UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

RAONE VICTOR DA CUNHA

COMPILAÇÃO DE CIRCUITOS DIGITAIS VIRTUAIS PARA APLICAÇÃO EM PLACA ARDUINO

PAU DOS FERROS
2021

RAONE VICTOR DA CUNHA

COMPILAÇÃO DE CIRCUITOS DIGITAIS VIRTUAIS PARA APLICAÇÃO EM PLACA ARDUINO

Monografia apresentada a Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Marco Diego Aurélio Mesquita, Prof. Me.

PAU DOS FERROS

2021

C972c Cunha, Raone Victor da.
Compilação de circuitos digitais virtuais para aplicação em placa Arduino / Raone Victor da Cunha. - 2021.
71 f. : il.

Orientador: Marco Diego Aurélio Mesquita.
Monografia (graduação) - Universidade Federal Rural do Semi-árido, Curso de Engenharia de Computação, 2021.

1. Circuitos Digitais. 2. Transpilador. 3. Simulador. 4. Placa Microcontrolada. I. Mesquita, Marco Diego Aurélio, orient. II. Título.

RAONE VICTOR DA CUNHA

COMPILAÇÃO DE CIRCUITOS DIGITAIS VIRTUAIS PARA APLICAÇÃO EM PLACA ARDUINO

Monografia apresentada a Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Engenharia de Computação.

Defendida em: 01 / 06 / 2021.

BANCA EXAMINADORA

Assinado Digitalmente

Marco Diego Aurélio Mesquita

Marco Diego Aurélio Mesquita, Prof. Me. (UFERSA)
Presidente

CECILIO MARTINS DE SOUSA NETO:06286737448

Assinado de forma digital por CECILIO MARTINS DE SOUSA NETO:06286737448
Dados: 2021.06.08 21:04:24 -03'00'

Cecílio Martins de Sousa Neto, Prof. Dr. (UFERSA)
Membro Examinador

FRANCISCO CARLOS GURGEL DA SILVA SEGUNDO:06062106444

Assinado de forma digital por FRANCISCO CARLOS GURGEL DA SILVA SEGUNDO:06062106444
Dados: 2021.06.08 20:11:39 -03'00'

Francisco Carlos Gurgel da Silva Segundo, Prof. Dr. (UFERSA)
Membro Examinador

AGRADECIMENTOS

Agradeço a minha família por toda a dedicação, apoio, confiança. Principalmente aos meus pais por decidirem morar em outra cidade quando houve a necessidade e em diversos momentos difíceis estarem lá comigo.

Agradeço ao Orientador Prof. Me. Marco Diego Aurélio Mesquita por aceitar me orientar no desenvolvimento desse trabalho e também no TCC do curso de Ciência e Tecnologia. Agradeço também pelo apoio, confiança, sugestões e ensinamentos durante o desenvolvimento desse trabalho e nas aulas ministradas.

Agradeço a Banca Examinadora composta pelo Prof. Dr. Cecílio Martins de Sousa Neto e pelo Prof. Dr. Francisco Carlos Gurgel da Silva Segundo pelas críticas, sugestões e assim, como todos os outros que também foram meus professores, pelos ensinamentos, experiências e convivência durante o curso.

Agradeço também aos amigos e colegas de curso por estarmos juntos nos estudos em grupo, trabalhos em equipe e momentos de descontração. Também aos amigos fora do curso, que mesmo estando distantes, continuam mantendo contato, me motivando e apoiando em todas as adversidades.

RESUMO

A tecnologia vem avançando a passos largos e os sistemas digitais também está acompanhando rapidamente essa evolução. Os sistemas vêm sendo projetados com muitos componentes e cada vez mais complexos. Se estão mais complexos, mais análises e cuidados precisam ser tomados. Por conta disso, os simuladores de circuitos digitais são bastante utilizados. Para expandir o desenvolvimento de circuitos digitais usando simulação, esse projeto tem como solução desenvolver um simulador que seja aplicado em uma placa Arduino. Esse simulador é gerado por um transpilador que lê a descrição de um circuito projetado pelo *software* Logisim. Essa simulação é aplicada diretamente na placa Arduino e permite uma experiência física no desenvolvimento de circuitos digitais. Essa experiência é ainda mais importante quando envolve o ambiente educacional. Os resultados das simulações aplicadas no Arduino são comparados com tabelas verdade geradas com ajuda do *software* Logisim. O desenvolvimento da simulação apresentou resultados esperados, nos quais os circuitos simulados na placa Arduino se comportou como os circuitos virtuais projetados pelo Logisim.

Palavras-chave: Circuitos Digitais. Transpilador. Simulador. Placa Microcontrolada.

ABSTRACT

Technology has been advancing at great strides and digital systems are also rapidly following this evolution. Systems are being designed with many components and are becoming more and more complex. If they are more complex, more analysis and care need to be taken. Because of this, digital circuit simulators are widely used. To expand the development of digital circuits using simulation, this project has as a solution to develop a simulator that can be applied on an Arduino board. This simulator is generated by a transpilator that reads the description of a circuit designed by Logisim software. This simulation is applied directly to the Arduino board and allows for a physical experience in the development of digital circuits. This experience is even more important when it involves the educational environment. The results of the simulations applied in Arduino are compared with truth tables generated with the help of Logisim software. The development of the simulation presented expected results, in which the circuits simulated on the Arduino board behaved like the virtual circuits designed by Logisim.

Keywords: Digital Circuits. Transpiler. Simulator. Microcontrolled board.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Hoje em dia, são utilizados tantos aparelhos com circuitos digitais que o termo “digital” já faz parte do nosso vocabulário diário. As técnicas digitais passaram a ser utilizadas em quase todas as áreas: automação, ciência médica, entretenimento, transporte e assim por diante (TOCCI; WIDMER; MOSS, 2011).

Um sistema digital é uma combinação de dispositivos projetados para manipular informação lógica ou quantidades físicas representadas no formato digital. Esses dispositivos podem ser mecânicos, magnéticos, pneumáticos, mas na maioria das vezes são eletrônicos. Os sistemas digitais mais conhecidos são computadores, calculadoras, equipamentos de áudio e vídeo e os sistemas de telecomunicações (TOCCI; WIDMER; MOSS, 2011).

Com o tempo esse tipo de sistema se tornou bastante utilizado, os analógicos quase sempre estão combinados com os digitais, formando sistemas híbridos. E com os avanços da tecnologia há hoje uma enorme variedade de equipamentos digitais, cada um com um grau diferente de complexidade, podendo ter de um a milhares de componentes. Quanto mais complexos, mais elementos precisa compor, tornando sua implementação mais detalhada e exigindo mais níveis de abstração no processo de projetá-los (TRINDADE JUNIOR; JULIÃO, 2012).

Nem sempre a construção completa de um dispositivo é viável apenas para a finalidade de testes. Em casos assim, é interessante simular o funcionamento do dispositivo a fim de se detectar e corrigir falhas em um projeto antes de sua construção. A simulação é entendida como uma imitação de um processo ou operação do mundo real, a geração de uma história artificial para análise de características operacionais. Buscando evitar erros e diminuir a dificuldade em produzir sistemas digitais complexos, são utilizados modelos de simulação. Esses modelos podem ser usados para predizer os efeitos causados por uma mudança no sistema e também como ferramenta de projeto para avaliar e validar o desempenho de novos sistemas (MIYAGI, 2006).

O uso de simuladores pode trazer diversas vantagens no desenvolvimento de sistemas, como o gerenciamento do tempo, o que poderia demorar meses no sistema real pode ser realizado em minutos usando a simulação. O tempo durante a simulação poderia ser expandido ou comprimido, permitindo observar fenômenos mais complexos com mais detalhes. Outra vantagem é que permite estudar os impactos que podem acontecer caso o sistema passe por alguma modificação, principalmente situações improváveis ou possíveis melhorias que a

modificação poderá proporcionar. Também será possível avaliar propostas quando os dados de entrada são insuficientes (ESTEVES, 2009).

Da mesma forma, os simuladores também possuem algumas desvantagens. Uma delas é a existência de dificuldades no desenvolvimento dos simuladores, podendo demorar bastante tempo para serem concluídos. Sistemas mais complexos podem levar mais tempo para serem desenvolvidos e ter um custo maior, sobretudo em casos em que os resultados são difíceis de obter. Outra desvantagem é que o responsável pela análise dos resultados precisa ser treinado, para obter ótima análise a qualidade da simulação também precisa ser ótima e, portanto, o analista precisa ser habilidoso (ESTEVES, 2009).

Considerando as desvantagens dos simuladores, os sistemas digitais reais terão como vantagens o realismo e a experiência direta proporcionadas aos seus desenvolvedores. Essas vantagens são ainda mais importantes quando envolvem o aprendizado. Quem está aprendendo pode observar os detalhes que a simulação não consegue abordar, principalmente detalhes que são influenciados por fatores externos ou fatores internos, como temperatura, falhas de operação, componentes avariados e as pequenas variações de parâmetros, já que na simulação, os valores são fixos durante toda a análise e a temperatura, falhas e avarias são ignoradas.

Apesar de suas vantagens, sistemas digitais reais também possuem desvantagens. Entre algumas desvantagens de circuitos digitais reais, pode-se citar a análise dos projetos. É possível construir alguns protótipos para serem analisados, mas com diversas medições e testes realizados, que são acompanhadas por trocas de componentes, a utilização de muitos protótipos não é viável. Construir esses protótipos envolve etapas complexas e demoradas e também maquinários caros, tornando importante que o desenvolvedor já saiba como será o comportamento dos sistemas antes da fabricação (MEHL, 2021).

A tecnologia dos sistemas digitais continua se desenvolvendo e com isso, vem tentando alcançar novas áreas de conhecimento e gerando também novas necessidades. Suprindo algumas dessas necessidades, desenvolveram-se os sistemas embarcados, que são computadores anexados ao sistema que ele controla com tarefas que foram predefinidas. Esses computadores são bastante conhecidos por microcontroladores. Como é usado para tarefas específicas, pode assim, aperfeiçoar um determinado produto e diminuir o seu tamanho, bem como seu custo final. Os sistemas embarcados estão por todo lugar, costumam ser projetados a partir de uma aplicação e geralmente são usados sem muita exigência (POZZEBOM, 2014).

Usando os microcontroladores é possível combinar as vantagens dos simuladores e dos circuitos reais. Já existem muitos componentes, como sensores, motores, etc. que podem ser conectados aos microcontroladores para implementar diversas funcionalidades. Inserindo um

algoritmo no microcontrolador que permita simular um circuito real e utilizando os componentes em conjunto, é possível aproveitar a experiencia direta da construção do projeto e dos seus protótipos. Por terem um preço acessível, o uso desses componentes garante o baixo custo das simulações mesmo trabalhando com componentes reais de um circuito. O tempo e a quantidade de tentativas também podem ser melhorados, pois não é necessário que os componentes estejam soldados para funcionar. Caso o algoritmo ou comportamento precise ser atualizado, os microcontroladores podem realizar a troca de códigos rapidamente.

Esse trabalho é uma continuação da monografia, com o título “Simulação de Circuitos Digitais com Aplicação em Placa Arduino”, de mesma autoria, apresentada como requisito para conclusão do curso de Bacharelado em Ciência e Tecnologia da UFERSA, campus Angicos. O trabalho anterior visava desenvolver uma linguagem de descrição de *hardware*, enquanto o presente trabalho utiliza como entrada a saída gerada pelo *software* Logisim. Dessa forma, há uma grande melhoria em usabilidade e aplicabilidade. Para tanto, foi necessário investigar o formato do arquivo gerado pelo *software* Logisim e também desenvolver algoritmos para possibilitar a conversão das informações no formato dado para algo que o transpilador criado possa usar para gerar código.

Uma outra diferença foi também o fato de que o presente trabalho também investigou a possibilidade de uso da ferramenta desenvolvida em um estudo de caso. Tal estudo demandou mudanças na ferramenta para que possa simular *clock*, tornando possível simulação de circuitos mais avançados. O circuito usado para o estudo de caso foi um simulador de semáforo de trânsito usando registradores do tipo D com possível oscilação. O fato de a ferramenta desenvolvida ser adequada para simular atraso de propagação possibilitou a simulação de um circuito que não poderia ser simulado fielmente no Logisim.

Com o objetivo de facilitar o acompanhamento da simulação, este trabalho, diferente do anterior, também torna possível o ajuste de uma espera (*delay*) arbitrária a cada iteração da simulação. Isso facilita a depuração da simulação e permite assistir passo a passo as mudanças de estado do circuito e a influência da simulação do atraso de propagação. Também com funcionalidade mais avançada desenvolvida exclusivamente nesse trabalho, o código gerado também permite um ajuste cuidadoso do formato do *clock*, inclusive com ajuste arbitrário do tamanho do pulso do clock.

Com tudo isso, a realização desse trabalho tem como objetivo desenvolver um *software* que gere como saída um código que simule um circuito digital e possa ser compilado para rodar em um microcontrolador, a partir de um circuito projetado utilizando o *software* Logisim. Pretende-se assim obter uma ferramenta que auxilie no desenvolvimento de projetos

envolvendo sistemas digitais, principalmente no ensino, possibilitando uma maior imersão do aluno no processo de construção desses sistemas. A imersão poderá ser proporcionada tanto a alunos que já frequentam aulas voltadas ao desenvolvimento de circuitos digitais como alunos que não frequentam, pare que esses últimos possam conhecer sobre o universo digital e visualizar com mais detalhes como ele funciona no mundo real.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Criar uma solução que permita, um microcontrolador se comportar como um circuito lógico projetado usando um *software* simulador de circuitos digitais.

1.1.2 Objetivos Específicos

Dadas as questões levantadas, o presente trabalho tem os seguintes objetivos:

- Identificar o formato de um arquivo gerado pelo *software* Logisim para um circuito simulado;
- Criar um transpilador que compile o arquivo gerado pelo simulador para a linguagem usada na placa Arduino (linguagem C);
- Testar a ferramenta desenvolvida através dos passos de: projeto, compilação e execução de um circuito simulado em uma placa Arduino e verificar que o comportamento condiz com o esperado.

1.2 ORGANIZAÇÃO DO TEXTO

No capítulo 2 será apresentado a fundamentação teórica sobre os circuitos digitais, mostrando seus principais componentes e o método utilizado para analisar seu comportamento, sobre simulações e a técnica de simulação implementada nesse trabalho. Também é apresentado as ferramentas utilizadas para o desenvolvimento do trabalho e o tipo de programa que foi desenvolvido. No capítulo 3 é descrito como é identificado os componentes do circuito projetado no *software* Logisim e o algoritmo desenvolvido para compilar esse circuito para a placa Arduino. No capítulo 4 são apresentados os resultados e discussões dos testes aplicados ao programa desenvolvido. Por último, no capitulo 5 é apresentado as conclusões deste trabalho e o que pode ser desenvolvido futuramente.

1.3 METODOLOGIA

O desenvolvimento desse projeto foi iniciado analisando a descrição dos circuitos nos arquivos ".circ" criados pelo *software* Logisim. Foi observado o que acontece na descrição quando algumas características dos componentes do circuito são alteradas, como o tamanho, posição e direção. O que diferencia na descrição quando componentes são bastante parecidos também foi observado.

Em seguida é desenvolvido um algoritmo no qual lerá esses arquivos ".circ" e identificará as informações que caracterizam os componentes do circuito. Em seguida, utilizando esses dados obtidos, o algoritmo cria um novo arquivo em linguagem C que rode na placa Arduino, no qual apresentará uma descrição do mesmo circuito projetado anteriormente pelo *software* Logisim.

E para realizar os testes, esses novos arquivos foram compilados em uma placa Arduino, sendo utilizados *jumpers*, resistores e LEDs (Diodo Emissor de Luz, do inglês, *Light Emitting Diode*) para auxiliar na obtenção dos resultados.

2 SIMULAÇÃO DE CIRCUITOS DIGITAIS

Nesta seção serão abordados alguns conceitos fundamentais que envolvem circuitos digitais. Também serão relacionados esses conceitos com as características de simulações, algumas técnicas e assuntos referentes ao projeto descrito por essa monografia.

2.1 CIRCUITOS DIGITAIS

Os circuitos digitais são projetos eletrônicos que produzem tensões de saída e respondem a tensões de entrada que estejam dentro de faixas definidas para os níveis 0 e 1. O circuito não deve distinguir as tensões de entrada alocadas em uma mesma faixa e tem que responder igualmente a todas as tensões na mesma faixa (TOCCI; WIDMER; MOSS, 2011).

Os circuitos digitais podem ser construídos com minúsculos dispositivos eletrônicos combinando-os de inúmeras maneiras. Esses minúsculos dispositivos são denominados portas e podem calcular várias funções dos níveis de tensão. Essas portas formam a base do *hardware* sobre a qual todos os projetos digitais são construídos (TANENBAUM, 2007).

Essas minúsculas portas podem ser construídas de diodos, transistores e resistores interconectados de modo que a saída seja um resultado de uma operação lógica básica. Por causa dessas operações as portas são bastante conhecidas como portas lógicas. A álgebra *booleana* é constituída praticamente de três operações lógicas básicas (*OR*, *AND* ou *NOT*), também conhecidas como operadores lógicos (TOCCI; WIDMER; MOSS, 2011). Para cada operador lógico é possível construir um componente de um circuito digital que o implementa. Assim, qualquer expressão lógica pode ser transformada em um circuito digital que produza uma saída equivalente dadas suas entradas correspondentes às variáveis da expressão.

2.1.1 Lógica Booleana

Em diversas situações do cotidiano pode-se dizer que algo pode apresentar-se somente em dois estados. Por exemplo, uma lâmpada pode estar acesa ou apagada ou uma porta pode estar fechada ou aberta. Em 1854, o matemático George Boole estudou como se toma decisões lógicas com base em circunstâncias que só podem ser respondidas com verdadeiro ou falso. Os resultados das suas pesquisas são conhecidos hoje em dia como lógica booleana e o sistema que utiliza símbolos e operadores para descrever essas decisões é chamado de álgebra booleana. A

álgebra booleana usa símbolos para representar uma expressão lógica que possui um ou dois valores possíveis, verdadeiro ou falso (TOCCI; WIDMER; MOSS, 2011).

Derivando da álgebra de Boole, surgiram as funções lógicas, as principais funções são: função E, função OU e função NÃO. As portas lógicas responsáveis pelas três operações lógicas básicas formam a base dos projetos digitais e são aplicações práticas dessas principais funções lógicas (IDOETA; CAPUANO, 2012).

Conhecer a lógica booleana é importante para compreender o funcionamento das portas lógicas, que será explanado posteriormente, e para construir a representação dessas portas na linguagem de programação usada para o desenvolvimento desse projeto.

2.1.2 Lógica Digital

A álgebra booleana é um instrumento precioso para criar uma relação de entrada e saída nos projetos de circuitos lógicos. Esse processo, quando acontece, é conhecido por síntese de circuitos lógicos, e mostra que a álgebra boolena não é apenas utilizada para análise e simplificação de sistemas lógicos (TOCCI; WIDMER; MOSS, 2011).

Aplicando essa ferramenta nos projetos de circuitos de interruptores ou circuitos de comutação, podemos dizer que estamos trabalhando com “Lógica Digital” para desenvolver sistemas digitais. A álgebra booleana serve como base para a lógica digital, mas, utilizando características físicas e simbologias relacionadas a operações com máquinas elétricas. Os valores lógicos serão representados por impulsos elétricos (DIAS, 2021).

Na lógica digital, de forma análoga a lógica booleana, podemos afirmar que existem dois estados lógicos, o estado lógico 1 e o estado lógico 0. Considerando um exemplo simples, de um circuito com um interruptor e uma lâmpada, podemos considerar o estado lógico 1 como sendo o interruptor fechado, havendo passagem de corrente, e consequentemente a lâmpada acesa. Assim, o estado lógico 0 indica o interruptor aberto, não ocorrendo passagem de corrente, logo, a lâmpada está apagada. Com o exemplo podemos ver que há uma relação entre a álgebra booleana e os circuitos elétricos, mostrando que as funções booleanas também podem ser aplicadas aos circuitos elétricos. Dessa associação das funções com alguns circuitos básicos surgiram as portas lógicas (DIAS, 2021).

2.1.3 Portas Lógicas

Através de um pequeno número de partes microscópicas de circuitos eletrônicos, que podem ser arranjados de várias formas diferentes, é possível construir os circuitos eletrônicos

digitais. Esses dispositivos eletrônicos que podem ter dimensões de alguns nanômetros são denominados portas lógicas e são usados para calcular várias funções lógicas dos níveis de tensão (TANENBAUM, 2007).

Ao se fazer uma ilustração para descrever um circuito, as portas lógicas (*AND*, *OR* e *NOT*), podem ser representadas através de símbolos, utilizando linhas como entrada onde serão colocados os valores a serem operados e uma linha como saída que representa o valor final da função. Uma expressão booleana complexa pode ser representada ligando entre si vários desses símbolos de forma que seja construída a função pretendida (ALVES, 2004).

Existe uma forma de reproduzir em uma tabela o modo como a porta lógica se comporta. Essa tabela é chamada de Tabela Verdade e funciona como um mapa onde são colocadas todas as situações da função lógica com seus respectivos resultados (IDOETA; CAPUANO, 2012).

A função E, também conhecida como *AND*, executa uma multiplicação de duas ou mais variáveis *booleanas*. É representado algebricamente por um ".", no qual se lê "e". Como resultado dessa função teremos o estado 1 se, somente se, todas as entradas estiverem com valor 1. Em qualquer outro caso o estado vai ser 0. Sua tabela verdade é:

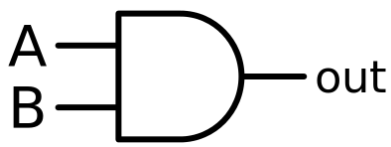
Tabela 1 - Tabela verdade da porta *AND*

A	**B**	**A . B**
0	0	0
0	1	0
1	0	0
1	1	1

Fonte: Elaborada pelo autor (2021)

E seu símbolo usado nas representações de circuito é:

Figura 1 - Símbolo da porta *AND*



Fonte: Wikiwand[1]

[1] Disponível em: <http://www.wikiwand.com/simple/Logic_gate>. Acesso em mar. 2021.

A função OU, também chamada de *OR*, é representada pelo “+”, onde se lê “ou”. No resultado dessa função somente será 0 quando os valores de entrada forem todos 0. Se houver pelo menos uma entrada com valor 1 o resultado da função será 1. Sua tabela verdade é:

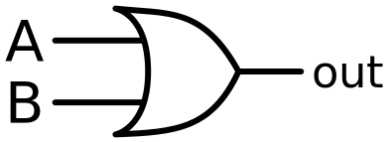
Tabela 2 - Tabela verdade da porta *OR*

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

Fonte: Elaborada pelo autor (2021)

E seu símbolo usado nas representações de circuito é:

Figura 2 - Símbolo da porta *OR*



Fonte: Wikiwand[2]

A função mais simples é a NÃO, conhecida também por *NOT*, é representada por um “–” em cima da variável. Essa função inverte os estados, se o estado for 1, então sua saída será 0, e vice-versa (IDOETA; CAPUANO, 2012). Sua tabela verdade é:

Tabela 3 - Tabela verdade da porta *NOT*

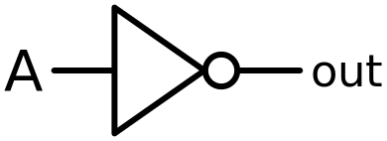
A	**$\bar{A}$**
0	1
1	0

Fonte: Elaborada pelo autor (2021)

[2] Disponível em: <http://www.wikiwand.com/simple/Logic_gate>. Acesso em mar. 2021.

E seu símbolo usado nas representações de circuito é:

Figura 3 - Símbolo da porta *NOT*



Fonte: Wikiwand[3]

É claro que é possível especificar outras portas lógicas simplesmente a partir de até duas entradas e uma saída. Da mesma forma, qualquer circuito digital também pode ser implementado utilizando-se somente um tipo de porta lógica: a negação da porta E ou a negação da porta OU. No entanto, as portas apresentadas são suficientes para especificar qualquer circuito lógico, possuem descrição intuitiva e qualquer detalhe mais aprofundado foge ao escopo deste trabalho.

2.2 SIMULAÇÃO

A simulação é uma ferramenta que permite a diversos profissionais realizar as atividades propostas. Através dela, podem adquirir capacidade de identificar, formular e solucionar problemas ligados as atividades de projetos, operação e sistemas de produção de bens e serviços (GAVIRA, 2003).

O cenário atual, no qual a capacidade computacional e as metodologias de simulação desenvolveram-se bastante, fez da simulação umas das técnicas mais utilizadas para análise e desenvolvimento de sistemas.

Os dados de saída de uma simulação devem satisfazer diretamente as saídas que seriam obtidas de um sistema real. Quando se modela o que acontece em sistemas reais, a simulação é mais vantajosa quando são utilizados poucos recursos. As principais vantagens da simulação, descritas por Miyagi (2006), são:

- Equipamentos, sistemas de transportes, arranjos físicos, etc. podem ser testados antes de investir recursos com as aquisições envolvidas;

[3] Disponível em: <http://www.wikiwand.com/simple/Logic_gate>. Acesso em mar. 2021.

- O tempo pode ser comprimido ou expandido, permitindo que o fenômeno em estudo possa ser acelerado ou retardado;
- Hipóteses de como e por que certos fenômenos ocorrem podem ser avaliados;
- Pontos onde as informações ou materiais tem seus fluxos comprometidos podem ser identificados.

É claro que nem tudo pode ser simulado de forma fiel ou não engloba completamente todas as características de uma experiência com sistemas reais. Assim, uma simulação tem desvantagens. Algumas das principais, conforme Miyagi (2006) descreve, são:

- A construção dos modelos, muitas vezes, necessita de um treinamento especial. As técnicas são aprendidas ao longo do tempo e envolvem o bom uso da experiência.
- Os resultados da simulação podem ser difíceis de interpretar. As saídas da simulação podem incluir variáveis aleatórias, assim, definir se as saídas são respostas efetivas do sistema ou se são derivadas da aleatoriedade da simulação não é uma tarefa trivial.
- A modelagem do sistema e a análise dos dados podem consumir muito tempo e muitos recursos. Por outro lado, diminuir o tempo e recursos podem gerar cenários insuficientes para atender os objetivos.

Diversos fatores surgiram para minimizar as desvantagens da simulação. Os avanços nas plataformas computacionais e pacotes com ferramentas desenvolvidos por fornecedores de *softwares* são fatores que permitem a construção mais rápida da simulação e mais facilidade na análise dos dados de saída (MIYAGI, 2006).

2.2.1 Modelos de Simulação

Considerando os dispositivos eletrônicos mais elaborados, sua complexidade tornou-se tão grande que nenhum simulador pode lidar com todos os aspectos da simulação ou fazê-lo em tempo hábil. Através disso, diferentes tipos de simuladores foram desenvolvidos para trabalhar com as diferentes áreas de simulação. Uma forma popular de classificar os simuladores é baseada em seu nível de abstração do sistema (CHEN, 1992).

Também é possível classificar os simuladores de acordo com o tipo de modelo interno de processamento. Podem ser divididos em dois tipos principais: simuladores compilados e simuladores baseados em eventos.

Um simulador compilado é executado através de um modelo de código compilado. A simulação é realizada executando o modelo resultante um determinado número de vezes. A simulação compilada é principalmente empregada para verificação funcional, onde a temporização do modelo não é tão preocupante. É também aplicável a modelos síncronos, no qual a temporização pode ser verificada separadamente.

Um simulador baseado em eventos pode reconhecer a quantidade de atividades dentro de um modelo e simular apenas os componentes que estão envolvidos nas mudanças dos valores da saída. Ele é chamado de baseado em eventos por que toda a simulação é dirigida pelos eventos. Considera-se a ocorrência de um evento quando ocorre uma mudança de sinal. Alguns componentes quando são ativados, podem por sua vez, alterar os valores de saída, gerando novos eventos. A passagem de tempo é fundamental para simulação baseada em eventos, sendo assim, aplicável a modelos assíncronos (CHEN, 1992).

2.2.2 Simulação Implementada

A simulação compilada é aplicável quando a lógica combinacional não possui algum tempo de atraso. Também é eficiente para modelos altamente ativos, para modelos com baixa atividade o método pode ser menos eficiente. A modelagem pode ser implementada usando uma linguagem de programação (por exemplo, linguagem C) e então um modelo executável será gerado. No final, a execução do programa indica que o sistema está sendo simulado (CHAKRABARTY, 2016).

Embora a simulação de modelos baseada em eventos tenha um grande potencial em termos de desempenho, a simulação compilada pode ser implementada de forma consideravelmente mais simples. Por essa característica, e considerando-se as limitações de um microcontrolador, optou-se por uma implementação usando-se simulação compilada no desenvolvimento desse projeto.

As operações de portas lógicas, realizadas em algoritmos para simular um circuito, são baseadas em sintaxes de linguagens de programação de alto nível. As sintaxes de atribuição, loops, condições lógicas, operadores de múltipla escolha e as chamadas de funções são usadas durante o processo. Nas etapas da simulação em que múltiplas portas seriam processadas simultaneamente, utilizando a modelagem compilada, é necessário que exista uma ordem de execução dessas portas no algoritmo, garantindo a estabilidade, por exemplo, os circuitos sequenciais com várias iterações (LAPIN; BULAKH; VAGAPOV, 2017).

2.3 LOGISIM

O Logisim é um *software* livre, disponível sob a Licença Pública Geral GNU (do inglês, GNU *General Public License*). Foi desenvolvido por Carl Burch, em 2001, atualmente professor de ciência da computação, no *Hendrix College*. É uma ferramenta educacional para concepção e simulação de circuitos lógicos. Simples o bastante para facilitar a aprendizagem dos conceitos mais básicos relacionados aos circuitos lógicos. Possui interface intuitiva e com ferramentas que podem simular circuitos à medida que são construídos. Estudantes de faculdades e universidades em todo o mundo utilizam o Logisim, em cursos de organização de computadores e até mesmo semestres inteiros em cursos mais avançados de arquitetura de computadores (BURCH, 2021).

Os circuitos projetados pelo Logisim são salvos em arquivos com formato ".circ". Neste trabalho, esses arquivos serão lidos e transpilados para um código em linguagem C que implementa um simulador para o circuito descrito. Esse código pode ser então compilado para rodar em uma placa microcontrolada de tal forma que a placa, então, se comporte como o circuito projetado.

Dado que pretende-se executar um simulador de circuitos digitais em uma placa microcontrolada, mais detalhes sobre essa ferramenta de prototipagem são necessários. Em seguida será possível conhecer sobre as placas microcontroladas e sobre o processo de transpilação.

2.4 PLACAS MICROCONTROLADAS

Os microcontroladores são pequenos computadores que são embutidos em diversos aparelhos. Podem desempenhar várias funções, como gerenciar esses aparelhos e manipular a interface de usuário. São encontrados em diferentes tipos de aparelhos, em diferentes categorias, como eletrodomésticos, equipamentos de entretenimento, brinquedos, dispositivos de venda, entre outros. Dentro de alguns anos, tudo que funciona por energia elétrica irá conter um microcontrolador (TANENBAUM, 2007).

Por si só, um microcontrolador é um componente eletrônico que pode ser adicionado a uma placa de circuito impresso para desempenhar seu papel. Para prototipagem rápida e uso mais simples, no entanto, não é conveniente reconstruir uma placa inteira e tampouco remover um microcontrolador de um circuito recém-construído, reprogramá-lo e inseri-lo novamente na placa cada vez que se deseje fazer pequenas alterações ou ajustes no circuito. A fim de agilizar

o desenvolvimento, uma solução bastante comum é o uso de placas que incluem um microcontrolador, alguma interface que permita sua programação e um conjunto de pinos ou conectores que permitem que essa placa interaja com outros dispositivos. A essas placas dá-se o nome de placas microcontroladas.

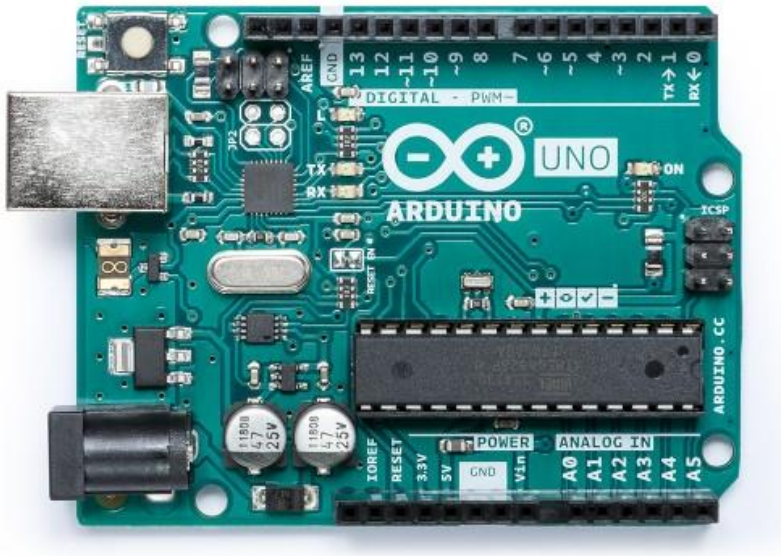
2.4.1 Arduino

O Arduino é uma ferramenta popular para o desenvolvimento de produtos IoT (*Internet of Things*, no português, Internet das Coisas), bem como uma das ferramentas mais bem-sucedidas para educação STEM (*Science*, *Technology*, *Engineering and Mathematics*, no português, Ciência, Tecnologia, Engenharia e Matemática), além de ser bastante usada, principalmente por profissionais, para inovar em músicas, brinquedos, agricultura, etc. Existem uma gama de ferramentas de *software*, plataformas de *hardware* e documentação, permitindo que qualquer pessoa seja praticamente criativa com a tecnologia.

O Arduino é o primeiro projeto de *hardware* de código aberto amplamente difundido. Várias placas e bibliotecas de *software* foram desenvolvidas, ampliando e diversificando os possíveis projetos criados pela comunidade (ARDUINO, 2018).

A placa Arduino é baseada em um microcontrolador da família AVR bastante versátil, podendo operar sozinha no controle de vários dispositivos, sendo assim, muito aplicada em robótica e sistemas embarcados. A plataforma dispõe de uma camada simples de software implementada na placa, um *bootloader* e uma interface amigável, IDE do Arduino, que utiliza uma linguagem baseada em C, a linguagem *Processing*. O *bootloader* facilita a programação do microcontrolador, não exigindo compiladores ou hardware adicional (SOUZA et al., 2011). O modelo Arduino UNO, que será usado neste projeto, é baseado no microcontrolador ATmega328P.

Figura 4 - Placa do modelo Arduino UNO



Fonte: Arduino[4]

O Arduino será responsável por realizar a simulação na prática, a partir do que foi projetado no *software* Logisim. Foi escolhido o Arduino como a placa microcontrolada para o desenvolvimento do projeto por ser muito popular e de fácil acesso.

2.5 COMPILADOR

O compilador é um *software* complexo, criado para converter um programa produzido por uma linguagem de programação de alto nível (também chamada de linguagem fonte) para um programa numa linguagem que possa ser executada por um computador (linguagem-objeto).

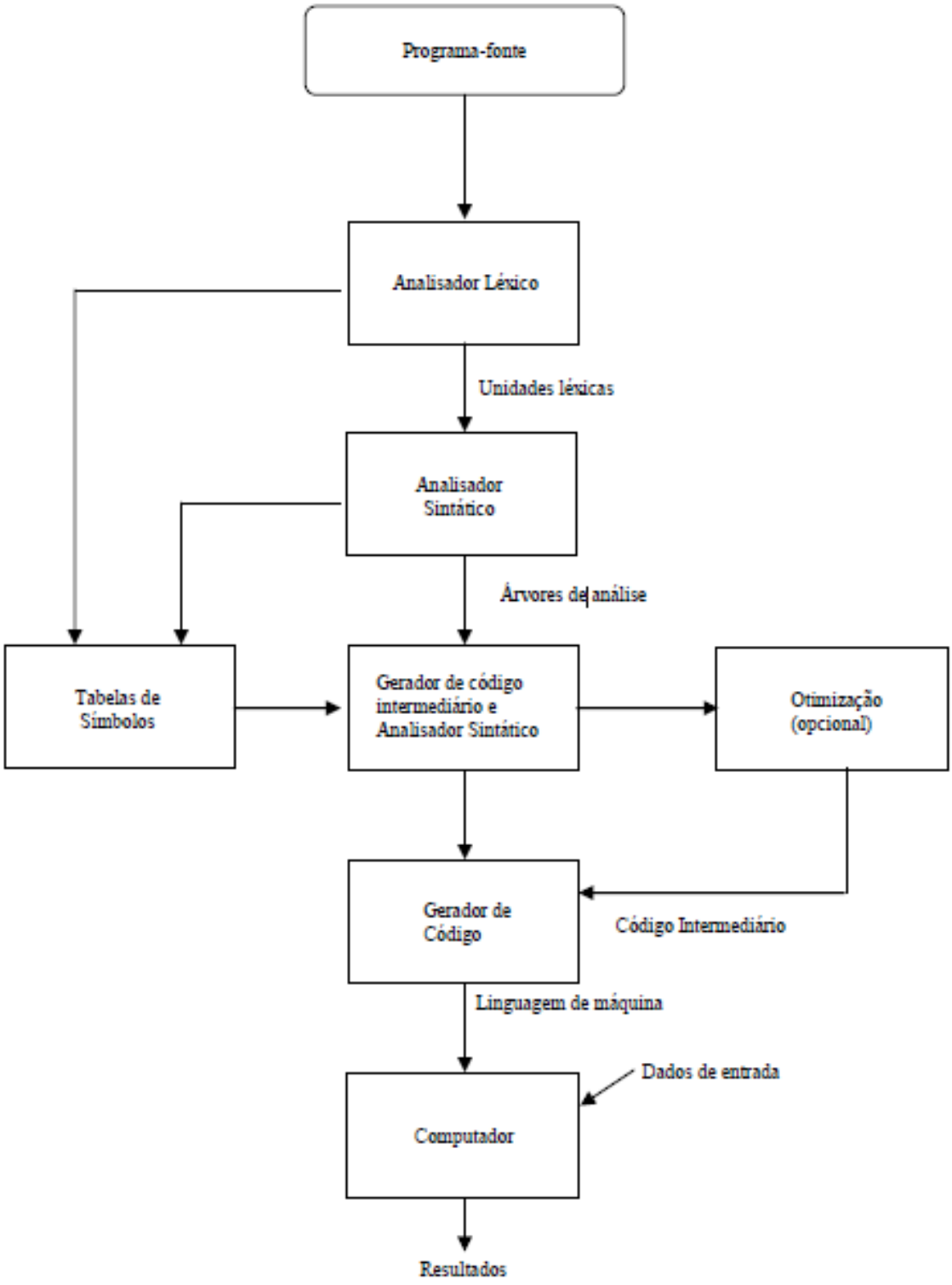
A construção do compilador precisa seguir dois princípios fundamentais: o compilador deve preservar o significado do programa fonte e deve melhorar esse programa de alguma forma perceptível (MARANGON, 2021). Geralmente um compilador produz um programa em linguagem simbólica, ou *assembly*, não diretamente código de máquina. Então, esse programa simbólico, que é semanticamente equivalente ao programa em linguagem de alto nível, é traduzido em linguagem de máquina através de *assemblers*.

[4] Disponível em: < https://store.arduino.cc/usa/arduino-uno-rev3>. Acesso em mar. 2021.

O processo de compilação pode ser desenvolvido em várias etapas. Em uma etapa o analisador léxico junta os caracteres do programa fonte, como palavras especiais e identificadores, em unidades léxicas. Comentários são ignorados, já que não serão úteis para o compilador. Na próxima etapa o analisador sintático usa as unidades léxicas da etapa anterior para construir estruturas hierárquicas chamadas árvores de análise, as quais representam a estrutura sintática do programa. Informações sobre tipos e atributos são salvos em tabelas pelos analisadores léxicos e sintáticos para serem usados mais a frente pelo analisador semântico e o gerador de código (URICER, 2001).

O gerador de código intermediário em seguida vai produzir um programa com nível de linguagem entre a do programa de alto nível e a linguagem do código objeto. O analisador semântico faz parte do gerador de código intermediário e verifica se existiu algum erro difícil, ou impossível de ser detectado durante a análise sintática, por exemplo, erros de tipo. O gerador de código, como última etapa, traduz a versão do código intermediário para um programa equivalente em linguagem de máquina. Pode existir uma etapa para otimização do código do programa final, mas essa etapa é opcional no processo de compilação (URICER, 2001). Através do fluxograma apresentado pela Figura 5 é possível observar melhor o processo de compilação.

Figura 5 - Processo de compilação



Fonte: (URICER, 2001)

2.5.1 Transpilador

A transpilação é um tipo particular de compilação. Se um compilador consegue converter um código de certo nível de linguagem para outra com o mesmo nível de abstração, então esse compilador é também um transpilador. Se ele só consegue converter uma linguagem para outras de níveis diferentes, normalmente do nível mais alto para um nível inferior, então é apenas um compilador. Assim, todos os transpiladores também são compiladores, mas nem todo compilador pode ser considerado transpilador (CABOT, 2017).

Por exemplo, em 1979, a Intel publica detalhes de um conversor chamado CONV86. Esse conversor pode converter o código-fonte do *assembly* 8080/8085 para o código-fonte do *assembly* 8086. O termo “transpilador” não é utilizado na publicação, surgindo algum tempo depois, mas é considerado uma transpilação, por converter códigos em linguagens com o mesmo nível de abstração (PADMANABHAN; P, 2019).

Conhecer bem uma linguagem não vai garantir que ao tentar aprender outra linguagem se torne mais fácil, dado que as sintaxes parecidas podem conter armadilhas. Cada linguagem apresenta características, sintaxes e paradigmas únicos, as vezes possuem semelhança sintática com outras, mas possuem comportamentos diferentes. Algumas podem ter sintaxes e paradigmas complicados, tornando muitas vezes pouco produtivo o seu uso. A utilização de transpiladores irá facilitar várias tarefas, pois no lugar de códigos complexos, pode-se usar uma linguagem com sintaxes e paradigmas mais agradáveis (REIS V, 2016).

Um exemplo de aplicação é em navegadores antigos que, neste caso, não têm suporte às versões atuais do JavaScript (linguagem de programação utilizada para manipular comportamentos nas páginas web). Assim pode-se programar nas versões atualizadas e depois converter o código para as versões antigas, sem prejudicar usuários com navegadores antigos. Com o transpilador é possível programar com as técnicas mais atualizadas e eficientes e converter para que o programa funcione em outras versões (CARVALHO, 2017).

O transpilador desenvolvido nesse projeto é responsável por ler os arquivos gerados pelo Logisim para os circuitos nele projetados e, em seguida, criar um código fonte que, quando compilado, simule o mesmo circuito projetado e que possa posteriormente ser executado em uma placa microcontrolada.

3 MÉTODO PROPOSTO

Utilizando uma placa microcontrolada podemos montar vários circuitos em que muitos de seus componentes são mais fáceis de se obter e não possuem custo alto, possibilitando experiências físicas aos desenvolvedores. Usando uma placa microcontrolada em conjunto com o simulador também será vantajoso. Os resultados da simulação poderão ser jogados diretamente no circuito físico, abrangendo as possíveis formas de análise e detecção de problemas.

Rodar um simulador de circuitos digitais em um microcontrolador permite usá-lo de forma parecida com um circuito digital real. Depois que a simulação é carregada para um microcontrolador, o simulador só precisará da fonte de energia para poder ser usado como um circuito real. Os microcontroladores funcionam em tempo real, quando recebem um estímulo devem responder de forma mais rápida possível. Então, qualquer mudança nos valores das entradas do microcontrolador, que correspondem as entradas do circuito simulado, serão processados imediatamente e seus resultados vão para as saídas do microcontrolador correspondentes às saídas do circuito simulado.

3.1 TRANSPILAÇÃO DA DESCRIÇÃO DE UM CIRCUITO

O circuito simulado no microcontrolador tem como base, um circuito projetado virtualmente pelo *software* Logisim. Para que o microcontrolador entenda e se comporte como o circuito desejado, é necessário que ocorra uma conversão entre o formato de arquivo gerado pelo Logisim e a linguagem usada para programar o microcontrolador. Para isso foi desenvolvido neste trabalho um transpilador.

O transpilador implementado, toma como entrada a descrição de um circuito, obtida através do arquivo “.circ” gerado pelo *software* Logisim . Como saída, o transpilador gera o código em linguagem C de um simulador do circuito descrito. Um exemplo de uso do transpilador é apresentado no Quadro 1:

Quadro 1 - Exemplo de uso do transpilador usando linha de comando

```
simulador.exe circuito.circ circuito.ino
```

Fonte: Elaborado pelo autor (2021)

O “simulador.exe” é o nome do transpilador, o “circuito.circ” indica o arquivo de entrada da descrição do circuito e “circuito.ino” indica a saída em Linguagem C do Arduino.

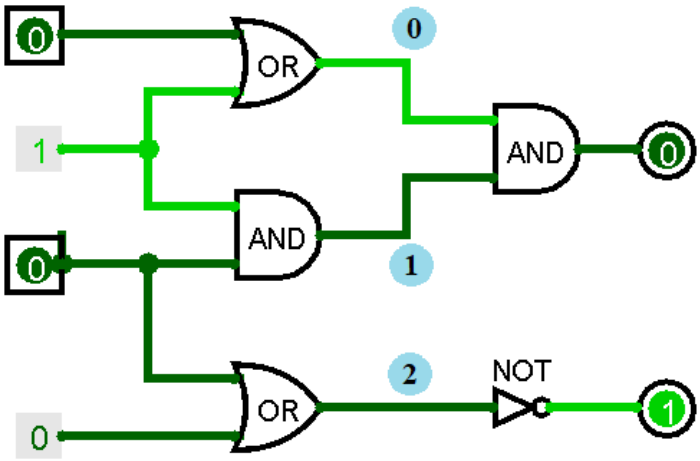
O código gerado pode então ser compilado e rodado em um dispositivo a fim de simular o circuito descrito. O gerador de código implementado atualmente permite a geração de um programa fonte que pode ser compilado para o Arduino, a mesma é indicada pelo “.ino” no final da linha de comando.

No processo de geração do código, os fios que serão simulados pelo código gerado pelo transpilador precisam receber os resultados das operações das portas lógicas para que sejam redirecionadas posteriormente para a próxima porta. Esses resultados são salvos em um vetor, sendo que cada posição do vetor indica um fio do circuito simulado. O primeiro fio que é identificado recebe a posição 0 do vetor e esse número ficará associado às entradas e às saídas que estão conectadas a esse fio. Os próximos fios que são identificados vão ocupando as próximas posições do vetor e seu número associado de forma semelhante ao primeiro fio.

Nos circuitos que apresentam características gráficas semelhantes ao circuito na Figura 6, os quadrados brancos e círculos brancos, com círculos verdes dentro de ambos, indicam entradas e saídas do circuito, respectivamente. Os quadrados cinzas indicam constantes conectadas, no qual o valor 1 neles indica nível lógico alto e o valor 0 indica nível lógico baixo. Os valores de níveis lógicos alto e baixo também são identificados pela tonalidade do verde presente nas entradas, saídas e fios, sendo uma tonalidade clara para nível lógico alto e uma tonalidade escura para nível lógico baixo.

A Figura 6 junto com a Tabela 4 ilustra um exemplo de um circuito simples e a forma de como os fios de um circuito serão simulados.

Figura 6 - Circuito simples com identificação dos fios durante simulação



Fonte: Elaborada pelo autor (2021)

Tabela 4 - Valores dos fios durante a simulação do circuito da Figura 6

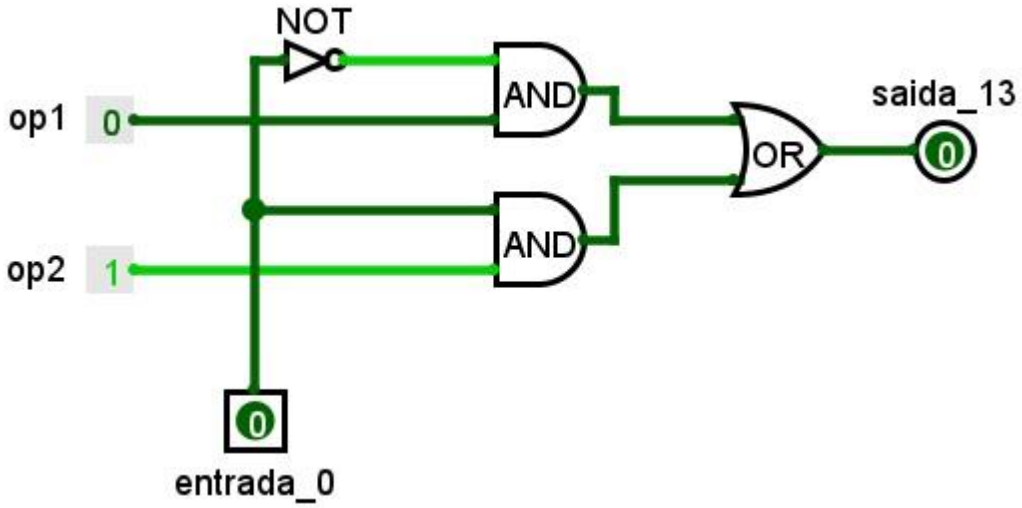
Fio	Valor
0	1
1	0
2	0

Fonte: Elaborada pelo autor (2021)

Um circuito multiplexador é utilizado para selecionar um, entre vários sinais de entrada, para que este seja transmitido diretamente para saída. Pode ter até 2^N entradas e para selecionar o sinal de saída, necessita que N sinais de seleção estejam presentes no circuito (LIMA, 2015). Para o circuito multiplexador apresentado na Figura 7 o sinal de seleção é identificado pelo pino de entrada “entrada_0”, enquanto as constantes “op1” e “op2” são as entradas disponíveis para serem selecionada. A saída do circuito é identificada pelo pino “saída_13”.

Os pinos da placa microcontrolada que serão usados para funcionar como entradas e saídas do circuito, são identificados pelos números seguidos do caractere “_” nos rótulos das entradas e saídas do circuito, como é exibido na Figura 7. Na descrição do circuito da Figura 7, é apresentado como ficarão as entradas e saídas no transpilador.

Figura 7 - Circuito multiplexador de duas entradas com descrições



Fonte: Elaborada pelo autor (2021)

Tabela 5 - Identificação das entradas e saídas das portas no transpilador

Tipo	Entrada 1	Entrada 2	Saída
NOT	-3	−	0
AND	0	-1	1
AND	-3	-2	2
OR	1	2	-14

Fonte: Elaborada pelo autor (2021)

Tabela 6 - Valor armazenado para cada fio

Fio	Valor
0	0
1	0
2	0

Fonte: Elaborada pelo autor (2021)

Os valores apresentados na Tabela 5 indicam o que está conectado as entradas e a saída de cada porta. Os valores negativos de entradas e saídas nas tabelas vistas podem parecer confusas à primeira vista. Todavia, isso é uma estratégia elaborada para permitir que um único valor possibilite diferenciar se a entrada ou saída de uma porta está conectada a um fio do circuito simulado, uma constante 0 ou 1 ou a um pino de entrada ou saída que será associado a um pino físico do microcontrolador. O transpilador executa as operações de cada porta na ordem que elas são descritas. Quando processa-se a primeira porta, verifica-se na Tabela 5 a identificação de cada entrada, se ela for maior ou igual 0 o valor que será utilizado está em um fio identificado por aquele número. Se for -1 o valor a ser usado é da constante 0, se -2 o valor é da constante 1 e se for menor que -2 é utilizado uma fórmula $(-(ID\ +\ 3))$ para indicar de qual pino físico do microcontrolador será retirado o valor. Depois que recebe os valores e realiza o cálculo, a tabela é analisada novamente para consultar a coluna de saída, se o identificador for igual ou maior que 0, o resultado é salvo no fio com essa identificação, se for menor que 0 o resultado é enviado para o pino correspondente. É possível determinar qual o pino físico do microcontrolador correspondente por meio da fórmula $(-(ID\ +\ 1))$.

O esquema descrito especifica uma forma de, utilizando-se apenas um valor numérico, determinar se cada terminal de uma porta lógica simulada está conectado a um valor constante,

um fio simulado ou um pino físico. Internamente, durante o processo de transpilação, o circuito a ser simulado é representado na forma de tabelas para seus fios e suas portas lógicas. Posteriormente, na fase de execução as mesmas tabelas são utilizadas.

Uma vez que a tabela que especifica a forma como as portas lógicas, fios e pinos estão conectados é preenchida, é possível gerar o código de um simulador para o circuito representado. O processo de transpilação finaliza quando os dados são organizados e manipulados no código em linguagem C. Como permite gerar um programa que compile em Arduino, ele também mantém as características, paradigmas e sintaxes da linguagem Arduino.

O trecho de código que contém as variáveis, estruturas e funções para as operações dos valores são os primeiros a serem escritos no programa fonte em linguagem que possa ser compilada para um Arduino, por ser um código que estará presente igualmente em qualquer programa gerado durante a simulação.

Depois é escrita a função *setup* do Arduino contendo os pinos usados nas descrições do circuito, as informações das portas lógicas e outros dados necessários para inicializar a simulação.

O Quadro 2 apresenta, para o circuito ilustrado, a função *setup* gerada pelo transpilador desenvolvido.

Quadro 2 - Função setup() gerada no final da transpilação do circuito descrito

```
void setup(){
	pinMode(0, INPUT);

	pinMode(13, OUTPUT);

	portas[0].tipo = NOT;
	portas[0].inp1 = IN_PORT(0);
	portas[0].out = 0;

	portas[1].tipo = AND;
	portas[1].inp1 = 0;
	portas[1].inp2 = IN_CONST_0;
	portas[1].out = 1;

	portas[2].tipo = AND;
	portas[2].inp1 = IN_PORT(0);
	portas[2].inp2 = IN_CONST_1;
	portas[2].out = 2;

	portas[3].tipo = OR;
	portas[3].inp1 = 1;
	portas[3].inp2 = 2;
	portas[3].out = OUT_PORT(13);

	c.portas = portas;
	c.fios = fios;
	c.n_portas = 4;
}
```

Fonte: Elaborado pelo autor (2021)

Em seguida é escrita a função *loop*, onde a simulação propriamente dita ocorre, chamando as funções que realizam os cálculos para serem constantemente executadas, mantendo o simulador sempre ativo. Através dessa função que as portas não precisam ser descritas na mesma ordem do circuito, por que, por exemplo, se a primeira porta for executada como uma das últimas, as portas que seguem a ordem original vão ser executadas novamente nas próximas chamadas da função *loop*, e assim os resultados podem ser passados adiante no simulador.

Quadro 3 - Função loop() gerada no final da transpilação do circuito descrito

```
void loop(){
        int i;
        for(i = 0; i < c.n_portas; i++){
                simula_porta(c, i);
        }
}
```

Fonte: Elaborado pelo autor (2021)

A função “simula_porta(c, i)” simplesmente simula a porta ‘i’ do circuito ‘c’. Apesar de ser o “coração” do simulador, a implementação dessa função é bastante simples. Primeiramente, a tabela de portas é consultada para determinar o tipo da porta ‘i’. Uma vez que o tipo da porta a ser simulada é determinado, o código é desviado usando um ‘switch’ para simular o tipo de porta em questão. Posteriormente os valores de entrada da porta são consultados de acordo com o esquema descrito anteriormente e seu valor de saída é calculado. Novamente utilizando-se o esquema descrito anteriormente, é determinado para onde o valor de saída da porta deve ser ajustado. No fluxograma da Figura 8 é apresentado os passos realizados pela função “simula_porta(c, i)”. Caso a porta identificada for a porta NOT o bloco “consultar valor na segunda porta” é ignorado sequência.

Figura 8 - Fluxograma da função "simula_porta(c, i)"

Fonte: Elaborada pelo autor (2021)

No final da transpilação, o código gerado identificará o tipo das portas também por números e através desse valor, escolherá qual operação lógica irá utilizar naquele momento. As portas são AND, OR, NOT, XOR, NAND, NOR e XNOR, os números que as representam são 0 para AND, 1 para OR, 2 para NOT e seguindo a ordem respectivamente. Assim o simulador trabalha com várias listas que representam os dados que foram lidos da descrição do circuito. A seguir as listas criadas em relação a descrição do circuito do Quadro 3.

Lista de Portas:

- Porta 0, tipo 2 (NOT);
- Porta 1, tipo 0 (AND);
- Porta 2, tipo 0 (AND);
- Porta 3, tipo 1 (OR).

Lista de Entradas:

- -3 (pino 0).

Lista de Saídas:

- -14 (pino 13).

Lista de Fios:

- 0;
- 1;
- 2.

Lista de Constantes:

- -1 (constante 0);
- -2 (constante 1).

3.2 DESCRIÇÃO GERADA PELO *SOFTWARE* LOGISIM

Um circuito digital projetado no Logisim é salvo em um arquivo com formato “.circ”. Esse arquivo descreve o circuito utilizando uma linguagem de marcação chamada XML (Linguagem de Marcação Extensível, do inglês, *Extensible Markup Language*). A linguagem XML é recomendada pela W3C (Consórcio da Rede Mundial de Computadores, do inglês, *World Wide Web Consortium*) para criação de documentos, onde seus dados são organizados hierarquicamente.

As linguagens de marcação são utilizadas para definir padrões e formas de exibição dentro de um documento para serem lidos por computadores ou pessoas. Utilizam o conceito de marcador ou *tag*, que ao serem reconhecidos por um sistema, o conteúdo será modificado

ou interpretado de acordo com o significado do marcador (PORTAL EDUCAÇÃO, 2021). O XML é usado para padronizar uma sequência de dados com o objetivo de organizar, separar o conteúdo e integrá-lo com outras linguagens. Possui uma sintaxe básica, combinando com outros padrões, torna-se possível separar o conteúdo do documento e reutilizar em outras aplicações para diferentes propósitos (PEREIRA, 2009).

O código XML encontrado no arquivo ".circ" possui uma *tag* chamada "<project>" contendo todas a informações relacionadas à construção do circuito no *software* Logisim. Dentro dessa, existem diversas outras *tags*, algumas podem ser empregadas como campos e opções que foram destacados, informações extras sobre o circuito ou para fornecer uma informação inicial antes do arquivo ser aberto totalmente. Entre elas, há a *tag* "<circuit>", que contém várias outras *tags* que representam os componentes do circuito simulado.

Dentro de "<circuit>" há a tag "<wire>", que indica um segmento de fio. Ela possui duas propriedades: *from* e *to*, cada uma seguida de seus valores, que são coordenadas de um plano. O plano na área de desenho do Logisim tem sua origem no canto superior esquerdo e as coordenadas aumentam de valor quando deslocam para a direita ou para baixo. A primeira coordenada indica o deslocamento na horizontal e a segunda coordenada o deslocamento na vertical. A propriedade *from* diz o ponto onde o segmento de fio começa, enquanto a propriedade *to* diz o ponto em que o segmento termina.

Quadro 4 - Exemplo de tag <wire> usada na descrição de segmento de fio

```
<wire from="(280,160)" to="(320,160)"/>
```

Fonte: Elaborado pelo autor (2021)

Dentro de "<circuit>" também tem a tag "<comp>", indicando os outros componentes do circuito, como portas, entradas, saídas e constantes. A tag possui a propriedade *loc*, que possui um valor de coordenada, semelhante ao *from* e o *to* de "<wire>", mas aqui indica o ponto de saída, para os pinos entrada, constantes e portas, ou ponto de entrada para pinos de saída. Outra propriedade é o *name*, o seu valor identifica a que componente aquelas informações pertencem. O valor é "Constant" para constantes, "Pin" para pinos de entrada e de saída (mais à frente será mostrado como são diferenciados) e "AND Gate" para a porta *AND*, por exemplo (no caso de outras portas o valor é o nome da porta seguida do texto "Gate").

Internamente nas tags "<comp>" podem existir as *tags* "<a>", indicando outras características do componente. Essa *tag* possui a propriedade *name*, cujo valor indica qual a

característica está sendo informada e a propriedade *val*, informando o valor referente a essa característica.

Se uma dessas *tags* adicionais tem o texto "*facing*" em *name*, a informação em *val* identifica a direção em que a entrada (ou saída) do componente estará posicionada. Se a informação for "*west*" a entrada (ou saída) estará no lado esquerdo do componente, se for "*north*" estará em cima, se for "*south*" está em baixo e se não tiver essa *tag* adicional, o componente terá, por padrão, sua entrada (ou saída) no seu lado direito.

Uma propriedade com "*label*" em *name*, aponta que tem um rótulo associado ao componente, o conteúdo em *val* é o texto do rótulo. Durante a transpilação, é esse rótulo que será usado para associar o pino na placa microcontrolada a uma entrada ou saída do circuito simulado.

Caso um componente identificado como pino possua uma *tag* adicional com o texto "output" em *name* e o "true" em *val*, esse pino é distinguido como pino de saída. Caso não tenha essa adicional, o pino é distinguido como pino de entrada. No componente apontado como constante pode haver uma tag adicional com o conteúdo "*value*" em *name* e "0x0" em *val*. Se existir, o componente é reconhecido como uma constante 0, se não existir é uma constante 1, por padrão.

Quadro 5 - Exemplo de tag <comp> usada na descrição de um pino de saída

```
<comp lib="0" loc="(380,170)" name="Pin">
    <a name="facing" val="west"/>
    <a name="output" val="true"/>
    <a name="label" val="saida_13"/>
    <a name="labelloc" val="north"/>
</comp>
```

Fonte: Elaborado pelo autor (2021)

Quadro 6 - Exemplo de tag <comp> usada na descrição de uma constante 0

```
<comp lib="0" loc="(120,160)" name="Constant">
    <a name="value" val="0x0"/>
</comp>
```

Fonte: Elaborado pelo autor (2021)

No Logisim são atribuídas três dimensões: pequeno, médio e grande. A propriedade com "*size*" em *name*, diferencia a dimensão dos componentes e ocorre somente nos

identificados como portas. O conteúdo em *val* indica a dimensão, 30 para portas pequenas, 70 para grandes e por padrão, nas portas médias não aparecem essa *tag* adicional. Para a porta *NOT* é um pouco diferente, é 20 para as portas pequenas, as grandes são padronizadas sem aparecer a *tag* e não existe as de dimensão média.

Quadro 7 - Exemplo de tag <comp> usada na descrição de um pino de saída

```
<comp lib="1" loc="(270,150)" name="AND Gate">
    <a name="size" val="30"/>
    <a name="label" val="AND"/>
</comp>
```

Fonte: Elaborado pelo autor (2021)

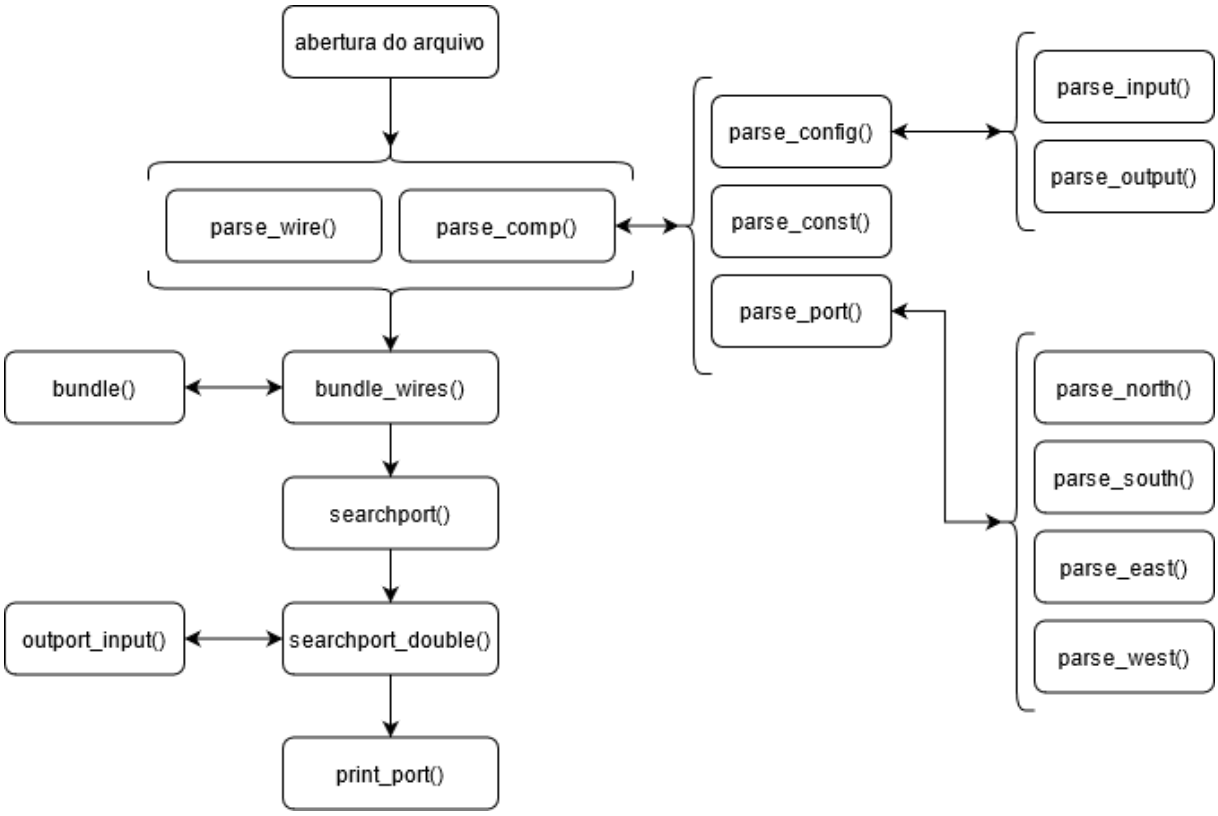
As coordenadas das entradas das portas não são descritas no código XML, mas esses valores de dimensão podem ser usados para descobrir a distância que o ponto de saída se encontra do ponto de entrada (na vertical ou na horizontal, depende da direção que se encontra a porta). No caso de portas diferentes de *NOT*, que possuem duas entradas, a distância é entre o ponto de saída e um ponto equidistante aos pontos das duas entradas.

Diversas *tags*, não discutidas anteriormente, não foram utilizadas e são ignoradas no processo de transpilação.

3.3 DESCRIÇÃO DO ALGORITMO DESENVOLVIDO

O algoritmo responsável por realizar a transpilação foi construído em linguagem C, linguagem também utilizada pelos microcontroladores, usando diversas métodos e estruturas de armazenamento de dados. Na Figura 9 é apresentado um fluxograma dos métodos implementados e em seguida é explicado a funcionalidade de cada um.

Figura 9 - Métodos implementados no algoritmo desenvolvido



Fonte: Elaborada pelo autor (2021)

As primeiras instruções a serem executadas são responsáveis por abrir o arquivo ".circ", para a leitura dos dados e algumas estruturas de repetição e de decisão para direcionar a execução do código a medida em que os dados são lidos. Caso o conteúdo lido no momento indica a *tag* "<wire>" o método **parse_wire()** é chamado, que será tratado adequadamente as características do segmento de fio. Caso o conteúdo lido indica o início da *tag* "<comp>" é chamado o método **parse_comp()** que irá tratar as características do componente.

No método **parse_wire()** são lidos os pontos de início e fim do segmento e são salvos em uma estrutura de dados, junto com um identificador do grupo de fios ao qual esse segmento pertence (essa variável do identificador de grupo é inicialmente preenchida por "NULL").

No **parse_comp()** será observado que tipo de componente a *tag* "<comp>" se refere e direcionará a execução para o método especifico do componente. Se for um pino, o método **parse_config()**, uma constante, o método **parse_const()** e se for uma porta o método **parse_port()**.

Em **parse_config()** os conteúdos das *tags* adicionais são obtidos, como o ponto de entrada (ou de saída) e o número identificando o pino físico do microcontrolador, em seguida é convocado o método específico do tipo de pino passando esses dados como parâmetro. Se for

um pino de entrada, é chamado o **parse_input()**, se for um pino de saída, o **parse_output()**. Os métodos são semelhantes, adicionando as informações em estruturas de dados, mas cada uma tem um contador da quantidade de pinos que são armazenados.

Em **parse_const()** as *tags* adicionais são lidas e é verificado qual tipo da constante. O ponto da saída e o valor da constante são salvos na estrutura de dados.

O método **parse_port()** também irá ler o conteúdo das *tags* adicionais, armazenar o tipo e o ponto da saída da porta na estrutura de dados e em seguida definir os pontos das entradas da porta. Na descrição do circuito em código XML esses pontos das entradas não são informados, mas é necessário determiná-los para que o algoritmo saiba quais outros componentes estão conectados a essas entradas. Se a direção da saída da porta estiver para cima, o método **inputs_north()** é chamado, se a direção estiver para baixo é chamado o **inputs_south()**, se a direção for para direita é chamado o **inputs_east()** e, por último, se a direção for para a esquerda é chamado **inputs_west()**. São passados como parâmetros o ponto de saída, o tipo de porta, a dimensão e um identificador para saber a localização na estrutura de dados que serão salvos os pontos das entradas.

No método **inputs_north()** é verificado primeiro o tipo da porta, pois algumas portas possuem uma distância diferente da entrada até a saída. Depois é verificado a dimensão da porta, nas portas pequenas a distância entre as duas entradas é menor que na porta grande e média. Para definir a coordenada na horizontal das entradas, é suficiente usar a coordenada na horizontal da saída e deslocar a esquerda em 20 unidades para uma entrada e deslocar a direita em 20 unidades para a outra entrada, se for uma porta pequena esses deslocamentos serão em 10 unidades. Para a coordenada na vertical das entradas, é suficiente usar a coordenada na vertical da saída e deslocar para baixo o valor da dimensão da porta, porém, para as portas *NAND, NOR* e *XOR* o deslocamento é acrescentado em 10 unidades e para a porta *XNOR* é acrescentado em 20 unidades.

Os métodos **inputs_south()**, **inputs_east()** e **inputs_west()** são bastante semelhantes ao **inputs_north()**. O que muda é que, por causa da posição da porta, deslocamentos na horizontal serão aplicados na vertical e vice-versa, ou os deslocamentos serão decrementados invés de acrescentados.

Após tudo isso e o arquivo ".circ" lido totalmente, é iniciado o agrupamento dos segmentos de fios com o método **bundle_wires()**. Nesse método, os pontos de saída (ou de entrada) de todos os pinos de entrada, de saída e constantes armazenados são buscados para agrupar os segmentos de fios e referencia-los a esses componentes. Quando simulado, um grande grupo de segmentos de fio que estão conectados serão tratados como um único fio.

Segmentos de fios que conectam uma porta a outra também serão agrupados para serem identificados como um fio só.

O ponto e uma *string* informando o tipo de agrupamento são passados como parâmetros para o método **bundle()**. Se o agrupamento se refere a um pino de entrada a *string* passada será "IN_PORT(%d)", se for um pino de saída será "OUT_PORT(%d)", onde "%d" nos dois casos é substituído pelo pino físico do microcontrolador que simulará essa entrada ou saída. Se for uma constante a *string* passada será "IN_CONST_%d" onde "%d" é substituído pelo valor da constante. Se for um fio que conecta uma porta a outra é passado na *string* um número, esse número inicia em 0 e aumenta em uma unidade à medida que novos grupos aparecem.

Esse método compara o ponto passado como parâmetro com os pontos do início e do fim do segmento de fio. Se com o ponto do início que são iguais, por exemplo, esse segmento de fio armazena o valor da *string* do tipo de agrupamento, que antes estava salvo como "NULL", e inicia uma nova comparação, recursivamente, utilizando o ponto do fim do segmento. Se o ponto do fim que são iguais, a nova comparação utilizará o ponto do início.

Depois de agrupar os segmentos de fio, os dados armazenados das portas serão modificados pelo método **searchport()**, para ficarem de acordo com o código transpilado apresentado na seção 4.1. Os pontos de entrada da porta são comparados com os pontos dos pinos de entrada, dos fios e das constantes. Se for igual a um pino de entrada, o ponto é substituído pelo texto "IN_PORT(%d)" com "%d" substituído pelo pino físico do microcontrolador escolhido para a simulação. Se for igual a uma constante, o ponto é substituído pelo texto "IN_CONST_%d" com %d substituído pelo valor da constante. Se for igual a um segmento de fio, o ponto é substituído pelo identificador do grupo desse segmento. O ponto de saída da porta é comparado com os pontos dos pinos de saída e dos fios. Se for igual a um pino de saída, o ponto é substituído pelo texto "OUT_PORT(%d)" com "%d" substituído pelo pino físico do microcontrolador. Se também for igual a um segmento de fio, o ponto é substituído pelo identificador do grupo.

Em alguns casos, os circuitos possuem a saída de uma porta ligada a um pino de saída e, ao mesmo tempo, ligado a entrada de outra porta. Na descrição do circuito no algoritmo, a outra porta vai estar com um pino de saída ligado à sua entrada. Nesses casos é preciso duplicar a primeira porta e considerar uma delas ligada somente ao pino de saída e a outra somente a entrada de outra porta. O método **searchport_double()** prepara uma estrutura de repetição para que sejam verificadas todas as portas. O método **outport_input()** é responsável por verificar, a cada iteração, se existe alguma porta com uma das entradas conectadas a um pino de saída. Se existir, a entrada da porta será conectada a um novo grupo de segmentos que identifica um fio,

e a porta, que possui a saída conectada ao pino de saída, será duplicada, mas com a saída conectada ao grupo de segmentos que foi criado anteriormente.

Após essas verificações, é iniciado a escrita do arquivo “.ino”, com instruções necessárias para que o arquivo seja executado adequadamente e informações referentes ao circuito que será simulado. Os dados das portas são escritos no arquivo pelo método **print_port()**, organizando de acordo com o que foi mostrado no Quadro 2 na seção 4.1.

Esse método, embora não tenha alguma demonstração formal de sua corretude, mostrou-se em testes robustos o suficiente para todos os casos experimentados inclusive casos que apresentavam laços. Além disso, mesmo para os maiores circuitos testados, o tempo de execução foi bastante curto.

4 RESULTADOS E DISCUSSÕES

Para realizar os testes das simulações foram transpilados e compilados para o Arduino alguns circuitos digitais projetados no Logisim. Foram projetados alguns circuitos sem aplicação conhecida, usados apenas para averiguar algumas condições que podem surgir na projeção dos circuitos. Essas condições são explicadas ao apresentar os resultados dos respectivos testes. Foram projetados também alguns circuitos conhecidos, como codificador binário-Gray, somador completo e *flip-flop* RS com *clock*. As projeções também serviram para gerar tabelas verdade para cada circuito e com isso, comparar os valores obtidos no Logisim com os resultados obtidos nas simulações com Arduino. Como último teste, buscando implementar uma aplicação real, foi projetado um semáforo utilizando *flip-flops* D.

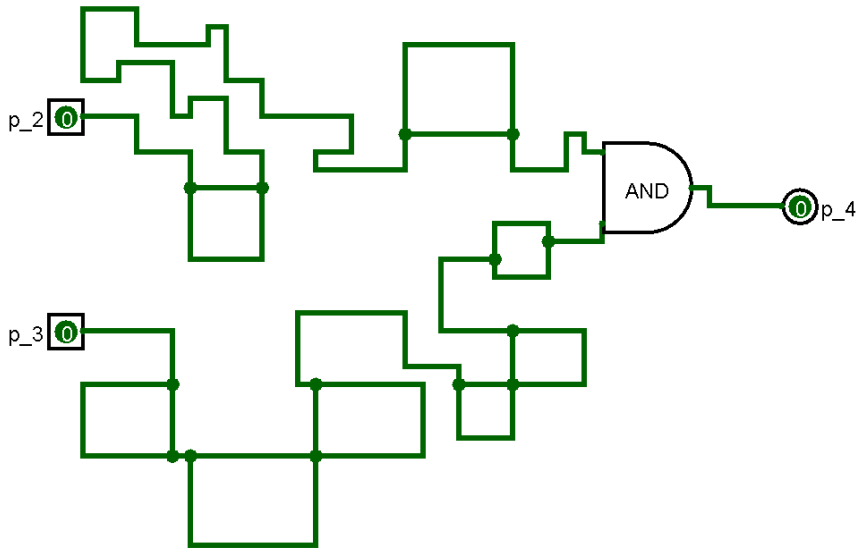
Para obter os resultados dos testes foi realizado uma pequena adição de instruções nos arquivos ".ino", mas sem alterar as instruções especificas dos circuitos transpilados ou comprometer a execução das simulações. Essas instruções permitiam exibir os valores obtidos nas saídas do circuito no monitor serial da IDE do Arduino. Assim, é possível reduzir a quantidade de componentes para montar o circuito e os resultados serão mostrados diretamente na tela.

4.1 CIRCUITOS SEM APLICAÇÃO USUAL

4.1.1 Circuito para teste de segmentos de fios

É um circuito simples, mas projetado para verificar e garantir que, mesmo usando muitos segmentos de fios e com ramos conectados no próprio fio, o transpilador irá agrupar corretamente cada segmento como um único fio. E com o circuito da Figura 10, por exemplo, descrever adequadamente que as entradas da porta *AND* estão conectadas aos pinos de entrada "p_2" e "p_3". O Quadro 8 apresenta como ficou a descrição do circuito gerada no processo de transpilação e a Tabela 7 compara os resultados do circuito no Logisim com o simulado no Arduino.

Figura 10 - Circuito para analisar o agrupamento dos segmentos de fios



Fonte: Elaborada pelo autor (2021)

Quadro 8 - Descrição do circuito para analisar o agrupamento dos segmentos de fios

```
void setup(){
        pinMode(2, INPUT);
        pinMode(3, INPUT);

        pinMode(4, OUTPUT);

        portas[0].tipo = AND;
        portas[0].inp1 = IN_PORT(2);
        portas[0].inp2 = IN_PORT(3);
        portas[0].out = OUT_PORT(4);

        c.portas = portas;
        c.fios = fios;
        c.n_portas = 1;
}
```

Fonte: Elaborado pelo autor (2021)

Tabela 7 - Comparação de resultados do circuito simulado no Logisim e no Arduino

		(tabela verdade)	(simulador)
p_2	p_3	p_4	p_4
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

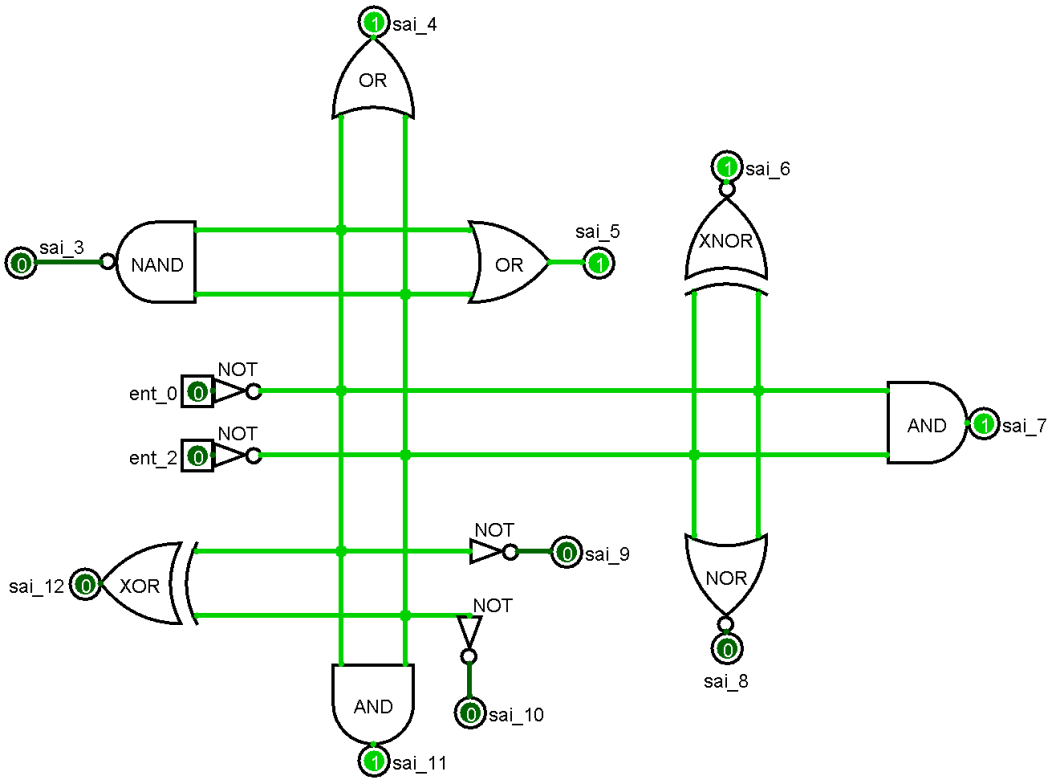
Fonte: Elaborada pelo autor (2021)

O tamanho do arquivo gerado pelo Logisim foi de 8.807 bytes. 1 porta foi descrita pelo transpilador gerando um arquivo “.ino” de 3.555 bytes.

4.1.2 Circuito para teste de ramificações de fios e pinos conectados diretamente nas portas

O circuito na Figura 11 foi projetado para verificar ramificações dos fios para diferentes portas. Também verifica os pinos de entrada conectados diretamente a entradas das portas e pinos de saída diretamente a saídas das portas, sem um segmento de fio entre esses componentes.

Figura 11 - Circuito para analisar ramos de fios e portas com pinos conectados diretamente



Fonte: Elaborada pelo autor (2021)

O Quadro 9 apresenta a descrição do circuito gerada pelo processo de transpilação. As Tabelas 8 e 9 contêm as tabelas verdade dos resultados obtidos do circuito simulado no Logisim e no Arduino.

Quadro 9 - Descrição do circuito para analisar ramos de fios e portas com pinos conectados diretamente

```
void setup(){
        pinMode(0, INPUT);
        pinMode(2, INPUT);

        pinMode(5, OUTPUT);
        pinMode(11, OUTPUT);
        pinMode(12, OUTPUT);
        pinMode(6, OUTPUT);
        pinMode(8, OUTPUT);
        pinMode(3, OUTPUT);
        pinMode(4, OUTPUT);
        pinMode(10, OUTPUT);
        pinMode(7, OUTPUT);
        pinMode(9, OUTPUT);

        portas[0].tipo = OR;
        portas[0].inp1 = 0;
        portas[0].inp2 = 1;
        portas[0].out = OUT_PORT(5);

        portas[1].tipo = NOT;
        portas[1].inp1 = 1;
        portas[1].out = OUT_PORT(10);

        portas[2].tipo = AND;
        portas[2].inp1 = 0;
        portas[2].inp2 = 1;
        portas[2].out = OUT_PORT(7);

        portas[3].tipo = AND;
        portas[3].inp1 = 0;
        portas[3].inp2 = 1;
        portas[3].out = OUT_PORT(11);

        portas[4].tipo = NOT;
        portas[4].inp1 = IN_PORT(0);
        portas[4].out = 0;

        portas[5].tipo = NOT;
```

```
        portas[5].inp1 = IN_PORT(2);
        portas[5].out = 1;

        portas[6].tipo = XOR;
        portas[6].inp1 = 0;
        portas[6].inp2 = 1;
        portas[6].out = OUT_PORT(12);

        portas[7].tipo = XNOR;
        portas[7].inp1 = 1;
        portas[7].inp2 = 0;
        portas[7].out = OUT_PORT(6);

        portas[8].tipo = NOT;
        portas[8].inp1 = 0;
        portas[8].out = OUT_PORT(9);

        portas[9].tipo = NOR;
        portas[9].inp1 = 1;
        portas[9].inp2 = 0;
        portas[9].out = OUT_PORT(8);

        portas[10].tipo = NAND;
        portas[10].inp1 = 0;
        portas[10].inp2 = 1;
        portas[10].out = OUT_PORT(3);

        portas[11].tipo = OR;
        portas[11].inp1 = 0;
        portas[11].inp2 = 1;
        portas[11].out = OUT_PORT(4);

        c.portas = portas;
        c.fios = fios;
        c.n_portas = 12;
}
```

Fonte: Elaborado pelo autor (2021)

Tabela 8 - Resultados do circuito simulado no Logisim

		(tabela verdade)									
ent_0	**ent_2**	**sai_3**	**sai_4**	**sai_5**	**sai_6**	**sai_7**	**sai_8**	**sai_9**	**sai_10**	**sai_11**	**sai_12**
0	0	0	1	1	1	1	0	0	0	1	0
0	1	1	1	1	0	0	0	0	1	0	1
1	0	1	1	1	0	0	0	1	0	0	1
1	1	1	0	0	1	0	1	1	1	0	0

Fonte: Elaborada pelo autor (2021)

Tabela 9 - Resultados do circuito simulado no Arduino

		(simulador)									
ent_0	**ent_2**	**sai_3**	**sai_4**	**sai_5**	**sai_6**	**sai_7**	**sai_8**	**sai_9**	**sai_10**	**sai_11**	**sai_12**
0	0	0	1	1	1	1	0	0	0	1	0
0	1	1	1	1	0	0	0	0	1	0	1
1	0	1	1	1	0	0	0	1	0	0	1
1	1	1	0	0	1	0	1	1	1	0	0

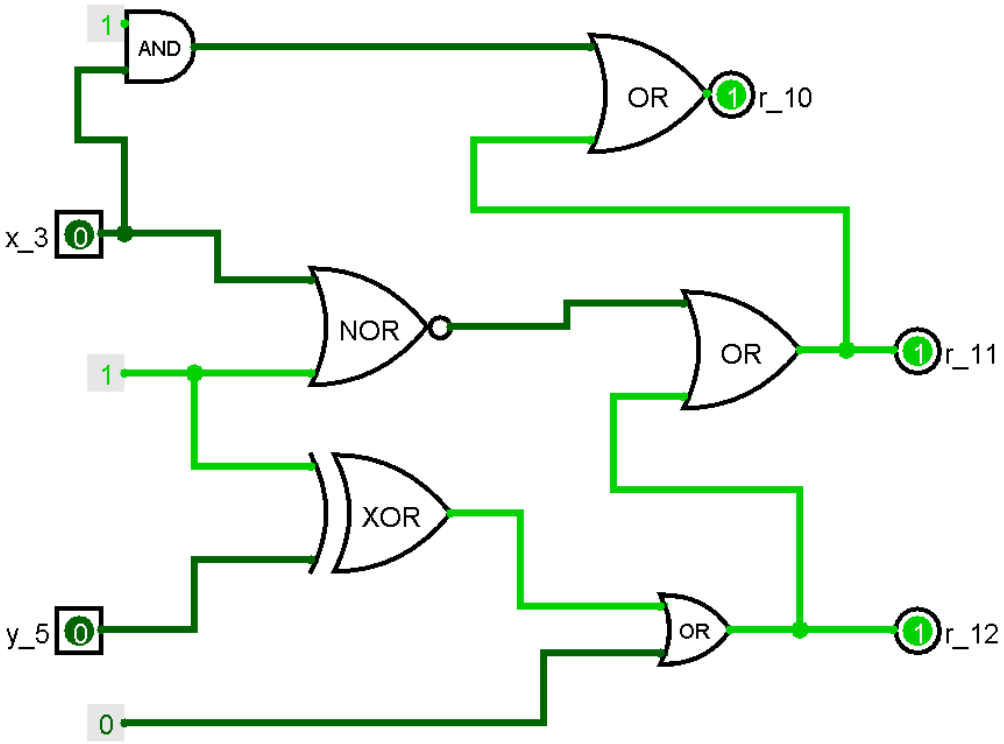
Fonte: Elaborada pelo autor (2021)

O tamanho do arquivo gerado pelo Logisim foi de 9.452 bytes. 12 portas foram descritas pelo transpilador gerando um arquivo “.ino” de 4.772 bytes.

4.1.3 Circuito para teste de constantes e duplicação de portas

Este circuito contém outros detalhes para serem analisados. O uso das constantes e a duplicação de portas, pois os pinos de saída “r_11” e “r_12” no circuito da Figura 12 também são conectados a entradas de outras portas. O Quadro 10 apresenta a descrição do circuito após o processo de transpilação. A tabela 10 apresenta a comparação de resultados do circuito simulado no Logisim e no Arduino.

Figura 12 - Circuito para analisar uso das constantes e duplicação de portas



Fonte: Elaborada pelo autor (2021)

Quadro 10 - Descrição do circuito para analisar uso das constantes e duplicação de portas

```
void setup(){
        pinMode(3, INPUT);
        pinMode(5, INPUT);

        pinMode(10, OUTPUT);
        pinMode(11, OUTPUT);
        pinMode(12, OUTPUT);

        portas[0].tipo = NOR;
        portas[0].inp1 = IN_PORT(3);
        portas[0].inp2 = IN_CONST_1;
        portas[0].out = 1;

        portas[1].tipo = OR;
        portas[1].inp1 = 1;
        portas[1].inp2 = 3;
        portas[1].out = OUT_PORT(11);

        portas[2].tipo = AND;
        portas[2].inp1 = IN_CONST_1;
```

```
        portas[2].inp2 = IN_PORT(3);
        portas[2].out = 2;

        portas[3].tipo = OR;
        portas[3].inp1 = 2;
        portas[3].inp2 = 4;
        portas[3].out = OUT_PORT(10);

        portas[4].tipo = XOR;
        portas[4].inp1 = IN_CONST_1;
        portas[4].inp2 = IN_PORT(5);
        portas[4].out = 0;

        portas[5].tipo = OR;
        portas[5].inp1 = 0;
        portas[5].inp2 = IN_CONST_0;
        portas[5].out = OUT_PORT(12);

        portas[6].tipo = OR;
        portas[6].inp1 = 0;
        portas[6].inp2 = IN_CONST_0;
        portas[6].out = 3;

        portas[7].tipo = OR;
        portas[7].inp1 = 1;
        portas[7].inp2 = 3;
        portas[7].out = 4;

        c.portas = portas;
        c.fios = fios;
        c.n_portas = 8;
}
```

Fonte: Elaborado pelo autor (2021)

Tabela 10 - Comparação de resultados do circuito para uso de constantes e duplicação de portas simulado no Logisim e no Arduino

		(tabela verdade)			**(simulador)**		
x_3	**y_5**	**r_10**	**r_11**	**r_12**	**r_10**	**r_11**	**r_12**
0	0	1	1	1	1	1	1
0	1	0	0	0	0	0	0
1	0	1	1	1	1	1	1
1	1	1	0	0	1	0	0

Fonte: Elaborada pelo autor (2021)

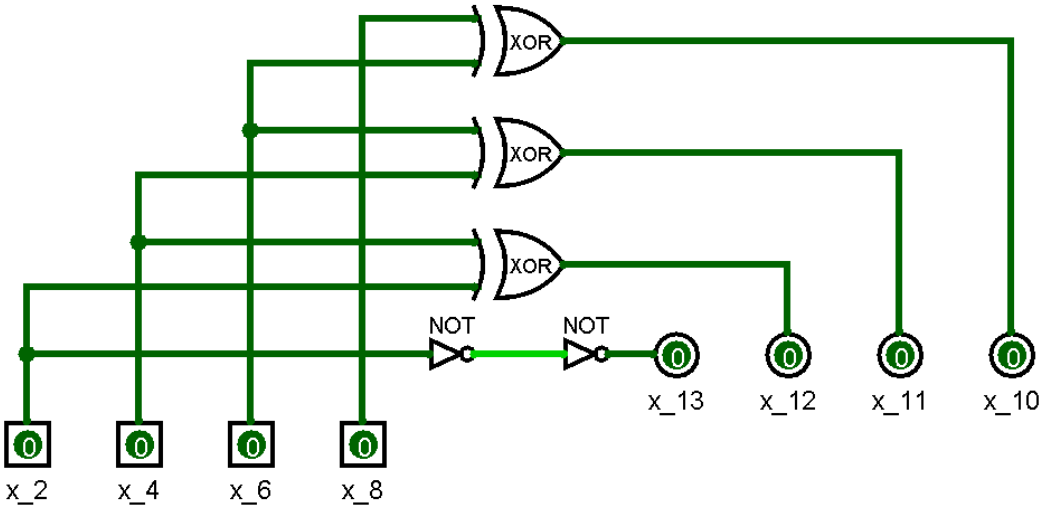
O tamanho do arquivo gerado pelo Logisim foi de 7.889 bytes. 8 portas foram descritas pelo transpilador gerando um arquivo “.ino” de 4.311 bytes.

4.2 CODIFICADOR BINÁRIO-GRAY

Analisando uma sequência de contagem binária, em vários momentos muitos bits são alterados de uma vez para dar sequência a contagem. Nesses casos a situação pode ser mal interpretada e não necessariamente ter ocorrido a sucessão do número.

O código Gray representa uma sequência de números, onde a única característica que distingue os números no código Gray de um código binário é que apenas um *bit* muda entre dois números sucessivos, assim reduz a má interpretação das mudanças nas entradas (TOCCI; WIDMER; MOSS, 2011). O circuito é representado na Figura 13 e os resultados da simulação do circuito se encontram na Tabela 11. O Quadro 11 apresenta a descrição do circuito depois do processo de transpilação.

Figura 13 - Circuito codificador binário-Gray usado na simulação



Fonte: Elaborada pelo autor (2021)

Quadro 11 - Descrição do circuito codificador binário-Gray usado na simulação

```
void setup(){
        pinMode(6, INPUT);
        pinMode(2, INPUT);
        pinMode(8, INPUT);
        pinMode(4, INPUT);

        pinMode(11, OUTPUT);
        pinMode(13, OUTPUT);
        pinMode(10, OUTPUT);
        pinMode(12, OUTPUT);

        portas[0].tipo = NOT;
        portas[0].inp1 = 0;
        portas[0].out = OUT_PORT(13);

        portas[1].tipo = XOR;
        portas[1].inp1 = IN_PORT(8);
        portas[1].inp2 = IN_PORT(6);
        portas[1].out = OUT_PORT(10);

        portas[2].tipo = NOT;
        portas[2].inp1 = IN_PORT(2);
        portas[2].out = 0;

        portas[3].tipo = XOR;
        portas[3].inp1 = IN_PORT(6);
        portas[3].inp2 = IN_PORT(4);
        portas[3].out = OUT_PORT(11);

        portas[4].tipo = XOR;
        portas[4].inp1 = IN_PORT(4);
        portas[4].inp2 = IN_PORT(2);
        portas[4].out = OUT_PORT(12);

        c.portas = portas;
        c.fios = fios;
        c.n_portas = 5;
}
```

Fonte: Elaborado pelo autor (2021)

Tabela 11 - Comparação dos resultados do circuito codificador binário-Gray simulado no Logisim e no Arduino

				(tabela verdade)				(simulador)			
x_2	**x_4**	**x_6**	**x_8**	**x_13**	**x_12**	**x_11**	**x_10**	**x_13**	**x_12**	**x_11**	**x_10**
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	0	1	0
0	1	0	0	0	1	1	0	0	1	1	0
0	1	0	1	0	1	1	1	0	1	1	1
0	1	1	0	0	1	0	1	0	1	0	1
0	1	1	1	0	1	0	0	0	1	0	0
1	0	0	0	1	1	0	0	1	1	0	0
1	0	0	1	1	1	0	1	1	1	0	1
1	0	1	0	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	0	1	1	1	0
1	1	0	0	1	0	1	0	1	0	1	0
1	1	0	1	1	0	1	1	1	0	1	1
1	1	1	0	1	0	0	1	1	0	0	1
1	1	1	1	1	0	0	0	1	0	0	0

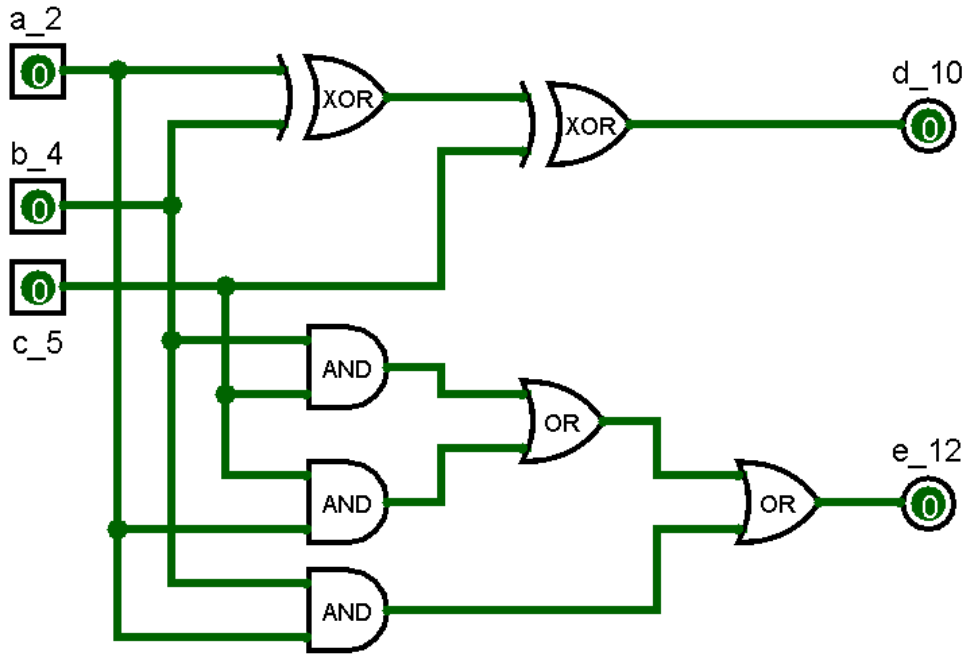
Fonte: Elaborada pelo autor (2021)

O circuito codificador binário-Gray foi utilizados nos testes por ser um circuito bastante conhecido e utilizado no desenvolvimento de projetos reais, tornando-se importante sua validação. O tamanho do arquivo gerado pelo Logisim foi de 7.935 bytes. 5 portas foram descritas pelo transpilador gerando um arquivo “.ino” de 4.066 bytes.

4.3 SOMADOR COMPLETO

O somador completo faz a adição de dois números binários de um *bit* cada com o *bit* de entrada de transporte (*Carry in*). Ele gera um *bit* de soma e um de saída de transporte (*Carry out*) como *bits* de saída. Vários somadores completos combinados podem realizar a soma de dois números de vários *bits*. O circuito é representado na Figura 14 e os resultados da simulação do circuito se encontram na Tabela 12. A entrada “c_5” representa o *Carry in* e a saída “e_12” representa *Carry out*. A descrição do circuito gerada pela transpilação é exibida no Quadro 12.

Figura 14 - Circuito somador completo usado na simulação



Fonte: Elaborada pelo autor (2021)

Quadro 12 - Descrição do circuito somador completo usado na simulação

```
void setup(){
	pinMode(4, INPUT);
	pinMode(5, INPUT);
	pinMode(2, INPUT);

	pinMode(12, OUTPUT);
	pinMode(10, OUTPUT);

	portas[0].tipo = AND;
	portas[0].inp1 = IN_PORT(4);
	portas[0].inp2 = IN_PORT(5);
	portas[0].out = 1;

	portas[1].tipo = AND;
	portas[1].inp1 = IN_PORT(5);
	portas[1].inp2 = IN_PORT(2);
	portas[1].out = 2;

	portas[2].tipo = XOR;
	portas[2].inp1 = IN_PORT(2);
	portas[2].inp2 = IN_PORT(4);
	portas[2].out = 0;

	portas[3].tipo = OR;
	portas[3].inp1 = 3;
	portas[3].inp2 = 4;
	portas[3].out = OUT_PORT(12);

	portas[4].tipo = OR;
	portas[4].inp1 = 1;
	portas[4].inp2 = 2;
	portas[4].out = 3;

	portas[5].tipo = AND;
	portas[5].inp1 = IN_PORT(4);
	portas[5].inp2 = IN_PORT(2);
	portas[5].out = 4;

	portas[6].tipo = XOR;
	portas[6].inp1 = 0;
	portas[6].inp2 = IN_PORT(5);
	portas[6].out = OUT_PORT(10);

	c.portas = portas;
	c.fios = fios;
	c.n_portas = 7;
}
```

Fonte: Elaborado pelo autor (2021)

Tabela 12 - Comparação dos resultados do circuito somador completo simulado no Logisim e no Arduino

			(tabela verdade)		(simulador)	
A	**B**	**Cin**	**soma**	**Cout**	**soma**	**Cout**
0	0	0	0	0	0	0
0	0	1	1	0	1	0
0	1	0	1	0	1	0
0	1	1	0	1	0	1
1	0	0	1	0	1	0
1	0	1	0	1	0	1
1	1	0	0	1	0	1
1	1	1	1	1	1	1

Fonte: Elaborada pelo autor (2021)

O somador completo é outro circuito bastante conhecido e utilizado em projetos reais de circuitos digitais, garantir seu funcionamento se torna importante para a conclusão desse projeto. O tamanho do arquivo gerado pelo Logisim foi de 8.189 bytes. 7 portas foram descritas pelo transpilador gerando um arquivo ".ino" de 4.219 bytes.

4.4 *FLIP-FLOP* SR COM *CLOCK*

Flip-flop é um tipo de circuito que armazena um único *bit*, em que esse dado poderá ser enviado através de uma saída Q. O *clock* é uma entrada que indica quando o *flip-flop* pode alterar o valor armazenado. Quando seu valor for alterado de 0 para 1, o valor no *flip-flop* mudará de acordo com o seu tipo. No tipo SR, quando houver a variação do nível do *clock*, o valor guardado no *flip-flop* será mantido se as entradas R (*Reset*) e S (*Set*) forem iguais a 0. Se R for 1, durante a variação, o valor guardado se tornará 0, se a entrada S que for 1, o valor armazenado é alterado para 1. Se R e S forem ambas 1, o comportamento não é especificado, se tornando uma opção que deve ser evitada ao se trabalhar com *flip-flops* (BURCH, 2021).

Tabela 13 - Comportamento do *flip-flop* SR

Set	**Reset**	**Q**
0	0	Q
0	1	0
1	0	1
1	1	?

Fonte: Elaborada pelo autor (2021)

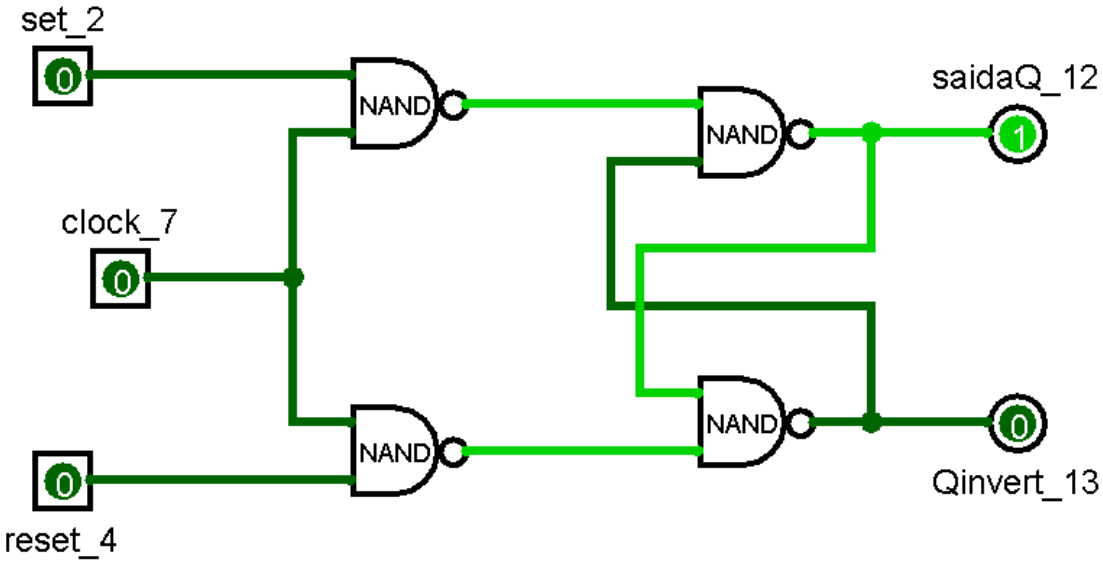
Tabela 14 - Comportamento do *clock* no circuito *flip-flop* SR

Clock	**Saída**
0	Q
1	FF SR

Fonte: Elaborada pelo autor (2021)

A Figura 15 apresenta o circuito flip-flop SR com clock e o Quadro 13 a descrição do circuito após o processo de transpilação.

Figura 15 - Circuito *flip-flop* SR com *clock* usado na simulação



Fonte: Elaborada pelo autor (2021)

Quadro 13 - Descrição do circuito *flip-flop* SR com *clock* usado na simulação

```
void setup(){
        pinMode(4, INPUT);
        pinMode(7, INPUT);
        pinMode(2, INPUT);

        pinMode(12, OUTPUT);
        pinMode(13, OUTPUT);

        portas[0].tipo = NAND;
        portas[0].inp1 = 2;
        portas[0].inp2 = 1;
        portas[0].out = OUT_PORT(13);
```

```
portas[1].tipo = NAND;
portas[1].inp1 = 0;
portas[1].inp2 = 3;
portas[1].out = OUT_PORT(12);

portas[2].tipo = NAND;
portas[2].inp1 = IN_PORT(7);
portas[2].inp2 = IN_PORT(4);
portas[2].out = 1;

portas[3].tipo = NAND;
portas[3].inp1 = IN_PORT(2);
portas[3].inp2 = IN_PORT(7);
portas[3].out = 0;

portas[4].tipo = NAND;
portas[4].inp1 = 0;
portas[4].inp2 = 3;
portas[4].out = 2;

portas[5].tipo = NAND;
portas[5].inp1 = 2;
portas[5].inp2 = 1;
portas[5].out = 3;
```

Fonte: Elaborado pelo autor (2021)

Tabela 15 - Comparação dos resultados do circuito *flip-flop* SR com *clock* simulado no Logisim e no Arduino

			(tabela verdade)		**(simulador)**	
set	***clock***	***reset***	**saidaQ**	**Qinvert**	**saidaQ**	**Qinvert**
0	0	0	saidaQ	Qinvert	saidaQ	Qinvert
0	0	1	saidaQ	Qinvert	saidaQ	Qinvert
0	1	0	saidaQ	Qinvert	saidaQ	Qinvert
0	1	1	0	1	0	1
1	0	0	saidaQ	Qinvert	saidaQ	Qinvert
1	0	1	saidaQ	Qinvert	saidaQ	Qinvert
1	1	0	1	0	1	0
1	1	1	1	1	1	1

Fonte: Elaborada pelo autor (2021)

Os resultados “saidaQ” e “Qinvert”, na Tabela 15, indicam que não teve alguma alteração no estado que se encontrava o circuito, justamente nas condições em que o *clock* não

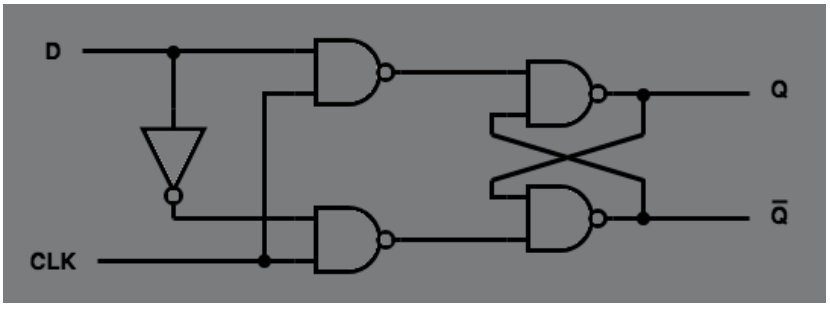
tinha variado e que as entradas “set” e ‘reset” estavam ambas desativadas, ou seja, se encontravam com valor 0. A última opção de entrada no teste não apresentou irregularidade alguma, mas durante algum projeto ainda deve ser evitado chegar nesse resultado. Na prática, em circuito real, essa configuração levaria a um estado indefinido ou oscilante.

O *flip-flop* SR com *clock* foi simulado para os testes por ser bastante utilizado nos projetos reais de circuitos digitais e por ser um pouco mais complexos que os outros circuitos testados. Importante para verificar que o transpilador pode ser usado para circuitos com complexidade semelhante aos *flip-flops*. O tamanho do arquivo gerado pelo Logisim foi de 6.986 bytes. O transpilador descreveu 6 portas para simulação gerando um arquivo “.ino” com tamanho de 4.091 bytes.

4.5 SEMÁFORO DE TRÂNSITO COM *FLIP-FLOPS* D

Algumas literaturas dão o significado de Dados (do inglês, *data*) para o D do *flip-flop* pelo fato de armazenar um bit de informação. Em outras dão o significado de Atraso (do inglês, *delay*) fazendo analogia ao comportamento de seus componentes, que se atrasam para realizar a mudança. O *flip-flop* armazena na saída Q o valor inserido na entrada D, mas o armazenamento irá ocorrer somente quando o *clock* possuir valor lógico 1 (PANTUZA, 2018). O circuito *flip-flop* D é exibido na Figura 16 e a Tabela 16 apresenta a tabela verdade do comportamento do circuito.

Figura 16 - Circuito *flip-flop* D



Fonte: (PANTUZA, 2018)

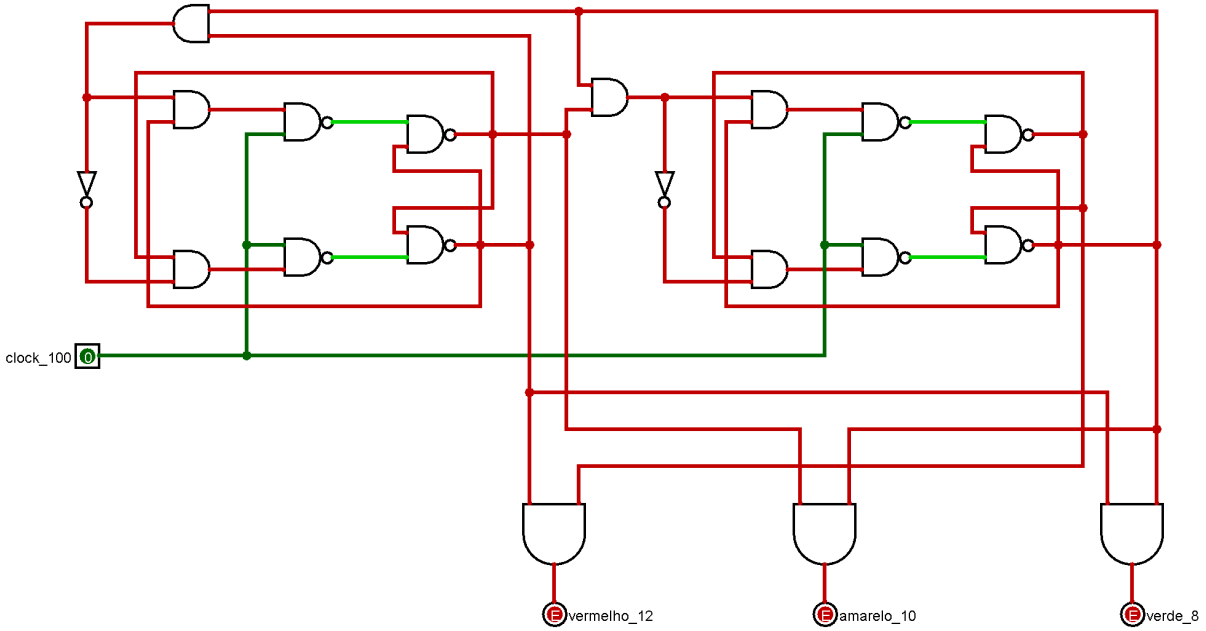
Tabela 16 - Comportamento do *flip-flop* D

clock	**D**	**Q**	$\overline{\mathbf{Q}}$
0	0	Q	$\overline{Q}$
0	1	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

Fonte: (PANTUZA, 2018)

O circuito desenvolvido para realizar a aplicação do semáforo é apresentado pela Figura 17. A entrada “clock_100” é definida como pino de entrada 100 para que o transpilador identifique que a transição de níveis realizadas pelo *clock* funcione automaticamente. Os pinos de saída 8, 10 e 12 são usados para acender LEDs que representam os sinais do semáforo, no qual o pino 8 identifica o sinal verde, o pino 10 o sinal amarelo e o pino 12 o sinal vermelho.

Figura 17 - Circuito do semáforo com *flip-flops* D usado para simulação



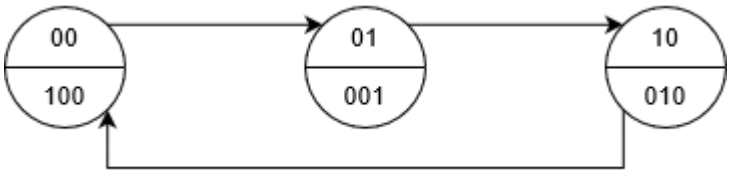
Fonte: Elaborada pelo autor (2021)

Com a máquina de estados finitos e a tabela de transição apresentadas pelas Figura 18 e Tabela 17, respectivamente, é possível compreender as transições que o circuito deve realizar para que a aplicação funcione corretamente. O círculo na Figura 18 identifica um estado em que o circuito pode apresentar durante seu funcionamento. A metade superior do círculo indica

os *bits* do estado interno do circuito, no caso, das saídas Q dos *flip-flops* enquanto a parte inferior indica os *bits* de saída do sistema. A seta aponta qual vai ser o próximo estado apresentado quando houver uma transição de estados.

Quando o par de *bits* for 00 no estado interno a saída do sistema de acender apenas o LED verde, identificado pelos *bits* 100. Quando o par for 01 no estado interno o LED que estará aceso é amarelo, identificado pelos *bits* 001. Quando o par de bits for 10 no estado interno o LED vermelho estará aceso e identificado pelos *bits* 010.

Figura 18 - Máquina de estados finitos do Semáforo



Fonte: Elaborada pelo autor (2021)

Analisando da esquerda pra direita o primeiro flip-flop tem a saída Q_1 e entrada D_1 enquanto o segundo tem a saída Q_2 e entrada D_2. Os pares de *bits* representados na Figura 18 são identificados pelos pares Q_2Q_1 na Tabela 17 e o par D_2D_1 identificam o próximo estado após a transição. As sequências de bits $O_3O_2O_1$ identificam a saída do sistema no estado atual.

Tabela 17 - Tabela de transição do semáforo

Q_2	**Q_1**	**D_2**	**D_1**	**O_3 (verde)**	**O_2 (vermelho)**	**O_1 (amarelo)**
0	0	0	1	1	0	0
0	1	1	0	0	0	1
1	0	0	0	0	1	0

Fonte: Elaborada pelo autor (2021)

O Quadro 14 apresenta a descrição do circuito do semáforo gerada pelo transpilador.

Quadro 14 - Descrição do circuito do semáforo usado na simulação

```
void setup(){
        clock = 1;
        iter = 0;

        pinMode(100, INPUT);

        pinMode(12, OUTPUT);
        pinMode(10, OUTPUT);
        pinMode(8, OUTPUT);

        portas[0].tipo = NOT;
        portas[0].inp1 = 6;
        portas[0].out = 7;

        portas[1].tipo = NAND;
        portas[1].inp1 = 11;
        portas[1].inp2 = IN_PORT(100);
        portas[1].out = 9;

        portas[2].tipo = AND;
        portas[2].inp1 = 4;
        portas[2].inp2 = 5;
        portas[2].out = OUT_PORT(12);

        portas[3].tipo = AND;
        portas[3].inp1 = 5;
        portas[3].inp2 = 7;
        portas[3].out = 2;

        portas[4].tipo = NOT;
        portas[4].inp1 = 8;
        portas[4].out = 13;

        portas[5].tipo = AND;
        portas[5].inp1 = 4;
        portas[5].inp2 = 1;
        portas[5].out = OUT_PORT(8);

        portas[6].tipo = AND;
        portas[6].inp1 = 6;
        portas[6].inp2 = 1;
        portas[6].out = 15;

        portas[7].tipo = NAND;
        portas[7].inp1 = IN_PORT(100);
```

```
portas[7].inp2 = 14;
portas[7].out = 0;

portas[8].tipo = NAND;
portas[8].inp1 = 9;
portas[8].inp2 = 4;
portas[8].out = 3;

portas[9].tipo = NAND;
portas[9].inp1 = 12;
portas[9].inp2 = 1;
portas[9].out = 5;

portas[10].tipo = AND;
portas[10].inp1 = 1;
portas[10].inp2 = 3;
portas[10].out = 6;

portas[11].tipo = AND;
portas[11].inp1 = 3;
portas[11].inp2 = 1;
portas[11].out = OUT_PORT(10);

portas[12].tipo = NAND;
portas[12].inp1 = IN_PORT(100);
portas[12].inp2 = 2;
portas[12].out = 10;

portas[13].tipo = NAND;
portas[13].inp1 = 3;
portas[13].inp2 = 0;
portas[13].out = 4;

portas[14].tipo = AND;
portas[14].inp1 = 3;
portas[14].inp2 = 13;
portas[14].out = 14;

portas[15].tipo = NAND;
portas[15].inp1 = 5;
portas[15].inp2 = 10;
portas[15].out = 1;

portas[16].tipo = AND;
portas[16].inp1 = 8;
```

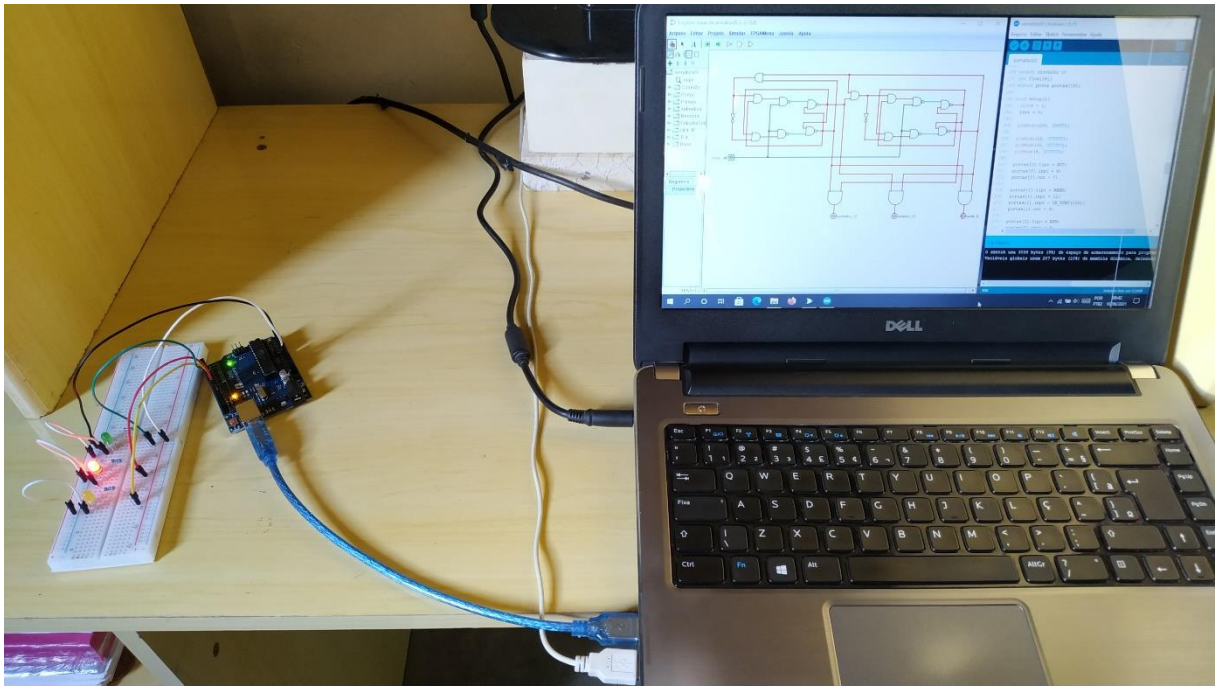
```
	portas[16].inp2 = 4;
	portas[16].out = 11;

	portas[17].tipo = NAND;
	portas[17].inp1 = 15;
	portas[17].inp2 = IN_PORT(100);
	portas[17].out = 12;

	portas[18].tipo = AND;
	portas[18].inp1 = 1;
	portas[18].inp2 = 4;
	portas[18].out = 8;

	c.portas = portas;
	c.fios = fios;
	c.n_portas = 19;
}

void loop(){
	int i;
	for(i = 0; i < c.n_portas; i++){
		simula_porta(c, i);
	}

	if(clock == 1){
		clock = 0;
	}
	iter++;
	if(iter == iters_per_clock){
		clock = 1;
		iter = 0;
	}
	delay(100);
}
```

Fonte: Elaborado pelo autor (2021)

O circuito do semáforo simulado no Arduino apresentou corretamente o atraso de tempo para ocorrer as transições de estado e também a sequência que os sinais devem ser seguidos, no caso, de verde para amarelo, de amarelo para vermelho e de vermelho para verde novamente, reiniciando o ciclo. No momento em que ocorre a troca de sinais do amarelo para o vermelho o sinal verde pisca uma vez rapidamente. É possível corrigir esse comportamento ajustando a duração de pulsos na geração do código para obter uma simulação mais realista. As Figuras 19 e 20 apresentam a simulação do semáforo sendo aplicada no Arduino. O arquivo gerado pelo

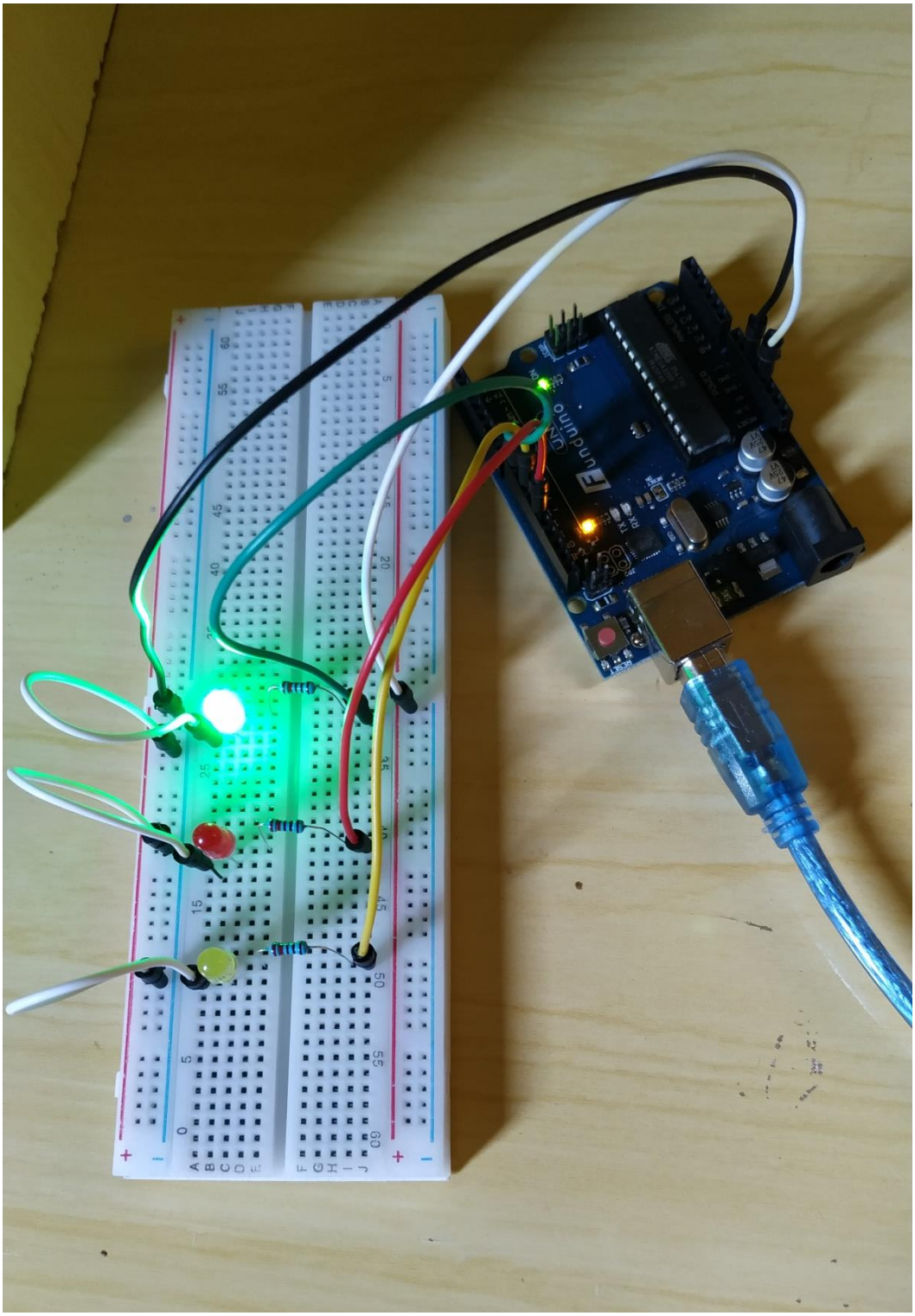
Logisim tem o tamanho de 11.344 bytes e o arquivo “.ino” gerado pelo transpilador tem o tamanho de 5.526 bytes. O transpilador descreveu 19 portas para realizar a simulação.

Figura 19 - Aplicando simulação do circuito do semáforo com *flip-flops* D



Fonte: Elaborada pelo autor (2021)

Figura 20 - Simulação do semáforo com *flip-flops* D exibindo sinal verde



Fonte: Elaborada pelo autor (2021)

5 CONCLUSÃO

O trabalho desenvolvido englobou a implementação de um transpilador capaz de, tomando como entrada um arquivo que descreve um circuito lógico projetado usado o *software* Logisim, gerar um código que implementa um simulador do circuito descrito e pode ser compilado para rodar em uma placa microcontrolada.

Foi verificado que é possível desenvolver um simulador de circuitos digitais que é capaz de rodar em um microcontrolador. Da mesma forma, também foi desenvolvido um método para gerar automaticamente um código-fonte para tal simulador.

Também foi feita uma investigação do formato do arquivo gerado pelo Logisim. A investigação foi cuidadosa o suficiente para permitir uma compreensão cujos detalhes permitem extrair, a partir do arquivo gerado pelo *software* Logisim, as informações necessárias para geração do código de um simulador do circuito descrito.

Também foi desenvolvido um algoritmo capaz de extrair a conectividade entre os elementos descritos no formato de arquivo do *software* Logisim. Uma vez que as informações que descrevem o circuito projetado são extraídas e processadas de forma a se obter as conectividades dos componentes do circuito, foi desenvolvido um algoritmo capaz de, a partir dessa informação, gerar o código de um simulador para o circuito descrito.

O transpilador desenvolvido foi testado nos casos: codificador binário-Gray, somador completo, *flip-flop* SR com *clock* e mais três circuitos projetados aleatoriamente. A saída gerada pelo transpilador em cada qual desses foi compilada e testada em um Arduino. Em todos os casos testados a simulação rodando na placa microcontrolada foi bem sucedida e o comportamento do circuito projetado foi obtido.

Validou-se que é possível desenvolver um transpilador, cujas características alcançadas contemplam os objetivos do projeto, utilizando-se um software mantido e recente e capaz de gerar código que pode ser compilado e executado em uma placa microcontrolada bastante popular. Isso permite um uso em condições bastante favoráveis para alunos e desenvolvedores uma vez que todo o ambiente necessário para execução do transpilador é de fácil acesso.

Os casos testados e bem sucedidos mostram que a ferramenta desenvolvida pode ser aplicada para casos de utilidade real.

5.1 TRABALHOS FUTUROS

Para o futuro, pretende-se integrar a funcionalidade desenvolvido com algum simulador de circuitos digitais de código aberto. Uma integração um pouco mais avançada pode permitir,

por exemplo, a possibilidade de se projetar um circuito, simulá-lo, transpilar e executá-lo em uma placa microcontrolada diretamente de um mesmo software, o que melhoraria substancialmente sua usabilidade e praticidade.

Os algoritmos desenvolvidos foram criados e testados em casos que foram cuidadosamente escolhidos para testar as características do algoritmo, verificar seu funcionamento e garantir que cenários importantes sejam bem sucedidos. No entanto, nenhuma demonstração formal do funcionamento do algoritmo foi feita. Garantir formalmente a corretude dos algoritmos desenvolvidos pode ser um trabalho valioso.

O simulador desenvolvido avalia cada porta lógica simulada em cada iteração de simulação. Isso permite uma simulação natural de atraso de propagação, mas torna a simulação muito custosa quando se considera que não há necessidade de reavaliar a saída de uma porta lógica se seus valores de entrada não foram alterados de uma iteração para outra. Reconhece-se que há casos que de otimização a serem investigados que podem ter um impacto positivo no desempenho da simulação. Além de analisar possibilidades de otimização da simulação, considerar a adoção de um simulador baseado em eventos é também um possível trabalho futuro.

Por ser uma ferramenta de valor didático, é possível utilizá-la em pesquisas que investiguem seu uso entre alunos a fim de se obter dados e estatísticas do seu impacto no aprendizado.

REFERÊNCIAS

ALVES, José Carlos. **Sistemas Digitais.** Porto: Feup/deec, 2004. 120 p.

ARDUINO. **What is Arduino?** 2018. Disponível em: https://www.arduino.cc/en/Guide/Introduction/. Acesso em: 08 mar. 2021.

BURCH, Carl. **Logisim**. Disponível em: http://www.cburch.com/logisim/pt/index.html. Acesso em: 10 mar. 2021.

_____. **Flip-Flops D/T/J-K/S-R**. Disponível em: http://www.cburch.com/logisim/docs/2.7/pt/html/libs/mem/flipflops.html. Acesso em: 28 maio 2021.

CABOT, Jordi. **La diferencia entre Transpilación y Compilación.** 2017. Disponível em: https://ingenieriadesoftware.es/diferencia-transpilacion-compilacion/. Acesso em: 09 mar. 2021.

CARVALHO, Ricardo de. **Javascript Transpiling.** 2017. Disponível em: http://www.timeraposa.com.br/2017/12/javascript-transpiling/. Acesso em: 18 maio 2021.

CHAKRABARTY, Krishnendu. **Logic Simulation.** Durham: Duke University, 2016. 14 slides, color. Disponível em: http://people.ee.duke.edu/~krish/teaching/ECE538/Logic_simulation.pdf. Acesso em: 17 maio 2021.

CHEN, Hue-hua Ann. **Event-Driven Logic Simulator.** 1992. 63 f. Tese (Doutorado) - Curso de Ciência da Computação, Lehigh University, Bethlehem, 1993. Disponível em: https://preserve.lehigh.edu/cgi/viewcontent.cgi?article=1134&context=etd. Acesso em: 17 maio 2021.

DIAS, Carlos Magno Corrêa. **Álgebra Booleana e Lógica Digital**: uma aplicação da lógica matemática. Uma Aplicação da Lógica Matemática. Disponível em: https://repositorio.utfpr.edu.br/jspui/bitstream/1/424/1/REV.%20ACAD._Dias%2C%20Carlos%20Magno%20Corr%C3%AAa_1994.pdf. Acesso em: 20 maio 2021.

ESTEVES, Joaquim Graciano de Sousa. **Simulação de sistemas de produção industriais**. 2009. 73 f. Dissertação (Mestrado) - Curso de Engenharia Electrotécnica e de Computadores, Faculdade de Engenharia da Universidade do Porto, Porto, 2009. Disponível em: https://repositorio-aberto.up.pt/bitstream/10216/59757/1/000134578.pdf. Acesso em: 06 jun. 2021.

GAVIRA, Muriel de Oliveira. **Simulação Computacional como uma Ferramenta de Aquisição de Conhecimento.** 2003. 162 f. Dissertação (Mestrado) - Curso de Engenharia de Produção, Escola de Engenharia de São Carlos, São Carlos, 2003.

IDOETA, Ivan Valeije; CAPUANO, Francisco Gabriel. **Elementos de Eletrônica Digital.** 41. ed. São José dos Campos: Érica, 2012.

LAPIN, Alexander; BULAKH, Dmitry; VAGAPOV, Yuriy. **Event-driven Simulation of Digital Circuits using Modified Petri Nets Algorithm.** 2017.

LIMA, Thiago. **MUX – Multiplexador**. 2015. Disponível em: https://www.embarcados.com.br/mux/. Acesso em: 07 jun. 2021.

MARANGON, Johni Douglas. **Compiladores para Humanos**. Disponível em: https://johnidm.gitbooks.io/compiladores-para-humanos/content/. Acesso em: 18 maio 2021.

MEHL, Ewaldo Luiz de Mattos. **Simulação de Circuitos Eletrônicos em Computadores**. Disponível em: http://www.eletrica.ufpr.br/mehl/te236/apostilaPSpice.pdf. Acesso em: 21 maio 2021.

MIYAGI, Paulo E. **Introdução a Simulação Discreta.** São Paulo, 2006. 52p. (Apostila).

PADMANABHAN, Arvind; P, Gurumoorthy. **Transpiler**. 2019. Disponível em: https://devopedia.org/transpiler. Acesso em: 18 maio 2021.

PANTUZA, Gustavo. **Elementos de memória - O circuito lógico Flip-Flop D**: Entenda o circuito lógico Flip-Flop D utilizado para implementar um elemento de memória em circuitos digitais. 2018. Disponível em: https://blog.pantuza.com/artigos/elementos-de-memoria-o-circuito-logico-flip-flop-d. Acesso em: 10 jun. 2021.

PEREIRA, Ana Paula. **O que é XML?** 2009. Disponível em: https://www.tecmundo.com.br/programacao/1762-o-que-e-xml-.htm. Acesso em: 25 maio 2021.

PORTAL EDUCAÇÃO. **Linguagem de marcação**. Disponível em: https://siteantigo.portaleducacao.com.br/conteudo/artigos/informatica/linguagem-de-marcacao/31639#. Acesso em: 25 maio 2021.

POZZEBOM, Rafaela. **O que são sistemas embarcados?** 2014. Disponível em: https://www.oficinadanet.com.br/post/13538-o-que-sao-sistemas-embarcados. Acesso em: 20 maio 2021.

REIS, Vinicius. **Ecossistema JavaScript—Parte 04: Transpilers.** 2016. Disponível em: https://blog.codecasts.com.br/ecossistema-javascript-parte-04-transpilers-734f77422316. Acesso em: 17 maio 2021.

SOUZA, Anderson R. de et al. A Placa Arduino: uma opção de baixo custo para experiências de física. **Revista Brasileira de Ensino de Física**, Rio de Janeiro, v. 33, n. 1, p.0-5, 21 mar. 2011.

TANENBAUM, Andrew S. **Organização Estruturada de Computadores.** 5. ed. São Paulo: Pearson Prentice Hall, 2007.

TOCCI, Ronald J.; WIDMER, Neal S.; MOSS, Gregory L. **Sistemas Digitais:** princípios e aplicações. 11. ed. São Paulo: Pearson Prentice Hall, 2011.

TRINDADE JUNIOR, Rosumiro; JULIÃO, Jodelson Moreira. **Circuitos Digitais.** Manaus: E-tec Brasil, 2012. 120 p.

URICER. ***Apostila de COMPILADORES,*** Erechim, 2001. 61p. (Apostila).

WIKIWAND (Org.). **Logic gate.** Disponível em: http://www.wikiwand.com/simple/Logic_gate. Acesso em: 08 mar. 2021.