

**UNIVERSIDADE REGIONAL DE BLUMENAU**  
**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**  
**CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO**

**PROTÓTIPO DE CLP PARA LINUX EMBARCADO**

**LEONARDO FERNANDES**

**BLUMENAU**  
**2014**

**2014/2-12**

**LEONARDO FERNANDES**

## **PROTÓTIPO DE CLP PARA LINUX EMBARCADO**

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Miguel Alexandre Wisintainer, Mestre - Orientador

**BLUMENAU  
2014**

**2014/2-12**

# **PROTÓTIPO DE CLP PARA LINUX EMBARCADO**

Por

**LEONARDO FERNANDES**

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente: 

---

Prof. Miguel Alexandre Wisintainer, Mestre – Orientador, FURB

Membro: 

---

Profa. Gabriele Jennrich Bambineti, Especialista – FURB

Membro: 

---

Prof. Francisco Adell Péricas, Mestre – FURB

Blumenau, 8 de dezembro de 2014

Dedico este trabalho a minha família, pelo amor, apoio e compreensão.

## **AGRADECIMENTOS**

A Deus pela oportunidade de viver este momento com as pessoas que amo.

A minha esposa Ana Paula Krause Fernandes, pelo amor, apoio e compreensão e por estar ao meu lado durante a realização deste trabalho.

A minha filha recém-nascida Isabela Fernandes, por tonar este trabalho um desafio ainda maior, proporcionando uma motivação infinitamente maior para sua conclusão.

A minha família, especialmente aos meus pais por sempre me incentivar durante toda a vida.

Ao amigo Gustavo Rufino Feltrin, por ter contribuído não só nesta etapa final do curso, mas também por ter feito parte da minha formação.

A empresa Circuitec, por flexibilizar meus horários de trabalho nesta etapa final do curso e pelo apoio a realização do mesmo.

Ao meu orientador, pela empolgação e auxílio na realização deste trabalho e por estar sempre à disposição para esclarecer minhas dúvidas.

Ao professor Everaldo Artur Grahl, pelo auxílio com o formulário de avaliação.

A professora Joyce Martins, por ter dedicado parte do seu tempo para esclarecimentos sobre compiladores.

Nunca ande por trilhas, pois assim só irá até onde outros já foram.

Alexander Graham Bell

## RESUMO

Este trabalho apresenta a implementação de uma ferramenta que permite um hardware com Linux Embarcado assumir funcionalidades de um CLP. A ferramenta é composta por um editor gráfico da linguagem de diagramas Ladder, que permite criar programas com as principais instruções de entrada, saída, comparação, lógica, aritmética, além das instruções do tipo temporizador e contador. Também permite traduzir o programa fonte Ladder para código em linguagem C a partir de uma representação intermediária da lógica criada. Essa representação intermediária é baseada na técnica denominada código de três endereços, através de uso de quádruplas. Para desenvolver o editor gráfico foi utilizada a linguagem de programação Java juntamente com a API gráfica Java2D. Como parte do protótipo foi desenvolvido uma placa de testes, um hardware composto por sensores e atuadores conectados a uma placa mãe composta por um módulo com Linux Embarcado, permitindo que o código C gerado seja compilado e executado no hardware. Este trabalho também apresenta os resultados obtidos através do teste de desempenho e uma avaliação da ferramenta realizada pela turma de Automação do décimo semestre do curso de Engenharia Elétrica da FURB.

Palavras-chave: CLP. Linux embarcado. Ladder. Java2D. Gerador de código. Tradutor de código. Código intermediário. Quádruplas.

## **ABSTRACT**

This work presents the implementation of a tool which allows hardware with Embedded Linux assume a PLC functionality. The tool consists of a graphical editor of the ladder diagrams language that allows you to create programs with the main instructions of input, output, comparison, logical, arithmetic, addition to the instructions of the timer and counter type. Also allows you to translate the source Ladder program to C code from an intermediate representation of the logic created. This intermediate representation is based on the technique called three-address code, by using quadruples. To develop graphic editor was used the programming language Java along with the Java2D graphics API. As part of the prototype was developed a testing board, compound hardware by sensors and actuators connected to a motherboard comprising a module with Embedded Linux, allowing the generated C code can be compiled and executed on the hardware. This work also presents the results of performance testing and evaluation of the tool held by the class of Automation of the tenth semester of Electrical Engineering of FURB.

Key-words: PLC. Embedded Linux. Ladder. Java2D. Code generator. Translator code. Intermediate code. Quadruple.



## LISTA DE FIGURAS

Figura 1 – Programa Ladder .....	18
Figura 2 – Fluxograma básico do sistema de operação de um CLP .....	19
Figura 3 – Módulo Aria G25 .....	21
Figura 4 – Exemplo de código intermediário .....	23
Figura 5 – Exemplo de quádrupla .....	24
Figura 6 – Editor Ladder do projeto PL $\mu$ X .....	25
Figura 7 – Interface gráfica do software LDmicro .....	26
Figura 8 – Módulo CUBLOC CB220.....	27
Figura 9 – Software CUBLOC Studio.....	28
Figura 10 – Casos de uso.....	30
Figura 11 – Diagrama de pacotes de instruções Ladder.....	32
Figura 12 – Diagrama de classes do dispositivo .....	33
Figura 13 – Diagrama de pacotes do gerador e tradutor de código.....	34
Figura 14 – Pacote <code>prototype.compiler</code> .....	35
Figura 15 – Pacote <code>prototype.compiler.frontend</code> .....	36
Figura 16 – Pacote <code>prototype.compiler.backend</code> .....	37
Figura 17 – Diagrama de sequência .....	38
Figura 18 – Blocos de instruções Ladder .....	39
Figura 19 – Instruções Ladder em paralelo .....	39
Figura 20 – Fluxograma de execução da aplicação embarcada.....	44
Figura 21 – Placa mãe .....	45
Figura 22 – Placa de testes .....	46
Figura 23 – Novo regulador de tensão.....	56
Figura 24 – Placa mãe montada.....	56
Figura 25 – Hardwares conectados.....	58
Figura 26 – Tela inicial da ferramenta de edição .....	58
Figura 27 – Criação de um novo projeto .....	59
Figura 28 – Aba Instrução .....	59
Figura 29 – Instruções adicionadas .....	59
Figura 30 – Aba Dispositivo.....	60
Figura 31 – Memórias associadas.....	60

Figura 32 – Definindo conexão com o hardware .....	61
Figura 33 – Tela de configuração de conexão .....	61
Figura 34 – Salvar projeto .....	61
Figura 35 – Gerar e traduzir o programa .....	62
Figura 36 – Mensagens de sucesso na compilação.....	62
Figura 37 – Conectar ao hardware.....	62
Figura 38 – Mensagens de sucesso de conexão.....	62
Figura 39 – Executar programa Ladder .....	63
Figura 40 – Mensagens referentes à execução do programa Ladder.....	63
Figura 41 – Teste do interruptor .....	63
Figura 42 – Cenário A. <i>Scan time</i> em relação ao acesso a interfaces E/S.....	65
Figura 43 – Cenário B. <i>Scan time</i> em relação às instruções.....	66
Figura 44 – Cenário B. <i>Scan time</i> em relação às instruções com maior performance .....	66
Figura 45 – Resultado do comando <code>pmap</code> .....	67
Figura 46 – Memória do processo no Cenário A .....	67
Figura 47 – Memória do processo no Cenário B.....	68
Figura 48 – Programa Ladder referente ao caso de uso proposto à turma de Automação .....	69
Figura 49 – Diagrama de classes de instrução reduzido.....	76
Figura 50 – Processos ativos no hardware listados pelo comando <code>ps -aux</code> .....	84
Figura 51 – Resultado do comando <code>free -m</code> .....	84
Figura 52 – Formulário de avaliação do protótipo .....	85

## LISTA DE QUADROS

Quadro 1 – Principais características do hardware.....	22
Quadro 2 – Requisitos Funcionais.....	29
Quadro 3 – Requisitos Não Funcionais .....	29
Quadro 4 – Caso de uso UC01 .....	30
Quadro 5 – Caso de uso UC02 .....	31
Quadro 6 – Caso de uso UC03 .....	31
Quadro 7 – Identificação das variáveis associativas .....	39
Quadro 8 – Identificação das variáveis de instrução .....	40
Quadro 9 – Compatibilidade entre instruções e variáveis associativas.....	40
Quadro 10 – Instruções do código intermediário .....	42
Quadro 11 – Implementação do método <code>add</code> da classe <code>Rung</code> .....	47
Quadro 12 – Implementação da classe <code>SemanticAnalyzer</code> .....	49
Quadro 13 – Exemplo de código intermediário representado em texto. ....	50
Quadro 14 – Código intermediário respectivo a instruções Ladder em paralelo .....	51
Quadro 15 – Implementação do método <code>genInstruction</code> da classe <code>IRGenerator</code> ....	52
Quadro 16 – Implementação do método <code>generate</code> da classe <code>IRGenerator</code> .....	53
Quadro 17 – Exemplo de programa Ladder vs. código intermediário vs. código C .....	54
Quadro 18 – Código gerado e traduzido da instrução <code>Count Up</code> .....	55
Quadro 19 – Ações do algoritmo de tradução .....	55
Quadro 20 – Visões da placa de testes montada.....	57
Quadro 21 – Aba <code>Memória</code> .....	60
Quadro 22 – Exemplo de código referente às instruções <code>And</code> e <code>Add</code> .....	67
Quadro 23 – Resultado da avaliação da ferramenta .....	69
Quadro 24 – Comparativo entre o trabalho proposto e os correlatos .....	71
Quadro 25 – Símbolos dos elementos estruturais.....	77
Quadro 26 – Símbolos das instruções Ladder .....	77
Quadro 27 – Arquivo <i>Makefile</i> gerado .....	82
Quadro 28 – Arquivo <i>shell script</i> Linux gerado .....	82
Quadro 29 – Pontos fortes observados .....	86
Quadro 30 – Pontos fracos observados .....	86

## LISTA DE SIGLAS

A/D – Analógico / Digital

API – *Application Programming Interface*

CLP – Controlador Lógico Programável

CPU – *Central Processing Unit*

E/S – Entrada / Saída

FURB – Fundação Universidade Regional de Blumenau

GCC – *GNU Compiler Collection*

GNU – *GNU is Not Unix*

GPIO – *General Purpose Input / Output*

IEC - *International Electrotechnical Commission*

LED – *Light Emitting Diode*

PLC – *Programmable Logic Controller*

RF – Requisito Funcional

RNF – Requisito Não Funcional

SSH – *Secure Shell*

UART – *Universal Asynchronous Receiver Transmitter*

UC – *Use Case*

VDC – *Volts Direct Current*

# SUMÁRIO

<b>1 INTRODUÇÃO.....</b>	<b>14</b>
1.1 OBJETIVOS.....	14
1.2 ESTRUTURA.....	14
<b>2 FUNDAMENTAÇÃO TEÓRICA .....</b>	<b>16</b>
2.1 AUTOMAÇÃO .....	16
2.2 CLP.....	17
2.3 LINGUAGEM LADDER.....	17
2.4 SISTEMA EMBARCADO .....	19
2.5 LINUX EMBARCADO .....	20
2.6 MÓDULO ARIA G25 .....	21
2.7 COMPILADOR.....	22
2.8 JAVA2D.....	24
2.9 TRABALHOS CORRELATOS.....	25
2.9.1 PL $\mu$ X.....	25
2.9.2 LDmicro .....	26
2.9.3 CUBLOC .....	27
<b>3 DESENVOLVIMENTO DO PROTÓTIPO.....</b>	<b>29</b>
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO.....	29
3.2 ESPECIFICAÇÃO .....	29
3.2.1 Especificação do software.....	29
3.2.1.1 Casos de uso .....	29
3.2.1.1.1 Criar programa Ladder.....	30
3.2.1.1.2 Gerar código intermediário .....	31
3.2.1.1.3 Traduzir código intermediário em linguagem C .....	31
3.2.1.2 Diagrama de classes.....	31
3.2.1.2.1 Diagrama de pacotes das instruções Ladder .....	32
3.2.1.2.2 Diagrama de classes do dispositivo .....	33
3.2.1.2.3 Diagrama de classes do gerador e tradutor de código.....	34
3.2.1.2.4 Pacote <code>prototype.compiler</code> .....	34
3.2.1.2.5 Pacote <code>prototype.compiler.frontend</code> .....	35
3.2.1.2.6 Pacote <code>prototype.compiler.backend</code> .....	36

3.2.1.3 Diagrama de sequência .....	37
3.2.1.4 Lexemas, sintaxe e semântica.....	38
3.2.1.5 Código intermediário .....	42
3.2.2 Especificação do hardware.....	43
3.2.2.1 Fluxograma .....	43
3.2.2.2 Diagrama esquemático.....	44
3.3 IMPLEMENTAÇÃO .....	46
3.3.1 Técnicas e ferramentas utilizadas.....	46
3.3.1.1 Gerador e tradutor de código .....	47
3.3.1.1.1 Restrições sintáticas .....	47
3.3.1.1.2 Análise semântica .....	48
3.3.1.1.3 Gerador de código intermediário .....	50
3.3.1.1.4 Tradutor de código .....	54
3.3.1.2 Montagem do hardware .....	56
3.3.2 Operacionalidade da implementação .....	57
3.4 RESULTADOS E DISCUSSÃO .....	64
3.4.1 Testes de desempenho do programa fonte Ladder no hardware .....	64
3.4.2 Teste de funcionalidade.....	68
3.4.3 Comparativo com os correlatos.....	70
<b>4 CONCLUSÕES.....</b>	<b>72</b>
4.1 EXTENSÕES .....	73
<b>REFERÊNCIAS .....</b>	<b>74</b>
<b>APÊNDICE A – Diagrama de classes de instruções reduzido .....</b>	<b>76</b>
<b>APÊNDICE B – Símbolos da linguagem Ladder. ....</b>	<b>77</b>
<b>APÊNDICE C – <i>Makefile</i> e <i>shell script</i> Linux.....</b>	<b>82</b>
<b>APÊNDICE D – Estado inicial do hardware .....</b>	<b>84</b>
<b>APÊNDICE E – Formulário de avaliação do protótipo.....</b>	<b>85</b>
<b>APÊNDICE F – Respostas descritivas referentes ao formulário de avaliação .....</b>	<b>86</b>

## 1 INTRODUÇÃO

Na década de 60, devido ao aumento da competitividade, produtividade e qualidade no setor automotivo, fazia-se necessário encontrar uma alternativa para sistemas de controle que até então eram baseados em relé. Este antigo sistema de controle tornava-se inviável quanto ao custo, quando a complexidade da lógica aumentava. Assim, em 1968, a Divisão Hydramatic da GM determinou os critérios para o projeto de um Controlador Lógico Programável (CLP), sendo que o primeiro dispositivo a atender às especificações foi desenvolvido pela Gould Modicom em 1969 (GEORGINI, 2000, p. 51).

Sabe-se atualmente que o CLP ainda é muito utilizado no meio industrial e residencial, porém, embora uma das principais ideias fosse a de reduzir custos, devido à evolução e a alta complexidade de alguns processos, o avanço da tecnologia e dos recursos do CLP, fez com que o custo desses equipamentos se tornasse muito alto para o uso em tarefas simples que não possuam alto valor agregado (ELEUTÉRIO; HOVADICH; BRAGA, 2011).

Diante do exposto, foi disponibilizado um protótipo de CLP baseado em um hardware com Linux Embarcado, através do desenvolvimento de uma ferramenta de programação para o protótipo. A ferramenta permite gerar código em linguagem C para o Linux a partir da linguagem Ladder, uma tradicional linguagem de programação de CLP.

### 1.1 OBJETIVOS

O objetivo deste trabalho é disponibilizar uma ferramenta que permita um hardware com Linux Embarcado assumir funcionalidades de um CLP.

Os objetivos específicos do trabalho são:

- a) disponibilizar um editor gráfico da linguagem Ladder com as instruções básicas de entrada, saída, temporização, contagem, comparação, lógica e aritmética;
- b) gerar um código intermediário a partir do programa fonte Ladder;
- c) permitir traduzir o código intermediário em código na linguagem C.

### 1.2 ESTRUTURA

O trabalho está estruturado em quatro capítulos. O segundo capítulo refere-se à fundamentação teórica necessária para o desenvolvimento do protótipo.

No terceiro capítulo são apresentados os requisitos do protótipo, especificação de forma textual e através de diagramas, implementação de acordo com as principais técnicas e ferramentas utilizadas, resultados e discussões obtidos com o protótipo.

Por fim, o capítulo quatro apresenta as conclusões e sugestões de extensão para trabalhos futuros.



## 2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais conceitos necessários ao desenvolvimento do protótipo. A seção 2.1 apresenta os conceitos relacionados à automação. A seção 2.2 apresenta os conceitos sobre CLP. A seção 2.3 apresenta os conceitos sobre a linguagem Ladder; A seção 2.4 apresenta conceitos sobre sistemas embarcados. A seção 2.5 apresenta os conceitos sobre Linux Embarcado. A seção 2.6 apresenta o hardware de referência Aria G25. A seção 2.7 apresenta os conceitos sobre compiladores. A seção 2.8 apresenta os conceitos sobre a *Application Programming Interface* (API) gráfica Java2D. Por último a seção 2.9 apresenta os trabalhos correlatos.

### 2.1 AUTOMAÇÃO

“A **Automação** é um conceito e um conjunto de técnicas por meio das quais se constroem sistemas ativos capazes de atuar com eficiência ótima pelo uso de informações recebidas do meio sobre o qual atuam.” (MARTINS, 2012, p. 6, grifo do autor). O autor afirma que a automação possui um objetivo básico, que é facilitar os processos produtivos permitindo produzir bens com menor custo, maior quantidade, menor tempo e maior qualidade.

Segundo Bishop (2008a, p. 17-1), sensores e atuadores são dois componentes críticos de todo sistema de controle onde a saída é dependente da entrada (malha fechada). Tipicamente estes sistemas são constituídos por uma unidade de sensoriamento, um controlador e uma unidade de atuação.

Sensor é um dispositivo que quando exposto a um fenômeno físico (temperatura, deslocamento, força, etc.) produz um sinal de saída proporcional (elétrico, mecânico, magnético, etc.). Existem muitos tipos de sensores e são classificados conforme seu objetivo. Basicamente sensores podem ser classificados como passivos ou ativos e analógicos ou digitais. Nos sensores passivos, a energia necessária para produzir a saída é proveniente do fenômeno físico por si mesmo, enquanto sensores ativos requerem uma fonte de energia externa. Sensores analógicos produzem continuamente um sinal que é proporcional ao parâmetro sentido e tipicamente requerem um conversor analógico para digital antes do controlador digital, enquanto sensores digitais produzem um sinal de saída diretamente ao controlador digital (BISHOP, 2008a, p. 17-2).

Atuadores são basicamente o músculo por trás de um sistema mecatrônico que aceita um comando de controle (geralmente elétrico) e produz uma mudança no sistema físico pela geração de força, calor, fluxo e assim por diante. Normalmente os atuadores são utilizados em

conjunto com o fornecimento de energia e um mecanismo de acoplamento, que atua como interface entre o atuador e o sistema físico. Podem ser classificados com base no tipo de energia. Eles são essencialmente elétricos, eletromecânicos, eletromagnéticos, pneumáticos ou do tipo hidráulicos (BISHOP, 2008a, p. 17-11).

Os Controladores controlam os dispositivos automatizados (sensores e atuadores). Monitora as informações dos sensores, podendo enviar comandos para que um atuador ative ou desative algum equipamento. De maneira geral podem possuir interfaces independentes, na forma de um controle remoto, ou serem sofisticadas centrais de automação (ALMEIDA, 2009 apud ACCARDI, DODONOV, 2012).

## 2.2 CLP

Segundo Natale (2000, p. 11), o CLP “É um computador com as mesmas características conhecidas do computador pessoal, porém, em uma aplicação dedicada na Automação de processos em geral [...]”.

O Controlador Lógico Programável, ou simplesmente PLC (Programmable Logic Controller), pode ser definido como um dispositivo de estado sólido – um Computador Industrial, capaz de armazenar instruções para implementação de funções de controle (sequência lógica, temporização e contagem, por exemplo), além de realizar operações lógicas e aritméticas, manipulação de dados e comunicação em rede, sendo utilizado no controle de Sistemas Automatizados. (GEORGINI, 2000, p. 48).

Georgini (2000, p.48) afirma que os principais blocos que compõe um CLP são:

- a) *Central Processing Unit* (CPU): corresponde ao processador, conjunto de memórias e os circuitos auxiliares de controle;
- b) circuito/módulos de Entrada/Saída (E/S): podem ser discretos (sinais digitais) ou analógicos. Correspondem aos sensores e atuadores;
- c) fonte de alimentação: responsável pela tensão de alimentação fornecida à CPU e circuitos/módulos de E/S;
- d) base ou *rack*: proporciona conexão mecânica e elétrica entre CPU, circuitos/módulos de E/S e a fonte de alimentação. Contém o barramento de comunicação e tensão de alimentação entre eles.

O CLP executa um programa que é desenvolvido por meio de uma ferramenta manual ou um software, através de uma lógica muito similar a programação convencional (GEORGINI, 2000, p. 50).

## 2.3 LINGUAGEM LADDER

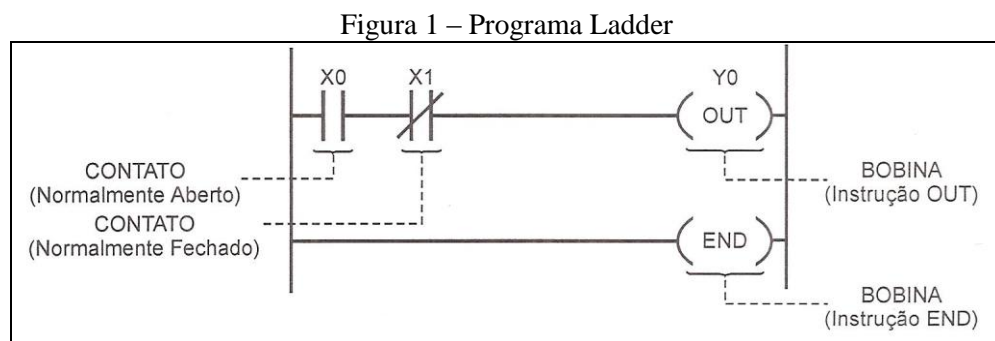
A norma IEC 61131-3 criada em 1993 foi o primeiro esforço real para padronizar linguagens de programação para automação industrial (PLCOPEN, 2014a). A norma aborda

as linguagens de programação, define a estrutura de um projeto, os tipos de dados e a organização interna do programa (GEORGINI, 2000, p. 85).

Segundo a PLCOPEN (2014b), a norma especifica a sintaxe e a semântica de cinco linguagens de programação, duas textuais, duas gráficas e um diagrama funcional sequencial, que respectivamente são, Lista de Instruções (IL), Texto Estruturado (ST), Diagrama Ladder (LD), Diagrama de Blocos de Função (FBD) e Diagrama Funcional Sequencial (SFC).

Ladder foi a primeira linguagem criada para programação de CLP. O fato de ser uma linguagem gráfica, baseada em símbolos semelhantes aos encontrados nos esquemas elétricos, foi determinante para aceitação do CLP por técnicos e engenheiros acostumados com os sistemas de controle baseado em relé. Provavelmente, é ainda a mais utilizada (GEORGINI, 2000, p. 48).

O nome Ladder deve-se à representação da linguagem se parecer com uma escada (ladder), na qual duas barras verticais paralelas são interligadas pela lógica de controle denominada *rung*, formando os degraus da escada. (GEORGINI, 2000, p. 48). A Figura 1 demonstra um exemplo de um programa em linguagem Ladder. O programa é composto por três instruções no primeiro *rung*, associadas às variáveis X0, X1 e Y0, e pela instrução END no segundo *rung*.

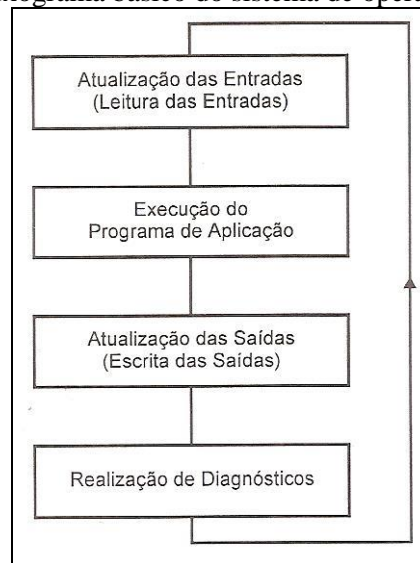


Fonte: Georgini (2000, p. 84).

Cada elemento da lógica de controle representa uma instrução da linguagem Ladder, porém um mesmo símbolo gráfico pode representar instruções diferentes dependendo da localização na lógica de controle (GEORGINI, 2000, p. 101).

“A CPU inicia a execução [...] a partir do primeiro rung [...], executando-o da esquerda para a direita e de cima para baixo, rung a rung, até encontrar a instrução **END** (FIM) [...]” (GEORGINI, 2000, p. 95, grifo do autor). A execução do programa faz parte do ciclo (*scan*) de operação do CLP, conforme Figura 2. O autor afirma que é possível perceber a seguinte característica: quanto maior for o programa, maior será o tempo de execução.

Figura 2 – Fluxograma básico do sistema de operação de um CLP



Fonte: Georgini (2000, p. 93).

“O CLP varre (executa) um programa de lógica Ladder várias vezes por segundo. Geralmente o tempo de execução varia de 5 a 100 milissegundos. Tempos de execução mais rápidos são necessários para processos que operam em uma velocidade maior.” (BISHOP, 2008b, p. 25-9, tradução nossa). Erickson (2010) afirma que este tempo de varredura é denominado *scan time*.

## 2.4 SISTEMA EMBARCADO

Um sistema embarcado é semelhante a um sistema de uso geral. Ambos possuem capacidades de controlar E/S, executar lógicas de controle, possuir teclado, mouse, conexão de rede *Ethernet* e interface gráfica com o usuário. Porém um sistema embarcado é desenvolvido para uma tarefa específica. Esta funcionalidade específica define o único propósito do *design* embarcado. (HOLLABAUGH, 2002, p. 9).

No passado, era muito mais fácil de distinguir um sistema embarcado de um sistema de propósito geral do que é hoje. [...] o aumento no desempenho do hardware e o baixo custo tornou turva a linha entre computadores de uso geral e embarcados. Os avanços tecnológicos tem tornado difícil dizer o que é embarcado (HOLLABAUGH, 2002, p. 9, tradução nossa).

Lombardo (2002, p. xvi, tradução nossa) afirma que “o mais conclusivo método para determinar se um sistema de computador se caracteriza como “propósito geral” ou “embarcado” requer um exame da missão do sistema em vida”. O autor esclarece que sistemas embarcados tentam ser uma solução de custo-benefício para um problema específico ou conjunto de especificações e possuem uma missão limitada. Já a missão de um computador de propósito geral é mais como a de um canivete suíço. Um canivete suíço pode cortar, serrar,

parafusar e depilar. Semelhantemente, o computador de propósito geral pode ser usado para processar folhas de pagamento, jogos, navegar na internet e mais.

## 2.5 LINUX EMBARCADO

O Linux é um sistema operacional *open-source* projetado em 1991 por Linus Torvalds, um estudante em fase de graduação, que motivado pelo desejo de aprender sobre a CPU da Intel o 80386, o levou a implementar um *kernel* completo, semelhante ao UNIX e compatível com *Portable Operating System Interface* (POSIX) (MAXWELL, 2000, p. 4).

Na verdade, o Linux não é o sistema operacional em sua totalidade. Quando instala o que comumente é denominado Linux, você está instalando uma enorme quantidade de ferramentas que funcionam em conjunto como um sistema verdadeiramente funcional. O Linux, por si só, é o *kernel* deste sistema operacional, seu coração, sua mente, seu sistema nervoso (MAXWELL, 2000, p. 1).

Torvalds (1997, tradução nossa) afirma que, “a versão inicial do Linux era extremamente não portátil”, segundo Maxwell (2000, p. 33) já no ano 2000 o *kernel* oficial havia sido portado para sistemas baseados em CPUs Alpha, ARM, Motorola 680x0, MIPS, PowerPC, SPARC e SPARC-64.

Linux Embarcado é geralmente definido como uma distribuição do Linux destinada a equipamentos embarcados (ENGINEERSGARAGE, 2014). Segundo Riese (2014, p. 1), Linux Embarcado possui propriedades que lhe dão vantagens sobre os sistemas operacionais embarcados tradicionais, como, qualidade, confiabilidade, alta coesão e baixo acoplamento do código Linux, fácil manutenção provida pelas funcionalidades divididas em módulos em diferentes arquivos. Outra grande vantagem do projeto Linux é a capacidade de decidir quais componentes a serem instalados. Isto dá ao Linux a capacidade de ser personalizado para uma determinada aplicação.

Entre os diversos recursos e ferramentas que o Linux suporta, é possível destacar:

- a) *shell script*: interpretador de linguagem de comando. Permite reunir e executar comandos em um arquivo, usando as construções de programação, como por exemplo, loops e declaração de *case*, para realizar rapidamente operações complexas que seriam difíceis de redigir muitas vezes pelos usuários (NEGUS, 2008, p. 4);
- b) *gpiolib*: camada de abstração de acesso ao *General Purpose Input Output* (GPIO), ou seja, pinos do processador. Ao invés de utilizar drives de GPIO específicos o acesso e configuração ocorrem por meio de arquivos de sistema, contidos a partir de `/sys/class/gpio` (KERNEL, 2014);

- c) *GNU Collection Compiler (GCC)*: coleção de compiladores que incluem *front ends* para C, C++, Objective C, Fortran, Java, Ada, entre outras, além de bibliotecas para estas linguagens (GNU, 2014);
- d) *make*: ferramenta para controlar o processo de construção e reconstrução de um software. O arquivo *make (makefile)* contém as regras que dizem o que construir e como construir (NEGUS, 2008, p. 679);
- e) *Secure Shell (SSH)*: protocolo de rede que permite que dados sejam trocados através de um canal seguro entre dois computadores. É normalmente usado para fazer *login* em uma máquina remota e executar comandos. Transferências de arquivos podem ser feitos através de protocolos associados (ARCHLINUX, 2014).

## 2.6 MÓDULO ARIA G25

É um módulo de baixo custo desenvolvido pela ACMESYSTEMS (2014) que suporta Linux Embarcado (Figura 3), com o intuito de reduzir drasticamente o tempo de desenvolvimento e projeto de dispositivos embarcados de baixa potência. Foi concebido para executar o boot de uma distribuição de Linux Embarcado compilado para ARM9, a partir de um cartão microSD.

Figura 3 – Módulo Aria G25



Fonte: Adaptado de Acmesystems (2014).

O Quadro 1 demonstra as principais características do hardware.

Quadro 1 – Principais características do hardware

CPU	ATMEL ARM9 AT91SAM9G25, 32 bits, 400 Mhz
Memória	128MB DDR2
Interfaces	1 Ethernet (10/100 Mbit) 3 USB sendo duas <i>host</i> e <i>device</i> 2.0 e outra <i>host</i> 1.1 6 serial UART 2 serial I2C 2 serial SPI 60 E/S 4 conversores Analógico/Digital (A/D)
Boot	A partir de um microSD externo, SD Card ou memória serial <i>flash</i> .
Alimentação	+3.3 Volts Direct Current (VDC), aproximadamente 210mA (690 <i>miliwatt</i> )
Temperatura suportada	0 a 70°C
Dimensão	40 x 40 mm
Peso	5g

## 2.7 COMPILADOR

Todo o software executando em computadores foi escrito em alguma linguagem de programação, mas antes que possa ser executado, um programa primeiro precisa ser traduzido para um formato que lhe permita ser executado por um computador. Os sistemas de software que fazem essa tradução são denominados compiladores (AHO, 2008, p. 1). Segundo o autor um compilador é composto por sete fases sendo elas, analisador léxico, analisador sintático, analisador semântico, gerador de código intermediário, otimizador de código dependente de máquina, gerador de código e otimizador de código independente de máquina.

A primeira fase de um compilador é chamada de análise léxica ou leitura (*scanning*). O analisador léxico lê uma sequência de caracteres que compõem o programa fonte e os agrupa em sequências significativas, denominadas lexemas. Para cada lexema, o analisador produz como saída, um *token* que será usado no próximo passo durante a análise sintática. (AHO, 2008, p. 3).

A segunda fase do compilador é a análise sintática. O analisador sintático recebe do analisador léxico uma cadeia de *tokens* representando o programa fonte e verifica se essa cadeia de *tokens* pertence à linguagem gerada pela gramática. O analisador deve ser projetado para emitir quaisquer erros de sintaxe encontrados no programa fonte. De forma conceitual o analisador sintático constrói uma árvore de derivação que representa o programa fonte e a passa para a próxima etapa do compilador. Na prática não é necessário construir uma árvore de derivação, o analisador sintático e o restante dos analisadores podem ser implementados em um único módulo (AHO, 2008, p. 122).

A terceira fase do compilador é a análise semântica. O analisador semântico utiliza as informações provenientes das fases anteriores para verificar a consistência semântica do

programa fonte com a definição da linguagem. Uma parte importante da análise é a verificação de tipo, em que o compilador verifica se cada operador possui operandos compatíveis (AHO, 2008, p. 6).

O gerador de código intermediário é a quarta fase do compilador. Segundo Louden (2004, p. 402), “O código intermediário [representação intermediária semelhante ao código-alvo] pode assumir muitas formas – existem quase tantos estilos de código intermediário quanto compiladores.” É particularmente útil quando o objetivo do compilador é produzir código extremamente eficiente, pois isso requer uma quantidade significativa de análise das propriedades do código-alvo, o que é facilitado, pelo código intermediário (LOUDEN, 2004, p. 402).

O código intermediário pode também ser útil para facilitar a geração de código para outros alvos. Se o código intermediário for relativamente independente da máquina alvo, gerar código para uma máquina-alvo diferente exigirá apenas que o tradutor do código intermediário para o código-alvo seja reescrito, o que em geral é mais fácil do que reescrever todo o gerador de código (LOUDEN, 2004, p. 402).

Aho (2008, p. 6) considera uma forma de representação intermediária chamada de código de três endereços, que consiste em uma sequência de instruções com três operandos por instrução. No código de três endereços existe no máximo um operador do lado direito de uma expressão. Admite-se o uso de nomes temporários gerados pelo compilador para resolver expressões com mais de um operador. A Figura 4 ilustra um código intermediário respectivo à expressão  $a = b + c * d$ . É possível verificar o uso de uma variável temporária `_t1` gerada pelo compilador.

Figura 4 – Exemplo de código intermediário

```
_t1 := c * d
a := b + _t1
```

Fonte: Ricarte (2003).

O código é construído a partir de dois conceitos, endereços e instruções. Endereços podem ser um nome, uma constante ou um temporário gerado pelo compilador. As instruções podem ser atribuição, cópia, desvios incondicionais, desvios condicionais e chamadas de rotinas (AHO, 2008, p. 232).

A descrição de instruções de três endereços especifica os componentes de cada tipo de instrução, mas não especifica a representação dessas instruções em uma estrutura de dados. Em um compilador, essas instruções podem ser implementadas como objetos ou como registros em uma estrutura de dados (AHO, 2008, p. 234).



Uma forma de representação do código intermediário é denominada Quádrupla. Uma quádrupla possui quatro campos sendo eles, operador, primeiro argumento, segundo argumento e resultado, conforme pode ser visto na Figura 5, que ilustra a representação em quádrupla do código intermediário da Figura 4 (AHO, 2008, p. 234).

Figura 5 – Exemplo de quádrupla

	operador	arg 1	arg 2	resultado
1	*	c	d	_t1
2	+	b	_t1	a

Fonte: Ricarte (2003).

Gerador de código é a sexta fase do processo de compilação que recebe como entrada uma Representação Intermediária (RI) a partir do programa fonte, produzida pelas fases de análise denominadas *front-end* e produz como saída um código objeto equivalente à entrada. Segundo Aho (2008, p. 321), “Os requisitos impostos sobre o gerador de código são severos. O código objeto precisa preservar o significado semântico do programa fonte e ser de alta qualidade [...]”. Compiladores que precisam produzir códigos objetos eficientes incluem uma fase de otimização antes da geração do código. As fases de otimização e geração de código são denominadas *back-end* (AHO, 2008, p. 321).

As fases cinco e sete referentes à otimização de código, fazem algumas transformações com o objetivo de produzir um código melhor. Normalmente, melhor significa mais rápido, mas outros objetivos podem ser desejados, como um código menor ou um código que consuma menos energia (AHO, 2008, p. 6).

## 2.8 JAVA2D

A API Java2D provê gráficos bidimensionais para programas Java através de extensões do *Abstract Windowing Toolkit* (AWT). Esse pacote abrangente suporta desenho de linhas, textos e imagens em uma estrutura flexível e cheia de recursos para o desenvolvimento de interfaces de usuário, programas de desenho sofisticados e editores de imagem (ORACLE, 2014). Segundo a ORACLE (2014) a API Java2D fornece os seguintes recursos:

- a) modelo uniforme de desenho para *displays* e impressoras;
- b) primitivas geométricas, tais como curvas, retângulos, elipses, bem como um mecanismo para desenhar praticamente qualquer forma geométrica;
- c) mecanismos para detecção de colisão em formas de texto e imagens;
- d) modelo de composição que permite controle sobre como os objetos sobrepostos são desenhados;

- e) suporte de cores aprimorado que facilita o gerenciamento de cores;
- f) suporte para impressão de documentos complexos;
- g) controle de qualidade do desenho através do uso de dicas de desenho.

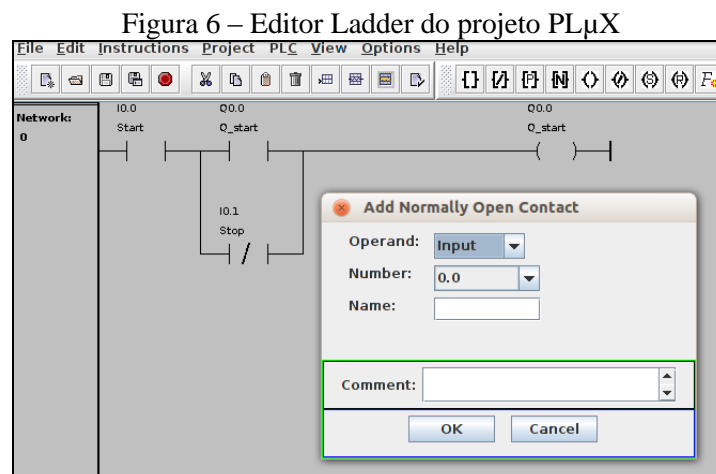
## 2.9 TRABALHOS CORRELATOS

Existem algumas soluções que tem por objetivo, criar um CLP baseado em um microcontrolador que permita a programação em linguagem Ladder através de ferramentas específicas. Dentre estas foram selecionadas o projeto PL $\mu$ X criada por Coninck (2012), o compilador *open source* LDmicro criado por Westhues (2007) e o produto comercial CUBLOC desenvolvido pela Comfile Technology (2014a).

### 2.9.1 PL $\mu$ X

PL $\mu$ X é um esforço para produzir um CLP baseado em Linux Embarcado criado por Coninck (2012). Utiliza a plataforma GNUBLIN<sup>1</sup>, um projeto originalmente voltado ao ensino de Linux Embarcado, mas agora com foco em desenvolvimento (GNUBLIN, 2014).

O projeto consiste em um editor de diagramas Ladder feito em Java conforme ilustrado na Figura 6 e um software para uma plataforma de hardware específica, que interpreta e executa o código objeto gerado pelo editor. O editor permite criar um programa Ladder, simular, copiar e executar o programa objeto através do protocolo SSH, porém se faz necessário primeiro instalar no hardware um servidor SSH, criar os diretórios apropriados e copiar o software que executa o código objeto. É um processo manual que exige conhecimento básico em Linux.



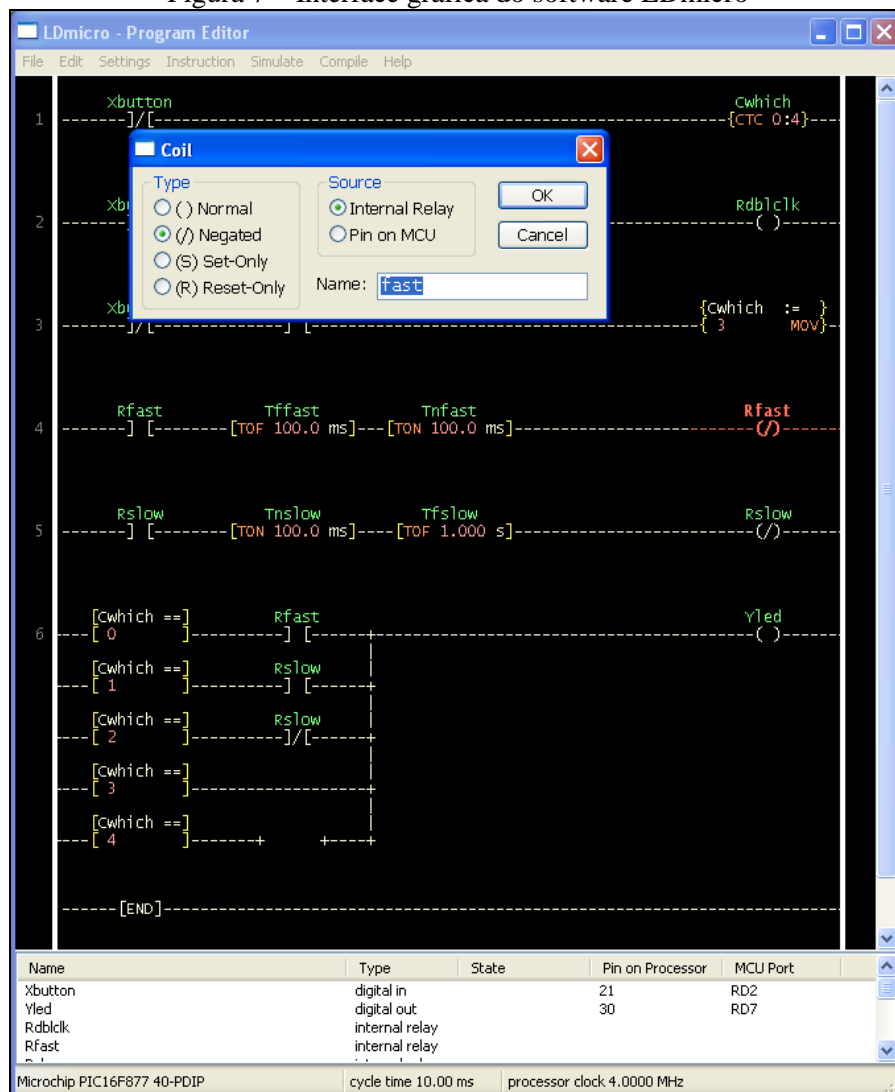
Fonte: Coninck (2012).

<sup>1</sup> “Como parte do projeto GNUBLIN um grande kit com muitas placas de expansão, aplicações e uma API para desenvolvimento de software de grandes sistemas de controle, *data loggers* e automação foi criado.” (GNUBLIN, 2014, tradução nossa).

## 2.9.2 LDmicro

O software LDmicro é um compilador e simulador em tempo real criado por Westhues (2007), que permite gerar código nativo para microcontroladores PIC16 e AVR a partir de diagramas Ladder. O software suporta além das principais instruções da linguagem Ladder conversor A/D, unidade *Pulse Width Modulation* (PWM) e serial *Universal Asynchronous Receiver Transmitter* (UART). Possui uma interface gráfica que representa o diagrama em caracteres, conforme pode ser vista na Figura 7. Segundo Westhues (2007) “meu código gerador para os AVRs é muito pobre [...] por exemplo, ele não tira vantagem do fato do AVR ter mais de um registrador. Muitos dos códigos gerados ficam pouco otimizados.” É específico para o sistema operacional Windows e é dependente de um gravador para os microcontroladores suportados pela ferramenta, compatível com o código gerado no formato *Intel Hexadecimal Object File* (HEX).

Figura 7 – Interface gráfica do software LDmicro

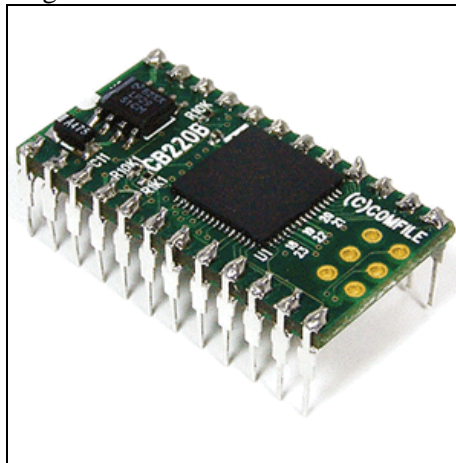


Fonte: Westhues (2007).

### 2.9.3 CUBLOC

O produto comercial CUBLOC desenvolvido pela Comfile Technology (2014a) possui a proposta de ser um CLP “On-Chip”, ou seja, um CLP em um único chip conforme Figura 8, permitindo aos desenvolvedores projetar Placas de Circuito Impresso (PCI) personalizados com apenas um microcontrolador.

Figura 8 – Módulo CUBLOC CB220



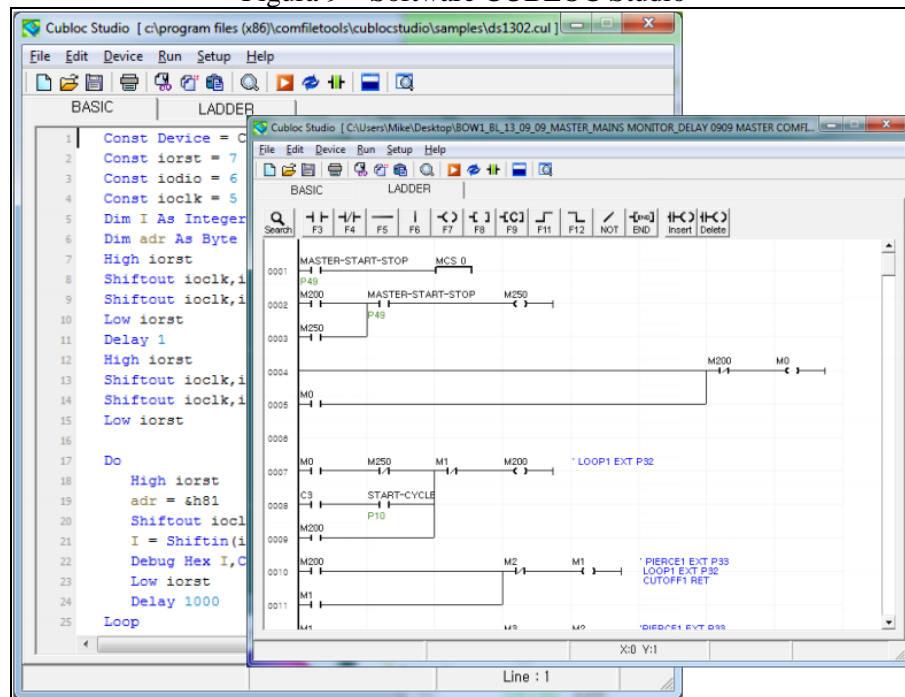
Fonte: Comfile Technology (2014b).

A proposta do CUBLOC é suprir as limitações da lógica Ladder com a linguagem Basic. A ferramenta CUBLOC Studio ilustrada na Figura 9 permite a programação do CLP através das duas linguagens. CUBLOC possui uma estrutura multitarefa e de tempo real, que permite executar Basic e Ladder simultaneamente. Também é possível programá-lo em apenas uma das duas linguagens. Uma das limitações superadas pelo Basic propostas é a de permitir a interação com diversas interfaces como display, teclado e computador. Uma característica que vale a pena ressaltar é de que todos os circuitos da lógica Ladder (*rungs*) são processados em paralelo, ou seja, todos processados ao mesmo tempo.

A estrutura interna do CUBLOC contém memória de programa e trabalho entre o interpretador Basic e o processador Ladder. Porém a memória de trabalho do processador Ladder pode ser acessada pelo interpretador Basic.

O software CUBLOC Studio é suportado apenas pelo sistema operacional Windows e efetua a programação do CLP por meio de comunicação serial.

Figura 9 – Software CUBLOC Studio



Fonte: Comfile Technology (2014c, p. 23).

### 3 DESENVOLVIMENTO DO PROTÓTIPO

Nas próximas seções são exibidos os requisitos do protótipo, a especificação do software e hardware, implementação, operacionalidade da implementação, os resultados e discussão.

#### 3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Nesta seção são apresentados os Requisitos Funcionais (RF) conforme o Quadro 2 e os Requisitos Não Funcionais (RNF) conforme o Quadro 3.

Quadro 2 – Requisitos Funcionais

REQUISITOS FUNCIONAIS
RF01: A ferramenta deverá permitir a criação de um programa em linguagem Ladder com as instruções básicas de entrada, saída, temporização, contagem, comparação, lógica e aritmética.
RF02: A ferramenta deverá gerar código intermediário a partir do programa fonte Ladder.
RF03: A ferramenta deverá permitir traduzir o código intermediário para código em linguagem C.

Quadro 3 – Requisitos Não Funcionais

REQUISITOS NÃO FUNCIONAIS
RNF01: A ferramenta deverá ser implementada utilizando a linguagem de programação Java.
RNF02: A ferramenta deverá ser implementada utilizando o ambiente de desenvolvimento Eclipse.
RNF03: A ferramenta deverá ser implementada utilizando a API Java2D no editor gráfico.
RNF04: A ferramenta deverá ser baseada na norma IEC 61131-3.
RNF05: O hardware deverá ser um módulo com Linux embarcado contendo o GCC.

#### 3.2 ESPECIFICAÇÃO

A especificação do protótipo foi dividida em duas subseções, 3.2.1 Especificação do software e 3.2.2 Especificação do hardware, apresentando as principais especificidades do protótipo.

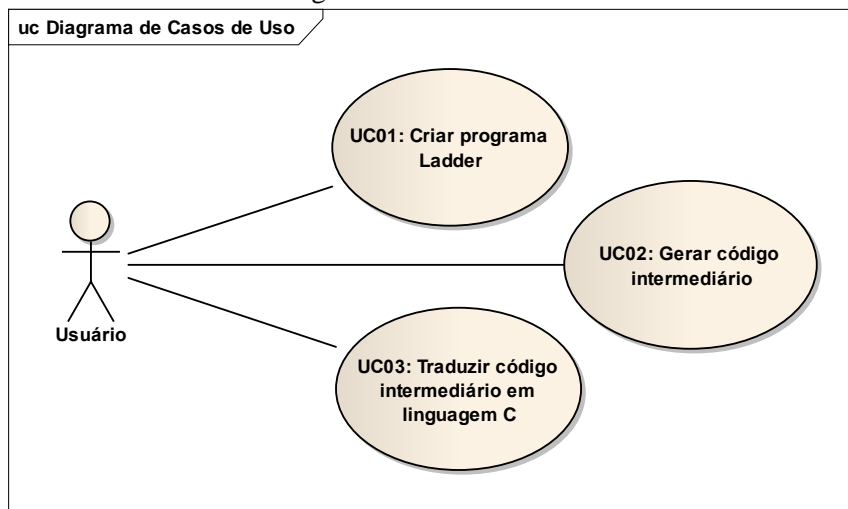
##### 3.2.1 Especificação do software

A ferramenta utilizada para a especificação do software através dos diagramas de casos de uso, pacotes, classes e sequência foi o Enterprise Architect.

##### 3.2.1.1 Casos de uso

Nesta seção são descritos os casos de uso relacionados ao protótipo conforme a Figura 10. O protótipo possui três casos de uso executados por um ator *Usuário*, que representa a pessoa que faz uso do editor gráfico.

Figura 10 – Casos de uso



A seguir são descritos os casos de uso. Nas descrições foram omitidas as funcionalidades de cancelar e sair da aplicação.

#### 3.2.1.1.1 Criar programa Ladder

O caso de uso *Criar programa Ladder* (Quadro 4), descreve como o usuário pode criar ou editar um programa Ladder. Além do cenário principal o caso de uso possui três cenários alternativos.

Quadro 4 – Caso de uso UC01

<b>UC01 – Criar programa Ladder: permite ao Usuário criar ou editar um programa Ladder, inserindo, editando ou removendo instruções.</b>	
<b>Requisitos atendidos</b>	RF01.
<b>Pré-condições</b>	Não possui.
<b>Cenário principal</b>	1) Usuário seleciona a opção criar ou abrir projeto.
	2) Software cria ou abre o projeto.
	3) Usuário insere, edita ou remove instruções Ladder.
	4) Software atualiza o projeto.
	5) Usuário salva projeto.
	6) Software grava as informações referentes ao projeto.
<b>Fluxo alternativo 01</b>	Durante o passo 3 do cenário principal, o usuário deseja incluir <i>rung</i> . 1) Usuário insere <i>rung</i> . 2) Software atualiza o projeto. 3) Volta ao passo 3 do cenário principal.
<b>Fluxo alternativo 02</b>	Durante o passo 3 do cenário principal, o usuário deseja excluir <i>rung</i> . 1) Usuário seleciona <i>rung</i> . 2) Usuário exclui <i>rung</i> . 3) Software atualiza o projeto. 4) Volta ao passo 3 do cenário principal.
<b>Fluxo alternativo 03</b>	Durante o passo 3 do cenário principal, o usuário deseja associar uma variável (ver seção 3.2.1.2.2) a uma instrução. 1) Usuário associa uma variável a uma instrução. 2) Software atualiza o projeto. 3) Volta ao passo 3 do cenário principal.
<b>Pós-condições</b>	Não possui.

### 3.2.1.1.2 Gerar código intermediário

O caso de uso Gerar código intermediário (Quadro 5) descreve como o usuário pode gerar o código intermediário. O caso de uso possui um cenário alternativo e dois cenários de exceção, porém a pré-condição é possuir um programa Ladder criado.

Quadro 5 – Caso de uso UC02

<b>UC02 - Gerar código intermediário: permite ao Usuário gerar código intermediário respectivo ao programa criado, disponibilizando um arquivo com o código gerado.</b>	
<b>Requisitos atendidos</b>	RF02.
<b>Pré-condições</b>	Programa Ladder criado.
<b>Cenário principal</b>	1) Usuário seleciona a opção para gerar código intermediário.
	2) Software realiza a análise semântica do programa Ladder.
	3) Software gera código intermediário em memória.
	4) Software cria o arquivo com o código intermediário gerado.
<b>Fluxo alternativo 01</b>	No passo 1 do cenário principal, caso o projeto não esteja salvo.
	1) Software solicita salvar o projeto.
	2) Usuário salva o projeto.
	3) Software salva as informações referentes ao projeto.
<b>Fluxo de exceção 01</b>	No passo 2 do cenário principal, caso ocorra erro ou alerta na análise semântica.
	1) Software informa o erro ou alerta.
	2) Volta ao passo 2 do cenário principal.
<b>Fluxo de exceção 02</b>	No passo 3 do cenário principal, caso ocorra algum erro semântico.
	1) Finaliza a geração de código intermediário.
<b>Pós-condições</b>	Não possui.

### 3.2.1.1.3 Traduzir código intermediário em linguagem C

O caso de uso Traduzir código intermediário em linguagem C (Quadro 6) descreve como o usuário pode traduzir o código intermediário em código C. O caso de uso possui apenas um cenário principal, porém a pré-condição é possuir o código intermediário gerado.

Quadro 6 – Caso de uso UC03

<b>UC03 – Traduzir código intermediário em linguagem C: permite ao Usuário traduzir o código intermediário gerado para código em linguagem C, disponibilizando um arquivo com o código traduzido.</b>	
<b>Requisitos atendidos</b>	RF03.
<b>Pré-condições</b>	Código intermediário gerado.
<b>Cenário principal</b>	1) Usuário seleciona a opção para traduzir o código intermediário.
	2) Software traduz o código intermediário em código na linguagem C.
	3) Software cria o arquivo com o código C gerado.
<b>Pós-condições</b>	Não possui.

### 3.2.1.2 Diagrama de classes

Para permitir o entendimento das principais classes do software, o diagrama de classes foi dividido em três seções, pacotes das instruções Ladder, classes do dispositivo de hardware

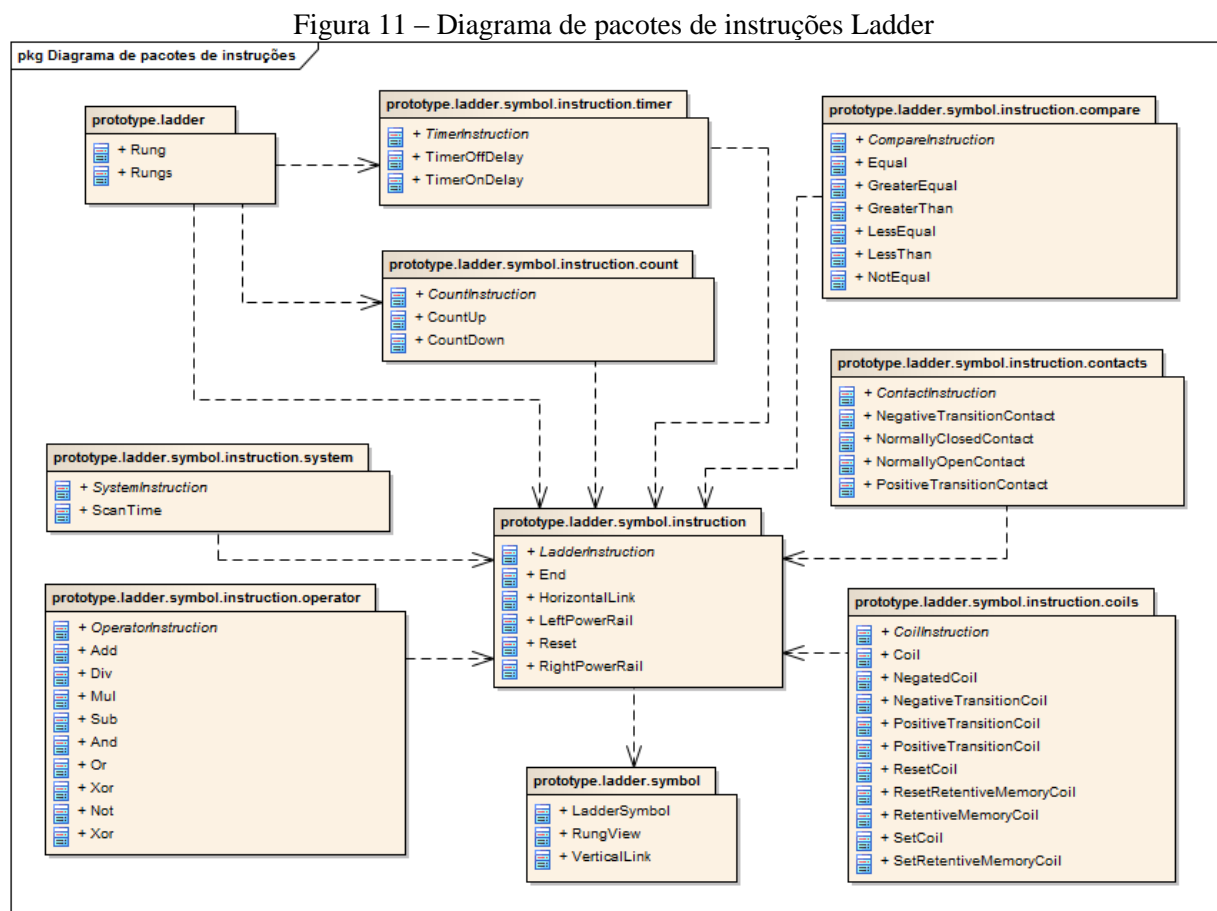


e classes do gerador e tradutor de código. Nos diagramas de classes foram omitidos os métodos *set* e *get* dos atributos e os métodos sublinhados representam os métodos estáticos.

### 3.2.1.2.1 Diagrama de pacotes das instruções Ladder

Cada instrução Ladder é um objeto na aplicação que tem a finalidade de desenhar a instrução, gerar o respectivo código intermediário, realizar parte da análise semântica e gerenciar as propriedades da instrução. O Apêndice A demonstra um diagrama de classes que evidencia a herança das instruções Ladder com os métodos e atributos omitidos.

Conforme a Figura 11 as classes das instruções Ladder estão reunidas em pacotes, representando a dependência entre si.



O pacote `prototype.ladder` possui as classes responsáveis pela estrutura do programa Ladder. Estas classes permitem adicionar, remover e atualizar instruções e *rungs*. O pacote `prototype.ladder.symbol` possui as classes referentes aos símbolos da linguagem. Todos os elementos da linguagem são símbolos Ladder e herdam a classe abstrata `LadderSymbol`, responsável por definir a orientação e a área dos símbolos Ladder na tela. O pacote `prototype.ladder.symbol.instruction` possui as classes de instruções Ladder que não derivam de nenhuma instrução específica. Todas as instruções Ladder estendem a classe

abstrata `LadderInstruction`, que permite as instruções gerar código intermediário, realizar a análise semântica que compete à classe e gerenciar as propriedades da instrução.

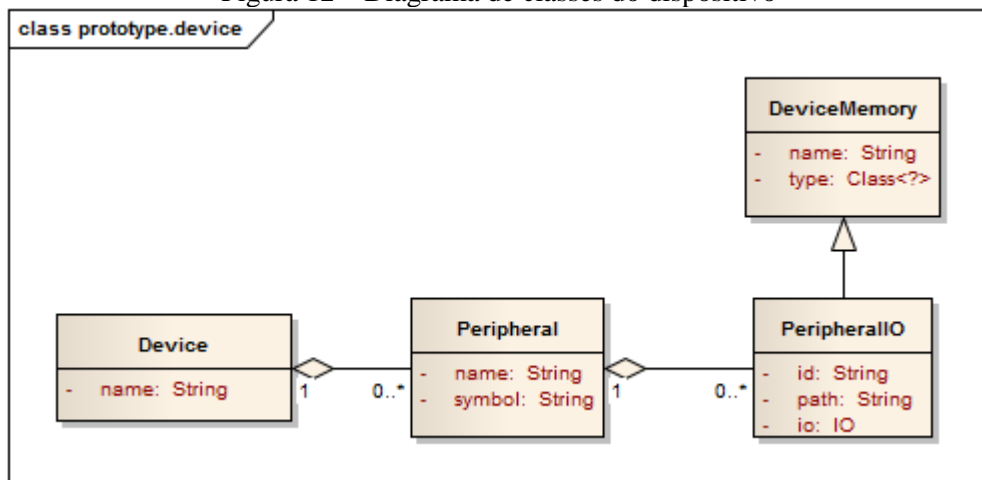
Todos os demais pacotes de instruções possuem uma classe abstrata que é estendida pelas instruções do pacote. Esta classe é responsável por implementar a análise semântica da instrução e a gerência das propriedades das instruções do pacote.

O pacote `prototype.ladder.symbol.instruction.coil` possui as classes que derivam da instrução `Coil`. O pacote `prototype.ladder.symbol.instruction.contact` possui as classes que derivam da instrução `Contact`. O pacote `prototype.ladder.symbol.instruction.compare` possui as classes que implementam a lógica de comparação. O pacote `prototype.ladder.symbol.instruction.operator` possui as classes que implementam as instruções de operadores lógicos e aritméticos, que justamente possuem esta nomenclatura no editor gráfico. O pacote `prototype.ladder.symbol.instruction.time` possui as classes que implementam as instruções de temporização. O pacote `prototype.ladder.symbol.instruction.count` possui as classes que implementam as instruções do tipo contador. O pacote `prototype.ladder.symbol.instruction.system` possui as classes que implementam as instruções do sistema.

### 3.2.1.2.2 Diagrama de classes do dispositivo

O diagrama de classes do dispositivo (Figura 12) possui as classes que representam as variáveis, os periféricos e interfaces E/S.

Figura 12 – Diagrama de classes do dispositivo



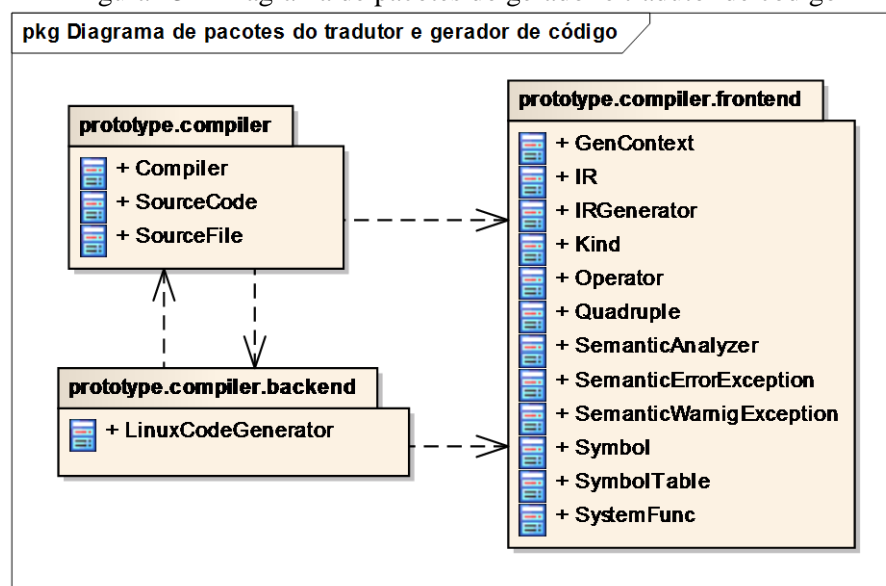
A classe `DeviceMemory` merece destaque, pois representa uma variável que pode ser associada a uma instrução `Ladder` (variável associativa). Essa variável pode representar na aplicação uma variável retentiva (variável de propósito geral), interface E/S, temporizador,

contador ou variável de instrução (compõem um temporizador ou contador). A classe `PeripheralIO` é responsável por identificar uma interface E/S de um periférico. Ela contém o nome absoluto de um arquivo do Linux que representa a interface E/S. A enumeração `IO` da classe `PeripheralIO` define se a interface de periférico é entrada ou saída. A classe `Peripheral` é responsável por identificar um periférico através do nome e símbolo usado para representar o dispositivo. A classe `Device` refere-se ao hardware em si, ela representa o dispositivo, seus periféricos e suas interfaces.

### 3.2.1.2.3 Diagrama de classes do gerador e tradutor de código

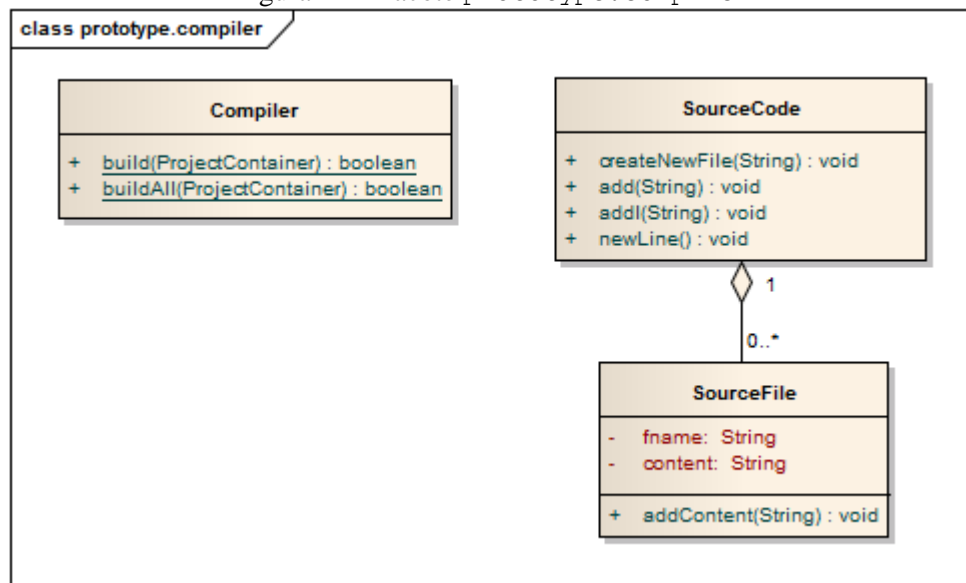
O diagrama de classes do gerador e tradutor de código (Figura 13) possui as classes responsáveis por gerar e traduzir o programa fonte Ladder para código intermediário e código em linguagem C. As classes estão reunidas em pacotes representando as dependências entre si, de acordo com as subseções a seguir.

Figura 13 – Diagrama de pacotes do gerador e tradutor de código



### 3.2.1.2.4 Pacote `prototype.compiler`

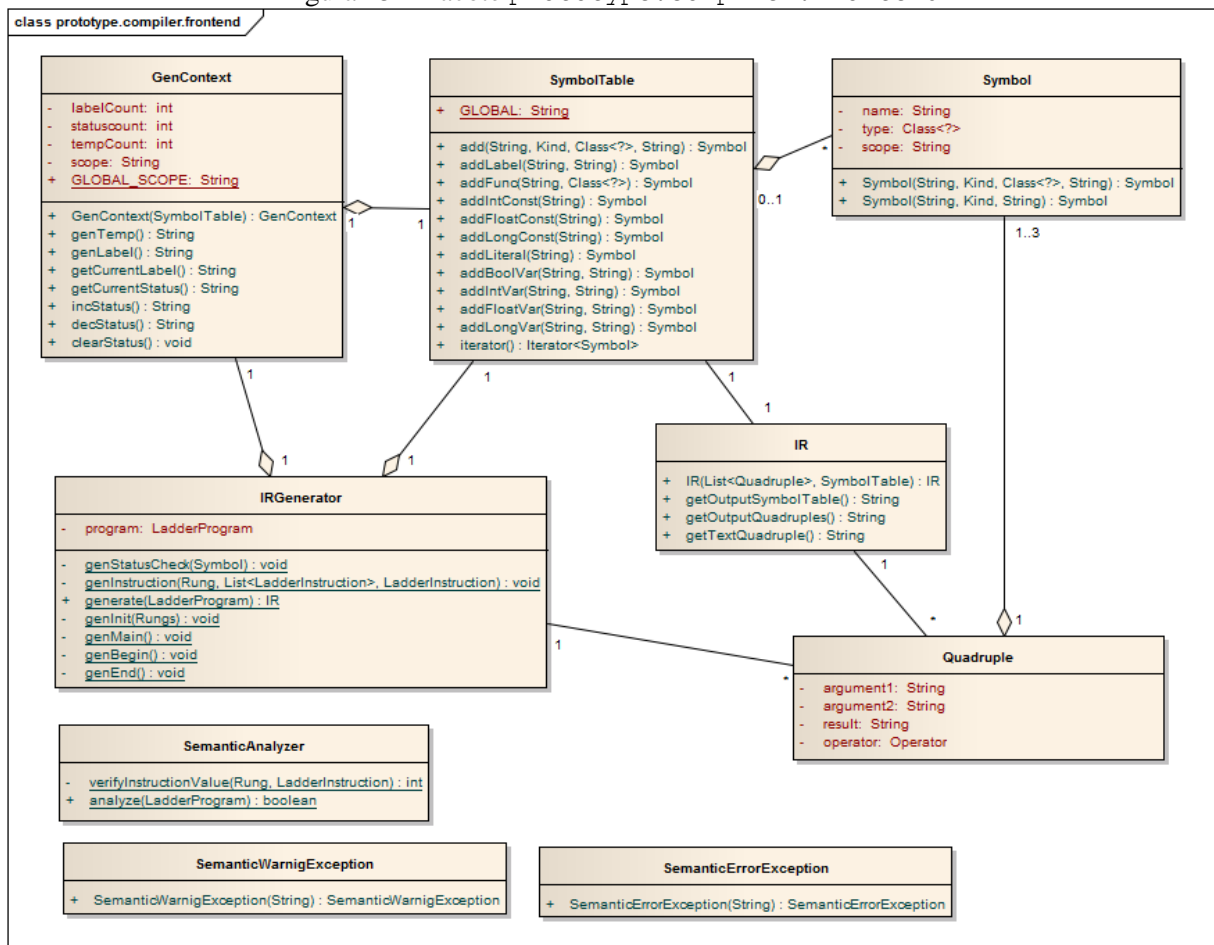
O pacote denominado `prototype.compiler` (Figura 14) possui as classes que executam a compilação do programa fonte Ladder e encapsulam o código gerado.

Figura 14 – Pacote `prototype.compiler`

A classe `Compiler` implementa os principais métodos responsáveis por gerar o código intermediário e traduzi-lo em código na linguagem C. A classe `SourceFile` é responsável por representar um arquivo de código fonte, possui como atributos o nome e conteúdo do arquivo. A classe `SourceCode` permite criar e armazenar arquivos de código fonte.

#### 3.2.1.2.5 Pacote `prototype.compiler.frontend`

O pacote denominado `prototype.compiler.frontend` (Figura 15) possui as classes responsáveis por realizar a análise semântica e a geração de código intermediário.

Figura 15 – Pacote `prototype.compiler.frontend`

A classe `Quadruple` representa uma quádrupla do código de três endereços. O enumerado `Operator` representa o operador da quádrupla. A classe `Symbol` é responsável por complementar as informações dos argumentos e resultado de uma quádrupla, através do tipo e escopo. O enumerado `Kind` da classe `Symbol` representa a categoria do símbolo como sendo uma variável, constante, rótulo ou função. A classe `IR` é responsável pela representação intermediária do programa `Ladder`. A classe `SymbolTable` representa uma tabela responsável por criar e evitar a duplicidade de símbolos. A classe `GenContext` representa o contexto do gerador de código intermediário, é responsável por gerenciar as variáveis temporárias, rótulos e *status* do *rung*. A classe `IRGenerator` é responsável por gerar o código intermediário. A classe `SemanticAnalyzer` é responsável por efetuar a análise semântica do programa fonte `Ladder`. Esta classe lança as exceções `SemanticErrorException` e `SemanticWarningException` caso ocorram erros ou alertas respectivamente.

### 3.2.1.2.6 Pacote `prototype.compiler.backend`

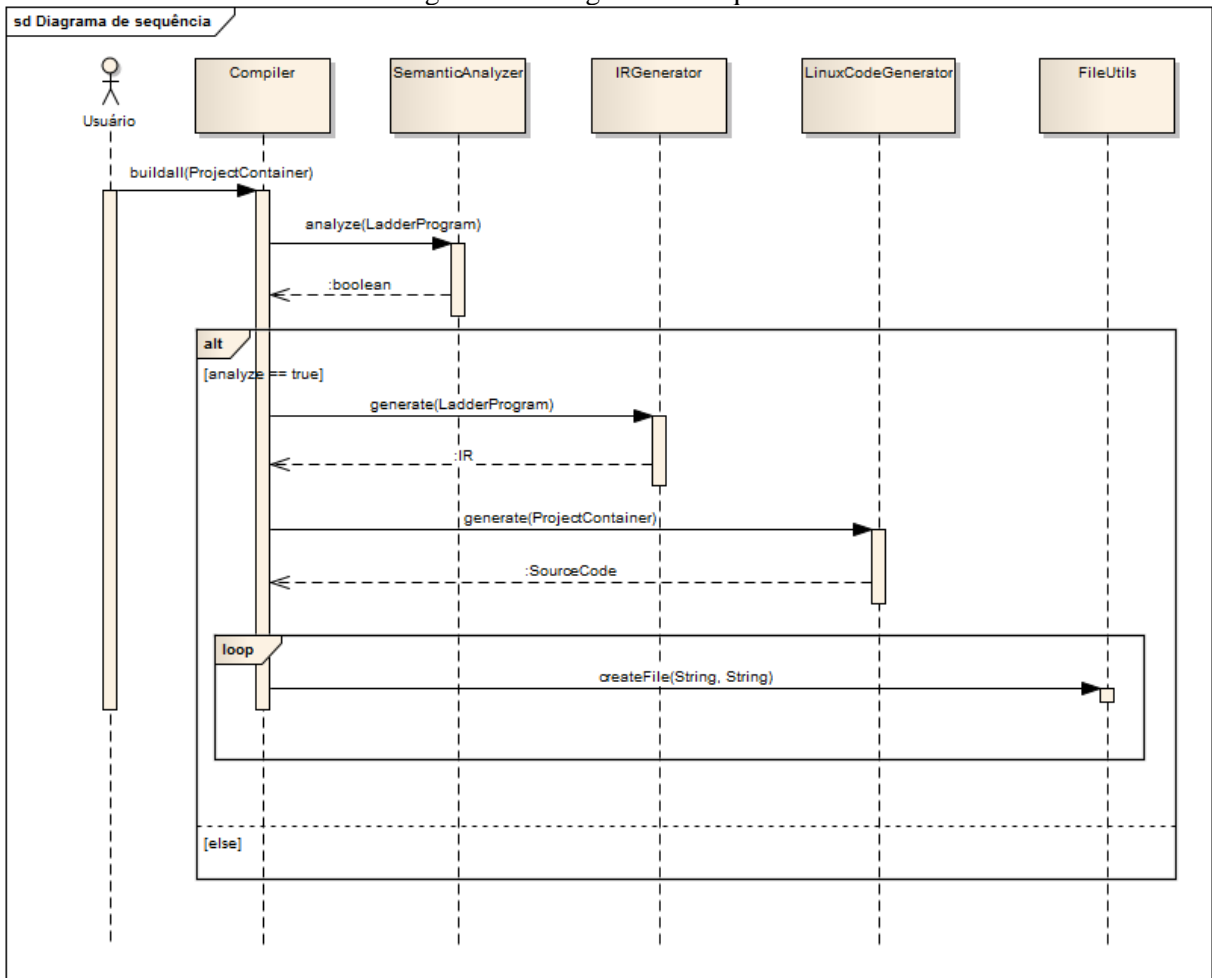
O pacote denominado `prototype.compiler.backend` (Figura 16) possui apenas a classe responsável pela tradução do código intermediário para código em linguagem C.

Figura 16 – Pacote `prototype.compiler.backend`

### 3.2.1.3 Diagrama de sequência

O diagrama de sequência (Figura 17) apresenta uma visão da comunicação entre as classes para a rotina `buildall`, referente aos casos de uso UC02 e UC03.

Figura 17 – Diagrama de sequência



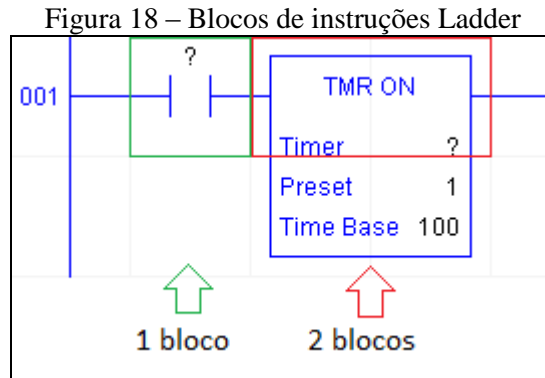
A classe `Compiler` recebe a solicitação do ator `Usuário` para iniciar o processo de geração de código intermediário e tradução para linguagem C. Por sua vez esta realiza uma chamada para a classe `SemanticAnalyzer` que efetua a análise semântica do programa Ladder. Caso a análise ocorra com sucesso a classe `IRGenerator` efetua a geração de código intermediário de todas as instruções, retornando o código gerado. Este código é enviado a classe `LinuxCodeGenerator` que o traduzirá para linguagem C e retornará o código traduzido. Ao fim do processo os arquivos com o código intermediário e o código em linguagem C são gerados e gravados em disco através da classe `FileUtils`.

#### 3.2.1.4 Lexemas, sintaxe e semântica.

A seguir são descritos os lexemas, sintaxe e semântica da linguagem Ladder especificados do protótipo, baseados na norma da seção 2.3.

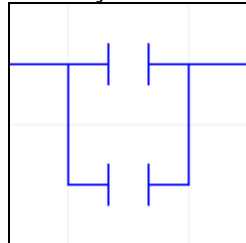
Pelo fato da linguagem Ladder se tratar de uma linguagem gráfica de diagramas não há uma análise léxica definida. A especificação léxica determina que os símbolos da linguagem Ladder sejam os *tokens*. O Apêndice B apresenta um quadro com os símbolos da linguagem

Ladder do protótipo. Um símbolo Ladder pode ser composto por um ou mais blocos conforme pode ser visto na Figura 18. A instrução que possui um quadrado verde ocupa um bloco e a instrução que possui um retângulo vermelho ocupa dois blocos. Estes blocos representam a área ocupada pela instrução no editor gráfico e tem a finalidade de orientar o usuário quanto à disponibilidade no *run* para instruções.



Os símbolos Ladder podem ser representados em paralelo através de linhas verticais conforme a Figura 19, que ilustra duas instruções Ladder em paralelo.

Figura 19 – Instruções Ladder em paralelo



As variáveis associativas (ver seção 3.2.1.2.2) que não representam uma variável de uma instrução possuem uma representação em texto formada por um rótulo que identifica o tipo da variável e um número que identifica a variável em um conjunto do mesmo tipo. O Quadro 7 demonstra o rótulo das variáveis, um exemplo da primeira variável do tipo e sua respectiva descrição.

Quadro 7 – Identificação das variáveis associativas

Rótulo	Exemplo	Descrição
I	I01	Interface de entrada.
Q	Q01	Interface de saída.
MI	MI01	Variável retentiva do tipo inteiro.
MF	MF01	Variável retentiva do tipo ponto flutuante.
T	T1	Instrução de temporização.
C	C1	Instrução de contagem.

As variáveis associativas que representam uma variável de instrução (temporizador ou contador) possuem uma representação em texto formada por um rótulo que identifica o nome da variável. Estas variáveis são identificadas no editor através da junção da representação da



instrução, o caractere dois pontos e o rótulo da variável. O Quadro 8 demonstra o rótulo das variáveis de instrução, um exemplo da primeira variável do tipo e sua respectiva descrição.

Quadro 8 – Identificação das variáveis de instrução

Rótulo	Exemplo	Descrição
PRE	T1:PRE	Variável do tipo inteiro que representa um valor pré-definido em uma instrução do tipo temporizador ou contador.
AC	T1:AC	Variável do tipo inteiro que representa um acumulador em uma instrução do tipo temporizador ou contador.
DN	T1:DN	Variável do tipo booleana que representa o término da ação executada em uma instrução do tipo temporizador ou contador.
EN	T1:EN	Variável do tipo booleana que representa a execução de uma ação em uma instrução do tipo temporizador.
CC	C1:CC	Variável do tipo booleana que representa a execução de uma ação em uma instrução do tipo contador.

As instruções possuem restrições quanto à associação de variáveis que podem ser efetuadas através do editor gráfico. O Quadro 9 demonstra a relação de associação permitida por instrução.

Quadro 9 – Compatibilidade entre instruções e variáveis associativas

Instrução Ladder	Variável associativa
Normally open Contact	I, Q, MI, MF, PRE, AC, DN,EN, CC
Normally closed contact	I, Q, MI, MF, PRE, AC, DN,EN, CC
Positive transition-sensing contact	I, Q, MI, MF, PRE, AC, DN,EN, CC
Negative transition-sensing contact	I, Q, MI, MF, PRE, AC, DN,EN, CC
Coil	I, Q, MI, MF, PRE, AC, DN,EN, CC
Negated coil	I, Q, MI, MF, PRE, AC, DN,EN, CC
Set coil	I, Q, MI, MF, PRE, AC, DN,EN, CC
Reset coil	I, Q, MI, MF, PRE, AC, DN,EN, CC
Retentive coil	I, Q, MI, MF, PRE, AC, DN,EN, CC
Set retentive coil	I, Q, MI, MF, PRE, AC, DN,EN, CC
Reset retentive coil	I, Q, MI, MF, PRE, AC, DN,EN, CC
Positive transition-sensing coil	I, Q, MI, MF, PRE, AC, DN,EN, CC
Negative transition-sensing coil	I, Q, MI, MF, PRE, AC, DN,EN, CC
End	
Reset	T, C
Timer On	
Timer Off	
Count Up	
Count Down	
Equal	I, Q, MI, MF, PRE, AC, DN,EN, CC
Greater Equal	I, Q, MI, MF, PRE, AC, DN,EN, CC
Greater Than	I, Q, MI, MF, PRE, AC, DN,EN, CC
Less Equal	I, Q, MI, MF, PRE, AC, DN,EN, CC
Less	I, Q, MI, MF, PRE, AC, DN,EN, CC
Not Equal	I, Q, MI, MF, PRE, AC, DN,EN, CC
And	I, Q, MI, MF, PRE, AC, DN,EN, CC
Not	I, Q, MI, MF, PRE, AC, DN,EN, CC
Or	I, Q, MI, MF, PRE, AC, DN,EN, CC
Xor	I, Q, MI, MF, PRE, AC, DN,EN, CC
Add	I, Q, MI, MF, PRE, AC, DN,EN, CC

Instrução Ladder	Variável associativa
Div	I, Q, MI, MF, PRE, AC, DN,EN, CC
Mul	I, Q, MI, MF, PRE, AC, DN,EN, CC

A sintaxe do programa Ladder refere-se às restrições quanto aos *rungs* e instruções. A sintaxe do programa determina que:

- a) deve haver no mínimo um *rung* no programa;
- b) a última instrução do último *rung* dever ser a instrução `End`;
- c) a instrução `End` não permite instruções em paralelo;
- d) não pode haver mais de uma instrução `End` no programa;
- e) instruções podem ser inseridas em qualquer ordem ou *rung*;
- f) instruções com 2 blocos necessitam de 2 blocos livres (`Horizontal link`) sequenciais para serem adicionadas;
- g) instruções em paralelo não podem ocupar mais de um bloco no editor gráfico;
- h) não há limite de instruções em paralelo;
- i) as instruções devem permitir associação de variáveis de acordo com o Quadro 9;
- j) as instruções do tipo operador (lógico ou aritmético) e comparação, devem permitir associação de variáveis a todos os operandos;
- k) as instruções do tipo operador (lógico ou aritmético) e comparação, devem permitir definir valores constantes para os operandos, com exceção ao operando `result` das instruções do tipo operador.

A semântica do programa Ladder refere-se às variáveis associativas. A semântica do programa determina que:

- a) instruções que permitem associação de variáveis devem possuir variável associada ou valores constantes se a instrução os suportar;
- b) instruções que permitem a associação de mais de uma variável como do tipo operador (lógico ou aritmético) e comparação, devem possuir variáveis associadas a todos os operandos ou valores constantes;
- c) variáveis que representam interfaces E/S podem ser associadas as instruções compatíveis, independente do significado da instrução Ladder. Por exemplo, uma instrução Ladder que representa uma saída pode ser associada a uma variável que representa uma interface de entrada e vice-versa;
- d) não podem haver instruções do tipo temporizador e contador com o mesmo número.

### 3.2.1.5 Código intermediário

O código intermediário de três endereços que representa o programa fonte Ladder permite representar instruções de acordo com o Quadro 10.

Quadro 10 – Instruções do código intermediário

Instrução	Descrição
<code>x = y op z</code>	Instrução de atribuição onde <code>op</code> é um operador, <code>x</code> é uma variável e <code>y</code> e <code>z</code> podem ser variável ou constante.
<code>x = op y</code>	Instrução de atribuição onde <code>op</code> é um operador unário, essencialmente uma negação, <code>x</code> é uma variável, <code>y</code> pode ser uma variável ou constante.
<code>x = y</code>	Instrução de atribuição onde <code>x</code> é uma variável e <code>y</code> pode ser uma variável ou constante.
<code>goto L</code>	Desvio incondicional para o rótulo <code>L</code> .
<code>if x goto L</code>	Desvio condicional que executa as instruções do rótulo <code>L</code> se <code>x</code> for verdadeiro (valor diferente de zero). <code>x</code> é uma variável.
<code>ifFalse x goto L</code>	Desvio condicional que executa as instruções do rótulo <code>L</code> se <code>x</code> for falso (valor igual a zero). <code>x</code> é uma variável.
<code>if x relop y goto L</code>	Desvio condicional que aplica um operador relacional <code>relop</code> ( <code>&lt;</code> , <code>&gt;</code> , <code>==</code> , <code>&gt;=</code> , <code>&lt;=</code> , <code>!=</code> ) a <code>x</code> e <code>y</code> e executa em seguida as instruções do rótulo <code>L</code> se a operação relacional for verdadeira. <code>x</code> e <code>y</code> são variáveis ou constantes.
<code>ifFalse x relop y goto L</code>	Desvio condicional que aplica um operador relacional <code>relop</code> ( <code>&lt;</code> , <code>&gt;</code> , <code>==</code> , <code>&gt;=</code> , <code>&lt;=</code> , <code>!=</code> ) a <code>x</code> e <code>y</code> e executa em seguida as instruções do rótulo <code>L</code> se a operação relacional for falsa. <code>x</code> e <code>y</code> são variáveis ou constantes.
<code>call p,n</code>	Chamadas de rotinas para o rótulo <code>p</code> , onde <code>n</code> é um valor constante referente à quantidade de parâmetros. As rotinas são encerradas com a instrução <code>return</code> .
<code>r = call p,n</code>	Chamadas de rotinas para o rótulo <code>p</code> , onde <code>n</code> é um valor constante referente à quantidade de parâmetros e <code>r</code> é uma variável. As rotinas são encerradas com a instrução <code>return x</code> , onde <code>x</code> é a variável de retorno.

Fonte: Adaptado de Aho (2008, p. 232).

O código intermediário também permite representar variáveis, constantes e rótulos, conforme especificado a seguir:

- variável: possui a mesma nomenclatura da especificação léxica, por exemplo `I01` e `T1:DN`. Instruções de detecção de borda possuem uma variável que indica o estado da variável no *scan* anterior, essa variável inicia com a palavra `LAST_`, seguida do nome da variável, por exemplo, `LAST_I01`. A variável de *status* criada pelo gerador de código é formada pela junção de `_S` e o número do status, por exemplo, `_S1`. A variável temporária de propósito geral criada pelo gerador de código é formada pela junção de `_T` e o número da variável, por exemplo, `_T1`;
- constante: é representada pelo seu valor numérico. A constante booleana é representada pelo caractere `1` para o estado *true* e `0` para o estado *false*;

- c) rótulos: os rótulos representam rotinas e trechos de código e são formados pelo nome seguido do caractere dois pontos. Rótulos que identificam um *rung* possuem o nome `Rung` seguido do número do *rung*, por exemplo, `Rung001:`. Os rótulos utilizados para desvios de trechos de código em um *rung* são formados pelo caractere `L` seguido do número do rótulo, por exemplo, `L1:`. Rótulos de rotinas do sistema iniciam com dois *underline*, por exemplo, `__init.`

O código intermediário gerado a partir de instruções Ladder deve respeitar as seguintes regras no que diz respeito a desvios incondicionais ou condicionais:

- a) não pode haver desvios para linhas que antecedam o desvio atual;
- b) não pode haver desvios de um *rung* para outro, ou outra rotina;
- c) desvio condicional (`if-goto`) não pode desviar para além das linhas de um desvio condicional anterior (rótulo), ainda não alcançado;
- d) desvios incondicionais (`goto`) devem representar a condição “se não” em relação a uma estrutura de controle, devem estar precedidos por um desvio condicional (`if-goto`) e possuir um rótulo como próximo elemento, proveniente de um desvio condicional.

Estas regras são essenciais para a etapa de tradução de código intermediário para código em linguagem C.

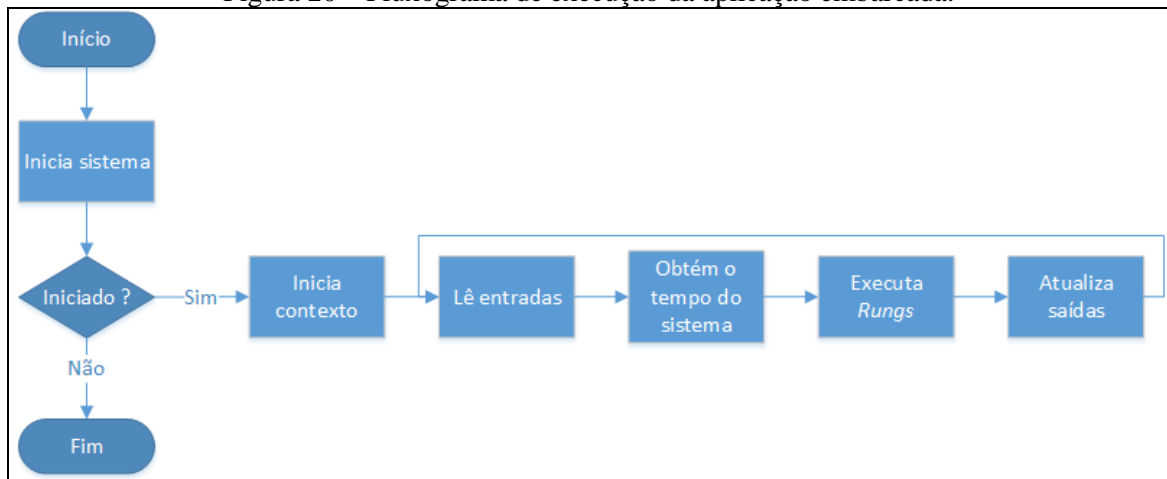
### 3.2.2 Especificação do hardware

As ferramentas utilizadas para a especificação do hardware foram o Microsoft Visio para o fluxograma e o Proteus Labcenter Electronics para o diagrama esquemático.

#### 3.2.2.1 Fluxograma

O fluxograma ilustrado na Figura 20 oferece uma visão sobre a execução da aplicação embarcada resultante do processo de geração e tradução de código. A aplicação é composta por seis processos distintos, sendo eles: `Inicia sistema`, `Inicia contexto`, `Lê entradas`, `Obtém o tempo do sistema`, `Executa rungs` e `Atualiza saídas`.

Figura 20 – Fluxograma de execução da aplicação embarcada.



Quando o circuito é ligado o processo *Inicia sistema* obtém o acesso aos recursos de hardware necessários à aplicação, processo este que consiste em abrir os arquivos correspondentes a entradas e saídas definidas no programa fonte Ladder. Os arquivos permanecem abertos durante toda a execução do programa para maximizar o tempo de leitura e escrita. Caso algum arquivo não possa ser aberto, a aplicação será encerrada.

O processo *Inicia contexto* é responsável por inicializar as variáveis relacionadas às instruções de contagem e temporização. Durante o processo *Lê entradas* ocorre a leitura das interfaces de entrada, mantendo seus estados em memória até que a próxima leitura ocorra ou que o programa fonte Ladder as modifique. Somente após a execução dos *rungs* que o processo *Atualiza saídas* atualizará o estado das interfaces de saída, que também são mantidas em memória durante a execução do programa.

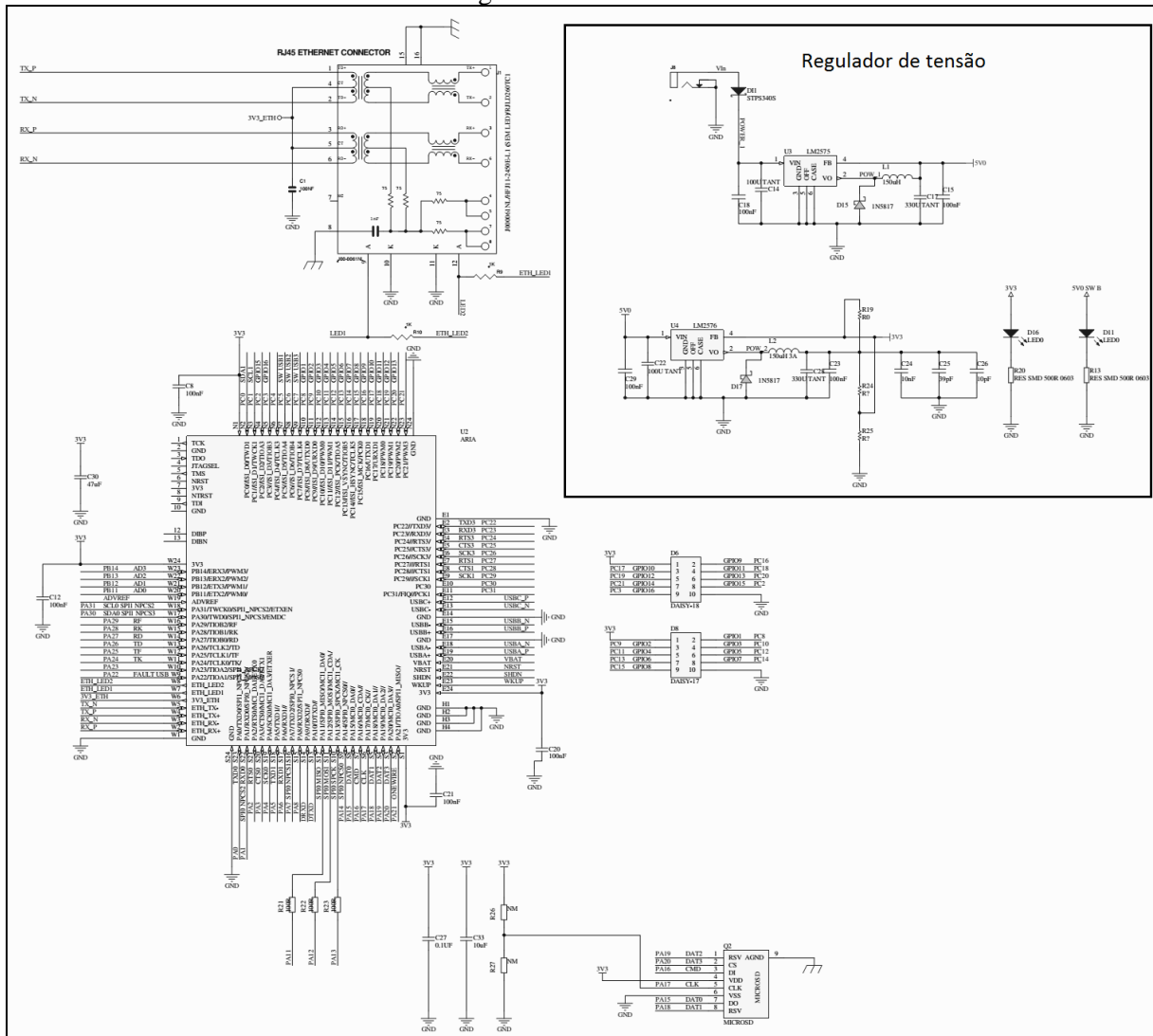
A semântica do programa Ladder é garantida pela sequência de execução dos processos: *Lê entradas*, *Executa Rungs* e *Atualiza saídas*. O processo *Executa Rungs* está representado de forma reduzida uma vez que a quantidade de *rungs* variam conforme cada programa Ladder. Neste processo os *rungs* são executados em sequência de acordo com o programa fonte Ladder.

Um processo denominado *Obtém o tempo do sistema* ocorre durante o ciclo de repetição do programa. Este é responsável por obter o tempo do sistema que será utilizado como base de tempo pelas instruções de temporização.

### 3.2.2.2 Diagrama esquemático

O protótipo de hardware é composto pelo módulo Aria G25 conforme o item 2.6, a placa mãe (Figura 21) e a placa de testes (Figura 22).

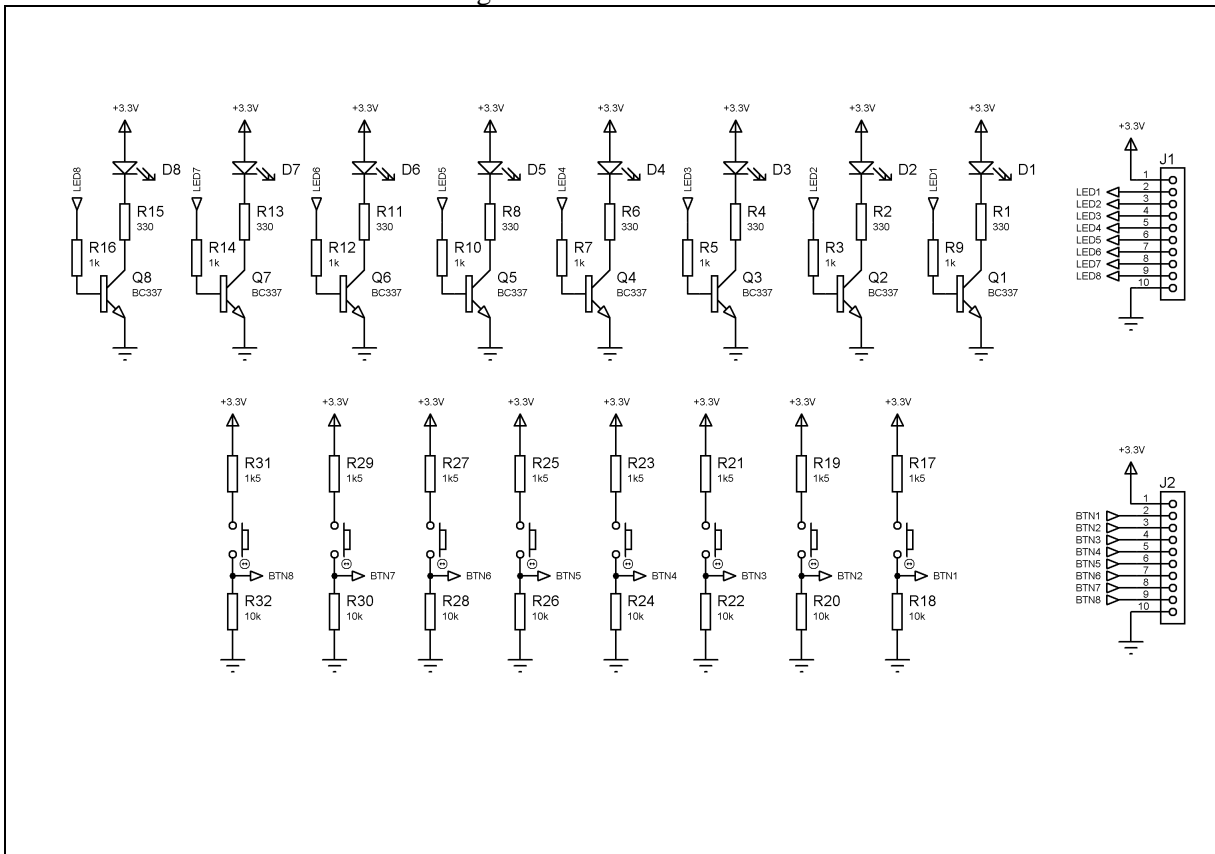
Figura 21 – Placa mãe



Fonte: Adaptado de Basic4ever (2014).

O diagrama da placa mãe foi adaptado tendo os elementos de hardware que não são necessários ao protótipo omitidos do diagrama.

Figura 22 – Placa de testes



A placa de testes é composta por oito atuadores (*LEDs*) e oito sensores (Chaves Tátil) que representam respectivamente as interfaces de saída e entrada.

### 3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas no software e hardware e a operacionalidade da implementação.

#### 3.3.1 Técnicas e ferramentas utilizadas

O protótipo foi desenvolvido utilizando duas linguagens de programação, Java na versão 7 e C na versão C90, além de uma representação intermediária do programa fonte Ladder. A linguagem de programação Java foi utilizada para o desenvolvimento do editor gráfico da linguagem Ladder baseado na norma IEC 61131-3. A linguagem C é utilizada no código que será gerado pela ferramenta e será compilado no hardware. A representação intermediária do programa fonte Ladder foi implementada de acordo com o conceito de código de três endereços e quádruplas abordados na seção 2.7.

As ferramentas utilizadas foram o Eclipse IDE for Java Developers para o desenvolvimento do software e o GCC versão 4.4.5 para compilar o código C. As APIs utilizadas foram o Java2D para desenvolvimento do editor gráfico e o Java Secure Channel

para efetuar a conexão com o hardware através do protocolo SSH. A distribuição do Linux fornecida pela ACMESYSTEMS (2014) para o módulo Aria G25 utilizado no protótipo foi o Debian GNU/Linux Kernel 2.6.39.

As seções a seguir descrevem detalhes sobre a implementação do gerador e tradutor de código e também a montagem do hardware.

### 3.3.1.1 Gerador e tradutor de código

Esta seção descreve a implementação do gerador e tradutor de código conforme especificado na seção 3.2.1.2.3, sendo dividida em quatro subseções, restrições sintáticas, análise semântica, gerador de código intermediário e tradutor de código.

#### 3.3.1.1.1 Restrições sintáticas

As restrições sintáticas ocorrem em tempo de edição do programa Ladder, ou seja, o editor não permite criar um programa que não esteja de acordo com a sintaxe especificada na seção 3.2.1.4. O Quadro 11 apresenta o método `add` da classe `Rung`, responsável por adicionar uma nova instrução no *rung*.

Quadro 11 – Implementação do método `add` da classe `Rung`

```

1 public boolean add(int x,int y,LadderInstruction instruction){
2     LadderInstruction li = getInstruction(x, y);
3     if(li != null){
4         if(li instanceof HorizontalLink){
5             return insertOverride(li, instruction);
6         }
7         if(!(li instanceof End) &&
8             !(li instanceof LeftPowerRail) &&
9             !(li instanceof RightPowerRail)){
10            return insertParallel(li, instruction);
11        }
12    }
13    return false;
14 }
```

Na linha 2 o método `getInstruction` itera sobre as instruções do *rung* retornando a instrução correspondente as coordenadas `x` e `y`. Caso não exista instrução para a respectiva coordenada o método `add` retorna `false`.

Conforme pode ser visto nas linhas 4 e 5, o método `insertOverride` sobrescreve a instrução correspondente as coordenadas `x` e `y` pela instrução passada por parâmetro se a instrução identificada for uma instância de `HorizontalLink` (instrução que representa uma região disponível para uma nova instrução). A implementação do método sobrescreve a instrução somente se houver espaço disponível no *rung* para a nova instrução, caso contrário ele retorna `false`.



O método `insertParallel` na linha 10 insere a instrução passada por parâmetro em paralelo a instrução correspondente as coordenadas  $x$  e  $y$ , se esta, não for um elemento estrutural da linguagem Ladder (linhas 8 e 9), ou uma instrução de término do programa (linha 7). A implementação do método insere a instrução em paralelo somente se a nova instrução e a instrução identificada através das coordenadas ocuparem apenas um bloco de instrução na lógica Ladder.

#### 3.3.1.1.2 Análise semântica

A análise semântica ocorre por meio de uma ação do usuário que deseja compilar o programa fonte Ladder. Ela ocorre antes da geração e tradução de código. O objetivo do analisador semântico do protótipo é verificar se todas as instruções Ladder do programa possuem variáveis associadas e as instruções do tipo temporizador e contador não estão duplicadas.

Parte das restrições semânticas ocorrem em tempo de edição do programa Ladder. As variáveis que uma instrução aceita estão implementadas através do método `addMemory` da classe `LadderInstruction`, os valores contidos nas variáveis de instrução e constantes que podem ser definidos são restringidos pela tela de propriedades da instrução, assim como o número das instruções do tipo temporizador e contador. O Quadro 12 demonstra a implementação da classe `SemanticAnalyzer` responsável pela análise semântica.

Quadro 12 – Implementação da classe SemanticAnalyzer

```

1 public class SemanticAnalyzer {
2
3     private static int verifyInstructionValue(Rung rung,
4                                             LadderInstruction instruction) {
5         int errs = 0;
6         if(instruction != null){
7             errs += verifyInstructionValue(rung, instruction.getDown());
8             errs += verifyInstructionValue(rung, instruction.getNext());
9             try{
10                instruction.analyze();
11            }catch(SemanticErrorException e){
12                errs++;
13                Mediator.getInstance().outputConsoleMessage(
14                    Strings.i18n.error() + "[rung="+rung.getNumber()+", col="+
15                    instruction.getCol()+", "+instruction+"]: "+
16                    e.getMessage() );
17            }catch(SemanticWarnigException e){
18                Mediator.getInstance().outputConsoleMessage(
19                    Strings.i18n.warning()+"[rung="+rung.getNumber()+", col="+
20                    instruction.getCol()+", "+instruction+"]: "+
21                    e.getMessage() );
22            }
23        }
24        return errs;
25    }
26
27    public static boolean analyze(LadderProgram ladderProgram) {
28        boolean success = true;
29        for(Rung rung:ladderProgram.getRungs()) {
30            if(verifyInstructionValue(rung, rung.getFirst()) != 0) {
31                success = false;
32            }
33        }
34        return success;
35    }
36 }

```

O método `analyze` na linha 27 realiza a análise semântica do programa Ladder recebido por parâmetro. Ele itera sobre os *rungs* do programa Ladder executando a verificação de cada *rung* através do método `verifyInstructionValue` na linha 30.

A análise das instruções ocorre de forma recursiva conforme pode ser visto nas linhas 7 e 8, sendo que cada instrução Ladder é responsável por efetuar sua própria análise (linha 10). Conforme pode ser visto nas linhas 11 e 17, a análise de uma instrução pode lançar duas exceções, `SemanticErrorException` e `SemanticWarnigException`, que significam respectivamente um erro semântico e um alerta respectivo aos atributos de uma instrução.

Independente das exceções lançadas, a análise percorre todas as instruções de todos os *rungs* contando os erros (linha 12) e exibindo as mensagens na console do editor (linhas 13 e 18). O método `analise` da classe retorna `false` se a quantidade de erros provenientes da análise for diferente de zero (linha 30).

### 3.3.1.1.3 Gerador de código intermediário

O gerador de código intermediário é a próxima fase do processo de compilação do programa fonte Ladder. É responsável por gerar uma representação intermediária do programa fonte Ladder. Essa representação implementada através de quádruplas e símbolos é mantida apenas internamente ao software, ou seja, o arquivo gerado fornece apenas uma representação em texto do código intermediário, conforme pode ser visto em um exemplo de código no Quadro 13.

Quadro 13 – Exemplo de código intermediário representado em texto.

```

1 Rung001:
2   _S0 = 1
3   return
4 InitContext:
5   return
6 main:
7   ini = call __init,0
8   ifFalse ini goto end
9   call InitContext,0
10 begin:
11  call __input,0
12  call __update,0
13  call Rung001,0
14  call __output,0
15  goto begin

```

A implementação da sequência de execução das instruções Ladder em um *rung* foi concebida através da criação de uma variável local denominada *status*, que representa o estado `true` ou `false` do *link* (ligação entre instruções Ladder) durante o *rung*. Esta variável é única por *rung* e inicia com o estado `true`, porém para cada instrução em paralelo o gerador de código cria uma cópia da variável *status* para ser utilizado pela instrução em paralelo. Após o fim do bloco paralelo é gerado código que verifica se alguma das instruções em paralelo (através da cópia da variável *status*) finalizaram com o estado `true`, que nesse caso significa que o estado da variável *status* do *rung* também será `true`. O Quadro 14 apresenta o código intermediário respectivo a instruções Ladder em paralelo.

Quadro 14 – Código intermediário respectivo a instruções Ladder em paralelo

Instruções Ladder	Código intermediário
	<pre> 1 Rung001: 2   _S0 = 1 3   _S1 = _S0 4   if I01 goto L1 5   _S0 = 0 6 L1: 7   _S2 = _S1 8   if I02 goto L2 9   _S1 = 0 10 L2: 11  if I03 goto L3 12  _S2 = 0 13 L3: 14  ifFalse _S2 goto L5 15  _S0 = 1 16  goto L4 17 L5: 18  ifFalse _S1 goto L4 19  _S0 = 1 20 L4: 21  Q01 = _S0 22  return </pre>

Nas linhas 3 e 7 é possível verificar a cópia da variável *status* (*\_S0*) para uma variável *status* temporária (*\_S1* e *\_S2*). Após as instruções em paralelo o código entre as linhas 14 e 20 verifica se alguma das variáveis de *status* temporárias (*\_S1* e *\_S2*) possuem o estado *true* (1), tornando *true* o estado da variável *status* do *rung* (*\_S0*, linhas 15 e 19). Caso nenhuma das variáveis de *status* temporárias sejam *true* a variável *status* do *rung* permanece com o estado atual.

O Quadro 15 apresenta o método `genInstruction` da classe `IRGenerator` que implementa a geração de código referente as variáveis *status* temporárias, além do código das instruções.

Quadro 15 – Implementação do método `genInstruction` da classe `IRGenerator`

```

1 private static void genInstruction(Rung rung, List<LadderInstruction>
2     parallel, LadderInstruction instruction) {
3     if(instruction != null) {
4         Symbol currentStatus = symbolTable.addBoolVar(
5             genContext.getCurrentStatus(), genContext.getScope());
6
7         // verifica se há instrução abaixo para guardar o status atual
8         if(instruction.getDown() != null) {
9             parallel.add(instruction.getDown());
10            Symbol downStatus = symbolTable.addIntVar(
11                genContext.incStatus(), genContext.getScope());
12            quadruples.add(Quadruple.createAssignment(
13                currentStatus, downStatus));
14            genContext.decStatus();
15        }
16
17        // gera o código da instrução
18        List<Quadruple> code = instruction.generateIR(genContext);
19        if(code != null) {
20            quadruples.addAll(code);
21
22            // verifica quantas instruções a referenciam como up
23            int ups = rung.getCountReferenceUp(rung.getFirst(),
24                instruction.getNext());
25            if(ups > 0) {
26                currentStatus = symbolTable.addIntVar(
27                    genContext.getCurrentStatus(), genContext.getScope());
28
29                // gera as instruções em paralelo
30                genContext.incStatus();
31                genInstruction(rung, parallel, parallel.remove(0));
32
33                // gera o código para verificar o status após paralelo
34                genStatusCheck(currentStatus);
35            } else if(parallel.size() > 0) {
36
37                // próxima instrução em paralelo
38                genContext.incStatus();
39                genInstruction(rung, parallel, parallel.remove(0));
40            }
41        }
42        // próxima instrução em série
43        genInstruction(rung, parallel, instruction.getNext());
44    }
45 }

```

O método `genInstruction` é responsável por gerar o código intermediário de um `rung`. Este método é recursivo conforme pode ser visto nas linhas 31, 39 e 43, ele itera sobre as instruções do `rung` gerando código para cada instrução e para os pontos de entrada e saída de instruções em paralelo, copiando e verificando as variáveis *status* temporárias. Na linha 4 é possível verificar a criação da variável *status* que pertence ao `rung`.

Como forma de controlar a cópia da variável *status* foi criada a classe `GenContext`, responsável por gerenciar as variáveis *status* temporários do `rung` além de variáveis temporárias auxiliares e rótulos das instruções.

O código correspondente às linhas de 8 a 15 é responsável por efetuar a cópia da variável *status* do *rung* antes de gerar código para as instruções em paralelo. Na linha 18 é possível observar a geração do código intermediário da instrução atual. A linha 20 contempla a inserção do código gerado na lista de quádruplas que posteriormente representará o programa fonte Ladder. Na linha 23 o método `getCountReferenceUp` da classe `Rung` verifica se a instrução atual está localizada após um paralelo de instruções. Caso esteja, é possível verificar na linha 31, que de forma recursiva é gerado código das instruções em paralelo. O método `genStatusCheck` da linha 34 é responsável por gerar o código que verifica o *status* das variáveis temporários após as instruções em paralelo.

O Quadro 16 apresenta o método `generate` responsável por gerar o código intermediário a partir do programa fonte Ladder.

Quadro 16 – Implementação do método `generate` da classe `IRGenerator`

```

1 public static synchronized IR generate(LadderProgram ladderProgram) {
2     if(ladderProgram == null){
3         throw new IllegalArgumentException("ladderProgram cannot be null");
4     }
5     program = ladderProgram;
6     symbolTable = new SymbolTable();
7     quadruples = new ArrayList<Quadruple>();
8     genContext = new GenContext(symbolTable);
9
10    Rungs rungs = program.getRungs();
11    for (Rung rung : rungs) {
12        genContext.setScope(String.format("Rung%03d", rung.getNumber()));
14        genContext.clearStatus();
15        genInstruction(rung, new ArrayList<LadderInstruction>(),
16            rung.getFirst());
17    }
18    genInit(rungs);
19    genMain();
20    genBegin();
21    for (Rung rung : rungs) {
22        Symbol rungLabel = symbolTable.addFunc(String.format("Rung%03d",
23            rung.getNumber()), Void.class);
24        Symbol zero = symbolTable.addIntConst("0");
25        quadruples.add( Quadruple.createCall( rungLabel , zero ) );
26    }
27    genEnd();
28    return new IR(quadruples, symbolTable);
29 }

```

Conforme pode ser visto nas linhas de 11 a 17, é gerado código para cada *rung* do programa através do método `genInstruction` na linha 15. Após a geração de código dos *rungs* o método `genInit` da linha 18 gera o código intermediário de inicialização das instruções. No caso apenas as instruções de temporização e contagem geram código de inicialização. Na linha 19 o método `genMain` é responsável por gerar o código *main* da aplicação. Esse código é responsável por inicializar o sistema, inicializar o contexto da

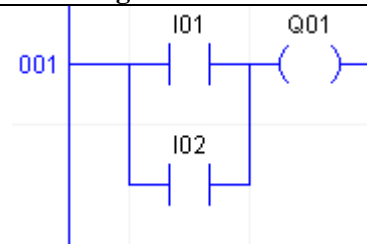
aplicação. Na linha 20 o método `genBegin` é responsável por iniciar o laço de repetição do programa e gerar o código que executa as chamadas de sistema para leitura das interfaces de entrada e obter o tempo do sistema. Nas linhas 21 a 26 pode ser visto a geração do código que efetua a chamada para os *rungs* já criados. Na linha 27 o método `genEnd` é responsável por gerar o código que executa a chamada de sistema para escrita na interface de saída e finaliza o laço de repetição do programa.

#### 3.3.1.1.4 Tradutor de código

O tradutor de código é a última fase do processo de compilação do programa fonte Ladder. É responsável por traduzir o código intermediário gerado para código em linguagem C, tendo o Linux como sistema operacional alvo.

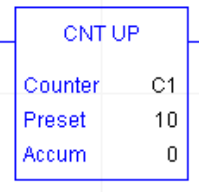
A tradução ocorre através do método `generate` da classe `LinuxCodeGenerator`, que recebe o projeto Ladder, traduz o código intermediário para código em linguagem C e retorna uma representação dos arquivos de código traduzidos. O Quadro 17 demonstra um exemplo de um programa Ladder, seu respectivo código intermediário e a tradução para uma função em código C, que representa o *rung*.

Quadro 17 – Exemplo de programa Ladder vs. código intermediário vs. código C

Programa Ladder	Código intermediário	Função em código C
	<pre> 1 Rung001: 2   _S0 = 1 3   _S1 = _S0 4   if I01 goto L1 5   _S0 = 0 6 L1: 7   if I02 goto L2 8   _S1 = 0 9 L2: 10  ifFalse _S1 goto L3 11  _S0 = 1 12 L3: 13  Q01 = _S0 14  return </pre>	<pre> 1 void Rung001(void) { 2   int _S0; 3   int _S1; 4   _S0 = 1; 5   _S1 = _S0; 6   if(!I01){ 7     _S0 = 0; 8   } 9   if(!I02){ 10    _S1 = 0; 11  } 12  if(_S1){ 13    _S0 = 1; 14  } 15  Q01 = _S0; 16 17 } </pre>

O método `addRung` da classe `LinuxCodeGenerator`, é responsável por traduzir os *rungs* representados em código intermediário para funções em linguagem C. O algoritmo de tradução de um *rung* requer que o código intermediário esteja implementado de acordo com o especificado na seção 3.2.1.5. A descrição do algoritmo é feita com base no Quadro 18, que representa a instrução Ladder `Count Up`, seu respectivo código intermediário e o código C traduzido.

Quadro 18 – Código gerado e traduzido da instrução Count Up

Instrução Count Up	Código intermediário	Código C
	<pre> 1  if !_S0 goto L1 2  C1:CC = 0 3  goto L2 4  L1: 5  if C1:CC goto L2 6  C1:AC = C1:AC + 1 7  C1:CC = 1 8  L2: 9  if C1:AC &lt; C1:PRE goto L3 10 C1:DN = 1 11 goto L4 12 L3: 13 C1:DN = 0 14 L4: 15  _S0 = C1:DN </pre>	<pre> 17 if(!_S0){ 18   C1.CC = 0; 19 }else{ 20   if(!C1.CC){ 21     C1.AC = C1.AC + 1; 22     C1.CC = 1; 23   } 24 } 25 if(C1.AC &gt;= C1.PRE){ 26   C1.DN = 1; 27 }else{ 28   C1.DN = 0; 29 } 30 _S0 = C1.DN; </pre>

O algoritmo traduz quadrupla a quadrupla de cima para baixo, ou seja, linha a linha da representação em texto do código intermediário. Com exceção aos desvios incondicionais (linhas 3 e 11) e os rótulos (linhas 4, 8, 12 e 14), todas as outras instruções são traduzidas de intermediário para C através de um código correspondente, ou seja, possuem um respectivo código C. A tradução dos desvios incondicionais e os rótulos determinam quando termina o teste condicional `if` em C e quando é gerado o comando `else`. O algoritmo utiliza uma estrutura de dados tipo lista que contém os rótulos das instruções de desvio que ainda não foram alcançados pela tradução. O Quadro 19 demonstra as ações do algoritmo conforme o tradutor identifica um operador.

Quadro 19 – Ações do algoritmo de tradução

Operador identificado	Ações
atribuição	Gera o código C respectivo à quadrupla.
chamada de rotina	Gera o código C respectivo à quadrupla.
desvio condicional ( <code>if</code> )	Inserir o rótulo do desvio identificado na lista.
	Gera o código C respectivo à quadrupla.
rótulo	Para cada ocorrência do rótulo identificado na lista cria um comando de fechamento de bloco <code>if ( )</code> .
	Remove todas as ocorrências do rótulo na lista.
	Remove o último rótulo inserido da lista.
desvio incondicional ( <code>goto</code> )	Gera o comando <code>else</code> .
	Adiciona na lista o rótulo identificado.

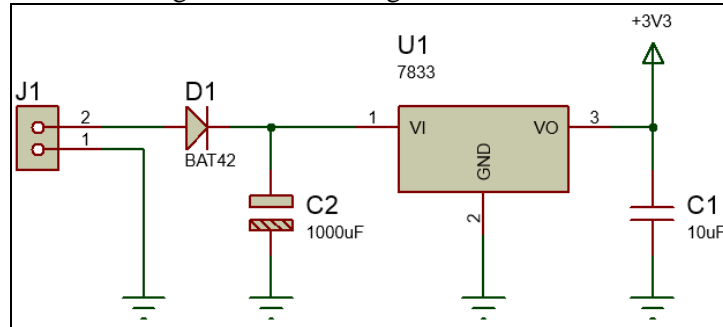
Após a tradução do código é gerado de forma estática os arquivos *Makefile* e *shell script* Linux, necessários para compilar e criar as interfaces de E/S respectivamente. O Apêndice C demonstra os arquivos gerados.



### 3.3.1.2 Montagem do hardware

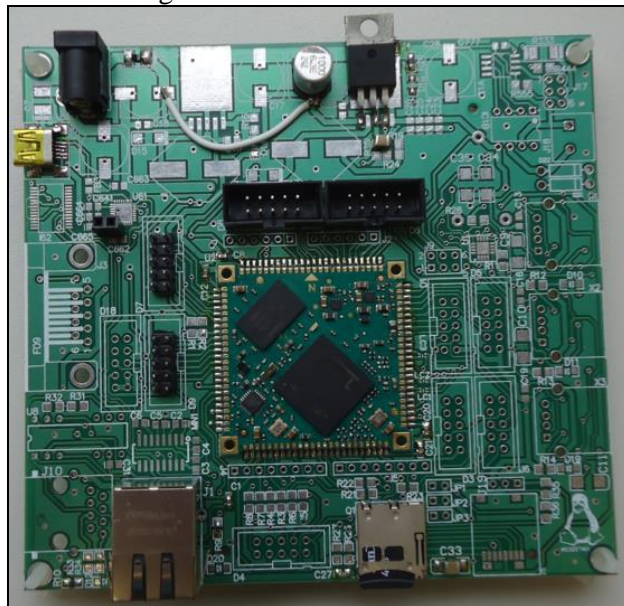
A montagem da placa mãe foi realizada baseada no diagrama da seção 3.2.2.2, tendo a parte que regula a tensão de entrada modificada para reduzir custos. A Figura 23 demonstra o diagrama do novo regulador de tensão montado.

Figura 23 – Novo regulador de tensão



O conector J1 permite a conexão com uma fonte de alimentação de +5VDC protegidos contra inversão de polaridade pelo D1. O U1 regula a tensão de entrada de +5VDC para +3.3VDC, necessário para a placa mãe e a placa de testes. A Figura 24 ilustra a placa mãe montada.

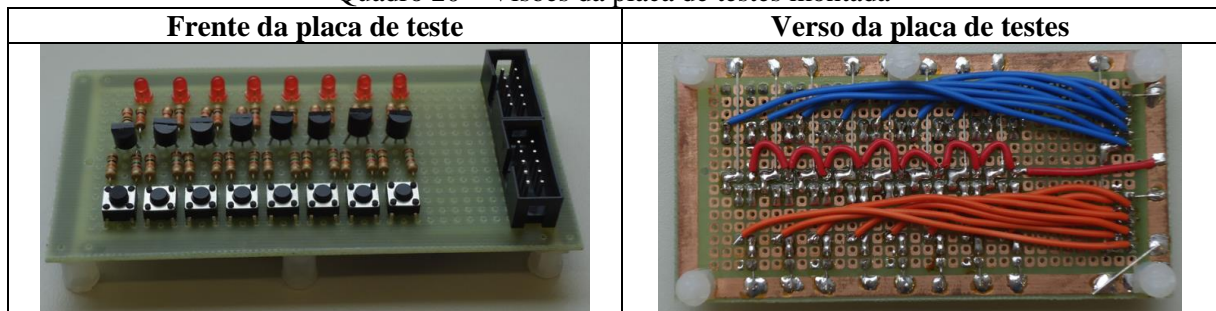
Figura 24 – Placa mãe montada



A placa mãe foi montada apenas com os componentes necessários. É composta pelo módulo Aria G25, o regulador de tensão conforme Figura 23 e conectores.

A montagem da placa de testes foi realizada de acordo com o diagrama da seção 3.2.2.2. O Quadro 20 demonstra as imagens de frente e verso da placa de testes montada.

Quadro 20 – Visões da placa de testes montada



A placa de testes foi confeccionada com uma placa de circuito universal que permite realizar as conexões elétricas através de condutores (fios) soldados. As oito Chaves Táctil montadas são as interfaces de entrada (sensores) e estão configuradas como *pull-up*, ou seja, tem-se nível lógico alto (1) quando pressionadas. Os oito *LEDs* montados são as interfaces de saída (atuadores), acendem quando o nível lógico da respectiva saída for alto.

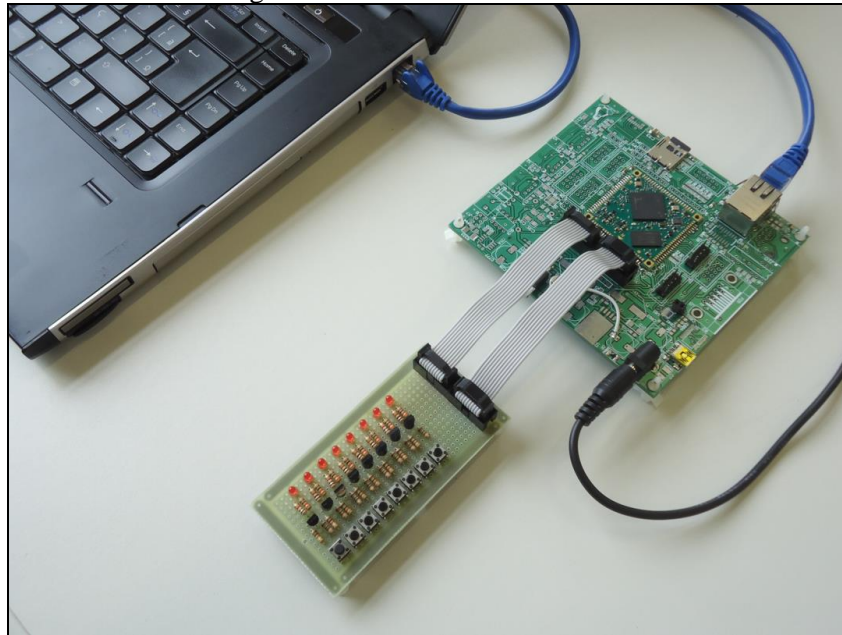
A conexão entre a placa mãe e a placa de testes é feita através de dois cabos. Um para a interface de saída e outro para a interface de entrada. A pinagem dos conectores das placas respeita a disposição original da placa Terra ACMESYSTEMS (2014) fornecida pelo fabricante como compatível ao módulo Aria G25 (ver seção 2.6), o que torna o protótipo compatível com as placas originais.

### 3.3.2 Operacionalidade da implementação

Esta seção apresenta um manual simplificado de uso do protótipo desenvolvido. Estas instruções permitem que o usuário utilize os recursos disponíveis para construir um programa Ladder e executá-lo na placa de testes. Para ilustrar as funcionalidades, foi proposto a criação de um programa Ladder que implementa a lógica de um interruptor elétrico.

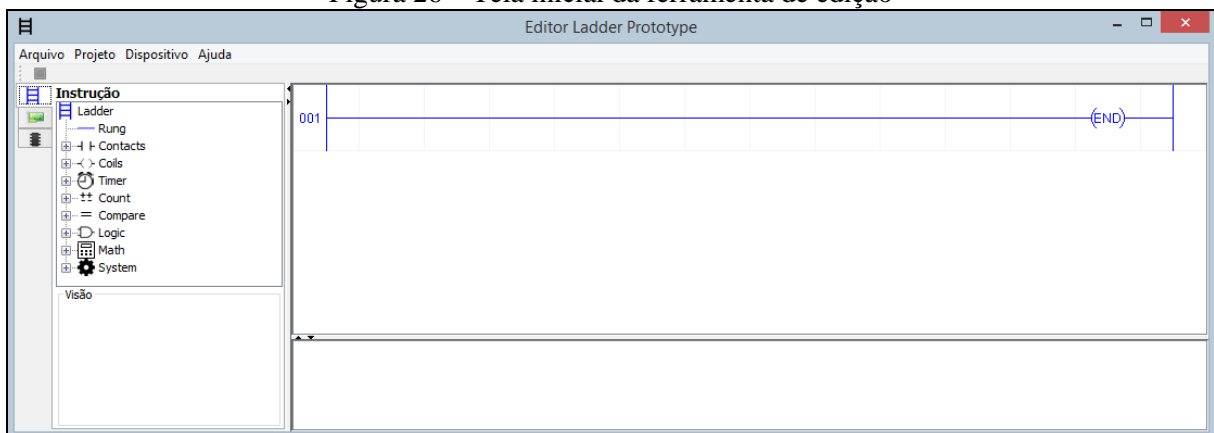
Inicialmente deve-se efetuar a conexão da placa mãe com a placa de teste e a um computador. A placa mãe contém gravado no microSD o sistema operacional Linux e os respectivos binários. Em seguida é feita a ligação da fonte de alimentação da placa mãe. A Figura 25 ilustra as conexões estabelecidas entre os hardwares.

Figura 25 – Hardwares conectados



Após o processo de *boot* do Linux o hardware está pronto para receber o programa Ladder. O usuário deve executar a ferramenta de edição de programas Ladder desenvolvida. Ao acessá-la o usuário irá visualizar uma tela conforme a Figura 26.

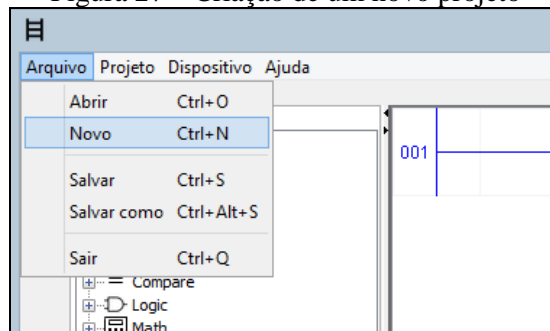
Figura 26 – Tela inicial da ferramenta de edição



A tela inicial é dividida em menus da aplicação na parte superior, três abas na lateral esquerda (Instrução, Dispositivo e Memória), editor Ladder na parte superior direita e caixa de mensagens na parte inferior direita. Abaixo do menu a barra de ferramentas possui um botão *stop* que encerra a aplicação em execução no hardware.

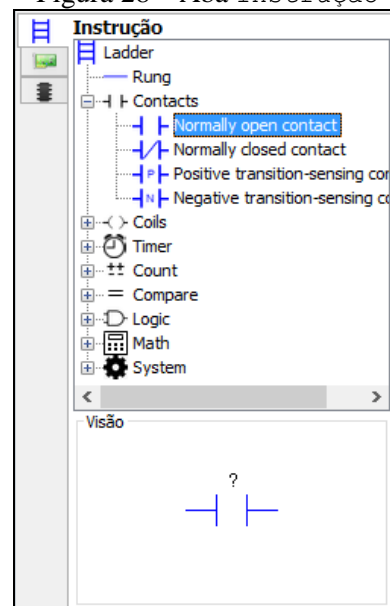
O primeiro passo para criar um programa Ladder consiste em criar um projeto. Para isso o usuário deve clicar na opção do menu *Novo*, para que a ferramenta crie uma nova estrutura que representa o programa como pode ser visto na Figura 27.

Figura 27 – Criação de um novo projeto



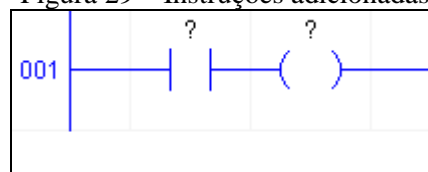
O segundo passo consiste em inserir as instruções Ladder. A aba *Instrução* contém as instruções Ladder organizadas em árvore conforme o tipo (categoria) de instrução, além de uma pré-visualização da instrução na parte inferior da aba (Figura 28). O editor permite adicionar as instruções no programa através do recurso de arrastar e soltar, para isso basta selecionar uma instrução, arrastá-la até uma posição livre no *rung* e soltar a instrução.

Figura 28 – Aba Instrução



Para implementar o caso de uso proposto deve-se adicionar respectivamente as instruções *Normally open contact* e *Coil*. Após a inserção das instruções o programa ficará conforme ilustrado na Figura 29.

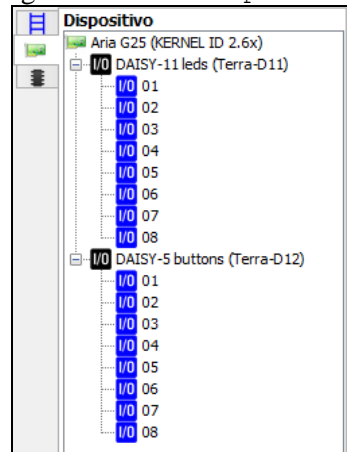
Figura 29 – Instruções adicionadas



O terceiro passo consiste em associar as variáveis as instruções. A aba *Dispositivo* contém as variáveis associativas que representam interfaces E/S do hardware, enumeradas de

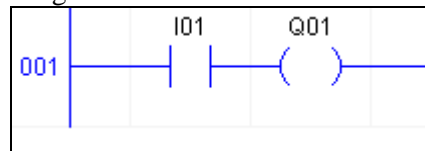
1 a 8, conforme a Figura 30. As variáveis podem ser associadas com as instruções através do mesmo recurso de arrastar e soltar.

Figura 30 – Aba Dispositivo



Deve-se associar a primeira instrução a variável 01 do item DAISY-11 leds (nomenclatura da interface de saída compatível com placas do fabricante ACMESYSTEMS) que representa uma saída. Em seguida deve-se associar a segunda instrução a variável 02 do item DAISY-5 buttons (nomenclatura da interface de entrada compatível com placas do fabricante ACMESYSTEMS) que representa uma entrada. Após a associação o programa ficará conforme ilustrado na Figura 31.

Figura 31 – Memórias associadas



Também é possível associar uma variável retentiva (ver seção 3.2.1.2.2) ou uma variável de instrução através da aba Memória, conforme pode ser visto no Quadro 21.

Quadro 21 – Aba Memória

Variável retentiva tipo inteiro	Variável de instrução

O quarto passo consiste em configurar a conexão com o hardware. Para isso o usuário deve clicar na opção do menu `Conexão SSH`, para que a ferramenta exiba a tela de configuração da conexão como pode ser visto na Figura 32 e Figura 33.

Figura 32 – Definindo conexão com o hardware

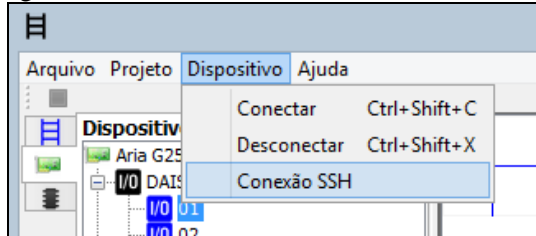
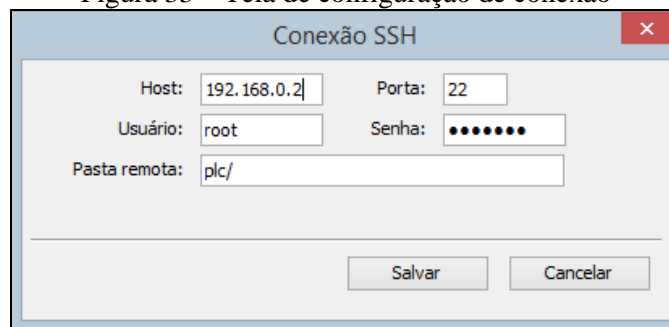


Figura 33 – Tela de configuração de conexão



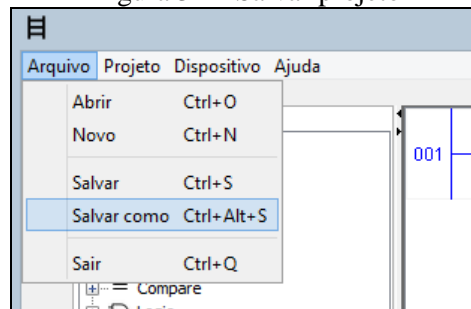
As configurações de conexão do hardware são as seguintes:

- a) *host*: 192.168.0.2;
- b) *porta*: 22;
- c) *usuário*: *root*;
- d) *senha*: *ariag25*;
- e) *pasta remota*: *plc/*.

O computador deve estar na mesma rede da placa mãe. Após a configuração ser efetuada o usuário deve clicar na opção salvar.

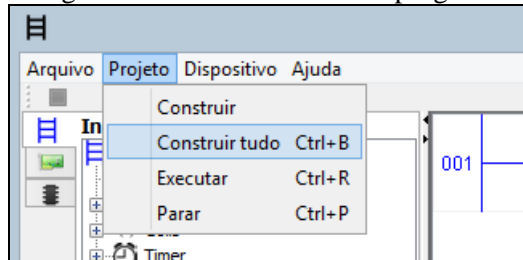
O quinto passo consiste em salvar o projeto. Para isso o usuário deve clicar na opção do menu `Salvar como`, para que a ferramenta exiba a caixa de diálogo que permite o usuário escolher o destino do arquivo, como pode se visto na Figura 34.

Figura 34 – Salvar projeto



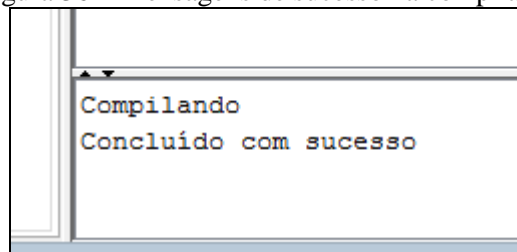
O sexto passo consiste em compilar o programa Ladder. Para isso o usuário deve clicar na opção do menu `Construir tudo`, para que a ferramenta gere o código intermediário e traduza para código em linguagem C, como pode ser visto na Figura 35.

Figura 35 – Gerar e traduzir o programa



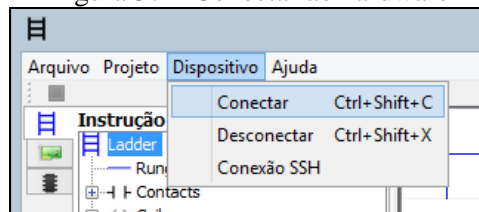
A Figura 36 exibe as mensagens referentes ao processo de compilação que serão exibidas na caixa de mensagens.

Figura 36 – Mensagens de sucesso na compilação



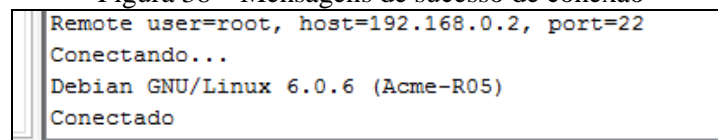
O sétimo passo consiste em conectar com o hardware. Para isso o usuário deve clicar na opção do menu `Conectar`, para que a ferramenta inicie a conexão com o hardware, como pode ser visto na Figura 37.

Figura 37 – Conectar ao hardware



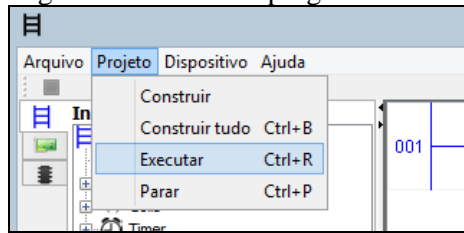
A Figura 38 exibe as mensagens referentes ao processo de conexão com o hardware que serão exibidas na caixa de mensagens.

Figura 38 – Mensagens de sucesso de conexão



O oitavo passo consiste em executar o código gerado. Para isso o usuário deve clicar na opção do menu `Executar`, para que a ferramenta possa executar os comandos para transferir, compilar e executar o programa Ladder, como pode ser visto na Figura 39.

Figura 39 – Executar programa Ladder



A Figura 40 exibe as mensagens referentes ao processo de transferência, compilação e execução, que serão exibidas na caixa de mensagens.

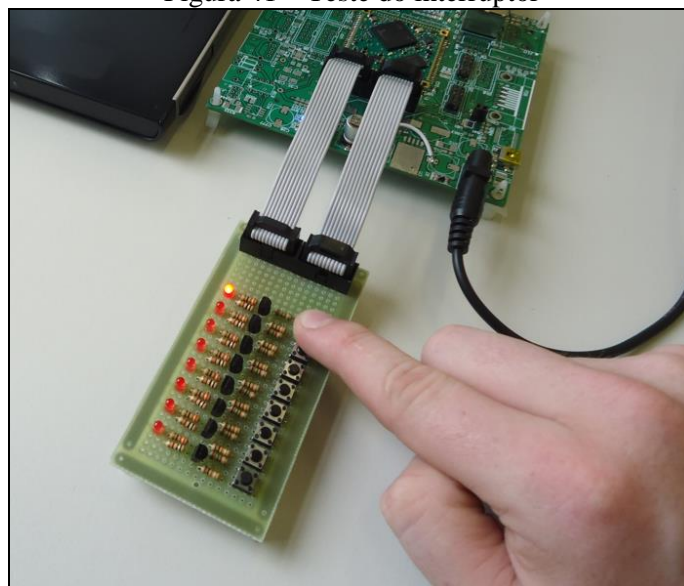
Figura 40 – Mensagens referentes à execução do programa Ladder

```

Stop plc exit-status: 0
Mkdir exit-status: 0
Copy plc.c exit-status: 0
Copy Makefile exit-status: 0
Copy plc.sh exit-status: 0
Make clean
rm -rf *.o
exit-status: 0
Make mrproper
rm -rf *.o
rm -rf plc
exit-status: 0
Make
cc -o plc.o -c plc.c
cc -o plc plc.o
exit-status: 0
Run scriptexit-status: 0
Run...
  
```

Por fim o usuário pode testar a programa Ladder através da placa de testes. Enquanto o usuário pressionar o botão a *LED* permanecerá aceso, como pode ser visto na Figura 41.

Figura 41 – Teste do interruptor





### 3.4 RESULTADOS E DISCUSSÃO

O presente trabalho teve como objetivo principal disponibilizar uma ferramenta que permita um hardware com Linux Embarcado assumir funcionalidades de um CLP. Para tal foi desenvolvido um editor gráfico da linguagem de diagramas Ladder através da API Java2D. O editor permite traduzir uma representação intermediária gerada a partir do programa fonte Ladder para código em linguagem C, suportado pelo hardware.

O desenvolvimento do editor gráfico demonstrou ser um desafio em relação à funcionalidade de adicionar instruções Ladder em paralelo e modificar esta estrutura. A linguagem Ladder não possui em sua especificação restrições sintáticas claramente definidas, o que permite as mais variadas combinações de instruções em paralelo. O escopo inicial do trabalho não restringia qualquer aspecto quanto ao paralelismo de instruções, porém, na terceira quinzena do desenvolvimento, esta funcionalidade foi identificada como um complicador em relação ao fator tempo de desenvolvimento. Como forma de reduzir a complexidade do editor gráfico em relação às instruções em paralelo, foi limitado este recurso a apenas o paralelo de instruções formadas por um bloco no editor (ver seção 3.2.1.4) e não se permitiu adicionar instruções em série a uma instrução paralela.

Os investimentos necessários para aquisição e montagem do módulo Aria G25, placa mãe e placa de testes, foram aproximadamente de R\$ 250,00. Inicialmente a placa de testes era composta apenas por duas interfaces de entrada e duas de saída, pois o intuito era apenas testar as interfaces. A partir do sucesso obtido, a placa de testes foi modificada suportando oito interfaces de entrada e oito de saída, permitindo a criação de programas mais complexos e uma melhor análise em relação à performance do programa Ladder.

As seções a seguir relatam o teste de desempenho do programa fonte Ladder no hardware, o teste de funcionalidade e o comparativo com os trabalhos correlatos. Os testes foram realizados com base no hardware especificado nas seções 2.6 e 3.2.2.2. O estado inicial do hardware em relação aos processos em execução no Linux e a quantidade de memória disponível podem ser vistos no Apêndice D.

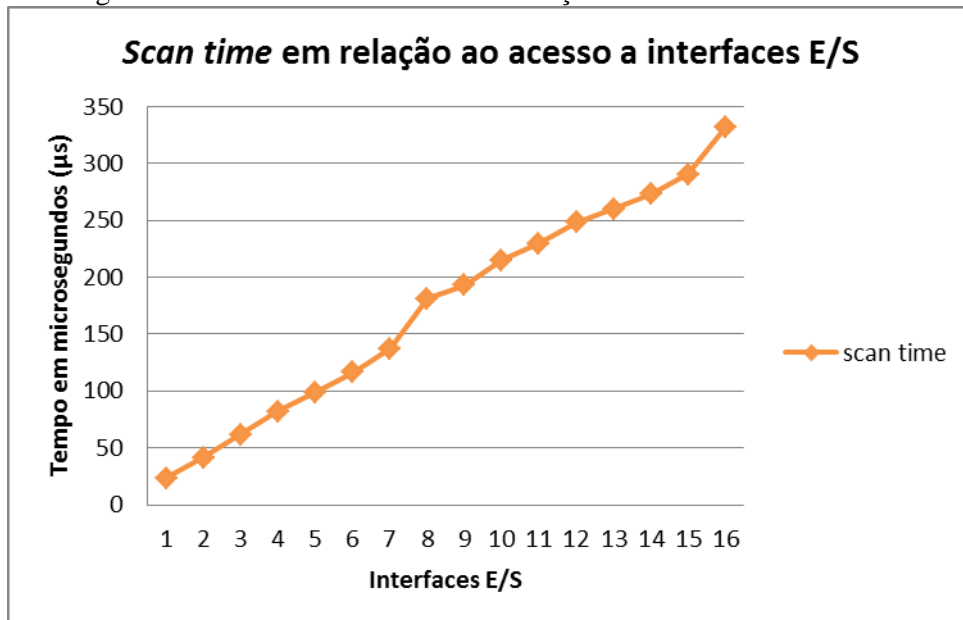
#### 3.4.1 Testes de desempenho do programa fonte Ladder no hardware

Os testes de desempenho do programa fonte Ladder no hardware foram realizados a partir do estado inicial do hardware. Consistem na verificação de três aspectos, a influência do acesso às interfaces E/S na performance do programa, tempo de execução das instruções Ladder e consumo de memória.

Os critérios utilizados para testar o desempenho foram o *scan time*, obtido através da instrução `System Scan Time` do editor Ladder e o comando `pmap` do Linux, que permite visualizar o mapa de memória de um processo de forma precisa, segundo Cardoso (2008). Assumiu-se uma amostragem de dez valores de *scan time* para se calcular a média de tempo.

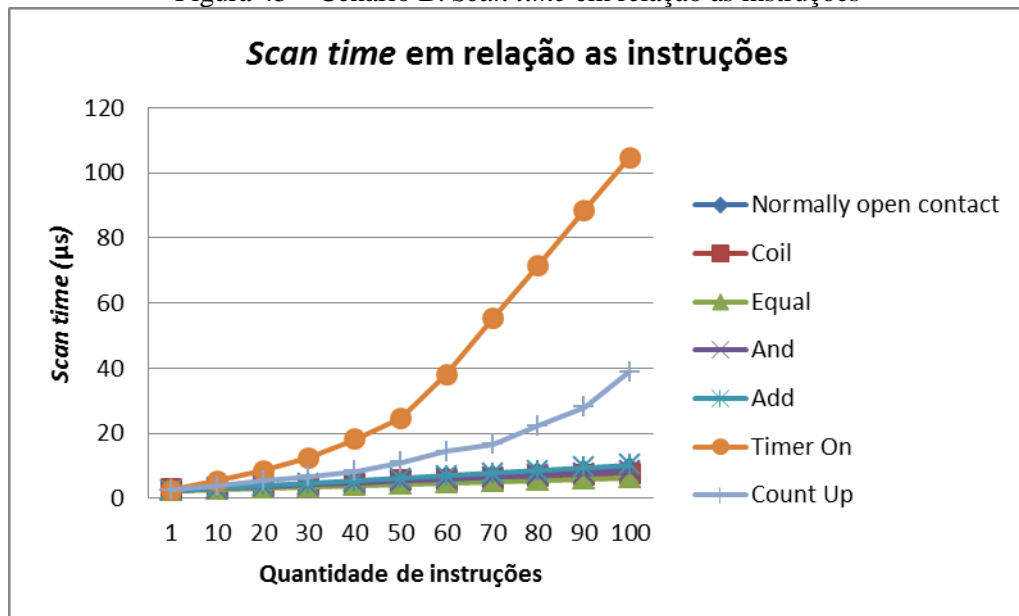
O primeiro teste refere-se ao quanto o acesso a uma interface E/S impacta no *scan time* do programa. Foi proposto um Cenário A cujo programa Ladder possui uma instrução associada a uma interface E/S por *rung*. Variando a quantidade de *rungs* com instruções de 1 a 16, tem-se o tempo de acesso em relação à quantidade de interfaces acessadas. A Figura 42 apresenta um gráfico referente ao cenário descrito, sendo que as oito primeiras interfaces de entrada foram associadas à instrução do tipo `Normally open contact` e as interfaces de saída restantes foram associadas à instrução do tipo `Coil`.

Figura 42 – Cenário A. *Scan time* em relação ao acesso a interfaces E/S

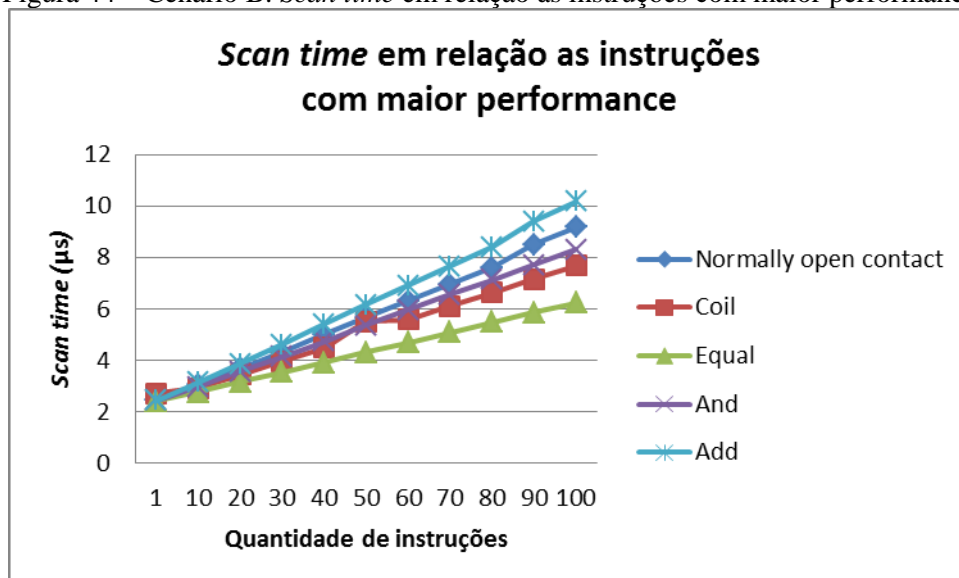


É possível perceber uma linearidade correspondente à quantidade de interfaces E/S acessadas, o que é proporcionalmente inverso a performance do programa. Quanto mais interfaces E/S fizerem parte do programa mais lento será o *scan time*.

O segundo teste refere-se ao tempo de execução das instruções Ladder. Foi proposto um Cenário B cujo programa Ladder varia a quantidade de instruções de 1 a 100, sendo que contém apenas uma instrução por *rung* e cada execução possui apenas um tipo de instrução. Foram submetidos ao teste uma instrução de cada categoria conforme ilustrado na Figura 43, associadas a variáveis retentivas do tipo inteiro para que não haja influência das interfaces E/S. As instruções do tipo temporizador e contador foram configuradas para permanecer contando durante o teste.

Figura 43 – Cenário B. *Scan time* em relação às instruções

É possível perceber que as instruções do tipo temporizador e contador são as que mais impactam na performance do programa de forma negativa, tendo destaque a instrução do tipo temporizador como a menos performática. A Figura 44 ilustra apenas as instruções que possuem maior performance.

Figura 44 – Cenário B. *Scan time* em relação às instruções com maior performance

As instruções estão representando sua categoria porque em suma, as outras instruções da categoria possuem uma implementação semelhante. As instruções *And* e *Add* merecem destaque, pois resultam em um código que apenas os operadores são diferentes, mas o tempo de execução das instruções é diferente. Isso se dá pelo fato das instruções lógicas possuírem uma operação de conversão de tipo (*cast*), implementada para truncar (arredondar) uma variável do tipo ponto flutuante, como pode ser visto no Quadro 22.

Quadro 22 – Exemplo de código referente às instruções And e Add

Instrução	Código em linguagem C
And	<code>x = ((int)y &amp; (int)z);</code>
Add	<code>x = y + z;</code>

Conforme já ilustrado na Figura 44 a instrução `Equal` da categoria de instruções de comparação mostrou ser a mais performática de todas as instruções testadas.

O terceiro teste refere-se ao consumo de memória. Através do comando `pmap -p` (PID do processo `plc`) executado no terminal do hardware é possível verificar o quanto de memória o processo do programa Ladder está consumindo. A Figura 45 ilustra o resultado do comando `pmap`, a quantidade de memória ocupada pelo processo do programa Ladder (`plc`) pode ser vista no final da imagem com o rótulo `writable/private:` totalizando 164KB. O comando foi executado com um programa sem instruções Ladder e um `rung`.

Figura 45 – Resultado do comando `pmap`

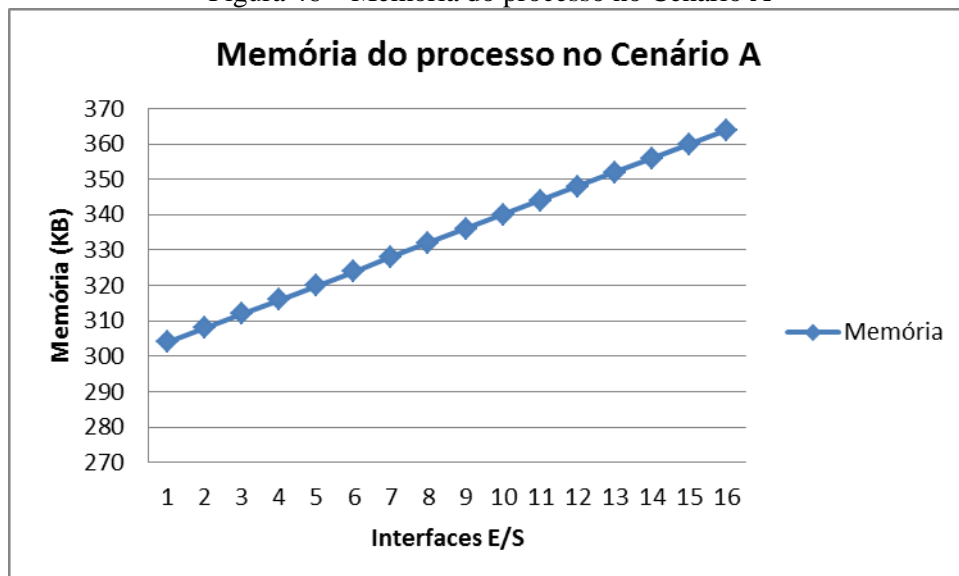
```

Address  Kbytes Mode  Offset                Device  Mapping
00008000    4 r-x--  0000000000000000    0b3:00002 plc
00010000    4 rwx--  0000000000000000    0b3:00002 plc
40030000   116 r-x--  0000000000000000    0b3:00002 ld-2.11.3.so
4004d000    4 rwx--  0000000000000000    000:00000 [ anon ]
40054000    4 r-x--  000000000001c000    0b3:00002 ld-2.11.3.so
40055000    4 rwx--  0000000000018000    0b3:00002 ld-2.11.3.so
40088000    4 rwx--  0000000000000000    000:00000 [ anon ]
400eb000   1164 r-x--  0000000000000000    0b3:00002 libc-2.11.3.so
4020e000    32 ----  0000000000123000    0b3:00002 libc-2.11.3.so
40216000    8 r-x--  0000000000123000    0b3:00002 libc-2.11.3.so
40218000    4 rwx--  0000000000125000    0b3:00002 libc-2.11.3.so
40219000   12 rwx--  0000000000000000    000:00000 [ anon ]
be866000   132 rw---  0000000000000000    000:00000 [ stack ]
ffff0000    4 r-x--  0000000000000000    000:00000 [ anon ]
mapped: 1496K  writable/private: 164K  shared: 0K

```

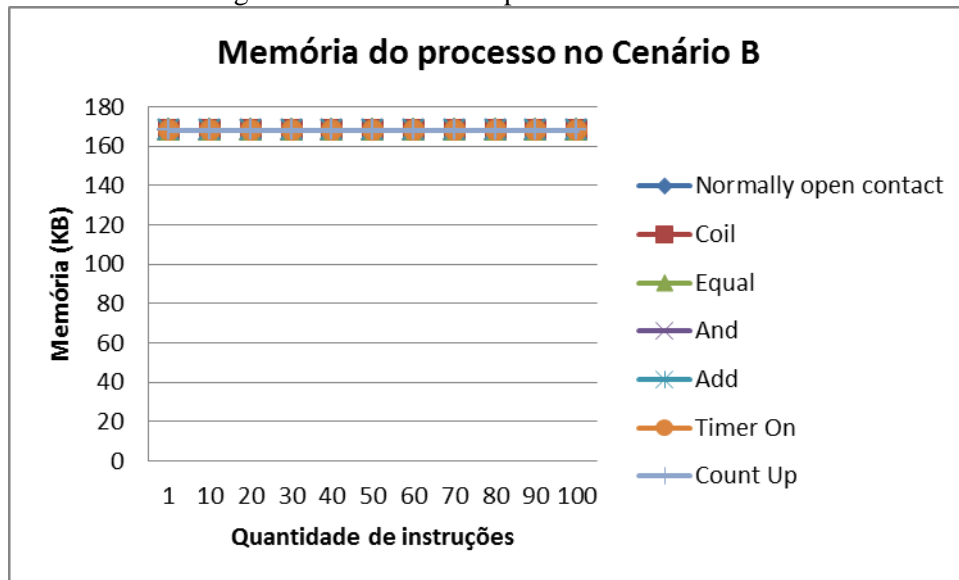
O teste de memória consumida foi realizado nos cenários A e B. No teste do Cenário A, conforme ilustrado na Figura 46, é possível verificar um padrão de consumo de memória a cada interface E/S. Esse valor é exatamente 4KB, obtidos através do comando `pmap`. O que significa que quanto mais interfaces E/S o programa Ladder utilizar mais memória será necessário ao processo.

Figura 46 – Memória do processo no Cenário A



No teste do Cenário B, conforme ilustrado na Figura 47, é possível observar que a memória consumida pelo processo de um programa Ladder com 1 ou 100 instruções, é a mesma. Esse valor é exatamente 168KB, 4KB a mais em relação a um programa sem instruções, porém estes 4KB a mais são provenientes da instrução `System Scan Time`.

Figura 47 – Memória do processo no Cenário B



### 3.4.2 Teste de funcionalidade

A fim de efetuar uma avaliação das funcionalidades do protótipo, foi elaborado um formulário de avaliação com base na norma ISO/IEC 25010:2011 segundo WAZLAWICK (2013, p. 232), contemplando adequação funcional e eficiência de desempenho, conforme o modelo de qualidade. O formulário proposto utiliza uma escala *Likert*<sup>2</sup> com cinco alternativas que variam de 1 (discordância total) a 5 (concordância total), para cada afirmação. O formulário de avaliação pode ser visto no Apêndice E.

No que diz respeito à adequação funcional foram elaboradas três afirmações sobre condições específicas:

- completude funcional: o editor permite criar um programa de acordo com as características da linguagem Ladder;
- corretude funcional: a execução do código gerado corresponde ao programa Ladder criado;
- funcionalidade apropriada: a utilização do editor facilita a criação da lógica Ladder.

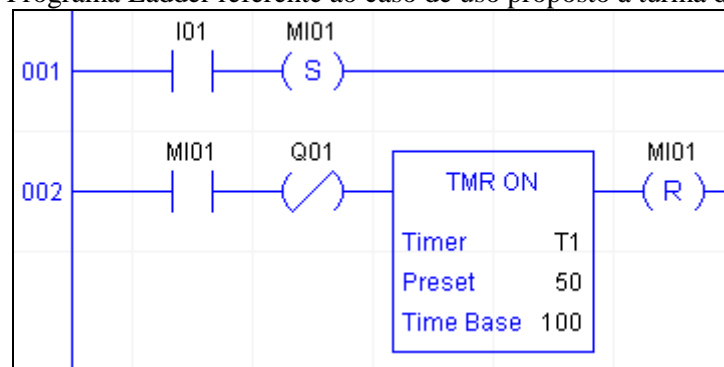
<sup>2</sup> “A escala *Lickert* mede atitudes e comportamentos utilizando opções de respostas que variam de um extremo ao outro (por exemplo: de nada provável para extremamente provável). Ao contrário de uma simples questão “sim ou não”, uma escala *Lickert* permite descobrir níveis de opinião” (SURVEYMONKEY, 2014).

Em relação à eficiência de desempenho foi elaborada uma afirmação sobre a utilização de recursos: o código gerado é executado com alta performance.

Além das afirmações o formulário permite que sejam destacados de forma opcional, pontos fortes e fracos observados.

Foi efetuada uma atividade de avaliação do protótipo com uma turma de Automação do décimo semestre do curso de Engenharia Elétrica da FURB. Os alunos foram convidados a utilizar o protótipo para resolver um caso de uso e posteriormente avaliar a ferramenta. O caso de uso proposto consiste em uma fechadura eletromagnética com uma botoeira. A fechadura permanece energizada até que seja pressionada a botoeira, que desliga a fechadura por cinco segundos. A fechadura é representada por uma interface de saída, um *LED* da placa de testes e a botoeira é representada por uma interface de entrada, um botão da placa de testes. A Figura 48 ilustra o programa Ladder respectivo ao caso de uso proposto.

Figura 48 – Programa Ladder referente ao caso de uso proposto à turma de Automação



Após o uso da ferramenta, foram obtidas avaliações de 13 alunos de acordo com o questionário estabelecido. O Quadro 23 apresenta o resultado da avaliação da ferramenta realizada pelos alunos.

Quadro 23 – Resultado da avaliação da ferramenta

	Afirmações	Escala				
		1	2	3	4	5
1	O editor permite criar um programa de acordo com as características da linguagem Ladder.				84,6%	15,4%
2	A execução do código gerado corresponde ao programa Ladder criado.				30,77%	69,23%
3	A utilização do editor facilita a criação da lógica Ladder.			15,38%	23,08%	61,54%
4	O código gerado é executado com alta performance.			15,38%	61,54%	23,08%

Em relação ao editor Ladder, 84,6% dos alunos responderam que concordam parcialmente e 15,4% responderam que concordam totalmente. Quanto à execução do programa corresponder ao programa Ladder, 69,23% dos alunos consideram totalmente correspondente e 30,77% consideram que o programa corresponde parcialmente. Tratando-se da facilidade que o editor permite para criar a lógica Ladder, 61,54% dos alunos concordam

totalmente, 23,08% afirmam que concordam parcialmente e 15,38% afirmam ser indiferentes quanto a facilidade do editor. Apenas 23,08% dos alunos responderam que o código gerado é executado com alta performance, 61,54% dos alunos concordam parcialmente com essa afirmação e 15,38% são indiferentes a este aspecto.

Em relação aos pontos fortes e fracos observados, a descrição está disponibilizada no Apêndice F. Um ponto fraco observado pela turma foi o fato do editor Ladder não permitir adicionar um novo *rung* abaixo do *rung* atual, somente acima. Segundo os alunos o programa é construído de forma incremental e facilitaria se esta funcionalidade estivesse disponível. Em contra ponto, uma funcionalidade que não foi especificada como objetivo, tão pouco como requisito e foi implementada para agilizar o teste do programa Ladder no hardware, foi muito elogiada pela turma. É a questão da transferência, compilação e execução do programa Ladder através do protocolo SSH entre editor e hardware. Outro ponto positivo que vale apenas ser comentado é a facilidade de utilização do editor gráfico Ladder, porém os elogios referentes a este aspecto não são tão evidenciados conforme a terceira afirmação do formulário de avaliação.

Como forma de obter mais resultados a respeito da ferramenta, do ponto de vista de desenvolvedores de tecnologia embarcada, que possuem o mesmo módulo Linux, foi criado um site sobre o presente trabalho, ao qual disponibilizava a ferramenta, *links* para avaliação, vídeos de tutoriais e um manual. O questionário de avaliação e o site foram traduzidos para a língua inglesa com o intuito de que possam ser compreendidos por pessoas que não são brasileiras. Pelo fato do formulário ser específico a um público, foram adicionados alguns elementos, como uma pergunta de qual periférico deveria ser adicionado no futuro, além do país de origem (opcional), nome (opcional) e *e-mail* (opcional).

Após o período de aproximadamente três semanas que permaneceu disponível o formulário de avaliação da ferramenta, as respostas obtidas foram inconclusivas devido à quantidade de apenas três respostas. A ferramenta foi divulgada em mais de cinco *web* fóruns internacionais, porém sem a repercussão desejada.

### 3.4.3 Comparativo com os correlatos

O Quadro 24 apresenta um comparativo sobre as principais características entre o presente trabalho e os trabalhos correlatos. O presente trabalho é denominado *Ladder Editor Prototype*.

Quadro 24 – Comparativo entre o trabalho proposto e os correlatos

Características	Ladder Editor Prototype	PL $\mu$ X	LDmicro	CUBLOC
Linux embarcado	X	X	-	-
Transferência e execução através do protocolo SSH	X	X	-	-
Simulador	-	X	X	-
Executa código nativo do hardware	X	-	X	-
Editor Ladder gráfico	X	X	-	X
Linguagem de programação alternativa em texto	-	-	-	X
Executa programa Ladder em multitarefas	-	-	-	X

Pode-se observar que a ferramenta disponibilizada não se enquadra em alguns itens. A capacidade de simular um CLP é desejável e recomendada como extensão ao presente trabalho na seção 4.1. A representação da lógica de programa de um CLP através de uma linguagem textual, não é totalmente contra o propósito da ferramenta proposta, porém, uma linguagem gráfica como o Ladder torna-se menos complexa do ponto de vista de utilização do usuário. O uso de multitarefas para executar um programa Ladder resulta em uma característica indesejada, que é a compatibilidade do editor com lógicas Ladder já estabelecidas para resolver algum problema, baseadas no fato de que o programa é executado de forma sequencial.

Em relação aos itens que a ferramenta se enquadra, a execução da lógica Ladder em código nativo do hardware, destaca-se como muito desejável, pelo fato que desta forma o *scan time* pode ser reduzido aumentando a performance do CLP. O Linux embarcado presente no trabalho proposto e no correlato PL $\mu$ X, beneficia ambos pelo protocolo SSH geralmente presente nas distribuições Linux, permitindo uma flexibilidade no quesito transferência do programa fonte para o CLP.



## 4 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de um editor gráfico da linguagem de diagramas Ladder, utilizando a API gráfica Java2D, que gera código em linguagem C a partir de uma representação intermediária do programa fonte e um hardware de testes baseado em um módulo com Linux Embarcado.

A API Java2D é uma tecnologia presente na linguagem de programação Java que provou através da avaliação da turma de Automação, ser possível desenvolver um editor gráfico de diagramas com os recursos da API. Os elogios quanto à interface deixaram claros que um ambiente gráfico com recurso de arrastar e soltar facilita a criação da lógica Ladder.

A limitação da ferramenta quanto à associação em paralelo de instruções Ladder não demonstrou ser um problema nos testes, quanto à lógica que pode ser representada. Porém é necessário considerar lógicas mais complexas baseadas no paralelismo de instruções, estas podem adicionar um grau de dificuldade a mais na criação de um programa Ladder, já que seria necessário reescrever a lógica com mais *rungs*.

Quanto ao gerador de código intermediário a partir do programa fonte Ladder, este possibilitou a tradução para a linguagem C mantendo a semântica do programa Ladder, característica esta que permite futuramente que a ferramenta gere código para outras arquiteturas, plataformas ou linguagens.

O teste de desempenho deixou claro que a quantidade de interfaces E/S que fazem parte da aplicação, influenciam diretamente sobre a performance do hardware. Porém não se pode afirmar que o protótipo não atende neste quesito, pois a performance é relativa a necessidade da aplicação. Automações que necessitam atuar em resposta aos sensores na casa dos milissegundos ou mais, com até oito sensores e oito atuadores, podem ser efetuadas com o protótipo. Automações que necessitam de tempos de resposta menor ou uma quantidade maior de sensores e atuadores, podem ser solucionadas com um hardware com maior performance, o que nesse caso não desqualifica o protótipo, pois o código C é gerado tendo Linux como sistema operacional alvo, o que o torna portátil para outros hardwares que possuem Linux Embarcado.

Por fim, é possível afirmar que o protótipo atingiu os objetivos propostos, embora o público alvo sejam profissionais da área da Automação, fato que dificulta a avaliação do protótipo e a ferramenta necessite atingir um grau de maturidade maior para se tornar um produto final, o protótipo de CLP para Linux Embarcado demonstrou ser uma solução de relação custo-benefício para automações de pequeno e médio porte.

#### 4.1 EXTENSÕES

Durante o desenvolvimento do protótipo foram identificados pontos que podem ser melhorados, como sugestão a extensões pode-se citar:

- a) implementar um simulador de CLP que permita testar a lógica Ladder em software e verificar o estado das variáveis do programa em execução;
- b) permitir incluir uma ou mais instruções em paralelo a outra independente do tamanho do bloco;
- c) portar a aplicação para outras plataformas de hardware com Linux Embarcado;
- d) implementar o *debug* do programa Ladder em execução através da interface SSH, que permita executar o programa passo a passo, definir *break point* e verificar o estado das variáveis em execução;
- e) incluir instruções de desvio incondicional na lógica Ladder para saltos entre *rungs*;
- f) permitir criar blocos funcionais com lógica Ladder, afim de torná-lo uma nova instrução/componente Ladder.

## REFERÊNCIAS

- ACCARDI, Adonis; DODONOV, Eugeni. **Automação residencial**: elementos básicos, arquiteturas, setores, aplicações e protocolos. São Carlos, 2012. Disponível em: <<http://revistatis.dc.ufscar.br/index.php/revista/article/download/27/30>>. Acesso em: 07 nov. 2014.
- ACMESYSTEMS. **Aria G25**: Low cost Linux Embedded SMD module. Ladispoli, 2014. Disponível em: <<http://acme.systems/aria>>. Acesso em: 17 out. 2014.
- AHO, Alfred V. et al. **Compiladores**: princípios, técnicas e ferramentas. 2. ed. São Paulo: Pearson Addison Wesley, 2008.
- ARCHLINUX. **Secure shell**. [S.I.], 2014. Disponível em: <[https://wiki.archlinux.org/index.php/Secure\\_Shell](https://wiki.archlinux.org/index.php/Secure_Shell)>. Acesso em: 08 nov. 2014.
- BASIC4EVER. **Carrier board**: weseetiny. Blumenau, 2014. Disponível em: <<http://www.basic4ever.com/interpretador.html>>. Acesso em 30 out. 2014.
- BISHOP, Robert H. **Mechatronic systems, sensors, and actuators**: fundamentals and modeling. 2. ed. Boca Raton: CRC Press, 2008a.
- \_\_\_\_\_. **Mechatronic system control, logic, and data acquisition**: fundamentals and modeling. 2. ed. Boca Raton: CRC Press, 2008b.
- CARDOSO, Heitor Augusto Murari. **Entendendo o uso de memória no Linux**. [S.I.], 2008. Disponível em: <<http://ha-mc.org/node/20>>. Acesso em: 09 nov. 2014.
- COMFILE TECHNOLOGY. **CUBLOC**. Virginia, 2014a. Disponível em: <<http://www.comfiletech.com/pages/embedded-controller/cubloc.html>>. Acesso em: 15 out. 2014.
- \_\_\_\_\_. **CB220**. Virginia, 2014b. Disponível em: <<http://www.comfiletech.com/cb220/>>. Acesso em: 15 out. 2014.
- \_\_\_\_\_. **User Manual**. Virginia, 2014c. Disponível em: <<http://www.comfiletech.com/content/cubloc/cublocmanual.pdf>>. Acesso em: 15 out. 2014.
- CONINCK, Ivo De. **PLμX . a programmable logic microcontroller on linux**. [S.I.], 2012. Disponível em: <<http://www.dcsite.be/plux.html>>. Acesso em: 15 out. 2014.
- ELEUTÉRIO, Warley A.; HOVADICH, Wagner A. A.; BRAGA, Eduardo Q. Controlador lógico programável utilizando PIC 18F4550. **e-xacta**, Belo Horizonte, v. 4, n. 3, 2011. Disponível em: <<http://revistas.unibh.br/index.php/dcet/article/viewFile/695/392>>. Acesso em: 10 nov.2014.
- ENGINEERSGARAGE. **Embedded Linux**: understanding the embedded Linux. [S.I.], 2014. Disponível em: <<http://www.engineersgarage.com/articles/what-is-embedded-linux>>. Acesso em: 16 out. 2014.
- ERICKSON, Kelvin T. **Programmable logic controller hardware**. [S.I.], 2010. Disponível em: <<https://www.isa.org/standards-and-publications/isa-publications/intech/2010/december/programmable-logic-controller-hardware/>>. Acesso em: 07 nov. 2014.
- GEORGINI, Marcelo. **Automação aplicada**: descrição e implementação de sistemas sequenciais com PLCs. 2. Ed. São Paulo: Érica, 2000.
- GNU. **GCC, the GNU compiler collection**. [S. I.], [2014?]. Disponível em: <<https://gcc.gnu.org/>>. Acesso em: 08 nov. 2014.

GNUBLIN. **What's GNUBLIN**. [S.I.], 2014. Disponível em: <<http://gnublin.embedded-projects.net/whats-gnublin/>>. Acesso em: 15 out. 2014.

HOLLABAUGH, Craig. **Embedded Linux: hardware, software, and interfacing**. New York: Sams, 2002.

KERNEL. **GPIO interfaces**. [S.I.], [2014?]. Disponível em: <<https://www.kernel.org/doc/Documentation/gpio/gpio-legacy.txt>>. Acesso em: 08 nov. 2014.

LOMBARDO, John. **Embedded Linux**. Indianapolis: New Riders, 2002.

LOUDEN, Kenneth C. **Compiladores: princípios e práticas**. São Paulo: Thomson Pioneira, 2004.

MARTINS, Geomar Machado. **Princípios da automação industrial**. Santa Maria, 2012. Disponível em: <[http://coral.ufsm.br/desp/geomar/automacao/Apostila\\_032012.pdf](http://coral.ufsm.br/desp/geomar/automacao/Apostila_032012.pdf)>. Acesso em: 6 abr. 2014.

MAXWELL, Scott. **Kernel do Linux**. São Paulo: Makron Books, 2000.

NATALE, Ferdinando. **Automação industrial**. 4. ed. São Paulo: Érica, 2000.

NEGUS, Christopher. **Linux: a bíblia**. Rio de Janeiro: Alta Books, 2008.

ORACLE. **Lesson: Overview of the Java 2D API concepts**. [S.I.], 2014. Disponível em: <<http://docs.oracle.com/javase/tutorial/2d/overview/index.html>>. Acesso em: 17 out. 2014.

PLCOPEN. **IEC 61131-3: a standard programming resource**. [S.I.], 2014a. Disponível em: <[http://www.plcopen.org/pages/tc1\\_standards/downloads/intro\\_iec.pdf](http://www.plcopen.org/pages/tc1_standards/downloads/intro_iec.pdf)>. Acesso em: 17 out. 2014.

\_\_\_\_\_. **Introduction into IEC 61131-3 programming languages: a standard programming resource**. [S.I.], 2014b. Disponível em: <[http://www.plcopen.org/pages/tc1\\_standards/iec\\_61131\\_3/](http://www.plcopen.org/pages/tc1_standards/iec_61131_3/)>. Acesso em: 17 out. 2014.

RICARTE, Ivan L. M. **Código de três endereços**. Campinas, 2003. Disponível em: <<http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node74.html>>. Acesso em: 08 nov. 2014.

RIESE, Kitrek. **Embedded Linux Systems**. Platteville, 2014. Disponível em: <<http://people.uwplatt.edu/~yangq/CSSE411/csse411-materials/s05/Riesek%20-%20EmbeddedLinuxSystems.doc>>. Acesso em: 16 out. 2014.

SURVEYMONKEY. **A escala de Likert explicada**. [S.I.], [2014?]. Disponível em: <<https://pt.surveymonkey.com/mp/likert-scale/>>. Acesso em: 12 nov. 2014.

TORVALDS, Linus. **Linux. a portable operating system**. Helsinki, 1997. Disponível em: <<http://mirror.linux.org.au/pub/linux/kernel/people/torvalds/thesis/torvalds97.pdf>>. Acesso em: 16 out. 2014.

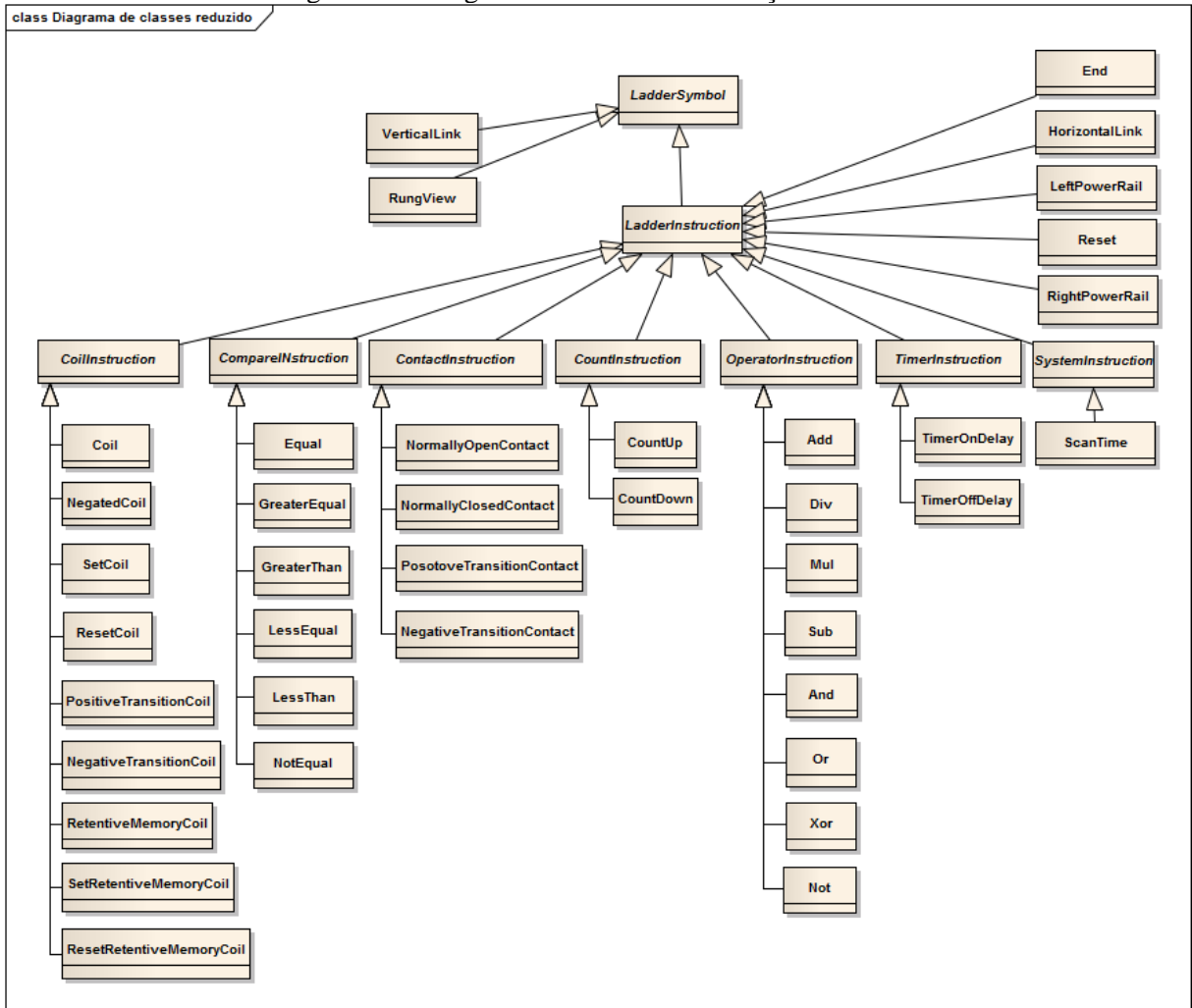
WAZLAWICK, Raul Sidnei. **Engenharia de software: conceitos e práticas**. Rio de Janeiro: Elsevier, 2013.

WESTHUES, Jonathan. **Ladder logic for PIC and AVR**. Oregon, [2007?]. Disponível em: <<http://cq.cx/ladder.pl>>. Acesso em: 15 out. 2014.

### APÊNDICE A – Diagrama de classes de instruções reduzido

A Figura 49 demonstra o diagrama de classes das instruções Ladder com os métodos e atributos omitidos.




Figura 49 – Diagrama de classes de instrução reduzido









## APÊNDICE B – Símbolos da linguagem Ladder.










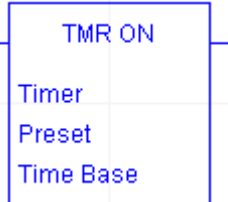
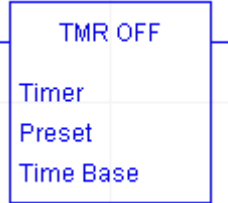
O Quadro 25 demonstra os símbolos dos elementos estruturais da linguagem Ladder, enquanto o Quadro 26 demonstra os símbolos das instruções Ladder do protótipo. Os quadros são divididos em três colunas sendo respectivamente o nome do símbolo, a figura do símbolo e a descrição da finalidade ou comportamento do símbolo no programa.



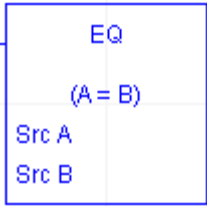
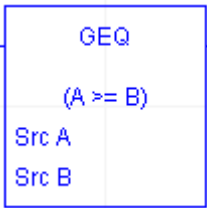
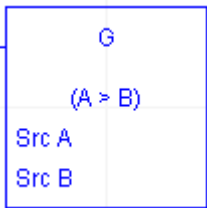
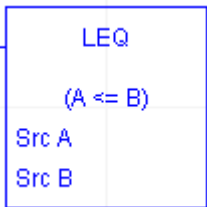
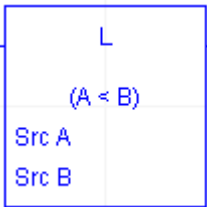
Quadro 25 – Símbolos dos elementos estruturais

Nome	Símbolo Ladder	Descrição
Left power rail		Início do <i>rung</i> . Define o estado do primeiro link à direita como <code>true</code> .
Right power rail		Fim do <i>rung</i> .
Horizontal link		Região disponível para instruções Ladder.
Vertical link		Conecta os <i>rungs</i> ou instruções Ladder em paralelo.

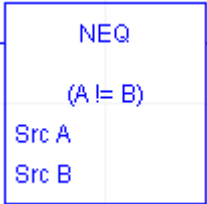


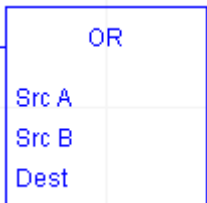
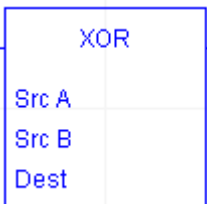
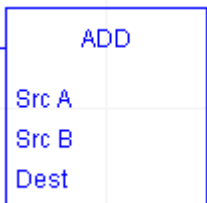
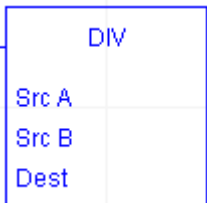
Quadro 26 – Símbolos das instruções Ladder

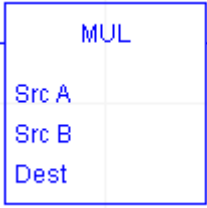
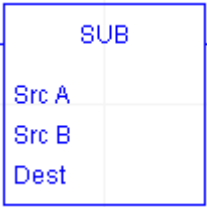
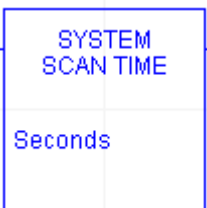
Nome	Símbolo Ladder	Descrição
Normally open Contact		O estado do <i>link</i> esquerdo é copiado para o <i>link</i> direito se o estado da variável booleana associada for <code>true</code> . Caso contrário o estado do <i>link</i> direito é <code>false</code> .
Normally closed contact		O estado do <i>link</i> esquerdo é copiado para o <i>link</i> direito se o estado da variável booleana associada for <code>false</code> . Caso contrário o estado do <i>link</i> direito é <code>false</code> .
Positive transition-sensing contact		O estado do <i>link</i> direito é <code>true</code> quando uma transição da variável associada de <code>false</code> para <code>true</code> for detectado ao mesmo tempo em que o estado do <i>link</i> esquerdo for <code>true</code> . O estado do <i>link</i> direito é <code>false</code> em todas as outras vezes.
Negative transition-sensing contact		O estado do <i>link</i> direito é <code>true</code> quando uma transição da variável associada de <code>true</code> para <code>false</code> for detectado ao mesmo tempo em que o estado do <i>link</i> esquerdo for <code>true</code> . O estado do <i>link</i> direito é <code>false</code> em todas as outras vezes.
Coil		O estado do <i>link</i> esquerdo é copiado para a variável booleana associada e para o <i>link</i> direito.
Negated coil		O estado do <i>link</i> esquerdo é copiado para o <i>link</i> direito. O inverso do estado do <i>link</i> esquerdo é copiado para a variável booleana associada, ou seja, se o estado do <i>link</i> esquerdo for <code>false</code> o estado da variável associada é <code>true</code> e vice versa.

Nome	Símbolo Ladder	Descrição
Set coil		A variável booleana associada é definida como <code>true</code> quando o estado do <i>link</i> esquerdo for <code>true</code> e mantém até ser resetado por um <code>Reset coil</code> .
Reset coil		A variável booleana associada é definida com estado <code>false</code> quando o estado do <i>link</i> esquerdo for <code>true</code> e mantém até ser definido por um <code>Set coil</code> .
Retentive coil		Idêntico a instrução <code>Coil</code> , exceto pelo fato que a variável associada é declarada para ser preferencialmente uma memória retentiva (variável global).
Set retentive coil		Idêntico a instrução <code>Set coil</code> , exceto pelo fato que a variável associada é declarada para ser preferencialmente uma memória retentiva (variável global).
Reset retentive coil		Idêntico a instrução <code>Reset coil</code> , exceto pelo fato que a variável associada é declarada para ser preferencialmente uma memória retentiva (variável global).
Positive transition-sensing coil		O estado da variável booleana associada é <code>true</code> a partir de uma próxima avaliação do elemento quando a transição do <i>link</i> esquerdo de <code>false</code> para <code>true</code> for percebido. O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.
Negative transition-sensing coil		O estado da variável booleana associada é <code>true</code> a partir de uma próxima avaliação do elemento quando a transição do <i>link</i> esquerdo de <code>true</code> para <code>false</code> for percebido. O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.
End		Representa o fim do programa Ladder.
Reset		A memória associada é limpa quando o estado do <i>link</i> esquerdo for <code>true</code> . O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.
Timer On		O acumulador da instrução é incrementado quando o estado do <i>link</i> esquerdo for <code>true</code> . O estado do <i>link</i> direito é <code>true</code> quando o estado do <i>link</i> esquerdo for <code>true</code> e o acumulador for igual ao valor de <code>preset</code> . Se o estado do <i>link</i> esquerdo for <code>false</code> o acumulador é definido como zero.
Timer Off		O acumulador da instrução é incrementado quando o estado do <i>link</i> esquerdo for <code>true</code> . O estado do <i>link</i> direito é <code>true</code> quando o estado do <i>link</i> esquerdo for <code>true</code> e o acumulador for diferente do valor de <code>preset</code> . Se o estado do <i>link</i> esquerdo for <code>false</code> o acumulador é definido como zero.

Nome	Símbolo Ladder	Descrição
Count Up		O acumulador da instrução é incrementado quando o estado do <i>link</i> esquerdo for true. O estado do <i>link</i> direito é true quando o estado do <i>link</i> esquerdo for true e o acumulador for maior ou igual ao valor de preset. Se o estado do <i>link</i> esquerdo for false o acumulador é definido como zero
Count Down		O acumulador da instrução é decrementado quando o estado do <i>link</i> esquerdo for true. O estado do <i>link</i> direito é true quando o estado do <i>link</i> esquerdo for true e o acumulador for menor ou igual ao valor de preset. Se o estado do <i>link</i> esquerdo for false o acumulador é definido como zero
Equal		O estado do <i>link</i> direito é true quando o estado do <i>link</i> esquerdo for true e as variáveis associadas Src A e Src B forem iguais.
Greater Equal		O estado do <i>link</i> direito é true quando o estado do <i>link</i> esquerdo for true e a variável associada Src A for maior ou igual que a variável associada Src B.
Greater Than		O estado do <i>link</i> direito é true quando o estado do <i>link</i> esquerdo for true e a variável associada Src A for maior que variável associada Src B.
Less Equal		O estado do <i>link</i> direito é true quando o estado do <i>link</i> esquerdo for true e a variável associada Src A for menor ou igual que variável associada Src B.
Less		O estado do <i>link</i> direito é true quando o estado do <i>link</i> esquerdo for true e a variável associada Src A for menor que variável associada Src B.



Nome	Símbolo Ladder	Descrição
Not Equal		O estado do <i>link</i> direito é <i>true</i> quando o estado do <i>link</i> esquerdo for <i>true</i> e a variável associada <i>Src A</i> for diferente que variável associada <i>Src B</i> .
And		A variável associada <i>Dest</i> recebe o resultado da operação lógica AND entre as variáveis associadas <i>Src A</i> e <i>Src B</i> . O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.
Not		A variável associada <i>Dest</i> recebe a negação da variável associada <i>Src A</i> . O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.
Or		A variável associada <i>Dest</i> recebe o resultado da operação lógica OR entre as variáveis associadas <i>Src A</i> e <i>Src B</i> . O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.
Xor		A variável associada <i>Dest</i> recebe o resultado da operação lógica XOR entre as variáveis associadas <i>Src A</i> e <i>Src B</i> . O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.
Add		A variável associada <i>Dest</i> recebe o resultado da operação aritmética de adição entre as variáveis associadas <i>Src A</i> e <i>Src B</i> . O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.
Div		A variável associada <i>Dest</i> recebe o resultado da operação aritmética de divisão entre as variáveis associadas <i>Src A</i> e <i>Src B</i> . O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.

Nome	Símbolo Ladder	Descrição
Mul		A variável associada <code>Dest</code> recebe o resultado da operação aritmética de multiplicação entre as variáveis associadas <code>Src A</code> e <code>Src B</code> . O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.
Sub		A variável associada <code>Dest</code> recebe o resultado da operação aritmética de subtração entre as variáveis associadas <code>Src A</code> e <code>Src B</code> . O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.
System Scan Time		A instrução <code>Scan Time</code> representa a instrução do tempo de varredura do programa em execução. O estado do <i>link</i> esquerdo é sempre copiado para o <i>link</i> direito.

## APÊNDICE C – *Makefile* e *shell script* Linux

O Quadro 27 demonstra o arquivo *Makefile* utilizado para compilar no hardware o programa Ladder traduzido em linguagem C. O Quadro 28 demonstra a arquivo *shell script* Linux utilizado no hardware para criar e configurar as interfaces E/S.

Quadro 27 – Arquivo *Makefile* gerado

```

1 SRC=gcc
2 EXEC=plc
3 OBJ=plc.o
4
5 plc: plc.o
6     $(CC) -o $@ $^
7 %.o: %.c
8     $(CC) -o $@ -c $<
9 .PHONY: clean mrproper
10 clean:
11     rm -rf *.o
12 mrproper: clean
13     rm -rf $(EXEC)

```

Quadro 28 – Arquivo *shell script* Linux gerado

```

1 #!/bin/bash -e
2
3 if [ ! -d /sys/class/gpio/gpio112 ]; then echo 112 >
4 /sys/class/gpio/export; fi
5 if [ ! -d /sys/class/gpio/gpio113 ]; then echo 113 >
6 /sys/class/gpio/export; fi
7 if [ ! -d /sys/class/gpio/gpio114 ]; then echo 114 >
8 /sys/class/gpio/export; fi
9 if [ ! -d /sys/class/gpio/gpio115 ]; then echo 115 >
10 /sys/class/gpio/export; fi
11 if [ ! -d /sys/class/gpio/gpio116 ]; then echo 116 >
12 /sys/class/gpio/export; fi
13 if [ ! -d /sys/class/gpio/gpio117 ]; then echo 117 >
14 /sys/class/gpio/export; fi
15 if [ ! -d /sys/class/gpio/gpio98 ]; then echo 98 >
16 /sys/class/gpio/export; fi
17 if [ ! -d /sys/class/gpio/gpio99 ]; then echo 99 >
18 /sys/class/gpio/export; fi
19 if [ ! -d /sys/class/gpio/gpio104 ]; then echo 104 >
20 /sys/class/gpio/export; fi
21 if [ ! -d /sys/class/gpio/gpio105 ]; then echo 105 >
22 /sys/class/gpio/export; fi
23 if [ ! -d /sys/class/gpio/gpio106 ]; then echo 106 >
24 /sys/class/gpio/export; fi
25 if [ ! -d /sys/class/gpio/gpio107 ]; then echo 107 >
26 /sys/class/gpio/export; fi
27 if [ ! -d /sys/class/gpio/gpio108 ]; then echo 108 >
28 /sys/class/gpio/export; fi
29 if [ ! -d /sys/class/gpio/gpio109 ]; then echo 109 >
30 /sys/class/gpio/export; fi
31 if [ ! -d /sys/class/gpio/gpio110 ]; then echo 110 >
32 /sys/class/gpio/export; fi
33 if [ ! -d /sys/class/gpio/gpio111 ]; then echo 111 >
34 /sys/class/gpio/export; fi
35
36 echo out > /sys/class/gpio/gpio112/direction
37 echo out > /sys/class/gpio/gpio113/direction
38 echo out > /sys/class/gpio/gpio114/direction

```

```
39 echo out > /sys/class/gpio/gpio115/direction
40 echo out > /sys/class/gpio/gpio116/direction
41 echo out > /sys/class/gpio/gpio117/direction
42 echo out > /sys/class/gpio/gpio98/direction
43 echo out > /sys/class/gpio/gpio99/direction
44
45 echo in > /sys/class/gpio/gpio104/direction
46 echo in > /sys/class/gpio/gpio105/direction
47 echo in > /sys/class/gpio/gpio106/direction
48 echo in > /sys/class/gpio/gpio107/direction
49 echo in > /sys/class/gpio/gpio108/direction
50 echo in > /sys/class/gpio/gpio109/direction
51 echo in > /sys/class/gpio/gpio110/direction
52 echo in > /sys/class/gpio/gpio111/direction
53
54 echo 0 > /sys/class/gpio/gpio112/value
55 echo 0 > /sys/class/gpio/gpio113/value
56 echo 0 > /sys/class/gpio/gpio114/value
57 echo 0 > /sys/class/gpio/gpio115/value
58 echo 0 > /sys/class/gpio/gpio116/value
59 echo 0 > /sys/class/gpio/gpio117/value
60 echo 0 > /sys/class/gpio/gpio98/value
61 echo 0 > /sys/class/gpio/gpio99/value
```

## APÊNDICE D – Estado inicial do hardware

A Figura 50 exibe o resultado do comando `ps -aux` através do terminal do Linux no hardware, após o término do *boot*. O comando lista os processos em execução no sistema exibindo informações sobre cada processo. Em seguida foi executado o comando `free -m`, que permite visualizar a utilização de memória no Linux, como ilustrado na Figura 51.

Figura 50 – Processos ativos no hardware listados pelo comando `ps -aux`

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	1.9	0.5	2076	696	?	Ss	01:00	0:00	init [2]
root	2	0.0	0.0	0	0	?	S	01:00	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	S	01:00	0:00	[ksoftirqd/0]
root	4	0.0	0.0	0	0	?	S	01:00	0:00	[kworker/0:0]
root	5	0.5	0.0	0	0	?	S	01:00	0:00	[kworker/u:0]
root	6	0.7	0.0	0	0	?	S	01:00	0:00	[rcu_kthread]
root	7	0.0	0.0	0	0	?	S<	01:00	0:00	[khelper]
root	8	0.0	0.0	0	0	?	S	01:00	0:00	[kworker/u:1]
root	176	0.0	0.0	0	0	?	S	01:00	0:00	[sync_supers]
root	178	0.0	0.0	0	0	?	S	01:00	0:00	[bdi-default]
root	180	0.0	0.0	0	0	?	S<	01:00	0:00	[kblockd]
root	192	0.0	0.0	0	0	?	S	01:00	0:00	[khubd]
root	225	0.0	0.0	0	0	?	S<	01:00	0:00	[l2cap]
root	228	0.0	0.0	0	0	?	S<	01:00	0:00	[cfg80211]
root	229	0.1	0.0	0	0	?	S	01:00	0:00	[kworker/0:1]
root	318	0.0	0.0	0	0	?	S	01:00	0:00	[kswapd0]
root	319	0.0	0.0	0	0	?	S	01:00	0:00	[fsnotify_mark]
root	320	0.0	0.0	0	0	?	S<	01:00	0:00	[crypto]
root	435	0.0	0.0	0	0	?	S	01:00	0:00	[wl_bus_master]
root	447	0.0	0.0	0	0	?	S<	01:00	0:00	[krfcommd]
root	452	0.0	0.0	0	0	?	S	01:00	0:00	[kworker/u:2]
root	455	1.2	0.0	0	0	?	S	01:00	0:00	[mmcqd/0]
root	462	0.0	0.0	0	0	?	S	01:00	0:00	[jbd2/mmcblk0p]
root	463	0.0	0.0	0	0	?	S<	01:00	0:00	[ext4-dio-unwr]
root	504	1.1	0.5	3144	720	?	S<S	01:00	0:00	udevd --daemon
root	565	0.0	0.5	3140	632	?	S<	01:00	0:00	udevd --daemon
root	566	0.0	0.5	3140	632	?	S<	01:00	0:00	udevd --daemon
root	606	0.0	0.0	0	0	?	S	01:00	0:00	[flush-179:0]
root	709	0.0	0.0	0	0	?	S	01:00	0:00	[jbd2/mmcblk0p]
root	710	0.0	0.0	0	0	?	S<	01:00	0:00	[ext4-dio-unwr]
daemon	785	0.0	0.3	1828	484	?	Ss	01:00	0:00	/sbin/portmap
root	874	0.8	1.0	27540	1352	?	S1	01:00	0:00	/usr/sbin/rsysl
root	905	0.0	0.4	3944	560	?	Ss	01:00	0:00	/usr/sbin/famd
root	936	0.0	0.6	2476	856	?	Ss	01:00	0:00	/usr/sbin/cron
root	953	0.0	0.7	6772	968	?	Ss	01:00	0:00	/usr/sbin/sshd
www-data	977	0.0	0.8	6612	1104	?	S	01:00	0:00	/usr/sbin/light
www-data	979	1.7	3.8	18240	4820	?	Ss	01:00	0:00	/usr/bin/php-cg
root	995	6.0	1.0	3156	1280	ttyS0	Ss	01:00	0:00	/bin/login --
www-data	996	0.0	1.6	18240	2016	?	S	01:00	0:00	/usr/bin/php-cg
www-data	997	0.0	1.6	18240	2016	?	S	01:00	0:00	/usr/bin/php-cg
root	998	1.6	1.3	3212	1652	ttyS0	S	01:00	0:00	-bash
root	1002	0.0	0.7	2576	984	ttyS0	R+	01:00	0:00	ps -aux

Figura 51 – Resultado do comando `free -m`

	total	used	free	shared	buffers	cached
Mem:	121	31	89	0	3	20
-/+ buffers/cache:		8	112			
Swap:	0	0	0			

**APÊNDICE E – Formulário de avaliação do protótipo**

Figura 52 – Formulário de avaliação do protótipo

## Avaliação do Protótipo de CLP para Linux Embarcado

A seguir leia a afirmação e indique seu grau de concordância com cada uma delas, segundo a escala que varia de 1 (discordância total) a 5 (concordância total).

**1. O editor permite criar um programa de acordo com as características da linguagem Ladder.**

1    2    3    4    5

---

Discordo totalmente                Concordo totalmente

---

**2. A execução do código gerado corresponde a programa Ladder criado.**

1    2    3    4    5

---

Discordo totalmente                Concordo totalmente

---

**3. A utilização do editor facilita a criação da lógica Ladder.**

1    2    3    4    5

---

Discordo totalmente                Concordo totalmente

---

**4. O código gerado é executado com alta performance.**

1    2    3    4    5

---

Discordo totalmente                Concordo totalmente

---

**5. Relate pontos fortes observados (opcional).**

**6. Relate pontos fracos observados (opcional).**

## APÊNDICE F – Respostas descritivas referentes ao formulário de avaliação

As respostas descritivas referentes ao formulário de avaliação efetuado com a turma de Automação estão descritas no Quadro 29 e Quadro 30. Os quadros foram redigidos exatamente como os alunos escreveram no formulário.

Quadro 29 – Pontos fortes observados

<b>Descrição</b>
Não precisa de download, já começa automaticamente.
Não precisa fazer download.
Fácil manuseio.
Fácil inclusão das variáveis.
Run – acumula funções, ambiente visual fácil manuseio.
Ambiente bem visual, fácil utilização.
Run é eficiente, já gera o download, maior desempenho.
Fácil adição e configuração.
Fácil de mexer e executar, tem uma linguagem muito simples.
Facilidade na execução dos códigos, interface (ilegível).
Facilidade de trabalhar com a interface.
Facilidade no uso, intuitivo.
Fácil utilização.

Quadro 30 – Pontos fracos observados

<b>Descrição</b>
Não apresentou.
Durante o teste não apresentou dificuldades, precisamos mais tempo para analisar.
A inclusão de linhas deveria ser em baixo não em cima.
Inclusão de linhas no programa deveria ser embaixo.
Add linhas abaixo p/ facilitar lógica.
Poder colocar linhas em baixo, Poder colocar as memórias nas entradas e saídas.
Algumas propriedades do software apresentam funcionalidades sua, lógica da linha adiciona uma linha mais cria acima, não segue uma lógica conveniente, o certo seria uma linha abaixo.
Adicionar linhas abaixo das feitas. Nas propriedades do Set e Reset conseguir colocar as memórias.
A adição de linhas deveria ser adicionada à baixo e não acima da linha programada.
Melhorar a opção de paralelo. Linhas criadas irem em para baixo.
Adição de linha acima. Poderia add linhas abaixo.