

Universidade Federal De Uberlândia

Jonas Mazza Fernando

**Desenvolvimento de um Controlador Lógico
Programável com Interface de Programação
Ladder via Aplicação Web Embarcada**

Uberlândia, Brasil

2017

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Jonas Mazza Fernando

**Desenvolvimento de um Controlador Lógico Programável
com Interface de Programação Ladder via Aplicação Web
Embarcada**

Trabalho apresentado como requisito parcial
de avaliação na disciplina Trabalho de
Conclusão de Curso 2 do Curso de
Engenharia Elétrica da Universidade Federal
de Uberlândia.

Orientador: Dr. Marcelo Barros de Almeida

Universidade Federal de Uberlândia – UFU

Faculdade de Engenharia Elétrica

Uberlândia, Brasil

2017

Jonas Mazza Fernando

Desenvolvimento de um Controlador Lógico Programável com Interface de Programação Ladder via Aplicação Web Embarcada

Trabalho apresentado como requisito parcial
de avaliação na disciplina Trabalho de
Conclusão de Curso 2 do Curso de
Engenharia Elétrica da Universidade Federal
de Uberlândia.

Dr. Marcelo Barros de Almeida
Orientador

Dr. Márcio José da Cunha

Me. Renato Ferreira Fernandes Júnior

Uberlândia, Brasil
2017

Dedico este trabalho a meus pais, cuja força e perseverança sempre me inspiraram.

Agradecimentos

Agradeço a meus pais pelo suporte incondicional, pois nunca mediram esforços para manter meus estudos. A meus avós pela constante preocupação e acompanhamento, e pelas longas seções de engenharia que fizeram projetos se tornarem realidade. A minha esposa, pelo eterno carinho e apoio. Ao Dr. Marcelo Barros, pela paciente orientação e excepcional empenho como professor.

Resumo

Este trabalho apresenta o desenvolvimento de um protótipo de controlador lógico programável, que contém uma interface de programação Ladder embarcada servida via aplicação web. Para isso, o projeto foi dividido em três partes: a elaboração de um editor de Ladder que funcione em navegadores web e que permita a criação de Ladder e sua tradução em bytecode; a concepção de uma máquina virtual que execute o bytecode gerado, cumprindo o papel do CLP; e a interligação destas duas partes através de um servidor web, que entregue a aplicação ao navegador e comunique-o com a máquina virtual, permitindo download e upload de código, além da execução de comandos de partida e parada. O protótipo foi implementado em uma placa de desenvolvimento com microcontrolador ARM Cortex-M7, porém o foco do projeto é manter o sistema independente da plataforma utilizada, aplicando-se conceitos de portabilidade.

Palavras-chave: Controlador Lógico Programável, Ladder, Sistema Embarcado, Aplicação Web.

Abstract

This work presents the development of a programmable logic controller prototype, which contains an embedded Ladder programming interface served by a web application. For this, the project was divided in three parts: the elaboration of a Ladder editor that works in web browsers and allows the creation of Ladder and its translation in bytecode; the design of a virtual machine that performs the generated bytecode, fulfilling the role of the CLP; and the interconnection of these two parts through a web server, which delivers the application to the browser, and communicates with the virtual machine, allowing download and upload of code, as well as the execution of start and stop commands. The prototype was implemented in a development board with ARM Cortex-M7 microcontroller, but the focus of the project is to keep the system independent of the platform used, applying concepts of portability.

Keywords: Programmable Logic Controller, Ladder, Embedded System, Web Application.

Lista de ilustrações

Figura 1 – Componentes básicos de um CLP	12
Figura 2 – <i>Ladder</i> com elementos em série	18
Figura 3 – <i>Ladder</i> com elementos em paralelo	19
Figura 4 – Exemplo para estrutura de dados	25
Figura 5 – Exemplo de grade	33
Figura 6 – Fluxograma do processo de desenho da <i>Ladder</i>	34
Figura 7 – Comprimento e profundidade de um nó série	35
Figura 8 – Exemplo de desenho da <i>Ladder</i> (Parte 1)	36
Figura 9 – Processo de desenho da <i>Ladder</i> (Parte 2)	37
Figura 10 – Processo de desenho da <i>Ladder</i> (Completa)	38
Figura 11 – Blocos funcionais na grade	39
Figura 12 – Conjunto final de desenhos	39
Figura 13 – <i>Layout</i> da aplicação no estado atual de desenvolvimento	41
Figura 14 – NUCLEO-F767ZI	42
Figura 15 – Montagem do circuito de testes em protoboard	42
Figura 16 – Configuração dos pinos no <i>software</i> STM32CubeMX	43
Figura 17 – Detalhamento da configuração da GPIO no <i>software</i> STM32CubeMX	43
Figura 18 – Contatos normalmente aberto e fechado	52
Figura 19 – Bobina de saída	52
Figura 20 – Bobinas de <i>set</i> e <i>reset</i>	52
Figura 21 – Bobinas de <i>set</i> e <i>reset</i>	53
Figura 22 – Blocos comparadores	54
Figura 23 – Blocos aritméticos	55
Figura 24 – Blocos contadores	55
Figura 25 – Diagrama de estados da instrução TON	56
Figura 26 – Diagrama de estados da instrução TOF	57
Figura 27 – Blocos de <i>timers</i>	57
Figura 28 – Exemplo de exceção na <i>Ladder</i>	58
Figura 29 – Organização da memória <i>Flash</i>	63
Figura 30 – Captura do <i>Wireshark</i> mostrando problemas no ARP	65
Figura 31 – Captura mostrando problemas em múltiplas conexões em paralelo	66
Figura 32 – Transferência de arquivos no navegador Firefox	66
Figura 33 – Inspeção dos pacotes da requisição de <i>Stop</i>	67
Figura 34 – Inspeção dos pacotes da requisição de <i>Download</i>	68
Figura 35 – Inspeção dos pacotes da requisição de <i>Upload</i>	69
Figura 36 – <i>Rung</i> utilizada no teste de tempo de execução	71

Lista de tabelas

Tabela 1 – Utilização da pilha	18
Tabela 2 – Introdução da pilha temporária	19
Tabela 3 – Primeiros <i>opcodes</i>	20
Tabela 4 – Tamanho das bibliotecas SVG	31
Tabela 5 – Prefixos implementados	50
Tabela 6 – Conjunto completo dos <i>opcodes</i> implementados	50
Tabela 7 – Exceção da instrução AND	58
Tabela 8 – Solução do caso especial	59

Lista de abreviaturas e siglas

CLP	Controlador Lógico Programável
MV	Máquina Virtual
GPIO	General Purpose Input/Output
CPU	Central Processing Unit
LIFO	Last In, First Out
HTTP	HyperText Transfer Protocol
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
JPG	Joint Photographic Experts Group
SVG	Scalable Vector Graphics
DOM	Document Object Model
JSON	JavaScript Object Notation
TCP	Transmission Control Protocol
IP	Internet Protocol
IDE	Integrated Development Environment
HAL	Hardware Abstraction Layer
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
AJAX	Asynchronous Javascript and XML
URL	Uniform Resource Locator
CGI	Common Gateway Interface

Sumário

1	INTRODUÇÃO	11
2	METODOLOGIA	15
2.1	Máquina Virtual	15
2.2	Editor de <i>Ladder</i>	22
2.2.1	Estudo de Viabilidade Técnica	22
2.2.2	Estrutura de Dados	24
3	DESENVOLVIMENTO	31
3.1	Editor de Ladder	31
3.2	Placa de Desenvolvimento	41
3.3	Máquina Virtual	43
3.3.1	Casos especiais	58
3.4	Servidor Web	60
3.5	Base de dados	62
4	RESULTADOS	65
5	CONSIDERAÇÕES FINAIS	72
	REFERÊNCIAS	74

1 Introdução

Os Controladores Lógicos Programáveis são sistemas microprocessados que executam funções de controle através de programas desenvolvidos pelo usuário. Estas funções estão geralmente associadas a sequências de operações discretas ou contínuas, que compõem a produção industrial.

Assim, o CLP é um dispositivo projetado para comandar e monitorar equipamentos e processos industriais. Historicamente foram criados para substituir a função exercida por painéis de relés, e ao longo do tempo foram ficando mais complexos, eficientes e mais baratos do que as tecnologias antigas de controle e automação (PRUDENTE, 2011).

Os principais requisitos levantados inicialmente (BRYAN; BRYAN, 1997) e que motivaram a criação dos CLPs foram essencialmente:

- Menores dimensões, comparados aos painéis de relés.
- Flexibilidade de programação, agilizando a alteração da funcionalidade do sistema.
- Redução de manutenção, devido a troca de sistemas eletromecânicos por eletrônicos.
- Redução de preços e custos, como consequência dos fatores acima.

Para se atingir esses objetivos, foram utilizados os microprocessadores e microcontroladores, que já tinham atingido um certo nível de desenvolvimento na época, possibilitando a migração do sistema eletromecânico para o eletrônico e microeletrônico. Apesar de terem evoluído muito em relação aos criados no início dos anos 60, a arquitetura desses dispositivos continua essencialmente a mesma e pode ser dividida basicamente em alimentação, processador, memórias e módulos de entrada e saída. A seguir, na Figura 1, esses componentes são descritos sucintamente.

1. Alimentação

É o circuito de alimentação do sistema, a fonte. Deve prover a tensão e corrente adequados e não permitir que o sistema seja afetado pelas interferências externas vindas do ambiente industrial.

2. Processador

É o componente central do sistema, responsável pela execução do programa criado pelo usuário.

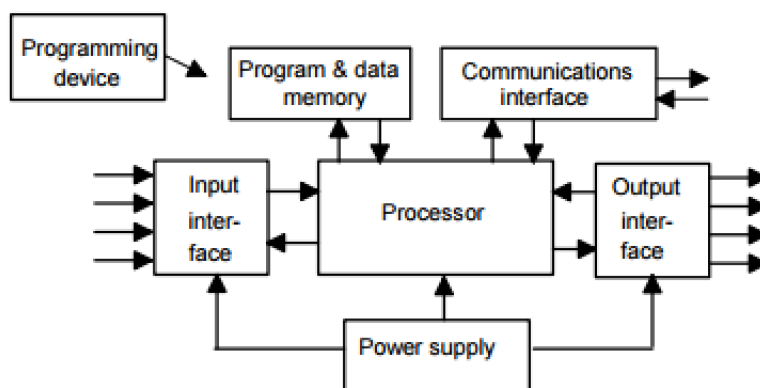


Figura 1 – Componentes básicos de um CLP

Fonte: BOLTON, W. Programmable Logic Controllers, 2006.

3. Memórias

Divididas basicamente em dois tipos principais: voláteis e não-voláteis. As memórias não-voláteis armazenam o código do sistema operacional e do programa do usuário. As memórias voláteis são responsáveis por armazenar os valores correntes das variáveis do processo.

4. Módulos de Entrada e Saída

É o *hardware* que faz a interface entre os sinais elétricos de entrada e saída externos, do processo, e os sinais elétricos internos ao controlador.

5. Interfaces de Comunicação

Interfaces responsáveis por enviar e receber dados nas redes de comunicação de outros CLP's e equipamentos em geral. Embora sua presença não seja essencial, a grande maioria dos CLP's apresenta essas interfaces.

Com essa composição básica foi possível implementar sistemas que atendessem aos requisitos iniciais que motivaram a concepção desses dispositivos, que se tornaram substitutos extremamente viáveis aos painéis de relés (BOLTON, 2005).

Com o surgimento dos primeiros CLPs surgiu também a necessidade de se programar tais dispositivos. Inicialmente a linguagem de programação utilizada era a *Assembly*, porém esta linguagem é intimamente ligada às características internas do *hardware* do processador, o que criava um grande obstáculo para a atualização do sistema, pois toda vez que se precisasse trocar de processador, todo o *software* deveria ser reescrito.

Além disso, um grande requisito da linguagem de programação era que esta fosse simples o suficiente para que os mesmos profissionais responsáveis pelos projetos de circuitos dos painéis de relés fossem capazes de criar programas que fizessem o

mesmo trabalho através dos CLPs, sem necessidade de conhecimento de linguagens mais complexas, como FORTRAN ou C (BRYAN; BRYAN, 1997).

Assim surgiu a linguagem *Ladder*, baseada nos métodos tradicionais de documentação dos antigos circuitos de relés, com o intuito de criar programas simples que manipulam as entradas e saídas digitais do sistema. Essas entradas e saídas são representadas nas formas de contatos, bobinas e blocos funcionais em linhas de circuito horizontais (IEC, 2013). Esses componentes são descritos sucintamente a seguir.

- Contatos

Representam as entradas digitais do sistema, como botoeiras, chaves de fim de curso e sensores de presença, por exemplo. Podem ser contatos normalmente abertos (a), simbolizando lógica direta ou normalmente fechados (b), simbolizando lógica inversa.

- Bobinas

Representam as saídas digitais do sistema, como luzes, bobinas de contadores e alarmes, por exemplo. Podem ser simples (a) ou bobinas de set e reset (b).

- Blocos Funcionais

Representam alguma lógica mais complexa do que simplesmente a leitura ou escrita direta de entradas e saídas. Possuem parâmetros de entrada e saída que utilizam para realizar suas operações. Existem diversos tipos de blocos funcionais, entre os principais deles estão os temporizadores e contadores.

Estes componentes serão discutidos em mais detalhes durante a apresentação do metodologia do trabalho.

Desta forma, como mencionado no início do texto, este trabalho visa construir um protótipo que reproduza a funcionalidade destes dispositivos. Assim, o objetivo geral do projeto é desenvolver um conjunto de máquina virtual e editor de *Ladder* que possa ser implementado em qualquer plataforma, desde computadores a microcontroladores simples. Com esse sistema, será possível desenvolver um protótipo de CLP na placa de desenvolvimento NUCLEO-F767ZI da *STMicroelectronics*.

Para isso deve-se primeiro determinar os aspectos básicos da máquina virtual, que envolvem o formato do *bytecode* com os *opcodes* necessários e seus parâmetros, e o sistema de execução deste código. Então desenvolver um algoritmo de resolução de *Ladder*, e a posterior tradução do diagrama para *bytecode*. Além disso, a máquina virtual deve ser escrita em linguagem C, aplicando-se técnicas de portabilidade. Dessa forma pode ser facilmente adaptada para qualquer sistema computacional.

O editor deve ser construído como uma aplicação *Web*, que funcione em qualquer navegador atual. Deve ocupar o mínimo de memória possível, de modo que possa ser inserido diretamente na memória não-volátil de microcontroladores.

Por fim, o sistema embarcado deve conter um servidor *web*, capaz de entregar o editor quando requisitado pelo navegador, e prover a troca de informações necessárias entre esta aplicação e o dispositivo.

2 Metodologia

2.1 Máquina Virtual

Um CLP deve executar programas do usuário, originalmente escritos em *Ladder*, uma linguagem de alto nível, e que serão traduzidos para uma outra linguagem que possa ser executada pelo seu processador. Para se implementar esse comportamento foram feitas algumas propostas.

Uma opção cogitada inicialmente foi a geração de código de máquina a ser executado diretamente pelo processador escolhido. Um dos *softwares* pesquisados neste trabalho, o *LDmicro*¹, apresenta esta solução, contemplando as arquiteturas PIC e AVR. Essencialmente, como descrito pelo seu próprio desenvolvedor, esta aplicação caracteriza um compilador completo.

Porém esta abordagem causaria uma dependência absoluta da arquitetura específica para qual foi destinada, além de requerer profundo conhecimento sobre a mesma. Ainda, implementações de processos complexos, como blocos funcionais mais elaborados e execuções paralelas, seriam impraticáveis, limitando o futuro aprimoramento do projeto. Portanto este método foi desconsiderado.

Dessa maneira, após a análise da proposta anterior, levantou-se que um dos princípios do projeto é manter a execução independente do hardware utilizado. Para isso, foi implementada uma máquina virtual.

Uma máquina virtual é um *software* que deve ser capaz de reproduzir as funcionalidades de *hardware* de um sistema computacional. Existem diferentes tipos de máquinas virtuais: as máquinas virtuais de sistema, que consistem em substitutos completos de um computador, provendo toda a funcionalidade necessária para executar sistemas operacionais, chegando a usar recursos avançados como a virtualização assistida por *hardware*; e as máquinas virtuais de processo, que são projetadas para executar programas em múltiplas plataformas (SMITH; NAIR, 2005).

No presente projeto utiliza-se uma máquina virtual de processo, ou seja um *software* que deve ser capaz de reproduzir as funcionalidades de *hardware* de um computador, constituindo uma camada de abstração para o acesso a operações nativas do sistema. Esta MV deve executar uma aplicação com código personalizado, independente da plataforma em que foi inserida, o que contempla também a criação e gerenciamento do contexto e das memórias do sistema. Pode-se citar como exemplo deste tipo de MV os

¹ *LDmicro, Ladder Logic for PIC and AVR*. Disponível em: <<http://cq.cx/ladder.pl>>. Acesso em: 01 jul. 2017.

interpretadores Java e Python, e também a plataforma .NET .

Este código elaborado especialmente para a máquina virtual é comumente chamado de *bytecode*. Ele deve ser gerado através da interpretação ou compilação de um código fonte, e deve conter instruções e operandos, comumente denominados de *opcodes*, que são decodificados e executados pela MV (CRAIG, 2006).

As máquinas virtuais de processo podem ser divididas em duas categorias básicas: as MVs de registro e as MVs de pilha. As de registro são implementadas com base no funcionamento físico dos registros (ou registradores) de uma CPU. Por esta razão, o código gerado para estas MVs é similar à linguagem *Assembly*, na qual as instruções devem conter os endereços de todos os operandos. Assim, o *bytecode* gerado para esta máquina virtual costuma ser mais extenso, porém a sua execução é mais eficiente, além de permitir diversas otimizações se comparado com o código gerado para MVs de pilha. São exemplos de MVs de registro as MVs Lua e Dalvik.

As máquinas virtuais de pilha são implementadas com base em uma pilha de dados para processamento. Estas operações se dão através da colocação e retirada de valores na pilha, ou empilhamento e desempilhamento. Durante a execução do programa os dados são desempilhados, processados e o resultado é reempilhado, não sendo necessário o uso de registradores específicos. Desta forma o código gerado é mais extenso e a execução do mesmo é menos otimizada, pois são necessárias as operações de empilhamento e desempilhamento. Porém tanto a implementação deste tipo de MV quanto a geração do seu *bytecode* são mais simples, devido a não necessidade de gerenciar o uso dos registradores. São exemplos de MVs de pilha as MVs Java e .NET (SHI et al., 2005).

Foi decidido que seria mais adequado ao projeto realizar a implementação de uma MV de pilha, devido às vantagens apresentadas, que consistem principalmente na facilidade de desenvolvimento do *firmware* e na maior simplicidade de geração do *bytecode* pelo editor.

Sabe-se que um CLP, do ponto de vista do usuário, possui um certo número de entradas e de saídas, além de memórias booleanas e de palavras (registros de 16 ou 32 bits), entre outros tipos de dados, e a todos esses elementos são atribuídos endereços para que possam ser acessados na composição da *Ladder* (BRYAN; BRYAN, 1997). Dessa maneira, é necessário que a MV contenha vetores associados a cada um desses tipos de dados, e que estes endereços indiquem o vetor correto e posição correta dentro dele, onde está armazenado o dado.

Tradicionalmente os endereços são compostos por duas partes, neste trabalho serão chamadas de prefixo e índice. O prefixo, representado por um caractere, indica o tipo de dado. O índice, representado por um número, indica sua posição no vetor correspondente. Por exemplo, o endereço representado por I3 poderia indicar a quarta posição (pois

mantém-se o primeiro elemento na posição 0) no vetor de entradas (*Input*).

É necessário que em algum momento da execução do programa seja feita a leitura das entradas para a atualização do respectivo vetor. Neste processo, a máquina virtual deve acessar o *hardware* da plataforma de forma a detectar o estado de cada um desses pontos. Por exemplo, quando o sistema for implementado em um microcontrolador, a MV deve conter chamadas a funções de acesso a GPIO (*General Purpose Input/Output*). Da mesma forma deve ser feita a atualização do estado das saídas com os valores contidos no vetor de saídas.

Deve-se realizar o gerenciamento de contexto. O contexto é um conjunto de variáveis que o sistema utiliza para desenvolver a execução de uma tarefa. Existem sistemas que suportam o compartilhamento de uma CPU por múltiplas tarefas. Nesses cenários deve acontecer o salvamento de contexto antes da troca da tarefa em execução, para que posteriormente ela possa ser restaurada e prosseguir. Embora a execução paralela não seja um objetivo do presente projeto, é conveniente que se utilize esta técnica pela simplicidade de desenvolvimento, e também visando possíveis aprimoramentos no futuro. A implementação do gerenciamento será apresentada posteriormente.

Estabelecidos estes conceitos básicos, inicia-se a concepção do código que representa a *Ladder*, e que será executado pela MV. Neste processo deve-se evidenciar o algoritmo utilizado para resolução da *ladder*, bem como os recursos criados como auxílio para o processamento e o formato do código final.

A primeira etapa foi o desenvolvimento de um conjunto básico de *opcodes* necessário para a resolução de um programa em *Ladder*. Opcodes são as instruções que a máquina virtual executará. Podem ser representados por mnemônicos, que são nomes dados a cada um, e também pelos valores numéricos, hexadecimais ou binários. Destaca-se que *opcode* se refere à instrução em si, independente da forma em que é apresentada.

Cada *opcode* pode ser acompanhado por operandos, que fornecem informações a serem processadas. É necessário que cada *opcode* tenha um número fixo de operandos e que a máquina virtual seja capaz de reconhecê-los. Como será mostrado posteriormente na implementação, estes operandos serão utilizados em pares, separados por vírgula. Por convenção, neste trabalho nos referimos a cada um destes pares de operandos pelo termo *parâmetro*, porém sem a intenção de entrar no mérito formal destas nomenclaturas.

Conforme introduzido anteriormente, a MV opera através de pilhas (ou *stacks*). As pilhas são filas (ou *buffers*) que armazenam dados no esquema LIFO (*Last In, First Out*), ou seja, o último dado a entrar é o primeiro a sair. Os dados gerados pela execução do programa são empilhados e desempilhados durante a execução do programa conforme a função dos *opcodes*.

A seguir apresenta-se alguns exemplos de *Ladder*, seguidos da sua tradução em

opcodes e da ilustração da execução de cada linha com o conteúdo das pilhas naquele momento.

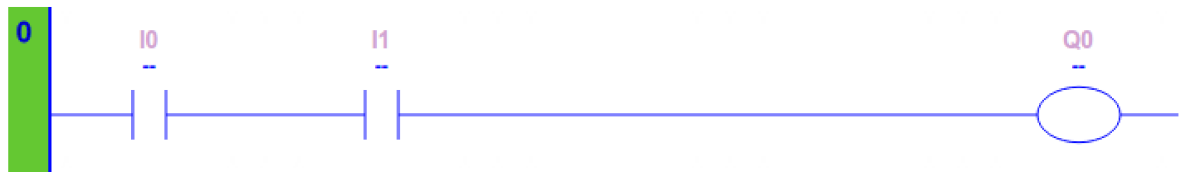


Figura 2 – *Ladder* com elementos em série

Pela *Ladder* apresentada na Figura 2, sabemos que para que a saída Q1 ser energizada precisa-se ativar I0 e I1, simultaneamente. Portanto, a tradução desta *rung* em *opcodes*, com as linhas numeradas, fica da seguinte forma:

```

1  LDR I ,00
2  LDR I ,01
3  AND
4  STR Q,00

```

A Tabela 1 mostra o conteúdo da pilha (preenchida de cima para baixo) após a execução de cada linha do código acima. O endereço dos elementos é utilizado para simbolizar o estado em que se encontrava no momento da execução, ressalta-se que no caso real esse estado deve ser 1 ou 0, ou seja, energizado ou não energizado.

Linha	1	2	3	4
Pilha	I0	I1	I0 . I1	-
	-	I0	-	-
	-	-	-	-

Tabela 1 – Utilização da pilha

Na primeira linha carrega-se o estado do contato associado ao endereço I0 (instrução LDR, com o parâmetro I, *offset* 00), que se encontra na primeira posição do vetor de entradas. O valor carregado é empilhado, dessa forma, após a execução dessa instrução a pilha contém apenas o estado de I0, na primeira posição.

Na segunda linha, analogamente, empilha-se o estado do contato I1. Após essa execução a pilha contém o estado de I0 na primeira posição e o estado de I1 na segunda posição.

Na terceira linha, a instrução AND retira os dois valores da pilha, reempilhando o resultado da operação lógica E, representada pelo operador ponto (.) na tabela. Portanto a pilha contém novamente um único valor após a execução dessa instrução.

Por fim, na quarta linha, a instrução **STR** retira esse valor da pilha e o armazena na primeira posição do vetor de saídas (indicado pelo prefixo **Q**, offset **01**).

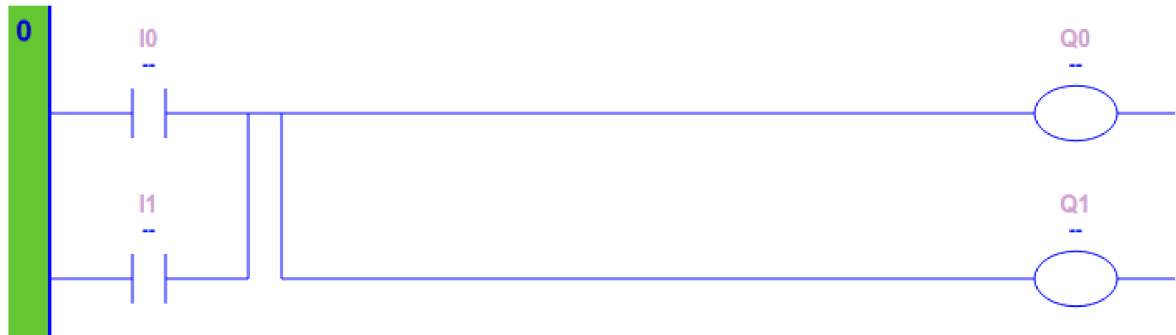


Figura 3 – *Ladder* com elementos em paralelo

No exemplo da Figura 3, observa-se que as saídas **Q0** e **Q1** serão energizadas se qualquer uma das entradas, **I0** e **I1**, forem acionadas. Esta *Ladder* pode ser traduzida nos seguintes *opcodes*:

```

1  LDR I,00
2  LDR I,01
3  OR
4  PUSH_TMP
5  STR Q,00
6  GET_TMP
7  STR Q,01
8  POP_TMP

```

Para solucionar essa nova configuração de *Ladder* foi necessário adicionar uma segunda pilha, chamada de temporária. Isso se deve ao fato de que a instrução **STR** consome um valor da pilha sem devolvê-lo, retirando a informação necessária para atualizar o próximo elemento final em paralelo. Assim, seguindo-se o mesmo raciocínio anterior, a Tabela 2 apresenta duas pilhas, a principal (P) do lado esquerdo, e a temporária (T), do lado direito, para cada linha de código.

Linha		1		2		3		4		5		6		7		8	
P	T	I0	-	I1	-	I0+I1	-	I0+I1	I0+I1	-	I0+I1	I0+I1	I0+I1	-	I0+I1	-	-
		-	-	I0	-		-	-	-	-	-	-	-	-	-	-	-
		-	-	-	-		-	-	-	-	-	-	-	-	-	-	-

Tabela 2 – Introdução da pilha temporária

A execução é muito similar à do exemplo anterior, portanto a seguir serão abordadas as principais diferenças. Na terceira linha ocorre a instrução **OR**, que executa

uma operação lógica OU, representada na tabela pelo operador +. Da mesma forma como a instrução AND, desempilha dois valores e empilha novamente o resultado da operação.

Na quarta linha ocorre a instrução PUSH_TMP, cuja função é copiar o valor do topo da pilha principal e inseri-lo na pilha temporária. Assim a MV pode executar a próxima instrução, STR, que deixará vazia a pilha principal.

Na sexta linha ocorre a instrução GET_TMP, cuja função é copiar o valor do topo da pilha temporária e inseri-lo na pilha principal. Dessa forma, ao final dessa execução as duas pilhas têm exatamente o mesmo conteúdo, ou seja, foi restaurado o estado anterior da pilha principal. Dessa forma a informação necessária para atualizar o estado de Q1 está disponível novamente para o consumo pela subsequente instrução STR.

Na oitava linha ocorre a instrução POP_TMP, cuja função é remover o valor do topo da pilha temporária, para que ao final da execução desta *rung* as pilhas estejam vazias.

Com as instruções demonstradas até agora é possível implementar a execução de qualquer *rung* que contenha contatos normalmente abertos e bobinas. Para que seja feita a execução de uma *Ladder* composta por múltiplas rungs, basta que se coloque os *opcodes* de cada uma em sequência, pois foi garantido que as pilhas estarão vazias ao final de cada *rung*.

Por fim deve-se definir instruções que marquem o início e o fim de um ciclo de *scan*. Serão as instruções BEGIN, para a inicialização e leitura das entradas e END, para a escrita das saídas e finalização do ciclo de *scan*.

Foi definido assim um conjunto mínimo de *opcodes* que serão utilizados para compor o *bytecode*: LDR, STR, AND, OR, PUSH_TMP, GET_TMP, POP_TMP, BEGIN e END. Com eles foi possível implementar uma primeira versão da máquina virtual que execute as funcionalidades descritas acima.

Para a finalização desta etapa, basta realizar a substituição dos mnemônicos pelo valor numérico representado, na ordem em que foram gerados. A seguir estipulam-se alguns valores de exemplo e se demonstra o processo.

BEGIN	0x01	PUSH_TMP	0x07
END	0x02	GET_TMP	0x08
LDR	0x03	POP_TMP	0x09
STR	0x04	I	0x01
AND	0x05	Q	0x02
OR	0x06		

Tabela 3 – Primeiros *opcodes*

Na Tabela 3 foram atribuídos valores hexadecimais a cada *opcode*. Serão eles que indicarão à MV qual função correspondente deve executar. Também foram atribuídos valores aos prefixos de endereço, que são o primeiro parâmetro das instruções LDR e STR.

Para finalidade de demonstração, as instruções **BEGIN** e **END** coincidem numericamente com os prefixos **I** e **Q**. Deve-se evidenciar que isso não acarreta nenhum problema na execução da MV, pois é garantido um mecanismo que permite diferenciar *opcodes* de parâmetros. Este mecanismo, o contador de programa, é implementado através do contexto da MV.

A seguir apresenta-se um trecho de código, supostamente gerado através do método já descrito, no qual a frente de cada linha foi apresentada a conversão dos mnemônicos em valores numéricos, segundo a Tabela 3.

BEGIN	0x01
LDR I,03	0x03 0x01 0x03
LDR I,05	0x03 0x01 0x05
OR	0x06
PUSH_TMP	0x07
STR Q,00	0x04 0x02 0x00
GET_TMP	0x08
LDR I,12	0x03 0x01 0x0C
AND	0x05
STR Q,10	0x04 0x02 0x0A
POP_TMP	0x07
END	0x02

Para realizar a conversão, basta substituir cada mnemônico pelo seu equivalente numérico, de cima para baixo e da esquerda para direita. Ressalta-se que os valores numéricos encontrados nos parâmetros, são copiados, bastando efetuar a mudança de base decimal para hexadecimal. Assim, o bytecode obtido, apresentado em formato de vetor na sintaxe da linguagem C, fica da seguinte forma:

```
uint8_t code[22] = {
    0x01, 0x03, 0x01, 0x03, 0x03, 0x01, 0x05, 0x06,
    0x07, 0x04, 0x02, 0x00, 0x08, 0x03, 0x01, 0x0C,
    0x05, 0x04, 0x02, 0x0A, 0x07, 0x02
};
```

Assim foram definidos os conceitos básicos necessários para o desenvolvimento da máquina virtual. Seguindo-se o mesmo raciocínio apresentado, pode-se adicionar suporte a mais elementos de *Ladder*, como contatos normalmente fechados, bobinas de *Set* e *Reset* e blocos funcionais, bastando apenas a criação de novos *opcodes*. Posteriormente, na seção 3.3, são apresentados todos os elementos e seus *opcodes*.

2.2 Editor de *Ladder*

Como foi indicado nos objetivos do projeto, existe a necessidade de uma interface de programação *Ladder* pequena, simples e eficiente, que anule a necessidade de instalação e manutenção de *software* especializado no computador do usuário. Dessa forma, essa interface deve estar embarcada no dispositivo.

Uma possibilidade de solução, comumente encontrada em mini CLP's disponíveis no mercado, como a linha LOGO da Siemens² e Zelio da Schneider Electric³, é a utilização de *displays* e teclas de comando. Essa opção se mostra inviável para o projeto pois além de acarretar grande aumento de custo do *hardware*, essas interfaces são pouco intuitivas e pobres em termos de experiência de usuário.

Determinou-se então que uma interface que fosse intuitiva e cômoda do ponto de vista do usuário seria possível através de uma aplicação *web*. Aplicações *web* são *softwares* utilizados através dos navegadores *web*. Devido ao atual estado da arte dessa área, pode-se criar interfaces gráficas bem elaboradas e com alto nível de interatividade, propiciando uma experiência de usuário muito rica.

Assim eliminaria-se a necessidade de *software* adicional pois qualquer sistema operacional comum inclui um navegador capaz de executar tais aplicações. Além disso atualmente a utilização de um *browser* é muito familiar a qualquer usuário em potencial do sistema, conferindo intuitividade inerente à aplicação.

Portanto foi determinado que o editor deve apresentar uma interface simples e intuitiva para a edição de *Ladder* e ser compatível com a maioria dos navegadores em uso hoje em dia. Seu tamanho final em memória não deve ultrapassar a ordem de algumas dezenas de *kilobytes*, para que seja possível armazená-lo na memória não-volátil interna de microcontroladores. Refere-se ao tamanho da aplicação ocupado em memória como *footprint*. Estabeleceu-se então o valor máximo de 128 *kilobytes*, já que um valor de memória de programa encontrado comumente nestes *chips* é de 256 *kilobytes*.

2.2.1 Estudo de Viabilidade Técnica

Desta forma iniciou-se um estudo de viabilidade técnica para determinar se seria possível implementar um editor que atendesse a esses requisitos.

Sabe-se que aplicações Web, como citado anteriormente, são aplicações cliente-servidor na qual o lado cliente (a interface de usuário) é executado em um navegador *web*, que se comunica com o lado servidor através do protocolo HTTP (GOURLEY;

² LOGO! Overview. Disponível em: <<http://w3.siemens.com/mcms/programmable-logic-controller/en/logic-module-logo/overview/pages/default.aspx>>. Acesso em: 01 jul. 2017.

³ Zelio Logic. Disponível em: <<http://www.schneider-electric.com/en/product-range/531-zelio-logic/>>. Acesso em: 01 jul. 2017.

(TOTTY, 2009). Estas aplicações comumente são compostas por três linguagens: HTML, CSS e Javascript. Em uma descrição básica, podemos dizer que a linguagem HTML é responsável pela estruturação da página web, a CSS pelo ajuste gráfico dos elementos da página, e o Javascript pela sua funcionalidade (ROBBINS, 2012).

Assim, uma aplicação *Web* contempla tipicamente três tipos de arquivos que contém o texto referente à programação de cada linguagem. Esses arquivos são enviados ao *browser* que os interpreta, gerando a visualização da página para o usuário. Um excelente recurso que todos os navegadores suportam atualmente é a recepção de arquivos compactados no formato *gzip*. Desse modo os arquivos podem ser compactados antes da gravação na memória do dispositivo, reduzindo significativamente o *footprint* do editor (KOZUCH; WOLFE, 2002).

Como parte do estudo de viabilidade técnica foi feita uma busca a aplicações similares à que se pretendia desenvolver. Foram encontrados diversos *softwares* relacionados ao desenvolvimento do editor, dentre eles, o *MBLogic*⁴ foi escolhido para um estudo mais aprofundado.

O *MBLogic* é uma coleção de pacotes de *software* livre e de código aberto usados para controle de processos industriais. Entre esses pacotes existe um editor de *Ladder* por aplicação *web*, que provê os elementos básicos que foram pensados para o presente projeto. Através da análise da demonstração deste editor, pode-se observar que o tamanho final da aplicação completa girava em torno de 300 kB, somando-se os arquivos .html, .js e .css, não comprimidos. Aplicando-se a compressão através do *gzip*, os arquivos passaram a ocupar cerca de 60 kB.

Com esse estudo pôde-se concluir que a implementação do editor seria factível e teria grande potencial de atender o requisito de *footprint*. Posteriormente foi feito um planejamento inicial da programação do editor propriamente dita, e sugeriu-se que o mesmo tivesse os seguintes componentes:

- Suporte para gráficos:

Duas opções se mostraram viáveis para a implementação da *interface* gráfica da *Ladder*: utilizando-se arquivos de imagem comprimidos, como no caso do editor do *MBLogic*, e utilizando-se gráficos vetorizados.

No primeiro caso basta que se criem os arquivos e se inclua na aplicação. Esses arquivos são comprimidos, como o formato JPG, e sua recompressão através do *gzip* não produz redução significativa do espaço ocupado em memória. Além disso, para cada símbolo diferente, necessita-se de um arquivo diferente, aumentando o *footprint*, o que poderia vir a inviabilizar a expansão do editor.

⁴ *MBLogic: for an open world in automation* Disponível em: <<http://mblogic.sourceforge.net>>. Acesso em: 01 jul. 2017.

Ao se utilizar gráficos vetorizados as imagens são criadas por código. Existe a necessidade de se utilizar uma biblioteca para isso, o que implica em mais espaço ocupado em memória. Mas, em contrapartida, pode-se reutilizar código para se criar imagens de símbolos similares, contribuindo para a previsibilidade do *footprint*.

- Interface de usuário:

O usuário deverá interagir com a aplicação como com qualquer editor tradicional de *Ladder*, através de imagens, botões e atalhos do teclado. Deverá ter suporte para salvamento e carregamento de programas criados pelo usuário. Funcionalidades intrínsecas do próprio Javascript.

- Comunicação com o dispositivo:

Deve existir uma forma de se trocar dados entre o *browser* e o dispositivo, para que se possa enviar o código gerado, requisitar o código gravado e efetuar comandos, como Run e Stop, por exemplo. Para isso deve-se utilizar os métodos GET e POST do protocolo HTTP.

2.2.2 Estrutura de Dados

Um dos pontos chave da implementação do editor, tão importante quanto o desenvolvimento do interface gráfica, é a representação da *Ladder* em memória para posterior interpretação e tradução da mesma em *bytecode* a ser executado pelo dispositivo. Através do estudo do código fonte do *software LDmicro* percebeu-se um possível caminho para a realização desse processo.

O *LDmicro* apresenta um editor em console para o sistema operacional Windows, porém este *software* faz a tradução da *Ladder* para um código em linguagem C, e finaliza com a compilação para microcontroladores de arquitetura PIC ou AVR. Apesar de esse não ser o objetivo do presente projeto, o método inicial utilizado pelo *software* resolve a primeira parte do problema. Através da interpretação da *Ladder* é criada uma estrutura de dados que posiciona os contatos e bobinas dentro de nós paralelos e séries, que representam a associação destes elementos no circuito. À frente do nome do elemento são armazenados os parâmetros deste, como endereço, tipo e outros dados necessários para a execução do elemento. Tomando-se por base inicial essa representação, pôde-se implementar o editor planejado.

Dessa forma, foi determinado que existem dois tipos de elementos: os elementos finais e os intermediários. Os elementos finais são aqueles desenhados no final de cada *rung*, na extrema direita do circuito, ou seja, não existe nenhum elemento em série após um elemento final. Estes elementos irão somente consumir informações da pilha, sem colocar nada de volta nela. Os elementos intermediários são aqueles inseridos antes dos elementos finais. Podem conter outros elementos em série ou em paralelo, antes ou após

os mesmos. Eles consomem dados da pilha, mas geram novos dados a serem empilhados, como abordado na seção 2.1.

Na Figura 4 apresenta-se um exemplo de *ladder* e ilustra-se sua representação na estrutura de dados idealizada.

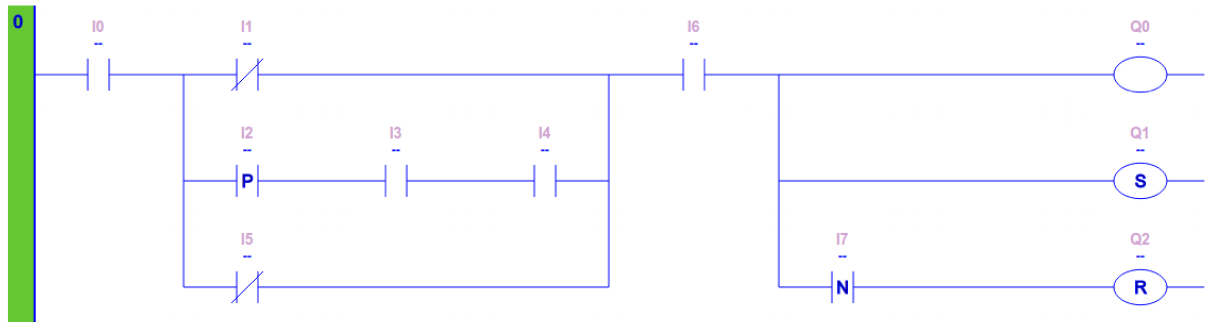


Figura 4 – Exemplo para estrutura de dados

```

INTERMED (NO , I0)
PARALLEL
|
|   INTERMED (NC , I1)
|   SERIES
|   |   INTERMED (P , I2)
|   |   INTERMED (NO , I3)
|   |   INTERMED (NO , I4)
|   END
|   INTERMED (NC , I5)
END
INTERMED (NO , I6)
PARALLEL
|   FINAL (OUT , Q0, ???)
|   FINAL (SET , Q1, B0)
|   SERIES
|   |   INTERMED (N , I7)
|   |   FINAL (RST , Q2, B1)
|   END
END

```

Percebe-se que o nó raiz, que contém a *rung* toda, é um nó do tipo série, porém essa informação não é incluída na estrutura de dados, ficando implícita. O primeiro elemento é um elemento intermediário, um contato normalmente aberto, representado pelo nome **INTERMED**, e recebe como parâmetros dentro dos parênteses o tipo do elemento,

NO (Normally Open - Normalmente Aberto), e o endereço do mesmo, **I0**, que é composto do índice **I**, vetor de entradas, e do *offset* **0**, que indica a posição de memória referente no vetor referente a este elemento.

O segundo elemento é um nó paralelo, como indicado pelo nome **PARALLEL**. No interior deste nó, representado pela indentação, existem os diversos elementos que estarão associados em paralelo, chamados filhos. O primeiro deles é um contato normalmente fechado, no endereço **I1**, assim como o terceiro, no endereço **I5**. O segundo elemento é um nó do tipo série. Indicado pelo nome **SERIES**. Analogamente, os elementos contidos no interior deste nó estão associados em série.

Continuando a análise do exemplo, pode-se perceber que no segundo nó paralelo existem elementos finais, bobinas, indicados pelo nome **FINAL**. Ressalta-se que esses elementos só podem existir em paralelo com outros finais ou com nós do tipo série que contenham um elemento final na última posição.

Assim, foi possível implementar essa estrutura em Javascript na forma de vetores aninhados, com cada elemento sendo representado por um vetor. Na primeira posição do vetor armazena-se uma *string* com o nome do elemento, e na segunda posição outro vetor, contendo em cada posição um parâmetro. No caso dos nós paralelos e série, na segunda posição armazena-se um vetor cujas posições contém cada vetor de cada elemento filho dentro do nó.

A a seguir mostra-se a estrutura referente ao exemplo anterior na forma de vetores aninhados, na sintaxe correta utilizada na programação em Javascript, com a mesma indentação da ilustração para melhor visualização.

```

[
  ["INTERMED" , ["NO" ,[" I0 "]]] ,
  ["PARALLEL" ,[
    ["INTERMED" ,["NC" ,[" I1 "]]] ,
    ["SERIES" ,[
      ["INTERMED" ,["P" ,[" I2 "]]] ,
      ["INTERMED" ,["NO" ,[" I3 "]]] ,
      ["INTERMED" ,["NO" ,[" I4 "]]]
    ]] ,
    ["INTERMED" ,["NC" ,[" I5 "]]]
  ]] ,
  ["INTERMED" ,["NO" ,[" I6 "]]] ,
  ["PARALLEL" ,[
    ["FINAL" ,["OUT" ,["Q0" ,"???" ]]] ,
    ["FINAL" ,["SET" ,["Q1" ,"B0" ]]] ,
    ["SERIES" ,[
      ["INTERMED" ,["N" ,[" I7 "]]] ,
      ["FINAL" ,["RST" ,["Q2" ,"B1" ]]]
    ]]
  ]]
]

```

Essa abordagem facilitou a realização das funções de edição, pois o problema se reduziu em adicionar, remover e realocar os vetores dentro da estrutura. O Javascript possui suporte nativo a essas operações através de diversas funções membros do objeto *Array*, destacando-se principalmente a função `splice` (FLANAGAN, 2011).

O método de edição foi desenvolvido com base nesse sistema. A forma mais simples de se adicionar um novo elemento na *Ladder* é selecionar um elemento existente e requisitar a adição numa posição relativa a ele. Existem quatro posições relativas: em série antes, em série após, em paralelo acima e em paralelo abaixo. Como afirmado anteriormente, não é permitido adicionar nada em série após elementos finais e somente elementos finais (ou nós série que terminem em elementos finais) podem estar em paralelo com outros elementos deste tipo.

No caso da requisição da adição em série, se o nó pai do elemento for do tipo série, basta adicionar o elemento na posição adequada. Caso o nó pai for do tipo paralelo, então deve-se remover o elemento selecionado, criar um nó do tipo série em seu lugar e então reinserir o elemento selecionado e o adicionado como filhos deste, na ordem adequada. O raciocínio é análogo no caso da adição de elementos em paralelo dentro de nós do tipo série.

Para se remover um elemento da *Ladder*, basta remover o vetor correspondente da estrutura. Porém deve-se cuidar para que isso não gere nós série ou paralelo com um único filho. Nessa ocasião deve-se retirar esse elemento restante, remover o nó pai, e reinserir o elemento em seu lugar.

Percebe-se que como a *rung* é representada por um vetor contendo os vários vetores aninhados, para adicionar suporte a mais *rungs* basta criar um nível superior na estrutura de dados, ou seja, um vetor que armazenará cada *rung*. Desse modo as mesmas funções de manipulação de vetores podem ser utilizadas para a edição de *rungs* na *Ladder*.

Com a definição deste sistema, foi possível realizar a implementação da interface de edição da *Ladder* com as funcionalidades básicas previstas de adição e remoção de elementos em série e em paralelo, além da criação, remoção e reposicionamento das *rungs*.

A próxima etapa a ser realizada é a geração do *bytecode*. É conveniente que se divida este processo em duas partes: a geração dos *opcodes* em forma de mnemônicos, para que se possa verificar mais facilmente o resultado, e por fim a geração da sequência de hexadecimais. Utiliza-se a seguinte estrutura para o próximo exemplo.

```

INTERMED (NO , I0)
PARALLEL
|      INTERMED (NO , I1)
|      INTERMED (NO , I2)
END
PARALLEL
|      FINAL (OUT , Q0)
|      SERIES
|      |      INTERMED (NO , I3)
|      |      FINAL (OUT , Q1)
|      END
END

```

Através da leitura sequencial da estrutura de dados é possível, por meio de um algoritmo, gerar os *opcodes* e seus parâmetros. Usando-se os mnemônicos definidos anteriormente na seção 2.1, ilustra-se o processo a seguir.

Todo programa deve começar com a instrução **BEGIN**, portanto é a primeira a ser inserida. Logo depois lê-se o primeiro elemento da estrutura, que é um intermediário do tipo **NO** (contato normalmente aberto). Deve-se então inserir o *opcode* **LDR**, cujos parâmetros podem ser inferidos pelo endereço indicado logo após o tipo, **I0**. Portanto indica-se no primeiro parâmetro o prefixo **I**, e no segundo o índice **0**. Neste momento tem-se o código:

LDR I,00

O segundo elemento é um nó paralelo. Diz-se então que se desceu um nível na estrutura, representado pela indentação. Deve-se implementar um mecanismo que acompanhe em que nível se encontra a leitura, para que se façam as operações adequadas na pilha.

Sabendo que se está no segundo nível e que nele não foi carregado nenhum valor na pilha, carregam-se os dois contatos subsequentes da mesma forma em que se carregou o primeiro. Como se está dentro de um nó paralelo e existem dois valores empilhados neste nível, deve-se executar um **OR**. Os *opcodes* gerados até então são:

LDR I,00

LDR I,01

LDR I,02

OR

Seguindo a leitura percebe-se que se terminou o conteúdo do paralelo e volta-se ao primeiro nível da estrutura. Agora existem dois valores empilhados referentes a este nível: o primeiro contato e o valor resultante do paralelo. Sendo assim, com dois valores na pilha, realiza-se um **AND**, pois o primeiro nível é considerado conteúdo de um nó série.

LDR I,00

LDR I,01

LDR I,02

OR

AND

O próximo elemento é outro nó paralelo. Observa-se que no seu conteúdo existem elementos finais, portanto deve-se utilizar a pilha temporária, como explicado na seção 2.1. Assim, ao se entrar neste nó executa-se um **PUSH_TMP**, empilhando na pilha temporária o valor do topo da principal. O primeiro filho é um elemento final do tipo **OUT**, analogamente aos contatos, executa-se um **STR**, com os devidos parâmetros. Destaca-se que neste caso ocorre um desempilhamento, deixando a pilha principal vazia.

LDR I,00

LDR I,01

LDR I,02

OR

AND

STR Q,00

A cada novo elemento desse nível deve se executar um **GET_TMP**, empilhando na

pilha principal o valor do topo da temporária. O segundo elemento desse nível é um nó do tipo série, que contém primeiramente um intermediário carregado através de um LDR. Neste momento, apesar de que só tenha sido empilhado um valor neste terceiro nível, a pilha principal contém dois, e deve-se executar um AND, pois o próximo elemento é um final, gerando um STR.

Prosseguindo, deve-se executar um POP_TMP, pois terminou-se o nó paralelo que continha os elementos finais. Como a estrutura chegou ao fim e neste exemplo só existe esta *rung*, finaliza-se o programa com um END. Ao executar todos esses passos, obtém-se o código completo.

```
LDR I,00
LDR I,01
LDR I,02
OR
AND
PUSH_TMP
STR Q,00
GET_TMP
LDR I,03
AND
STR Q,01
POP_TMP
```

Analogamente, para implementar o suporte a blocos funcionais, basta adicionar novos *opcodes* para cada um deles. Blocos funcionais contêm diversas entradas e saídas, portanto pode ser necessário que estes *opcodes* tenham diversos parâmetros. Desta forma, cada novo parâmetro deve ser separado por um espaço. O trecho a seguir, uma instrução de um bloco contador (explicado na seção 3.3), ilustra este conceito.

```
CTU B,01 M,02 W,03 W,04
```

Com isso descreveu-se o algoritmo básico utilizado para a geração dos *opcodes* através da estrutura de dados que representa a *Ladder*. O único passo restante será a substituição dos mnemônicos pelos valores hexadecimais para composição do *bytecode* a ser enviado para a MV. Este processo já foi abordado na seção 2.1.

3 Desenvolvimento

Neste capítulo descreve-se o processo de implementação do editor e do protótipo do CLP na placa de desenvolvimento NUCLEO-F767ZI. São apresentadas as ferramentas utilizadas, bem como evidenciados os problemas encontrados e as funcionalidades extras adicionadas. Inicia-se descrevendo a criação do editor, prosseguindo com a implementação da máquina virtual e a integração destas duas partes com o servidor web.

3.1 Editor de Ladder

O processo de desenvolvimento do editor começou com a criação do sistema de desenho da *ladder*. Como abordado anteriormente na seção 2.2, foi escolhido utilizar o formato de imagem vetorizado SVG para compor os gráficos. Aplicações *web* têm suporte nativo à renderização dessas imagens e à criação das mesmas através de código em Javascript (BELLAMY-ROYDS; CAGLE, 2017). Porém a manipulação destes gráficos será muito frequente e intensa durante a elaboração do editor, e o uso destas funções nativas se resume a chamadas do DOM, uma interface voltada para o acesso às estruturas do HTML através dos *scripts*.

Portanto é conveniente usar uma biblioteca que facilite estas operações. Foi feita uma pesquisa e surgiram algumas opções, entre elas as bibliotecas SVG.js¹, Snap.svg², Raphael³, e Vanilla JS⁴. O quesito principal de seleção foi o tamanho final da biblioteca quando comprimida em formato *gzip*, tendo em vista a *footprint* da aplicação. A Tabela 4 mostra os tamanhos dessas bibliotecas, que justifica a escolha da SVG.js para o projeto.

SVG.js	Snap.svg	Raphael	Vanilla JS
18,3 kB	28,6 kB	31,8 kB	25,0 kB

Tabela 4 – Tamanho das bibliotecas SVG

O HTML deve ser estruturado de forma a acomodar as diversas áreas da aplicação. Sabe-se que será necessário dividir em três áreas: a barra de ferramentas, que conterá os botões de edição e ficará fixa no topo da página; o corpo, que conterá a *Ladder*, situado no centro da página; e o console, fixo no rodapé, que mostrará informações para auxiliar no desenvolvimento, como a estrutura de dados, os *opcodes* gerados, etc.

¹ *SVG.js*, *The lightweight library for manipulating and animating SVG*. Disponível em: <<http://svgjs.com/>>. Acesso em: 01 jul. 2017.

² *Snap.svg* *The JavaScript SVG library for the modern web*. Disponível em: <<http://snapsvg.io/>>. Acesso em: 01 jul. 2017.

³ *Raphaël—JavaScript Library*. Disponível em: <<http://dmitrybaranovskiy.github.io/raphael/>>. Acesso em: 01 jul. 2017.

⁴ *Vanilla JS*. Disponível em: <<http://vanilla-js.com/>>. Acesso em: 01 jul. 2017.

Foi usada a técnica básica do HTML, que é a criação de tags do tipo *div* para demarcar cada conteúdo citado acima. No centro, cada *rung* criada dinamicamente por Javascript terá sua própria *div*, com ID único, para fácil acesso através das funções do DOM. Desta forma ficam demarcadas as *tags* principais da página HTML.

Os *scripts* foram organizados segundo papel que desempenham. Pode-se dividi-los em quatro grandes áreas:

- Gráficos

Inclui as funções de desenho da *Ladder* e de interface de usuário.

- Edição

Inclui a manipulação da estrutura de dados, com a adição e remoção de elementos e *rungs*. Também abrange as funções de configuração dos elementos, como a atribuição de endereços e parâmetros.

- Dados

Contempla um pequeno banco de dados, que armazena os objetos que contém as informações da *Ladder* e provê acesso a elas, e também auxilia na criação de novos elementos. Possui as funções de conversão dos dados em JSON, salvamento e carregamento de arquivos.

- Geração de código

Abrange a geração dos *opcodes* e do *bytecode*, além de testes de uso da pilha.

A estratégia comum dos editores para desenhar a *Ladder* é criar uma grade retangular, com um certo número de elementos na horizontal e na vertical. Em cada posição são desenhados os elementos e entre eles as linhas que os interligam, evidenciando quais estão em série e quais estão em paralelo.

Na implementação, esta grade foi projetada de maneira que contivesse uma divisão para o símbolo do elemento e outras duas para as conexões entre os elementos, uma do lado esquerdo e outra do lado direito. A Figura 5 mostra uma grade de 2x2 elementos.

A abordagem utilizada no projeto segue esta linha de raciocínio. Como na geração do *bytecode*, realizou-se a leitura da estrutura de dados, elemento a elemento. Durante essa leitura, passo a passo, os símbolos são criados e posicionados nos seus devidos lugares, e os espaços vazios entre eles são preenchidos com as linhas.

O fluxograma da Figura 6 ilustra o algoritmo seguido para realizar o posicionamento dos símbolos na *Ladder*. A posição onde será desenhado o elemento é expressa em linhas e colunas, como em uma matriz. Inicia-se na primeira linha e primeira coluna, lendo-se os elementos do primeiro nível da estrutura de dados.



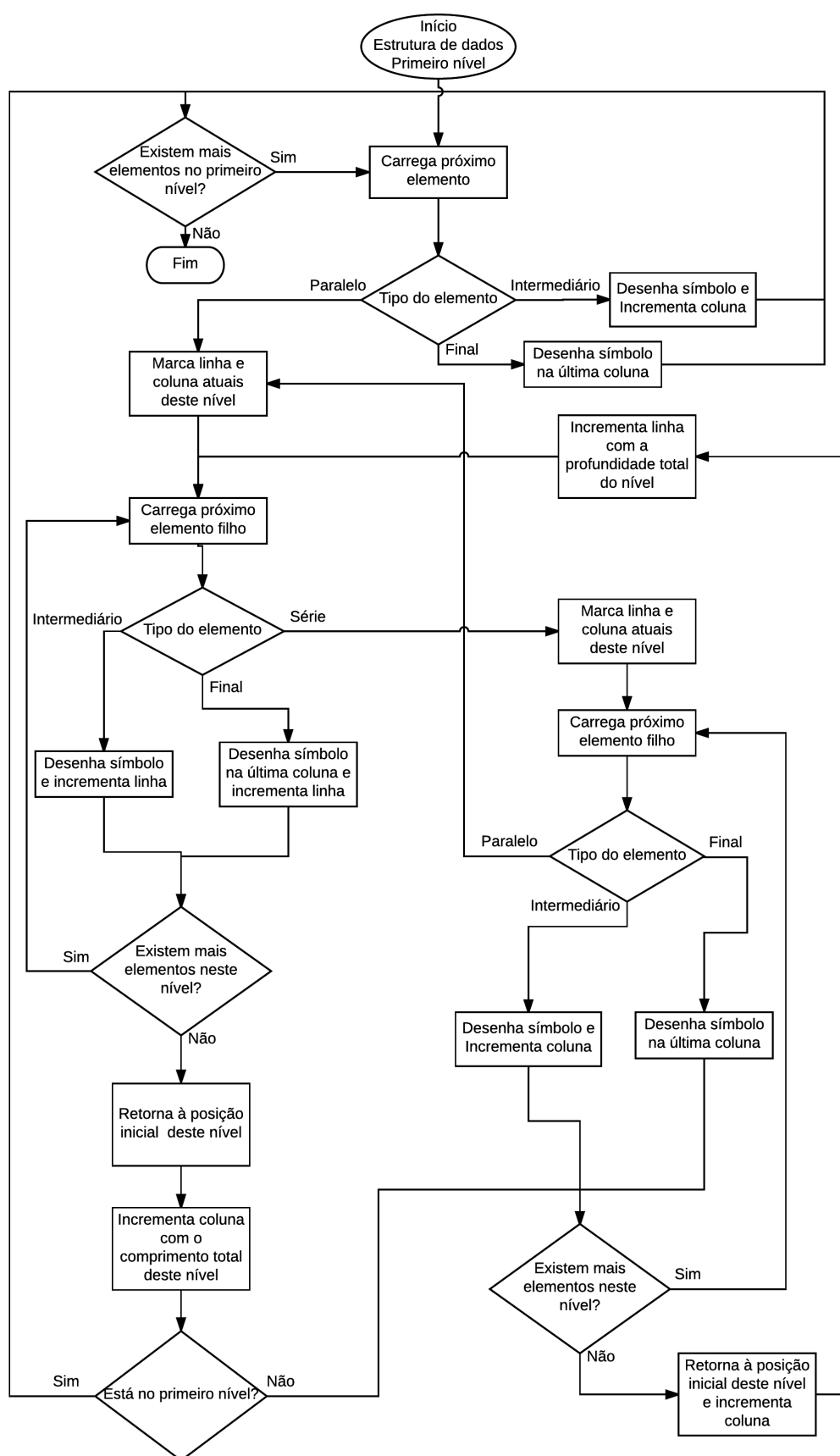
Figura 5 – Exemplo de grade

No primeiro nível, o nível zero, como já mencionado, tudo está em série. Portanto toda vez que um elemento intermediário for encontrado neste nível, deve-se desenhá-lo na posição atual e incrementar, ou seja, somar um à coluna. Desta forma, o próximo elemento deverá ser desenhado na coluna a frente. Caso um elemento final seja encontrado no primeiro nível, deve-se apenas desenhá-lo na última coluna, pois nada deverá existir após ele. Como toda *Ladder* gerada pelo editor é válida, garante-se que este elemento é o último da estrutura e que o desenho será finalizado.

Caso um nó paralelo seja encontrado, deve-se registrar a linha e coluna atuais e relacioná-las com o nível um. A partir de então segue-se o mesmo processo de leitura dos elementos contínuos neste paralelo, porém, quando encontrado um intermediário, deve-se desenhá-lo na posição atual e incrementar o valor da linha. Desta forma, a próxima posição passa a ser a inferior. Da mesma forma, elementos finais são desenhados na última coluna, porém agora deve-se também incrementar a linha, pois podem existir outros elementos finais em paralelo.

Um nó paralelo pode conter nós série, assim como nós série (e o nó raiz) podem conter nós paralelo, assim não devem existir dois nós do mesmo tipo aninhados. Os nós série seguem a mesma lógica dos paralelo, com a diferença de que após se desenhar um intermediário, deve-se incrementar a coluna.

É importante notar que ao se sair de um nó paralelo ou série, quando acabam-se os elementos contidos nele, deve-se retornar à posição inicial daquele nó, ou seja, deve-se retornar à linha e coluna do primeiro elemento do nó. Então deve-se somar à linha ou coluna, dependendo do tipo do nó, um valor suficiente para ultrapassar todo o comprimento ou profundidade (altura) do conteúdo deste nó. A Figura 7 ilustra este conceito, mostrando um nó série com dois elementos de comprimento e três de profundidade.

Figura 6 – Fluxograma do processo de desenho da *Ladder*

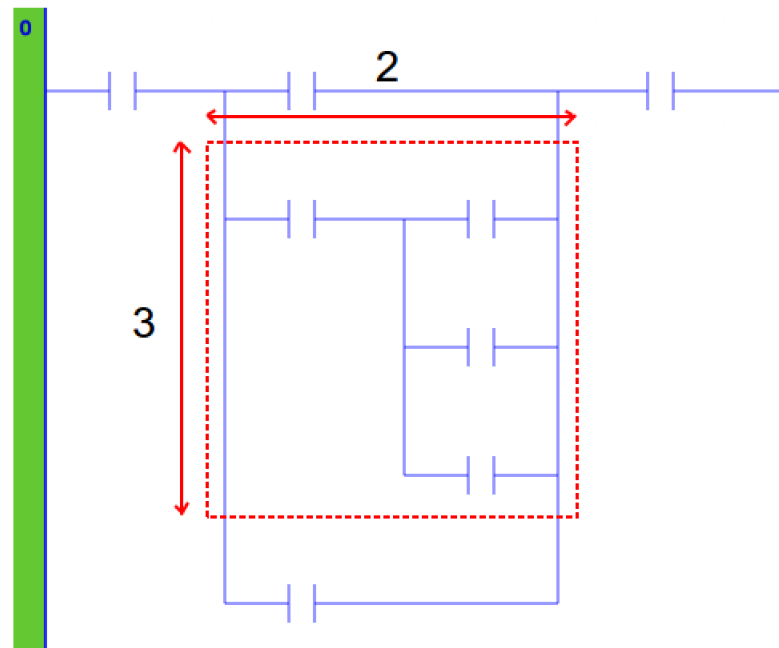


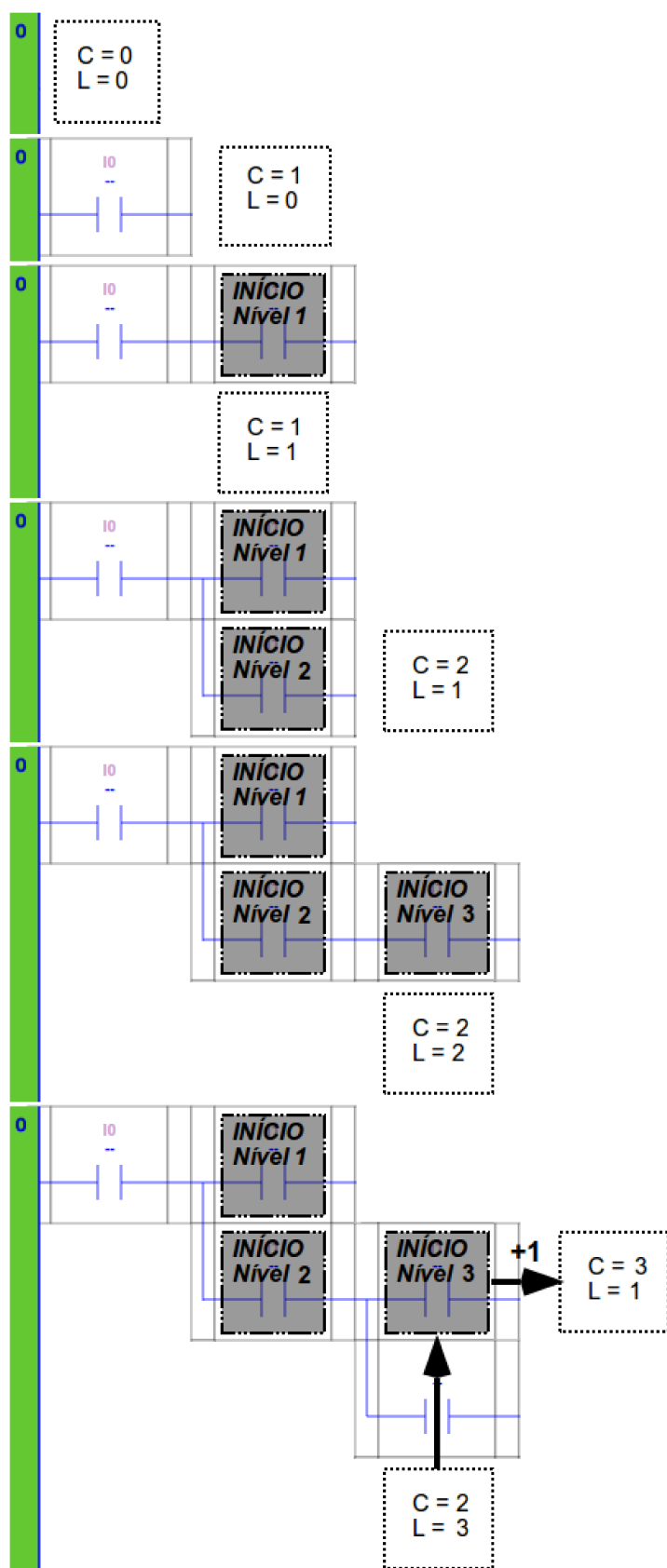
Figura 7 – Comprimento e profundidade de um nó série

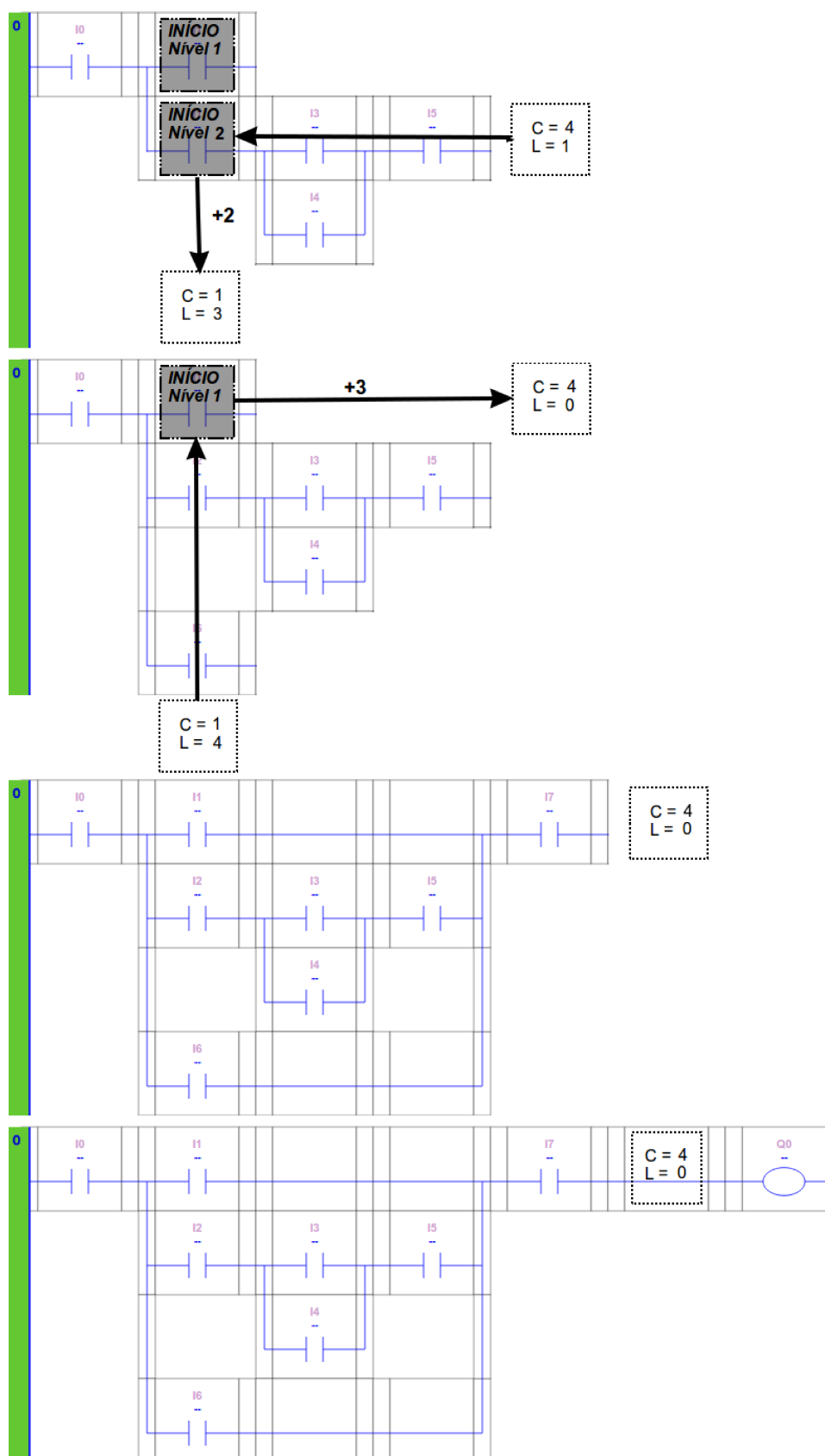
As Figuras 8 e 9 exemplificam o processo descrito acima. Para isto foi utilizada a seguinte *Ladder* mostrada na Figura 10. A estrutura de dados referente a ela se encontra a seguir.

```

PARALLEL
|      INTERMED (NO , I1 )
|      SERIES
|      |      INTERMED (NO , I2 )
|      |      PARALLEL
|      |      |      INTERMED (NO , I3 )
|      |      |      INTERMED (NO , I4 )
|      |      END
|      |      INTERMED (NO , I5 )
|      END
|      INTERMED (NO , I6 )
END
INTERMED (NO , I7 )
FINAL (OUT , Q0)

```

Figura 8 – Exemplo de desenho da *Ladder* (Parte 1)

Figura 9 – Processo de desenho da *Ladder* (Parte 2)

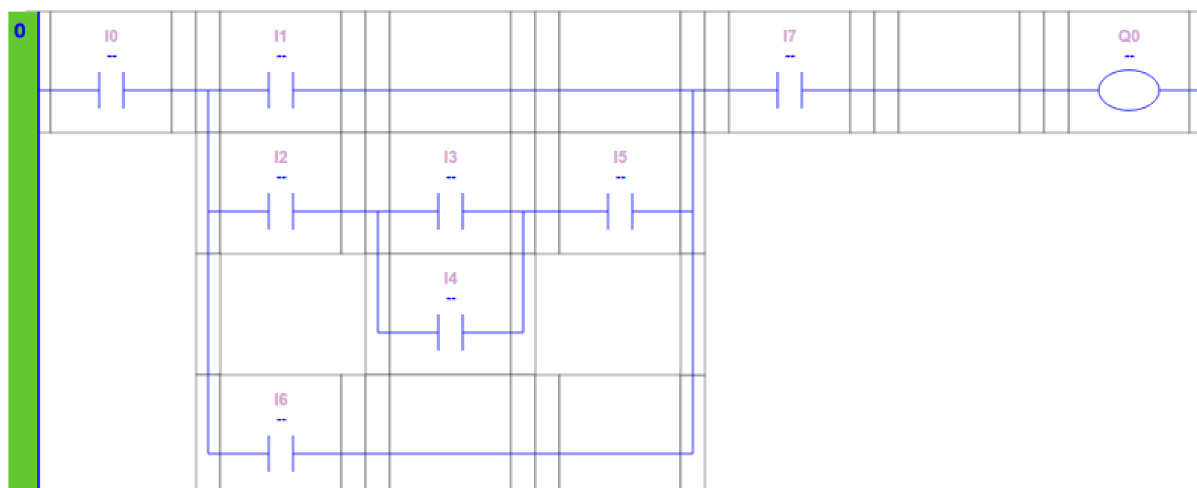


Figura 10 – Processo de desenho da *Ladder* (Completa)

A Figura 10 foi retirada diretamente do editor, apenas foram adicionados contornos a cada retângulo da grade. Percebe-se que existem espaços vazios, indicando que a grade está incompleta. Isso se dá ao fato de que os SVGs são criados conforme o algoritmo é executado e transladados para a posição indicada pela linha e coluna. Portanto não existem elementos onde não foi necessário desenhar nada.

Estabelecido este algoritmo, sempre que se deseja redesenhar a *Ladder* deve-se remover todos os SVGs criados e refazer todo o processo. Esta abordagem, embora não tão otimizada, possibilitou uma agilidade maior no desenvolvimento do editor.

Durante a implementação desta interface gráfica foram criados os símbolos das principais variações dos contatos e bobinas. São eles os contatos normalmente fechados, borda de subida e descida (também conhecidos como OSR e OSF) e bobinas de Set e Reset (JOHN; TIEGELKAMP, 2010).

Posteriormente foi adicionado suporte a blocos funcionais na *Ladder*. A primeira tentativa foi de encontrar um tamanho de elemento que comportasse tanto contatos quanto bobinas, de forma a não precisar alterar o sistema. Mas logo desistiu-se desta solução, pois deixaria a composição visual da *Ladder* muito prejudicada. Então foi feita uma atualização de forma que os blocos ocupariam mais de um elemento na grade. Nesta segunda implementação, foram feitas alterações para adicionar o comportamento de incrementar as colunas e linhas com o número de células ocupadas pelo bloco, em largura e altura, após desenhá-lo na *Ladder*. Assim o fluxograma da Figura 6 não se altera, exceto pelo valor incrementado.

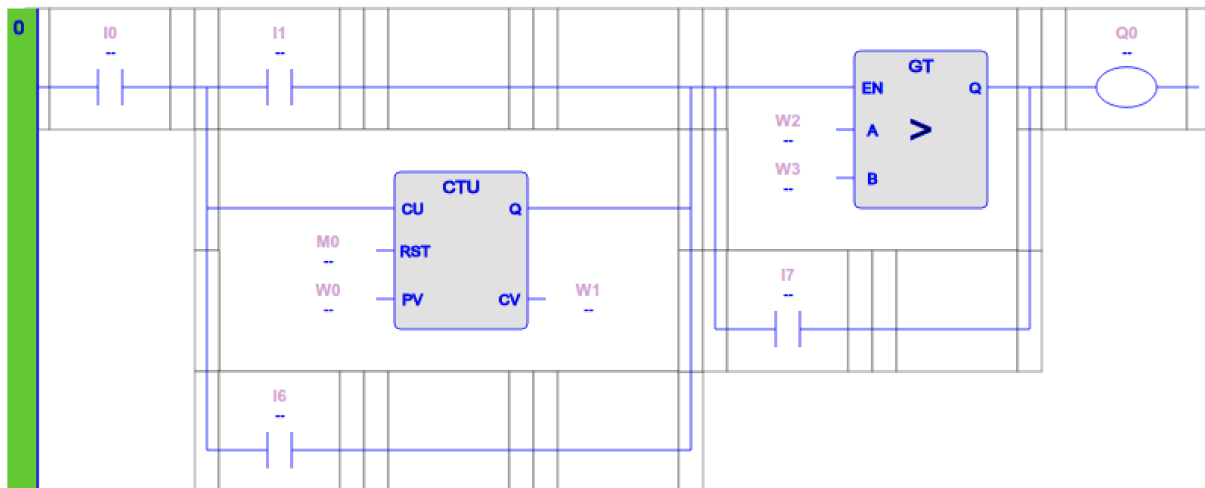


Figura 11 – Blocos funcionais na grade

Neste ponto finalizou-se a criação da interface de desenho da *Ladder*. O resultado final, com todos os elementos criados pode ser observado na Figura 12.

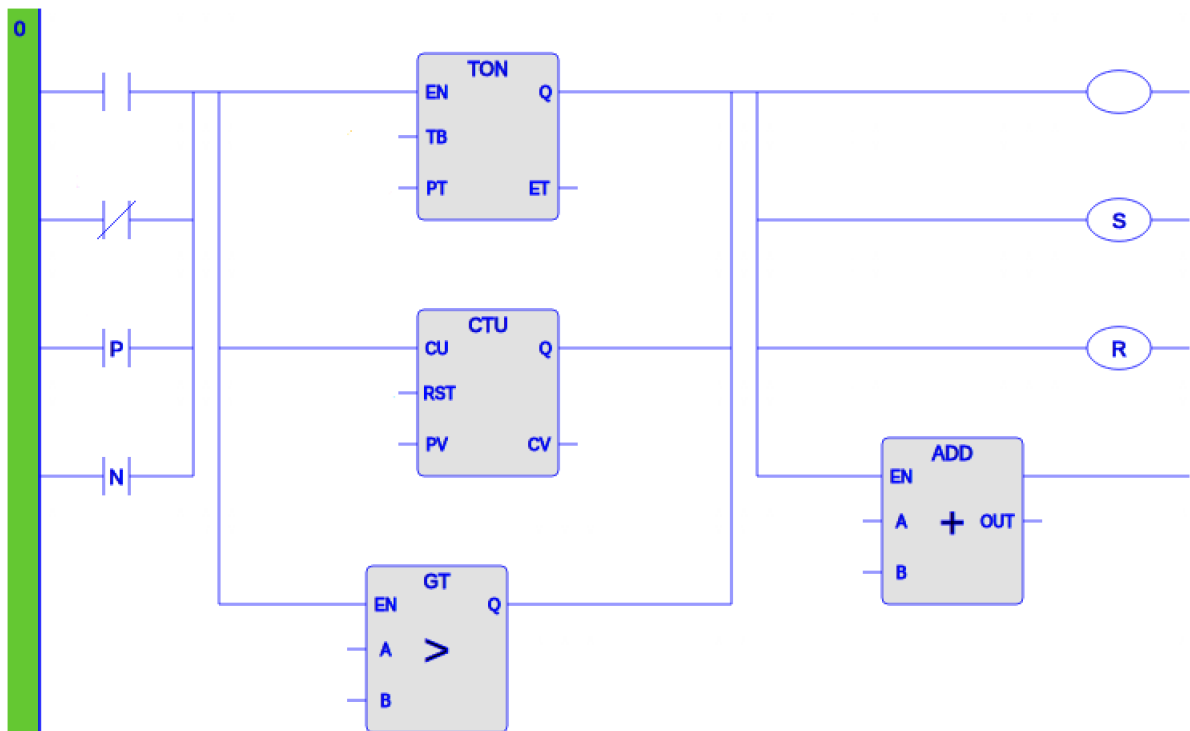


Figura 12 – Conjunto final de desenhos

O restante da implementação da interface gráfica do editor se resume a concepção da barra de ferramentas que deve ser populada com os botões e a criação do console, com as abas necessárias. Decidiu-se que inicialmente não haveria uma preocupação acentuada

com a experiência do usuário, bastando que o editor tivesse o suporte necessário para o teste de todas as funcionalidades previstas.

Assim foi criado um botão para cada ação de edição, por exemplo, existem quatro botões para se adicionar um contato, pois pode-se adicioná-lo em série a frente ou atrás, ou em paralelo acima ou abaixo.

Além disso foi adicionado um console, fixo no rodapé da página, podendo ser minimizado e restaurado. A princípio contém quatro abas, uma para exibir a estrutura de dados, outra para a visualização dos mnemônicos dos *opcodes*, a terceira exibindo o *bytecode* e uma quarta para informações gerais.

Para permitir a edição dos elementos foram utilizados os *inputs* do HTML, que consistem em caixas de texto editável. Também foi possível adicionar a cada um deles uma lista *dropdown* com as possibilidades de preenchimento daquele campo. Com isso foi possível atribuir endereços e *tags* a cada elemento.

Por fim, percebeu-se a oportunidade de se facilitar o salvamento da *Ladder* através da conversão dos vetores para formato JSON⁵. Convertendo-se a estrutura de dados para este formato pode-se salvá-la em um arquivo, e o processo inverso também é muito simples, facilmente carrega-se o arquivo e recupera-se os dados. Por se tratar basicamente de uma *string*, pode-se enviá-la diretamente através de requerimento POST para armazenamento em memória no dispositivo, lado a lado com o *bytecode*.

Desta forma, a Figura 13 mostra o resultado final do layout do editor, apresentando a barra de ferramentas com todos os botões, uma *Ladder* de exemplo e o console exibindo a estrutura de dados.

⁵ RFC 7169 Disponível em: <<https://tools.ietf.org/html/rfc7159>>. Acesso em: 01 jul. 2017.

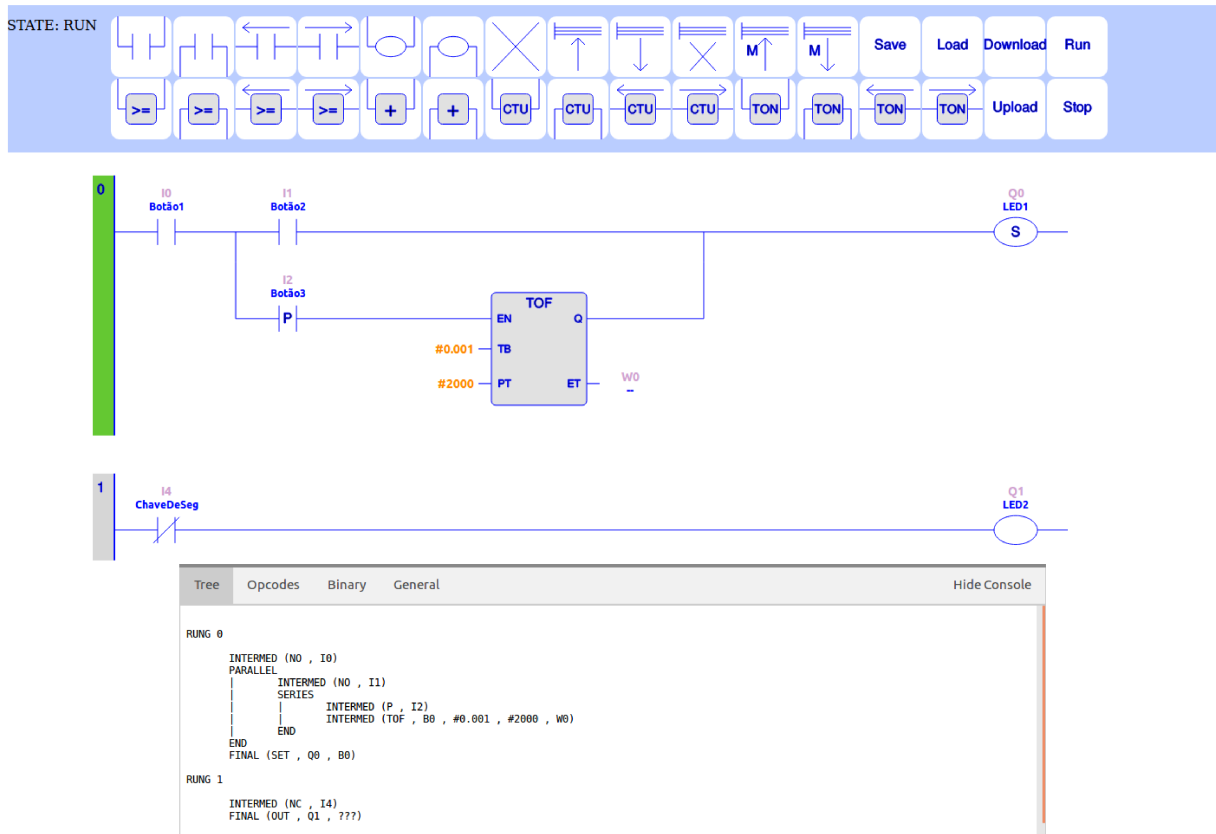


Figura 13 – *Layout* da aplicação no estado atual de desenvolvimento

3.2 Placa de Desenvolvimento

Como citado anteriormente, foi utilizada a placa de desenvolvimento NUCLEO-F767ZI da STMicroelectronics, mostrada na Figura 14, para criação do protótipo. Ela apresenta um microcontrolador ARM Cortex-M7 (STM32F767ZI), com 2048 KB de memória *Flash* e 512 KB de SRAM. Também contém uma interface *ethernet* completa, com transceiver e conector RJ45, necessária para o projeto.

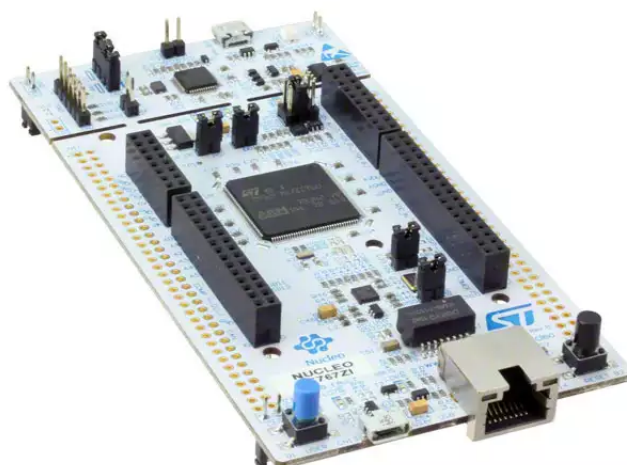


Figura 14 – NUCLEO-F767ZI

Foi construído um circuito simples em *protoboard* incluindo quatro botões, uma chave e quatro LEDs, caracterizando cinco entradas e quatro saídas do CLP. Esta montagem pode ser observada na Figura 15.

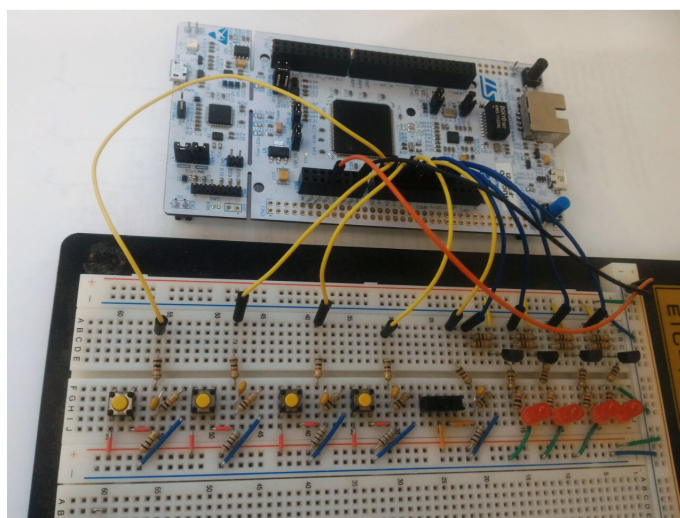
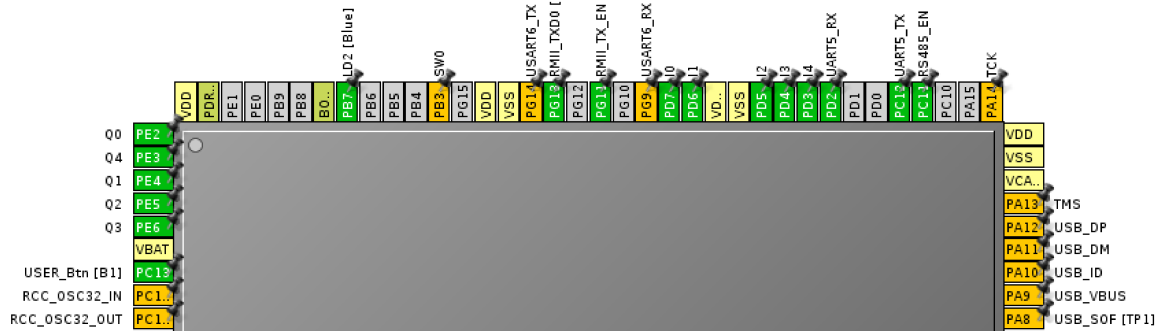


Figura 15 – Montagem do circuito de testes em protoboard

Então usou-se o *software* STM32CubeMX para a geração do código de configuração dos periféricos, *drivers* e a *stack* TCP/IP (lwIP, abordado posteriormente). A Figura 16 mostra a tela principal de configuração.

Figura 16 – Configuração dos pinos no *software* STM32CubeMX

Pin Name	Signal on ...	GPIO outp...	GPIO mode	GPIO Pull...	Maximum ...	Fast Mode	User Label
PB7	n/a	Low	Output P...	No pull-u...	Low	Disable	LD2 [Blue]
PB14	n/a	Low	Output P...	No pull-u...	Low	n/a	LD3 [Red]
PC11	n/a	Low	Output P...	No pull-u...	Low	n/a	RS485_EN
PC13	n/a	n/a	External I...	No pull-u...	n/a	n/a	USER_Btn...
PD3	n/a	n/a	Input mode	No pull-u...	n/a	n/a	I4
PD4	n/a	n/a	Input mode	No pull-u...	n/a	n/a	I3
PD5	n/a	n/a	Input mode	No pull-u...	n/a	n/a	I2
PD6	n/a	n/a	Input mode	No pull-u...	n/a	n/a	I1
PD7	n/a	n/a	Input mode	No pull-u...	n/a	n/a	I0
PE2	n/a	High	Output O...	No pull-u...	High	n/a	Q0
PE3	n/a	High	Output O...	No pull-u...	High	n/a	Q4
PE4	n/a	High	Output O...	No pull-u...	High	n/a	Q1
PE5	n/a	High	Output O...	No pull-u...	High	n/a	Q2
PE6	n/a	High	Output O...	No pull-u...	High	n/a	Q3

Figura 17 – Detalhamento da configuração da GPIO no *software* STM32CubeMX

Na Figura 17 observa-se a configuração de cada pino da GPIO. Nota-se que as entradas foram nomeadas de I0 a I4, e as saídas de Q0 a Q4, porém Q4 não é usada.

A stack TCP/IP e o servidor HTTP serão apresentadas na seção 3.4. Gerou-se o projeto para a IDE *SystemWorkbench* e nela foi desenvolvido o firmware do projeto.

3.3 Máquina Virtual

A implementação da máquina virtual foi dividida em quatro partes: as funções de chamada externa, que serão utilizadas pelo *firmware* para rodar a MV; as funções de execução dos *opcodes*, de uso interno, serão executadas conforme a leitura do *bytecode*; funções de gerenciamento das pilhas, chamadas pelos *opcodes* durante o processamento; e as funções de acesso ao *hardware*, que proveem a portabilidade do código.

Inicialmente deve-se criar uma estrutura para auxiliar no gerenciamento do contexto. Esta estrutura deve conter um ponteiro para a região de memória onde está

armazenado o *bytecode*, uma variável que indique o *byte* que está sendo lido pela MV, fazendo o papel de um contador de programa, e uma variável que indique o tamanho em *bytes* do *opcode* e seus parâmetros, para que se possa mover o contador corretamente para o próximo *opcode*.

O funcionamento da MV se dá essencialmente por uma tabela que relaciona o número do *opcode*, seu tamanho em *bytes* incluindo os parâmetros, e o endereço da função correspondente. Desta forma, o primeiro passo da execução é ler o primeiro *byte* do *bytecode* e acessar a entrada correta na tabela. Realiza-se a chamada da função endereçada nesta entrada da tabela. Após esse processo, o contexto conhece o tamanho do *opcode*, permitindo deslocar o contador de programa para que aponte para o próximo *opcode* a ser executado.

A seguir são exibidos trechos do código que implementa o contexto e os *opcodes*.

```
struct ladder_context_s
{
    int8_t *code;
    uint8_t size;
    uint8_t pc;
};

struct opcodes_s
{
    USHORT id;
    USHORT size;
    exec_routine exec;
};

enum opcodes_e
{
    EOPC_BEGIN =    0x01 ,
    EOPC_END =      0x02 ,
    EOPC_AND =      0x03 ,
    EOPC_OR =       0x04 ,
    EOPC_LOAD =     0x05 ,
    EOPC_STORE =    0x06
}

int opc_begin_exec()
{
```

```

        stacks_read_in();
        return 0;
    }

    int opc_and_exec()
    {
        uint8_t val_1 = stack_pop();
        uint8_t val_2 = stack_pop();
        stack_push(val_1 & val_2);
        return 0;
    }

    struct opcodes_s opcodes_table[] =
    {
        { EOPC_BEGIN,    1,      opc_begin_exec  },
        { EOPC_END,      1,      opc_end_exec    },
        { EOPC_AND,      1,      opc_and_exec    },
        { EOPC_OR,       1,      opc_or_exec     },
        { EOPC_LOAD,     3,      opc_load_exec   },
        { EOPC_STORE,    3,      opc_store_exec  }
    };

```

É necessário que existam vetores que armazenem os dados da Ladder, como o estado das entradas e saídas, memórias, blocos, números, etc. Para cada tipo de dado foi criado um vetor, com um tamanho fixo, definido em tempo de compilação, evitando a alocação dinâmica de memória. Também devem existir vetores que representem as pilhas, com variáveis (*stack pointers*) que indiquem o topo destas. Todas estas características devem ser encapsuladas por funções.

```

uint8_t STACK[MAX_STACK_SIZE];
uint8_t TMP_STACK[MAX_TMP_STACK_SIZE];
uint8_t stack_pointer;
uint8_t tmp_stack_pointer;

extern uint8_t R_IN[2][INPUT_SIZE];
extern uint8_t R_OUT[2][OUTPUT_SIZE];
extern uint8_t R_MEM[2][MEMR_SIZE];
extern uint8_t R_CNT[2][COUNTER_SIZE];
extern uint8_t R_ART[2][ARITH_SIZE];
extern uint8_t R_TM[2][TIMER_SIZE];

```

```

extern uint8_t R_SET[2][SET_RST_SIZE];
extern int32_t R_WORD[WORD_SIZE];

extern uint8_t W_IN[INPUT_SIZE];
extern uint8_t W_OUT[OUTPUT_SIZE];
extern uint8_t W_MEM[MEMR_SIZE];
extern uint8_t W_CNT[COUNTER_SIZE];
extern uint8_t W_ART[ARITH_SIZE];
extern uint8_t W_TIM[TIMER_SIZE];
extern uint8_t W_SET[SET_RST_SIZE];
extern int32_t W_WORD[WORD_SIZE];

extern uint32_t curr_io_stack;

/* — the code array — */
extern int8_t code_mem[CODE_MAX_SIZE];

extern uint32_t ticks;

void stack_push(uint8_t val)
{
    STACK[stack_pointer++] = val;
}

uint8_t stack_pop()
{
    uint8_t temp = STACK[--stack_pointer];
    STACK[stack_pointer] = 0;
    return temp;
}

void stacks_read_in(void)
{
    read_inputs(&R_IN[curr_io_stack][0]);

    for (int i = 0; i < OUTPUT_SIZE; i++)
        P_OUT[curr_io_stack][i] = W_OUT[i];

    for (int i = 0; i < MEMR_SIZE; i++)

```

```

    R_MEM[curr_io_stack][i] = W_MEM[i];

    for (int i = 0; i < WORD_SIZE; i++)
        R_WORD[i] = W_WORD[i];
}

void stacks_write_out(void)
{
    write_outputs((BOOL * const) &W_OUT[0]);

    for (int i = 0; i < COUNTER_SIZE; i++)
        R_CNT[curr_io_stack][i] = W_CNT[i];

    for (int i = 0; i < ARITH_SIZE; i++)
        R_ART[curr_io_stack][i] = W_ART[i];

    for (int i = 0; i < TIMER_SIZE; i++)
        R_TIM[curr_io_stack][i] = W_TIM[i];

    for (int i = 0; i < TIMER_SIZE; i++)
        R_SET[curr_io_stack][i] = W_SET[i];
}

```

Acima foram apresentadas as declarações dos vetores e as funções de empilhamento e desempilhamento da pilha principal. Os vetores de dados estão declarados com a palavra-chave **extern**, permitindo que as funções dos opcodes os acessem diretamente. Alguns detalhes foram omitidos, como os *defines* dos comprimentos dos vetores, que podem ser alterados facilmente. Também foram omitidas as funções de manipulação da pilha temporária, que são análogas às da pilha principal.

Destaca-se também que existem dois vetores para cada tipo de dado, um para leitura (cujo nome se inicia com *R_*) e outro para escrita (cujo nome se inicia com *W_*). Isso é necessário para que os dados não sejam alterados durante a execução da *Ladder*, pois a atualização deve ocorrer somente no fim do ciclo de *scan*, quando os vetores de escrita são copiados para os de leitura.

Como explicado na seção 3.3.1, alguns blocos devem ser executados somente quando ativados, como os blocos funcionais de *timers* e contadores. Alguns agem somente no ciclo em que foram ativados ou desativados, outros têm diversos estágios de execução que dependem destes eventos e do tempo entre eles. Portanto deve existir algum mecanismo de detecção de mudança de estado.

Para que fosse elaborado tal mecanismo para cada elemento, foi necessário criar dois vetores de leitura para cada tipo de dado. Estes vetores são atualizados alternadamente entre cada ciclo de *scan*, de forma que um deles contém os valores atuais e o outro contém os valores do último ciclo. Assim, quando estes valores forem diferentes, indica-se que aconteceu uma mudança de estado. Utilizou-se a variável `curr_io_stack` para auxiliar no acompanhamento desta alternância.

Assim, além dos vetores de entrada (IN), saída (OUT), memórias (MEM) e *words* (WORD), também foram criados vetores para cada um destes novos elementos. Utiliza-se um novo parâmetro nos *opcodes* para se referir a estes vetores, com o prefixo B, como apresentado posteriormente nesta seção. Cada ocorrência de um *opcode* que necessite desta funcionalidade deve indicar uma posição única através do *offset*.

Por fim, deve-se implementar as funções de acesso ao *hardware* utilizando técnicas de portabilidade. Criou-se um arquivo de cabeçalho que contém as declarações das funções chamadas pela MV, deixando a implementação destas funções em um arquivo separado, evidenciando que basta adaptar este arquivo para portar o código para outra plataforma. O trecho a seguir mostra a parte essencial do arquivo de cabeçalho.

```
#define HW_N_OUTPUTS    5    /* number of outputs */
#define HW_N_INPUTS     5    /* number of inputs */

void write_outputs(uint8_t * const w_out);
void read_inputs(uint8_t *r_in);
void update_code(int8_t *code_buf);
void tick_1ms_callback();
```

Como apresentado na seção 3.2, foi feita a configuração dos periféricos através do *software* STM32CubeMX. O código gerado inclui uma camada de abstração provida pelo fabricante do microcontrolador (STM32F767ZI). Assim, para se acessar a GPIO deste controlador, basta realizar chamadas às funções da HAL (*Hardware Abstraction Layer* - Camada de Abstração de *Hardware*). A seguir mostra-se a implementação das funções de escrita e leitura das entradas e saídas.

```
void write_outputs(uint8_t * const w_out)
{
    static const uint16_t q_pins[5] =
    {Q0_Pin, Q1_Pin, Q2_Pin, Q3_Pin, Q4_Pin};

    for(int i = 0; i < HW_N_OUTPUTS; i++)
        HAL_GPIO_WritePin(GPIOE, q_pins[i], (w_out[i] ? 0 : 1));
}
```

```

void read_inputs(uint8_t *r_in)
{
    static const uint16_t i_pins[5] =
    {I0_Pin, I1_Pin, I2_Pin, I3_Pin, I4_Pin};

    for(int i = 0; i < HW_N_INPUTS; i++)
        r_in[i] = HAL_GPIO_ReadPin(GPIOD, i_pins[i]);
}

```

Como o sistema pode ser implementado tanto em sistemas *bare metal*, que é o caso deste protótipo, tanto em plataformas com sistema operacional, o armazenamento do *bytecode* não tem sempre a mesma forma. Portanto deve-se implementar a função `update_code`. Nela, o *bytecode* deve ser acessado da forma adequada e copiado para o vetor `code_mem`, armazenado em RAM e interno à MV.

```

void update_code(int8_t *code_buf)
{
    for(int i = 0; i < CODE_MAX_SIZE; i++)
        code_mem[i] = code_buf[i];
}

```

Na Seção 3.5 se discorre sobre a base de dados criada para armazenamento dos dados em memória não volátil.

Enfim deve haver uma base de tempo para os blocos funcionais de *Timers*. A solução tradicional é a criação de uma variável *ticks*, incrementada pela função `systick_callback`. Esta função deve ser executada a cada um milissegundo pela plataforma em que se está implementando a MV. Na plataforma STM32, basta utilizar a função de mesmo nome da HAL, `HAL_SYSTICK_Callback`, chamada por interrupção pelo *timer* do sistema.

Neste ponto do desenvolvimento tem-se a estrutura básica da MV pronta, assim como todas as funcionalidades do editor. Realizou-se então a criação dos próximos elementos e blocos a serem suportados, tanto a parte gráfica como a implementação das funções da MV relativas a estes novos *opcodes*. Primeiramente, para suportar todas as operações de todos os *opcodes* foram criados sete prefixos. A Tabela 5 mostra seus mnemônicos e valores numéricos, bem como um resumo sobre sua função e a Tabela 6 mostra o conjunto completo de *opcodes* desenvolvido, com os argumentos possíveis.

Prefixo	Valor	Descrição
I	0x01	Entrada: Indica o vetor de entradas
Q	0x02	Saída: Indica o vetor de saídas
M	0x03	Memória: Indica o vetor de memórias, que armazena bits auxiliares
W	0x04	Word: Indica o vetor que armazena valores numéricos
#	0x05	Constante: Indica que o valor inserido é uma constante numérica
B	0x06	Bloco: Indica um vetor auxiliar no instanciamento de blocos

Tabela 5 – Prefixos implementados

Opcode	Valor	Formato
BEGIN	0x01	BEGIN
END	0x02	END
AND	0x03	AND
OR	0x04	OR
LDR	0x05	LDR <I,Q,M>,<off>
STR	0x06	STR <Q,M>,<off>
NOT	0x08	NOT
SET	0x09	SET ,<off> <Q,M>,<off>
RESET	0x0A	RESET ,<off> <Q,M>,<off>
OSR	0x0B	OSR <I,Q,M>,<off>
OSF	0x0C	OSF <I,Q,M>,<off>
GT	0x0D	GT <#,W>,<off>
GTE	0x0E	GTE <#,W>,<off>
LT	0x0F	LT <#,W>,<off>
LTE	0x10	LTE <#,W>,<off>
ADD	0x11	ADD ,<off> <#,W>,<off> <#,W>,<off>
SUB	0x12	SUB ,<off> <#,W>,<off> <#,W>,<off>
MUL	0x13	MUL ,<off> <#,W>,<off> <#,W>,<off>
DIV	0x14	DIV ,<off> <#,W>,<off> <#,W>,<off>
CTU	0x15	CTU ,<off> <I,Q,M>,<off> <#,W>,<off> <W>,<off>
CTD	0x16	CTD ,<off> <I,Q,M>,<off> <#,W>,<off> <W>,<off>
TON	0x17	TON ,<off> <#>,<off> <#,W>,<off> <W>,<off>
TOF	0x18	TOF ,<off> <#>,<off> <#,W>,<off> <W>,<off>
PUSH_TMP	0x1A	PUSH_TMP
GET_TMP	0x1B	GET_TMP
POP_TMP	0x1C	POP_TMP

Tabela 6 – Conjunto completo dos *opcodes* implementados

Como descrito na seção 2.1, os prefixos são utilizados nos parâmetros dos *opcodes* para indicar o vetor correspondente na memória da máquina virtual, juntamente com um *offset*, que indica a posição do dado neste vetor. Desta forma, os *offsets* são valores numéricos hexadecimais de dois dígitos, representando um *byte*. Porém o prefixo # introduz um comportamento diferente, indicando que o valor contido no *offset* deve ser tratado

como uma constante numérica. Neste caso deve-se utilizar oito dígitos neste campo, por se tratar de valores hexadecimais, para suportar constantes de 32 bits.

A seguir são apresentados todos os *opcodes* (incluindo os já mostrados na seção 2.1), os elementos relacionados e uma breve descrição da implementação da sua execução na máquina virtual.

- BEGIN

É sempre o primeiro *opcode* a ser executado em um ciclo de *scan*. Chama a função de leitura das entradas, atualizando estes valores.

- END

É sempre o último *opcode* a ser executado em um ciclo de *scan*. Chama a função de escrita das saídas, atualizando estes valores.

- AND

Desempilha dois valores da pilha principal, executa um E lógico entre eles e empilha o resultado desta operação. Este opcode resulta da associação de elementos em série na *Ladder*. Posteriormente, na seção 3.3.1, discorre-se sobre uma exceção no comportamento desta função.

- OR

Desempilha dois valores da pilha principal, executa um OU lógico entre eles e empilha o resultado desta operação. Este opcode resulta da associação de elementos em paralelo na *Ladder*.

- PUSH_TMP

Apenas copia o valor do topo da pilha principal e empilha na pilha temporária.

- GET_TMP

Apenas copia o valor do topo da pilha temporária e empilha na pilha principal.

- POP_TMP

Desempilha um valor da pilha temporária.

- LDR

Busca o valor no vetor indicado pelo prefixo e posição indicada pelo *offset* e o empilha. Destaca-se que somente valores de entrada, saída ou memória virtual podem ser carregados. Este *opcode* é gerado através dos contatos normalmente abertos e normalmente fechados. Nota-se que os contatos normalmente fechados inserem um opcode NOT logo após o LDR gerado.

Exemplo de uso e tradução para *bytecode*:

LDR M,01 -> 0x050301



Figura 18 – Contatos normalmente aberto e fechado

- STR

Desempilha um valor da pilha principal e armazena-o no vetor indicado. Destaca-se que valores de entrada não podem ser alterados desta maneira pois são somente leitura. Desta forma, pode-se armazenar dados nos vetores de saída e memória virtual. Este opcode é gerado através das bobinas de saída.

Exemplo de uso e tradução para *bytecode*:

STR Q,03 -> 0x060203

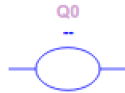


Figura 19 – Bobina de saída

- SET e RESET

Realiza a ação de armazenar 1 (SET) ou 0 (RESET) no endereço indicado. Esta ação é executada somente quando houver uma transição de 0 para 1 no estado do elemento, como mencionado anteriormente. Esta transição é monitorada através do vetor auxiliar de blocos, indicado pelo prefixo B. Ainda, como explicado anteriormente, nota-se que no código apresentado anteriormente existem apenas os vetores R_SET e W_SET, pois são compartilhados por ambos *opcodes*.

Desta maneira, para cada ocorrência de qualquer uma destas instruções, deve-se atribuir um *offset* único ao prefixo B no primeiro parâmetro.

Exemplo de uso e tradução para *bytecode*:

SET B,01 M,02 -> 0x0906010302

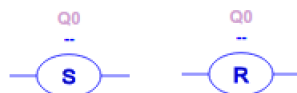


Figura 20 – Bobinas de *set* e *reset*

- OSR e OSF

Estas instruções realizam a ativação de um contato apenas no ciclo de *scan* em que ocorreu uma transição de estado. Estes *opcodes* recebem estes mnemônicos em referência às tradicionais instruções *one-shot-rising* e *one-shot-falling*, que indicam bordas de subida e bordas de descida, respectivamente. Portanto, a instrução OSR empilha 1 no ciclo em que o estado do elemento mudar de 0 para 1 e a instrução OSF empilha 1 no ciclo em que o estado mudar de 1 para 0.

Como o vetor de leitura **R_IN**, no momento da execução da instrução contém o valor anterior, não é necessário o uso de um vetor auxiliar para detectar a transição de estado.

Exemplo de uso e tradução para *bytecode*:

OSR I,00 -> 0x0B0100

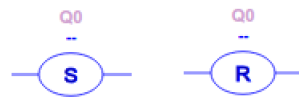


Figura 21 – Bobinas de *set* e *reset*

- GT, GTE, LT e LTE

Estas instruções realizam a comparação de dois valores, que podem ser *words* ou constantes. Na sua execução desempilham um valor da pilha principal, e se este valor for 1, ou seja, se o elemento estiver ativado, realizam a comparação indicada e empilham o resultado. Na *Ladder* existem quatro blocos comparadores que geram estes opcodes. Estes blocos contém duas entradas, A e B, e a comparação é feita na forma $A > B$.

Exemplos de uso e tradução para *bytecode*:

GTE W,02 W,01 -> 0x0E04020401

GTE #,0000ABBA W,01 -> 0x0E050000ABBAA0401

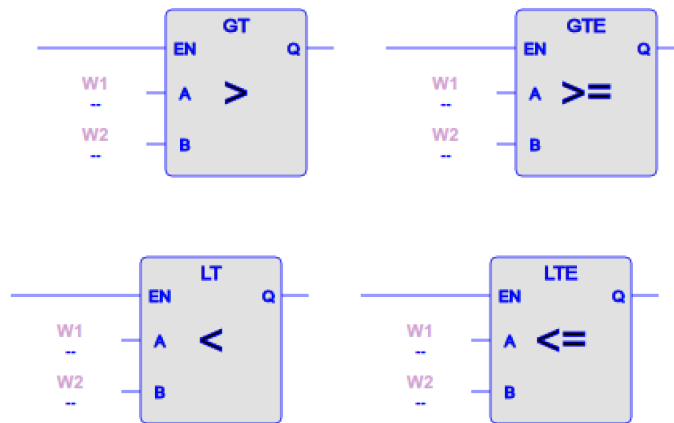


Figura 22 – Blocos comparadores

- ADD, SUB, MUL e DIV

Estas instruções realizam uma operação aritmética entre dois valores, que podem ser *words* ou constantes e armazenam o resultado em uma *word*. Na sua execução desempilham um valor da pilha principal, e se este valor for 1, ou seja, se o elemento estiver ativado, realizam a operação indicada.

Estes blocos só devem executar uma vez a cada ativação, e para isso, como explicado anteriormente, utilizam os vetores `R_ART` e `W_ART`. O *opcode* realiza esta associação através do primeiro parâmetro (prefixo B) e seu *offset*.

Destaca-se que há um desempilhamento, mas não há empilhamento, portanto esta instrução consome um valor da pilha. Por isso, os blocos aritméticos são tratados como elementos finais na *Ladder*. Estes blocos contém duas entradas, A e B, e uma saída OUT. A operação é feita na forma $OUT = A + B$.

Exemplos de uso e tradução para *bytecode*:

ADD W,02 W,01 -> 0x0E04020401

GTE #,0000ABBA W,01 -> 0x0E050000ABBAA0401

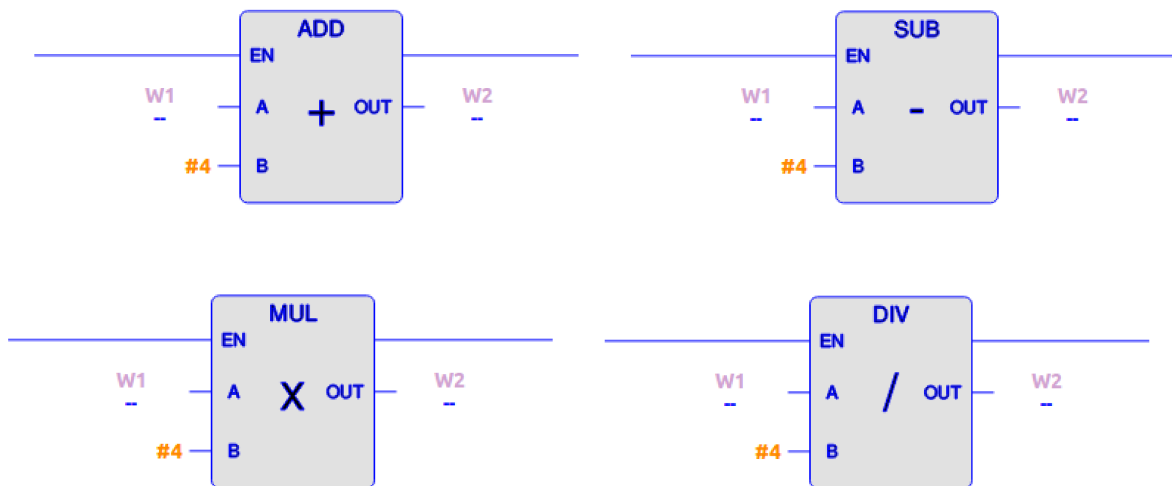


Figura 23 – Blocos aritméticos

- CTU e CTD

Estas instruções realizam consecutivamente o incremento ou decremento de uma *word* (CV) e a comparação dela com outra *word* ou uma constante (PV). Um terceiro parâmetro (RST) zera a variável do contador (CV).

Analogamente aos *opcodes* apresentados acima, utiliza-se os vetores R_CNT e W_CNT para detecção de mudança de estado e realiza esta associação através do primeiro parâmetro (prefixo B) e seu *offset*.

Exemplos de uso e tradução para *bytecode*:

CTU B,01 M,02 W,03 W,04 -> 0x150601030204030404

CTU B,01 M,02 #,FFFC0CA W,03 -> 0x150601030205FFFFC0CA0403

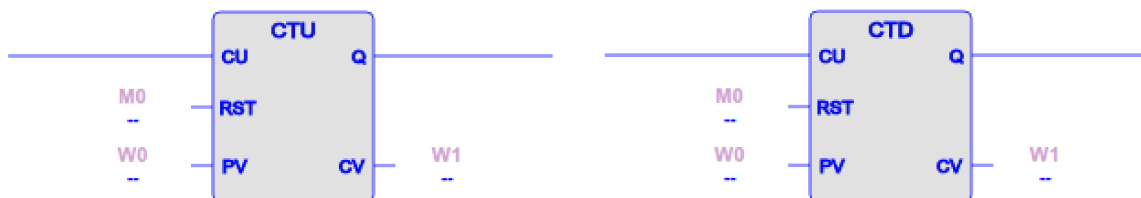


Figura 24 – Blocos contadores

- TON e TOF

Estas instruções realizam atrasos e persistências temporais de ativação na *Ladder*.

A instrução TON detecta quando houve a ativação da entrada do bloco (EN), inicia a contagem do tempo na variável de saída (ET), utilizando a base de tempo indicada

(TB) e somente após o intervalo de tempo indicado (PT) ser alcançado, ativa a saída do bloco (Q). O diagrama de estados da Figura 25 detalha este funcionamento.

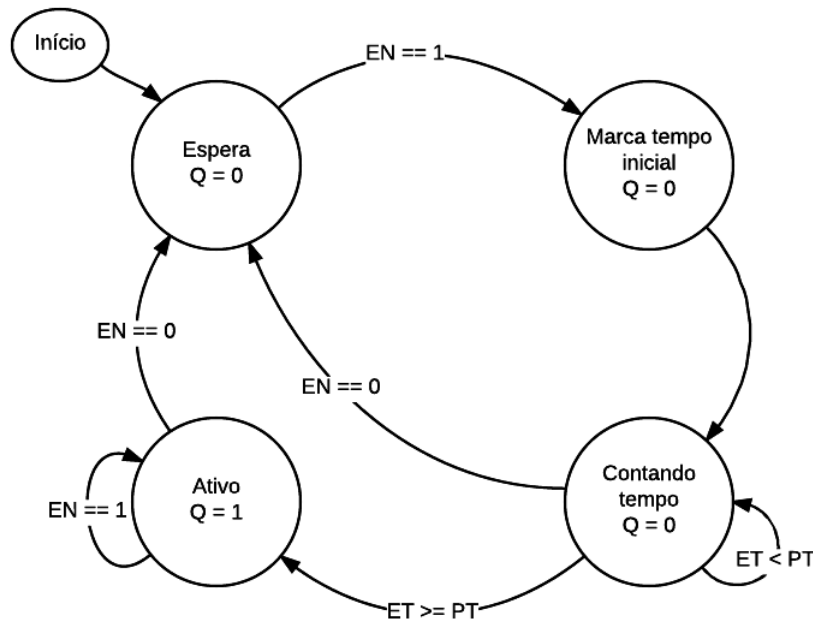


Figura 25 – Diagrama de estados da instrução TON

A instrução TOF detecta quando houve a ativação da entrada do bloco (EN) e imediatamente ativa a saída do bloco (Q), inicia a contagem do tempo na variável de saída (ET), utilizando a base de tempo indicada (TB) e somente após o intervalo de tempo indicado (PT) ser alcançado, desativa a saída do bloco. O diagrama de estados da Figura 26 detalha este funcionamento.

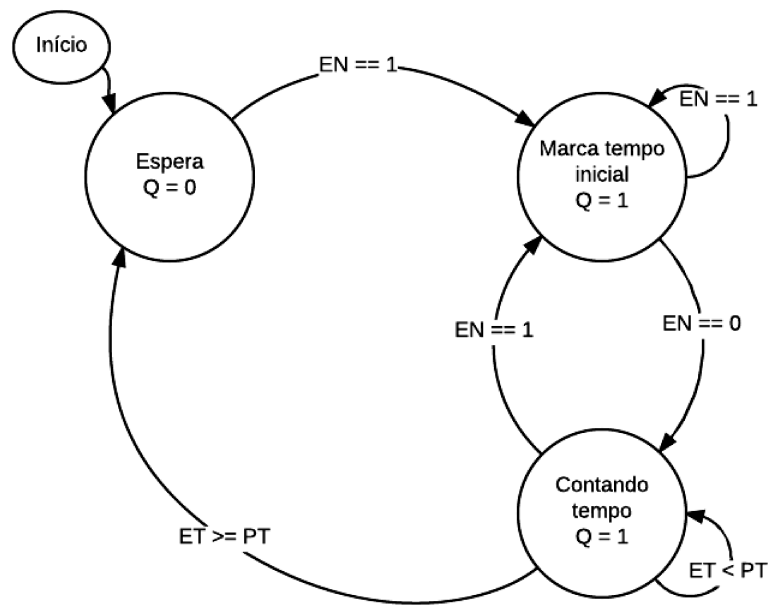


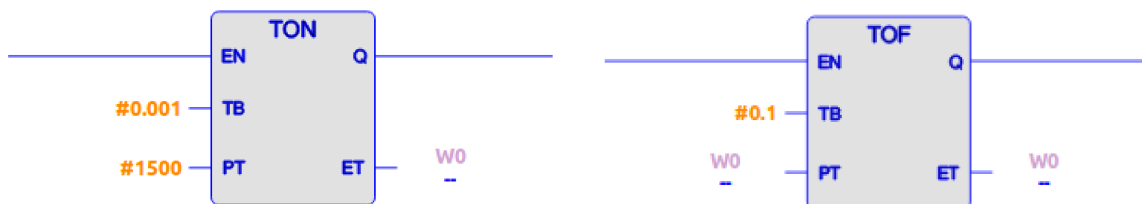
Figura 26 – Diagrama de estados da instrução TOF

Analogamente aos *opcodes* apresentados acima, utiliza-se os vetores R_TIM e W_TIM para detecção de mudança de estado e realiza esta associação através do primeiro parâmetro (prefixo B) e seu *offset*.

Exemplos de uso e tradução para *bytecode*:

TON B,01 #,00000001 W,03 W,04 -> 0x170601050000000104030404

TON B,01 #,0000000A #,000005DC W,04 -> 0x170601050000000A05000005DC0404

Figura 27 – Blocos de *timers*

3.3.1 Casos especiais

Durante o desenvolvimento dos blocos contadores e *timers*, percebeu-se que eles deveriam se comportar de maneira diferente dos outros elementos. Os contadores e todos os outros blocos quando ativam suas saídas, sempre têm sua entrada ativada. Desta forma, em analogia com um circuito elétrico, para que estes elementos energizem suas saídas, necessariamente precisam estar com a entrada energizada.

Existem situações em que os blocos TOF, CTU e CTD não seguem esta regra, ou seja, mesmo com a entrada desenergizada, energizam a saída. Isto pode ser observado facilmente no diagrama de estados do bloco TOF na Figura 26, onde no estado *Contando tempo* percebe-se que a entrada (EN) está desativada (zero) e mesmo assim a saída (Q) está ativada (um).

Nestes casos, a função AND, da forma como foi apresentada, impede o funcionamento correto da *Ladder*.

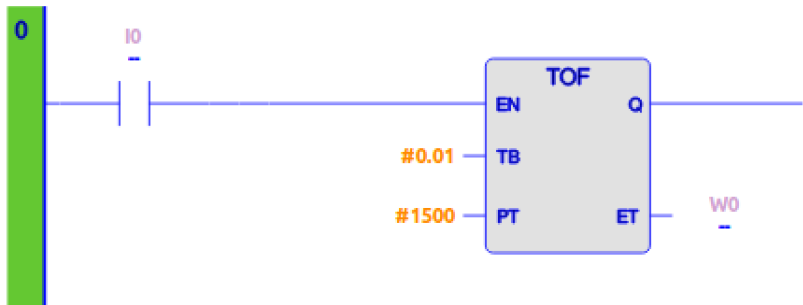


Figura 28 – Exemplo de exceção na *Ladder*

A Figura 28 mostra um trecho de *Ladder*, que gera os *opcodes* a seguir:

```
LDR I,00
TOF B,00 #,00000001 #,000005DC W,00
AND
```

Supondo que o contato esteja desativado, ou seja, empilha o valor 0, e que o bloco esteja ativado, empilhando o valor 1, quando a instrução AND for executada irá desempilhar estes dois valores e reempilhar 0 novamente. A Tabela 7 mostra os estados da pilha após cada instrução.

Instrução	LDR	TOF	AND
Pilha	0	1	0
	-	0	-
	-	-	-

Tabela 7 – Exceção da instrução AND

Este não é o comportamento esperado nesta situação, espera-se que o bloco permaneça ativo e que o elemento logo a frente seja energizado. Então foi elaborado um método para se adaptar a implementação da máquina virtual sem se fazer alterações profundas em seu funcionamento.

Toda vez que estes blocos estiverem com a saída energizada e a entrada desenergizada, ao invés de empilharem o valor 1, empilham o valor 2. Na representação binária do número 2 (10b) o segundo bit vale 1. Este bit é utilizado para indicar que a operação AND não deve realizar sua função comum, mas tratar estes valores como casos especiais.

Como explicado na seção anterior, a instrução AND desempilha dois valores (um por vez) no início de sua execução. Agora, caso o primeiro valor tenha o segundo bit marcado, apenas deve-se reempilhar o valor 2. Caso o segundo valor tenha o segundo bit marcado, executa-se um E lógico entre os dois valores, e se o resultado for zero, empilha zero, se não, empilha 2. Caso nenhum dos valores tenha este bit marcado, a instrução executa sua funcionalidade tradicional. A Tabela 8 mostra o resultado destas mudanças na execução do código.

Instrução	LDR	TOF	AND
Pilha	0	2	2
	-	0	-
	-	-	-

Tabela 8 – Solução do caso especial

É possível que esta situação ocorra dentro de nós paralelos, e que instruções OR também realizem operações com estes valores especiais na pilha. Porém, como esta função realiza um OU lógico, o segundo bit é mantido sempre que presente em um dos operandos. Desta maneira, a instrução OR não necessita ser alterada.

A seguir são exibidas as funções implementadas na máquina virtual.

```
void opc_and_exec()
{
    uint8_t val_1 = stack_pop();
    uint8_t val_2 = stack_pop();

    if(val_2 & 2)
        stack_push(val_1 & 1 ? 2 : 0);

    else if(val_1 & 2)
        stack_push(2);

    else
        stack_push(val_1 & val_2);
}

void opc_or_exec()
{
    uint8_t val_1 = stack_pop();
    uint8_t val_2 = stack_pop();

    stack_push(val_1 | val_2);
}
```

3.4 Servidor Web

Como apontado no na seção 3.2, a placa de desenvolvimento utilizada conta com o *hardware* da interface de rede ethernet pronto para uso. A plataforma de configuração STM32CubeMX provê a configuração e geração de drivers e aplicações do protocolo TCP/IP, fazendo uso da biblioteca *lwIP*.

Foi feita uma configuração simples, com endereço IP estático (192.168.1.1/24) sem suporte a DHCP e DNS, a fim de se obter somente a funcionalidade de rede essencial, evitando quaisquer problemas decorrentes de protocolos adicionais.

Esta biblioteca inclui uma implementação de servidor web, chamada de *httpd*, com suporte a um sistema de arquivos, no qual os estes são apresentados como vetores de caracteres. A biblioteca também inclui um *script*, fora do projeto, que gera estes vetores a partir do código fonte do editor e os coloca em um único arquivo. Desta maneira estes dados são gravados em *Flash*.

O *https* suporta os métodos de requisição POST e GET do protocolo HTTP. O

método GET é amplamente utilizado para requisitar informações do servidor, como os arquivos HTML e Javascript. O método POST é empregue no envio de informações ao servidor, por exemplo em formulários *web*.

Por estes métodos foi implementada a comunicação do navegador com o microcontrolador. Através de botões no editor o usuário pode enviar os comandos de Run e Stop, para iniciar e parar a execução da *ladder*, e pode fazer *download* e *upload* do código. Ressalta-se que "*download*" remete ao envio do código do computador para o CLP, o que pode ser confuso do ponto de vista cliente-servidor *web*, no qual "*download*" remete ao envio de informações do dispositivo para o navegador. A seguir exemplifica-se estes processos.

Para se enviar o comando de Run, quando acionado o botão, o navegador deve montar a requisição, enviá-la e aguardar respostas de estado e *status* do servidor, utilizando-se as técnicas AJAX. Por meio do objeto `XMLHttpRequest` inicializa-se um novo requerimento POST via URL do servidor, acrescida de "`cgi/plcrun`", gerando uma nova conexão TCP e enviando o *payload* "`plcrun`". A seguir mostra-se esta forma tradicional, porém destaca-se que estes requerimentos não são exibidos na barra de endereço, como esperado (POWELL, 2008).

```
192.168.1.1 / cgi / plcrun ? plcrun
```

Neste momento, no microcontrolador, o *lwIP* deve aceitar a conexão e encaminhar os dados ao *httpd*, que interpreta a URL, e durante a vida deste requerimento chama três funções implementadas pelo usuário. A primeira delas é o `httpd_post_begin`, chamada uma única vez quando for recebido o primeiro pacote do requerimento. Se houverem pacotes subsequentes, evidentemente o *payload* era muito grande e teve de ser dividido pelo TCP. Para cada um destes pacotes ocorre uma chamada à função `httpd_post_receive_data`, onde deve-se coletar seus dados. Por fim, quando toda a informação tiver sido transmitida, chama-se `httpd_post_finished`. Nesta função deve se montar uma resposta a ser enviada para o navegador, utilizando para isso o ponteiro `response_uri`.

Criou-se um mecanismo simples para distinguir se o comando foi enviado corretamente. É constituído de uma estrutura com dois valores, um indicando o estado da conexão e o outro indicando o comando. No início do processamento de um requerimento marca-se o estado como ocupado, de forma que não se atenda um segundo antes que este não tenha sido finalizado. Registra-se também qual foi o comando utilizado, através da leitura da URL.

Desta forma foi possível implementar a resposta do sistema ao comando emitido pelo editor. Para os comandos Run e Stop, o sistema chama funções que param e resumem a execução da MV de maneira simples, apenas alterando uma *flag*. Nestes casos a resposta

do servidor é uma mensagem de sucesso ou erro.

O requerimento de *download* de código (transferência do navegador para o CLP) é montado de forma similar, porém com dois campos que contém o *bytecode* e o JSON. Como mencionado na seção 2.2, foi decidido no projeto, por conveniência, que a estrutura de dados que representa a *Ladder* no editor também seria transferida para a memória do dispositivo, em formato JSON, pois facilitaria muito a sua recuperação através do *upload*. No mais, o processamento deste pelo *lwIP* funciona da mesma maneira, respondendo com uma mensagem de erro ou sucesso. O trecho abaixo ilustra o formato do requerimento.

```
192.168.1.1/cgi/download?bin=010502030802&json="[["INTERMED",
["NO",["I0"]]],["FINAL",["OUT",["Q0","???"]]]"
```

O código fonte da CGI descrita acima encontra-se no Apêndice A.

3.5 Base de dados

É evidente a necessidade de se manter os vetores do *bytecode* e do JSON em memória *Flash*, para que os dados permaneçam gravados no microcontrolador quando o mesmo for desenergizado. Assim, foi necessário fazer algumas modificações no *script* do *Linker*. Primeiramente é preciso especificar uma nova área de memória, batizada de *user_data*. Foi especificado o endereço e o tamanho desta área, que será o setor 11 da *Flash*, como indicado na Figura 29, retirada do manual de usuário do controlador.

Block	Name	Bloc base address on AXIM interface	Block base address on ICTM interface	Sector size
Main memory block	Sector 0	0x0800 0000 - 0x0800 7FFF	0x0020 0000 - 0x0020 7FFF	32 KB
	Sector 1	0x0800 8000 - 0x0800 FFFF	0x0020 8000 - 0x0020 FFFF	32 KB
	Sector 2	0x0801 0000 - 0x0801 7FFF	0x0021 0000 - 0x0021 7FFF	32 KB
	Sector 3	0x0801 8000 - 0x0801 FFFF	0x0021 8000 - 0x0021 FFFF	32 KB
	Sector 4	0x0802 0000 - 0x0803 FFFF	0x0022 0000 - 0x0023 FFFF	128 KB
	Sector 5	0x0804 0000 - 0x0807 FFFF	0x0024 0000 - 0x0027 FFFF	256 KB
	Sector 6	0x0808 0000 - 0x080B FFFF	0x0028 0000 - 0x002B FFFF	256 KB
	Sector 7	0x080C 0000 - 0x080F FFFF	0x002C 0000 - 0x002F FFFF	256 KB
	Sector 8	0x0810 0000 - 0x0813 FFFF	0x0030 0000 - 0x0033 FFFF	256 KB
	Sector 9	0x0814 0000 - 0x0817 FFFF	0x00340000 - 0x0037 FFFF	256 KB
	Sector 10	0x0818 0000 - 0x081B FFFF	0x0038 0000 - 0x003B FFFF	256 KB
	Sector 11	0x081C 0000 - 0x081F FFFF	0x003C 0000 - 0x003F FFFF	256 KB
Information block	System memory	0x1FF0 0000 - 0x1FF0 EDBF	0x0010 0000 - 0x0010 EDBF	60 Kbytes
	OTP	0x1FF0 F000 - 0x1FF0 F41F	0x0010 F000 - 0x0010 F41F	1024 bytes
	Option bytes	0x1FFF 0000 - 0x1FFF 001F	-	32 bytes

Figura 29 – Organização da memória *Flash*

Abaixo são mostrados os trechos modificados no arquivo *ld*.

MEMORY

{

RAM (xrw) : ORIGIN = 0x20000000 , LENGTH = 512K

FLASH (rx) : ORIGIN = 0x8000000 , LENGTH = 1792K

DATA (xrw) : ORIGIN = 0x081C0000 , LENGTH = 256K

}

SECTIONS

{

/*...*/

.user_data :

{

. = ALIGN(4);

*(.user_data)

. = ALIGN(4);

} >DATA

/*...*/

}

Deve-se então declarar os vetores da seguinte forma para indicar ao compilador a intenção de armazená-los na seção *user_data*:

```
__attribute__((__section__(".user_data")))  
    const SHORT code_file[CODE_MAX_SIZE];  
__attribute__((__section__(".user_data")))  
    const CHAR json_file[JSON_MAX_SIZE];
```

O acesso a essa região de memória pode ser feito normalmente, mas para se realizar a escrita de dados nesta nova seção deve-se utilizar as funções do driver da *Flash* da biblioteca HAL. Para se escrever deve-se previamente destravar a *Flash*, apagar os dados do setor todo através da função `FLASH_Erase_Sector` e realizar a gravação, registro a registro, pela função `HAL_FLASH_Program`. Este processo é mostrado no Apêndice A.

4 Resultados

Após o desenvolvimento do editor e os testes feitos entre ele e a máquina virtual implementada, pôde-se fazer a interligação com o servidor web e iniciar a conclusão de uma primeira versão do protótipo com todas as funcionalidades mínimas previstas.

O primeiro problema observado foi no funcionamento do protocolo ARP. Esporadicamente o *lwIP* não responde a pedidos de identificação do endereço MAC, nem atualiza a própria tabela com o endereço do host cliente. Não foi possível detectar a origem deste comportamento, por isso ainda persiste no protótipo. Em uma tentativa de resolver o problema foi criado um código que força uma entrada na tabela ARP com o endereço MAC do computador utilizado para os testes, sem sucesso. Desta forma, o problema continua sendo contornado através do reset da placa até que o protocolo funcione. A Figura 30 mostra uma captura deste problema pelo *Wireshark*.

Time	Source	Destination	Protocol	Length	Info
0.00000...	Micro-St_7f:15:ec	Broadcast	ARP	42	Gratuitous ARP for 192.168.1.10 (Reply)
2.29664...	Micro-St_7f:15:ec	Broadcast	ARP	42	Gratuitous ARP for 192.168.1.10 (Request)
15.5198...	Stmicroe_00:00:00	Broadcast	ARP	60	Gratuitous ARP for 192.168.1.1 (Request)
16.4935...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.0.1? Tell 192.168.1.10
17.4935...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.0.1? Tell 192.168.1.10
29.1149...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.1.1? Tell 192.168.1.10
30.1135...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.1.1? Tell 192.168.1.10
31.1135...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.1.1? Tell 192.168.1.10
32.1135...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.1.1? Tell 192.168.1.10
33.1135...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.1.1? Tell 192.168.1.10
34.1136...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.1.1? Tell 192.168.1.10
35.1136...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.1.1? Tell 192.168.1.10
36.1135...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.1.1? Tell 192.168.1.10
37.1135...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.1.1? Tell 192.168.1.10
47.3037...	Stmicroe_00:00:00	Broadcast	ARP	60	Gratuitous ARP for 192.168.1.1 (Request)
47.7464...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.0.1? Tell 192.168.1.10
48.7455...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.0.1? Tell 192.168.1.10
49.7455...	Micro-St_7f:15:ec	Broadcast	ARP	42	Who has 192.168.0.1? Tell 192.168.1.10
59.9329...	Stmicroe_00:00:00	Broadcast	ARP	60	Who has 192.168.1.10? Tell 192.168.1.1
59.9329...	Micro-St_7f:15:ec	Stmicroe_...	ARP	42	192.168.1.10 is at d8:cb:8a:7f:15:ec

Figura 30 – Captura do *Wireshark* mostrando problemas no ARP

Nas primeiras tentativas de se implementar o servidor, a aplicação era composta de dez arquivos, sendo um HTML, um CSS e oito arquivos Javascript. Observou-se que o servidor não conseguia entregá-los simultaneamente, visto que o navegador faz todas as requisições em paralelo. Na Figura 31 observa-se que o servidor fechava as conexões logo após o recebimento das requisições.

No.	Time	Source	Destination	Protocolo	Length	Info
19	5.38407...	192.168.1.10	192.168.1.1	TCP	74	40230 → 80 [SYN] Seq=0 Win=29200 Len=0
20	5.38462...	192.168.1.10	192.168.1.1	TCP	74	40232 → 80 [SYN] Seq=0 Win=29200 Len=0
21	5.40158...	192.168.1.1	192.168.1.10	TCP	590	[TCP segment of a reassembled PDU]
22	5.40159...	192.168.1.10	192.168.1.1	TCP	54	40222 → 80 [ACK] Seq=320 Ack=2681 Win=0 Len=0
23	5.42375...	192.168.1.1	192.168.1.10	TCP	590	[TCP segment of a reassembled PDU]
24	5.42376...	192.168.1.10	192.168.1.1	TCP	54	40222 → 80 [ACK] Seq=320 Ack=3217 Win=0 Len=0
25	5.46445...	192.168.1.1	192.168.1.10	TCP	60	80 → 40222 [RST, ACK] Seq=3217 Ack=3217 Win=0 Len=0
26	5.46471...	192.168.1.10	192.168.1.1	TCP	74	40234 → 80 [SYN] Seq=0 Win=29200 Len=0
27	5.47076...	192.168.1.1	192.168.1.10	TCP	60	80 → 40224 [SYN, ACK] Seq=0 Ack=1 Win=0 Len=0
28	5.47078...	192.168.1.10	192.168.1.1	TCP	54	40224 → 80 [ACK] Seq=1 Ack=1 Win=29200 Len=0
29	5.47088...	192.168.1.10	192.168.1.1	HTTP	337	GET /style.css HTTP/1.1
30	5.51746...	192.168.1.1	192.168.1.10	TCP	60	80 → 40224 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
31	5.51787...	192.168.1.10	192.168.1.1	TCP	74	40236 → 80 [SYN] Seq=0 Win=29200 Len=0
32	5.52126...	192.168.1.1	192.168.1.10	TCP	60	80 → 40226 [SYN, ACK] Seq=0 Ack=1 Win=0 Len=0
33	5.52128...	192.168.1.10	192.168.1.1	TCP	54	40226 → 80 [ACK] Seq=1 Ack=1 Win=29200 Len=0
34	5.52141...	192.168.1.10	192.168.1.1	HTTP	323	GET /svg.min.js HTTP/1.1
35	5.56202...	192.168.1.1	192.168.1.10	TCP	60	80 → 40226 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
36	5.56229...	192.168.1.10	192.168.1.1	TCP	74	40238 → 80 [SYN] Seq=0 Win=29200 Len=0
37	5.56581...	192.168.1.1	192.168.1.10	TCP	60	80 → 40228 [SYN, ACK] Seq=0 Ack=1 Win=0 Len=0
38	5.56582...	192.168.1.10	192.168.1.1	TCP	54	40228 → 80 [ACK] Seq=1 Ack=1 Win=29200 Len=0
39	5.56589...	192.168.1.10	192.168.1.1	HTTP	325	GET /FileSaver.js HTTP/1.1
40	5.57687...	192.168.1.1	192.168.1.10	TCP	60	80 → 40222 [RST, ACK] Seq=2681 Ack=3217 Win=0 Len=0
41	5.58794...	192.168.1.1	192.168.1.10	TCP	60	80 → 40222 [RST, ACK] Seq=3217 Ack=3217 Win=0 Len=0
42	5.62869...	192.168.1.1	192.168.1.10	TCP	60	80 → 40228 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
43	5.62897...	192.168.1.10	192.168.1.1	TCP	74	40240 → 80 [SYN] Seq=0 Win=29200 Len=0
44	5.63255...	192.168.1.1	192.168.1.10	TCP	60	80 → 40236 [SYN, ACK] Seq=0 Ack=1 Win=0 Len=0
45	5.63257...	192.168.1.10	192.168.1.1	TCP	54	40236 → 80 [ACK] Seq=1 Ack=1 Win=29200 Len=0
46	5.63267...	192.168.1.10	192.168.1.1	HTTP	321	GET /graph.js HTTP/1.1
47	5.63495...	192.168.1.10	192.168.1.1	TCP	74	40242 → 80 [SYN] Seq=0 Win=29200 Len=0
48	5.63496...	192.168.1.10	192.168.1.1	TCP	74	40244 → 80 [SYN] Seq=0 Win=29200 Len=0
49	5.64353...	192.168.1.1	192.168.1.10	TCP	60	80 → 40226 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
50	5.65461...	192.168.1.1	192.168.1.10	TCP	60	80 → 40228 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
51	5.69535...	192.168.1.1	192.168.1.10	TCP	60	80 → 40236 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Figura 31 – Captura mostrando problemas em múltiplas conexões em paralelo

A solução encontrada foi realizar a concatenação de todos os Javascript em um único arquivo, chamado *all.js*, além da inclusão do CSS no arquivo HTML. Desta forma a aplicação contém apenas dois arquivos, e assim o servidor consegue entregá-los sem problemas. A Figura 32 mostra a aba *network* do console de desenvolvedor do navegador Firefox após baixar a aplicação.

Status	Method	File	Domain	Ca...	Type	Tra...	Size	0 ms
● 200	GET	/	192.168.1.1	doc...	html	3,32 KB	11,74 KB	→ 1107 ms
● 200	GET	all.js	192.168.1.1	script	js	52,27 KB	309,70 ...	→ 8862 ms
●	POST	plcstate	192.168.1.1	xhr	xml	3 B	3 B	→ 8221 ms
●	POST	upload	192.168.1.1	xhr	xml	694 B	694 B	→ 8316 ms

Figura 32 – Transferência de arquivos no navegador Firefox

Como indicado na Figura 32, o *download* dos dois arquivos da aplicação levou no

No.	Time	Source	Destination	Protocol	Length	Info
725	782.197...	192.168.1.10	192.168.1.1	TCP	74	42642 → 80 [SYN] Seq=0 Win=29200 Le...
726	782.271...	192.168.1.1	192.168.1.10	TCP	60	80 → 42642 [SYN, ACK] Seq=0 Ack=1 W...
727	782.271...	192.168.1.10	192.168.1.1	TCP	54	42642 → 80 [ACK] Seq=1 Ack=1 Win=29...
728	782.271...	192.168.1.10	192.168.1.1	HTTP	392	POST /cgi/plcstop HTTP/1.1 (text/p...
729	782.522...	192.168.1.1	192.168.1.10	TCP	60	[TCP segment of a reassembled PDU]
730	782.522...	192.168.1.10	192.168.1.1	TCP	54	42642 → 80 [ACK] Seq=339 Ack=5 Win=...
731	790.420...	192.168.1.1	192.168.1.10	TCP	60	80 → 42642 [FIN, ACK] Seq=5 Ack=339...
732	790.420...	192.168.1.10	192.168.1.1	TCP	54	42642 → 80 [FIN, ACK] Seq=339 Ack=6...
733	790.504...	192.168.1.1	192.168.1.10	TCP	60	80 → 42642 [ACK] Seq=6 Ack=340 Win=...

Figura 33 – Inspeção dos pacotes da requisição de *Stop*

total 9969 ms, e os dois arquivos foram carregados corretamente. Este tempo é percebido totalmente pelo usuário, e embora não seja ideal, é bastante razoável, dado que só é necessário carregar o editor uma vez, como a abertura dos programas de edição.

Logo após foi feita a requisição pelo estado do CLP, que levou 8221 ms para ser completada. Destaca-se que este tempo não é totalmente percebido pelo usuário, pois trata-se de uma requisição assíncrona, ou seja, não congela a aplicação até sua finalização. O mesmo se aplica ao *upload*.

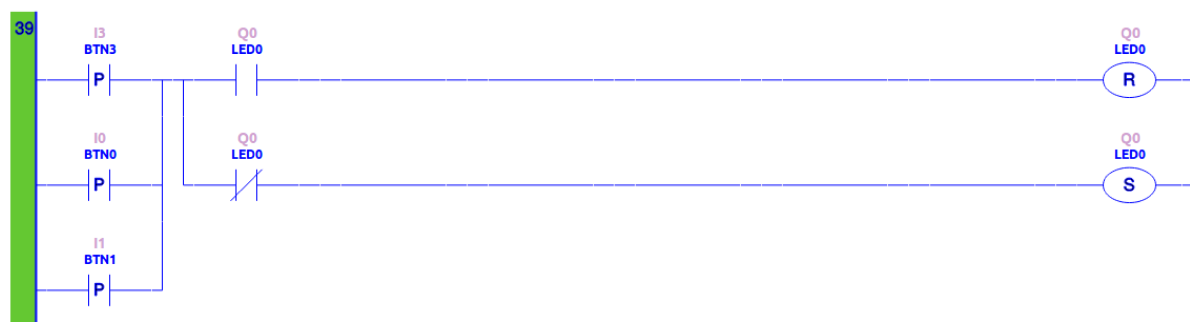
Nesta versão mais recente, logo após o carregamento a aplicação envia um requerimento para o servidor pedindo o estado do CLP para informar o usuário se o mesmo está em modo *Run* ou *Stop*, como mostrado na figura 33. O comportamento destes requerimentos pelo método POST são diferentes em cada navegador. O navegador Google Chrome não detecta as transições de *readyState* 2 para 3, pulando direto para o 4, o que não ocorre no Mozilla Firefox. Por esta razão, a aplicação só funciona perfeitamente neste navegador.

Com isso foram apresentados os principais problemas encontrados no desenvolvimento, concentrados principalmente na implementação do servidor web, relacionados à comunicação com o navegador. No entanto todas as requisições ocorrem com sucesso e é importante ressaltar que todos estes problemas são decorrentes do *lwIP* e do driver de rede fornecido pelo fabricante. Como esta implementação do servidor tem o intuito apenas de fechar a funcionalidade do protótipo como uma prova de conceito e não


```
LDR R,00
PUSH_TMP
GET_TMP
OSR I,03
AND
GET_TMP
OSR I,00
AND
OR
GET_TMP
OSR I,01
AND
OR
POP_TMP
AND
PUSH_TMP
LDR Q,00
AND
RESET B,00 Q,00
GET_TMP
LDR Q,00
NOT
AND
SET B,01 Q,00
POP_TMP
```

Esta *rung* foi repetida 40 vezes compondo a *Ladder* a ser testada. Desta forma, o *bytecode* final era composto por 1000 instruções. Como a intenção deste procedimento era apenas de se ter a noção da ordem de grandeza do tempo de execução, a aferição foi feita através do *Systick* do processador, que gera interrupções a cada 1ms.

Desta forma o tempo máximo de execução aferido para este programa foi de 3ms, um resultado plenamente aceitável para muitas aplicações, visto que um valor comum para o tempo de *scan* fixo de CLP's comerciais é de 10ms (BOLTON, 2005).

Figura 36 – *Rung* utilizada no teste de tempo de execução

5 Considerações Finais

É importante ressaltar que o protótipo desenvolvido tem o intuito de compor apenas uma prova de conceito, ou seja, é uma implementação incompleta da ideia apresentada que visa demonstrar que o projeto é exequível. Portanto não foram executados validações ou testes rigorosos, tampouco foram seguidos minuciosamente os padrões e normas técnicas referentes ao assunto, apenas suas linhas gerais. Apesar disso, os resultados e códigos desenvolvidos podem ser utilizados como base para a implementação futura de um protótipo operacional.

Posto que os objetivos traçados para esta etapa do projeto foram o desenvolvimento do editor, da máquina virtual e do servidor, atendendo a todos os requisitos discutidos, pode-se afirmar que foram atingidos. O editor permite a criação de programas simples em *Ladder* de forma rápida e intuitiva, além de não ser necessário um processo de compilação, pois não permite a criação de programas inválidos. A máquina virtual executa adequadamente o código gerado, realizando todas as funções pretendidas. O servidor, apesar de todos os problemas apresentados, cumpre seu papel de servir a aplicação e prover a troca de dados entre o navegador e o dispositivo.

Assim, como proposto, pelo pequeno tamanho final da aplicação web (40 KB) e pela portabilidade da máquina virtual, este sistema é potencialmente compatível com qualquer sistema embarcado que suporte *ethernet* e contenha o mínimo necessário de memória não-volátil. Este último requisito pode ser facilmente contornado com a adaptação do projeto para comportar memórias externas. Com isso, podem ser desenvolvidas plataformas de baixo custo com aplicações didáticas ou de automação residencial, por exemplo.

Como discutido inicialmente, o protótipo está longe do que deve ser um produto final, existem características fundamentais a um CLP que ainda não puderam ser implementadas, por exemplo a garantia de tempo de *scan* e suporte a redes industriais, além da maior diversidade de elementos necessária.

Com a aplicação estabelecida, é relativamente simples adicionar suporte a novos elementos básicos de *Ladder*. Simplificadamente, para criar um novo elemento basta criar um novo símbolo, um novo tipo na estrutura da dados, adicionar um *opcode*, e implementar a função deste elemento na máquina virtual. Desta forma, o desenvolvimento do projeto deve abranger a criação de diversos novos elementos, como por exemplo elementos que façam a leitura de valores analógicos.

Além da expansão dos elementos suportados pela aplicação, em continuidade ao projeto poderiam ser feitas diversas melhorias, como a integração de um sistema

operacional de tempo real, implementação de um modo de monitoramento para o editor de *Ladder*, e a execução de *Ladders* em paralelo, por exemplo.

Referências

- BELLAMY-ROYDS, A.; CAGLE, K. *Using SVG with CSS3 and HTML5: Vector Graphics for Web Design*. 1a. ed. Sebastopol, CA, EUA: O'Reilly, 2017. Citado na página [31](#).
- BOLTON, W. *Programmable Logic Controllers*. 5a. ed. Oxford, UK: Newnes, 2005. Citado 2 vezes nas páginas [12](#) e [70](#).
- BRYAN, L. A.; BRYAN, E. A. *Programmable Controllers: Theory and Implementation*. 5a. ed. Atlanta, GA, EUA: Industrial Text, 1997. Citado 3 vezes nas páginas [11](#), [13](#) e [16](#).
- CRAIG, I. D. *Virtual Machines*. 1a. ed. [S.l.]: Springer, 2006. Citado na página [16](#).
- FLANAGAN, D. *Javascript: The Definitive Guide*. 1a. ed. Sebastopol, CA, EUA: O'Reilly, 2011. Citado na página [27](#).
- GOURLEY, D.; TOTTY, B. *HTTP: The Definitive Guide*. 1a. ed. Sebastopol, CA, EUA: O'Reilly, 2009. Citado na página [23](#).
- INTERNATIONAL ELECTROTECHNICAL COMISSION. *IEC 61131-3:2013: Programmable controllers - part 3: Programming languages*. [S.l.], 2013. Citado na página [13](#).
- JOHN, K.-H.; TIEGELKAMP, M. *IEC 61131-3: Programming Industrial Automation Systems*. 2a. ed. Alemanha: Springer, 2010. Citado na página [38](#).
- KOZUCH, M.; WOLFE, A. Compression of embedded system programs. *IEEE*, 2002. Citado na página [23](#).
- POWELL, T. A. *AJAX: The Complete Reference*. 1a. ed. [S.l.]: McGraw Hill, 2008. Citado na página [61](#).
- PRUDENTE, F. *Automação industrial: PLC: teoria e aplicações*. 2a. ed. [S.l.]: LTC, 2011. Citado na página [11](#).
- ROBBINS, J. N. *Learning Web Design*. 4a. ed. Sebastopol, CA, EUA: O'Reilly, 2012. Citado na página [23](#).
- SHI, Y. et al. Virtual machine showdown: Stack versus registers. *VEE: First International Conference On Virtual Execution Environments*, 2005. Citado na página [16](#).
- SMITH, J. E.; NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes*. 1a. ed. San Francisco, CA, EUA: Morgan Kaufmann, 2005. Citado na página [15](#).