PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Programa de Graduação em Engenharia de Computação

Diego Silva Caldeira Rocha

CSLADDER MIC – COMPILADOR E SIMULADOR DA LINGUAGEM LADDER PARA O MICROCONTROLADOR ATMEGA328

Belo Horizonte
2012

Diego Silva Caldeira Rocha

CSLADDER MIC – COMPILADOR E SIMULADOR DA LINGUAGEM LADDER PARA O MICROCONTROLADOR ATMEGA328

Monografia apresentada ao Curso de Engenharia de Computação da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Luiz Carlos Figueiredo

Belo Horizonte
2012

Diego Silva Caldeira Rocha

CSLADDER MIC – COMPILADOR E SIMULADOR DA LINGUAGEM LADDER PARA O MICROCONTROLADOR ATMEGA328

Monografia apresentada ao Curso de Engenharia de Computação da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação.

_____

Luiz Carlos Figueiredo (Orientador) – PUCMINAS

_____

Romanelli Lodron Zuim – PUCMINAS

_____

Rosely Maria Velloso Campos - PUCMINAS

Belo Horizonte, 21 de junho de 2012

Aos meus pais, Edivaldo Caldeira Rocha
e Maria Nilza Silva Caldeira Rocha
pelo incentivo e carinho

AGRADECIMENTOS

Renunciar sempre é mais fácil, mas a GLÓRIA da VITÓRIA após uma batalha desafiadora e sacrificante é o alcançar da PLENITUDE.

Agradeço a Deus, pois sem Ele nada seria possível. Ao meu PAI pela confiança e por estar sempre ao meu lado. A minha MÃE, por todo apoio e incentivo nesta conquista. Aos meus familiares, Irmãs, Sobrinha, Avós, Avô, Tios, Tias, Primos e Primas por estar comigo nesta jornada. Meu Bem pelo auxílio, carinho e compreensão e a todas as pessoas que de alguma forma me auxiliaram para alcançar este momento.

Agradeço meu orientador Luiz Carlos Figueiredo pela motivação, auxílio nas dificuldades e em muitas ocasiões proporcionar um ambiente amigável, eliminando os obstáculos entre aluno e professor.

Agradeço aos professores do curso de Engenharia de Computação que devido aos seus conhecimentos transmitidos, me auxiliaram nas escolhas das estratégias e conteúdos utilizados para elaboração deste projeto.

Agradeço a instituição que trabalho Polícia Militar de Minas Gerais por me manter em horário compatível com os estudos.

RESUMO

O presente trabalho apresenta a construção de um compilador e simulador da linguagem Ladder para o microcontrolador ATmega328. O compilador recebe o programa fonte em Ladder criado através de uma interface gráfica e após a execução das etapas de compilação é gerado o código objeto na linguagem C/C++ para plataforma Arduino Duemilanove. O microcontrolador exerce a função de Controlador Lógico Programável (CLP) e contém o código hexadecimal gerado a partir da compilação do código objeto pelo compilador Arduino Alpha. O simulador faz o papel do CLP relacionando a interface com o campo, simulando os sinais de entrada e saída através de botões virtuais, Leds, dentre outros. A partir do embasamento teórico da linguagem Ladder, processadores de linguagem e microcontroladores, foi possível realizar a construção do projeto de *software*, *hardware* e do protótipo para teste. Os testes foram compostos por exercícios de automatização, sendo programados em Ladder. Em seguida, foi testada a simulação em *software* e a eficiência da aplicação no microcontrolador. Obteve-se êxito em todos os testes e o microcontrolador teve sua funcionalidade como um CLP. Todos objetivos propostos no projeto foram alcançados, sendo possível criar uma ferramenta didática de baixo custo.

Palavras-chave: Ladder. Compilador. CLP. Microcontrolador. Automatização industrial. Linguagens de Programação. Depurador.

ABSTRACT

The following study aim to build a compiler and a simulator of Ladder Language to ATmega328 microcontroller. The compiler receives Ladder source-code built by a graphic interface and, after the execution of each step of compilation, the object-code is built in C/C++ language to Arduino Duemilanove development board. The microcontroller has a function of a Programmable Logic Controller (PLC) and it has the hexadecimal code generated from the object-code compiled by Arduino Alpha compiler. The Simulator do the PLC function related to field interface, simulating the input and output signals by virtual buttons, leds and others. From Ladder language theoric knowledge, language processors and microcontrollers was possible to build the software project, hardware and then, the prototype. The tests were composed by automation exercises programmed in Ladder. Then, the software simulation and microcontroller aplication efficiency was tested. All tests were success fully done and the microcontroller has its functionality like a PLC. Every objective proposed in this project was achieved, being possible to build a low cost didatic tool.

Keywords: Ladder. Compiler. PLC. Microcontroller. Industrial automation. Programming Languages. Debugger.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIAÇÕES

ADC	- *Analog-to-Digital Converter*
AS	- Analisado Sintático
C	- Contadores
CI	- Circuito Integrado
CLPs	- Controladores Lógicos Programáveis
CSN	- Companhia Siderúrgica Nacional
CTU	- *Counter Up*
DAC	- *Digital-to-Analog Converter*
EEPROM	- *Electrically-Erasable Programmable Read-Only Memory*
EN	- Entradas
ENIAC	- *Electrical Numerical Integrator and Calculator*
EUA	- Estados Unidos da América
F	- *Flags*
GUI	- *Grafic Users Interface*
IEC	- *International Electrotechnical Commission*
LD	- Ladder *Diagram*
LP	- Linguagem de Programação
NA	- Normal Aberto
NF	- Normal Fechado
OTE	- *Output Terminal Energize*
OTL	- *Output Terminal Energize Latch*
OTU	- *Output Terminal Unlatch*
PLC	- Programmable Logic Controller
PWM	- *Pulse-Width Modulation*
RES	- *Reset*
RISC	- *Reduced Instruction Set Computer*
S	- Saídas
SRAM	- *Static Random Access Memory*
T	- Temprorizadores
TON	- *Timer On Delay*
TSI	- Tabela de Símbolos de Instruções
XIC	- *eXamine If closed*

XIO - *eXamine If Open*

SUMÁRIO

1 INTRODUÇÃO

A procura da humanidade pelo desenvolvimento tecnológico passa pela industrialização dos processos de fabricação dos bens de consumo. Em meados do século passado com internacionalização da economia brasileira houve uma expansão da automatização das etapas industriais, em conjunto a este progresso ocorreram acréscimos de interfaces de comunicações entre homens e máquinas.

A necessidade da industrialização ocorreu no século XVIII onde o Reino Unido estabeleceu uma política econômica liberal e realizou vários tratados com outros países a fim de fornecer seus produtos manufaturados. Com o aumento das encomendas as máquinas se tornaram atrativas para a expansão produtiva. A primeira máquina que expressou a automatização industrial foi o motor a vapor de Thomas Newcomem, que retirava as águas das minas de ferro e carvão encaminhando-as para coloração de tecidos (BRAICK; MOTA, 2010).

Em 1946 nos Estados Unidos da América (EUA) foi criado o primeiro computador digital eletrônico de grande escala, conhecido como *Electrical Numerical Integrator and Calculator* (ENIAC) (TORRES, 1996). No Brasil a industrialização acentuou-se logo após esse período com a criação da Companhia Siderúrgica Nacional (CSN) e a instalação de fábricas vindas de outros países, principalmente as indústrias automobilísticas (CAROLINE, 2011). A partir de então houve uma sofisticação da comunicação do homem com a máquina, que passou de um simples acionamento humano de uma botoeira para um programa automático realizado por uma Linguagem de Programação (LP), composta por regras semânticas e sintáticas, as quais definem um conjunto de instruções tomadas pelos instrumentos.

Uma das tecnologias utilizada na indústria para receber conjuntos de instruções em vários tipos de linguagens são os Controladores Lógicos Programáveis (CLPs), sendo computadores baseados em um microprocessador que permite fazer o controle industrial, desde pequenas tarefas até as mais complexas. Cada CLP possui um programa específico desenvolvido pelo usuário e um conjunto de entradas e saídas. A partir dos sinais de entrada é realizado o controle industrial pelos atuadores. Normalmente os CLPs utilizam linguagem de alto nível de abstração que pode ser compreendida facilmente pelo usuário. Uma das linguagens utilizada e ministrada pelos cursos técnicos profissionalizantes é a linguagem gráfica

Ladder. Ela consiste em um conjunto de diagramas elétricos que definem as respostas aos processos.

Este trabalho teve a finalidade de construir um ambiente de programação e simulação da linguagem Ladder com utilização de um microcontrolador exercendo a função de CLP. O compilador Ladder posteriormente gera o código objeto na linguagem C/C++ para compilação no Arduino *Alpha* que grava o código em hexadecimal para o microcontrolador ATmega328. Assim, o microcontrolador exerce a função de um CLP para fins didáticos em conjunto com uma linguagem de programação específica para aplicações industriais.

1.1 Justificativa

A transformação da teoria em prática está ligada a uma boa qualificação profissional. O uso de equipamentos industriais para o auxilio didático, muitas vezes, tem o custo elevado, mas garante a eficiência da aprendizagem. Aumentar as alternativas para a capacitação tecnológica tem sido uma busca constante nos últimos anos.

A formação do profissional ocorre normalmente em escolas técnicas e superiores. Nelas são encontradas um elevado investimento para estruturas de auxilio à aprendizagem. Perante a esta situação muitas escolas técnicas utilizam a programação em Ladder, que é de fácil compreensão para programar CLPs. Normalmente, os CLPs são estruturas de custos elevados, o que obriga alguns cursos técnicos a deslocar seus alunos para outras escolas que possuam estruturas melhores e assim obterem melhor formação nos laboratórios.

Os microcontroladores são *hardwares* de pequeno porte e baixo custo. Normalmente, são vendidos juntamente com um kit de gravação e seu funcionamento é similar ao CLP. Entretanto, sua alternativa de programação possui o nível de abstração inferior às linguagens ministradas em cursos técnicos. A implementação da linguagem Ladder em microcontrolador é muito atrativa, pois representa a diminuição de investimento nos laboratórios técnicos e superiores. Além disso, é uma alternativa didática para exemplificação da linguagem Ladder, contribuindo para formação da estrutura de criação algorítmica do aluno para resolver os problemas de automatização do dia-a-dia.

Este trabalho procura servir como base para estudos futuros. Para maior flexibilidade do mesmo, foi escolhida a linguagem C/C++, que é utilizada na maioria das aplicações envolvendo microcontrolador e também foi utilizado plataforma Arduino Duemilanove com Atmega328, pois é uma ferramenta *open-source* de contribuição dos desenvolvedores.

Portanto, com a construção deste ambiente didático para desenvolvimento da linguagem Ladder, os alunos poderão associar a teoria fornecida pelo professor com a prática de montagem de circuitos com o microcontrolador.

1.2 Objetivos Gerais

Construir uma interface em *hardware* e *software* de baixo custo, para usuários de níveis técnicos e acadêmicos, que propicie desenvolver e simular programas para CLPs utilizando os microcontroladores. Associar a aprendizagem teórica à prática permitindo o desenvolvimento de aplicativos mais complexos como os CLP's.

1.3 Objetivos Específicos

- Desenvolver de forma simples um programa gráfico e transformá-lo em uma linguagem que descreva as ações do microcontrolador.
- Construir um protótipo de *hardware* para que seu funcionamento seja similar à simulação do aplicativo em Ladder.
- Criar uma ferramenta simples e de fácil manipulação para programação em Ladder.
- Fornecer estrutura coerente para que haja integração entre *software* e *hardware*.
- Fazer análises da programação em Ladder e fornecer para os usuários os eventuais erros.
- Melhorar a compreensão do CLP, visando sua otimização.
- Possibilitar uma simulação do programa em Ladder para que os usuários previamente visualizem eventuais falhas.
- Possibilitar que a ferramenta de desenvolvimento seja compatível com a maioria dos computadores pessoais.

1.4 Estado da Arte

O aumento dos profissionais no desenvolvimento de *software* e *hardware* provocou a expansão do número de LPs. Logo, cria-se a necessidade de ter estruturas de conversões das linguagens como: compiladores e tradutores. Assim, cresce o número de trabalhos relacionados.

Westhues (2009) desenvolveu um compilador na linguagem C, chamado LDmicro para programação em Ladder para os microcontroladores do pic16 e AVR, o autor disponibiliza em seu site o código do compilador para modificações não comerciais, a compilação do programa em Ladder passa por várias etapas até chegar geração do código em hexadecimal.

Hidalgo, Oliveira e Oliveira (2008) apresentam um tradutor da linguagem Ladder que para cada comando é realizada a tradução para o código *assembly* para o microcontrolador AT89S52 (família 8051), conseguindo um bom resultado ao implementar uma aplicação comum para testar CLP.

Pagano e Diniz (2009) propõem reescrever o código do compilador de Westhues (2009) na linguagem C#, para melhorar a interação com os usuários, contudo devido a falta de tempo os autores desenvolveram uma interface simples para implementação em Ladder exportando o arquivo de saída para compilação no LDmicro.

Neto (2009) cria um ambiente de desenvolvimento da linguagem Ladder e posteriormente utiliza do microcontrolador PIC acoplado a um computador pessoal, através da porta serial para que ambos funcionem em conjunto como um CLP, expandindo a capacidade de processamento do microcontrolador com uso do computador.

Diante destes estudos, verifica-se que a ferramenta elaborada neste trabalho é de grande interesse acadêmico e auxilia para o surgimento de novas pesquisas de aperfeiçoamento do tema, contribuindo para o desenvolvimento científico.

1.5 Estrutura do Trabalho

O presente trabalho divide-se em cinco etapas. No capítulo 1 está contida a introdução do tema, a justificativa, os objetivos gerais e específicos e alguns trabalhos similares ao tema proposto, contidos no Estado da Arte. O capítulo 2 é

direcionado para o conceito de linguagem Ladder, explicação teórica de compiladores e do funcionamento dos microcontroladores (em especial do ATmega328). O capítulo 3 apresenta as estratégias para o desenvolvimento do aplicativo e implementação do protótipo de testes. No capítulo 4 exibe os testes, dificuldades encontradas e os resultados. O capítulo 5 contém a conclusão e sugestões para trabalhos futuros.

2 REVISÃO LITERÁRIA

Este capítulo faz a revisão literária dividida nas seguintes seções:

- Linguagens de Programação;
- Processadores de Linguagem;
- Microcontroladores ATmega328.

2.1 Linguagens de programação

> Uma Linguagem de programação (LP) é uma ferramenta utilizada pelo profissional de computação para escrever programas, isto é, conjuntos de instruções a serem seguidas pelo computador para realizar um determinado processo (VAREJÃO, 2004, p.16).

Devido às limitações físicas dos primeiros computadores, inicialmente eles só poderiam ser programados em linguagens de programação muito simples, que se caracterizavam por serem exclusivas de uma única máquina, hoje essas linguagens são conhecidas como sendo linguagem de máquina ou linguagem de baixo nível. Na medida em que o tempo evoluía a computação avançava cada vez mais e as aplicações ficavam cada vez mais complexas, então se percebeu que o uso de linguagens tão simples reduzia o desempenho dos programadores e limitava o uso dos recursos computacionais. Para solucionar este problema surgiram as linguagens de programação de alto nível, caracterizadas por não serem exclusiva de um computador e possuírem um conjunto vasto de instruções (VAREJÃO, 2004).

Portanto as LPs de alto nível surgiram para beneficiar os desenvolvedores, uma vez que as linguagens de máquina, normalmente são programadas por um conjunto de bits binário, compostos por *opcodes* de instrução, dados de registradores, variáveis e etc., tornando assim a dificuldade, pois para criar um aplicativo era necessário realizar inúmeras operações binárias. O objetivo das LPs é tornar mais efetivo o processo de desenvolvimento de *software*.

2.1.1 *Linguagens gráficas*

São linguagens que utilizam dos recursos visuais como símbolos, fios, diagramas elétricos e etc., para implementação dos programas. São ferramentas vantajosas, pois os elementos gráficos são mais fáceis de serem compreendidos do que os textos, também permitem que os usuários sem afinidade na programação possam elaborar programas e possui eficiência na velocidade de implementação. Devido ao recurso gráfico, sua programação limita a flexibilidade do aplicativo, funcionando assim somente para aplicações específicas. Podem se dividir em Linguagem de programação visual híbrida e linguagem de programação visual pura.

2.1.1.1 Linguagem de programação visual híbrida

> Nessa categoria, incluem-se as linguagens de programação que não são somente visuais, mas têm parte de sua especificação determinada de forma visual. Normalmente este tipo de linguagem de programação visual não é simplesmente uma "linguagem", mas um "produto", que é comercializado por uma empresa de *software*. Em alguns casos esse "produto" está mais para uma linguagem de programação convencional (por textos), dotada de diversas ferramentas de programação visual, do que realmente linguagens de programação visual. Em outros casos, as ferramentas de programação visual efetivamente participam da elaboração do programa, como por exemplo, na determinação de hierarquia de classes e métodos, em linguagens orientadas a objeto. As linguagens que incluem- se nesta categoria são normalmente desenvolvimentos de linguagens de programação convencional, acrescidas das ferramentas visuais que lhes dão o "status" de visuais. Exemplos incluem o Delphi, o Visual Basic, o Visual C++ (GUDWIN, 1997, p.14).

2.1.1.2 Linguagem de programação visual pura

> São aquelas em que o programa é determinado exclusivamente por meio dos gráficos e diagramas que o especificam. O paradigma mais eficiente nesse tipo de programação é a elaboração de diagramas com nós e arcos, sendo que os nós representam módulos de *software*, a partir de uma biblioteca prévia de módulos, e os arcos determinam a conexão entre os diferentes módulos. Como esses programas dependem fortemente dos módulos de *software* que existem previamente no sistema, seu uso é limitado a aplicações bem definidas (GUDWIN, 1997, p. 14-15).

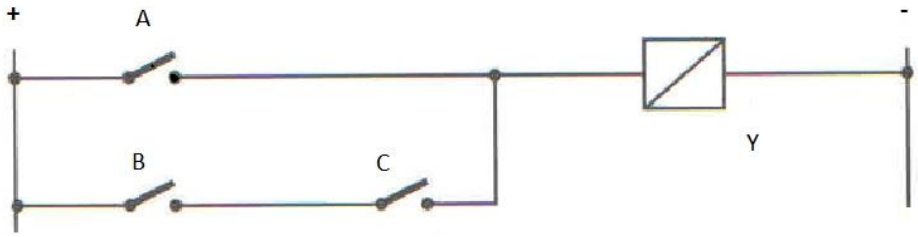
2.1.2 *A Linguagem de programação Ladder*

A linguagem Ladder ou *Diagram Ladder* (LD) também conhecida como: Diagrama elementar, de linha ou em escada (RIBEIRO, 2001). Ela é uma linguagem

gráfica de alto nível que se originou nos Estados Unidos da América (EUA). Baseia-se na representação gráfica de relés em escada. Na *International Electrotechnical Commission* (IEC) onde foi divulgada a norma IEC 61131-3, que está entre as cinco linguagens padronizadas para programar CLP.

Atualmente LD é a linguagem mais utilizada para programar CLP, pois é similar a linguagem dos Diagramas Lógicos (portas lógicas) e apresenta facilidade de desenho. A Figura 1 apresenta um trecho de programa Ladder e a respectiva a álgebra de Boole. Quando as chaves das variáveis de entrada estão fechadas tem-se o nível lógico "1" (verdadeira) e em caso aberta apresenta o nível lógico "0" (falsa).

FIGURA 1 – Álgebra de Boole: $Y = A + B \cdot C$



Fonte: MORAES, 2007

2.1.2.1 Instruções da linguagem Ladder

As instruções da linguagem Ladder estão divididas em dois grupos: instruções de entrada e de saída. Nas instruções de entrada ficam formuladas as perguntas e nas instruções de saída correspondem as ações realizadas com base de afirmação ou negação das instruções de entrada.

- **Normal Aberto (NA)**

A instrução Normal Aberto ou eXamine If Closed (XIC), representada pela Figura 2, verifica se o endereço da entrada lógica está verdadeiro, em caso afirmativo é dado continuidade lógica para a sequência do circuito e caso contrário não é realizado a sequência como descrito na Tabela 1 (MORAES, 2007).

FIGURA 2 – Representação da instrução XIC

Fonte: MORAES, 2007

TABELA 1 – Tabela Verdade NA

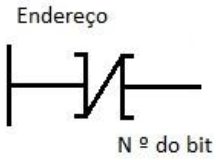
Estado do Bit	Instrução NA
0	FALSO
1	VERDADEIRO

Fonte: MORAES, 2007

- **Normal Fechado (NF)**

A instrução NF ou eXamine If Open (XIO), representada pela Figura 3, verifica se o endereço da entrada lógica está falso e nesta condição é dado continuidade lógica para a sequência do circuito e em caso contrário não é realizado o prosseguimento como descrito na Tabela 2 (MORAES, 2007).

FIGURA 3 – Representação instrução XIO



Fonte: MORAES, 2007

TABELA 2 – Tabela Verdade NF

Estado do Bit	Instrução NF
0	VERDADEIRO
1	FALSO

Fonte: MORAES, 2007

- ***Output Terminal Energize* (OTE)**

É uma Bobina energizada, sendo uma instrução de saída representada pela Figura 4. Caso apresente continuidade lógica até a instrução o bit endereçado pelo comando receberá o estado lógico 1 e em caso contrário 0 (MORAES, 2007).

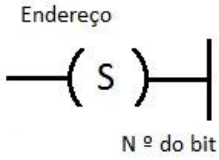
FIGURA 4 – Representação instrução OTE

Fonte: MORAES, 2007

- ***Output Terminal Energize Latch*** **(OTL)**

É uma bobina energizada com retenção, sendo uma instrução de saída representada pela Figura 5. Caso apresente continuidade lógica até ela, o bit endereçado pela instrução receberá o estado lógico 1. Diferentemente da instrução OTE, se uma única vez ocorrer à continuidade lógica ela mantém permanentemente a bobina de saída em nível alto (MORAES, 2007).

FIGURA 5 – Representação instrução OTL

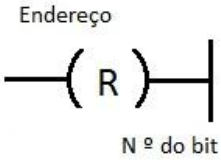


Fonte: MORAES, 2007

- ***Output Terminal Unlatch*** **(OTU)**

É uma instrução para desabilitar saída com retenção, representada pela Figura 6. Caso apresente continuidade lógica até ela o bit endereçado pela instrução será desabilitado da instrução OTL e manterá o estado lógico em 0 (MORAES, 2007).

FIGURA 6 – Representação instrução OTU

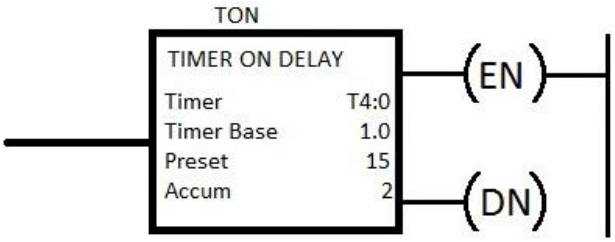


Fonte: MORAES, 2007

- ***Timer On Delay*** **(TON)**

É um temporizador crescente sem retenção à energização sendo uma instrução de saída representada pela Figura 7. Caso apresente continuidade lógica até ela o bit EN é colocado no estado lógico 1 e será realizado uma contagem de tempo. Baseando no intervalo de tempo fornecido durante a programação no final da contagem de tempo o bit DN receberá o nível lógico 1 (MORAES, 2007).

FIGURA 7 – Representação instrução TON

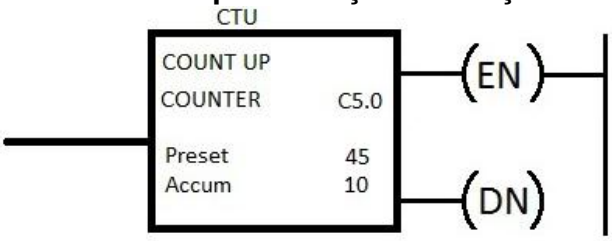


Fonte: MORAES, 2007

- ***Counter Up*** **(CTU)**

É um contador crescente sendo uma instrução de saída representada pela Figura 8. A qual se verifica se há ou não continuidade lógica na linha de instrução. A cada transição da condição lógica da linha de falsa para verdadeira, a instrução CTU incrementa o valor de Accum e coloca o bit EN em nível lógico alto e quando o valor de Accum for igual ao *Preset* o bit DN fica em nível lógico 1 (MORAES, 2007).

FIGURA 8 – Representação instrução CTU

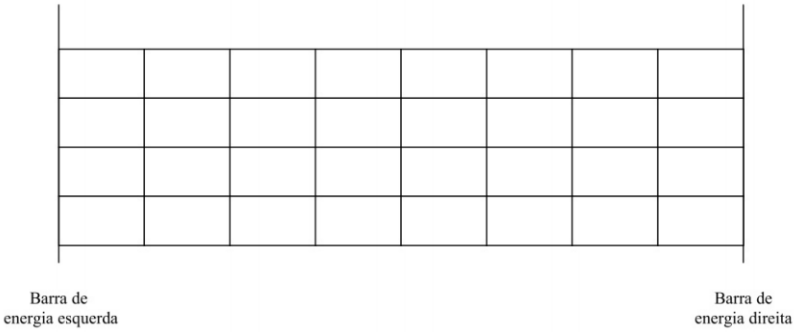


Fonte: MORAES, 2007

2.1.2.2 Funcionamento da gramática Ladder

LD é composto *rungs* (degraus) ou linhas formando uma escada onde os componentes estão conectados, entre as duas barras, existe uma matriz por onde estarão os componentes conforme a Figura 9.

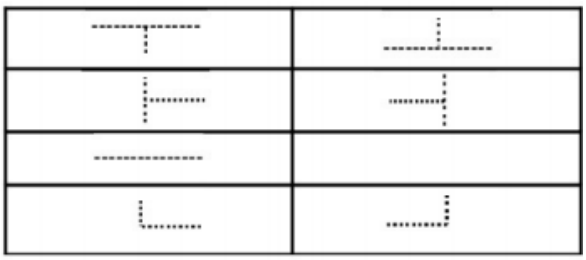
FIGURA 9 – Representação das células da matriz lógica do LD



Fonte: RICHTER, 2001

Cada célula da matriz conterá o desenho gráfico da instrução Ladder ou conexões como a Figura 10.

FIGURA 10 – Conexões da gramática Ladder



Fonte: HIDALGO; OLIVEIRA; OLIVEIRA, 2008

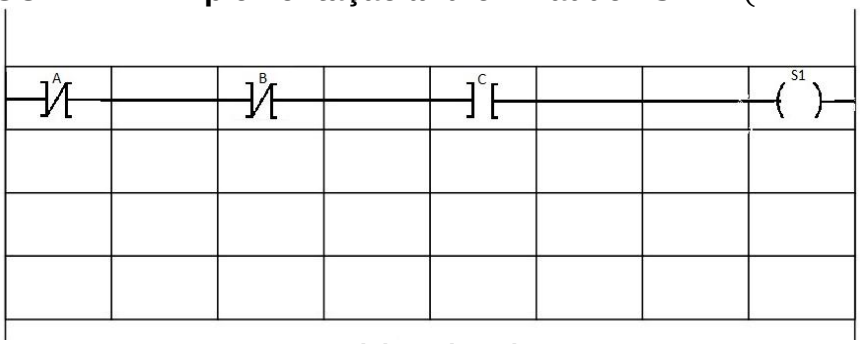
Para cada aplicação que se implementa em Ladder é definido o tamanho máximo de linhas e colunas de uma matriz lógica. Para realização deste trabalho foi

utilizada o tamanho de 9 colunas e 4 linhas, totalizando 36 células, entretanto em um mesmo programa pode haver várias matrizes lógicas.

A lógica a ser implementa é de forma em que as instruções ou conexões sejam energizados, contendo um caminho corrente entre a barra da esquerda até a barra da direita. Na matriz lógica somente a última coluna pode receber instruções de saída, portanto em cada matriz pode se ter no máximo 4 instruções de saída, para facilitar a implementação cada matriz tem somente uma instrução de saída. As demais células recebem conexões ou as instruções de entrada NA e NF.

O LD representa um conjunto de instruções com plano da lógica *and* e *or*, de forma em que as instruções de entrada em série na mesma linha da matriz, representam o plano de *and´s*, como a Figura 11.

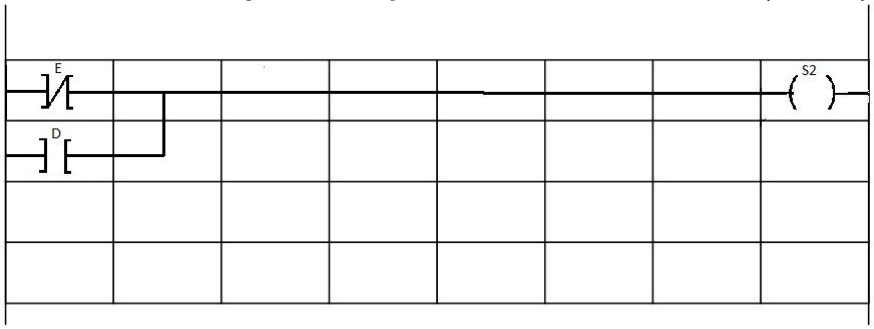
FIGURA 11 – Implementação *and* em Ladder: $S1 = (\overline{A} \cdot \overline{B} \cdot C)$



Fonte: Elaborado pelo autor

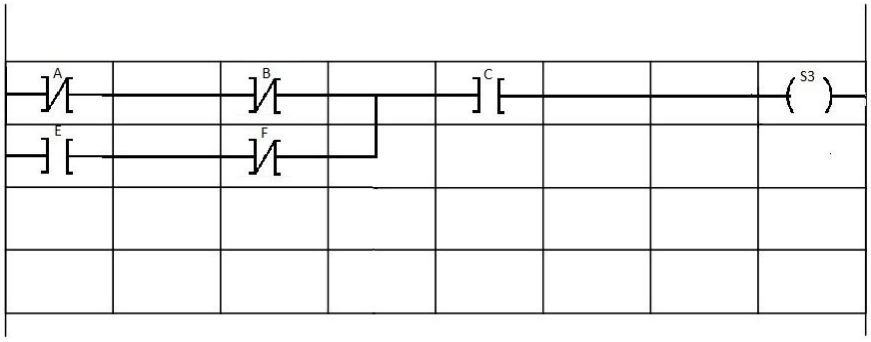
Já o conjunto de instruções de entrada em paralelo na mesma coluna, representa o plano *or's*, como a Figura 12 e diante dos planos *and* e *or* pode ter ambos aninhados como a Figura 13.

FIGURA 12 – Implementação *or* em Ladder: $S2 = (\overline{E} + D)$



Fonte: Elaborado pelo autor

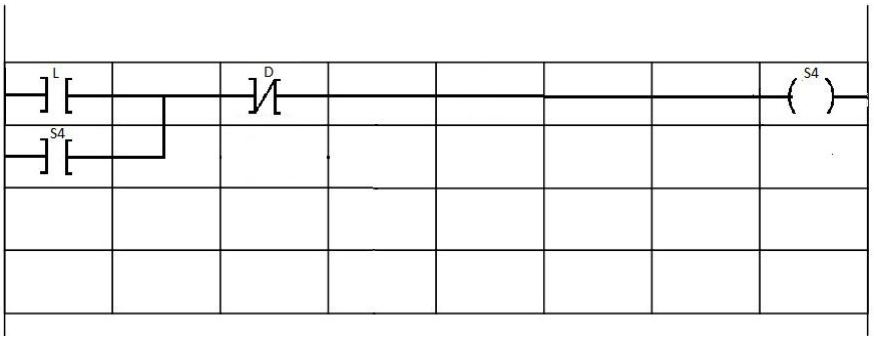
FIGURA 13 – Implementação *and* e *or* em Ladder: $S3 = ((\overline{A} \cdot \overline{B}) + (E \cdot \overline{F}) \cdot C)$



Fonte: Elaborado pelo autor

A gramática Ladder permite que bobinas de saída possam ser reaproveitadas como instruções de entrada como a Figura 14.

FIGURA 14 – Implementação liga e desliga em Ladder: $S4 = ((L + S4) \cdot \overline{D})$



Fonte: Elaborado pelo autor

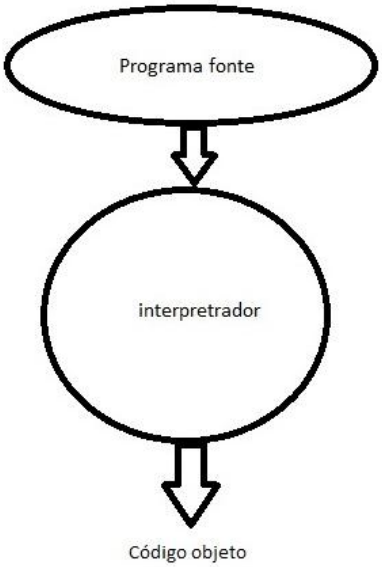
2.2 Processadores de linguagens

Um programa descrito em qualquer LP para ser executado é necessário fazer uma transição para outra linguagem. Os programas que fazem a conversão de código-fonte para código-objeto são chamados de Interpretadores e compiladores.

Os interpretadores são processadores, que para cada entrada de dado realizado no programa fonte é feito a tradução para a linguagem fim, descrito pela Figura 15. São vantajosos, pois são simples para implementação, porém não possuem ferramentas de otimização, acarretando lentidão na execução.

Os compiladores são processadores complexos que após a inserção de um código-fonte, se passa por várias etapas até produzir um código conhecido como *assembly*. São programas complexos para implementação e os códigos objeto são rápidos no momento de execução.

FIGURA 15 - Interpretação



Fonte: SEBESTA, 2003.

Os compiladores são processadores complexos que após a inserção de um código-fonte, se passa por várias etapas até produzir um código conhecido como *assembly*. São programas complexos para implementação e os códigos objeto são rápidos no momento de execução.

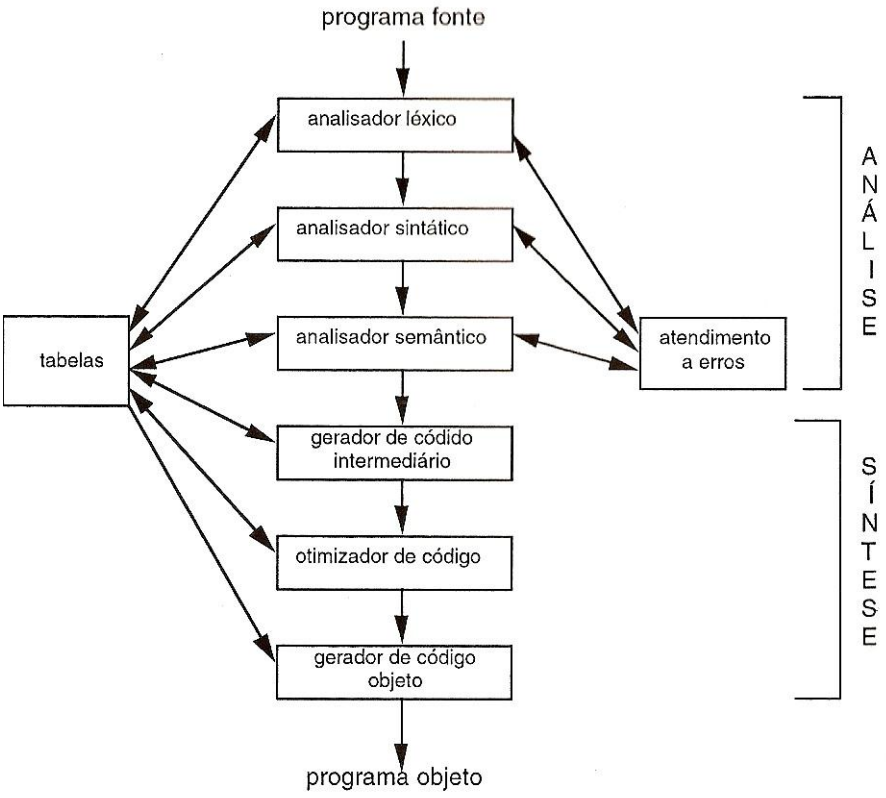
Para realização deste trabalho a arquitetura do processador de linguagem foi híbrida contendo estruturas de compiladores e interpretador.

2.2.1 *Estrutura de um Compilador*

O processo de tradução de um Compilador é subdividido em várias fases, conforme a Figura 16. No ponto de vista de implementação a divisão das fases não são claras, ou seja, certa etapa pode se utilizar métodos de etapas passadas.

A seguir serão descritas as etapas de um Compilador.

FIGURA 16 – Estrutura de um Compilador



Fonte: PRICE; TOSCANI, 2008.

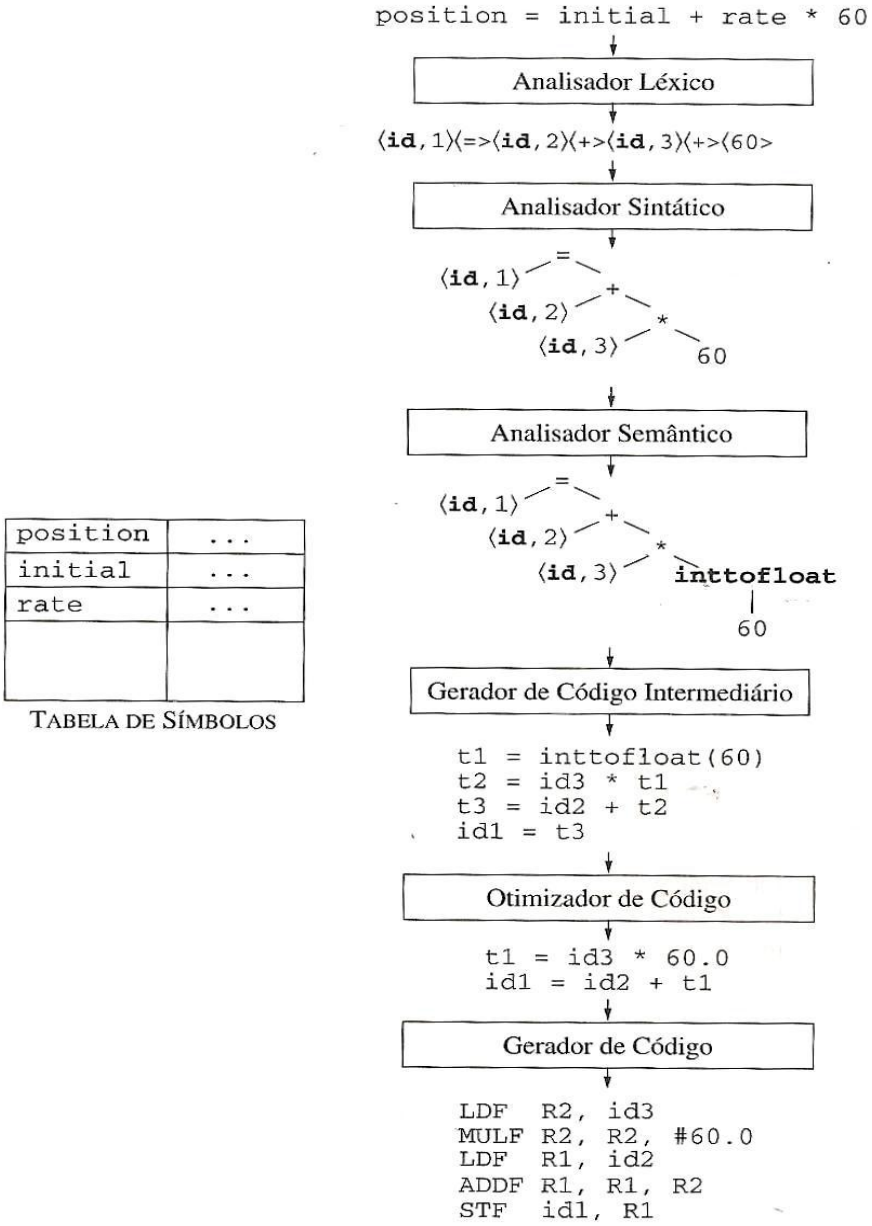
2.2.1.1 Análise léxica

Esta fase se objetiva a identificar o seguimento de caracteres da linguagem e os agrupa em sequências significativas chamadas *lexemas*. Para cada *lexema* o Analisador Léxico gera um *token* (PRICE; TOSCANI, 2008). Os *tokens* são compostos por identificadores, palavra reservada, delimitadores e etc. Outra função do Léxico é realizar o preenchimento da Tabela de Símbolos atribuindo um valor para os identificadores encontrados. O Analisador Léxico funciona como uma sub-rotina comandada pelo analisador sintático (AHO et al., 2008).

2.2.1.2 Análise sintática

A Análise Sintática é também conhecida como *parser* sendo a segunda fase de um Compilador onde através dos *tokens* fornecidos pela etapa anterior é realizado uma representação intermediária de uma árvore com a estrutura gramatical da sequência encontrada como a Figura 17.

FIGURA 17 – Tradução de uma instrução de atribuição



Fonte: AHO et al., 2008.

O objetivo desta etapa é verificar se a árvore produzida pelo programa fonte é uma sentença válida na gramática da LP e também é dada a continuidade na inserção na Tabela de Símbolos com a visibilidade dos identificadores nos escopos.

2.2.1.3 Análise semântica

Esta etapa utiliza-se da árvore sintática e Tabela de Símbolos para verificar a consistência semântica do programa fonte. Uma análise importante feita nesta etapa é a de verificar se para cada operando estão relacionados operadores compatíveis,

a chamada verificação de tipos. Em caso de incompatibilidade, apontar o erro para o usuário ou aplicar técnica de conversão implícita nos operadores para o tipo desejável (AHO et al., 2008).

2.2.1.4 Geração de código intermediário

Esta etapa se designa em elaborar uma representação intermediária próxima ao código alvo. Seu objetivo é fornecer facilidade de conversão para o código de máquina como a Figura 17. A diferença entre o Código Intermediário e o alvo é que no intermediário não é apresentado detalhes de registradores e local de armazenamento na memória.

2.2.1.5 Otimização de Código

O seu objetivo é fornecer no desfecho um código objeto menor sendo mais rápido, com a eliminação de códigos não necessários ou troca por instruções mais rápidas. Um exemplo de otimização ocorre quando em duas linhas seguidas um mesmo identificador é atribuído com valores diferentes, logo serão eliminados os comandos da primeira linha no Código Intermediário.

2.2.1.6 Tabela de Símbolos

A Tabela de Símbolos é uma estrutura de dados auxiliar, onde ficam registradas as informações das variáveis, como: tipo, nome, escopo e etc. Fornecer um acesso rápido para que o Compilador consulte e altere as informações dos identificadores.

2.2.1.7 Geração de código

Esta é a última etapa de um Compilador, onde utiliza-se do Código Intermediário e a Tabela de Símbolos para enfim gerar o código alvo. Um problema crítico feito nesta etapa é conseguir atribuir os registradores, pois são limitados. Porém, evitam que constantemente sejam lidos e escritos dados na memória.

2.3 Microcontrolador Atmega328

Um microcontrolador é um computador encapsulado em um chip ou Circuito Integrado (CI). Os componentes básicos que integram um microcontrolador são processador; memórias e dispositivos de entrada e saída. Atualmente, a maioria dos componentes eletrônicos como: rádio; televisão; micro-ondas e etc. possuem embarcados um ou vários microcontroladores. São responsáveis em seguir uma sequência predeterminada de comandos.

O microcontrolador além das unidades lógicas e aritméticas comuns aos processadores possuem outros componentes como: memória *Electrically-Erasable Programmable Read-Only Memory (*EPROM); *Static Random Access Memory* (SRAM); *Digital-to-Analog Converter* (DAC); *Analog-to-Digital Converter* (ADC) e outros periféricos de entradas e saídas.

Normalmente as implementações básicas com um microcontrolador se baseiam em coleta de informações do ambiente, através de sensores ligados aos canais ADC ou entradas digitais. Através da sequência lógica dos programas, as entradas são analisadas e as respostas são praticadas pelos atuadores nos canais DAC ou nas saídas digitais.

A empresa Atmel é a fabricante do microcontrolador ATmega328. Este micro-controlador possui a arquitetura *Reduced Instruction Set Computer* (RISC), ou seja, número de instruções reduzidas, mas com a velocidade de execução acelerada e com o mesmo intervalo de tempo. Possui as seguintes características:

- 32 registradores de propósito geral de trabalho;
- 3 temporizadores flexíveis;
- Contadores com comparador de modos;
- Porta serial SPI;
- 6 canais 10-bit conversor (ADC);
- *Watchdog* timer programável com Oscilador Interno;
- 5 modos de economia de *software* selecionável (ATMEL, 2011).

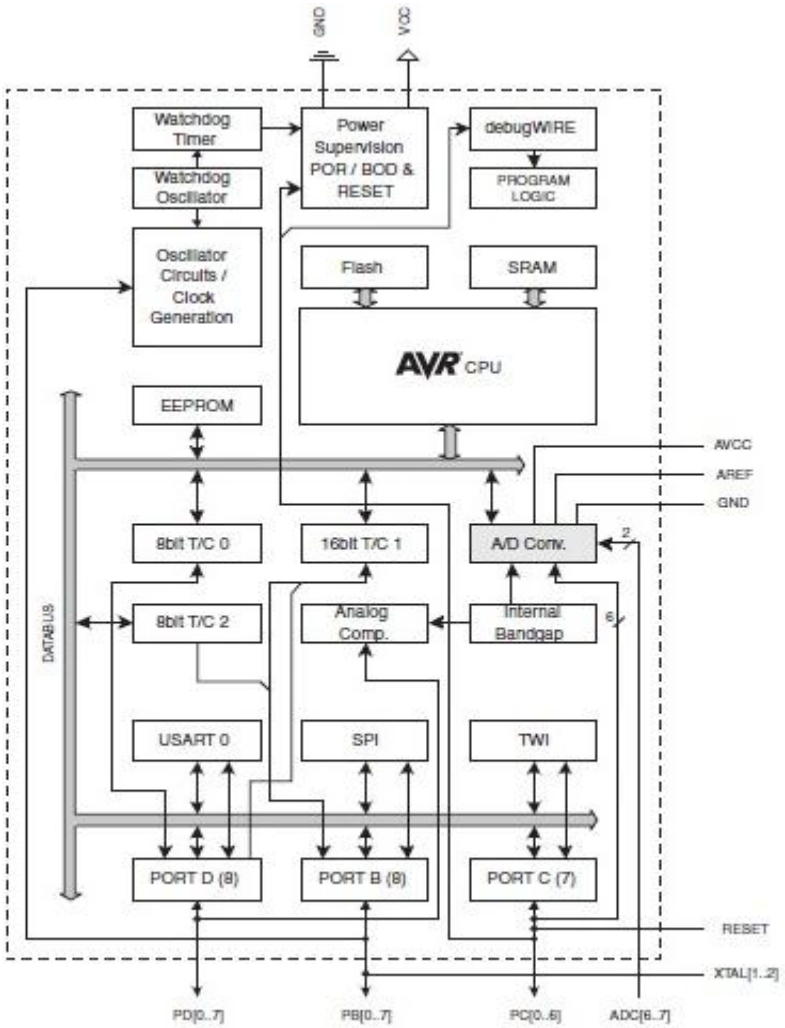
Na tabela 3 estão as demais características do ATmega328 e o diagrama da arquitetura do microcontrolador está representado na Figura 18.

TABELA 3 – Tabela de características do Atmega328

Fabricante	ATMEL
Microcontrolador	Atmega328
Tensão operacional	5 V
Tensão de alimentação	7 - 12 V
Pinos de entrada analógica	6 - 20 V
pinos I/O digitais	14 (6 podem ser saídas PWN)
Corrente continua por pino I/O	40 mA
Corrente continua para o pino 3.3 V	50 mA
Memória flash	32 KB
SRAM	2 KB
EEPROM	1KB
Frequência do clock	16 MHZ

Fonte: FONSECA; BEPPU, 2010.

FIGURA 18 - Diagrama Interno do Atmega328



Fonte: ATMEL, 2011.

2.3.1 *Arduino Duemilanove*

O Arduino surgiu em 2005 na Itália, sua ideia básica é utilizar do conceito de *hardware* e *software* disponível para toda sociedade, para construir projetos de forma mais econômica do que outras (FONSECA; BEPPU, 2010).

O Arduino é uma plataforma em que se integra o microcontrolador juntamente com seus recursos em uma placa de entrada e saída microcontrolada e desenvolvida sobre uma biblioteca simplificada para a programação C/C++. Portanto, o kit Arduino é composto por uma placa com soquetes de entrada/saída e componentes eletrônicos, microcontrolador, compilador e gravador de microcontrolador.

A grande vantagem desta ferramenta é possuir o desenvolvimento e aperfeiçoamento por uma comunidade que divulga na *internet* os códigos e os projetos como *open-source* a fim de que qualquer pessoa com um pequeno conhecimento de programação possa modificar para seu ambiente de desenvolvimento. A Figura 19 apresenta o diagrama básico da plataforma Arduino (FONSECA; BEPPU, 2010).

FIGURA 19 – Diagrama de Blocos

Fonte: FONSECA; BEPPU, 2010.

O kit de desenvolvimento Arduino Duemilanove é uma das séries do Arduino baseada no microcontrolador ATmega168 ou ATmega328, esta plataforma foi selecionada para desenvolver este trabalho, por ser de baixo custo e bastante flexível para a programação e elaboração do protótipo (FONSECA; BEPPU, 2010).

2.3.1.1 Alimentação

A alimentação do Duemilanove pode ser fornecida através da USB, da fonte externa ou dos soquetes V_{in} e G_{nd} da placa. A tensão de alimentação da fonte e dos soquetes pode variar entre 6 a 20 Volts e pela USB será de 5 Volts. A configuração de tensão para operação é ajustada automaticamente (FONSECA; BEPPU, 2010).

2.3.1.2 Memórias

As memórias utilizadas no Duemilanove são *flash*, EEPROM e SRAM já informadas no Atmega328.

2.3.1.3 Entrada e Saída

Cada um dos 14 pinos digitais do Duemilanove pode ser utilizado como entrada ou saída sendo previamente configurado. Possui resistor de *pull-up* interno sendo operado em 5 Volts, fornecendo no máximo 40mA. Além disso, alguns pinos têm funções especializadas:

- **Serial:** 0 (RX) e 1 (TX) - Usados para receber (RX) e transmitir (TX) dados seriais TTL;
- **O *Pulse-Width Modulation* (PWM):** 3, 5, 6, 9, 10, e 11 - Fornecem uma saída analógica PWM de 8-bit;
- **O *LED* 13** - Há um *LED* já montado e conectado ao pino digital 13 (FONSECA; BEPPU, 2010).

Além disso, o Arduino Duemilanove possui 6 entradas analógicas de 10 bits (1024 possibilidades), podendo variar entre 0 a 5 Volts.

3 DESENVOLVIMENTO

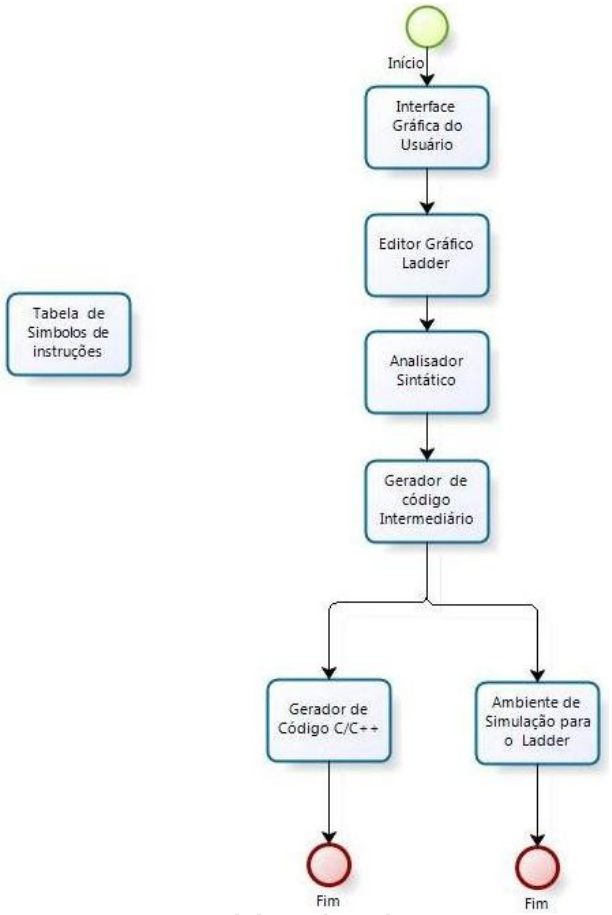
O desenvolvimento do projeto foi baseado nos estudos transcritos no capítulo de Revisão de Literatura sobre a linguagem de programação Ladder, estruturas sequenciais utilizadas para programação de Compiladores e os estudo da arquitetura do microcontrolador ATmega328.

A seguir serão descritos os passos para o desenvolvimento do aplicativo e a montagem da Plataforma de testes em *hardware*.

3.1 Aplicativo de Edição, Compilação e Simulação Ladder

A elaboração do aplicativo CSLadder Mic foi realizado de acordo com as etapas descritas no fluxograma da Figura 20.

FIGURA 20 – Fluxograma das etapas de desenvolvimento



Fonte: Elaborado pelo autor

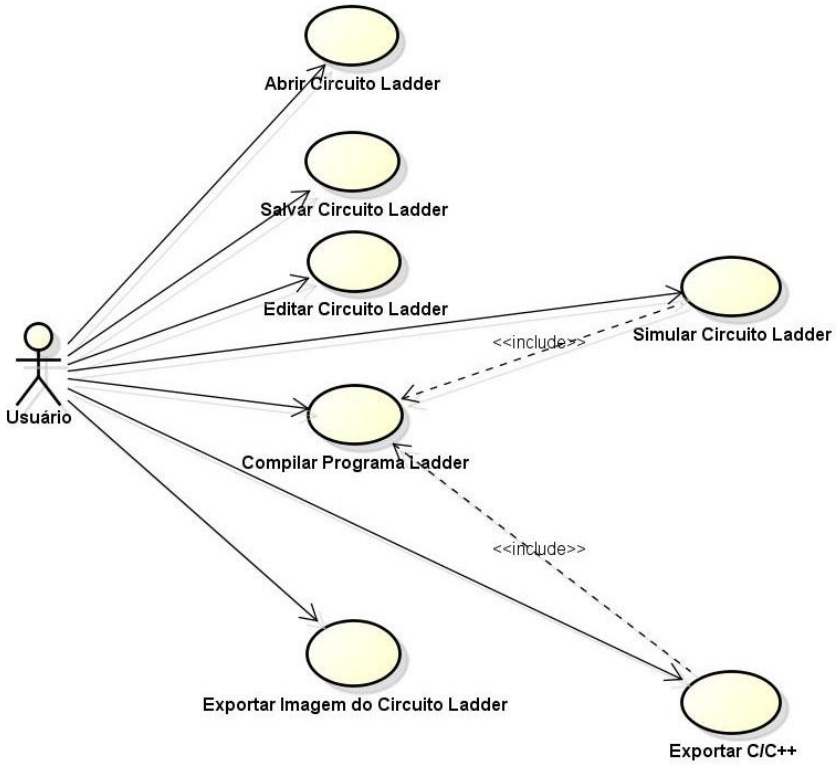
A seguir será descrito a funcionalidade de cada nível e sua respectiva programação.

3.1.1 *Interface Gráfica do Usuário*

Também chamado de *Grafic Users Interface* (GUI) é uma etapa de grande responsabilidade, pois é capaz de coletar as informações dos usuários e responder a eventuais erros que ocorram nas análises das fases seguintes.

A Figura 21 apresenta o Diagrama de Caso de Uso que resume o comportamento do aplicativo e seu ator (usuário) (LARMAN, 2003).

FIGURA 21 – Caso de Uso do CSLadder Mic

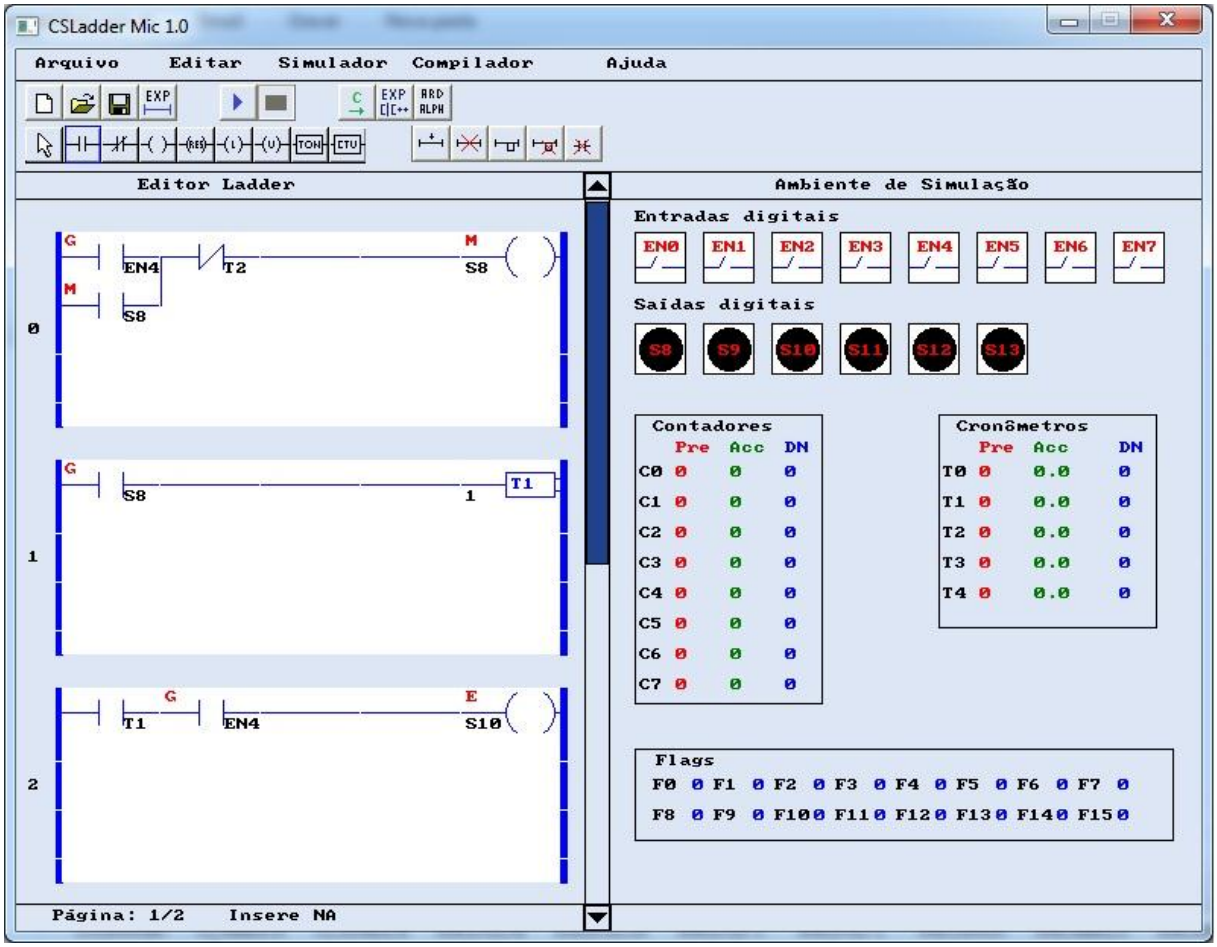


Fonte: Elaborado pelo autor

O desenvolvimento do GUI foi realizado no *Microsoft Visual Studio* 2010 programado em linguagem C/C++ com suporte para plataforma x86. Na criação da *interface* de interação com usuário foi utilizada a biblioteca *Allegro* 4.0, normalmente utilizada em jogos, facilitando a implementação de animações na simulação e todo aplicativo foi desenvolvido no Sistema Operacional *Windows* 7 Professional 64 bits.

A Figura 22 apresenta a janela principal do GUI do CSLadder Mic.

FIGURA 22 – Janela Principal do CSLadder Mic



Fonte: Elaborado pelo autor

3.1.2 *Editor Gráfico Ladder*

Esta etapa consiste na edição do programa em Ladder elaborado pelo usuário através do mouse. Sua funcionalidade é similar ao Analisador Léxico dos Compiladores, pois fornece uma estrutura para ser analisada nas etapas seguintes. Também não permite ao usuário elaborar edições em que as conexões não apresentem um caminho possível a ser energizado da barra da esquerda até a da direita, evitando assim a presença de erros sintáticos.

3.1.2.1 Considerações de Edição

Para elaboração da edição foi utilizado o conceito de polimorfismo em que as classes abstratas representam o comportamento das classes concretas que referenciam, ou seja, o uso de várias versões para o mesmo método (BRAUDE, 2009). A Figura 23 apresenta o Diagrama de Classe do aplicativo desenvolvido.

A classe abstrata é chamada de *componente* e suas derivações são subclasses que compõe a linguagem Ladder. As subclasses são representadas como: NA, NF, OTE, OTL, OTU, TEMP (TON), CONT (CTU), RES (*reset* de TEMP ou CONT) e as conexões representadas pela subclasse CONEX. Para evitar o uso de ponteiro nulo que pode acarretar problemas nas chamadas de métodos ou funções dos objetos, foi criada a subclasse SKIP em que seus objetos não apresentam nenhum comportamento, ou seja, a ausência de uma instrução ou conexão.

Na elaboração da matriz lógica foi necessário a modulação de uma matriz polimórfica da classe c*omponente*, com dimensões $4x9$. Sabendo que $M_{i,j}$, então tem-se a variação de $1 \leq i \leq 4$ e $1 \leq j \leq 9$, totalizando 36 células.

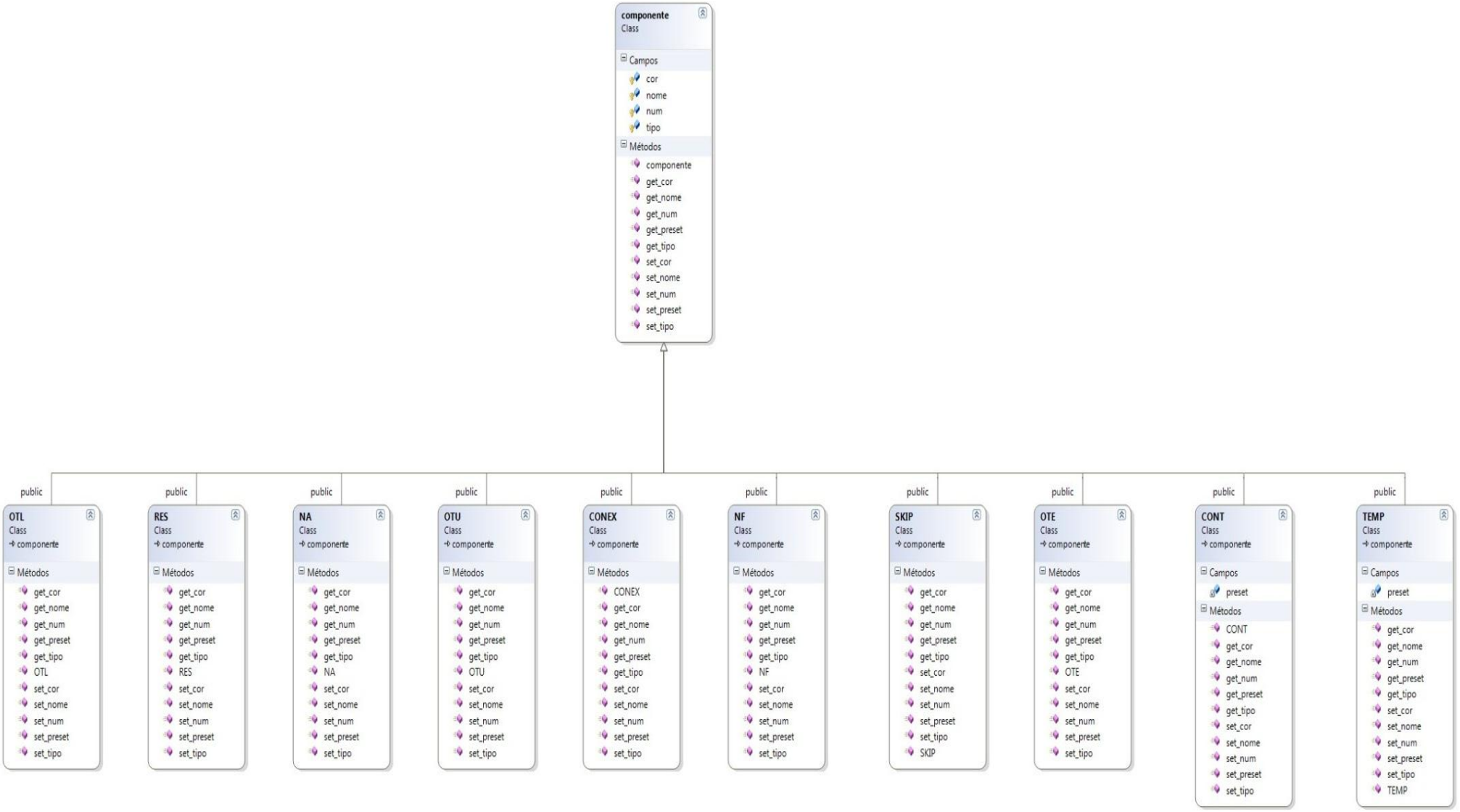
Por padrão, nas colunas ímpares apenas podem ser inseridas instruções ou conexão de continuidade e nas colunas pares somente são inseridas as conexões da linguagem Ladder referente à Figura 10.

O número de matrizes lógicas é limitado à quantidade de memória do computador. Cada matriz só poderá conter uma instrução de saída na célula $M_{1,9}$, por isso as células $M_{2,9}$, $M_{3,9}$ e $M_{4,9}$ destinadas para instruções de saída não terão funcionalidade.

A memória e os pinos de entrada e saída do microcontrolador são limitados, por isso foi necessário criar uma padronização dos pinos digitais e limitar o número de instruções utilizadas no programa Ladder. O padrão utilizado será apresentado a seguir:

- **Entradas (EN)** - são entradas digitais do microcontrolador, variando de EN0 (pino digital 0) até EN7 (pino digital 7), totalizando 8 entradas;
- **Saídas (S)** - são saídas digitais do microcontroladores, variando de S8 (pino digital 8) até S13 (pino digital 13), totalizando 6 saídas;
- **Contadores (C)** - são os contadores Ladder da instrução CTU, variando de C0 (contador 0) até C7 (contador 7), totalizando 8 contadores;

FIGURA 23 – Diagrama de Classes dos componentes da Linguagem Ladder



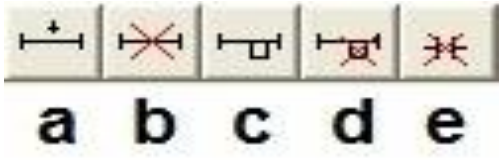
Fonte: Elaborado pelo autor

- **Temporizadores (T) ou cronômetros** - são os temporizadores Ladder da instrução TON, variando de T0 (temporizador 0) até T4 (temporizador 4), totalizando 5 temporizadores;
- ***Flags* (F)** - são bobinas internas utilizadas para expandir o número de instruções Ladder NA, NF, OTE, OTU e OTL, variando de F0 (*flag* 0) até F15 (*flag* 15), totalizando 16 *flags*.

3.1.2.2 Construção das ramificações

O usuário pode adicionar ou remover uma matriz lógica por vez, através dos botões apresentados na Figura 24 itens "a" e "b", respectivamente.

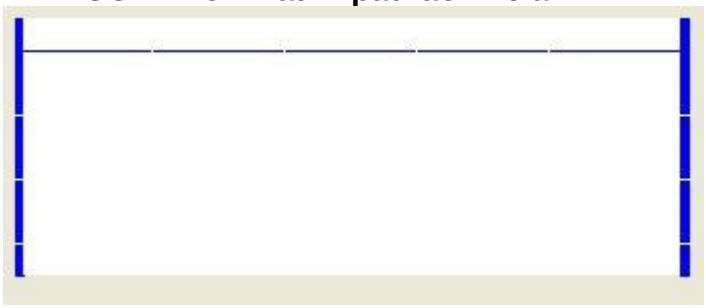
FIGURA 24 - Botões da ferramenta de edição



Fonte: Elaborado pelo autor

Inicialmente toda matriz a ser editada possui um conjunto de conexões de seguimento na primeira linha que leva a barra da esquerda à direita, como mostrado pela Figura 25.

FIGURA 25 - Matriz padrão inicial



Fonte: Elaborado pelo autor

Para construir as ramificações ou degraus, o usuário deve construir das linhas superiores às inferiores e em cada uma destas é realizada a construção das colunas da esquerda à direita. Já as desconstruções devem seguir o caminho inverso sendo realizado das linhas inferiores às superiores e para cada uma destas se destrói as colunas mais a direita.

A construção ou destruição de degraus é realizada passo-a-passo, sendo que o semicircuito que será construído ou destruído aparecerá selecionado ao usuário, para enfim decidir se deseja remover ou construir o circuito correto. Na Figura 24 itens "c" e "d", são apresentados os botões de construção e destruição da ramificação, respectivamente. O usuário deve escolher um deles e depois selecionar o local da matriz onde deseja realizar a edição.

Para facilitar a edição do programa Ladder, só é permitido ao usuário construir conexões em paralelo à esquerda e seguida de conexões em séries ou paralelas. A Figura 26 apresenta uma matriz Ladder que não pode ser construída e sua adaptação para ser construída no CSLadder Mic é representada pela Figura 27.

FIGURA 26 - Lógica em Ladder

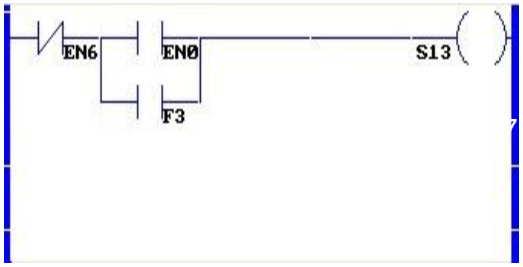
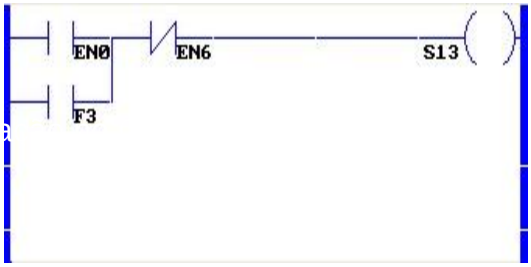


FIGURA 27 - Adaptação no CSLadder Mic



Fonte: Elaborado pelo autor

Na Figura 24 item "e" é utilizado para remover uma instrução.

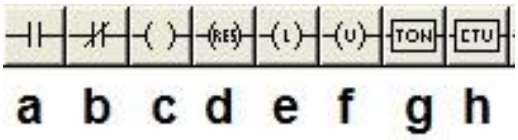
É sugerido ao usuário construir primeiramente todas as ramificações de cada matriz e em seguida inserir as instruções Ladder na mesma.

3.1.2.3 Inserção de instrução

Para inserir uma instrução de entrada ou saída o usuário deverá selecionar a instrução na barra de ferramentas representa pela Figura 28 e direcionar ao local a ser inserido, assim será substituída uma conexão de continuidade pela instrução desejada.

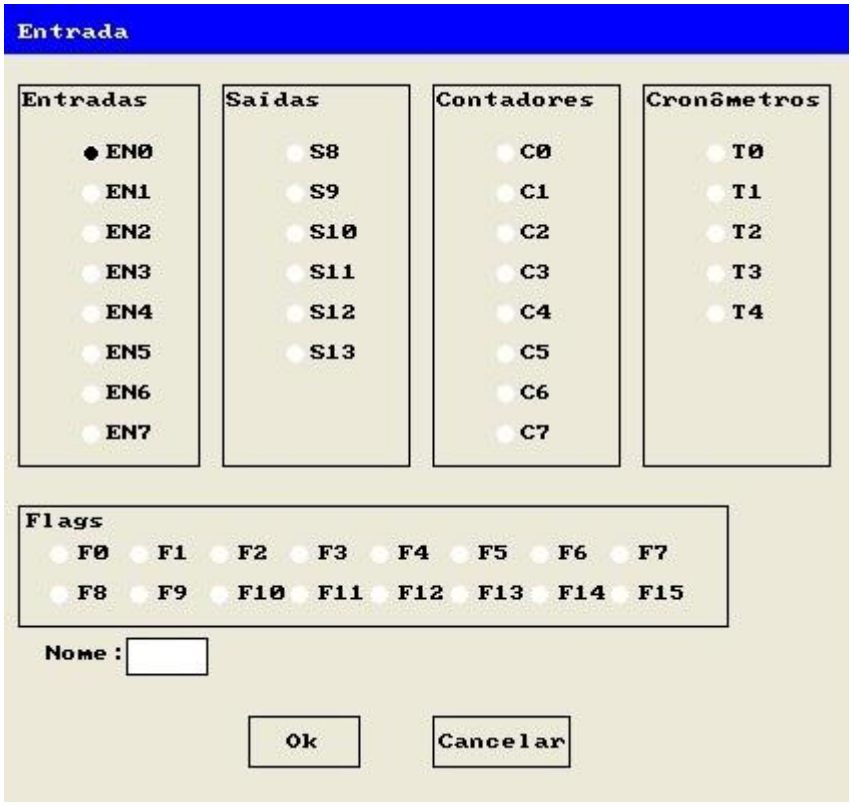
A Figura 28 itens "a" e "b", apresentam as instruções NA e NF, respectivamente. Quando inseridas em local correto aparecerá à janela representada na Figura 29 onde são mostradas as opções de instruções.

FIGURA 28 - Botões de instruções



Fonte: Elaborado pelo autor

FIGURA 29 - Inserção de instrução NA e NF

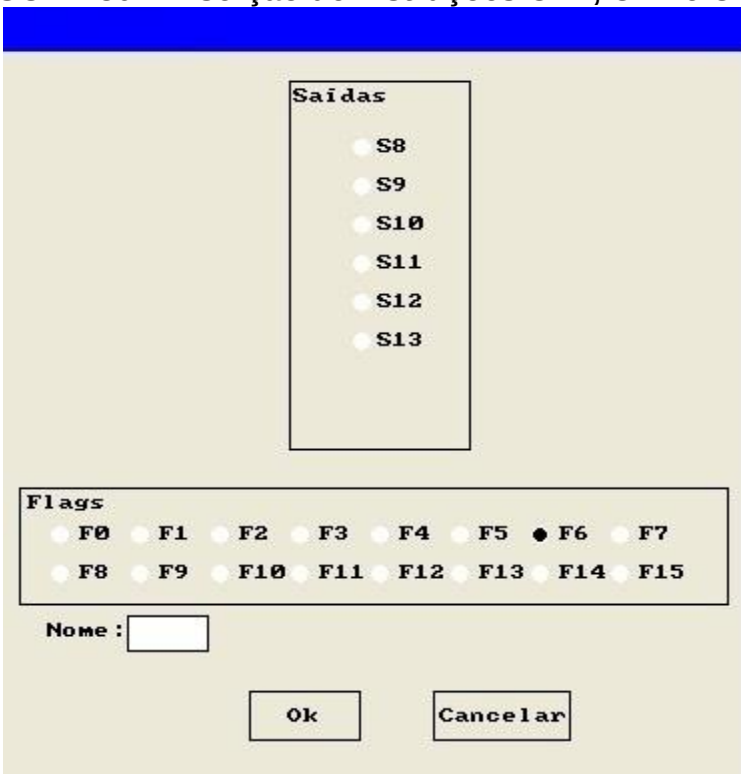


Fonte: Elaborado pelo autor

As instruções de entradas e saídas se refere aos pinos digitais do microcontrolador e seus respectivos estados lógicos. Os “Cronômetros” referem se ao estado do bit DN de um temporizador, os “Contadores” referem se ao bit DN de um contador e as “*Flags”* referem se ao estado lógico de uma *flag* utilizada no programa.

A Figura 28 itens "c" (referente à instrução OTE), "e" (L referente à instrução OTL) e "f" (U referente à instrução OTU), são as bobinas de saídas de uma matriz. Quando inseridas em local correto a janela da Figura 30 será exibida e também apresentará as opções de utilização para saídas digitais e *Flags*.

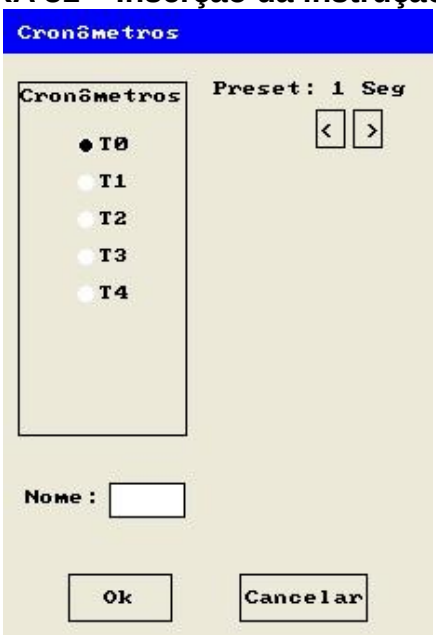
FIGURA 30 – Inserção de instruções OTE, OTL e OTU



Fonte: Elaborado pelo autor

Na Figura 28 o item "g" (TON) é utilizado para inserção de um temporizador em uma matriz. Quando inserido em local correto, será exibida a janela da Figura 31 que apresenta os temporizadores específicos e o usuário deverá definir o temporizador e o tempo em segundos do *Preset* do mesmo.

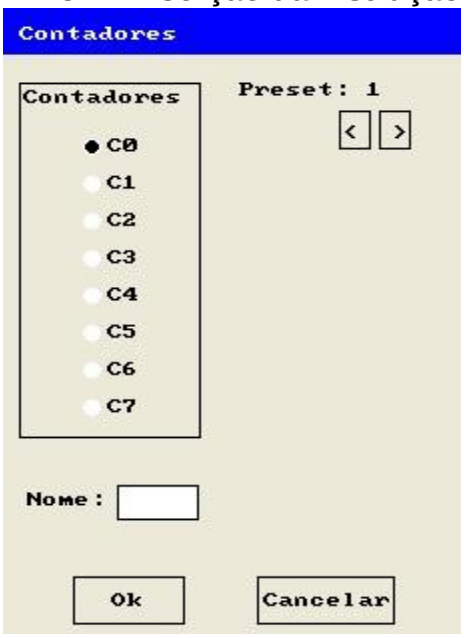
FIGURA 31 – Inserção da instrução TON



Fonte: Elaborado pelo autor

Na Figura 28 o item "h" (CTU) é utilizado para inserção de um contador em uma matriz. Quando inserido em local correto, aparecerá à janela da Figura 32 que apresenta os contadores específicos e o usuário deverá definir o contador e o valor do *Preset* do mesmo.

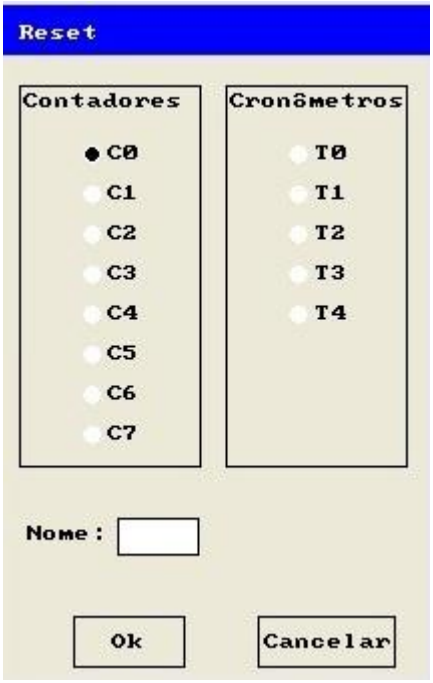
FIGURA 32 – Inserção da instrução CTU



Fonte: Elaborado pelo autor

Na Figura 28 no item "d" foi criada a instrução *Reset* (RES) de uma matriz, para que seja possível reiniciar um contador ou temporizador específicos. Quando inserido em local correto, aparecerá a janela da Figura 33 onde apresenta as opções para que os contadores ou cronômetros utilizados possam voltar ao estado inicial.

FIGURA 33 – Inserção de instrução RES



Fonte: Elaborado pelo autor

3.1.3 *Tabela de Símbolos de Instruções*

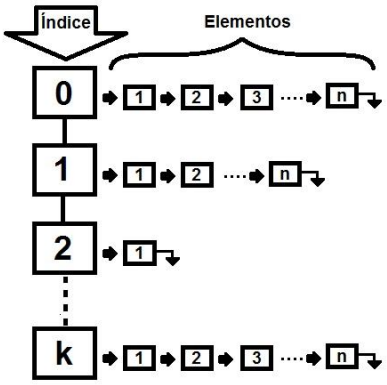
A Tabela de Símbolos de Instruções (TSI) possui o funcionamento similar às demais Tabelas de Símbolos comuns aos Compiladores. Porém, não possui estrutura de inserção ou remoção de escopo, pois o LD possui um único escopo.

É uma estrutura de auxílio para as fases seguintes de compilação, sua inserção é inicializada toda vez que o usuário necessitar compilar o programa.

A estrutura foi implementada através de várias listas encadeadas que é uma abstração de dado em que o primeiro elemento aponta para o próximo da lista e o último elemento aponta para o nulo. Como mostrado pela Figura 34, cada item implementado pelo usuário no CSLadder Mic possui uma lista sendo um atributo da classe *tabela_simbolos*, onde cada posição refere ao índice do elemento escolhido na programação sendo:

- Lista FLG com 16 posições, referente às *flags*;
- Lista TEMP com 5 posições, referente aos temporizadores;
- Lista E_S com 14 posições, referente a entradas e saídas do microcontrolador;
- Lista CONT com 8 posições, referente aos contadores.

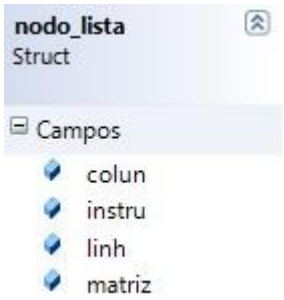
FIGURA 34 – Listas dos componentes



Fonte: Elaborado pelo autor

Cada elemento da lista apresenta as informações da Figura 35, ou seja, possui a matriz, linha, coluna e a instrução de cada elemento programado em Ladder.

FIGURA 35 – Elemento das listas



Fonte: Elaborado pelo autor

A inserção na Tabela de Símbolos é feita da seguinte forma, primeiramente é inserida todas as instruções de saída das matrizes em sua respectiva lista, exceto a instrução RES, logo em seguida é inserida todas as instruções de entrada das matrizes em sua respectiva lista.

3.1.4 *Analisador Sintático*

O Analisador Sintático (AS) consiste em verificar os possíveis erros sintáticos realizados pelo usuário na edição do LD. Será analisado todo o programa do usuário e caso haja erros, será exibida uma mensagem com o número de erros e os locais onde se encontram.

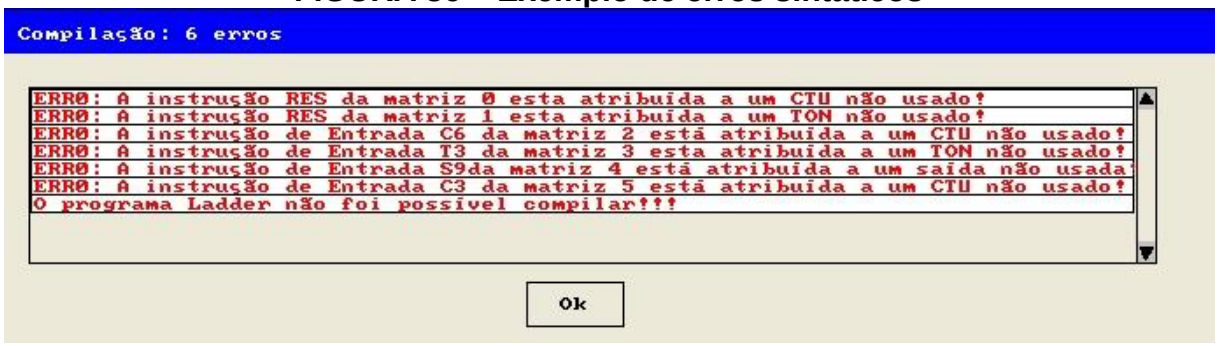
Para ser feita esta análise não foi necessário à utilização de árvore sintática abstrata comumente utilizada nos Compiladores, pois na edição do programa em Ladder, as matrizes polimórficas apresentam uma estrutura de abstração eficiente de fácil acesso para ser analisada. Esta análise ficou menos sobrecarregada, tendo em vista que, no momento de edição não será possível o usuário criar programas com conexões incompatíveis. Para ser feita a análise sintática, além da utilização das matrizes, também utiliza da TSI preenchida. Foi criada uma variável inteira chamada erro para contabilizar o número de erros sintáticos.

A seguir serão descritos os passos do Analisador Sintático:

i. **Passo 1** - Verificar a existência de uma instrução de saída RES nas matrizes onde existe um temporizador ou contador utilizado como instrução de saída através da TSI. Toda vez que uma instrução RES não for atribuída a um contador ou temporizador a variável erro será incrementada e será adicionada uma mensagem de erro mostrando o local onde se encontra a instrução RES incorreta;
ii. **Passo 2** - Verificar se uma instrução TON está associada várias vezes a um mesmo temporizador, através da TSI. Na ocorrência de multiplicidade é incrementada a variável erro e é adicionada uma mensagem com os locais onde o problema está ocorrendo;
iii. **Passo 3** - Verificar se uma instrução CTU está associada várias vezes a um mesmo contador, através da TSI. Na ocorrência de multiplicidade é incrementada a variável erro e é adicionada uma mensagem com os locais onde o problema está ocorrendo;
iv. **Passo 4** - Verificar se as instruções de entrada ligadas aos pinos digitais S e as *Flags*, estão sendo utilizadas como instrução de saída em alguma matriz do programa através da TSI. Caso as *flags* ou saídas estejam sendo utilizadas como instrução NA ou NF e não forem usadas em nenhum momento como instrução de saída, a variável erro é incrementada e será adicionada uma mensagem com os locais onde mostra as instruções de entradas inúteis para a programação;

Após a realização de todos os passos é exibida ao usuário uma janela exemplificada pela Figura 36 com os possíveis erros e a não ocorrência de erros exibe uma mensagem em que o programa foi compilado corretamente.

FIGURA 36 – Exemplo de erros sintáticos



Fonte: Elaborado pelo autor

3.1.5 *Gerador de Código Intermediário*

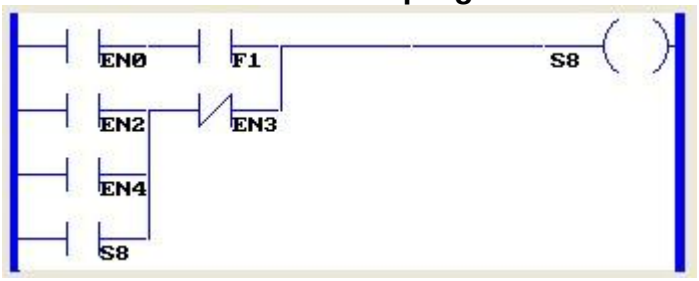
O Gerador de Código Intermediário (GCI) consiste em capturar as expressões booleanas e os comandos fornecidos por cada matriz lógica. O GCI é necessário, pois consegue fornecer ao ambiente de simulação uma sequência lógica eficiente e ainda facilita a geração de código C/ C++. Também deixa uma estrutura pronta, para que nos trabalhos futuros possa ser utilizado para otimizações de códigos.

O código intermediário é somente gerado quando não ocorrem erros sintáticos e é um pré-requisito para as etapas seguintes do CSLadder Mic.

Para cada matriz lógica que possui instrução de saída é gerado o código intermediário equivalente com sua expressão booleana. Numa matriz sem instrução de saída, sua expressão não terá funcionalidade na aplicação final, sendo descartada a geração da mesma.

A seguir serão descritos os passos para geração de código intermediário para cada matriz, exemplificado por um trecho de um programa Ladder mostrado pela Figura 37.

FIGURA 37 – Trecho de um programa em Ladder



Fonte: Elaborado pelo autor

i. **Passo 1** - Consiste na criação de um grafo direcionado que representa os fluxos contendo todos os caminhos correntes entre a barra da esquerda à direita, sabendo que:

> Um grafo é uma forma de representar relacionamentos que existem entre pares de objetos. Isto é um conjunto de objetos, chamado de vértices, juntamente com uma coleção de conexões entre pares de vértices.
> [...]
> Visto de forma abstrata, um grafo **G** é simplesmente um conjunto **V** de vértices e uma coleção **E** de pares de vértices de **V**, chamado de arestas. Assim, um grafo é uma forma de representar conexões ou relações entre pares de objetos de algum conjunto (GOODRICH; TAMASSIA, 2007, p.508).

Para construção do grafo se faz uma varredura na matriz lógica localizando as instruções NA ou NF que estão em sequência, até encontrar uma conexão que não seja de seguimento ou de continuidade, criando assim os vértices do grafo e ao final se cria um vértice F que representa o fim da expressão como exemplificado pela Figura 38.

Na Figura 38 cada vértice possui os seguintes elementos e instruções:

- **A** - EN0 (NA) e F1(NA);
- **B** - EN2 (NA);
- **C** - EN3 (NF);
- **D** - EN4 (NA);
- **E** - S8 (NA);
- **F** - Vértice acrescentado representando o final da expressão.

Posteriormente, se faz uma varredura nas conexões das colunas pares da matriz, criando arestas direcionadas que representam os possíveis

caminhos que levam as correntes da barra da esquerda à direita. Como mostrado pela Figura 39.

Ao final se tem o grafo apresentado pela Figura 40.

FIGURA 38 – Exemplificação da criação dos vértices

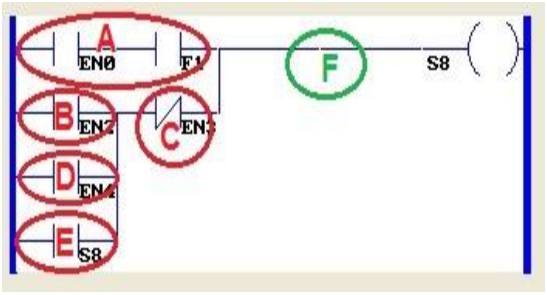
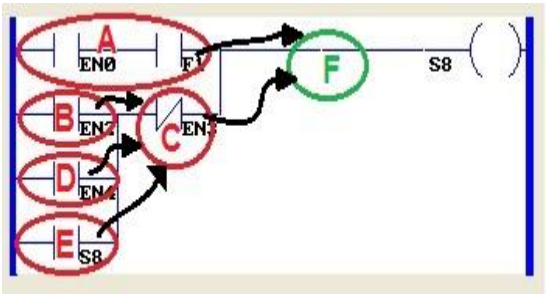
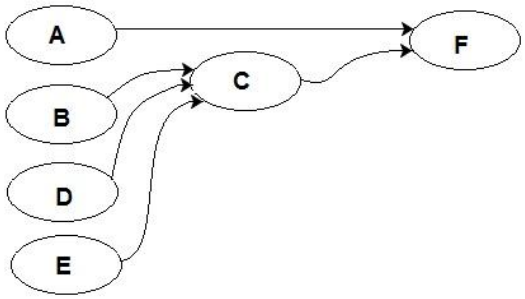


FIGURA 39 – Exemplificação da criação das arestas



Fonte: Elaborado pelo autor

FIGURA 40 – Grafo gerado através de um trecho Ladder



Fonte: Elaborado pelo autor

Para finalizar o passo é aplicado o algoritmo de Lin-Alderson mostrado a seguir, em que dado um vértice v_i inicial e um vértice v_f final, retorna todos os possíveis caminhos entre estes:

- *Dados os vértices inicial vi e final vf;*
- *P1: Incluir vi no conjunto de vértices visitados:* $S = \{vi\}$;
- *P2: Adicionar a S o primeiro vértice viável* $(vj \in S)$ *destino do último vértice inserido em S (vi, na primeira iteração);*
- *P3: Repetir o passo P2 enquanto houver vértice viável que seja destino do último vértice em S e enquanto* $vf \notin S$;
- *P4: Se* $vf \in S$, *um caminho entre vi e vf foi encontrado e deve ser armazenado;*

- *P5: Quando não houver vértice viável ou se vf foi encontrado, realizar um backtracking, removendo de S o último vértice adicionado. Retornar ao passo P2. O processo termina quando S contém apenas o vértice vi e não existir mais nenhum vértice viável a ser adicionado a S* (ALDERSON; LIN, 1973).

Este algoritmo é aplicado como v_i sendo todos os vértices que estão conectados a barra da esquerda, pela Figura 39 será realizado para os vértices A, B, D e E. O v_f será sempre o vértice F criado.

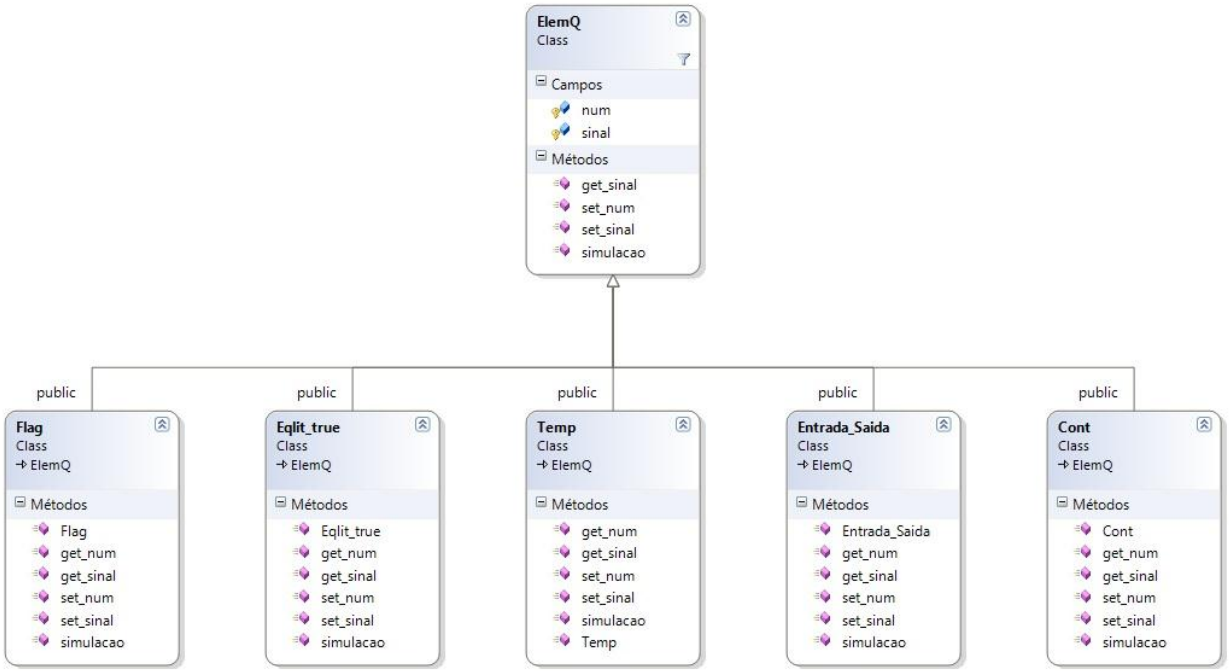
Ao final deste passo serão obtidos todos os possíveis caminhos correntes da matriz lógica. Para o exemplo da Figura 40 terá os seguintes caminhos:

- **Caminho 1**: A - F;
- **Caminho 2**: B - C - F;
- **Caminho 3**: D - C - F;
- **Caminho 4**: E - C - F.

O grafo é representado através de uma matriz de adjacências.

ii. **Passo 2** - Para cada caminho resultante do passo 1, cria-se um vetor polimórfico dinâmico, contendo todos elementos de cada vértice e sua respectiva instrução NA ou NF, sendo os operando da expressão. Observa-se que o vértice F é de auxílio, não contendo elementos. A Figura 41 apresenta o Diagrama de Classe dos possíveis elementos deste vetor, tendo como classe abstrata *ElemQ* e as classes derivadas os possíveis elementos. No final é adicionada uma variável objeto da classe *ElemQ* que representa a instrução de saída da matriz lógica com sua respectiva instrução OTE; OTL; OTU; RES; TON ou CTU através do método *set_sinal*, que atribui uma constante para cada tipo de instrução.

FIGURA 41 – Diagrama de Classe ElemQ



Fonte: Elaborado pelo autor

iii. **Passo 3** - Para cada vetor criado no passo 2, se coloca o operador *and* entre seus elementos. Em seguida, se o número de vetores de expressão for maior que 1 anexa o operando *or* entre cada vetor e por último exibe o comando a ser realizado se caso a expressão seja verdadeira, através da variável polimórfica adicionada no final do passo 2.

Para exemplo da Figura 37 é obtido o código intermediário que é descrito pela Tabela 4.

TABELA 4 – Código Intermediário Gerado

Expressão
(E0 AND F1) OR (E2 AND $\overline{E3}$) OR (E4 AND $\overline{E3}$) OR (S8 AND $\overline{E3}$)
Comando
SET S8

Fonte: Elaborado pelo autor

3.1.6 *Ambiente de Simulação da Linguagem Ladder*

Este ambiente é uma das saídas do CSLadder Mic. Consiste em botões de entradas gráficos e painéis de visualização dos atributos das instruções LD. O Ambiente de Simulação, Depurador, *Debugger* ou Simulador possui um objetivo didático, pois simula o funcionamento do programa Ladder para que o usuário possa visualizar previamente as ações realizadas pelo seu programa para uma aplicação específica.

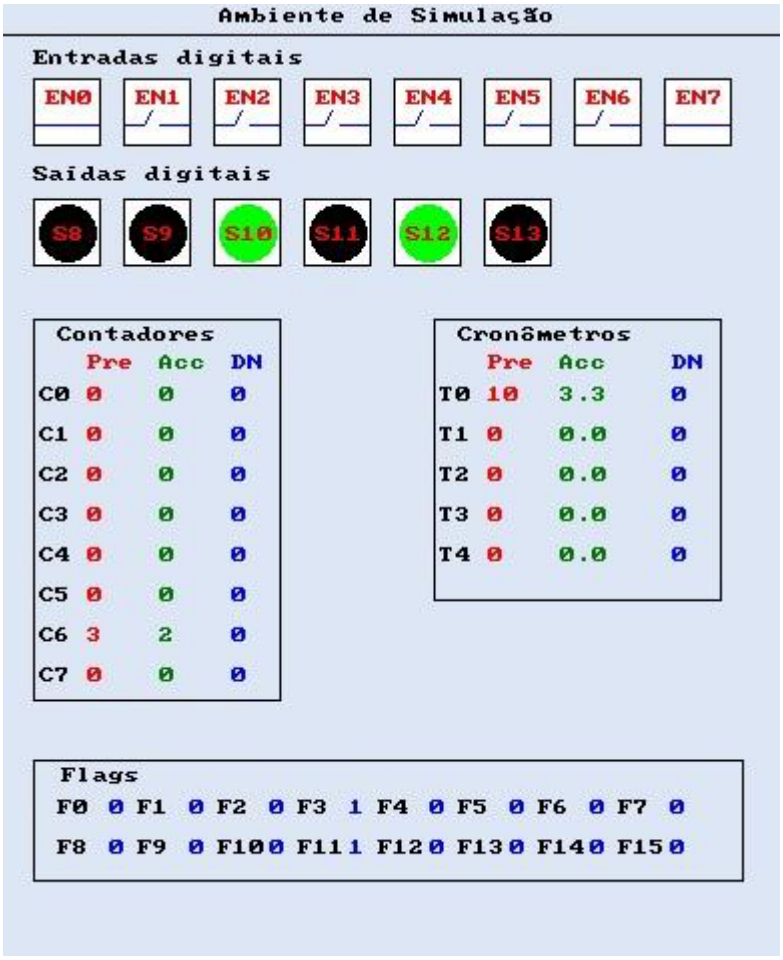
A Figura 42 apresenta o painel de simulação interativo.

A seguir será descrita a funcionalidade de cada componente do ambiente de simulação.

- **Entradas digitais -** EN0 até EN7 são botões virtuais que simulam chaves ligadas às entradas digitais disponíveis. A alteração do estado da chave é realizada pelo usuário através do clique do mouse. Quando a chave está fechada o estado da entrada é verdadeiro e quando aberta é falso;
- **Saídas digitais** - S8 até S13 são bobinas visuais, que simulam *led's* virtuais ligados às saídas digitais disponíveis. A alteração de estado dos *led's* é realizada através da lógica antecipadamente programada, sendo que quando o *led* está com a cor verde o estado da saída é verdadeiro ou alto e quando preto é falso ou baixo;
- **Contadores** - C0 até C8 são contadores visuais, que através de uma tabela simulam os valores dos atributos internos da instrução CTU. A alteração dos valores dos atributos é realizada através da lógica antecipadamente programada, sendo que em vermelho possui o valor do *Preset*, verde valor do *Accum* e azul o estado lógico do bit DN;
- **Cronômetros** - T0 até T4 são temporizadores visuais, que através de uma tabela, simulam os valores dos atributos internos da instrução TON. A Alteração dos valores dos atributos é realizada através da lógica antecipadamente programada, sendo que em vermelho possui o valor do *Preset*, verde valor do *Accum* e azul o estado lógico do bit DN;
- ***Flags*** - F0 até F15 são bobinas internas visuais que simulam o estado do bit das *flags.* A alteração é realizada através da lógica antecipadamente

programada, sendo que em azul possui seu estado lógico em 1 quando verdadeiro e 0 quando falso.

FIGURA 42 – Painel de Simulação



Fonte: Elaborado pelo autor

Para os itens do painel foi criado um vetor para cada atributo que possa ser exibido pelo simulador como mostrado a seguir:

- **Entradas e saídas digitais** - são representadas por um vetor booleano com 14 posições;
- **Contadores** - Os valores dos *Preset´s* e *Accum´s* são representados cada com um vetor inteiro com 8 posições e o valor do bit DN é representado por um vetor booleano com 8 posições;
- **Temporizadores** - os valores dos *Preset´s* e Accum´s são representados cada com um vetor inteiro e real respectivamente, com 5 posições e o valor do bit DN é representado por um vetor booleano com 5 posições;

- ***Flags*** - são representadas por um vetor booleano com 16 posições.

Inicialmente os valores dos atributos são zerados, exceto o caso dos *Preset´s*; contadores; temporizadores e EN digitais que são determinados pelo usuário previamente ou no momento de simulação.

A simulação é realizada através de um laço de repetição, em que todo momento se faz a leitura das expressões booleanas geradas pelo GCI. Se algum comando causar alteração nos atributos, os mesmos serão atualizados no painel, obtendo uma simulação animada.

Os temporizadores contabilizam o tempo em paralelo com outras ações desempenhadas pelo programa, sendo necessária a criação de uma *thread*[1] para cada temporizador utilizado no LD, atualizando seus atributos em tempo real.

3.1.7 *Gerador de Código C/C++*

Este gerador é a outra saída do CSLadder Mic sua função se baseia em juntar o TSI com o GCI finalizado e gerar um código na linguagem C/C++ similar as ações do programa fonte em Ladder, para que possa ser compilado no compilador da plataforma Arduino Duemilanove (Arduino *Alpha*) e gravar o código hexadecimal no ATmega328.

Sabendo que este trabalho possui um objetivo didático, foi criado uma biblioteca *csladdermic.h* apresentada no Apêndice A. Onde suas classes modelam as configurações e os sinais lógicos das portas; *flags*; temporizadores e contadores e cada um deste possui uma classe específica. Esta biblioteca tem a finalidade de aproximação do programa fonte Ladder com o código objeto C/C++.

A seguir serão descritos os passos da geração de código C/C++ com a utilização da biblioteca *csladdermic.h*:

i. **Passo 1** - Consiste em através da TSI, declarar um identificador no escopo global para cada item utilizado através das listas da TSI. Sendo que, para cada entrada utilizada declarar um objeto da classe *Entrada* e instancia-lo. Para cada saída declarar um objeto da classe *Saída* e instancia-lo. O

[1] Unidade básica de processamento independente (SILBERSCHATZ, 2008).

mesmo procedimento deve ser aplicado para a *flag* (classe *Flag*), Temporizador (classe *Ton*) e Contador(classe *Cont*).

Foi utilizada a biblioteca *pt.h* que possui suporte para a criação de funções de execução em paralelo à aplicação principal, com *thread´s*. Desta forma para cada temporizador é necessário declarar duas variáveis, ou seja, uma booleana para verificar a ativação da contagem de tempo e outra *thread* para utilizar como parâmetro na função *thread*. A função *thread* é responsável pela contagem do tempo de cada temporizador.

O Apêndice C apresenta um trecho do código objeto C/C++ com a exemplificação do Passo 1;

ii. **Passo 2** - Fornecer as configurações inicias das portas digitais de entrada; saída e dos temporizadores que utilizam *thread´s*, dentro do escopo do método setup. Nas entradas digitais foi utilizado resistor interno de *pull-up* para proteção do microcontrolador.

O Apêndice D apresenta um trecho do código C/C++ objeto com a finalização do passo 2;

iii. **Passo 3** - Criação do fluxo principal do programa objeto no escopo do método *loop*. Com auxílio da TSI, é realizada a leitura de cada entrada digital utilizada. Através da assistência do GCI é criado o comando *if* para cada código intermediário gerado. A condição do *if* é similar a expressão de cada GCI e o comando do *if* é similar ao comando do GCI. Assim, o comando será executado apenas se a condição for verdadeira. Nas instruções OTE, TON e CTU terão o *else* referente ao *if*, pois nestas instruções quando a condição for falsa é necessário ter direção adversa ao comando do *if*. No código intermediário exibido pela Tabela 4 gera o trecho em C/C++ mostrado no Apêndice E ao final do passo 3.

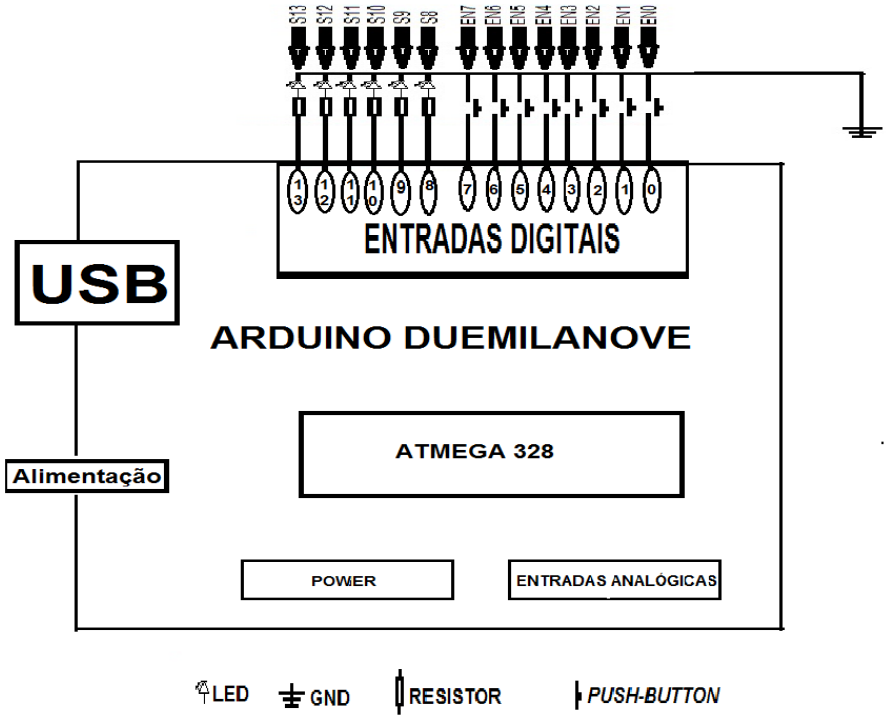
3.2 Plataforma de testes

Esta plataforma foi montada em *hardware* para testar os programas objetos gerados pelo CSLadder Mic, quanto seu desempenho e eficiência na automatização proposta. Após o programa objeto ser compilado para o Arduino *Alpha*, seu código hexadecimal é inserido no microcontrolador. O protótipo foi montado com a placa da

plataforma Arduino Duemilanove, para facilitar a gravação imediata de programas diversos no microcontrolador para os testes.

A Figura 43 apresenta o diagrama de bloco do circuito montado, onde nas entradas digitais EN0 a EN7 estão conectadas a oito *push button´s* (botoeiras), com o objetivo de funcionamento similar aos sensores digitais como: chaves de fim de curso; botões; termostatos; sensores óticos e etc. Nas saídas digitais S8 a S13 foram adicionados seis led´s em série com resistências que limitam a corrente, com a finalidade de funcionar como atuadores digitais: contatores; solenoides; relés; lâmpadas; sirenes; motores e etc.

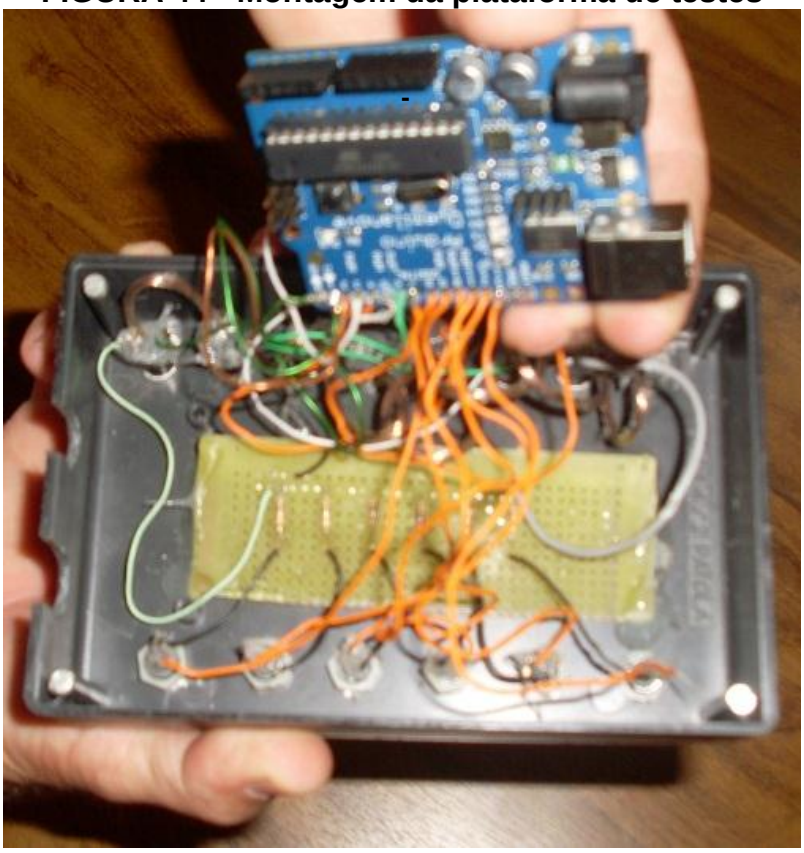
FIGURA 43 – Diagrama de bloco da plataforma de testes



Fonte: Elaborado pelo autor

A Figura 44 apresenta a foto com a montagem do circuito armazenado em uma caixa plástica e na Figura 45 exibe a imagem com a vista superior da plataforma.

FIGURA 44 - Montagem da plataforma de testes



Fonte: Foto do autor

FIGURA 45 - Plataforma de testes finalizada

Fonte: Foto do autor

4 RESULTADOS OBTIDOS

Este capítulo demonstra alguns testes práticos de automatização implementados no CSLadder Mic e em seguida serão apresentadas as dificuldades encontradas e os resultados alcançados.

4.1 Testes

O objetivo do presente projeto foi a construção de uma ferramenta didática para a aprendizagem. Por isso, foram propostos alguns problemas de automatização de CLP, que puderam ser editados e compilados no CSLadder Mic através de seus diagramas Ladder. As simulações foram realizadas no Ambiente de Simulação e posteriormente, os códigos C/C++ gerados, foram exportados e compilados no Arduino *Alpha* e embutidos os hexadecimais na plataforma de testes.

Os testes propostos foram selecionados de forma a utilizar todas as instruções existentes no CSLadder Mic que foi baseado nas instruções da linguagem Ladder.

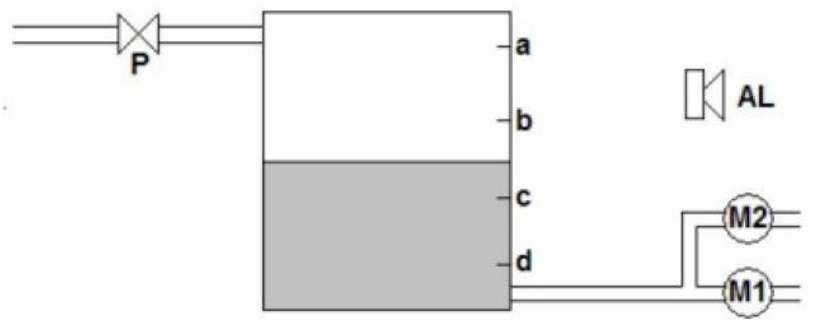
Todos os testes utilizados neste projeto foram filmados em três etapas distintas, sendo elas: a implementação do programa fonte no CSLadder Mic, a simulação do problema implementado através do ambiente de simulação e por último a gravação do funcionamento na plataforma de testes.

4.1.1 *Sistema de Reservatório*

> Implementar um programa em Ladder para o CLP em um sistema de reservatório composto de uma válvula de entrada P, duas bombas (acionadas por M1 e M2), um alarme AL e quatro sensores de nível (a, b, c, d), conforme ilustrado na Figura 46. As condições de funcionamento são as seguintes: se o nível for 'a', então fecha-se a válvula P. Se o nível for inferior a 'b', então abre-se a válvula P. Acima de 'b', M1 e M2 bombeiam. Abaixo de 'b', somente M1 bombeia. Abaixo de 'c', soa o alarme AL. Em 'd', nenhuma das bombas deverá funcionar (SILVEIRA; SANTOS, 2009, p.114-115).

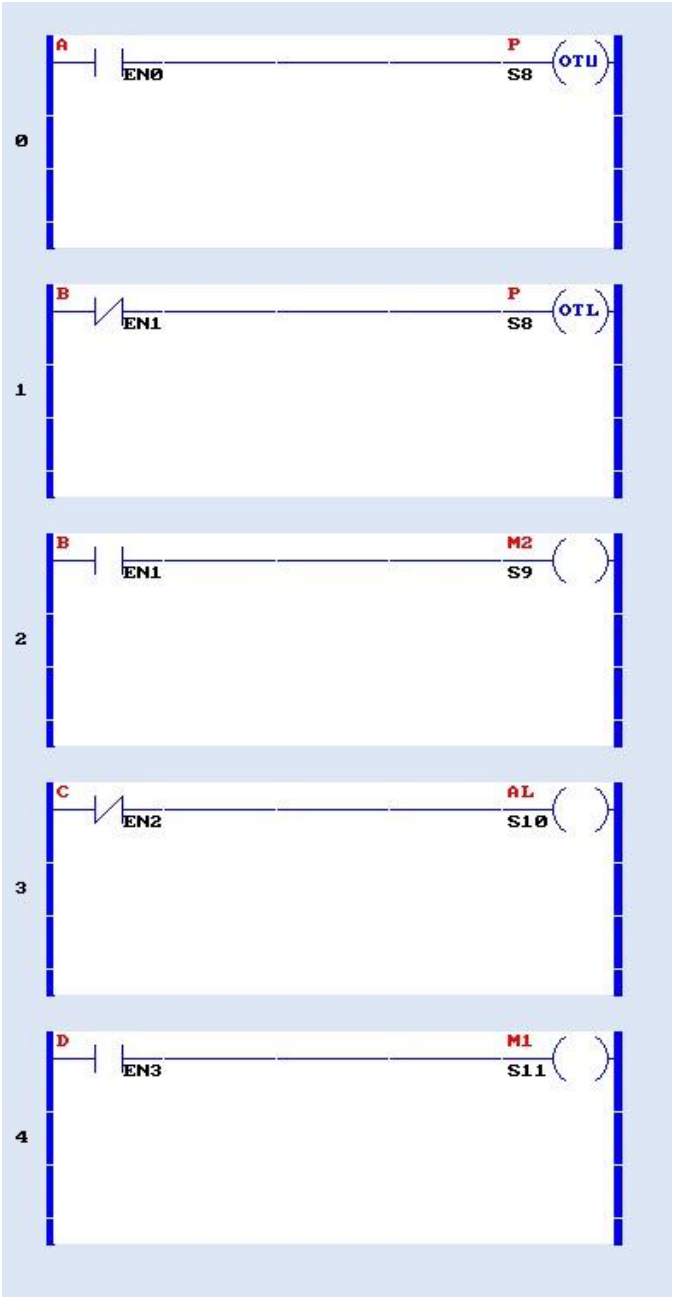
A Figura 47 apresenta a solução em LD do problema e o Apêndice F apresenta o código objeto C/C++ do problema proposto.

FIGURA 46 - Sistema de Reservatório



Fonte: SILVEIRA; SANTOS, 2009.

FIGURA 47 - Programa fonte em Ladder para o Sistema de Reservatório



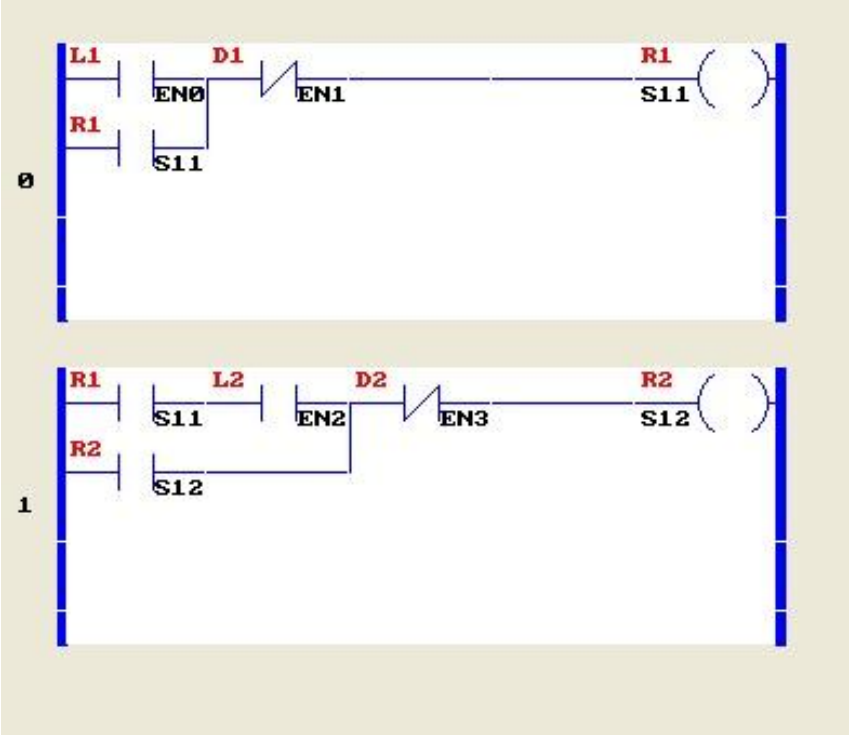
Fonte: Elaborado pelo autor

4.1.2 *Relés com Dependência*

> Elaborar um programa CLP para controlar dois relés (R1 e R2) de tal maneira que R1 pode atuar de forma independente e R2 só pode atuar se R1 estiver ligado, mas pode continuar ligado após o desligamento de R1. Os relés são ligados pelas botoeiras L1 e L2, e são desligados pelas botoeiras D1 e D2 (SILVEIRA; SANTOS, 2009, p.115).

A Figura 48 apresenta a solução em LD do problema e no apêndice G apresenta o código objeto C/C++ gerado.

FIGURA 48 - Programa fonte em Ladder para os Relés com Dependência



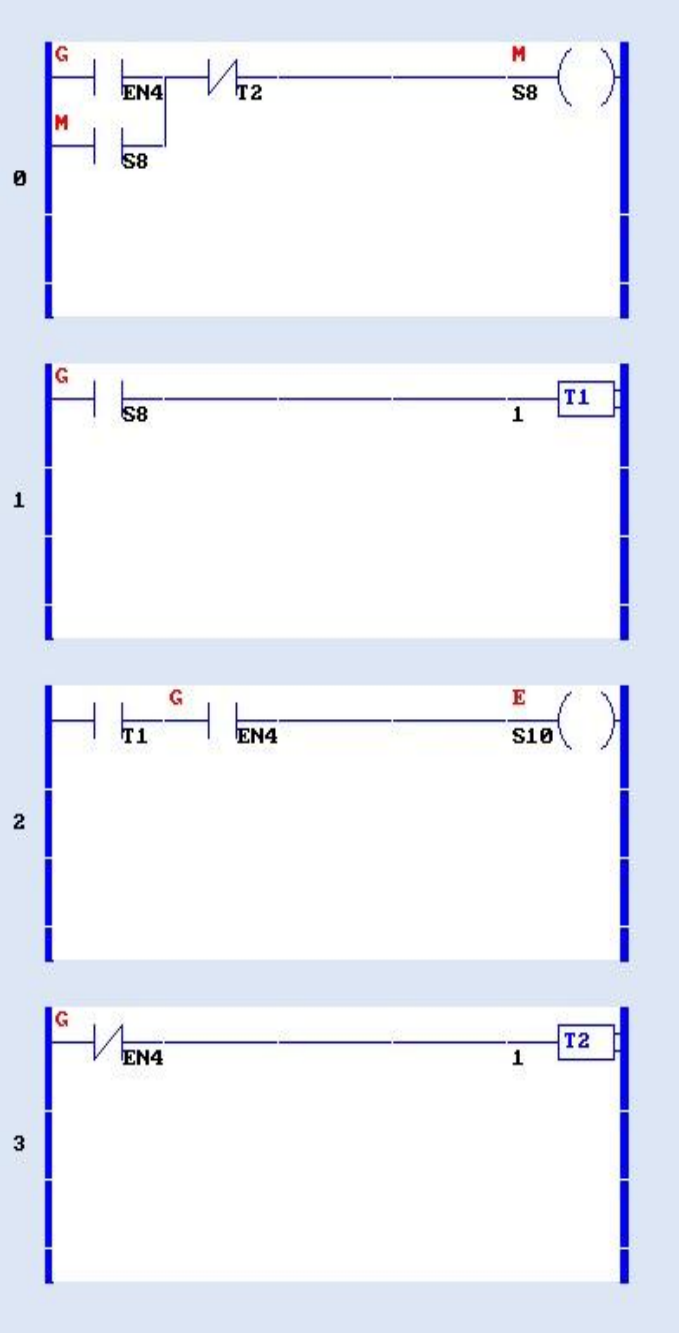
Fonte: Elaborado pelo autor

4.1.3 *Máquina de Solda*

> Em uma máquina de solda, há dois elementos controlados por um CLP: um contator (A) para fechamento do arco, e um relé (E) para avanço do motor do eletrodo. Quando o operador aciona o gatilho (G), a máquina deve entrar em funcionamento, atuando primeiramente o motor e 1 segundo após atuar o eletrodo. No momento em que o operador solta o gatilho, uma operação reversa deve ocorrer, ou seja, primeiramente desliga-se o eletrodo e após 1 segundo desliga-se o motor. Com base nestas informações elabore um programa CLP para realizar tal controle (SILVEIRA; SANTOS, 2009, p.116).

A Figura 49 apresenta a solução em Ladder para o problema proposto e o Apêndice H apresenta o código objeto C/C++ gerado.

FIGURA 49 - Programa fonte em Ladder para a Máquina de Solda



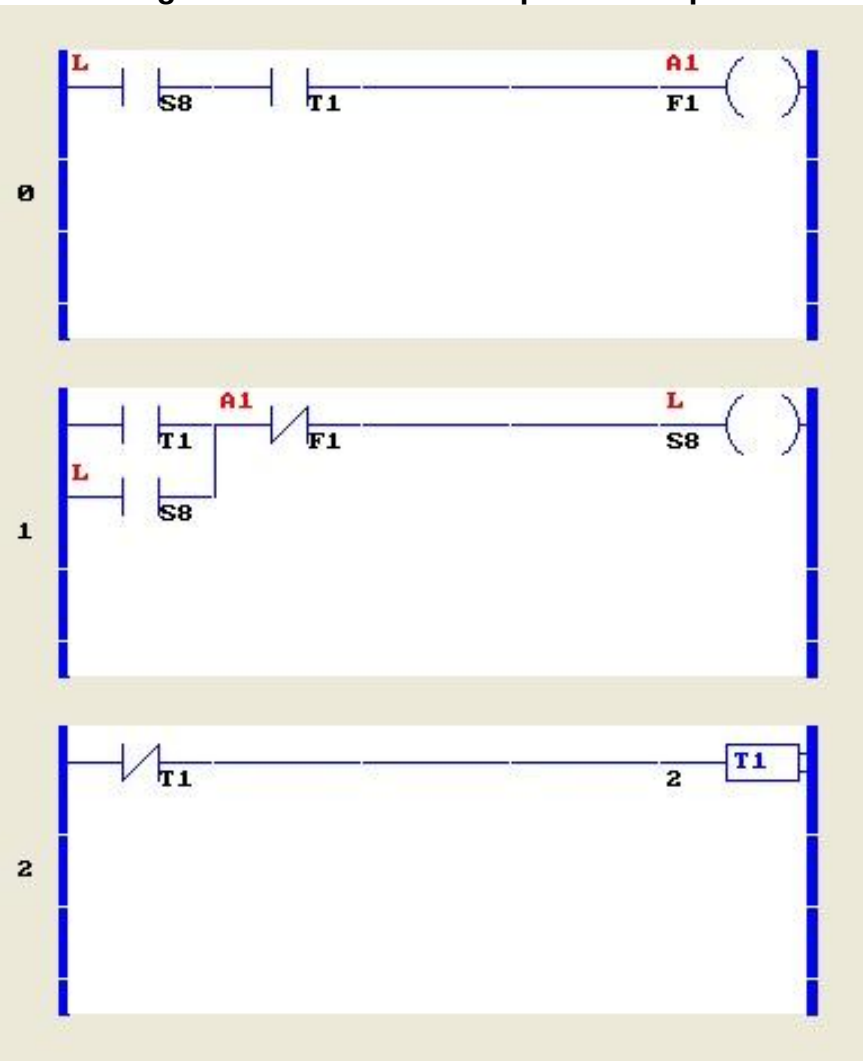
Fonte: Elaborado pelo autor

4.1.4 *Lâmpada Intermitente*

"Utilizando apenas um elemento temporizador, elabore um programa PLC capaz de acionar uma lâmpada de sinalização piscante com período de 2 segundos (SILVEIRA; SANTOS, 2009, p.116)".

A Figura 50 apresenta o LD para o problema proposto e o Apêndice I apresenta o código objeto C/C++ resultante.

FIGURA 50 - Programa fonte em Ladder para a Lâmpada Intermitente



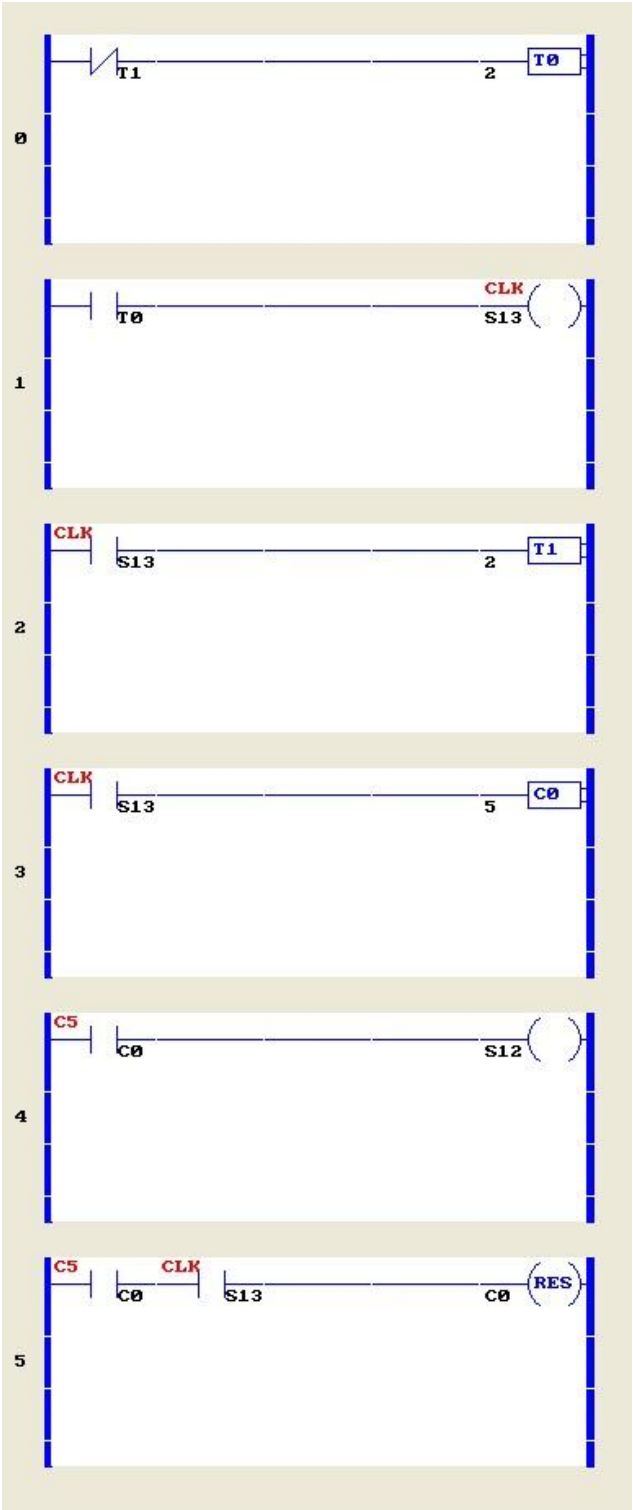
Fonte: Elaborado pelo autor

4.1.5 *Contador de pulsos Módulo Cinco*

> Utilizando-se dos recursos de contagem de CLP, elabore um programa capaz de acionar uma lâmpada sinalizadora sem que o números de pulsos recebidos em sua entrada for igual for múltiplo de 5 (cinco). Assim, no desfecho do quinto pulso a lâmpada acende, desligando-se no recebimento do sexto; novamente acende no décimo e desliga no décimo primeiro (SILVEIRA; SANTOS, 2009, p.116).

A Figura 51 apresenta o Ladder para o problema proposto e no Apêndice J apresenta o Código objeto C/C++ fornecido.

FIGURA 51 - Programa fonte em Ladder para o Contador de Pulsos Módulo Cinco



Fonte: Elaborado pelo autor

4.2 Resultados

Para todos os testes descritos, foram realizadas simulações através do Ambiente de Simulação, onde se obteve êxito frente aos problemas de automatização. Posteriormente, os códigos objetos C/C++ fornecidos pelo CSLadder Mic foram compilados sem erros e os seus respectivos códigos hexadecimais foram gravados no microcontrolador ATmega328 através do compilador Arduino *Alpha*. Os aplicativos do microcontrolador foram testados através da Plataforma de testes e para todas as soluções propostas foram alcançados resultados eficientes.

Durante o teste da seção 4.1.4 Lâmpada Intermitente, que consiste em implementar um programa em Ladder utilizando somente um temporizador, tendo em vista que, a temporização no ambiente de simulação é cronometrada através de uma *thread*, foi observado que quando a contagem atinge ao fim, atribui o bit DN como verdadeiro e em seguida este alterna para falso. Porém, as demais expressões da simulação não percebiam esta alternância, não sendo possível piscar o Led virtual. Portanto, para solucionar este problema foi adicionado um atraso de alternância, para que todas as expressões capturem tal mudança.

Outra condição adversa é que o microcontrolador Atmega328 não possui um controle de transição de estado em um sensor digital, na ocorrência da chamada histerese que é quando uma saída apresenta o sinal diferente da entrada fornecida (CAMPILHO, 2000). Isto atrapalha as aplicações que utilizam entradas digitais para incrementar o valor do contador, contabilizando-o de forma errônea. Para solucionar esta adversidade, foi permitido ao usuário definir um tempo de atraso de leitura em milissegundos, para que o sinal seja lido após sua estabilização.

Contudo, esta solução não se aplica a todos os tipos de sensores, pois este tempo de alternância pode variar em uma mesma entrada digital. No entanto, é possível utilizar modificações em *hardware* que solucione este problema de forma eficiente.

5 CONCLUSÃO

A ferramenta criada de caráter multidisciplinar é de baixo custo para um laboratório, tendo em vista que o CSLadder Mic pode ser disponibilizado gratuitamente para o aprendizado. O depurador didático serve como auxílio na aprendizagem da programação em Ladder para estudantes, fornecendo uma visualização previa da programação fim e instrui para que os alunos consigam elaborar programas mais complexos em CLPs industriais. O Editor Gráfico Ladder, possibilitou uma criação de programa fonte em Ladder de forma simples e sujeito a alterações futuras, modelando as ações do microcontrolador como CLP otimizado, possibilitando com a aprendizagem a criação de programa mais complexos nos CLPs. O Analisador Sintático fornece aos usuários mensagens de erros para implementações em Ladder incorretas. A criação da plataforma de testes permitiu analisar as respostas da simulação em *software* bem como o funcionamento do aplicativo no microcontrolador, ambas obtiveram resultados coerentes, integrando assim *software* e *hardware*.

Tendo em vista, de acordo com o site Netmarketshare (2012) cerca de 92% dos computadores, utilizam o Sistema Operacional *Windows* e o CSLadder Mic é compatível com todas as versões do mesmo, por outro lado, o programa foi desenvolvido em linguagem C/C++ e pode ser compilado em outras plataformas (ex. Linux). Sendo assim, o aplicativo pode ser utilizado na maioria dos computadores pessoais existentes.

Enfim, o projeto contribui para aplicar os conteúdos expostos em sala de aula nos desempenhos de um engenheiro de computação para solução criativa dos problemas de custos, ferramenta de auxílio á aprendizagem técnica ou acadêmica e percepção da área de atuação da Engenharia de Computação frente aos problemas da sociedade.

5.1 Sugestão de trabalhos futuros

Tendo em vista que a ferramenta construída neste trabalho é inicial, isto ocasiona a possibilidade de surgimento de vários estudos futuros para aperfeiçoar e expandir o CSLadder Mic. A seguir serão citadas algumas sugestões de temas futuros:

- Autorizar a inserção de mais de uma instrução de saída nas matrizes lógicas em Ladder;
- Possibilitar a inserção de novas instruções de saída existentes no LD e utilização dos conversores analógico-digital do microcontrolador nas implementações Ladder;
- Permitir aos usuários a criação de uma matriz dinâmica nas edições Ladder;
- Admitir ao usuário a escolha de qual pino do microcontrolador será utilizado como entrada ou saída;
- Possibilitar a geração de código C/C++ para outros microcontroladores;
- Integrar ao microcontrolador ATmega328 ao módulo *Bluetooth* para que envie e receba sinais dos parâmetros sem fios nas aplicações;
- Criar um módulo com a finalidade de otimização de código no CSLadder Mic que utilize simplificação através da álgebra Booleana ou outras otimizações propostas.

REFERÊNCIAS

AHO, Alfred. V. et al. **Compiladores princípios, técnicas e ferramentas**. 2. ed. São Paulo: Pearson Addison Wesley, 2008. ISBN 978-85-886390-24-9.

ALDERSON, Gary E.; LIN, P. M. ***Computer generation of symbolic network functions – a new theory and implementation***. IEEE *Transactions on Circuit Theory*, v. 20, 1973.

ATMEL. ***8-bit atmel microcontroller with 4/8/16/32k bytes in-system program-mable ash***. 2011. Disponível em: <http://www.atmel.com/dyn/resources/prod_documents/doc8271.pdf>. Acesso em: 11 Out. 2011.

BRAICK, Patrícia Ramos; MOTA, Myriam Becho. **História das cavernas ao terceiro milênio**. São Paulo: Editora Moderna, 2010. ISBN 85-16-04703-2 e 85-16-04704-0.

BRAUDE, Eric J. **Projeto de software:** da programação à arquitetura: uma abordagem baseada em Java. Porto Alegre: Bookman, 2009. ISBN 8536304936.

CAMPILHO, A. **Instrumentação electrónica:** métodos e técnicas de medição. Porto: FEUP, 2000. ISBN 9789727520428.

CAROLINE, Amanda. **O espaço industrial no Brasil:** indústrias no Brasil. 2011. Disponível em: <http://www.mundovestibular.com.br/articles/5093/1/O-espaco-industrial-no-Brasil Industrias-no-Brasil/Paacutegina1.html>. Acesso em: 15 Ago. 2011.

FONSECA, Erika Guimarães Pereira da e BEPPU, Mathyan Motta. **Apostila Arduino**. 2010. 23 f. Projeto de Pesquisa – Universidade Federal Fluminense, Programa de Educação Tutorial, Niterói.

GOODRICH, M.; TAMASSIA, R. **Estruturas de dados e algoritmos em JAVA**. 4. ed. Porto alegre: BOOKMAN, 2007. ISBN 9788560031504.

GUDWIN, Ricardo. R. **Linguagens de programação**. 1997. 17 f. Projeto de Pesquisa – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, Campinas.

HIDALGO, Antônio R.; OLIVEIRA, Levi P. B. de; OLIVEIRA, Lecio N. **Linguagem ladder para programar microcontroladores em aplicações como controladores industriais programáveis**. In: CONGRESSO BRASILEIRO DE AUTOMÁTICA, 17, 2008, Juiz de Fora. **Anais...** São Cristóvão: Universidade Federal de Sergipe, 2008. p. 1 – 5.

LARMAN, Craig. **Utilizando UML e padrões:** uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. 3. ed. Porto Alegre: Bookman, 2003. ISBN 9788560031528.

MORAES, Cicero C. **Engenharia de Automação Industrial**. 2. ed. Rio De Janeiro: LTCn, 2007. ISBN 85-216-1532-9.

NETMARKETSHARE. ***Desktop operating system market share***. 2012. Disponível em: <http://netmarketshare.com/>. Acesso em: 20 Mai. 2012.

NETO, Rogério Costa Pacheco. **Desenvolvimento de um ambiente para programação de um micro-CLP**. 2009. 62 f. Monografia (Conclusão de curso) – Universidade Federal do Espirito Santo, Vitória.

PAGANO, D. M.; DINIZ, W. F. D. S. ***Hardware* e *software* para a implementação de um controlador lógico programável didático**. 2009. 78 f. Monografia (Conclusão de curso) - Centro Universitário do Sul de Minas, Varginha.

PRICE, Ana Maria A.; TOSCANI, Simão S. **Implementação de Linguagens de Programação:** Compiladores. 3. ed. Porto Alegre: Bookman, 2008. ISBN 978-85-7780-348-4.

RIBEIRO, M. A. **Automação Industrial**. 4. ed. Salvador: Tek Treinamento & Consultoria *Ltda*, 2001.

RICHTER, C. **Curso de Automação Industrial**. [S.l.], 2001.

SEBESTA, Robert W. **Conceitos de Linguagens de programação**. 5. ed. Porto Alegre: Bookman, 2003. ISBN 85-363-0171-6.

SILBERSCHATZ, A. **Sistemas Operacionais com Java**. 7. ed. Rio de Janeiro: Campus, 2008. ISBN 9788535224061.

SILVEIRA, Paulo Rogério da; SANTOS, Winderson E. dos. **Automatização e Controle Discreto**. 3. ed. São Paulo: Érica, 2009. ISBN 9788571945913.

TORRES, G. **A história dos computadores**. Rio de Janeiro, 1996. Disponível em: <http://www.i9ensino.com/disciplinascadeiras/doc_view/67-historia-dos-computadores.html>. Acesso em: 18 Ago. 2011.

VAREJÃO, Flávio Miguel. **Linguagens de programação**. 4. ed. Rio de Janeiro: Editora: Campus, 2004. ISBN 85-352-1317-1.

WESTHUES, Jonathan. **An Idmicro tutorial**. 2009. Disponível em: <http://cq.cx/ladder-tutorial.pl>. Acesso em: 15 Set. 2011.

APÊNDICE A – BIBLIOTECA CSLADDERMIC.H

Este Apêndice apresenta o enunciado das classes da biblioteca *csladder-mic.h*, utilizada para auxilio na geração de código objeto C/C++.

```
1   /*
2   BIBLIOTECA CSLADDERMIC.H
3   OBJETIVO: UTILZADA PARA GERAÇÃO DO CÓDIGO C++
4   AUTOR: DIEGO SILVA CALDEIRA ROCHA*/
5
6   #ifndef csladdermic_h
7   #define csladdermic_h
8   #include "WProgram.h" //BIBLIOTECA PADRÃO ARDUINO
9   #include "Bounce.h" // BIBLIOTECA PARA CONTROLE DE HISTERESE
10
11  //CLASSE Entrada PARA UTILIZAÇÃO DE ENTRADAS DIGITAIS
12  class Entrada{
13
14  private:
15      bool sinal;//NIVEL ALTO OU BAIXO
16      int pino;// VALOR DO PINO
17      int Delay_read;//TEMPO DE HISTERESE
18
19  public:
20
21      Entrada(int Pin, int D_r);//CONSTRUTOR
22      void Setup();//CONFIGURAÇÃO INICIAL DA ENTRADA DIGITAL
23      bool get_state();//RETORNA O SINAL DA ENTRADA
24      int get_pino();//RETORNA O PINO UTILIZADO
25      void Read(); //LEITURA DO SINAL DA ENTRADA
26
27  };
28
29  //CLASSE Saida PARA UTILIZAÇÃO DE SAIDAS DIGITAIS
30  class Saida{
31
32  private:
33      bool sinal;//NIVEL ALTO OU BAIXO
34      int pino;// VALOR DO PINO
35
36  public:
37      Saida(int Pin);//CONSTRUTOR
38      void Setup();//CONFIGURAÇÃO INICIAL DA SAÍDA DIGITAL
39      bool get_state();//RETORNA O SINAL DA SAIDA
40      int get_pino();//RETORNA O PINO UTILIZADO
41      void Read();//LEITURA DO SINAL DA ENTRADA
42      void set();//COLOCA A SAÍDA DIGITAL EM NÍVEL ALTO
43      void reset(); //COLOCA A SAÍDA DIGITAL EM NÍVEL BAIXO
44  };
45
46  //CLASSE Flag PARA UTILIZAÇAO NA LÓGICA INTERNA
47  class Flag{
48  private:
49      bool sinal;//NIVEL ALTO OU BAIXO
50  public:
51      Flag(bool t);//CONSTRUTOR
52      bool get_state();//RETORNA O SINAL DA FLAG
```

```
53          void set(); //COLOCAO A FLAG EM NÍVEL ALTO
54          void reset();//COLOCA A FLAG EM NÍVLE BAIXO
55      };
56
57      //CLASSE Ctu PARA UTILIZAÇAO DE CONTADORES
58      class Ctu{
59      private:
60          bool sinal, //NIVEL ALTO OU BAIXO
61                       sinal_ant;//GUARDA O SINAL ANTERIOR
62          int presset,//ATE´QUAL VALOR SE DESEJA CONTAR
63                      accum;//VALOR ATUAL DA CONTAGEM
64
65      public:
66          Ctu(int PRST);//CONSTRUTOR
67          bool get_state();//RETORNA O SINAL CONTADOR
68          void set();//COLOCAO O CONTADOR EM NÍVEL ALTO
69          void reset();//COLOCA O CONTADOR EM NÌVEL BAIXO
70          void restart();//REINICIALIZA O CONTADOR
71      };
72
73      //CLASSE Ton PARA UTILIZAÇAO DE TEMPORIZADORES
74      class Ton{
75      private:
76          bool sinal;//NIVEL ALTO OU BAIXO
77          int presset;//ATE´QUAL VALOR SE DESEJA TEMPORIZAR
78
79      public:
80          Ton(int PRST);//CONSTRUTOR
81          void set();//COLOCAO O TEMPORIZADOR EM NÍVEL ALTO
82          bool get_state();//RETORNA O SINAL DO TEMPORIZADOR
83          void reset();//REINICIALIZA O TEMPORIZADOR
84
85      };#end
```

APÊNDICE B – BIBLIOTECA CSLADDERMIC.CPP

Este Apêndice apresenta as implementações dos métodos das classes da biblioteca *csladdermic.h*, utilizada para auxilio na geração de código objeto C/C++.

```
1   /*
2   BIBLIOTECA CSLADDERMIC.CPP
3
4   OBJETIVO: IMPLEMENTAÇÃO DAS CLASSES DA CSLADDERMC.HI
5
6   AUTOR: DIEGO SILVA CALDEIRA ROCHA
7
8   */
9
10  #include "WProgram.h"  //BIBLIOTECA PADRÃO DO ARDUINO
11  #include <csladdermic.h>//CABEÇARIO
12  #include "Bounce.h" // BIBLIOTECA PARA CONTROLE DE HISTERESE
13
14  /*IMPLEMENTAÇÃO DOS MÉTODS DA CLASSE Entrada*/
15
16  //CONSTRUTOR PAR. 1 VALOR DO PINO, PAR. 2 VALOR DE ATRASO DE HISTERESE;
17  Entrada :: Entrada(int Pin, int D_r)
18  {
19      pino=Pin;//SET PINO
20      D_r=Delay_read;//ATRASO DE HISTERESE
21      sinal=false; // TODA ENTRADA INICIALMENTE ESTA EM NÍVEL BAIXO
22
23  }
24
25  //INSTALAÇÕES INICIAIS DE ENTRADA
26  void Entrada::Setup()
27  {
28      pinMode(pino, INPUT);//colocando o pino como entrada
29      digitalWrite(pino, HIGH);//colocando o pino para o funcionamento DO reistor pull up ;
30  }
31
32  //RETORNA O ESTADO DA ENTRADA
33  bool Entrada :: get_state()
34  {
35      return sinal;
36  }
37
38  //RETORNA O PINO DE ENTRADA
39  int Entrada:: get_pino()
40  {
41      return pino;
42  }
43
44  //FAZ A LEITURA DA ENTRADA DIGITAL
45  void Entrada::Read()
46  {
47
48      //CONTROLE DE HISTERESE
49      Bounce bouncer =  Bounce( pino,Delay_read );
50      bouncer.update ( );
51      if(bouncer.read()==0)//pull-up inverte a logica
```

```
52                  sinal=true;
53          else
54                  sinal=false;
55  }
56
57  /* IMPLEMENTAÇÃO DOS MÉTODoS DA CLASSE Saida*/
58
59  //CONSTRUTOR PAR. 1 VALOR DO PINO
60  Saida :: Saida(int Pin)
61  {
62          pino=Pin;
63          sinal=false;
64          digitalWrite(pino, LOW);
65  }
66
67  //INSTALAÇÕES INICIAIS DE ENTRADA
68  void Saida ::Setup()
69  {
70          pinMode(pino, OUTPUT);
71  }
72
73  //RETORNA O ESTADO DA SAÍDA
74  bool Saida:: get_state()
75  {
76          return sinal;
77  }
78
79  //RETORNA O PINO DE SAÍDA
80  int Saida:: get_pino()
81  {
82          return pino;
83  }
84
85  //COLOCA UMA SAÍDA DIGITAL EM NIVEL ALTO
86  void Saida::set()
87  {
88          digitalWrite(pino, HIGH);
89          sinal=true;
90  }
91
92  //COLOCA UMA SAÍDA DIGITAL EM NÍVEL BAIXO
93  void Saida::reset()
94  {
95          digitalWrite(pino, LOW);
96          sinal=false;
97  }
98
99  /* IMPLEMENTAÇÃO DOS MÉTODoS DA CLASSE Flag*/
100
101 //CONSTRUTOR PAR. 1 ESTADO INICIAL DA FLAG
102 Flag :: Flag(bool t )
103 {
104         sinal=t;
105
106 }
107
108 //RETORNA O ESTADO DA FLAG
109 bool Flag :: get_state()
110 {
111         return sinal;
```

```
112  }
113
114  //COLOCA A FLAG EM NÍVEL ALTO
115  void Flag :: set()
116  {
117       sinal=true;
118  }
119
120  //COLOCA A FLAG EM NÍVEL BAIXO
121  void Flag :: reset()
122  {
123       sinal=false;
124  }
125
126  /* IMPLEMENTAÇÃO DOS MÉTODoS DA CLASSE Ctu */
127
128  //CONSTRUTOR PAR 1 VALOR FIM DA CONTAGEM
129  Ctu :: Ctu(int PRST)
130  {
131       presset=PRST;
132       sinal=false;
133       sinal_ant=false;
134       accum=0;
135  }
136
137  //RETORNA O ESTADO DO CONTADOR
138  bool Ctu:: get_state()
139  {
140       return sinal;
141  }
142
143  //COLOCA O ESTADO DO CONTADOR EM NÍVEL ALTO
144  void Ctu::set()
145  {
146       sinal_ant=true;
147  }
148
149  //COLOCA O ESTADO DO CONTADOR EM NÍVEL BAIXO
150  void Ctu::reset()
151  {
152       if(sinal_ant==true&&accum<presset)
153       {
154            accum++;
155            sinal_ant=false;
156       }
157       if(accum==presset)
158            sinal=true;
159
160  }
161
162  //REINICIA A CONTAGEM DO CONTADOR
163  void Ctu::restart()
164  {
165       accum =0;
166       sinal=false;
167       sinal_ant=false;
168  }
169  /*IMPLEMENTAÇÃO DOS MÉTODoS DA CLASSE Ton*/
170  Ton :: Ton(int PRST)
171  {
```

```
172         presset=PRST;
173         sinal=false;
174
175     }
176
177     //RETORNA O ESTADO DO SINAL DO TEMPORIZADOR
178     bool Ton:: get_state()
179     {
180         return sinal;
181     }
182
183     //COLOCA O TEMPORIZADOR EM NÍVEL ALTO
184     void Ton::set()
185     {
186         sinal=true;
187     }
188
189     //COLOCAR O TEMPORIZADOR EM NÍVEL BAIXO
190     void Ton::reset()
191     {
192         sinal=false;
193     }
```

APÊNDICE C – EXEMPLIFICAÇÃO DO PASSO 1 DA GERAÇÃO CÓDIGO C/C++

Este Apêndice apresenta uma exemplificação de um trecho do código objeto C/C++, com o primeiro passo da geração do código objeto.

```
1   /*
2   OBJETIVO: PASSO 1 DA GERAÇÃO DO CÓDIGO C/C++
3   AUTOR: DIEGO SILVA CALDEIRA ROCHA
4   */
5
6   #include <csladdermic.h>
7   #include <pt.h> //UTILIZADA PARA THREAD
8   #define hist 8 //TEMPO DEFINIDO DE HISTERES DEFINIDO PELO USUÁRIO
9
10  //______SAÍDA DIGITAIS
11  Saida S13(13);
12
13  //____TEMPORIZADOR
14  Ton T0(5);
15  Ton T1(5);
16
17  //____AUXILIO PARA TEMP
18  bool f_t0=false;
19  bool f_t1=false;
20
21  //_____THREAD
22  static struct pt pt0;
23
24  //_____FUNÇAO THREAD__
25  //parâmetro 1: thread
26  //parâmetro 2: contagem inicializada
27  //parâmetro 3: temporização finalizada com êxito
28
29  static int temp_thread0 (struct pt *pt, bool conte, bool & fim_tempo )
30  {
31      static unsigned long timestamp = 0;
32      PT_BEGIN(pt);
33      if(!fim_tempo)
34      {
35          PT_WAIT_WHILE(pt, millis() - timestamp < 5000 && conte);
36          timestamp = millis();
37          if(conte)
38              fim_tempo=true;
39      }
40      PT_END(pt);
41  }
42
43  (...)
```

APÊNDICE D – EXEMPLIFICAÇÃO DO PASSO 2 DA GERAÇÃO CÓDIGO C/C++

Este Apêndice apresenta uma exemplificação de um trecho do código objeto C/C++, com o segundo passo da geração do código objeto.

```
1   /*
2   OBJETIVO: PASSO 2 DA GERAÇÃO DO CÓDIGO C/C++
3   AUTOR: DIEGO SILVA CALDEIRA ROCHA
4   */
5
6   (...)
7
8         void setup()//INSTALAÇÕES INICIAIS
9   {
10
11        EN2.Setup();
12        S13.Setup();
13        PT_INIT(&pt0);
14
15  }//FIM DAS INSTALAÇÕES
16
17  (...)
```

APÊNDICE E – EXEMPLIFICAÇÃO DO PASSO 3 DA GERAÇÃO CÓDIGO C/C++

Este Apêndice apresenta uma exemplificação de um trecho do código objeto C/C++, com o terceiro passo da geração do código objeto.

```
1   /*
2   OBJETIVO: PASSO 3 DA GERAÇÃO DO CÓDIGO C/C++
3   AUTOR: DIEGO SILVA CALDEIRA ROCHA
4   */
5
6   (...)
7
8   void loop()//FLUXO PRINCIPAL
9   {
10      //___LEITURA DAS ENTRADAS DIGITAIS
11      E0.Read();
12      E2.Read();
13      E3.Read();
14      E4.Read();
15      E7.Read();
16
17      //EXPRESSÕES E COMANDOS
18      if((E0.get_state()&&F1.get_state())      ||(E2.get_state()&&!E3.get_state())
19  ||(E4.get_state()&&!E3.get_state())||(S8.get_state()&&!E3.get_state()))
20      {
21              S8.set();
22      }
23      else
24      {
25              S8.reset();
26      }
27      (..)
28  }//FIM FLUXO PRINICIPAL
```

APÊNDICE F – CÓDIGO OBJETO C/C++ PARA O SISTEMA DE RESERVATÓRIO

Este Apêndice apresenta o código objeto C/C++ para o teste da seção 4.1.1.

```
/*
AUTOR: DIEGO SILVA CALDEIRA ROCHA
OBJETIVO: TESTE - SISTEMA DE RESERVATÓRIO
*/

#include <Bounce.h>
#include <csladdermic.h>
#include <pt.h>
#define hist 8

//_______ENTRADAS DIGITAIS
Entrada EN0(0,hist);
Entrada EN1(1,hist);
Entrada EN2(2,hist);
Entrada EN3(3,hist);

//______SAÍDAS DIGITAIS
Saida S8(8);
Saida S9(9);
Saida S10(10);
Saida S11(11);

void setup()//CONFIGURAÇÕES INICIAIS
{
      EN0.Setup();
      EN1.Setup();
      EN2.Setup();
      EN3.Setup();
      S8.Setup();
      S9.Setup();
      S10.Setup();
      S11.Setup();
}//FIM CONFIGURAÇÕES INICIAIS

void loop()//FLUXO PRINCIPAL DO PROGRAMA
{
      //__LEITURA DAS ENTRADA DIGITAIS
      EN0.Read();
      EN1.Read();
      EN2.Read();
      EN3.Read();

      //EXPRESSÕES E COMANDOS
      if((EN0.get_state()))
      {
            S8.reset();}
      if((!EN1.get_state()))
      {
            S8.set();
      }if((EN1.get_state()))
      {
            S9.set();
      }
      else
```

```
55          {
56                  S9.reset();
57          }
58          if((!EN2.get_state()))
59          {
60                  S10.set();
61          }
62          else
63          {
64                  S10.reset();
65          }
66          if((EN3.get_state()))
67          {
68                  S11.set();
69          }
70          else
71          {
72                  S11.reset();
73          }
74
75  }//FIM DO FLUXO PRINCIPAL
```

APÊNDICE G – CÓDIGO OBJETO C/C++ PARA RELÉS COM DEPENDÊNCIA

Este Apêndice apresenta o código objeto C/C++ para o teste da seção 4.1.2.

```
/*
AUTOR: DIEGO SILVA CALDEIRA ROCHA
OBJETIVO: TESTE - RELÉS COM DEPENDÊNCIA
*/

#include <Bounce.h>
#include <csladdermic.h>
#include <pt.h>
#define hist 8

//_____ENTRADAS DIGITAIS
Entrada EN0(0,hist);
Entrada EN1(1,hist);
Entrada EN2(2,hist);
Entrada EN3(3,hist);

//_____SAÍDAS DIGITAIS
Saida S11(11);
Saida S12(12);


void setup()//INSTALAÇÕES INICIAIS
{

	EN0.Setup();
	EN1.Setup();
	EN2.Setup();
	EN3.Setup();
	S11.Setup();
	S12.Setup();
}//FIM DAS INSTALAÇÕES INICIAIS

void loop()//FLUXO PRINCIPAL DO PROGRMA
{
	//__LEITURA DA ENTRADAS DIGITAIS
	EN0.Read();
	EN1.Read();
	EN2.Read();
	EN3.Read();

	//___EXPRESSÕES E COMANDOS
	if((EN0.get_state()&&!EN1.get_state())||(S11.get_state()&&!EN1.get_state()))
	{
		S11.set();
	}
	else
	{
		S11.reset();
	}
	if((S11.get_state()&&EN2.get_state()&&!EN3.get_state())
		||
		(S12.get_state()&&!EN3.get_state()))
	{
		S12.set();
```

```
55          }
56          else
57          {
58                  S12.reset();
59          }
60  }//FIM DO FLUXO PRINCIPAL DO PROGRAMA
```

APÊNDICE H – CÓDIGO OBJETO C/C++ PARA A MÁQUINA DE SOLDA

Este Apêndice apresenta o código objeto C/C++ para o teste da seção 4.1.3.

```
/*
AUTOR: DIEGO SILVA CALDEIRA ROCHA
OBJETIVO: TESTE - MÁQUINA DE SOLDA
*/

#include <Bounce.h>
#include <csladdermic.h>
#include <pt.h>
#define hist 8

//_______ENTRADAS DIGITAIS
Entrada EN5(5,hist);

//_______SAÍDAS DIGITAIS
Saida S8(8);
Saida S10(10);

//________TEMPORIZADORES
Ton T1(1);
Ton T2(1);

//_______BOOL AUXILIAR
bool f_t1=false;
bool f_t2=false;

//_______THREAD
static struct pt pt1;
static struct pt pt2;


//_____FUNÇAO THREAD__
//parâmetro 1: thread
//parâmetro 2: contagem inicializada
//parâmetro 3: temporização finalizada com êxito

static int temp_thread1(struct pt *pt, bool conte, bool & fim_tempo )
{
	static unsigned long timestamp = 0;
	PT_BEGIN(pt);
	if(!fim_tempo)
	{
		PT_WAIT_WHILE(pt, millis() - timestamp < 1000 && conte);
		timestamp = millis();
		if(conte)
			fim_tempo=true;
	}
	PT_END(pt);
}//FIM FUNÇÃO THREAD


//_____FUNÇAO THREAD__
//parâmetro 1: thread
```

```
54  //parâmetro 2: contagem inicializada
55  //parâmetro 3: temporização finalizada com êxito.
56  static int temp_thread2(struct pt *pt, bool conte, bool & fim_tempo )
57  {
58      static unsigned long timestamp = 0;
59      PT_BEGIN(pt);
60      if(!fim_tempo)
61      {
62          PT_WAIT_WHILE(pt, millis() - timestamp < 1000 && conte);
63          timestamp = millis();
64          if(conte)
65              fim_tempo=true;
66      }
67      PT_END(pt);
68  }//FIM FUNÇÃO THREAD
69
70  void setup()//INSTALAÇÕES INICIAS
71  {
72      EN5.Setup();
73      S8.Setup();
74      S10.Setup();
75      PT_INIT(&pt1);
76      PT_INIT(&pt2);
77  }//FIM DAS INSTALAÇÕES INICIAS
78
79  void loop()//FLUXO PRINICPAL
80  {
81      //_LEITURA DA ENTRADA DIGITAL
82      EN5.Read();
83
84      //___EXPRESSÕES E COMANDOS
85      if((EN5.get_state()&&!T2.get_state())||(S8.get_state()&&!T2.get_state()))
86      {
87          S8.set();
88      }
89      else
90      {
91          S8.reset();
92      }
93      if((S8.get_state()))
94      {
95          if(!f_t1)
96              temp_thread1(&pt1,true,f_t1);//INICIA A CONTEMPORIZAÇÃO
97          else
98              T1.set();
99      }
100     else
101     {
102         f_t1=false;
103         temp_thread1(&pt1,false,f_t1 );//INTERROMPER A CONTEMPORIZAÇÃO
104         T1.reset();
105     }
106     if((T1.get_state()&&EN5.get_state()))
107     {
108         S10.set();
109     }
110     else
111     {
112         S10.reset();
113     }
```

```
114         if((!EN5.get_state()))
115         {
116                 if(!f_t2)
117                         temp_thread2(&pt2,true,f_t2);//INICIA A CONTEMPORIZAÇÃO
118                 else
119                         T2.set(); }
120         else
121         {
122                 f_t2=false;
123                 temp_thread2(&pt2,false,f_t2 );//INTERROMPE A CONTEMPORIZAÇÃO
124                 T2.reset();
125         }
126
127     }//FIM DO FLUXO PRINCIPAL
```

APÊNDICE I – CÓDIGO OBJETO C/C++ PARA A LÂMPADA INTERMITENTE

Este Apêndice apresenta o código objeto C/C++ para o teste da seção 4.1.4.

```
/*
AUTOR: DIEGO SILVA CALDEIRA ROCHA
OBJETIVO: TESTE - LÂMPADA INTERMITENTE
*/

#include <Bounce.h>
#include <csladdermic.h>
#include <pt.h>
#define hist 8

//______SAÍDAS DIGITAIS
Saida S8(8);

//_____FLAGS
Flag F1(false);

//____TEMPORIZADORES
Ton T1(2);

//_______BOOL AUXILIAR
bool f_t1=false;

//_______THREAD
static struct pt pt1;

//_____FUNÇAO THREAD__
//_____FUNÇÕES THREAD´S
//parâmetro 1: thread
//parâmetro 2: contagem inicializada
//parâmetro 3: temporização finalizada com exito
static int temp_thread1(struct pt *pt, bool conte, bool & fim_tempo )
{
	static unsigned long timestamp = 0;
	PT_BEGIN(pt);

	if(!fim_tempo)
	{
		PT_WAIT_WHILE(pt, millis() - timestamp < 2000 && conte);
		timestamp = millis();
		if(conte)
			fim_tempo=true;
	}
	PT_END(pt);
}

void setup()//INSTALAÇÕES INICIAS
{
	S8.Setup();
	PT_INIT(&pt1);
}//FIM INSTALAÇÕES INICIAIS

void loop()//FLUXO PRINCIPAL DO PROGRAMA
{
```

```
54          //EXPRESSÕES E COMANDOS
55          if((S8.get_state()&&T1.get_state()))
56          {
57                      F1.set();
58          }
59          else
60          {
61                      F1.reset();
62          }
63          if((T1.get_state()&&!F1.get_state())||(S8.get_state()&&!F1.get_state()))
64          {
65                      S8.set();
66          }
67          else
68          {
69                      S8.reset();
70          }
71          if((!T1.get_state()))
72          {
73                      if(!f_t1)
74                                  temp_thread1(&pt1,true,f_t1);//INÍCIO DE CONTEMPORIZAÇÃO
75                      else
76                                  T1.set();
77          }
78          else
79          {
80                      f_t1=false;
81                      temp_thread1(&pt1,false,f_t1 );//INTERROMPE A CONTEMPORIZAÇÃO
82                      T1.reset();
83          }
84
85      } //FIM DO FLUXO PRINCIPAL DO PROGRAMA
86
```

APÊNDICE J – CÓDIGO OBJETO C/C++ PARA O CONTADOR DE PULSOS MÓDULO CINCO

Este Apêndice apresenta o código objeto C/C++ para o teste da seção 4.1.5.

```
1   /*
2   AUTOR: DIEGO SILVA CALDEIRA ROCHA
3   OBJETIVO: TESTE - CONTADOR DE PULSOS MÓDULO CINCO
4   */
5
6   #include <Bounce.h>
7   #include <csladdermic.h>
8   #include <pt.h>//UTILIZASA PARA THREAD
9   #define hist 8 //CONTROLE DE HISTERESE
10
11  //______SAÍDAS DIGITAIS
12  Saida S12(12);
13  Saida S13(13);
14
15  //_______CONTADOR
16  Ctu C0(5);
17
18  //_______TEMPORIZADOR
19  Ton T0(2);
20  Ton T1(2);
21
22  //_______BOOL AUXILIAR
23  bool f_t0=false;
24  bool f_t1=false;
25
26  //VARIÁVEIS THREAD´S
27  static struct pt pt0;
28  static struct pt pt1;
29
30
31  //_____FUNÇÕES THREAD´S
32  //parametro 1: thread
33  //parametro 2: contagem inicializada
34  //parametro 3: temporização finalizada com exito
35  static int temp_thread0(struct pt *pt, bool conte, bool & fim_tempo )
36  {
37
38      static unsigned long timestamp = 0;
39      PT_BEGIN(pt);
40
41      if(!fim_tempo)
42      {
43          PT_WAIT_WHILE(pt, millis() - timestamp < 2000 && conte);
44          timestamp = millis();
45          if(conte)
46              fim_tempo=true;
47      }
48
49      PT_END(pt);
50  }
51
52
```

```
53  //_____FUNÇÕES THREAD´S
54  //parametro 1: thread
55  //parametro 2: contagem inicializada
56  //parametro 3: temporização finalizada com exito
57
58  static int temp_thread1(struct pt *pt, bool conte, bool & fim_tempo )
59  {
60
61      static unsigned long timestamp = 0;
62      PT_BEGIN(pt);
63
64      if(!fim_tempo)
65      {
66          PT_WAIT_WHILE(pt, millis() - timestamp < 2000 && conte);
67          timestamp = millis();
68          if(conte)
69              fim_tempo=true;
70      }
71
72      PT_END(pt);
73
74  }
75
76  void setup()//INSTALAÇÕES INICIAIS
77  {
78
79      S12.Setup();
80      S13.Setup();
81
82      PT_INIT(&pt0);
83      PT_INIT(&pt1);
84
85  }
86
87  void loop()//FLUXO PRINCIPAL
88  {
89      if((!T1.get_state()))
90      {
91          if(!f_t0)
92              temp_thread0(&pt0,true,f_t0);//INICIO DE CONTEMPORIZAÇÃO
93          else
94              T0.set();
95      }
96      else
97      {
98          f_t0=false;
99          temp_thread0(&pt0,false,f_t0 );//INTERROMPER CONTEMPORIZAÇÃO
100         T0.reset();
101     }
102
103     if((T0.get_state()))
104     {
105         S13.set();
106     }
107     else
108     {
109         S13.reset();
110     }
111
112     if((S13.get_state()))
```

```
113         {
114                 if(!f_t1)
115                         temp_thread1(&pt1,true,f_t1);//INICIO DE CONTEMPORIZAÇÃO
116                 else
117                         T1.set();
118         }
119         else
120         {
121                 f_t1=false;
122                 temp_thread1(&pt1,false,f_t1 );//INTERROMPER CONTEMPORIZAÇÃO
123                 T1.reset();
124         }
125
126         if((S13.get_state()))
127         {
128                 C0.set();
129         }
130         else
131         {
132                 C0.reset();
133         }
134
135         if((C0.get_state()))
136         {
137                 S12.set();
138         }
139         else
140         {
141                 S12.reset();
142         }
143
144         if((C0.get_state()&&S13.get_state()))
145         {
146                 C0.restart();//REINICIO DE CONTADOR
147         }
148
149 }//FIM FLUXO PRINCIPA
```