

**INSTITUTO FEDERAL DO TRIÂNGULO MINEIRO  
ENGENHARIA DE COMPUTAÇÃO**

**DESENVOLVIMENTO E OTIMIZAÇÃO DE UM  
SIMULADOR DE CLP COM INTERFACE  
INTERATIVA E CENÁRIO DE TRÁFEGO**

**Autores:**

Jamilly Moura

Pedro Franco de Camargo

Pedro Henrique Cândido Silva

**Orientador:**

Prof. Robson Rodrigues

**UBERABA - MG**

**2025**

# Sumário

<b>1</b>	<b>Introdução</b>	<b>3</b>
1.1	Objetivos . . . . .	3
<b>2</b>	<b>Arquitetura e Reestruturação do Sistema</b>	<b>5</b>
2.1	Ambiente de Desenvolvimento . . . . .	5
2.2	O Ciclo de Varredura (Scan Cycle) . . . . .	5
2.3	Correção de Assets Gráficos . . . . .	8
<b>3</b>	<b>Otimização do Núcleo (Backend)</b>	<b>9</b>
3.1	Correção de Inicialização de Memória (M0) . . . . .	9
3.2	Correção de Sintaxe (TOF e Numéricos) . . . . .	9
3.3	Monitor de Variáveis (Data Table) . . . . .	10
3.4	Limpeza de Memória Visual . . . . .	11
<b>4</b>	<b>Cenário de Simulação: Traffic Light</b>	<b>12</b>
4.1	Implementação Gráfica e Perspectiva . . . . .	13
4.2	Mapeamento de E/S (Hardware Virtual) . . . . .	13
4.3	Física e Correção de "Tunneling" . . . . .	13
4.4	Sensores e Detecção de Colisão . . . . .	13
<b>5</b>	<b>Validação e Testes</b>	<b>15</b>
5.1	Casos de Teste . . . . .	15
<b>6</b>	<b>Empacotamento e Distribuição</b>	<b>17</b>
6.1	Executáveis e Instalador . . . . .	17
6.2	Requisitos de Sistema . . . . .	17

6.3	Interface de Ajuda	17
<b>7</b>	<b>Conclusão</b>	<b>18</b>

# Capítulo 1

## Introdução

Este relatório técnico descreve o processo de desenvolvimento, reestruturação e implementação de novas funcionalidades no software de Simulação de Controlador Lógico Programável (CLP). O projeto tem como objetivo principal fornecer uma ferramenta didática robusta, capaz de interpretar a linguagem Lista de Instruções (IL - *Instruction List*) conforme a norma IEC 61131-3, simulando o comportamento de sistemas industriais em tempo real.

A motivação para este desenvolvimento baseia-se na necessidade de ferramentas acessíveis para o ensino de automação, similares ao software comercial LogixPro, permitindo aos estudantes testar lógicas de controle, temporizadores, contadores e tratamento de intertravamentos em um ambiente seguro e visual.

O projeto atual representa uma evolução significativa de versões anteriores, focando na estabilidade do interpretador, na migração para tecnologias modernas (Java JDK 23) e na criação de um cenário de simulação complexo denominado "Traffic Light"(Semáforo), que emula um cruzamento urbano com física de veículos e sensores indutivos.

### 1.1 Objetivos

- Recuperar e reestruturar o código-fonte legado, aplicando padrões de projeto MVC.
- Corrigir falhas críticas no núcleo do interpretador de instruções e na gestão de memória.
- Desenvolver um novo cenário gráfico de trânsito utilizando renderização vetorial

(Java 2D), livre de dependências de imagens externas.

- Implementar algoritmos de física para movimentação de veículos, detecção de colisão e sensores virtuais.
- Empacotar e distribuir a solução com instaladores profissionais para ambiente Windows.

# Capítulo 2

## Arquitetura e Reestruturação do Sistema

Para garantir a escalabilidade e a manutenção do simulador, foi realizada uma ampla reestruturação do ambiente de desenvolvimento. O software segue o padrão MVC (*Model-View-Controller*), separando a lógica de execução da interface visual.

### 2.1 Ambiente de Desenvolvimento

O projeto original encontrava-se consolidado em arquivos de texto brutos. Foi desenvolvido um script em Python (`extrair.py`) para automatizar a extração das classes e a migração para a IDE NetBeans. Para suportar recursos modernos da linguagem Java, o ambiente foi configurado com o **Oracle JDK 23 (x64)**. Problemas de variáveis de ambiente no Windows (erro de reconhecimento do ‘javac’) foram solucionados, estabelecendo uma rotina de compilação via terminal e Ant.

### 2.2 O Ciclo de Varredura (Scan Cycle)

Para emular fielmente um CLP real, implementamos um ciclo de varredura contínuo controlado por um `javax.swing.Timer` com base de tempo de 100ms. O ciclo executa os seguintes passos a cada *tick*:

1. **Leitura das Entradas:** Captura o estado físico dos botões e sensores da cena atual.

2. **Processamento Lógico:** O interpretador percorre o código IL linha por linha, atualizando as memórias internas.
3. **Atualização das Saídas:** Os valores calculados são enviados para os atuadores (Luzes, Motores) na interface gráfica.

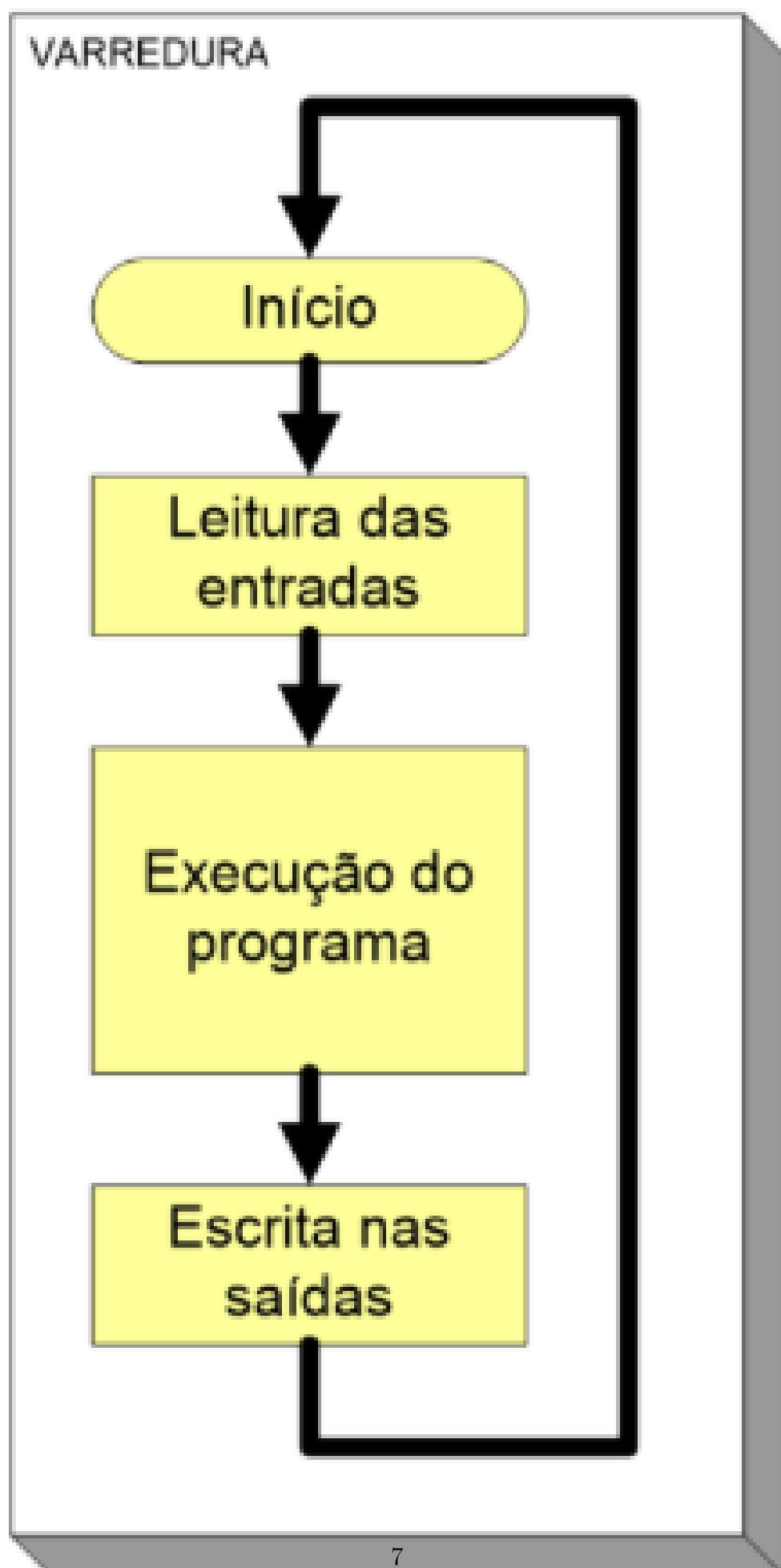


Figura 2.1: Fluxograma do Ciclo de Varredura (Scan Cycle).

## 2.3 Correção de Assets Gráficos

Identificou-se uma falha crítica (*NullPointerException*) relacionada ao carregamento de imagens. O compilador Java não move automaticamente arquivos não-Java para a pasta de binários. Foi implementada uma etapa de *build* manual para copiar recursivamente o diretório `Assets` para a pasta `bin`, garantindo o carregamento dos ícones da interface.

# Capítulo 3

## Otimização do Núcleo (Backend)

O núcleo do sistema é a classe `Interpreter.java`. Diversas correções foram aplicadas para garantir conformidade técnica.

### 3.1 Correção de Inicialização de Memória (M0)

Identificou-se um bug crítico onde o interpretador falhava ao ler variáveis de memória (ex: LD M0) antes que elas fossem escritas (ST). **Solução:** Implementou-se uma lógica de "Auto-Inicialização". Se o código tenta ler uma variável que ainda não existe no mapa de memória, o sistema a instancia automaticamente com valor falso (0), prevenindo falhas e permitindo lógicas de selo na primeira linha do programa.

### 3.2 Correção de Sintaxe (TOF e Numéricos)

- **Parser Numérico:** O sistema falhava ao reconhecer variáveis contendo dígitos '0' e '9'. O código foi refatorado para aceitar o intervalo completo de caracteres ASCII numéricos.
- **Instrução TOF:** A sintaxe do temporizador *Timer Off-Delay* foi corrigida de "TOFF" para o padrão "TOF".

### 3.3 Monitor de Variáveis (Data Table)

A tela `ListaDeVariaveisPg.java` apresentava problemas de performance e "piscava" intermitente.

**Otimização (Upsert):** Substituímos a lógica de recriação da tabela por uma verificação inteligente. Se o ID da variável já existe, apenas o valor é atualizado; caso contrário, insere-se uma nova linha. Adicionalmente, implementou-se um renderizador condicional que exibe valores **TRUE** em verde e **FALSE** em vermelho.

ID	CurrentValue	Counter	MaxTimer	EndTimer
I1.5	false			
I1.6	false			
I1.7	false			
Q0.0	true			
Q0.1	false			
Q0.2	false			
Q0.3	false			
Q0.4	false			
Q0.5	true			
Q0.6	false			
Q0.7	false			
Q1.0	false			
Q1.1	false			
Q1.2	false			
Q1.3	false			
Q1.4	false			
Q1.5	false			
Q1.6	false			
Q1.7	false			
T1	true	50	50	true
T2	true	20	20	true
T3	true	15	15	true
T4	true	50	50	true
T5	true	10	20	false
T6	false	0	15	false

Figura 3.1: Monitor de variáveis otimizado com indicação visual de estado.

### **3.4 Limpeza de Memória Visual**

Foi criado o método `clearMemoryLabels()` para corrigir um erro onde valores antigos de temporizadores e contadores permaneciam na tela após reiniciar a simulação. Agora, o botão START força uma limpeza visual completa.

# Capítulo 4

## Cenário de Simulação: Traffic Light

O principal desenvolvimento foi o cenário "Traffic Light" (Semáforo), replicando exercícios clássicos do LogixPro. Todo o cenário foi renderizado programaticamente via `java.awt.Graphics2D`, sem uso de imagens estáticas, garantindo leveza.

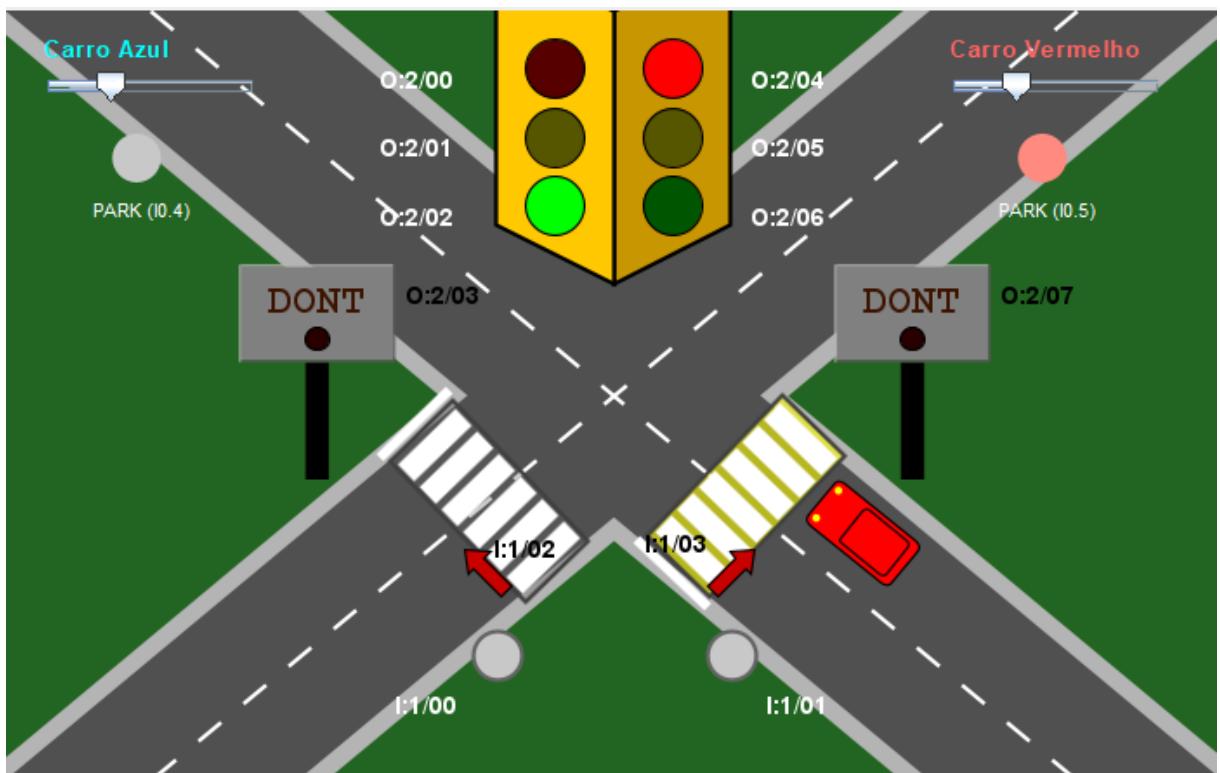


Figura 4.1: Visão geral do cenário Traffic Light renderizado via Java 2D.

## 4.1 Implementação Gráfica e Perspectiva

Utilizou-se perspectiva isométrica com ângulo de aproximadamente 34º. Os elementos (ruas, calçadas, torre do semáforo) são desenhados como objetos `Polygon`. As faixas de pedestres utilizam transformações geométricas (`AffineTransform`) para rotacionar as listras alinhadas à rua.

## 4.2 Mapeamento de E/S (Hardware Virtual)

O cenário interage com a memória do CLP conforme o mapeamento:

- **Saídas (Q):**

- Norte-Sul: Q0.0 (Vermelho), Q0.1 (Amarelo), Q0.2 (Verde).
- Leste-Oeste: Q0.3 (Vermelho), Q0.4 (Amarelo), Q0.5 (Verde).
- Pedestres: Q0.3 e Q0.7 ("DONT WALK").

- **Entradas (I):**

- I1.0 / I1.1: Botões de Pedestre.
- I1.2 / I1.3: Sensores Indutivos de Asfalto.
- I1.4 / I1.5: Chaves "Park" (Habilitação dos carros).

## 4.3 Física e Correção de "Tunneling"

A movimentação dos veículos é calculada vetorialmente. Identificou-se um problema onde carros em alta velocidade "pulavam" a linha de parada (Tunneling). **Solução:** Implementou-se um algoritmo preditivo. O sistema calcula a posição futura do quadro seguinte; se a posição cruzar a linha de retenção e o sinal estiver vermelho, o carro é forçado a parar na coordenada exata da faixa.

## 4.4 Sensores e Detecção de Colisão

**Sensores Indutivos:** Os sensores I1.2 e I1.3 mudam de cor (Cinza para Amarelo) e enviam sinal lógico quando o carro está dentro de uma zona específica (progresso entre

25% e 39%).

**Colisão (Crash):** Foi definida uma "Zona Crítica" no centro do cruzamento. Se ambos os carros entrarem nesta zona simultaneamente (devido a erro na lógica do aluno):

1. A simulação para imediatamente.
2. Renderiza-se uma explosão visual nas coordenadas de impacto.
3. Exibe-se mensagem de erro crítico.

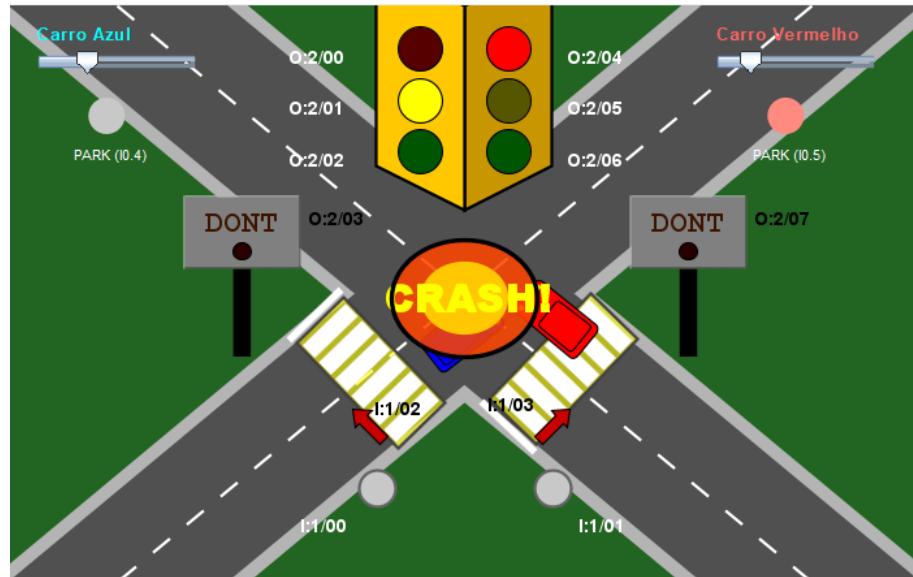


Figura 4.2: Simulação de acidente crítico por falha na lógica de controle.

# Capítulo 5

## Validação e Testes

Para validar as correções no interpretador e a física do cenário, foi desenvolvido um pacote de testes cobrindo 8 algoritmos distintos.

### 5.1 Casos de Teste

1. **Lógica Básica:** Teste de portas AND, OR e saídas diretas.
2. **Lógica Inversa:** Validação de LDN, STN, ANDN, ORN.
3. **Temporizadores:** Diferenciação prática entre TON e TOF.
4. **Contadores:** Teste de contagem com borda de subida e travamento.
5. **Semáforo Cascata:** Sistema complexo com 6 temporizadores e margem de segurança (vermelho duplo) na troca de vias.
6. **Pedestre Interativo:** Sistema reativo onde o sinal abre apenas sob demanda do botão.
7. **Validação de Hardware:** Integração GUI vs Lógica (Botões e Sensores).
8. **Contadores Independentes:** Múltiplos contadores (C1 e C2) simultâneos.

Todos os 8 cenários foram validados com sucesso.

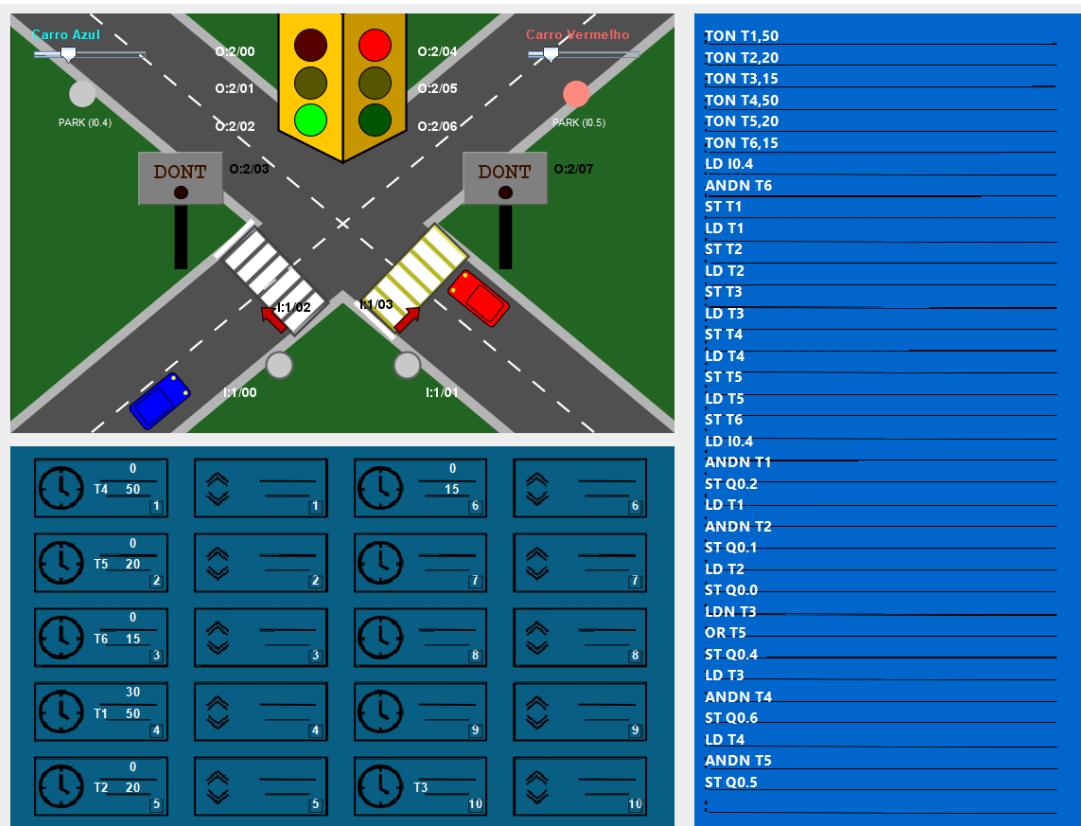


Figura 5.1: Execução de código IL para validação dos casos de teste.

# Capítulo 6

## Empacotamento e Distribuição

### 6.1 Executáveis e Instalador

Utilizou-se a ferramenta **Launch4j** para converter o arquivo `.jar` em um executável `.exe` nativo, permitindo verificação de versão do JRE. Para a instalação, foi criado um script no **Inno Setup Compiler**, gerando o arquivo `Instalador_Simulador_CLP.exe`. Este instalador gerencia a cópia de bibliotecas, criação de atalhos no Desktop e configuração de diretórios.

### 6.2 Requisitos de Sistema

Devido ao uso de recursos como *Switch Expressions* e *Text Blocks*, o software requer **Java JDK 21** ou superior para execução, decisão tomada para manter a base de código moderna e sustentável.

### 6.3 Interface de Ajuda

O sistema de ajuda foi reformulado utilizando HTML/CSS dentro do Swing, oferecendo tabelas de comandos e exemplos práticos coloridos, substituindo as antigas janelas de texto simples.

# Capítulo 7

## Conclusão

O projeto atingiu seus objetivos de entregar um Simulador de CLP funcional, robusto e pedagogicamente eficaz. A reestruturação da arquitetura para MVC e as correções no interpretador (bugs de memória e temporizadores) garantiram a estabilidade necessária.

A implementação do cenário "Traffic Light" provou a capacidade do sistema em simular processos complexos em tempo real, com física correta e tratamento de erros. O software final, devidamente empacotado e testado, encontra-se pronto para distribuição e uso acadêmico.