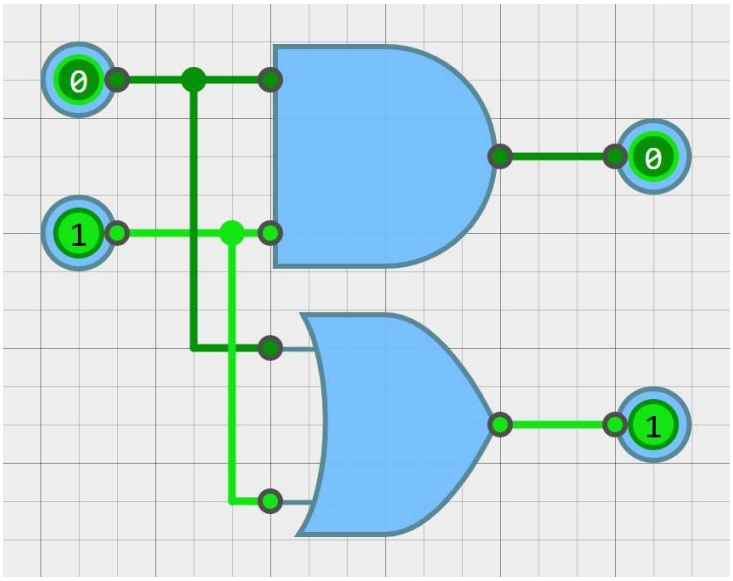
Figura 25 – Apresentação de um circuito em modo de edição na área de desenho



Fonte: Autoria própria

Quando o usuário inicia a simulação, a coloração dos fios passa a representar o valor aferido pela simulação, conforme a Figura 26. A Seção 3.4 possui mais detalhes sobre quais são os possíveis valores que podem ser assumidos por fios.

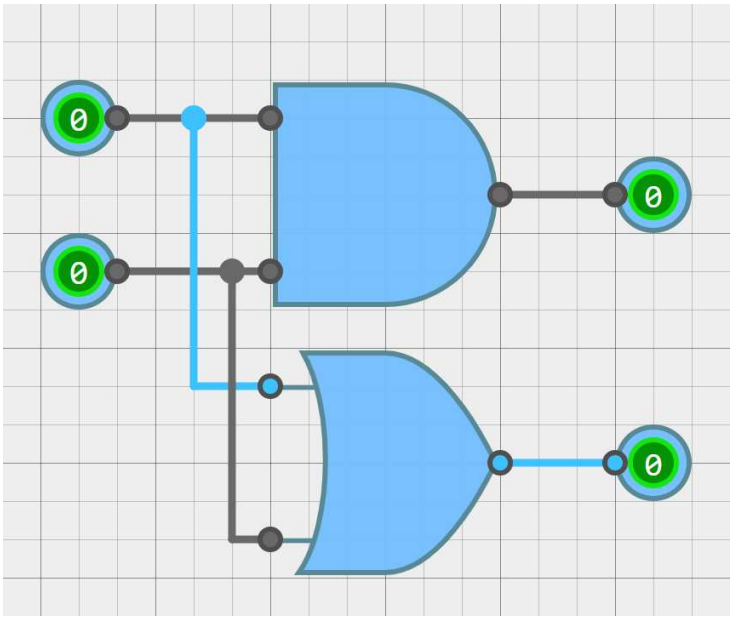
Figura 26 – Apresentação de um circuito em modo de simulação na área de desenho



Fonte: Autoria própria

Para excluir um fio, o usuário pode selecionar o fio a ser removido ao clicar com o ponteiro posicionado ele e pressionar a tecla `DELETE` do teclado. O usuário pode selecionar múltiplos fios para remoção segurando a tecla `SHIFT`. Fios selecionados ficam em destaque, com a coloração alterada para azul, conforme a Figura 27.

Figura 27 – Fios selecionados pelo usuário em destaque

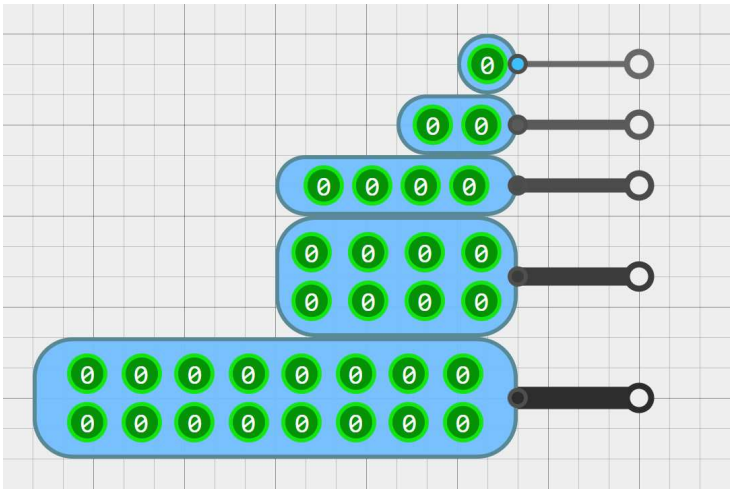


Fonte: Autoria própria

Outra característica que foi adicionada aos fios é a adição da propriedade de largura de *bits*. Esta propriedade permite que vários *bits* de informação sejam transmitidos por uma

só conexão. Cada fio pode assumir uma largura de 1, 2, 4, 8 ou até 16 *bits*. Conexões com maior largura de *bits* possuem uma largura maior na sua representação visual, como ilustrado na Figura 28.

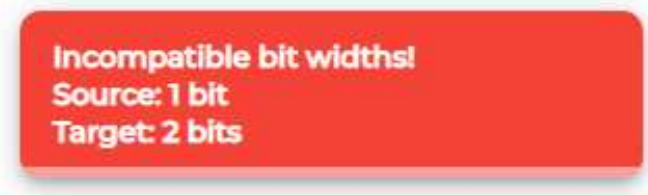
Figura 28 – Representação visual de diferentes larguras de *bits*



Fonte: Autoria própria

Conexões de um fio com uma dada largura de bits só podem ser realizadas entre portas ou outros fios com a mesma largura de bits. Portanto, caso o usuário tente realizar uma conexão entre o primeiro e o segundo fios da Figura 28, a mensagem de erro da Figura 29 será exibida ao usuário.

Figura 29 – Mensagem de erro para larguras de *bits* incompatíveis



Fonte: Autoria própria

A funcionalidade de copiar e colar, presente na maioria dos programas de computador, também precisou ser implementada à área de desenho, pois não é fornecida pelo React Diagrams por padrão. No Logossim, o usuário pode copiar e colar componentes selecionando-os e pressionando as teclas `CTRL + C` e `CTRL + V`. O usuário pode também duplicar os componentes selecionados pressinando as teclas `CTRL + D`.

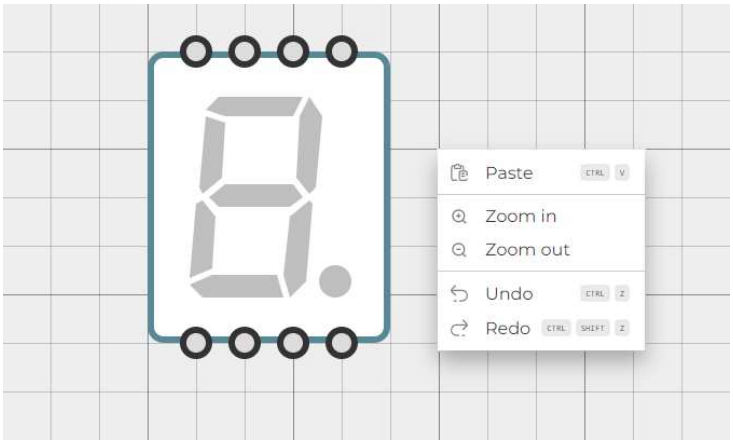
Outra funcionalidade importante que precisou ser implementada foi a de desfazer e refazer. Para desenvolvê-la, foi utilizado o padrão de desenvolvimento de software conhecido por *Command Pattern*. Este padrão encapsula uma ação num objeto (objeto de comando), que

contém as informações necessárias para refazer e desfazer tal ação (SARCAR, 2018). Quando o usuário adiciona um componente no Logossim, por exemplo, um comando de adição de componente é adicionado a uma pilha, contendo todas as informações necessárias (tipo de componente, posição em que foi adicionado, configurações, etc.) para refazer e desfazer esta ação. Isto ocorre para todas as ações realizadas pelo usuário que alteram a área de desenho, tais como:

- adição de componente à área de desenho;
- edição das configurações de um componente na área de desenho;
- alteração da posição de um componente na área de desenho;
- remoção de um componente da área de desenho;
- criação de um fio na área de desenho;
- remoção de um fio da área de desenho.

Para ajudar o usuário a encontrar todas as ações disponíveis, foi desenvolvido um menu de contexto específico para a área de desenho. Ao clicar com o botão direito na grade da área de desenho, o usuário verá o menu da Figura 30, contendo opções para colar, aumentar e diminuir o zoom e desfazer e refazer ações.

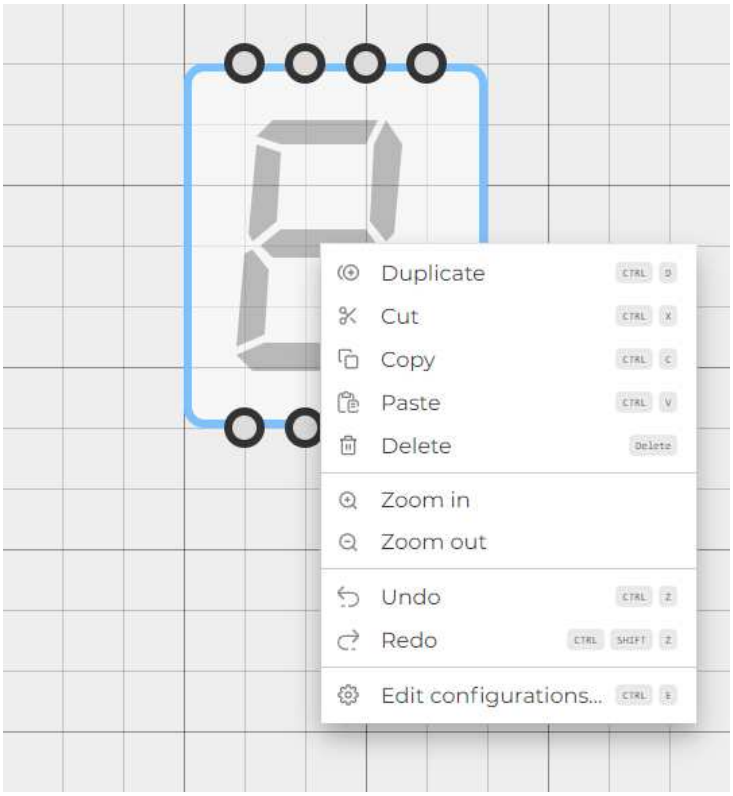
Figura 30 – Opções do menu de contexto ao clicar na área de desenho



Fonte: Autoria própria

Caso o usuário clique com o botão direito com o ponteiro sobre um componente, as opções de duplicar, recortar, copiar, deletar e editar configurações do componente também estarão visíveis, como na Figura 31.

Figura 31 – Opções do menu de contexto ao clicar em um componente

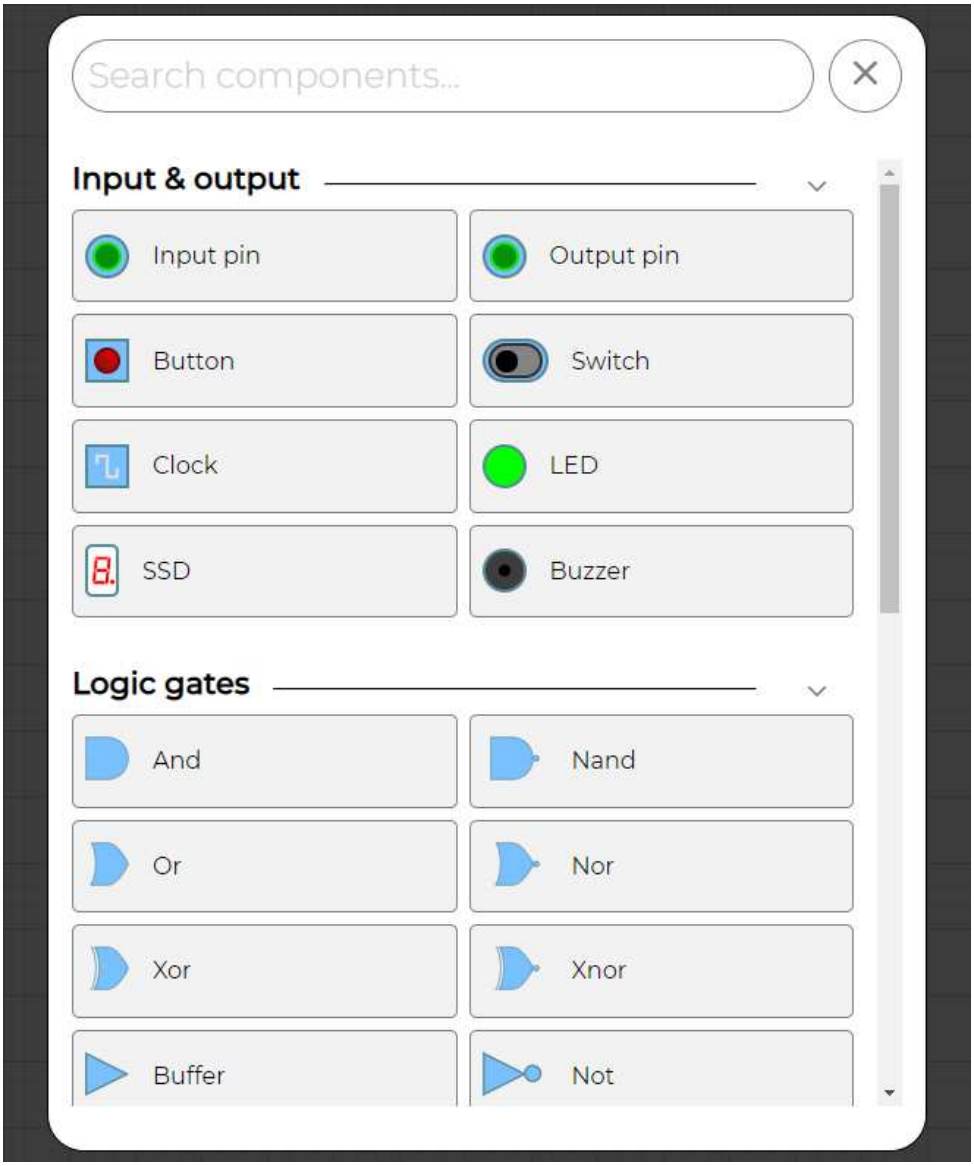


Fonte: Autoria própria

3.3.2 Menus de adição de componentes

Os menus de adição e configuração de componentes foram implementados como uma caixa de diálogo. O menu de adição de componentes (Figura 32) exibe uma lista de todos componentes disponíveis na aplicação, agrupados por categoria (entrada e saída, portas lógicas, cabeamento, multiplexadores, memória e miscelânea). O usuário pode realizar uma pesquisa textual.

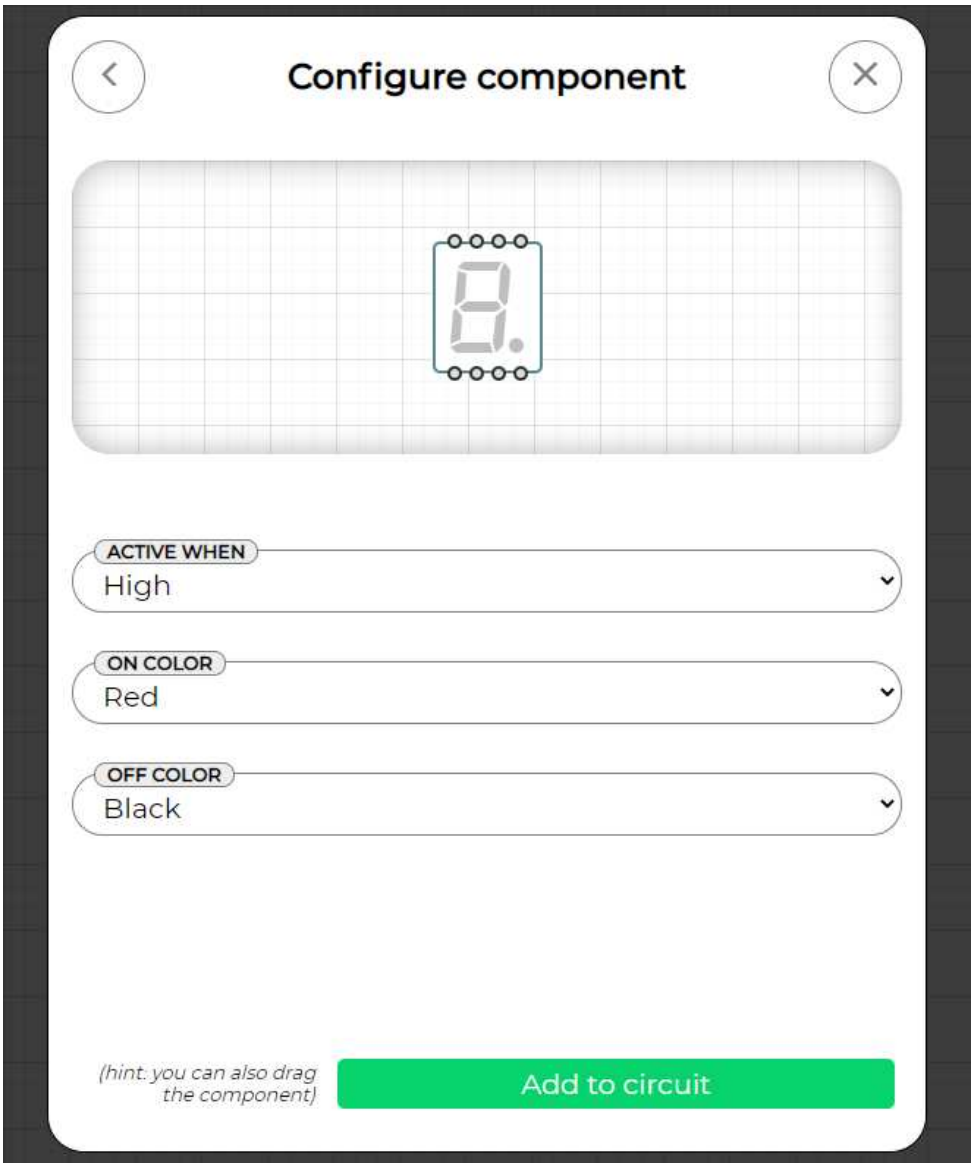
Figura 32 – Caixa de diálogo para adição de componentes



Fonte: Autoria própria

Ao selecionar um dos componentes no menu de adição, é exibido ao usuário o menu de configuração de componentes. A Figura 33 ilustra a tela exibida caso o usuário clique no componente SSD, que possui três configurações possíveis. Cada componente possui seu próprio conjunto de configurações disponíveis. O Apêndice A contém todas as configurações disponíveis para cada componente do projeto.

Figura 33 – Caixa de diálogo para configuração do componente *display* de sete segmentos (SSD)



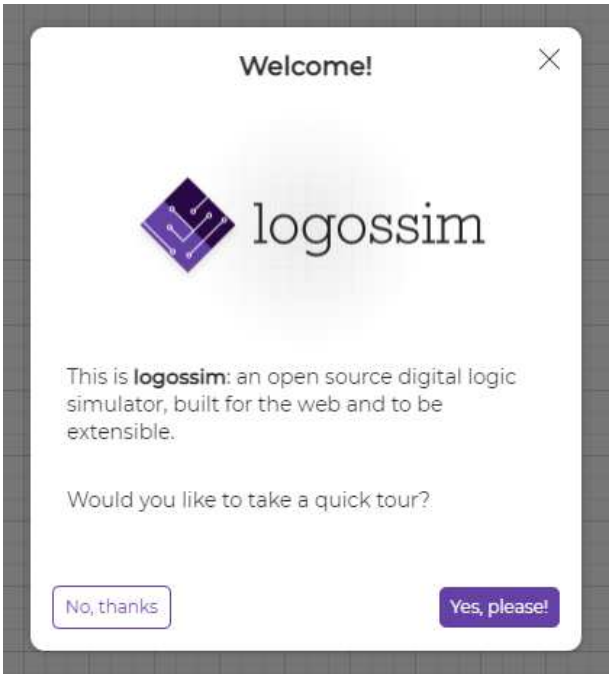
Fonte: Autoria própria

3.3.3 Tutorial introdutório guiado

Foi implementado, utilizando React Joyride, um tutorial opcional para o Logossim (Figura 34). Este tutorial é exibido a primeira vez que o usuário acessa a aplicação. A detecção de primeiro acesso foi implementada utilizando a API de armazenamento local (ou *local storage*), disponível em navegadores *web* (MOZILLA, 2021b).

O objetivo do tutorial guiado é facilitar a habituação do usuário com a interface gráfica do sistema. Um estudo conduzido por Stål (2020) concluiu que usuários que decidem participar de tutoriais como este tem um percentual maior de sucesso na execução de uma tarefa, e realiza a tarefa consideravelmente mais rápido que usuários que decidem não participar.

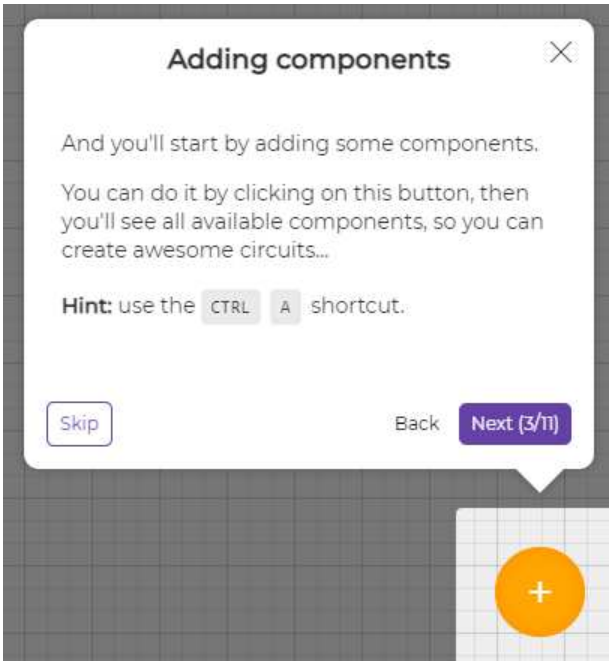
Figura 34 – Caixa de diálogo de informações adicionais do projeto



Fonte: Autoria própria

Este tutorial consiste de 11 passos e tem o objetivo de apresentar os elementos importantes da interface de usuário, como por exemplo o segundo passo (Figura 35), que destaca o botão de adição de componentes e descreve sua funcionalidade.

Figura 35 – Caixa de diálogo de informações adicionais do projeto

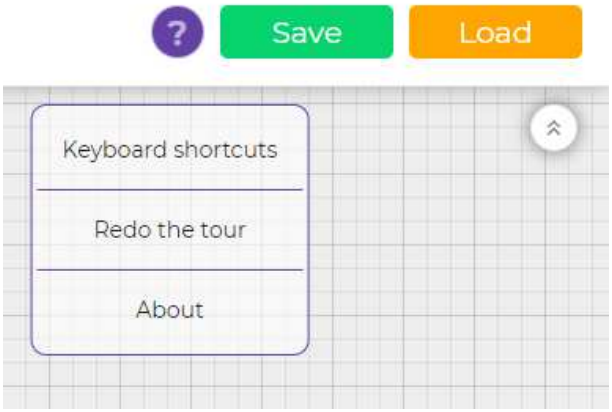


Fonte: Autoria própria

3.3.4 Menu de ajuda

O Logossim conta com um menu de ajuda, que pode ser acessado pelo botão com o símbolo "ponto de interrogação", localizado à direita da barra de menu, conforme mostra a Figura 36. Ao clicar neste botão, três opções são apresentadas ao usuário: "atalhos de teclado", "refaça o tutorial" e "sobre".

Figura 36 – Opções do menu de ajuda



Fonte: Autoria própria

Durante o desenvolvimento do projeto, notou-se a necessidade da implementação de atalhos de teclado para facilitar o acesso a funcionalidades e ferramentas. Clicando na primeira opção do menu de ajuda, uma caixa de diálogo que contém todos os atalhos de teclado disponíveis na aplicação será exibida ao usuário. O Quadro 2 apresenta estes atalhos.

Quadro 2 – Atalhos de teclado implementados ao Logossim

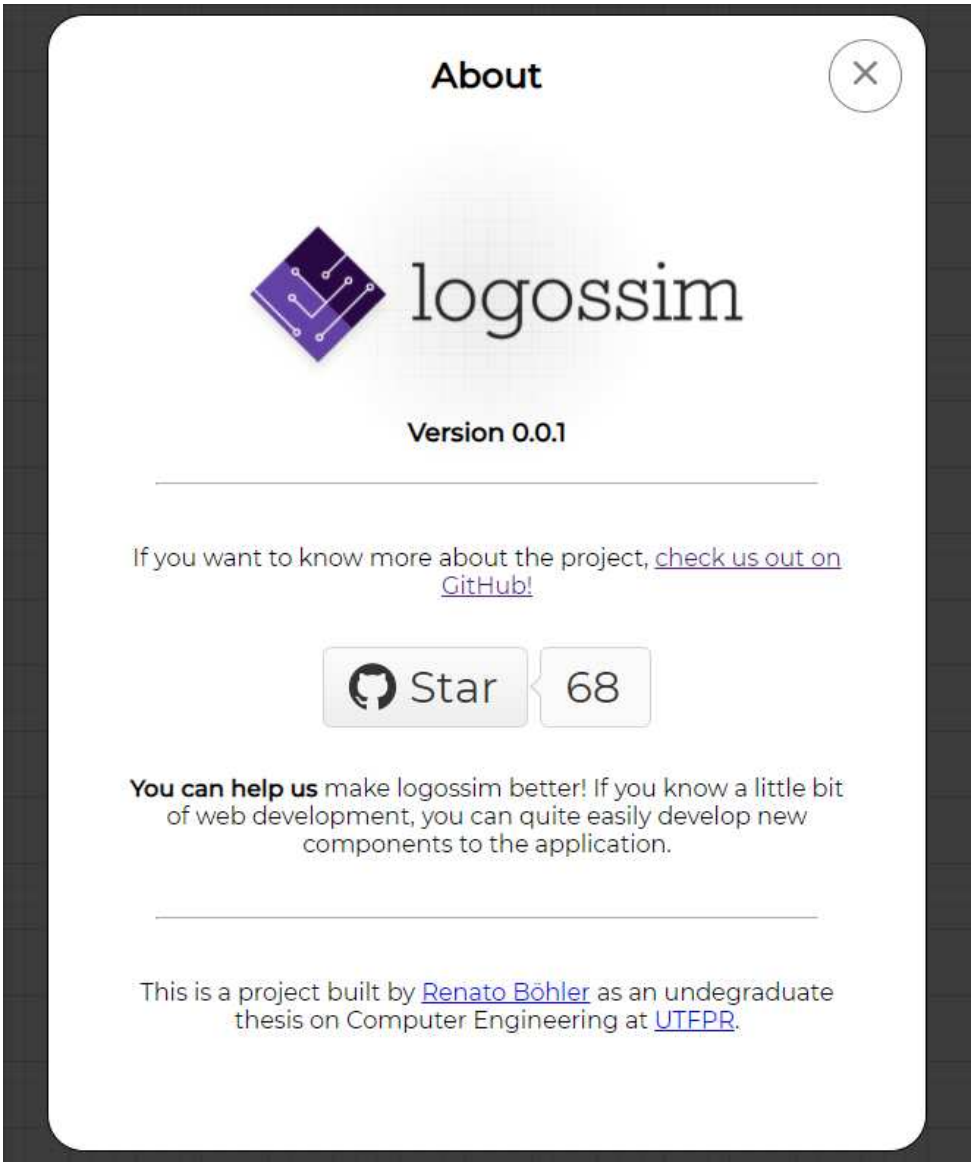
Atalho	**Ação**
CTRL + S	Salva o circuito
CTRL + L	Carrega um circuito
CTRL + Z	Desfaz a última ação da área de desenho
CTRL + SHIFT + Z	Refaz a última ação desfeita na área de desenho
CTRL + E	Edita configurações do componente selecionado
CTRL + D	Duplica os componentes selecionados
CTRL + C	Copia os componentes selecionados
CTRL + X	Recorta os componentes selecionados
DELETE	Remove os componentes e fios selecionados da área de desenho
CTRL + V	Cola os componentes copiados ou recortados
BARRA DE ESPAÇO	Inicia ou pausa a simulação
ESC	Interrompe a simulação

Fonte: Autoria própria

A segunda opção do menu dá ao usuário a oportunidade de refazer o tutorial guiado,

e a opção "sobre" traz algumas informações do projeto, como versão, autoria e um *link* para o repositório de código do projeto, hospedado no GitHub (Figura 37).

Figura 37 – Caixa de diálogo com informações sobre o projeto



Fonte: Autoria própria

3.4 Desenvolvimento dos componentes

No Logossim, um componente é composto por três partes principais: representação visual (como discutido na Subseção 3.3.1), configurações e modelagem.

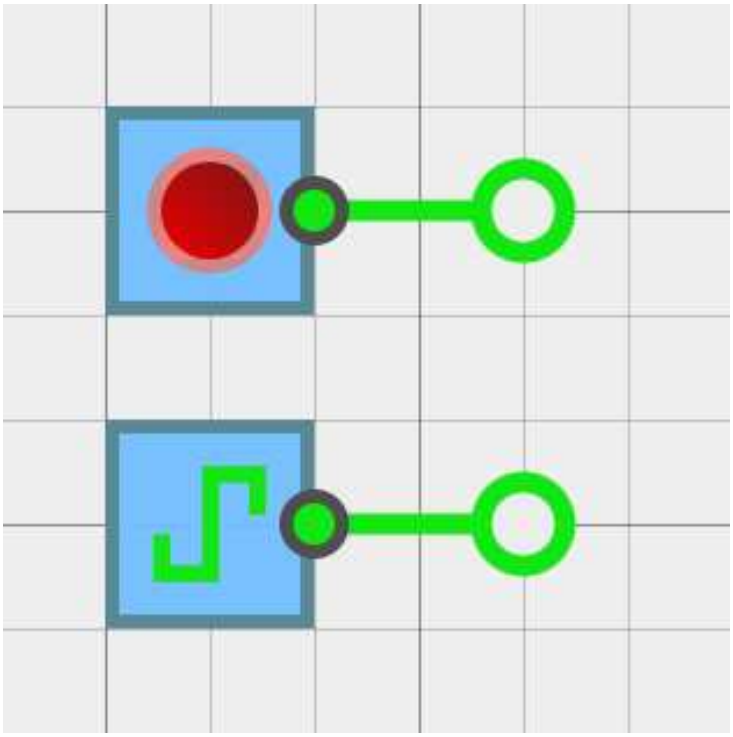
No total, foram desenvolvidos 27 componentes, sendo 8 componentes de entrada e saída, 10 portas lógicas, 4 componentes utilitários para cabeamento, além dos componentes multiplexador, demultiplexador, *RAM*, *ROM* e um contador. Uma descrição mais detalhada do funcionamento individual de cada componente pode ser encontrada no Apêndice A.

Nesta seção, serão apresentadas as configurações possíveis de um componente, bem como a definição do método de modelagem de um componente. Antes de discutir sobre modelagem e configurações de componentes, no entanto, será necessário primeiro apresentar alguns conceitos básicos.

3.4.1 Conceitos básicos

Os componentes do Logossim podem ser divididos em duas categorias: componentes ativos e componentes passivos. Um componente ativo é um componente que emite novos valores em suas portas de saída mesmo que nenhuma de suas entradas tenha sido alterada. Dois exemplos de componentes ativos são o *Button* e *Clock* (Figura 38). O componente *Button* emite o valor 1 enquanto está sendo pressionado pelo usuário, 0 em qualquer outro momento. Já o componente *Clock* alterna entre 0 e 1 com o período definido pela sua configuração.

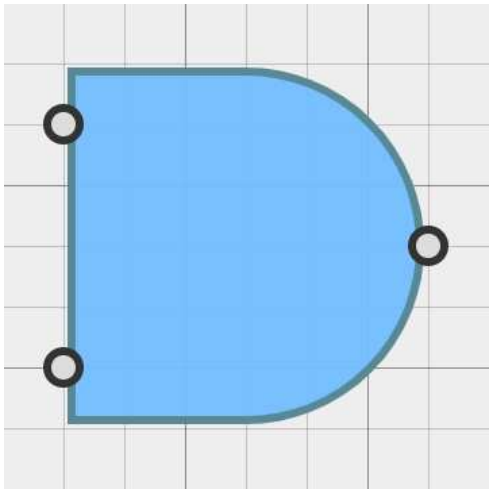
Figura 38 – Componentes passivos: *Button* e *Clock*



Fonte: Autoria própria

Já um componente passivo é um componente que altera uma de suas saídas como resposta à alteração de um dos seus valores de entrada. Como exemplo, podem ser citados todos os componentes da categoria "portas lógicas", como o componente *And* da Figura 39.

Figura 39 – Componente ativo: *And*

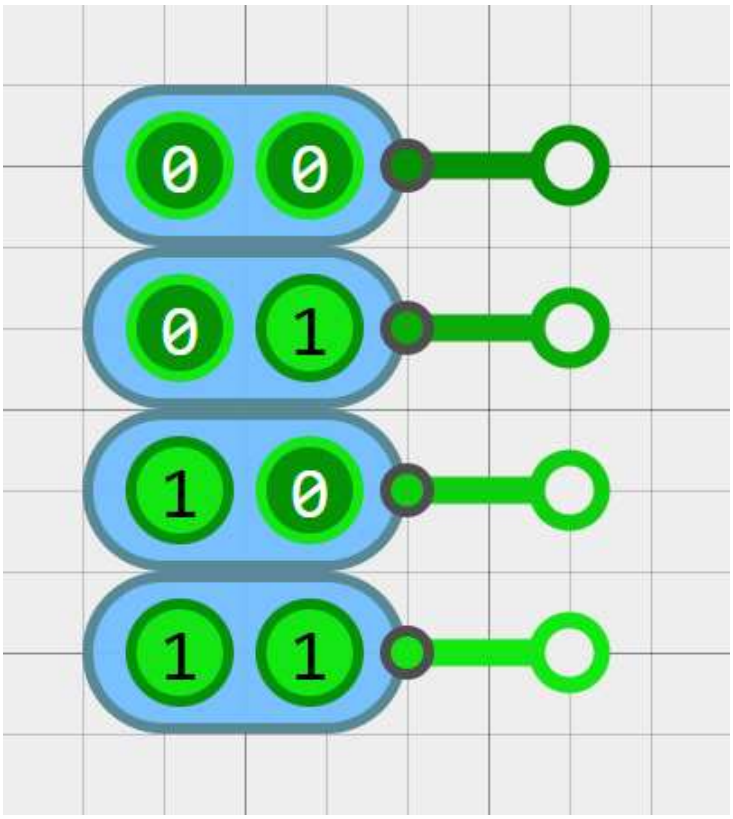


Fonte: Autoria própria

Portas de componentes, assim como fios, tem definida uma largura de *bits*, que pode ser 1, 2, 4, 8 ou 16. As portas podem assumir três tipos de valores: numéricos, flutuantes e erro.

Um valor numérico representa um valor definido pela porta. Por exemplo: uma porta com 1 *bit* de largura pode assumir os valores numéricos 0 e 1, já uma porta com 2 *bits* de largura pode assumir os valores numéricos 00, 01, 10 e 11. Estes valores são representados por tons de verde durante a simulação: quanto maior o valor numérico que uma porta assume, mais claro é o tom de verde, como ilustrado na Figura 40.

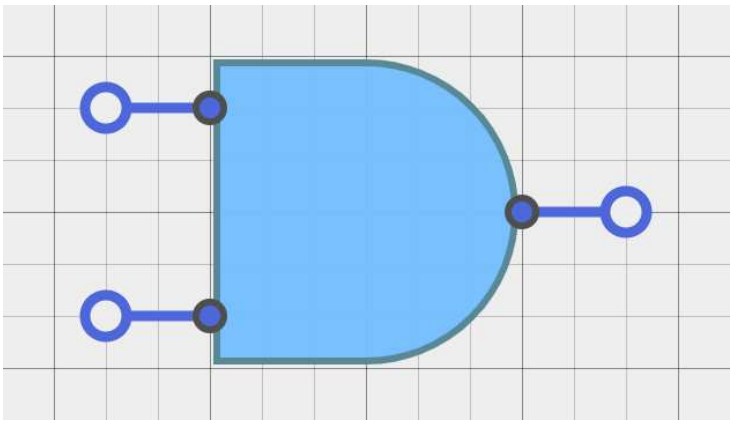
Figura 40 – Representação visual de todos os valores numéricos possíveis para uma largura de 2 *bits*



Fonte: Autoria própria

Um valor flutuante indica uma entrada ou saída indefinida, e é representado pelo símbolo x. Este é o caso, por exemplo, exibido na Figura 41: nenhuma das portas de entrada do componente *And* está conectada a outros componentes, portanto o valor de entrada é flutuante, e por consequência a saída também é.

Figura 41 – Representação visual de valores flutuantes

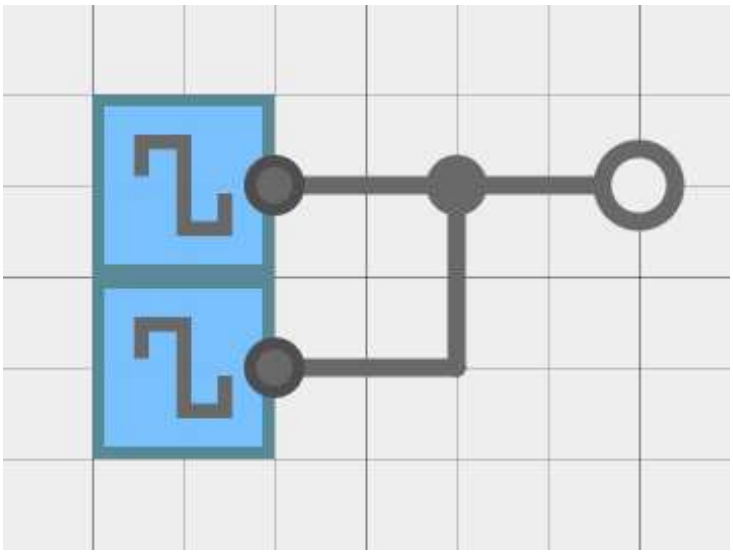


Fonte: Autoria própria

As portas de um componente transferem valores de saídas para os fios com os quais

estão conectados. Em alguns casos, no entanto, duas portas de saída podem estar conectadas ao mesmo fio, como na Figura 42.

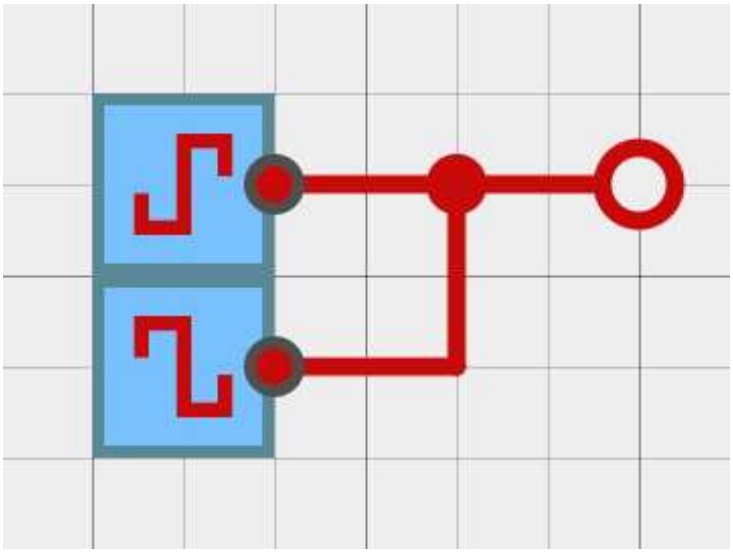
Figura 42 – Portas de saídas conectadas



Fonte: Autoria própria

Neste cenário, existe o risco de que as duas saídas emitam valores distintos entre si. Por exemplo: se um dos *Clock*s da Figura 42 estiver configurado com um período diferente do outro, em algum momento durante a simulação o valor 0 e o valor 1 serão emitidos no mesmo fio concorrentemente. Neste caso, o fio assume um valor específico para representar um cenário de erro. Este valor é indicado pela cor vermelha (Figura 43), e é representado pelo símbolo e.

Figura 43 – Portas de saídas conectadas: valores inconsistentes



Fonte: Autoria própria

O Quadro 3 resume os valores que portas e fios podem assumir durante a simulação.

Quadro 3 – Valores que portas e fios podem assumir

Nome	**Significado**	**Cor**	**Símbolo**
Numérico	Representa um valor definido	Tons de verde, quanto maior, mais claro	—
Flutuante	Representa um fio desconectado	Azul	x
Erro	Representa um fio com duas entradas de valores incoerentes	Vermelho	e

Fonte: Autoria própria

De modo geral, todo fio tem seu valor definido pelas portas de saída com a qual está conectado. Caso uma das portas possua o valor erro, o fio conduzirá o valor de erro (Figura 44a). Caso as portas possuam valores numéricos distintos entre si, o fio também conduzirá o valor de erro (Figura 44b). Caso todas as portas possuam o mesmo valor (seja ele numérico, flutuante ou erro), o fio conduzirá este valor (Figura 44c). Caso todas as portas, exceto as que possuem valor flutuante, possuam o mesmo valor, este valor será conduzido pelo fio (Figura 44d).

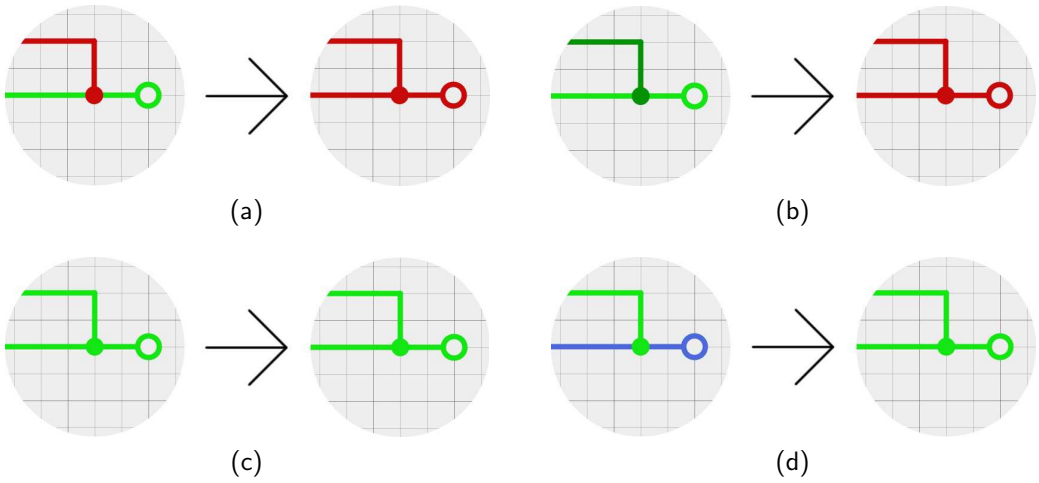


Figura 44 – Possíveis cenários de fios conectados a múltiplas saídas

Para conexões de largura de *bit* maior que 1, as regras descritas são aplicadas *bit* a *bit*. Numa conexão de largura de 4 *bits*, portanto, é possível que tenhamos um *bit* de valor 1, um *bit* de valor 0, um *bit* de valor flutuante e o último com valor de erro. Nestes cenários, a cor do conector respeita a seguinte regra: caso um dos *bits* possua o valor de erro, a coloração é vermelha; senão, caso um dos *bits* possua o valor flutuante, a coloração é azul; senão, a coloração é o tom de verde correspondente ao valor numérico.

3.4.2 Configurações em componentes

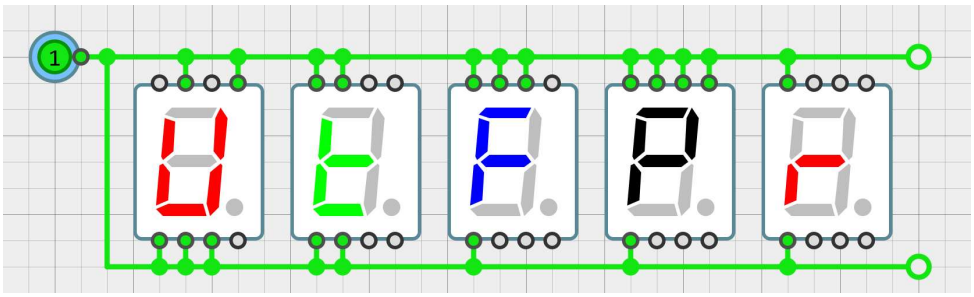
Assim como no Logisim, os componentes do Logossim são configuráveis. Configurações de componentes são definidas pelo usuário na caixa de diálogo da Figura 33, e parametri-

zam como o componente se comporta, tanto visualmente quanto durante a simulação. As configurações de um componente podem possuir três tipos diferentes: numérico, seletor e binário.

Uma configuração numérica é uma configuração que espera um número como valor. Este é o tipo de configuração utilizada, por exemplo, para parametrizar o período de oscilação do componente *Clock*.

Uma configuração por seleção possui opções pré-definidas para que o usuário escolha uma. Este tipo é utilizado no componente *display* de sete segmentos (SSD), para alterar a coloração dos segmentos ativos entre vermelho, verde, azul e preto, como ilustra a Figura 45.

Figura 45 – Parametrização da cor dos segmentos no componente SSD

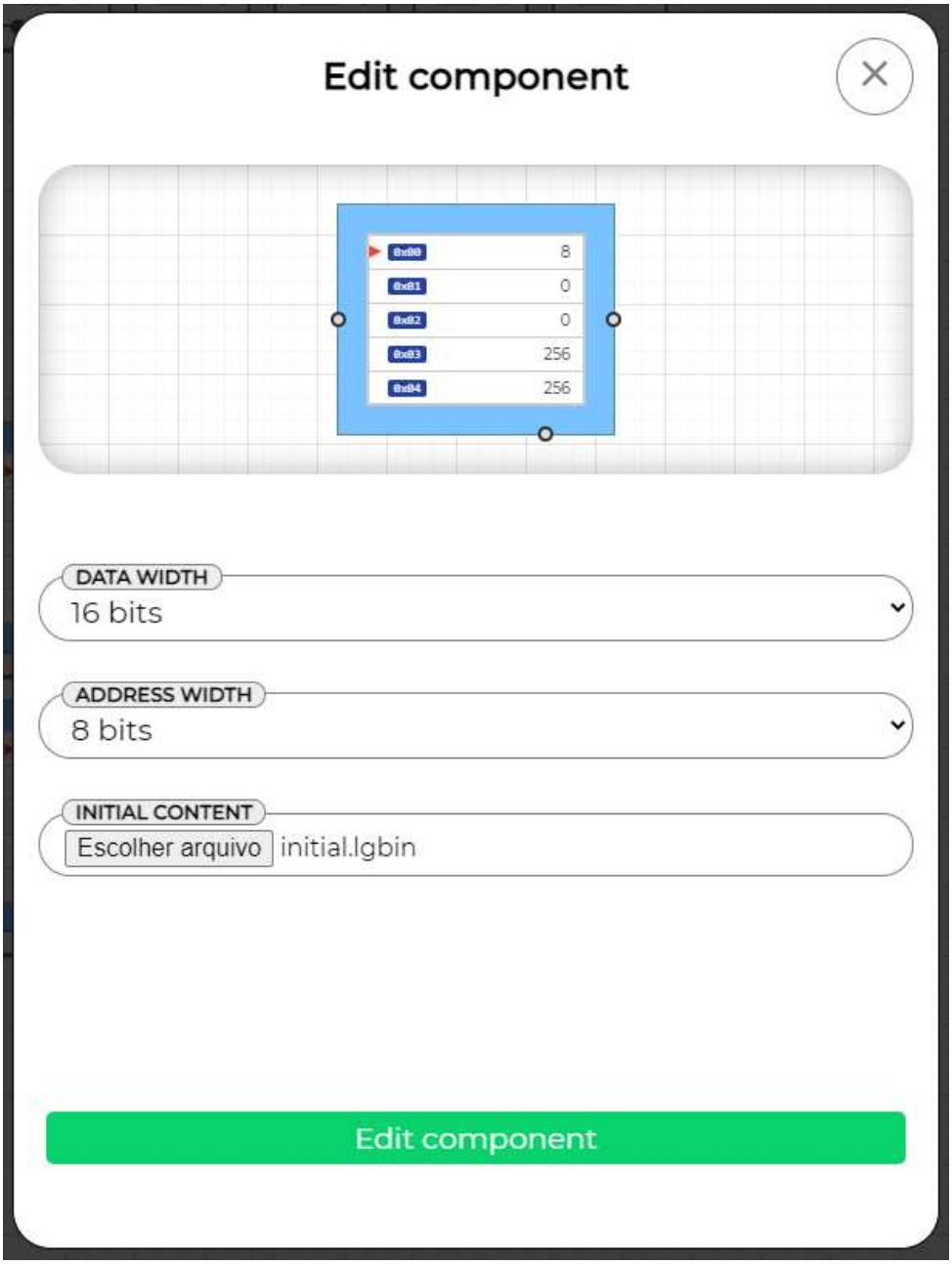


Fonte: Autoria própria

Uma configuração binária permite que o usuário selecione um arquivo com a extensão `lgbin`. Trata-se de um arquivo de texto, do qual serão extraídos os caracteres 0 e 1, que serão transformados nos valores numéricos correspondentes e repassado para o componente como valor de configuração.

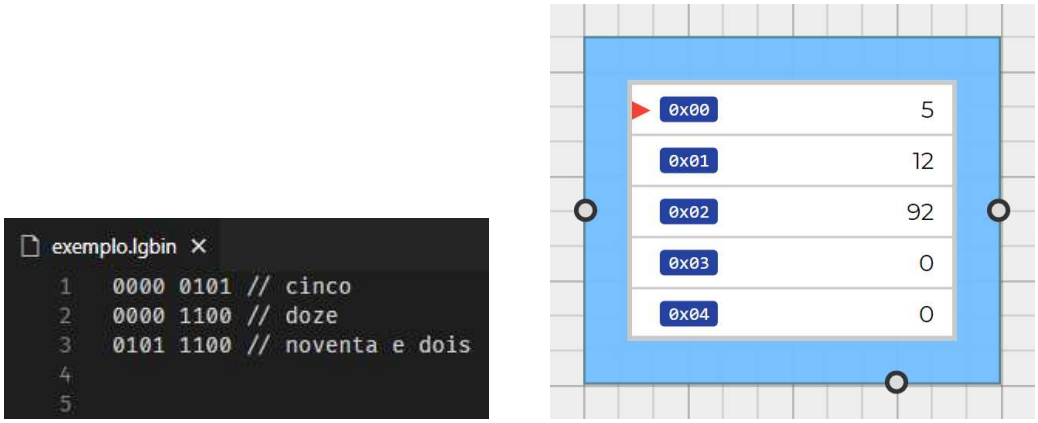
Este tipo de configuração é utilizado nos componentes RAM e ROM, para permitir que o usuário carregue valores iniciais nas células de memória, como ilustra a Figura 46.

Figura 46 – Caixa de diálogo de configuração do componente ROM



Fonte: Autoria própria

A Figura 47a ilustra o conteúdo de um arquivo `lgbin` que contém os números 5, 12 e 92 representados em 8 *bits*. Note que é possível utilizar espaços, quebras de linhas e até mesmo comentários textuais para organizar este arquivo. Já a Figura 47b exibe o componente ROM com os valores inicializados pelo arquivo de exemplo.



(a) (b)

Figura 47 – Exemplo de uso da configuração binária

O Quadro 4 resume todos os diferentes tipos de configuração que um componente pode receber.

Quadro 4 – Tipos de configurações disponíveis no Logossim

Tipo	**Significado**
Numérico	Permite que o usuário informe somente números
Seletor	Permite que o usuário selecione opções pré-definidas
Binário	Permite que o usuário selecione um arquivo `lgbin`

Fonte: Autoria própria

3.4.3 Modelagem para simulação

O comportamento de um componente durante a simulação é implementado numa classe chamada de `Model`, que estende a classe `BaseModel` exportada pelo pacote `@logossim/core`. A classe `Model` de um dado componente pode implementar os métodos do ciclo de vida do componente durante a simulação, que são 6 no total.

Três destes métodos dizem respeito ao estado de execução da simulação:

- `onSimulationStart`
- `onSimulationPause`
- `onSimulationStop`

O método `onSimulationStart` é executado quando a simulação é iniciada pelo usuário, e pode ser utilizado para emitir um valor nas portas de sáida do componente, ou

para definir um intervalo para execução de um código utilizando a função `setInterval` do JavaScript (MOZILLA, 2021c). O método `onSimulationPause` é executado quando a simulação é pausada pelo usuário, e pode ser utilizada para limpar intervalos definidos utilizando `setInterval`. O método `onSimulationStop` é executado quando a simulação é finalizada pelo usuário. Pode ser utilizado para limpar intervalos definidos por `setInterval` e para desalocar recursos, para liberar memória do sistema.

Outros três destes métodos são executados quando o valor de uma das entradas do componente são alteradas:

- `step`
- `stepFloating`
- `stepError`

O método `step` é executado quando todos os *bits* da entrada do componente são numéricos (Figura 48a). O método `stepError` é executado quando pelo menos um dos *bits* de entrada do componente possui o valor de erro (Figura 48b). O método `stepFloating` é executado quando pelo menos um dos *bits* de entrada possui o valor flutuante e nenhum *bit* possui o valor de erro (Figura 48c). O retorno destes métodos é um objeto que define um novo valor para as portas de saída do componente. Cada propriedade neste objeto de retorno é composto por uma chave, que deve ser o nome da porta de saída, e um valor, que será o novo valor assumido pela porta de saída.

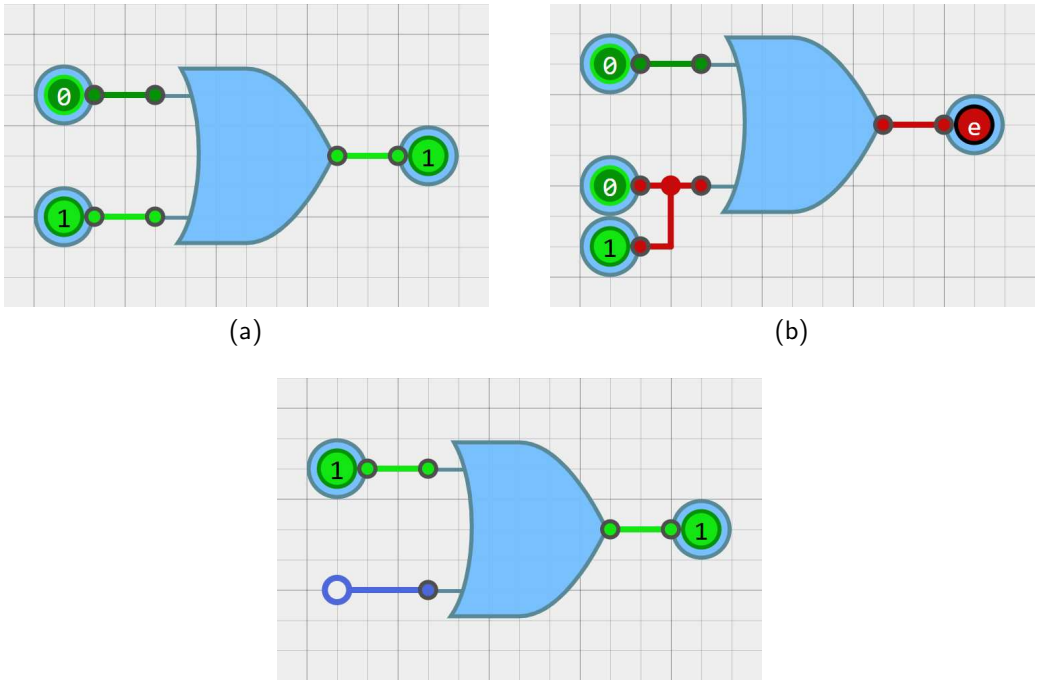


Figura 48 – Cenários de execução dos métodos *step*

O Quadro 5 resume os métodos de ciclo de vida de um componente durante a

simulação.

Quadro 5 – Métodos do ciclo de vida de um componente durante a simulação

Método	**Momento de execução**
`onSimulationStart`	Simulação iniciada
`onSimulationPause`	Simulação pausada
`onSimulationStop`	Simulação parada
`step`	Valor de entrada alterado. Todos *bits* de entrada são numéricos
`stepError`	Valor de entrada alterado. Pelo menos um *bit* de entrada é o valor de erro
`stepFloating`	Valor de entrada alterado. Pelo menos um *bit* de entrada é flutuante, e nenhum é valor de erro

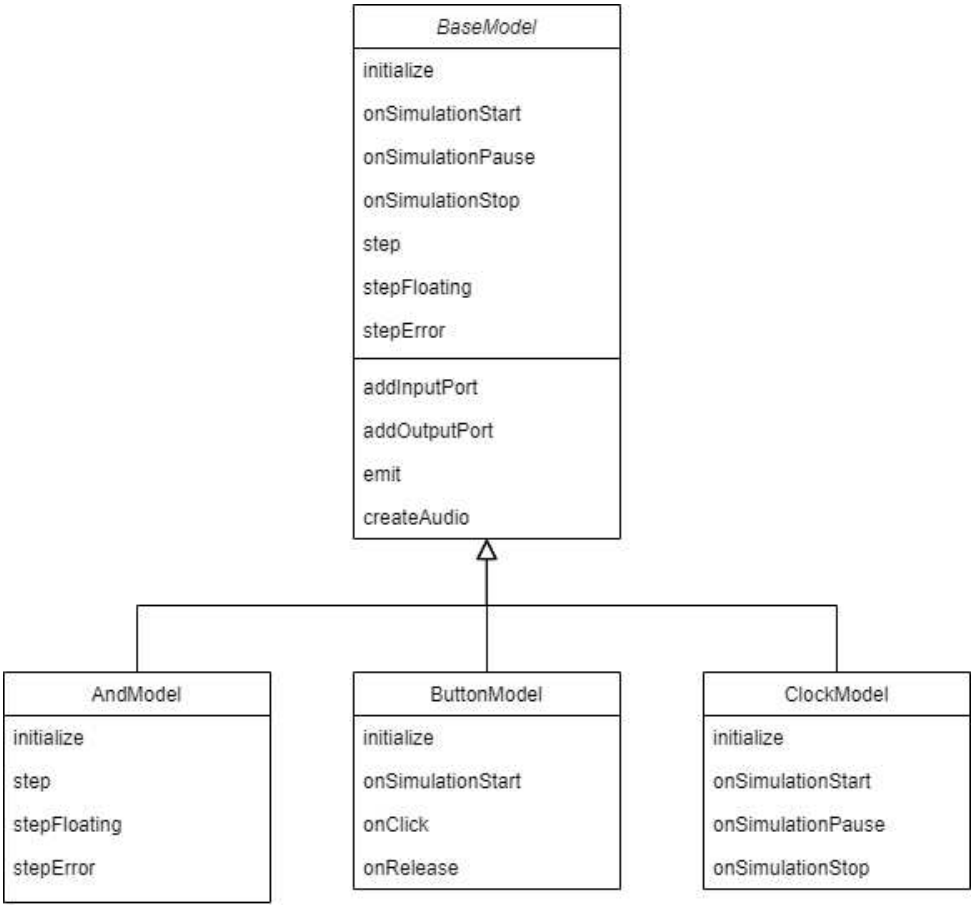
Fonte: Autoria própria

Além dos métodos do ciclo de vida do componente durante a simulação, existe o método `initialize`, que é executado quando o usuário adiciona um componente na área de desenho ou quando as configurações de um componente são alteradas. Este método recebe por parâmetro um objeto com os valores da configuração do componente, e é utilizado principalmente para configurar a largura de *bits* das portas do componente.

A classe `BaseModel` possui a implementação dos métodos `addInputPort`, para configurar portas de entrada, `addOutputPort`, para configurar portas de saída, `emit` para emissão de valores de saída em componentes ativos e ainda o método `createAudio`, que utiliza a API `AudioContext` do JavaScript (MOZILLA, 2021a) para gerar ondas sonoras. Estes métodos podem ser executados pelas classes `Model` dos componentes.

A Figura 49 exibe o diagrama de classes para os componentes *And* (passivo), *Button* e *Clock* (ativos).

Figura 49 – Diagrama de classes para os componentes *And*, *Button* e *Clock*



Fonte: Autoria própria

O Algoritmo 1 mostra uma versão simplificada da implementação da classe modelo para o componente *And*[5]. Esta implementação simplificada só dá suporte a largura de 1 bit, e não possui tratamento para entradas com valor de erro ou flutuantes. Das linhas 4 a 13, o método `initialize` usa o valor da configuração `INPUT_PORTS_NUMBER` para configurar a quantidade correta de portas de entrada e uma de saída com largura de 1 *bit*. Das linhas 15 a 19, o método `step` define como o componente responde a alterações na entrada, transformando o valor da porta de saída `out` para 1 caso todos os *bits* de entrada sejam 1, 0 caso contrário.

```
1 import { BaseModel } from '@logossim/core';
2
3 export default class AndModel extends BaseModel {
4   initialize(configurations) {
5     const INPUT_PORTS_NUMBER = Number(
6       configurations.INPUT_PORTS_NUMBER,
7     );
8
```

[5] A implementação completa está disponível em: <https://github.com/renato-bohler/logossim/blob/HEAD/packages/@logossim/components/And/AndModel.js>

```
9       for (let i = 0; i < INPUT_PORTS_NUMBER; i += 1) {
10        this.addInputPort(`in${i}`, { bits: 1 });
11      }
12      this.addOutputPort('out', { bits: 1 });
13    }
14
15    step(input) {
16      return {
17          out: Object.values(input).every(bit => bit === 1) ? 1 : 0;
18      };
19    }
20
21    stepError(input) {
22      return this.step(input);
23    }
24
25    stepFloating(input) {
26      return this.step(input);
27    }
28  }
```

Algoritmo 1 – Implementação simplificada de AndModel (passivo)

O Algoritmo 2 é a implementação da classe modelo do componente *Button*. No método `initialize`, a única porta de saída (chamada de `out`) é configurada. Ao inicializar a simulação, na linha 9, o valor 0 é emitido. Das linhas 12 a 18, os métodos `onClick` e `onRelease` são declarados. Estes métodos são executados pelo componente React `ButtonWidget`, ao apertar e soltar o botão primário do *mouse*[6].

```
1   import { BaseModel } from '@logossim/core';
2
3   export default class ButtonModel extends BaseModel {
4     initialize() {
5       this.addOutputPort('out');
6     }
7
8     onSimulationStart() {
9       this.emit({ out: 0 });
10    }
11
12    onClick() {
13      this.emit({ out: 1 });
14    }
15
16    onRelease() {
```

[6] <https://github.com/renato-bohler/logossim/blob/HEAD/packages/@logossim/components/Button/ButtonWidget.jsx#L87-L88>

```
17      this.emit({ out: 0 });
18    }
19  }
```

Algoritmo 2 – Implementação de ButtonModel (ativo)

O Algoritmo 3 é uma implementação simplificada da classe de modelo do componente *Clock*, considerando somente a configuração da frequência de oscilação `FREQUENCY_HZ`[7]. Na inicialização do componente (linhas 4 a 10), a porta de saída `out` é configurada e algumas propriedades auxiliares são definidas: `output` representa o valor de saída atual, `emitInterval` é o ID do intervalo gerado pelo JavaScript e `periodMs` é o período de oscilação definido em função da frequência configurada pelo usuário, em milissegundos. Assim que a simulação é iniciada, um intervalo é criado para executar a função definida nas linhas 15 a 18 a cada meio período, alternando o valor de saída entre 0 e 1 e o emitindo. Quando a simulação é pausada ou finalizada (linhas 21 a 27), o intervalo `emitInterval` é limpo.

```
1   import { BaseModel } from '@logossim/core';
2
3   export default class ClockModel extends BaseModel {
4     initialize(configurations) {
5       this.addOutputPort('out');
6
7       this.output = 0;
8       this.emitInterval = null;
9       this.periodMs = 1000 / configurations.FREQUENCY_HZ;
10    }
11
12    onSimulationStart() {
13      this.emit({ out: this.output });
14
15      this.emitInterval = setInterval(() => {
16        this.output = this.output === 0 ? 1 : 0;
17        this.emit({ out: this.output });
18      }, this.periodMs / 2);
19    }
20
21    onSimulationPause() {
22      clearInterval(this.emitInterval);
23    }
24
25    onSimulationStop() {
26      clearInterval(this.emitInterval);
27    }
28  }
```

[7] A implementação completa está disponível em: <https://github.com/renato-bohler/logossim/blob/HEAD/packages/@logossim/components/Clock/ClockModel.js>

Algoritmo 3 – Implementação simplificada de ClockModel (ativo)

3.4.4 Testes unitários

Foram implementados, utilizando Jest, testes unitários de modelagem e de comportamento visual para todos os componentes da aplicação. Todo teste unitário foi organizado em três passos: arrumar, agir e aferir.

- **arrumar:** preparar o cenário do teste, instanciando o componente com as configurações desejadas para o teste;
- **agir:** consiste em executar um método da classe, ou simular uma interação do usuário com um componente visual da aplicação;
- **aferir:** é o ato de validar os resultados obtidos, seja ele o retorno do método chamado ou a verificação de que outros métodos foram corretamente executados durante o teste. Esta é a etapa que conclui se o teste unitário passou ou falhou.

A unidade de teste da modelagem é a classe `Model` de um componente. Este tipo de teste impõe à classe situações de simulação e afere os resultados obtidos. Por exemplo: um dos testes unitários da classe `ClockModel`[8] garante que o sinal emitido pelo componente possui os valores e tempos corretos, dada uma configuração específica. Uma série de testes unitários da classe `AndModel`[9] simula um conjunto de entradas e afere o valor de retorno.

Para testar o comportamento visual de um componente, a unidade de testes é o arquivo `Widget` deste componente. Neste tipo de teste, é possível verificar que o componente possui a quantidade correta de portas dada uma configuração ou até mesmo simular interações do usuário e verificar o correto funcionamento do componente. Como exemplo, os testes de `ButtonWidget`[10] verificam que existe uma porta de saída para este componente, e que os métodos `onClick` e `onRelease` da instância da classe de modelo são executados quando o usuário pressiona e solta o botão do *mouse*, correspondentemente.

Executar os testes unitários do projeto é simples: basta abrir o terminal na pasta do projeto e executar o comando `yarn test`. Por padrão, o Jest executa testes somente de arquivos alterados. O usuário pode pressionar a tecla `A` para executar todos os testes do projeto.

Também é possível gerar um relatório de cobertura de testes, executando o comando `yarn test:coverage`.

A Tabela 1 apresenta valores percentuais para diferentes tipos de cobertura de todos os componentes da aplicação. Segundo Arguelles, Ivanković e Bender (2020), testes com cobertura

[8] <https://github.com/renato-bohler/logossim/blob/HEAD/packages/@logossim/components/Clock/__tests__/ClockModel.test.js#L23-L42>

[9] <https://github.com/renato-bohler/logossim/blob/HEAD/packages/@logossim/components/And/__tests__/AndModel.test.js#L32-L108>

[10] <https://github.com/renato-bohler/logossim/blob/HEAD/packages/@logossim/components/Button/__tests__/ButtonWidget.test.jsx>

de 60% são considerados aceitáveis, e 90% considerados exemplares. Na média, portanto, a cobertura de testes de cada tipo atingiu no mínimo níveis aceitáveis, e em boa parte, níveis exemplares.

Tabela 1 – Cobertura de testes dos componentes

	Declarações	Ramificações	Funções	Linhas
`And`	85,71%	64,71%	88,24%	85,29%
`Buffer`	90,48%	66,67%	87,50%	94,44%
`Button`	92,00%	62,50%	84,62%	92,00%
`Buzzer`	76,47%	50,00%	84,62%	83,87%
`Clock`	73,21%	50,00%	68,75%	72,73%
`ControlledBuffer`	81,25%	70,59%	70,00%	84,00%
`ControlledInverter`	84,62%	78,26%	72,73%	86,21%
`Counter`	63,83%	50,00%	63,64%	60,98%
`Demux`	96,88%	68,75%	90,00%	100,00%
`Ground`	92,31%	33,33%	80,00%	100,00%
`Input`	97,44%	81,25%	94,44%	97,22%
`Joiner`	93,55%	75,00%	85,71%	96,43%
`Led`	93,33%	81,25%	90,00%	100,00%
`Mux`	96,77%	68,75%	88,89%	100,00%
`Nand`	85,71%	64,71%	88,24%	85,29%
`Nor`	86,67%	75,76%	89,47%	86,49%
`Not`	91,67%	72,73%	88,89%	95,00%
`Or`	86,67%	75,76%	89,47%	86,49%
`Output`	95,83%	81,82%	93,33%	95,65%
`Power`	92,31%	40,00%	80,00%	100,00%
`Ram`	95,52%	84,21%	92,31%	96,61%
`Rom`	94,12%	75,00%	91,67%	95,56%
`Splitter`	93,10%	83,33%	85,71%	96,30%
`Ssd`	97,83%	65,63%	94,44%	97,67%
`Switch`	95,24%	60,00%	91,67%	95,24%
`Xnor`	85,11%	75,68%	90,00%	84,62%
`Xor`	85,11%	75,68%	90,00%	84,62%
Média	88,99%	67,83%	85,72%	90,84%

Fonte: Autoria própria

3.5 Serialização e desserialização de circuitos

O Logossim permite que o usuário salve um circuito num arquivo com extensão `lgsim`. Este arquivo pode então ser carregado no Logossim, recuperando o circuito anteriormente salvo.

Para poder gerar este arquivo, é preciso serializar o circuito. Serializar o circuito significa transformar todas as informações necessárias para recriar este mesmo circuito, de

modo que elas possam ser salvas num arquivo. Analogamente, o processo que recria o circuito a partir de um arquivo se chama desserialização.

Convenientemente, a biblioteca React Diagrams possui a implementação de serialização e desserialização de diagramas.

O arquivo `lgsim` possui o formato JSON, e contém entre outras, as seguintes informações:

- nome do circuito;
- data e hora da criação do circuito;
- data e hora da última modificação realizada no circuito;
- componentes existentes no circuito, sendo que cada componente possui as seguintes informações:
  - posição do componente no circuito;
  - configurações do componente;
  - portas existentes no componente.
- fios existentes no circuito, sendo que cada fio possui as seguintes informações:
  - portas com as quais ele está conectado;
  - localização dos pontos por onde o fio passa;
  - largura de bits do fio.

3.6 Motor de simulação

Para evitar que a aplicação se torne irresponsiva e melhorar a performance durante a simulação, o algoritmo de simulação (discuto em maiores detalhes na Seção 3.7) foi implementado para ser executado dentro de um *web worker*.

Um *web worker* executa código JavaScript numa *thread* e contexto diferentes da *thread* principal. A *thread* principal se comunica com o *web worker* através de mensagens. Nesta seção, será detalhado como esta comunicação entre as duas *threads* é realizada.

Cada mensagem enviada representa um comando. Comandos podem ser enviados pela *thread* principal para a *thread* de simulação e vice-versa.

O comando `START` é o comando executado para inicializar a simulação. Como a *thread* de simulação não possui acesso a elementos da área de desenho (pois ela está num contexto de *web worker*), a *thread* principal envia um objeto que representa a estrutura do circuito atual para a *thread* de simulação. Neste objeto, um identificador único universal (UUID) é atribuído a cada componente, porta e fio para identificá-los. Este objeto é uma versão simplificada do circuito, contendo somente as informações necessárias para a execução do algoritmo de simulação: uma lista de todos componentes no circuito e todas as conexões entre eles.

O comando `PAUSE` é executado quando o usuário pausa a simulação. Isto faz com que a execução do algoritmo de simulação seja temporariamente interrompida. O comando `START`,

quando a simulação está pausada, retoma a execução do algoritmo de simulação.

Quando o usuário finaliza a simulação, o comando `STOP` é enviado à *thread* de simulação. Este comando interrompe definitivamente a execução do algoritmo de simulação, além de desalocar os recursos utilizados pela *thread*.

Para componentes que emitem valores como resposta à interação do usuário (como o componente *Button*, por exemplo), o comando `EMIT` é responsável por transmitir eventos de emissão de valores à *thread* de simulação.

O único comando que parte da *thread* de simulação para a *thread* principal é o `DIFF`. Este comando envia alterações em valores de portas e fios de volta à *thread* principal, para que a área de desenho possa ser atualizada correspondentemente.

O Quadro 6 apresenta um resumo dos comandos implementados.

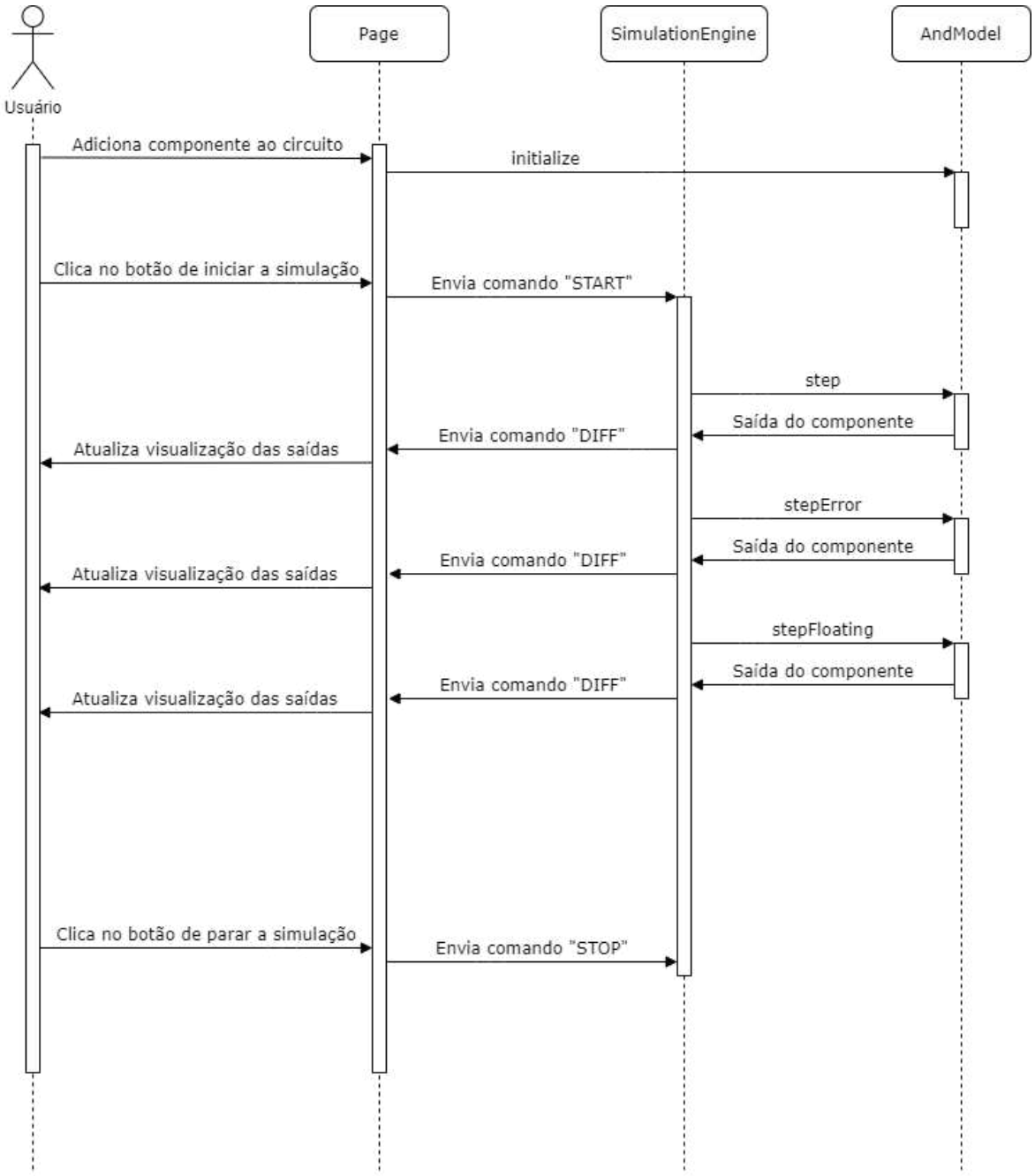
Quadro 6 – Comandos para comunicação entre as *threads* principal e de simulação

Comando	**Descrição**
`START`	Inicia o algoritmo de simulação
`PAUSE`	Interrompe temporariamente a execução da simulação
`STOP`	Interrompe definitivamente a execução da simulação
`EMIT`	Transmite valores emitidos por componentes
`DIFF`	Envia alterações de valores da simulação para a interface gráfica

Fonte: Autoria própria

Como exemplo, considere o componente passivo *And*. Os valores de saída deste componente são definidos pelo retorno dos métodos `step`, `stepError` e `stepFloating`. A Figura 50 exibe um diagrama de sequência que exemplifica momentos em que os comandos são transmitidos pela interface para a *thread* de simulação.

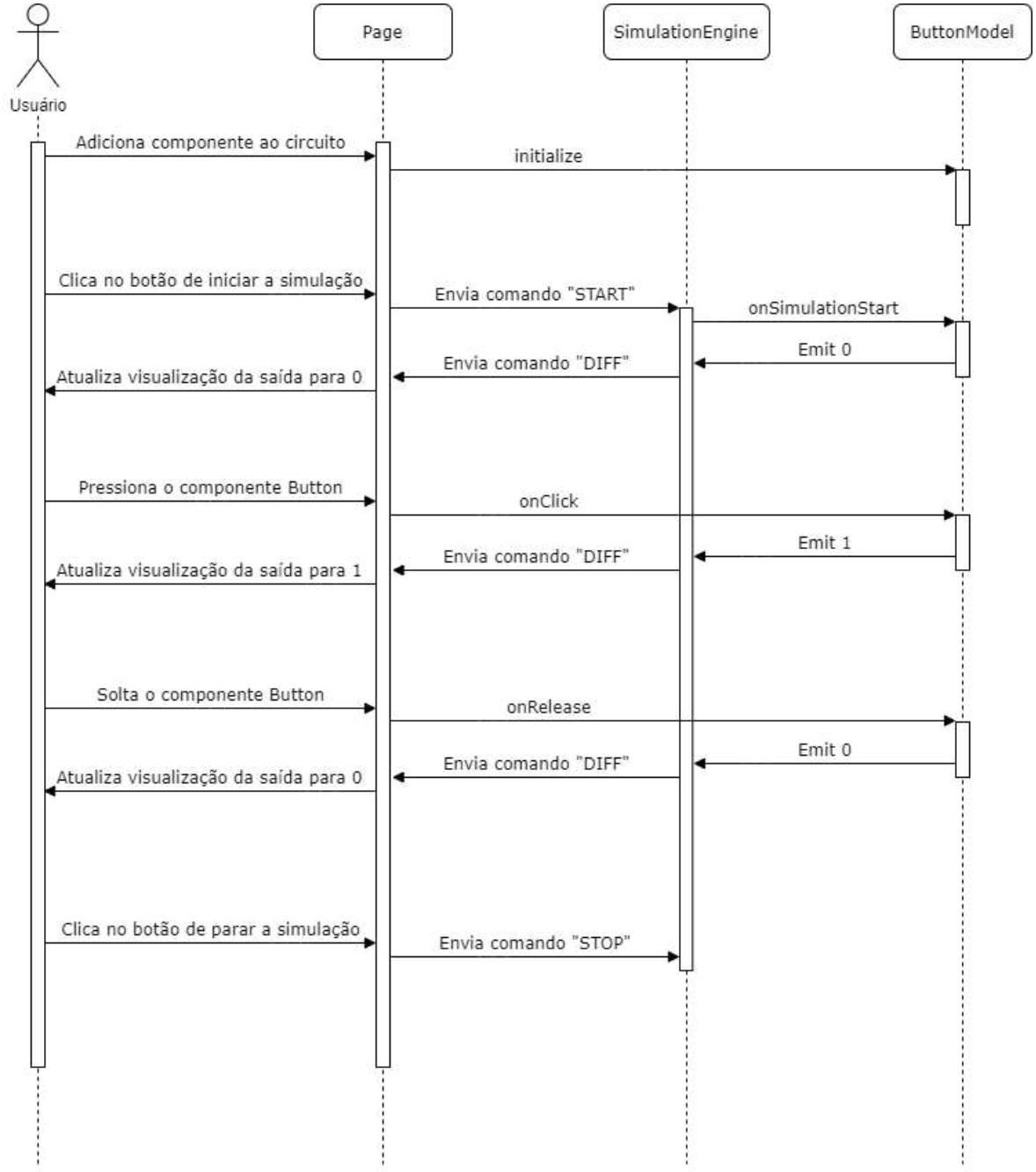
Figura 50 – Exemplo de diagrama de sequência para AndModel (passivo)



Fonte: Autoria própria

O componente *Button* é um componente ativo, que altera seus valores de saída chamando o método `emit`. Este componente emite o valor 0 ao inicializar a simulação, emite 1 quando o usuário pressiona o botão e 0 quando o usuário o solta. A Figura 51 exibe um diagrama de sequência que detalha os eventos envolvidos durante a interação do usuário com este componente.

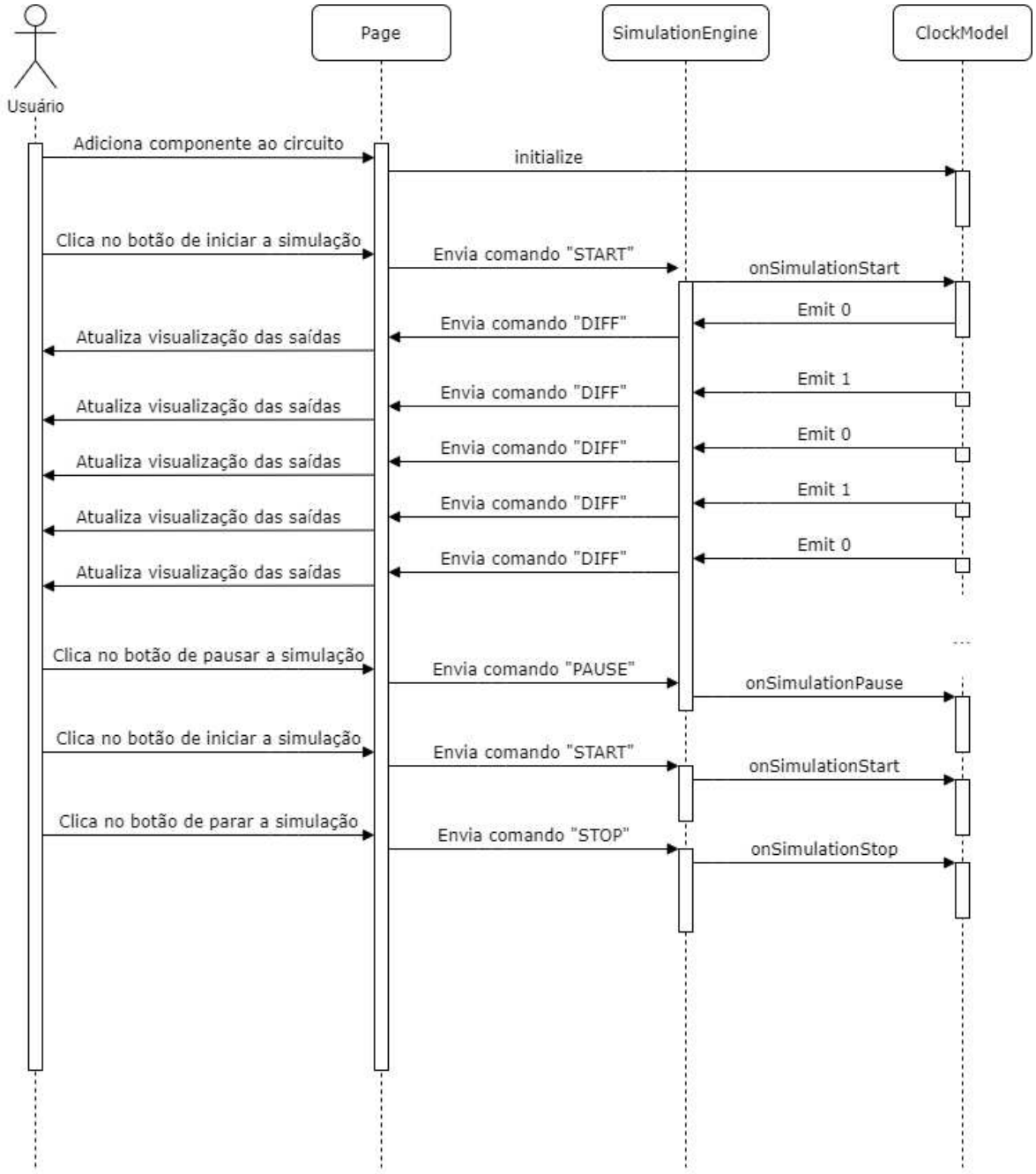
Figura 51 – Exemplo de diagrama de sequência para ButtonModel (ativo)



Fonte: Autoria própria

O *Clock* também é um componente ativo, mas difere do *Button* no fato de que novos valores de saída são emitidos sem a necessidade da interação do usuário com o componente, conforme ilustra o diagrama de sequências da Figura 52.

Figura 52 – Exemplo de diagrama de sequência para ClockModel (ativo)



Fonte: Autoria própria

3.7 Algoritmo de simulação

O algoritmo de simulação é o responsável por inicializar e executar a simulação, controlando a emissão de valores de componentes ativos e propagando estas alterações para os componentes conectados, até que os valores do circuito se estabilizem. A implementação deste algoritmo foi baseada no comportamento observado no programa Logisim, utilizando a ferramenta "Simulação passo-a-passo" do mesmo.

Durante a inicialização, o valor flutuante é atribuído a todos os fios. O método `onSimulationStart` é então executado em todos os componentes do circuito, permitindo que componentes ativos emitam seus valores iniciais.

Quando o valor de uma porta de saída é alterado, o valor dos fios conectados a esta saída é atualizado (conforme Figura 44). Caso o valor que o fio transmite tenha sido alterado, esta alteração é propagada para todos os componentes que possuem portas de entrada conectadas a este fio, executando o método `step` correspondente aos valores de entrada do componente (conforme Figura 48).

Para ilustrar o comportamento que este algoritmo implementa, o circuito da Figura 53 apresenta um cenário de simulação. O primeiro estado (Figura 53a) exibe os fios inicializados com o valor flutuante. Logo em seguida, os dois botões emitem o valor inicial de 0 (Figura 53b). Este valor é propagado para a porta lógica "OU", que altera sua saída para 0 (Figura 53c). Quando o usuário clica em um dos botões, o valor 1 é emitido em sua saída (Figura 53d). Esta alteração é propagada para a porta lógica, que altera sua saída para 1, e, por sua vez, esta alteração é propagada para o componente LED, que é ativado (Figura 53e).

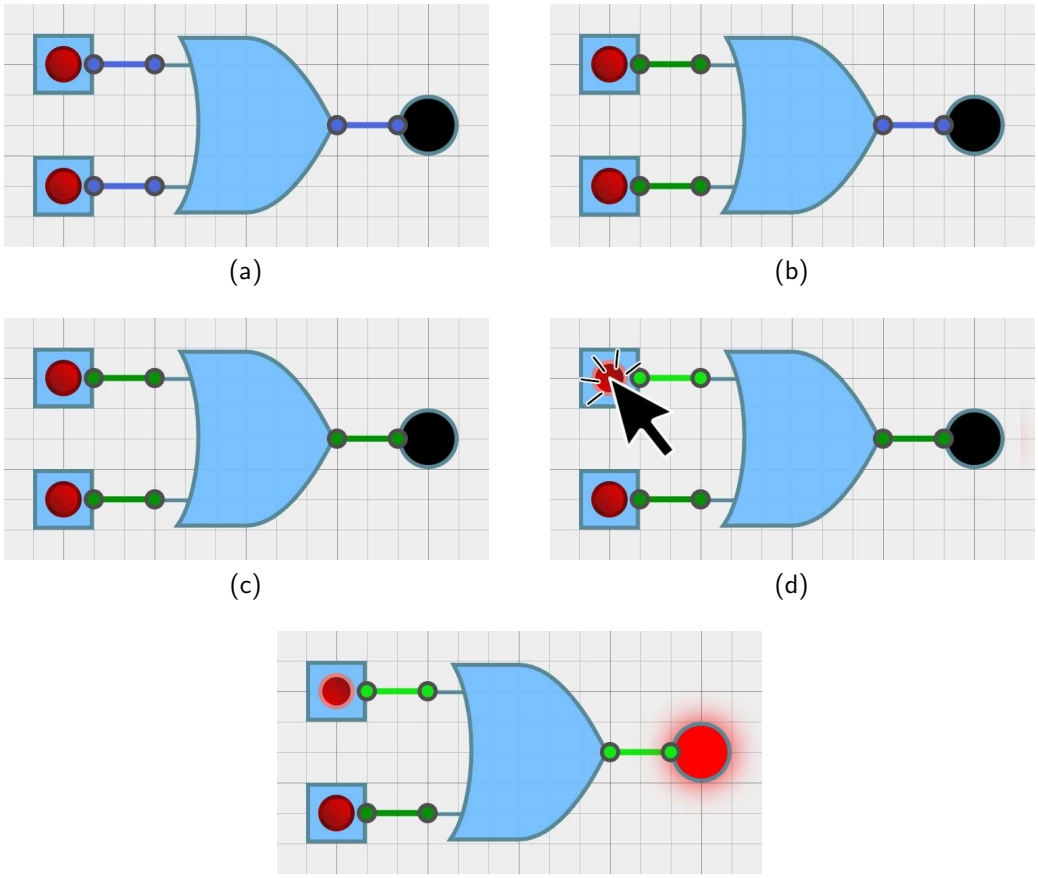


Figura 53 – Fases de um cenário simples de simulação

A implementação deste algoritmo utiliza duas filas: uma fila de emissão e uma de propagação. A fila de emissão contém valores emitidos por componentes ativos que estão com sua execução pendente. Enquanto a fila de emissão não estiver vazia, o algoritmo aplica o valor emitido à saída correspondente e adiciona todos os componentes conectados a esta saída à fila de propagação. O método `step` de todos os componentes da fila de propagação é executado, o que possivelmente irá alterar suas saídas, adicionando mais componentes à fila de propagação. Todos os componentes da fila de propagação devem ser tratados antes que o próximo valor emitido seja executado.

Pelas características do algoritmo de simulação, o Logossim pode ser categorizado como um simulador orientado a eventos de atraso zero. Uma simulação orientada a eventos executa a simulação somente sobre componentes que sofreram alterações nos seus sinais de entrada, e simulações de atraso zero são simulações que desconsideram atrasos inerentes à propagação de sinais e dos próprios circuitos (MICZO, 2007).

3.8 Criador de componentes

Para facilitar a criação de componentes durante o desenvolvimento do projeto, foi desenvolvida uma ferramenta de linha de comando utilizando a biblioteca Plop.

Para utilizar esta ferramenta, é necessário abrir o terminal na pasta raiz do projeto e executar o comando `yarn create-component`, seletor da Figura 54 será exibido, onde o usuário poderá selecionar um modelo que servirá como base para a criação do componente.

Figura 54 – Primeira questão do utilitário de linha de comando para criação de componentes

Fonte: Autoria própria

Após a seleção do componente base, o usuário deverá digitar um nome e descrição para seu novo componente, conforme ilustrado nas Figuras 55 e 56, respectivamente.

Figura 55 – Segunda questão do utilitário de linha de comando para criação de componentes

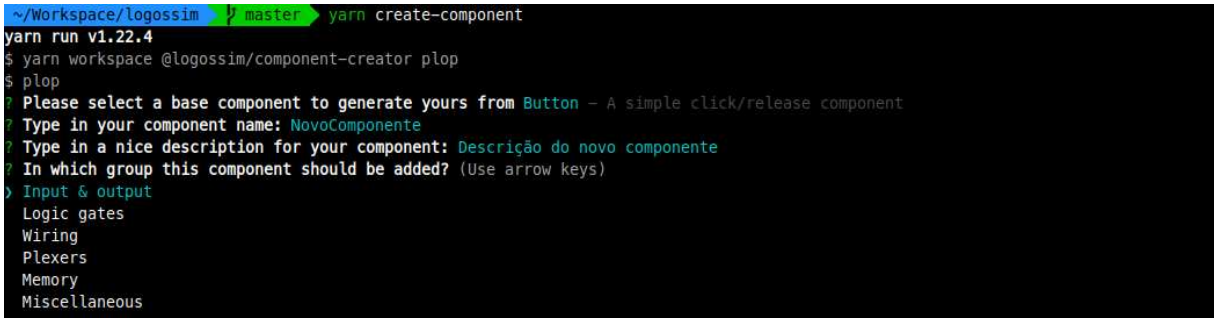
Fonte: Autoria própria

Figura 56 – Terceira questão do utilitário de linha de comando para criação de componentes

Fonte: Autoria própria

A última questão diz respeito à categoria a qual o novo componente deverá pertencer. Um seletor com todas as categorias disponíveis no Logossim é disponibilizada ao usuário, conforme mostra a Figura 57.

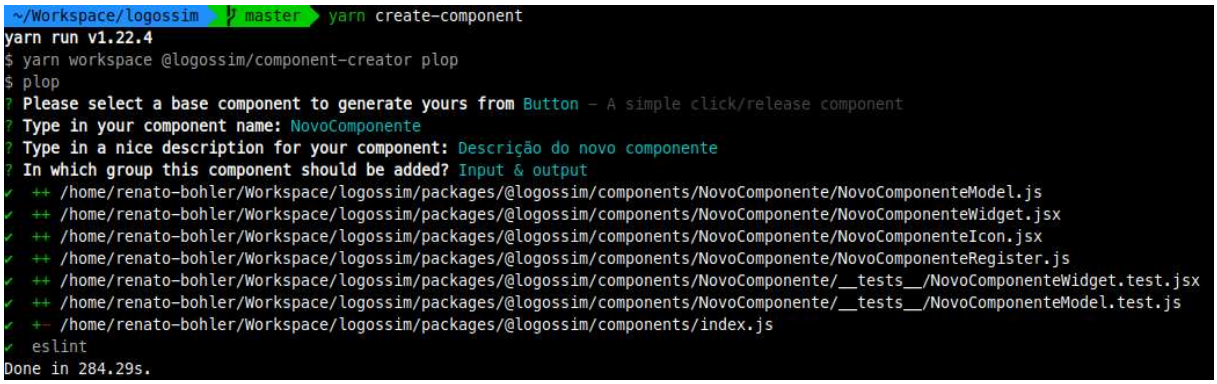
Figura 57 – Quarta questão do utilitário de linha de comando para criação de componentes



Fonte: Autoria própria

Como resultado, todos os diretórios e arquivos necessários para o componente (incluindo testes unitários) são automaticamente criados de acordo com as configurações que o usuário selecionou. A Figura 58 exibe o nome dos arquivos criados para este exemplo.

Figura 58 – Resultado da execução do utilitário de linha de comando para criação de componentes



Fonte: Autoria própria

3.9 Publicação da aplicação

Para a publicação do Logossim na *web*, foram utilizados os serviços GitHub Pages e Travis CI.

O repositório do projeto foi configurado para que toda alteração enviada ao ramo principal de desenvolvimento ativasse automaticamente uma máquina virtual no Travis CI. Esta máquina executa todos os testes unitários do projeto, e em caso de sucesso, executa a construção da aplicação. Após a construção do projeto, ele é automaticamente publicado no GitHub Pages do Logossim[11]. Todo o histórico de publicações é disponibilizado, e contém os *logs* de execução dos testes, da construção e publicação da aplicação[12].

[11] <https://renato-bohler.github.io/logossim>
[12] Disponível em: <https://travis-ci.com/github/renato-bohler/logossim>

4 RESULTADOS E DISCUSSÕES

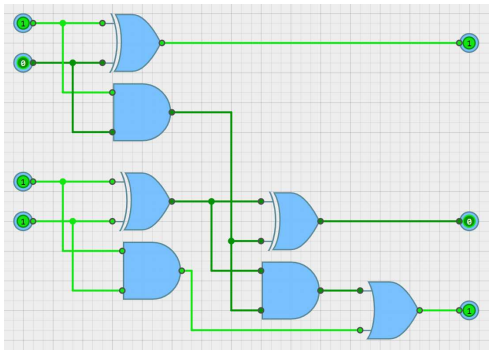
Neste capítulo, serão analisados e discutidos alguns resultados obtidos no desenvolvimento do projeto.

4.1 Circuitos testados

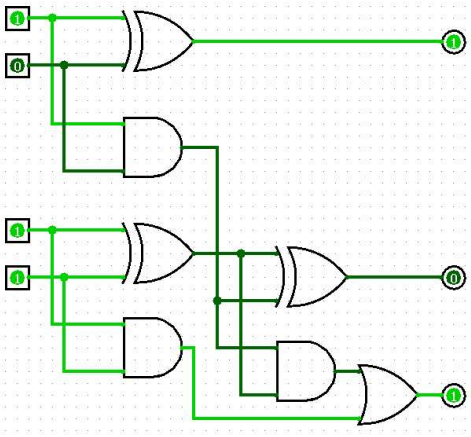
Para garantir que o Logossim funciona corretamente como simulador, uma série de circuitos foram desenhados tanto no Logossim quanto no Logisim e os resultados entre as duas aplicações foi comparado.

4.1.1 Circuitos combinacionais

Dentre os circuitos combinacionais validados estão o somador de 2 *bits*[1] da Figura 59 e o decodificador binário-SSD[2] da Figura 60. Ambos exibiram o mesmo comportamento nos dois *softwares*.



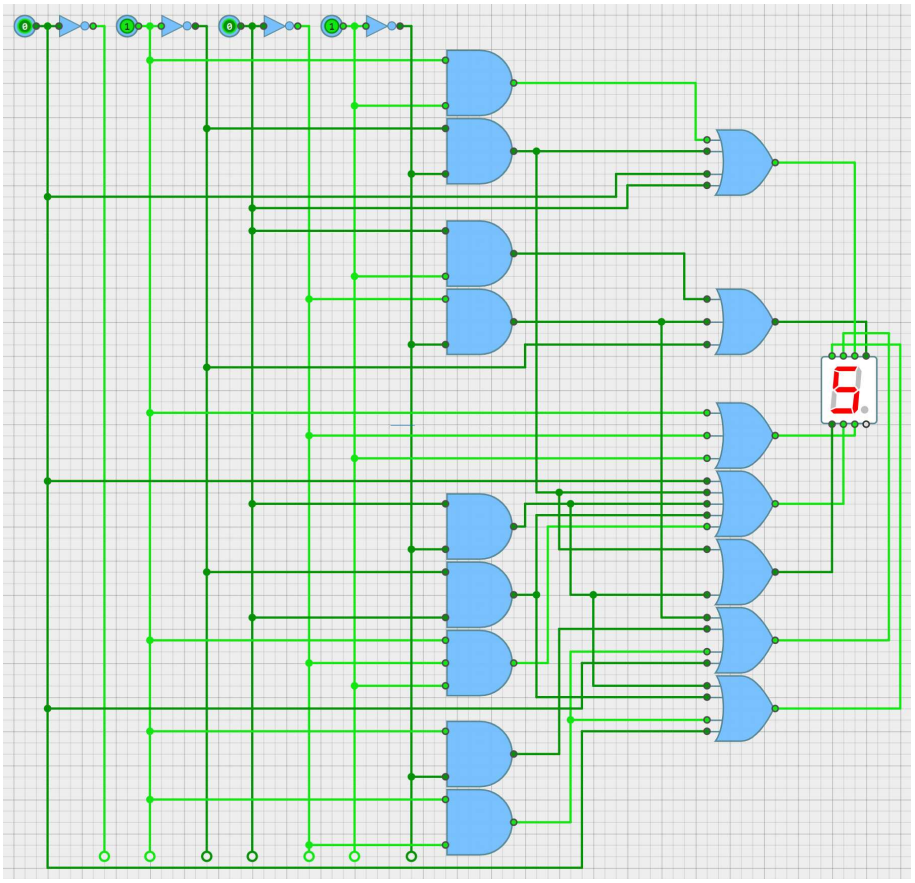
(a)



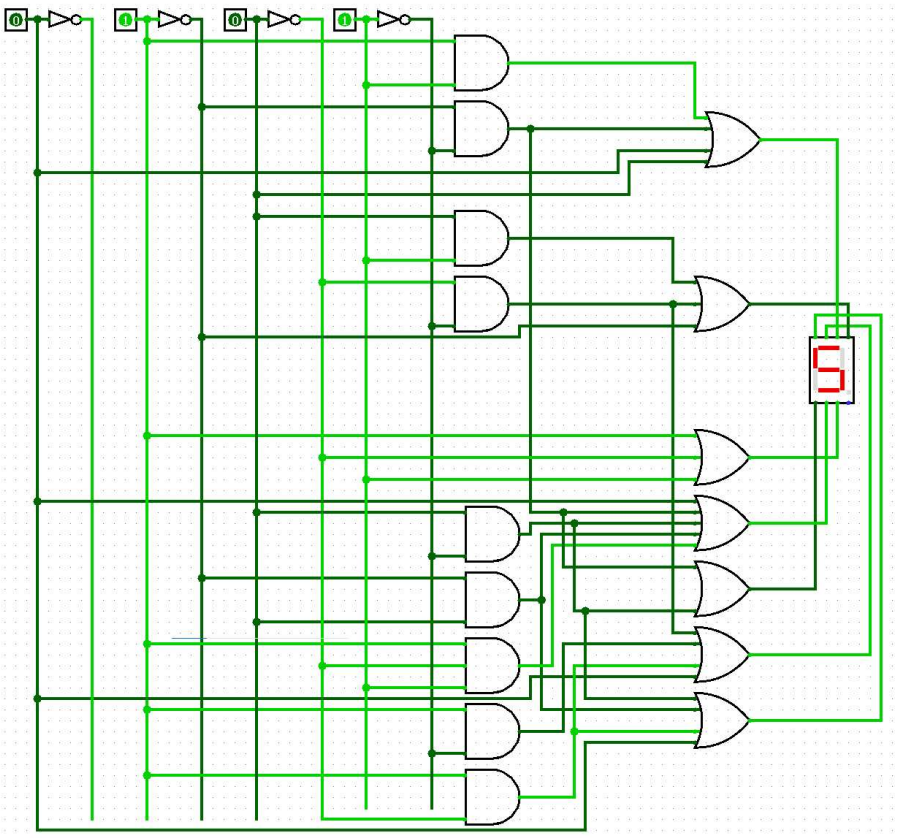
(b)

Figura 59 – Somador de 2 *bits*

[1]Disponível em: <https://renato-bohler.github.io/logossim/?example=2%20bit%20adder>
[2]Disponível em: <https://renato-bohler.github.io/logossim/?example=Binary%20to%20SSD%20decoder>



(a)

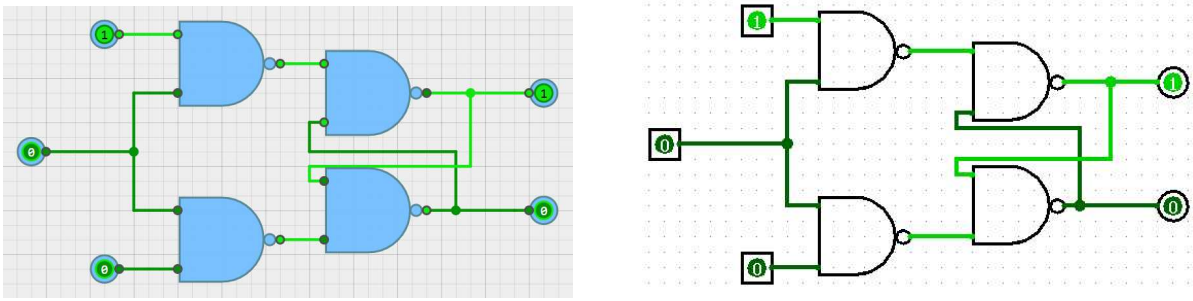


(b)

Figura 60 – Decodificador binário-SSD

4.1.2 Circuitos sequenciais

O *latch* RS síncrono[3] da Figura 61 foi um dos circuitos combinacionais validados no Logossim. Vários outros tipos de *latches* e *flip-flops* foram validados, e todos apresentaram o mesmo comportamento tanto no Logisim quanto no Logossim.



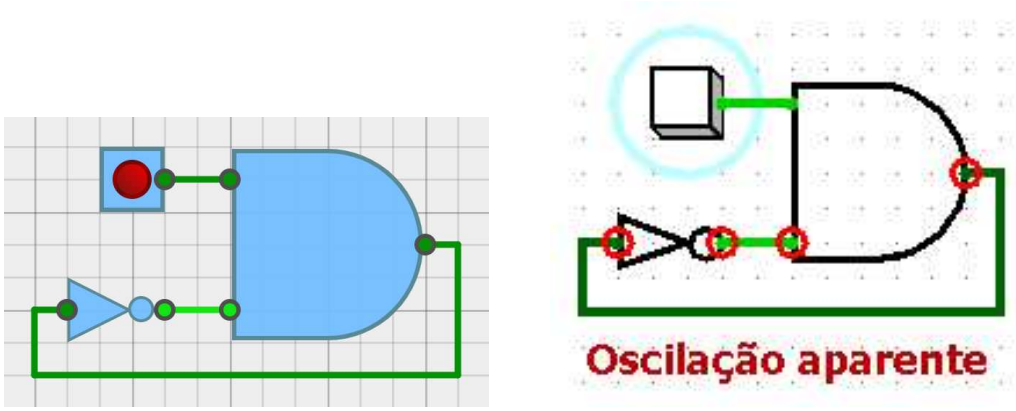
(a) (b)

Figura 61 – *Latch* RS síncrono

4.1.3 Circuitos oscilantes

Alguns circuitos com elos de realimentação, em alguns casos, podem nunca estabilizar. Tome por exemplo o circuito da Figura 62. Inicialmente, o circuito encontra-se numa condição estável. Quando o botão é ativado, no entanto, o valor de saída do circuito oscila entre 0 e 1.

Este comportamento ocorre em ambos os *softwares*. O Logisim, no entanto, identifica a oscilação e marca as portas envolvidas neste ciclo infinito. O Logossim não realiza esta identificação, mas continua a funcionar normalmente após um certo período de tempo.



(a) (b)

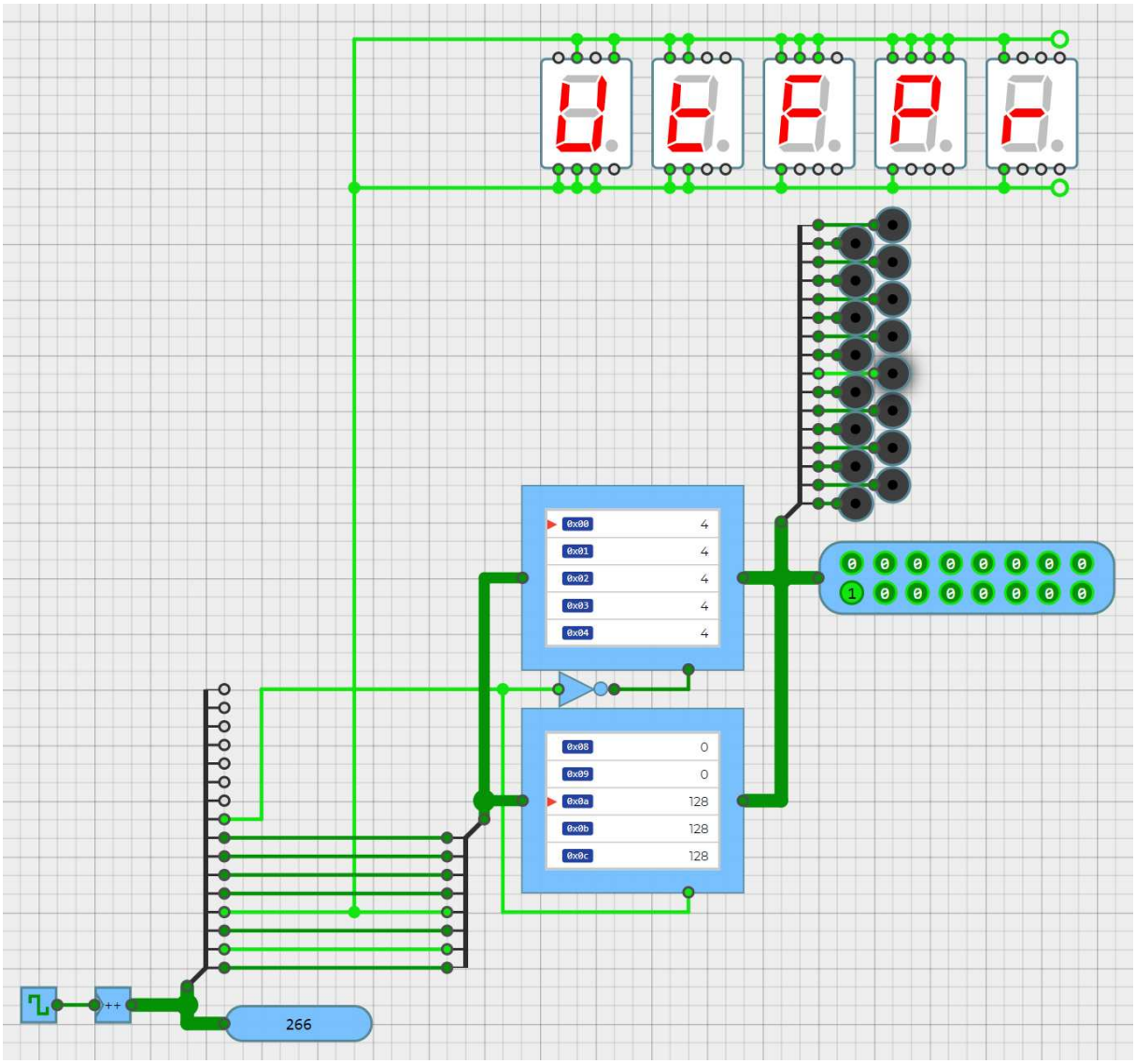
Figura 62 – Circuito oscilante

[3]Disponível em: <https://renato-bohler.github.io/logossim/?example=Synchronous%20RS%20flip-flop>

4.1.4 Circuito musical

Para testar diferentes tipos de componentes em circuitos mais complexos, alguns circuitos como o circuito musical[4] da Figura 63 foram construídos. Este circuito é composto 16 *buzzers* configurados com diferentes frequências e por duas memórias ROM, cujo conteúdo representa quais *buzzers* ativar em qual momento. O resultado é uma melodia de *The Imperial March*, da franquia *Star Wars*.

Figura 63 – Circuito musical



Fonte: Autoria própria

[4]Disponível em: <https://renato-bohler.github.io/logossim/?example=Darth+Vader>

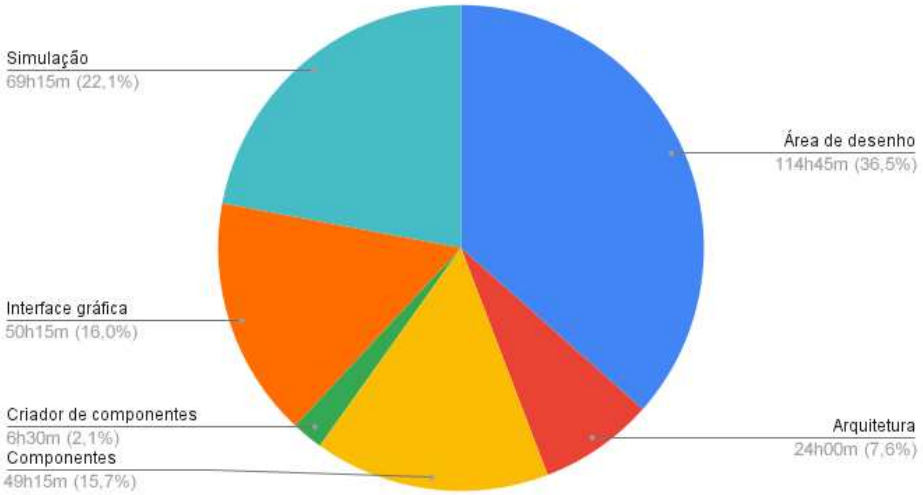
4.2 Atividades

Todas atividades de implementação do projeto do Logossim foram cadastradas, categorizadas e tiveram o seu tempo de execução medidos. No total, 47 atividades foram cadastradas em 5 grupos distintos:

- **arquitetura:** atividades que envolviam a inicialização, configuração e publicação da aplicação;
- **interface gráfica:** atividades que envolviam a construção e estilização da interface gráfica da aplicação (interface principal, caixas de diálogo e menus de contexto);
- **área de desenho:** atividades que compreendiam implementações, configurações e customizações da biblioteca React Diagrams para adaptação ao Logossim (bifurcações em fios, funcionalidades de copiar e colar, desfazer e refazer, etc.);
- **simulação:** compreende a implementação do motor e algoritmo de simulação;
- **criador de componentes:** implementação da ferramenta utilitária para criação de componentes.

No total, foram despendidas cerca de 314 horas de implementação no projeto Logossim. A Figura 64 exibe um gráfico de setores que exibe a quantidade e percentual de tempo despendido em cada um destes grupos.

Figura 64 – Quantidade de horas despendidas em cada grupo de atividades



Fonte: Autoria própria

A implementação de elementos gráficos (interface gráfica e área de desenho) compõe mais da metade das horas investidas no projeto. O desenvolvimento dos componentes e da simulação correspondem a 37,8% das horas. O restante dos valores está distribuído entre atividades de arquitetura e da implementação do criador de componentes.

5 CONCLUSÃO

Programas de simulação de lógica digital tendem a ser de difícil acesso, seja porque necessitam de instalação, pagamento ou porque são excessivamente complexos. O Logossim foi desenvolvido pensando em preencher esta lacuna, oferecendo uma aplicação fácil de ser acessada e compreendida: com poucos cliques, um usuário leigo pode começar a explorar o mundo de circuitos digitais.

O sistema desenvolvido apresentou uma boa performance em diferentes dispositivos, e possui funcionalidades básicas praticamente idênticas ao Logisim, sua maior inspiração. Alguns pontos, no entanto, ainda poderiam ser aprimorados, como por exemplo a falta de suporte para dispositivos móveis ou para a internacionalização da aplicação, que está atualmente disponível somente em inglês.

As restrições impostas pela pandemia do COVID-19 dificultaram a execução de um teste de usabilidade da aplicação, que poderia ter sido realizado com alunos da própria instituição. Um teste de usabilidade poderia apontar outros possíveis pontos de melhoria no software.

Por ser um projeto de código aberto e possuir uma ferramenta que auxilia na criação de componentes, é possível que o Logossim ganhe tração na comunidade de desenvolvedores e continue a ser aprimorado. Na realidade, o Logossim já foi mencionado no Hackaday[1], um dos maiores *sites* especializados em projetos de *hardware* na internet, e o repositório do projeto no GitHub já conta com 70 estrelas[2] de desenvolvedores de todas as regiões do mundo.

5.1 TRABALHOS FUTUROS

Uma série de funcionalidades presentes no Logisim poderiam ser portadas ao Logossim, tais como:

- **análise combinacional[3]:** criar uma ferramenta para geração de circuitos combinacionais, onde o usuário lista entradas e saídas, define o comportamento do circuito através de expressões ou de uma tabela verdade, e o programa gera um circuito que atende às especificações;
- **subcircuitos[4]:** permitir que o usuário crie subcircuitos que abstraiam parte da lógica, para serem utilizados num circuito maior;
- **identificar oscilações[5]:** marcar quais portas participam de uma oscilação assim que uma ocorrer, para permitir que o usuário identifique e corrija problemas no seu desenho;
- **bancada de testes[6]:** permitir executar testes sobre circuitos sem a necessidade de

[1] <https://hackaday.com/2020/11/02/ttl-simulator-in-javascript>
[2] <https://github.com/renato-bohler/logossim/stargazers>
[3] <http://www.cburch.com/logisim/docs/2.7/pt/html/guide/analyze/index.html>
[4] <http://www.cburch.com/logisim/docs/2.7/pt/html/guide/subcirc/index.html>
[5] <http://www.cburch.com/logisim/docs/2.7/pt/html/guide/prop/oscillate.html>
[6] <http://www.cburch.com/logisim/docs/2.7/pt/html/guide/verify/index.html>

interagir com a interface, definindo valores para entrada e os valores de saída esperados. Esta ferramenta seria particularmente útil para facilitar que instrutores validem as soluções feitas por alunos.

- **atribuir nome a componentes:** criar uma configuração disponível a todos os componentes, que permita atribuir a qualquer componente um nome que rotula cada componente no circuito na área de desenho.

Além destas, algumas outras funcionalidades e melhorias poderiam ser adicionadas ao projeto:

- **suporte a dispositivos móveis:** implementar melhorias na interface gráfica e área de desenho para permitir que usuários de dispositivos móveis tenham acesso a todas as funcionalidades do Logossim. Atualmente, um usuário em dispositivo móvel não consegue mover componentes ou criar conexões entre eles, estando basicamente limitado a acessar exemplos prontos e interagir com eles;
- **suporte a internacionalização:** adicionar ao projeto (interface gráfica e componentes) uma estratégia de internacionalização, para permitir que o Logossim seja disponibilizado em outros idiomas;
- **compartilhamento de circuitos:** implementar uma funcionalidade de compartilhamento de circuitos. Dado que o Logossim é uma aplicação *web*, seria útil permitir que o usuário gere uma URL que carregue seu circuito, sem a necessidade de baixar arquivos.

Referências

ARGUELLES, C.; IVANKOVIć, M.; BENDER, A. Code coverage best practices. 2020. Disponível em: <https://testing.googleblog.com/2020/08/code-coverage-best-practices.html>. Acesso em: 15 de junho de 2021. Citado na página 47.

BANKS, A.; PORCELLO, E. **Learning React**. 1. ed. [S.l.]: O'Reilly, 2017. Citado na página 11.

BISSELL, C. Historical perspectives - the moniac a hydromechanical analog computer of the 1950s. 2007. Disponível em: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4064850>. Acesso em: 6 de maio de 2021. Citado na página 7.

BOOLE, G. **An Investigation of the Laws of Thought**. 1. ed. Londres: Walton and Maberly, 1854. Citado na página 3.

BRITO, G.; TERRA, R.; VALENTE, M. T. Monorepos: A multivocal literature review. 2018. Disponível em: <https://arxiv.org/pdf/1810.09477.pdf>. Acesso em: 30 de maio de 2021. Citado na página 15.

BURCH, C. **Schools using Logisim**. 2013. Disponível em: <http://www.cburch.com/logisim/usage.html>. Acesso em: 27 de abril de 2021. Citado na página 1.

CREATE-REACT-APP. **Set up a modern web app by running one command**. 2021. Disponível em: <https://github.com/facebook/create-react-app>. Acesso em: 1 de julho de 2021. Citado na página 18.

DIAGRAMS, B. R. **A collection of lightweight React components and hooks to build diagrams with ease**. 2021. Disponível em: <https://github.com/beautifulinteractions/beautiful-react-diagrams>. Acesso em: 1 de julho de 2021. Citado na página 11.

DIAGRAMS, R. **A super simple, no-nonsense diagramming library written in react that just works**. 2021. Disponível em: <https://github.com/projectstorm/react-diagrams>. Acesso em: 30 de junho de 2021. Citado na página 11.

FLOYD, T. L. **Sistemas Digitais**. 9. ed. [S.l.]: Bookman, 2007. Citado na página 8.

FOWLER, M. Continuous integration. 2000. Disponível em: <https://www.martinfowler.com/articles/continuousIntegration.html>. Acesso em: 25 de maio de 2021. Citado na página 15.

FRENCH, R. S. et al. A general method for compiling event-driven simulations. 1995. Disponível em: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.7843&rep=rep1&type=pdf>. Acesso em: 16 de maio de 2021. Citado na página 9.

GREEN, I. **Web Workers**. 1. ed. [S.l.]: O'Reilly, 2012. Citado na página 13.

HOTA, A. K.; PRABHU, D. M. Node.js: Lightweight, event driven io web development. 2014. Disponível em: <https://informaticsweb.nic.in/sites/default/files/node.js_.pdf>. Acesso em: 17 de maio de 2021. Citado na página 10.

IDOETA, I. V. **Elementos de Eletrônica Digital**. 40. ed. São Paulo: Editora Érica, 2008. Citado na página 6.

JEST. **Delightful JavaScript Testing**. 2021. Disponível em: <https://jestjs.io>. Acesso em: 1 de julho de 2021. Citado na página 14.

JIRA. **Software para acompanhamento de itens e projetos**. 2021. Disponível em: <https://www.atlassian.com/br/software/jira>. Acesso em: 30 de junho de 2021. Citado na página 10.

JOYRIDE, R. **Create guided tours in your apps**. 2021. Disponível em: <https://github.com/gilbarbara/react-joyride>. Acesso em: 30 de junho de 2021. Citado 2 vezes nas páginas 12 e 13.

LEIBNIZ, G. W. von. Explication de l'arithmétique binaire. 1703. Citado na página 3.

MARDAN, A. **React Quickly**. 1. ed. [S.l.]: Manning, 2017. Citado na página 11.

MAURER, P. M. The inversion algorithm for digital simulation. 1997. Disponível em: <https://baylor-ir.tdl.org/bitstream/handle/2104/5450/InversionBaylor.pdf?sequence=1&isAllowed=y>. Acesso em: 16 de maio de 2021. Citado na página 8.

MICZO, A. **Digital Logic Testing and Simulation**. 2. ed. [S.l.]: Wiley-Interscience, 2007. Citado 2 vezes nas páginas 8 e 55.

MOZILLA. **Documentação da interface AudioContext**. 2021. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/API/AudioContext>. Acesso em: 1 de julho de 2021. Citado na página 43.

MOZILLA. **Documentação do Local Storage**. 2021. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/API/Window/localStorage>. Acesso em: 1 de julho de 2021. Citado na página 30.

MOZILLA. **Documentação do método setInterval**. 2021. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/API/WindowOrWorkerGlobalScope/setInterval>. Acesso em: 1 de julho de 2021. Citado na página 42.

M&SCO. **DoD Modeling and Simulation (M&S) Glossary**. [S.l.], 1998. 136,157 p. Disponível em: <https://apps.dtic.mil/dtic/tr/fulltext/u2/a349800.pdf>. Acesso em: 6 de maio de 2021. Citado na página 7.

NODE.JS. **Fast, Reliable and Secure Dependency Management**. 2021. Disponível em: <https://nodejs.org>. Acesso em: 1 de julho de 2021. Citado na página 16.

PLOP. **Consistency Made Simple**. 2021. Disponível em: <https://plopjs.com>. Acesso em: 1 de julho de 2021. Citado na página 14.

RIEPE, M. A. et al. Ravel-xl: A hardware accelerator for assigned-delay compiled-code logic gate simulation. 1994. Disponível em: <https://www.eecs.umich.edu/techreports/cse/94/CSE-TR-202-94.pdf>. Acesso em: 16 de maio de 2021. Citado na página 8.

ROSSI, G. et al. **Web Engineering: Modelling and Implementing Web Applications**. 2. ed. Londres: Springer, 2007. Citado na página 2.

SALEH, H. **JavaScript Unit Testing**. 1. ed. [S.l.]: Packt Publishing, 2013. Citado na página 14.

SARCAR, V. **Design Patterns in C#**. 1. ed. [S.l.]: Apress, 2018. Citado na página 27.

SHANNON, C. **A Symbolic Analysis of Relay and Switching Circuits**. Dissertação (Mestrado) — Massachusetts Institute of Technology, Massachusetts, 1937. Citado na página 6.

SMITH, R. D. Simulation article. 1998. Disponível em: <http://www.modelbenders.com/encyclopedia/encyclopedia.html>. Acesso em: 6 de maio de 2021. Citado na página 7.

STåL, F. Interactive user onboarding and its effect on activation rates. 2020. Disponível em: <https://www.diva-portal.org/smash/get/diva2:1471418/FULLTEXT01.pdf>. Acesso em: 10 de junho de 2021. Citado na página 30.

WANG, J.; TROPPER, C. Compiled code in distributed logic simulation. 2006. Disponível em: <https://www.cs.mcgill.ca/~carl/cidvs.pdf>. Acesso em: 16 de maio de 2021. Citado na página 8.

WOHLGETHAN, E. Supporting web development decisions by comparing three major javascript frameworks: Angular, react and vue.js. 2018. Disponível em: <https://reposit.haw-hamburg.de/bitstream/20.500.12738/8417/1/BA_Wohlgethan_2176410.pdf>. Acesso em: 1 de julho de 2021. Citado na página 10.

YARN. **Fast, Reliable and Secure Dependency Management**. 2021. Disponível em: <https://classic.yarnpkg.com/en>. Acesso em: 1 de julho de 2021. Citado na página 16.

ZHAO, Y. et al. The impact of continuous integration on other software development practices: A large-scale empirical study. 2017. Disponível em: <https://web.cs.ucdavis.edu/~filkov/papers/CI_adoption.pdf>. Acesso em: 25 de maio de 2021. Citado na página 15.

Apêndices

APÊNDICE A – Componentes do Logossim

A.1 Pino de entrada

Descrição: componente básico para entrada do usuário.
Categoria: entrada e saída.
Tipo: componente ativo.
Configurações: ver Quadro 7.

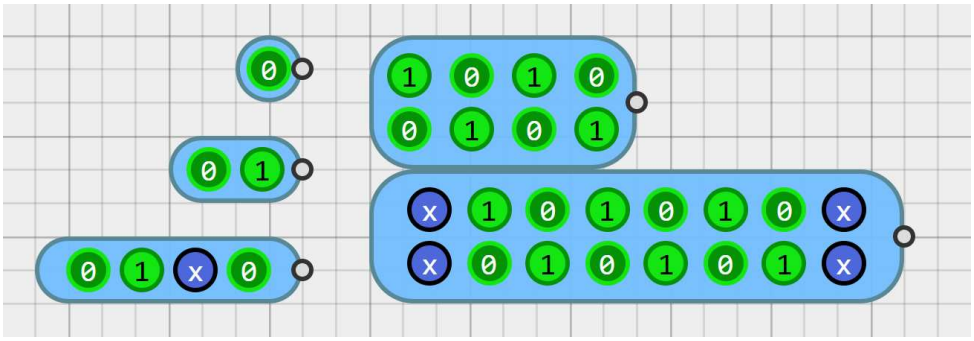
Quadro 7 – Possíveis configurações para o componente pino de entrada

Nome	**Tipo**	**Propósito**
Data bits	Seletor	Permite definir a largura de *bits* do componente
Three state	Seletor	Configura se o usuário pode emitir valores flutuantes

Fonte: autoria própria

Aparência: ver Figura 65.

Figura 65 – Componente pino de entrada com diferentes configurações



Fonte: autoria própria

Comportamento: responde à ação de clique do usuário em um dos pinos, alternando o valor atual entre 0 e 1 e emitindo o novo valor.

A.2 Pino de saída

Descrição: componente que se liga a um fio e exibe o valor que ele está propagando.
Categoria: entrada e saída.
Tipo: componente passivo.
Configurações: ver Quadro 8.

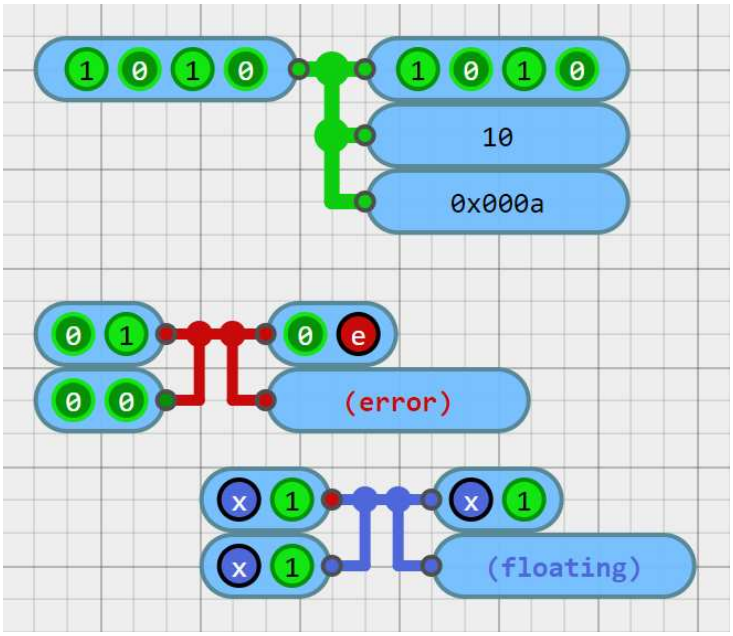
Quadro 8 – Possíveis configurações para o componente pino de saída

Nome	**Tipo**	**Propósito**
Data bits	Seletor	Permite definir a largura de *bits* do componente
Output format	Seletor	Configura o formato de exibição do valor lido, podendo ser *bit* a *bit*, decimal ou hexadecimal.

Fonte: autoria própria

Aparência: ver Figura 66.

Figura 66 – Componente pino de saída com diferentes configurações



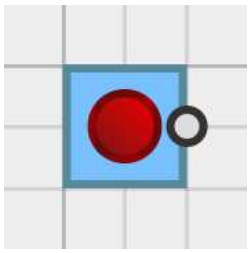
Fonte: autoria própria

Comportamento: exibe o valor do fio conectado à sua porta de entrada de acordo com a configuração aplicada.

A.3 Botão

Descrição: um simples botão.
Categoria: entrada e saída.
Tipo: componente ativo.
Configurações: não há.
Aparência: ver Figura 67.

Figura 67 – Componente botão



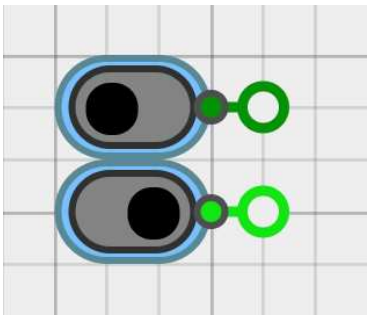
Fonte: autoria própria

Comportamento: emite o valor 0 ao iniciar a simulação. Emite 1 enquanto o usuário estiver pressionando o botão com o *mouse*, e 0 caso contrário.

A.4 Interruptor

Descrição: um simples interruptor.
Categoria: entrada e saída.
Tipo: componente ativo.
Configurações: não há.
Aparência: ver Figura 68.

Figura 68 – Componente interruptor



Fonte: autoria própria

Comportamento: emite o valor 0 ao iniciar a simulação. Alterna e emite o valor entre 0 e 1 como resposta ao clique do usuário.

A.5 Clock

Descrição: componente que gera um sinal de sincronização (*clock signal*).

Categoria: entrada e saída.

Tipo: componente ativo.

Configurações: ver Quadro 9.

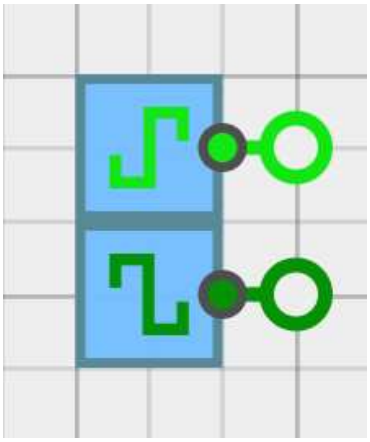
Quadro 9 – Possíveis configurações para o componente *clock*

Nome	**Tipo**	**Propósito**
Frequency (in Hz)	Numérico	Define a frequência da oscilação, em hertz
High duration (in cycles)	Numérico	Define quantos ciclos o sinal gerado deve permanecer com o valor 1
Low duration (in cycles)	Numérico	Define quantos ciclos o sinal gerado deve permanecer com o valor 0

Fonte: autoria própria

Aparência: ver Figura 69.

Figura 69 – Componente *clock*



Fonte: autoria própria

Comportamento: automaticamente alterna entre os valores 0 e 1 com base nas suas configurações.

A.6 LED

Descrição: componente que representa um LED.

Categoria: entrada e saída.

Tipo: componente passivo.

Configurações: ver Quadro 10.

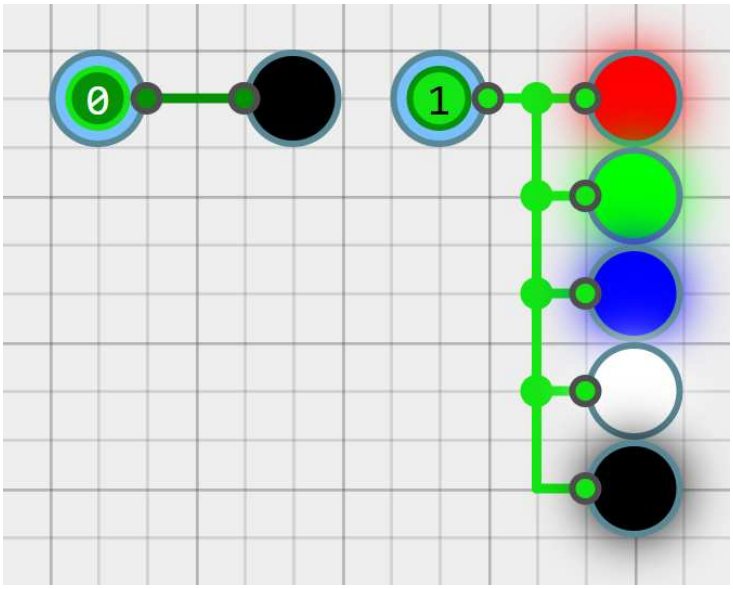
Quadro 10 – Possíveis configurações para o componente LED

Nome	**Tipo**	**Propósito**
Active when	Seletor	Define se o LED é ativo em alta (valor 1) ou baixa (valor 0)
On color	Seletor	Configura a cor do LED quando ativo.
Off color	Seletor	Configura a cor do LED quando inativo.

Fonte: autoria própria

Aparência: ver Figura 70.

Figura 70 – Componente LED em diferentes cores



Fonte: autoria própria

Comportamento: exibe um efeito de iluminação quando ativo, com as cores definidas pela configuração.

A.7 SSD

Descrição: componente que representa um *display* de sete segmentos (SSD).
Categoria: entrada e saída.
Tipo: componente passivo.
Configurações: ver Quadro 11.

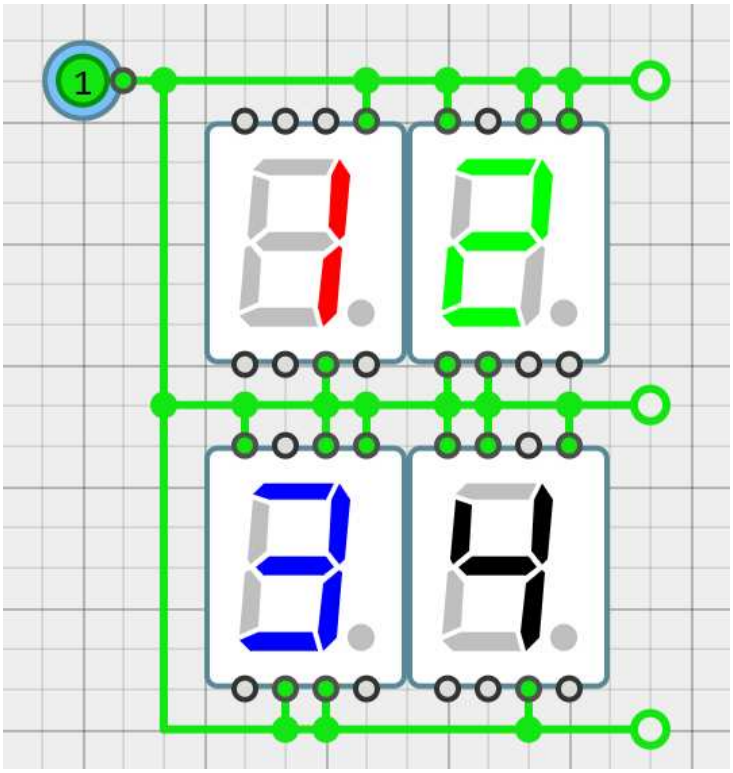
Quadro 11 – Possíveis configurações para o componente SSD

Nome	**Tipo**	**Propósito**
Active when	Seletor	Define se os segmentos do *display* estão ativos em alta (valor 1) ou baixa (valor 0)
On color	Seletor	Configura a cor dos segmentos quando ativos.
Off color	Seletor	Configura a cor dos segmentos quando inativos.

Fonte: autoria própria

Aparência: ver Figura 71.

Figura 71 – Componente SSD em diferentes cores



Fonte: autoria própria

Comportamento: ilumina os segmentos ativos conforme sua configuração.

A.8 Buzzer

Descrição: componente que emite sinais sonoros.

Categoria: entrada e saída.

Tipo: componente passivo.

Configurações: ver Quadro 12.

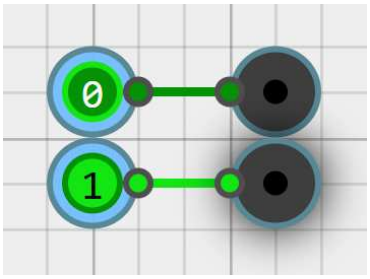
Quadro 12 – Possíveis configurações para o componente *buzzer*

Nome	**Tipo**	**Propósito**
Sound frequency (in hertz)	Numérico	Define a frequência do sinal sonoro a emitir, em hertz
Waveform	Seletor	Define o formato de onda a ser emitido (senoidal, quadrada, dente de serra ou triangular).

Fonte: autoria própria

Aparência: ver Figura 72.

Figura 72 – Componente *buzzer*



Fonte: autoria própria

Comportamento: emite um sinal sonoro no formato e frequência configurados quando ativo.

A.9 Porta lógica AND

Descrição: porta lógica que realiza a operação "E".

Categoria: portas lógicas.

Tipo: componente passivo.

Configurações: ver Quadro 13.

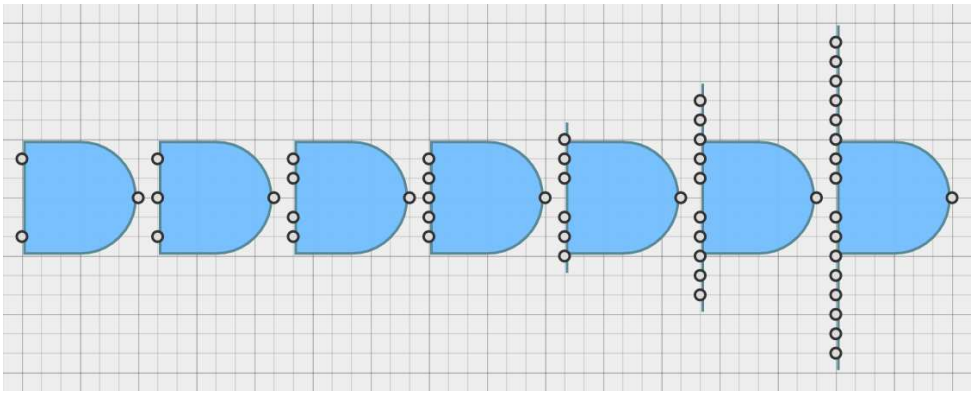
Quadro 13 – Possíveis configurações para a porta lógica AND

Nome	Tipo	Propósito
Number of input ports	Numérico	Define o número de portas de entrada do componente (de 2 a 16)
Data bits	Seletor	Permite definir a largura de *bits* que o componente atua

Fonte: autoria própria

Aparência: ver Figura 73.

Figura 73 – Porta lógica AND com diferentes números de entrada



Fonte: autoria própria

Comportamento: aplica os valores da tabela verdade 2. Caso exista mais do que 2 entradas, a operação é realizada sobre os dois primeiros *bits*, e o resultado é aplicado aos próximos sucessivamente. Para entradas com vários *bits*, esta mesma operação é realizada *bit* a *bit*.

Tabela 2 – Tabela verdade da porta lógica AND

	0	1	x	e
0	0	0	0	0
1	0	1	e	e
x	0	e	e	e
e	0	e	e	e

Fonte: autoria própria

A.10 Porta lógica NAND

Descrição: porta lógica que realiza a operação "NE" (negação de "E").
Categoria: portas lógicas.
Tipo: componente passivo.
Configurações: ver Quadro 14.

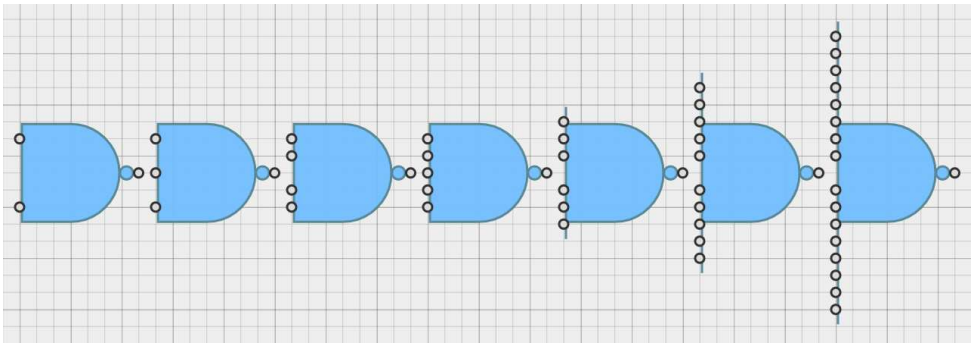
Quadro 14 – Possíveis configurações para a porta lógica NAND

Nome	**Tipo**	**Propósito**
Number of input ports	Numérico	Define o número de portas de entrada do componente (de 2 a 16)
Data bits	Seletor	Permite definir a largura de *bits* que o componente atua

Fonte: autoria própria

Aparência: ver Figura 74.

Figura 74 – Porta lógica NAND com diferentes números de entrada



Fonte: autoria própria

Comportamento: aplica os valores da tabela verdade 3. Caso exista mais do que 2 entradas, a operação é realizada sobre os dois primeiros *bits*, e o resultado é aplicado aos próximos sucessivamente. Para entradas com vários *bits*, esta mesma operação é realizada *bit* a *bit*.

Tabela 3 – Tabela verdade da porta lógica NAND

	0	1	x	e
0	1	1	1	1
1	1	0	e	e
x	1	e	e	e
e	1	e	e	e

Fonte: autoria própria

A.11 Porta lógica OR

Descrição: porta lógica que realiza a operação "OU".
Categoria: portas lógicas.
Tipo: componente passivo.
Configurações: ver Quadro 15.

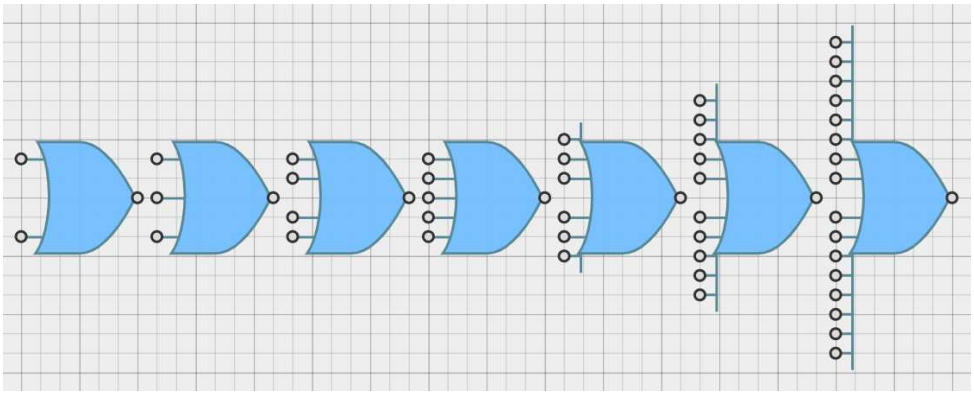
Quadro 15 – Possíveis configurações para a porta lógica OR

Nome	Tipo	Propósito
Number of input ports	Numérico	Define o número de portas de entrada do componente (de 2 a 16)
Data bits	Seletor	Permite definir a largura de *bits* que o componente atua

Fonte: autoria própria

Aparência: ver Figura 75.

Figura 75 – Porta lógica OR com diferentes números de entrada



Fonte: autoria própria

Comportamento: aplica os valores da tabela verdade 4. Caso exista mais do que 2 entradas, a operação é realizada sobre os dois primeiros *bits*, e o resultado é aplicado aos próximos sucessivamente. Para entradas com vários *bits*, esta mesma operação é realizada *bit* a *bit*.

Tabela 4 – Tabela verdade da porta lógica OR

	0	1	x	e
0	0	1	e	e
1	1	1	1	1
x	e	1	e	e
e	e	1	e	e

Fonte: autoria própria

A.12 Porta lógica NOR

Descrição: porta lógica que realiza a operação "NOU" (negação de "OU").
Categoria: portas lógicas.
Tipo: componente passivo.
Configurações: ver Quadro 16.

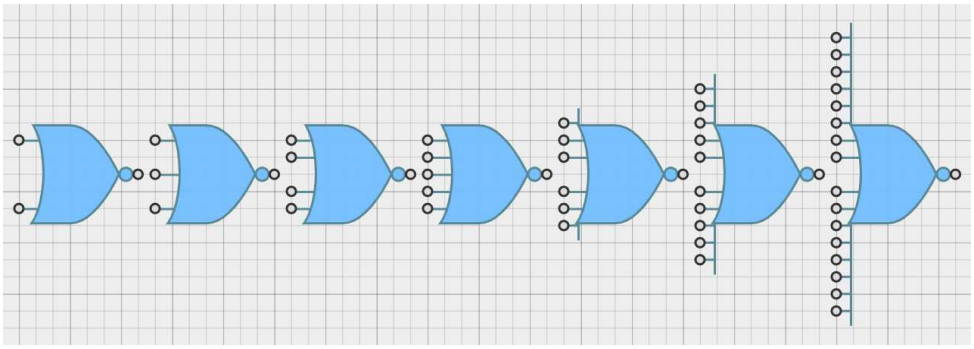
Quadro 16 – Possíveis configurações para a porta lógica NOR

Nome	**Tipo**	**Propósito**
Number of input ports	Numérico	Define o número de portas de entrada do componente (de 2 a 16)
Data bits	Seletor	Permite definir a largura de *bits* que o componente atua

Fonte: autoria própria

Aparência: ver Figura 76.

Figura 76 – Porta lógica NOR com diferentes números de entrada



Fonte: autoria própria

Comportamento: aplica os valores da tabela verdade 5. Caso exista mais do que 2 entradas, a operação é realizada sobre os dois primeiros *bits*, e o resultado é aplicado aos próximos sucessivamente. Para entradas com vários *bits*, esta mesma operação é realizada *bit* a *bit*.

Tabela 5 – Tabela verdade da porta lógica NOR

	0	1	x	e
0	1	0	e	e
1	0	0	0	0
x	e	0	e	e
e	e	0	e	e

Fonte: autoria própria

A.13 Porta lógica XOR

Descrição: porta lógica que realiza a operação "OU EXCLUSIVO".
Categoria: portas lógicas.
Tipo: componente passivo.
Configurações: ver Quadro 17.

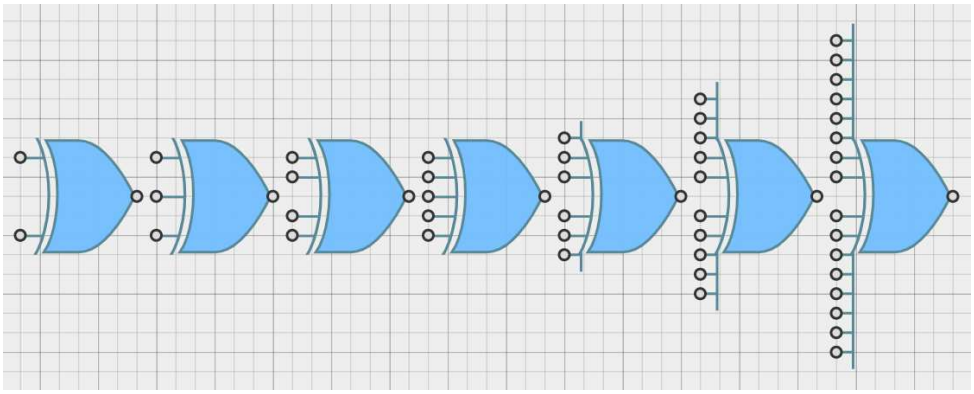
Quadro 17 – Possíveis configurações para a porta lógica XOR

Nome	Tipo	Propósito
Number of input ports	Numérico	Define o número de portas de entrada do componente (de 2 a 16)
Data bits	Seletor	Permite definir a largura de *bits* que o componente atua
Multiple input behavior	Seletor	Define o modo de operação da porta para 3 ou mais entradas (padrão ou detector de imparidade).

Fonte: autoria própria

Aparência: ver Figura 77.

Figura 77 – Porta lógica XOR com diferentes números de entrada



Fonte: autoria própria

Comportamento: quando configurado no modo padrão, a porta lógica retornará 1 somente quando apenas um dos *bits* assumir o valor 1, e retornará 0 caso contrário. Quando configurado no modo de detecção de imparidade, a porta lógica retornará 1 quando existir um número ímpar de valores 1 na entrada, 0 caso contrário. Caso qualquer um dos *bits* de entrada seja o valor flutuante ou de erro, a saída assumirá o valor de erro. Para entradas com vários *bits*, esta mesma operação é realizada *bit* a *bit*.

A.14 Porta lógica XNOR

Descrição: porta lógica que realiza a operação "NOU EXCLUSIVO" (negação de "OU EXCLUSIVO").

Categoria: portas lógicas.

Tipo: componente passivo.

Configurações: ver Quadro 18.

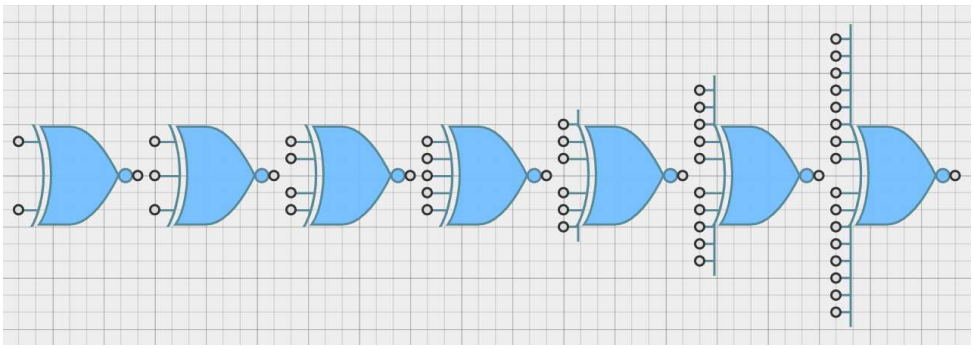
Quadro 18 – Possíveis configurações para a porta lógica XNOR

Nome	Tipo	Propósito
Number of input ports	Numérico	Define o número de portas de entrada do componente (de 2 a 16)
Data bits	Seletor	Permite definir a largura de *bits* que o componente atua
Multiple input behavior	Seletor	Define o modo de operação da porta para 3 ou mais entradas (padrão ou detector de paridade).

Fonte: autoria própria

Aparência: ver Figura 78.

Figura 78 – Porta lógica XNOR com diferentes números de entrada



Fonte: autoria própria

Comportamento: quando configurado no modo padrão, a porta lógica retornará 0 somente quando apenas um dos *bits* assumir o valor 1, e retornará 1 caso contrário. Quando configurado no modo de detecção de paridade, a porta lógica retornará 1 quando existir um número par de valores 1 na entrada, 0 caso contrário. Caso qualquer um dos *bits* de entrada seja o valor flutuante ou de erro, a saída assumirá o valor de erro. Para entradas com vários *bits*, esta mesma operação é realizada *bit* a *bit*.

A.15 Buffer

Descrição: este componente reflete o valor de entrada para a saída, sem modificações.
Categoria: portas lógicas.
Tipo: componente passivo.
Configurações: ver Quadro 19.

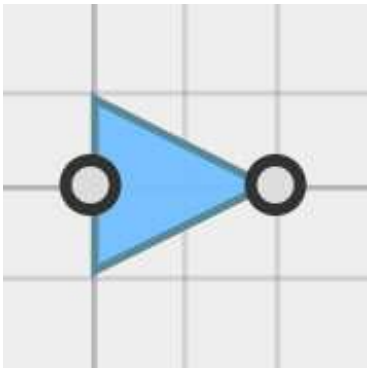
Quadro 19 – Possíveis configurações para o componente *buffer*

Nome	**Tipo**	**Propósito**
Data bits	Seletor	Permite definir a largura de *bits* que o componente atua

Fonte: autoria própria

Aparência: ver Figura 79.

Figura 79 – Componente *buffer*



Fonte: autoria própria

Comportamento: reflete a entrada para a saída, seja o valor de entrada numérico, flutuante ou de erro.

A.16 Inversora

Descrição: a porta inversora inverte (ou nega) o valor de entrada para a saída.
Categoria: portas lógicas.
Tipo: componente passivo.
Configurações: ver Quadro 20.

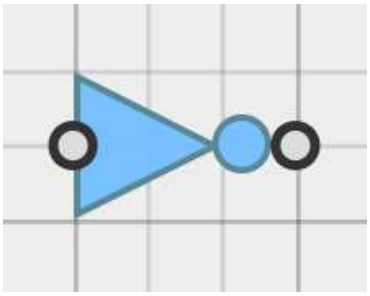
Quadro 20 – Possíveis configurações para a porta inversora

Nome	**Tipo**	**Propósito**
Data bits	Seletor	Permite definir a largura de *bits* que o componente atua

Fonte: autoria própria

Aparência: ver Figura 80.

Figura 80 – Porta inversora



Fonte: autoria própria

Comportamento: inverte os bits 0 e 1 da entrada para a saída. Caso a entrada seja o valor de erro, a saída será o valor de erro. Caso a entrada seja o valor flutuante, a saída será o valor flutuante. Para entradas com vários *bits*, esta mesma operação é realizada *bit* a *bit*.

A.17 Buffer controlado

Descrição: um *buffer* cuja saída é controlada pela porta de controle. Também conhecido como *tri-state buffer*.

Categoria: portas lógicas.

Tipo: componente passivo.

Configurações: ver Quadro 21.

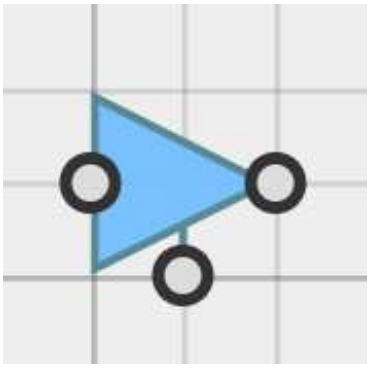
Quadro 21 – Possíveis configurações para o *buffer* controlado

Nome	**Tipo**	**Propósito**
Data bits	Seletor	Permite definir a largura de *bits* que o componente atua

Fonte: autoria própria

Aparência: ver Figura 81.

Figura 81 – *Buffer* controlado



Fonte: autoria própria

Comportamento: quando a porta de controle possui valor 1, reflete a entrada para a saída, seja o valor de entrada numérico, flutuante ou de erro. Quando a porta de controle possui valor 0, a saída assume o valor flutuante.

A.18 Inversora controlada

Descrição: uma porta inversora cuja saída é controlada pela porta de controle. Também conhecido como inversor *tri-state*.

Categoria: portas lógicas.

Tipo: componente passivo.

Configurações: ver Quadro 22.

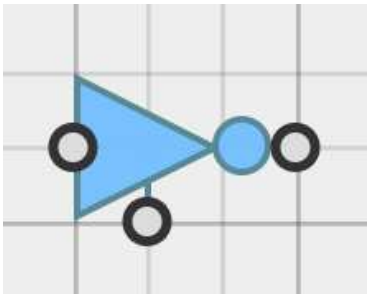
Quadro 22 – Possíveis configurações para a porta inversora controlada

Nome	**Tipo**	**Propósito**
Data bits	Seletor	Permite definir a largura de *bits* que o componente atua

Fonte: autoria própria

Aparência: ver Figura 82.

Figura 82 – Porta inversora controlada



Fonte: autoria própria

Comportamento: quando a porta de controle possui valor 1, inverte o valor de entrada da mesma maneira que a porta inversora. Quando a porta de controle possui valor 0, a saída assume o valor flutuante.

A.19 Divisor

Descrição: um componente que decompõe um fio de vários *bits*.
Categoria: cabeamento.
Tipo: componente passivo.
Configurações: ver Quadro 23.

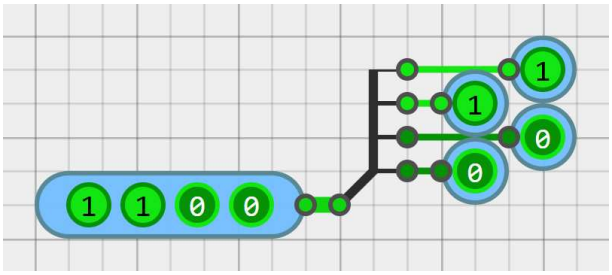
Quadro 23 – Possíveis configurações para o divisor

Nome	**Tipo**	**Propósito**
Input data bits	Seletor	Define a largura de *bits* da entrada

Fonte: autoria própria

Aparência: ver Figura 83.

Figura 83 – Componente divisor



Fonte: autoria própria

Comportamento: decompõe todos os *bits* de entrada em N saídas de 1 *bit*, sendo o *bit* mais significativo correspondente à saída mais acima.

A.20 Agregador

Descrição: um componente que compõe N fios de 1 *bit* num de N *bits*.
Categoria: cabeamento.
Tipo: componente passivo.
Configurações: ver Quadro 24.

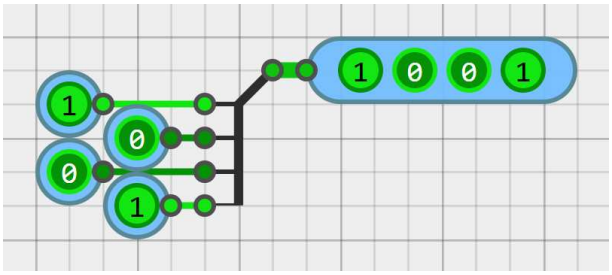
Quadro 24 – Possíveis configurações para o agregador

Nome	**Tipo**	**Propósito**
Output data bits	Seletor	Define a largura de *bits* da saída

Fonte: autoria própria

Aparência: ver Figura 84.

Figura 84 – Componente agregador



Fonte: autoria própria

Comportamento: agrega todas as N entradas de 1 *bit* numa saída de N *bits*, sendo o *bit* mais significativo correspondente à entrada mais acima.

A.21 Power

Descrição: emite um valor alto à saída.
Categoria: cabeamento.
Tipo: componente ativo.
Configurações: ver Quadro 25.

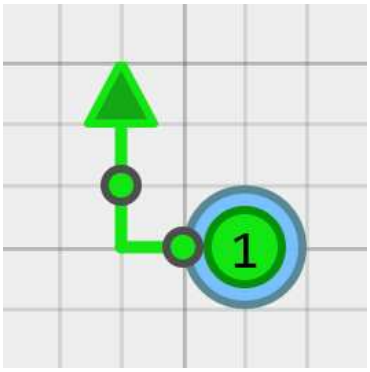
Quadro 25 – Possíveis configurações para o componente *power*

Nome	**Tipo**	**Propósito**
Data bits	Seletor	Permite definir a largura de *bits* do componente

Fonte: autoria própria

Aparência: ver Figura 85.

Figura 85 – Componente *power*



Fonte: autoria própria

Comportamento: emite o maior valor possível para a data largura de *bits* no início da simulação.

A.22 Ground

Descrição: emite um valor baixo à saída.
Categoria: cabeamento.
Tipo: componente ativo.
Configurações: ver Quadro 26.

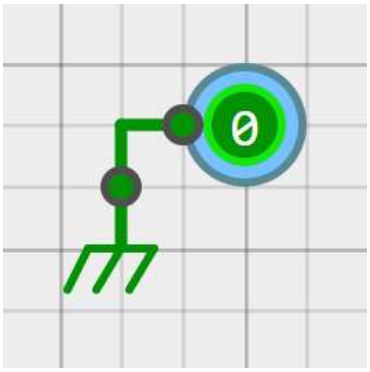
Quadro 26 – Possíveis configurações para o componente *ground*

Nome	**Tipo**	**Propósito**
Data bits	Seletor	Permite definir a largura de *bits* do componente

Fonte: autoria própria

Aparência: ver Figura 86.

Figura 86 – Componente *ground*



Fonte: autoria própria

Comportamento: emite 0 na saída ao iniciar a simulação.

A.23 ROM

Descrição: memória somente de leitura.
Categoria: memória.
Tipo: componente passivo.
Configurações: ver Quadro 27.

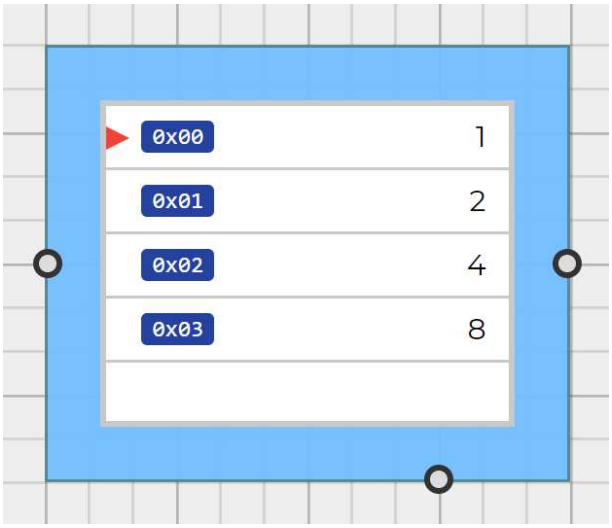
Quadro 27 – Possíveis configurações para o componente ROM

Nome	**Tipo**	**Propósito**
Data width	Seletor	Define a largura de *bits* de dados que cada endereço armazena
Address width	Seletor	Define a largura de *bits* destinada a endereçar a memória
Initial content	Binário	Define o conteúdo da memória

Fonte: autoria própria

Aparência: ver Figura 87.

Figura 87 – Componente ROM



Fonte: autoria própria

Comportamento: transfere à porta `data` (direita) o valor armazenado na célula de memória cujo endereço corresponde ao valor de entrada da porta `address` (esquerda). Caso a porta de entrada `select` (baixo) possua o valor 0, a saída assumirá o valor flutuante para todos seus *bits*. O endereço selecionado para leitura é indicado visualmente pela seta vermelha.

A.24 RAM

Descrição: memória de escrita e leitura.
Categoria: memória.
Tipo: componente passivo.
Configurações: ver Quadro 28.

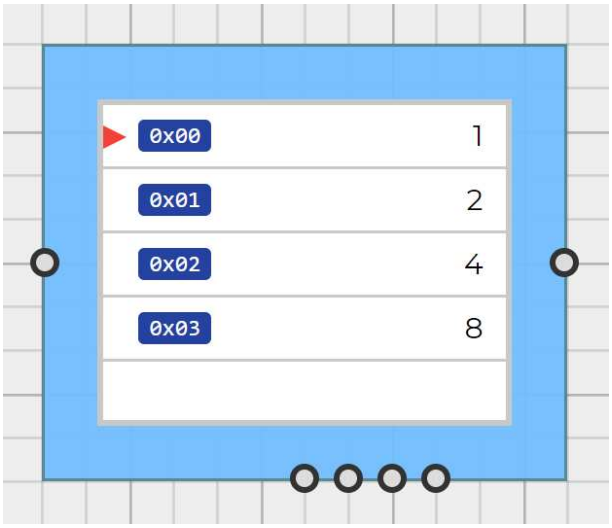
Quadro 28 – Possíveis configurações para o componente RAM

Nome	Tipo	Propósito
Data width	Seletor	Define a largura de *bits* de dados que cada endereço armazena
Address width	Seletor	Define a largura de *bits* destinada a endereçar a memória
Initial content	Binário	Define o conteúdo da memória

Fonte: autoria própria

Aparência: ver Figura 88.

Figura 88 – Componente RAM



Fonte: autoria própria

Comportamento: possui 6 portas: `address` (esquerda), `data` (direita), `clock`, `load`, `clear` e `select` (baixo, nesta ordem). Quando a porta `load` assume o valor 1, a memória está em modo de leitura e funciona exatamente como a memória ROM. Quando a porta `load` assume o valor 0, a memória está no modo de escrita. O modo de escrita é síncrono, sendo que a escrita na memória acontece no momento que a porta `clock` está em borda de subida. Neste momento, o conteúdo da memória do endereço `address` é substituído pelo valor da porta `data`. A porta `clear`, funciona assincronamente: quando em borda de subida, transforma o conteúdo de todos os endereços para 0. Caso a porta de entrada `select` (baixo) possua o

valor 0, a saída assumirá o valor flutuante para todos seus *bits*. O endereço selecionado para leitura/escrita é indicado visualmente pela seta vermelha.