UNIVERSIDADE ESTADUAL DE FEIRA DE SANTANA – UEFS

NILS ALEXANDRE LIMA BERGSTEN

UM COMPILADOR PORTUGOL-ASSEMBLY PARA MICROCONTROLADOR

FEIRA DE SANTANA-BA
2012

NILS ALEXANDRE LIMA BERGSTEN

UM COMPILADOR PORTUGOL-ASSEMBLY PARA MICROCONTROLADOR

Trabalho de Conclusão de Curso apresentado ao Colegiado de Engenharia de Computação da Universidade Estadual de Feira de Santana, como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Thiago D'Martin Maia

FEIRA DE SANTANA-BA
2012

RESUMO

O ensino de programação para comunidades lusófonas é dificultado pelo fato de que a maioria das linguagens de programação tem suas palavras reservadas em inglês. Disso surge a necessidade de se criarem ferramentas que permitam o desenvolvimento e a compilação diretamente a partir de algoritmos editados em portugol. Aliado a isso, diversas experiências têm sido realizadas no ensino de programação para crianças por meio da robótica educativa, demonstrando diversos benefícios desta prática. A ideia neste trabalho é criar uma ferramenta que possibilite a programação de microcontroladores PIC utilizando o português estruturado, facilitando o ensino de computação, além da possibilidade de uso como compilador de português estruturado, em disciplinas introdutórias de cursos de programação de computadores.

Palavras-chave: Tradutor; Compilador; Portugol; *Assembly*; Microcontroladores; Educação.

ABSTRACT

The teaching of computer's programming for lusophone communities is hindered by the fact that most programming languages have their keywords in english. From this occurs the need to create tools that enable the development and compilation directly from algorithms written in portugol. Indeed, several experiments have been conducted in the teaching of programming for children through educational robotics, showing many benefits of this practice. The idea in this work is to create a tool that enables the programming of PIC microcontrollers using structured portuguese to facilitate the teaching of computing, beyond the possibility of use it as structured portuguese compiler, in introductory programming courses.

Keywords: Translator, Compiler; Portugol; Assembly; Microcontrollers; Education.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Como as linguagens de programação de computadores mais difundidas no mercado e na academia, em geral, apresentam a edição de seus códigos, em nível cognitivo humano, na língua inglesa, nos países anglófonos o ensino de construção de algoritmos se dá no uso daquelas linguagens como especificações diretas, ou muito próximas disso, dos próprios algoritmos tratados. No ensino de algoritmos nos níveis básico, fundamental e superior em países lusófonos, há décadas se tem utilizado como método inicial (e anterior ao uso de linguagens de programação propriamente ditas) a especificação de instruções algorítmicas ou pseudocódigos em português, o chamado portugol. A partir de tal especificação inicial do algoritmo idealizado, em português, parte-se para a tradução humana ou natural desse algoritmo em um código de fato compilável e, consequentemente, executável por uma máquina. A diferença entre resultados de aprendizagem nesses referidos modos de ensino é que, no caso dos estudantes lusófonos, a execução de código que se obtém não é diretamente resultante da especificação original em português, o que tende a causar dúvidas, deficiência de aproveitamento e até abandono de disciplinas introdutórias em cursos de graduação (GOMES, 2007).

Segundo (ARAÚJO, 2009), o ideal é que se tenha uma ferramenta de ensino de construção de algoritmos, para comunidades lusófonas, estabelecida e utilizada em uma plataforma livre. Isso significa um editor de algoritmos em portugol associado a um tradutor (compilador) dessa especificação para o nível de código executável por máquina, em linguagem *assembly*. Existem alguns compiladores com tal característica, sendo o G-Portugol (SILVA, 2006) o mais difundido e aquele que servirá como termo de comparação de uso para a ferramenta desenvolvida neste projeto.

O que caracteriza o desenvolvimento deste projeto é a especificação da gramática de um dialeto básico do português, sua implementação em um compilador de seis fases (AHO, et al., 2007) voltado para *hardware* embarcado, primeiro para fim de ensino de construção de algoritmos e com sua utilização posterior no ensino auxiliado por robótica, pois a compilação para *hardware* embarcado oferecerá, além da aprendizagem de construção de algoritmos escritos em português e desta língua, traduzidos ou compilados, a possibilidade de se criarem brinquedos didáticos para estimular a criatividade e a solução de problemas de forma concreta. Esses brinquedos são kits de robótica ou minilaboratórios de experiências com eletrônica, entre outros, que podem inclusive ser utilizados em sala de

aula, de forma a auxiliar o professor em sua prática pedagógica (AZEVEDO, et al., 2010). O microcontrolador PIC (*Programmable Interrupt Controller* ou controlador de interrupção programável – ou ainda *Peripheral Interface Controller*, controlador de interface periférico) foi escolhido para uso no projeto por ser amplamente utilizado em computação embarcada e consequentemente em robótica (Microchip Technology Inc., 2001).

A perspectiva central no projeto foi desenvolver uma linguagem de programação de sintaxe simples que tem como características léxicas (AHO, et al., 2007; LOUDEN, 2004; GRUNE, et al., 2002) palavras reservadas em português, sendo de fácil aprendizado para pessoas de todas as idades, além de ser compilável em *hardware* embarcado (um microcontrolador PIC), apresentando assim utilização com *kits* de robótica.

Esta perspectiva surgiu considerando-se que experimentos realizados com crianças de até sete anos de idade mostram que, por exemplo, o ensino do Prolog, linguagem de programação lógica, a jovens dessa idade oferece diversos benefícios aos mesmos. Entre esses benefícios podem-se citar maior clareza de raciocínio, maior facilidade na expressão verbal e aumento do desempenho em disciplinas como matemática, lógica e gramática (SEBESTA, 2011). Outra experiência bastante utilizada é a do LEGO *Mindstorms* para jovens de 11 a 14 anos. Segundo (GOMES, 2007), "nesta fase, o aluno começa a desenvolver o raciocínio abstrato, é capaz de criar hipóteses mais elaboradas e inferir consequências. [...] O aluno desenvolve, com a ajuda de um computador, a capacidade de pensar sobre o pensamento (metacognição)". A Robótica Educativa "[...] estimula a criatividade dos alunos devido a sua natureza dinâmica, interativa e até mesmo lúdica, além de servir de motivador para estimular o interesse dos alunos no ensino tradicional".

Assim, supomos que o mesmo ocorra com uso de linguagens procedimentais, mais notadamente se os códigos base forem escritos em dialeto da língua primeira do aluno, tendo sua execução em *hardware* embarcado voltado para experimentos educacionais que inclusive envolvam robótica.

O objetivo deste trabalho é a implementação de um tradutor/ compilador de portugol para *assembly*, voltado para ensino de construção de algoritmos, inclusive com uso de *hardware* embarcado para robótica. Para isso foi necessária a especificação formal da gramática de um dialeto do português, a comparação dos efeitos derivativos dessa gramática com as características do G-Portugol e de outras implementações afins, a implementação da linguagem a partir de sua tradução/ compilação para microcontroladores

PIC, a coleta de resultados com testes (em *hardware* embarcado) e a especificação de estudos de caso futuros com robótica.

No segundo capítulo deste trabalho, relataremos a fundamentação teórica do mesmo, abordando temas como educação, compiladores e microcontroladores. No terceiro capítulo é descrita a metodologia, detalhando o processo de especificação e implementação da linguagem. No quarto capítulo, se discutem os resultados obtidos nos testes realizados. No quinto capítulo, se apresenta a elaboração de uma possível aplicação prática. No sexto capítulo é feita a conclusão, seguida das referências utilizadas.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 EDUCAÇÃO EM INFORMÁTICA

(ARAÚJO, 2009) destaca diversos problemas relativos ao ensino de algoritmos nos cursos de computação, e aponta como solução ferramentas de desenvolvimento em uma linguagem derivada do português estruturado. Segundo o mesmo, o G-Portugol é o projeto com maior potencial para ser utilizado em cursos de computação nas disciplinas de algoritmos em ambientes GNU/ Linux, caracterizando funcionalmente a linguagem, para justificar a sua escolha:

- Desenvolvida em C++;
- Licença GPL;
- Atende a compilação e a interpretação;
- Contém editor;
- Nativo para GNU/ Linux, mas também executável em Windows;
- Documentação excelente;
- Projeto ativo;
- Usabilidade excelente;

(ARAÚJO, 2009) definiu também diretrizes para ferramentas de aprendizagem de algoritmos para comunidades lusófonas:

1. A linguagem deve ser totalmente em dialeto do português e respeitar acentuações;
2. Deve ser simples, uniforme e coerente;
3. Deve ser configurável a diferentes abordagens de ensino.

Ainda segundo (ARAÚJO, 2009), o G-Portugol atende parcialmente à primeira diretriz, pois as palavras reservadas respeitam as acentuações, mas as variáveis não; atende à segunda diretriz, pois tem diversos aspectos positivos em sua sintaxe; mas não atende à terceira diretriz.

O produto deste trabalho visa a obedecer pelo menos em parte às diretrizes citadas, de modo que possa ser utilizado nas disciplinas introdutórias de algoritmos e programação. Como se trata de compilação voltada a *hardware* embarcado, o projeto também poderá ser utilizado em outras disciplinas da área, como robótica, sistemas embarcados etc.

2.2 INFORMÁTICA NA EDUCAÇÃO

Assim como este trabalho pode ser utilizado no ensino de computação, ele objetiva também ser utilizado como uma ferramenta de auxílio para o ensino na educação de base.

Segundo (RIBEIRO, et al., 2007), o uso de tecnologia no processo educativo se fundamenta na teoria de Seymour Papert, designada como *construcionismo,* teoria baseada no construtivismo de Piaget, mas que acrescenta a importância de construções físicas como suporte à construção intelectual. Papert defende que os seres humanos aprendem melhor quando estão envolvidos no planejamento e na construção de artefatos concretos, e por isso tecnologias como o microcomputador, *kits* de robótica e *kits* de eletrônica podem ser ferramentas extremamente úteis na educação de base.

Papert estabeleceu quatro pilares de sua teoria (RIBEIRO, et al., 2007):

- Aprender construindo, envolvendo o educando na tomada de decisões, resolução de problemas e trabalho cooperativo;
- Usar objetos concretos, que podem ser utensílios tecnológicos;
- Usar ideias poderosas, ferramentas intelectuais que permitam novas formas de pensar;
- Realizar uma autorreflexão sobre o processo, que pode ser bastante apoiada por uma boa documentação do mesmo.

A robótica educativa, segundo (RIBEIRO, et al., 2007), apresenta diversas potencialidades que podem ser exploradas no ensino básico:

- Motivação e entusiasmo dos alunos, por ser uma atividade lúdica que facilmente desperta o interesse;
- Interdisciplinaridade, podendo envolver diversas matérias como matemática, física, ciências, artes etc.;

- Resolução de problemas;
- Trabalho em equipe e competências de comunicação;
- Imaginação e criatividade, para se criar, inovar e resolver os problemas;
- Raciocínio lógico e pensamento abstrato.

Este projeto consiste no desenvolvimento de uma ferramenta que também possibilite a programação de dispositivos eletrônicos microcontrolados de baixo custo, utilizando uma linguagem mais acessível para comunidades lusófonas, visando a seu uso na educação assistida por tecnologia, tanto por iniciativa de escolas quanto de pais, assim como ocorre com os brinquedos educativos LEGO Mindstorms.

2.3 COMPILADORES

Para produzir a ferramenta de nosso objetivo, é necessário implementar um compilador de uma especificação do português estruturado, tendo como linguagem-alvo o *assembly* do dispositivo microcontrolador. Um compilador, de acordo com (AHO, et al., 2007; LOUDEN, 2004; GRUNE, et al., 2002), pode ser divido em seis etapas que serão descritas a seguir.

2.3.1 ANÁLISE LÉXICA

A análise léxica é a primeira fase na execução de um compilador. A função do analisador léxico, também chamado de *scanner*, é ler o código fonte, caractere a caractere, separando-o em componentes significativos, denominados símbolos léxicos ou *tokens*, que constituem o dicionário reservado da linguagem. Assim se determina se os símbolos contidos pelo código fonte são válidos ou não lexicamente.

A palavra *token* pode ser traduzida em diversas formas, sendo que a tradução mais comum é “símbolo”. Esta palavra pode ser usada no sentido de um passe ou uma ficha. Em redes de computadores, assume o sentido de uma senha ou identificador, para garantir segurança de dados. Assim, definimos *token* como um símbolo que segue um padrão determinante do contexto de sua aceitação ou do contrário. Cada símbolo terminal da gramática, encontrado no código-fonte, é portanto um *token*.

O léxico de uma linguagem é designado por uma gramática regular. Uma gramática regular é aquela cujos símbolos não-terminais produzem no máximo um símbolo terminal e um símbolo não-terminal, representando uma linguagem regular. Um autômato finito, ou uma máquina de estados finita, é o modelo mais simples capaz de representar uma linguagem regular. Por conseguinte, o analisador léxico é construído a partir de uma máquina de estados finita baseada no léxico da linguagem, de modo que possa reconhecê-lo. (LOUDEN, 2004)

Supondo que temos um símbolo léxico definido pela expressão regular , um autômato reconhecedor deste símbolo é indicado na Figura 1.

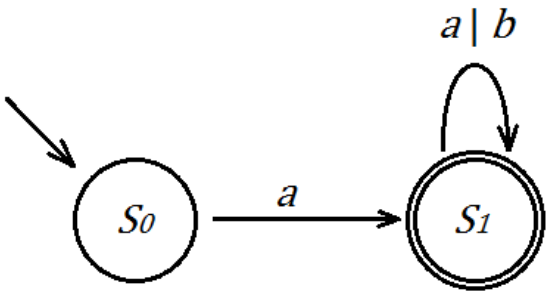

Figura 1: Exemplo de Autômato Finito

Para desenvolver, portanto, o analisador léxico, se faz necessário estabelecer as regras de formação de cada símbolo léxico, e a partir dessas projetar um autômato correspondente. Assim, o analisador léxico ou *scanner*, ao ler um caractere, inicia a verificação sobre se o que segue ao mesmo obedece a regras estabelecidas, aceitando ou não essa parte o código-fonte.

Durante a fase de análise léxica, são desconsiderados para tratamento posterior os elementos de efeito apenas cognitivo do código-fonte, como espaços em branco e comentários. A saída do analisador léxico é uma lista de *tokens*, que é utilizada na construção da tabela de símbolos, que guardará o tipo de cada variável e os procedimentos declarados, e na análise sintática.

2.3.2 ANÁLISE SINTÁTICA

A análise sintática é o processo de se determinar se a sequência de símbolos léxicos pode ser gerada pela gramática livre de contexto correspondente à linguagem envolvente. Uma gramática livre de contexto é aquela cujos símbolos não-terminais produzem uma cadeia de símbolos terminais e não-terminais definidores de escopo, representando uma linguagem livre de contexto. O analisador sintático, ou *parser*, é o cerne do compilador, sendo responsável por verificar se os símbolos do programa fonte estão devidamente organizados, formando um programa válido.

Uma gramática livre de contexto é geralmente representada na forma de Backus–Naur (BNF), cuja principal característica é ter os símbolos não-terminais representados entre “<” e “>”.

A saída do analisador sintático é uma árvore sintática abstrata (ASA). Uma árvore sintática é uma estrutura de dados em árvore que representa a estrutura sintática da cadeia de entrada de acordo com a gramática especificada. A raiz de uma árvore sintática é o símbolo inicial da gramática, e, nas extremidades da mesma, ou seja, nas folhas, se encontram os símbolos léxicos obtidos do programa fonte, ou seja, os símbolos terminais da gramática. A Figura 2 ilustra como seria a árvore sintática para a expressão a partir da gramática implementada, anexa.

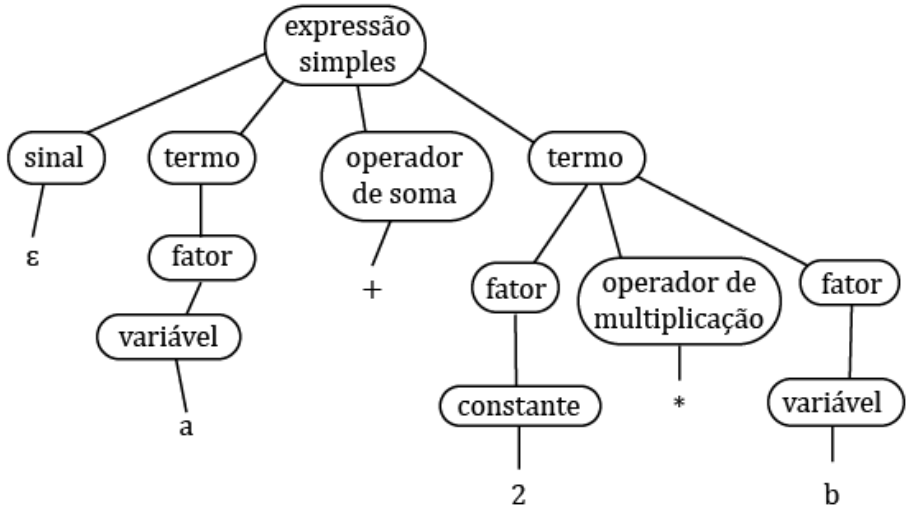

Figura 2: ASA para a expressão

O analisador sintático pode ser implementado de duas formas: descendente (*top-down*), iniciando-se com o símbolo "de disparo" e buscando derivá-lo até a entrada de dados, o arquivo de texto; e ascendente (*bottom-up*), iniciando-se com a entrada de dados e tentando reduzi-la até o símbolo "de disparo" da gramática. Nos analisadores *top-down*, a construção da árvore sintática começa pela raiz e prossegue em direção às folhas, enquanto que nos analisadores *bottom-up*, a construção começa pelas folhas e prossegue em direção à raiz. A maior utilização do *top-down* deve-se ao fato de que um *parser* eficiente pode ser implementado mais facilmente à mão, nesse caso. Analisadores *bottom-up*, no entanto, podem lidar com uma classe maior de gramáticas e esquemas de tradução, e por isso ferramentas de *software* para a geração automática de analisadores de gramáticas costumam usar métodos *bottom-up*. Uma gramática é dita LL(1) quando analisa o código de entrada da esquerda para a direita, e deriva as produções também da esquerda para a direita; além disto, é necessário consultar apenas um símbolo adiante para se tomarem decisões. (AHO, et al., 2007)

2.3.3 ANÁLISE SEMÂNTICA

Após a verificação sintática do código fonte, é necessário verificar se o código pode ser executado de acordo com sua organização. O papel do analisador semântico é fazer esta análise com o uso da tabela de símbolos e da ASA, e prover uma maneira do código ser executado. (LOUDEN, 2004)

Os analisadores léxicos representam gramáticas regulares, os analisadores sintáticos são capazes de representar gramáticas livres de contexto, e os analisadores semânticos representam gramáticas sensíveis ao contexto. Quando uma instrução passa pelo *parser*, o analisador semântico verifica se a mesma faz sentido dentro do contexto tratado. A instrução , por exemplo, é aprovada pelo analisador léxico, já que os *tokens* pertencem ao dicionário da linguagem, porém não passa pelo *parser*, já que não exercem funções sintáticas de acordo com a gramática. Já a instrução , por exemplo, é aprovada pelo *parser*, mas pode não passar pelo analisador semântico, que examinará o contexto da instrução, verificando, por exemplo, se as variáveis e foram declaradas como inteiros.

Segundo (LOUDEN, 2004), as principais funções da análise semântica são a verificação do tipo de dado de uma variável, a verificação do valor de uma expressão, a

verificação da localização de uma variável na memória, a verificação do código-objeto de um procedimento e a verificação do número de algarismos significativos em um número.

2.3.4 GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

Na fase de geração de código intermediário, a ASA é transformada em um código em uma linguagem próxima a *assembly* (código objeto), de forma que possa ser mais facilmente manipulada para a otimização do código. A própria ASA também pode ser considerada uma linguagem intermediária.

A forma mais comum de representar a linguagem intermediária é por meio da quádrupla: operador, endereço do primeiro operando, endereço do segundo operando e endereço do resultado. O endereço do resultado pode ser omitido em algumas formas de representação, caso o mesmo seja armazenado no primeiro operando, conforme a arquitetura alvo.

Criada para compiladores de Pascal, sendo muito utilizada principalmente nas décadas de 70 e 80, outra forma de representar o código intermediário é utilizando o P-Código. Esta forma de representação é uma linguagem de apenas um endereço orientada a uma máquina hipotética e baseada em pilhas, as quais abstraem as variáveis auxiliares, como será explicado adiante. Até hoje é utilizada em alguns compiladores por ser portável e de fácil transformação para *assembly*, apesar de deixar o código maior que o ideal (LOUDEN, 2004).

Segue um exemplo de P-Código para a expressão .

```
ldc 2     ;carrega constante 2
lod a     ;carrega valor da variável a
adi       ;adição de inteiros
lod b     ;carrega valor da variável b
ldc 3     ;carrega constante 3
sbi       ;subtração de inteiros
mpi       ;multiplicação de inteiros
```

Uma operação lógica e aritmética em P-Código é sempre feita entre os dois últimos valores carregados na pilha. Estes são retirados durante a operação, e o resultado é

guardado na pilha. No exemplo anterior, são carregados os valores “2” e “a” na pilha e, seguindo, é feita a operação de soma. O efeito desta operação é que os valores usados são retirados da pilha e o resultado “2+a” é guardado na mesma. A subtração no exemplo é feita entre os dois outros valores a serem carregados na pilha, “b” e “3”, sendo estes valores retirados e sendo o resultado guardado na pilha. A multiplicação também é realizada entre os dois próximos valores considerados por essa ordem, que neste caso são os dois resultados anteriores.

Para implementar uma expressão aritmética em *assembly*, quase sempre se faz necessária a utilização de variáveis auxiliares. As posições da pilha do P-Código abstraem as variáveis auxiliares, no sentido de que cada posição utilizada da pilha corresponde a uma variável auxiliar. No exemplo descrito, foram utilizadas três posições da pilha, e portanto seriam usadas três variáveis auxiliares, uma para cada posição. A instrução de soma, por exemplo, guardaria na primeira variável auxiliar o resultado da soma entre a primeira e a segunda auxiliares.

2.3.5 OTIMIZAÇÃO DE CÓDIGO

A otimização de código consiste em examinar o código intermediário e gerar um código melhor. Um código melhor geralmente significa mais rápido, porém também pode significar código menor, ou que use menos acessos à memória.

Segundo (RANGEL, 2000), das oportunidades de otimização de código pode-se citar: a eliminação de subexpressões comuns; a eliminação de código “morto”, isto é, código que não pode ser alcançado durante a execução de um programa; a renomeação de variáveis temporárias; as transformações baseadas em propriedades algébricas, como a comutatividade e a associatividade; entre outras.

2.3.6 GERAÇÃO DE CÓDIGO FINAL

A geração de código final é a última fase da compilação. A saída desta fase é o código-objeto na linguagem alvo, o *assembly*. Para cada instrução do código intermediário, é feita a conversão para a correspondência no código-objeto. Como o P-Código abstrai as variáveis auxiliares com a pilha, algumas vezes a correspondência em *assembly* a uma instrução em P-Código será mais de uma instrução, caso a máquina-alvo não seja de 0-

endereço. Para esta fase, é necessário ter compreensão da arquitetura alvo, pois isso será refletido na forma com que cada instrução de código intermediário será convertida para código-alvo, assim como determinará as limitações do código.

2.4 O MICROCONTROLADOR PIC

2.4.1 DEFINIÇÃO

Microcontroladores são componentes eletrônicos que possuem os periféricos mais comuns dos microprocessadores embutidos no *chip*. Costumam apresentar em uma única pastilha memórias de dados e de programa, canal serial, temporizadores, interfaces de entrada e saída, memória EEPROM etc. (FONTANIVE, 1999).

O microcontrolador PIC é um circuito integrado produzido pela Microchip Technology, Inc. É essencialmente um controlador de entrada e saída baseado na arquitetura Harvard, que será explicada adiante. Segundo (FONTANIVE, 1999) e (Microchip Technology Inc., 2001), o PIC dispõe de todos os elementos tipicamente presentes em um sistema microprocessado, ou seja:

- Uma unidade central de processamento (UCP);
- Uma memória programável (FLASH);
- Uma memória RAM;
- Diversos pinos de E/ S;
- Diversos dispositivos auxiliares, como gerador de *clock*, barramento, contador etc.;
- Registradores.

2.4.2 ARQUITETURA

A maioria dos microprocessadores e alguns microcontroladores tem sua arquitetura baseada em Von Neumann, a qual prevê um barramento único de comunicação entre a UCP e as memórias. O PIC, ao contrário, é baseado na arquitetura Harvard, "que prevê várias vias de comunicação entre CPU e periféricos" (FONTANIVE, 1999), o que aumenta

a velocidade por permitir que algumas operações sejam feitas simultaneamente. A Figura 2 mostra o diagrama de blocos do PIC16F84A.

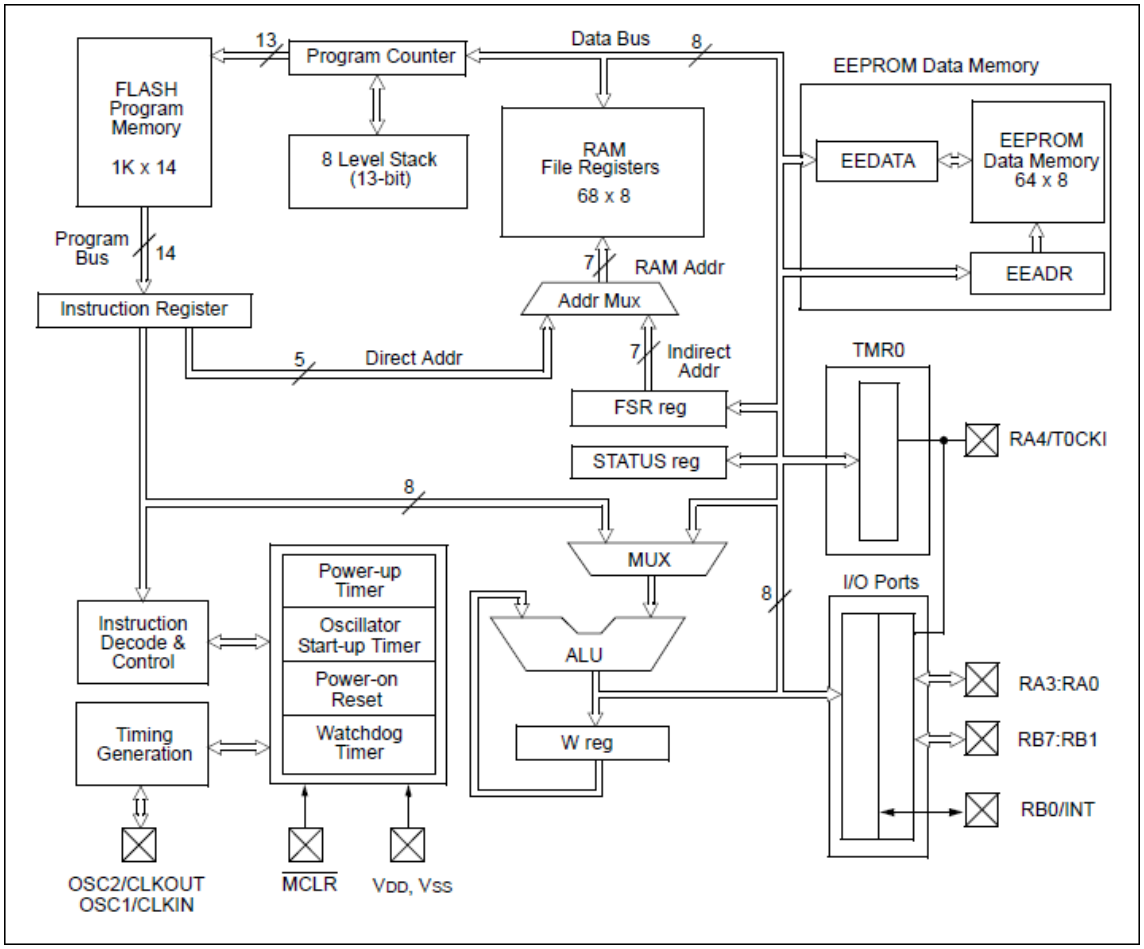

Figura 3: Arquitetura do PIC16F84A
Fonte: (Microchip Technology Inc., 2001)

A memória de programa contém 1K palavras, o que corresponde a 1024 instruções, uma vez que cada palavra de 14 bits da memória do programa é da mesma largura que cada instrução do dispositivo. A memória de dados (RAM) contém 68 bytes. Há também 13 pinos de entrada e saída que são configurados pelo usuário. Alguns pinos são multiplexados com outras funções do dispositivo. Estas funções incluem: (Microchip Technology Inc., 2001).

- Interrupção externa;
- Mudança na interrupção PORTB;
- Entrada de clock Timer0.

O PIC também pode ser classificado como um processador RISC, pois apresenta um conjunto reduzido de instruções. Segundo (FONTANIVE, 1999), este conjunto reduzido facilita sua programação e o processamento.

2.4.3 O MICROCONTROLADOR PIC16F84A

A seguir, uma descrição mais detalhada do microcontrolador PIC16F84A, que será utilizado neste projeto, baseada no *data sheet* do mesmo, que pode ser encontrado em (Microchip Technology Inc., 2001).

2.4.3.1 PINOS

RAn, n:0-3. É um pino de I/ O programável em entrada ou saída da unidade. Corresponde ao BIT n da PORTA A.

RA4/ RTCC ou T0CKI. É um pino multifuncional que pode ser programado como uma linha normal de I/ O ou como linha de *clock* para entrada, em sentido ao contador RTCC ou TMR0. Se programado como pino de I/ O corresponde ao BIT 4 da PORTA, ao contrário de outra linha de I/ O. Quando esta linha funciona como saída, trabalha em coletor aberto.

RB0. É um pino de I/ O programável em entrada ou em saída. Corresponde ao BIT n da PORTB e pode ser programado para gerar interrupção.

RBn, n:1-7. É um pino de I/ O programável em entrada ou em saída. Corresponde ao BIT 0 da PORTA B.

MCLR/ VPP. Em condição normal de funcionamento desenvolve a função de Master CLeaR, ou seja, *reset*, sendo ativo em nível 0. Pode ser conectado a um circuito de *reset* externo ou simplesmente conectado ao VCC/ VDD da alimentação. Quando o PIC estiver posto em *Program Mode*, será utilizado como entrada para a tensão de programação Vpp.

VSS. É o pino que se conecta ao terra da tensão de alimentação.

VDD. É o terminal VCC/ VDD de alimentação do PIC. Nas três versões disponíveis do PIC16F84 (comercial, industrial e automotiva), a tensão pode assumir um valor que vai de um mínimo de 2.0 volts a um máximo de 6.0 volts.

OSC1/ CLKIN. É um pino de conexão para o caso de se utilizar um cristal de quartzo ou um circuito RC para gerar o *clock*.

OSC2/ CLKOUT. É um pino de conexão para o caso de se utilizar um cristal de quartzo para gerar o *clock*. E como saída de *clock* caso seja aplicado um oscilador RC externo na entrada OSC1.

2.4.3.2 REGISTRADORES PARA PROPÓSITO ESPECÍFICO

INDF. Endereçamento indireto
TMR0. Registrador de contagem do *timer* 0
PCL. Parte baixa do contador de programa
STATUS. Registrador de *status* para controle da CPU
FSR. Ponteiro para o endereçamento indireto
PORTA. Registrador dos pinos do PORTA
PORTB. Registrador dos pinos do PORTB
EEDATA. Dado lido/ gravado na EEPROM
EEADR. Endereço para ler/ gravar na EEPROM
PCLATH. Parte alto-contadora de programa
INTCON. Registrador INTCON para controle da CPU
OPTION. Registrador OPTION para controle da CPU
TRISA. Direção dos pinos do PORTA
TRISB. Direção dos pinos do PORTB
EECON1. Controle da EEPROM
EECON2. Controle da EEPROM

2.4.3.3 CONJUNTO DE INSTRUÇÕES

2.4.3.3.1 LEGENDA

W. As operações lógicas e aritméticas são realizadas por um bloco chamado de ULA (unidade lógica e aritmética) que possui um registrador próprio chamado W (*Working register* ou acumulador).

f. File, registrador que está sendo acessado; indica uma posição do banco de registradores implementado em memória RAM, mais os registradores para propósito específico. Como este registrador tem apenas 7 *bits*, ele endereça até 128 registradores, mas com o recurso do MPlab, *software* utilizado para desenvolvimento de aplicações para sistemas embarcados, usando a diretiva EQU, *equate*, podemos escrever os nomes dos registradores que a Microchip sugere (como por exemplo: STATUS, OPTION, TRISB), sendo que com o *equate* o compilador substitui o nome do registrador pelo seu identificador numérico, ao gerar o código hexadecimal.

d. Destino, indica para onde deve ir o resultado da operação. Devido ao conjunto reduzido de instruções e a que uma instrução já contém seu *opcode*, o operando e o destino, só existem duas possibilidades de destino do resultado:

- d = 0, o resultado será armazenado no registrador W;
- d = 1, o resultado será armazenado no registrador f.

b. Indica qual *bit* do registro está sendo acessado, podendo-se usar o mnemônico do *bit* ou o número do *bit* (0 - 7), notando-se que há apenas três *bits* disponíveis nas instruções orientadas a *bit*.

K. Pode ser uma constante numérica ou uma indicação de posição de memória de programa:

- Constante numérica de 8 *bits* (0 - 255), valor literal para as operações matemáticas e lógicas;
- Indicação de posição, apenas nas instruções CALL e GOTO, possui 11 bits (0 - 2047), podendo ser um número imediato, ou um *label*, e nesse caso toda vez que o compilador encontra um *label* no programa, troca por um número na sequência crescente. Por exemplo: Numa instrução GOTO *repete*, se o *repete* é o primeiro *label*, o compilador troca por GOTO 000.

x. Independe, pode ser zero ou 1, o compilador da Microchip gera zero, e recomenda isso.

Constante decimal. D'valor' ou d'valor'. Exemplo: 55 decimal equivale a d'55' ou ainda a D'55'.

Constante hexadecimal. 0Xvalor ou valorH. Exemplo: 3B hexa = 0x3B ou 3Bh ou 3BH. Importante: no caso da constante começar com letra, devemos colocar um zero antes. Exemplo: E3h fica 0E3h.

Constante binária. B'valor' ou b'valor'. Exemplo: 10101010 binário = b'10101010'.

2.4.3.3.2 INSTRUÇÕES ORIENTADAS A BYTES

ADDWF f,d. Faz a soma entre W e f (r = W + f). Se d = 1, o resultado r será armazenado no registrador f. Se d = 0, será armazenado em W.

ANDWF f,d. Faz a operação lógica AND entre W e f (r = W AND f). Se d = 1, o resultado r será armazenado no registrador. Se d = 0, será armazenado em W.

CLRF f. Zera o conteúdo do registrador f.

CLRW. Zera o conteúdo de W.

COMF f,d. Complementa f: os *bits* do registrador f são complementados em 1, ou seja, invertidos. Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

DECF f,d. Decrementa f: decrementa o conteúdo do registrador f em 1 (r = f -1). Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

DECFSZ f,d. Decrementa f e salta, se f = 0: decrementa o conteúdo do registrador f em 1 (r = f - 1), e ainda testa se o resultado chegou a zero. Se o resultado chegar a zero, salta uma linha de instrução. Se d = 1, o resultado será armazenado no registrador f. Se d = 0, será armazenado em W.

INCF f,d. Incrementa f: esta instrução incrementa o conteúdo do registro f em 1 (d = f + 1). Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

INCFSZ f,d. Incrementa f, e salta se f = 0: esta instrução incrementa o conteúdo do registro f em 1 (r = f + 1) e testa o resultado, saltando uma linha de instrução se o mesmo for igual a zero. Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

IORWF f,d. Função OR entre W e f (r = W OR f). Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

MOVF f,d. Move para f: esta instrução copia o conteúdo do registrador f para W ou para ele mesmo. Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

MOVWF f. Move W para f: copia o conteúdo de W para o registrador f (faz f = W).

NOP. Nenhuma operação: esta instrução só serve para tomar o tempo equivalente a um ciclo de máquina.

RLF f,d. Desloca uma vez os *bits* do registrador f à esquerda, passando pelo *flag* de *carry* C. Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

RRF f,d. Desloca uma vez os *bits* do registrador f à direita, passando pelo *flag* de *carry* C. Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

SUBWF f,d. Subtrai W de f: faz a operação aritmética de subtração do conteúdo de W do conteúdo do registrador f (r = f - W). Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

SWAPF f,d. Troca os *nibbles* em f: esta instrução troca os 4 *bits* mais significativos pelos 4 menos significativos no registro f . Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

XORWF f,d. OU exclusivo entre W e f (r = f W). Se d = 1, o resultado é armazenado no registrador f. Se d = 0, o resultado é armazenado em W.

2.4.3.3.3 INSTRUÇÕES ORIENTADAS A BITS

BCF f,b. Zera o *bit* de f: esta instrução coloca o b-ésimo (b=0,...,7) bit de f no nível lógico zero.

BSF f,b. Seta o *bit* de f: a instrução coloca o b-ésimo (b=0,...,7) bit de f no nível lógico um.

BTFSC f,b. Testa o *bit* de f e salta se igual a zero: esta instrução testa o *bit* indicado por 'b', do registrador f, e, se este *bit* for zero, é saltada uma instrução.

BTFSS f,b. Testa o *bit* de f e salta se igual a um: ao contrário da instrução anterior, esta testa o *bit* indicado por 'b', do registro f, e, se este *bit* for 1, é saltada uma instrução.

2.4.3.3.4 INSTRUÇÕES ORIENTADAS A CONSTANTES E DE CONTROLE

ADDLW k. Soma W a uma constante k: esta instrução somará o conteúdo de W a uma constante (literal) imediata k.

ANDLW k. Operação AND entre W e uma constante k: a instrução executará a função lógica AND entre o conteúdo de W e a constante imediata k.

CALL k. Chama uma sub-rotina: a instrução CALL chama uma sub-rotina endereçada por k, e com isso o contador de programa PC é atualizado com o valor de k, salvando na pilha o endereço de retorno da sub-rotina. A pilha suporta no máximo oito chamadas aninhadas.

CLRWDT. Zera o *timer* do *WatchDog*: esta instrução irá zerar o conteúdo do temporizador *Watch Dog*, evitando um *reset* se este estiver habilitado.

GOTO k. Faz desvio para a *Label* k: esta instrução faz com que o programa desvie para um novo endereço indicado por k, e com isso o valor do PC é atualizado com o valor de k.

IORLW k. Operação OR entre W e k: a instrução realizará a função lógica OR entre o conteúdo de W e um valor imediato k.

MOVLW k. Faz W = k: esta instrução faz o registrador W assumir o valor imediato k. Esta instrução é muito utilizada, principalmente quando se quer alterar o conteúdo de algum registrador, ou seja, em toda vez que se queira inserir um valor imediato em um registrador que não seja W, esta instrução é necessária em conjunto com MOVWF.

RETFIE. Retorna da interrupção: toda vez que o PIC atender uma rotina de interrupção, ao final da mesma deverá ocorrer esta instrução.

RETLW k. Retorna com o valor de k: a execução desta instrução faz o conteúdo de W retornar de uma sub-rotina com o valor imediato indicado por k. Esta instrução é muito usada para teste, por exemplo, de códigos.

RETURN. Retorna da sub-rotina: esta instrução deve estar sempre no fim de uma sub-rotina chamada pela instrução CALL, exceto quando se utiliza uma das duas instruções citadas anteriormente.

SLEEP. Entra no modo SLEEP: a instrução faz com que o PIC entre em baixo consumo, o oscilador para, o *Watch Dog* e o *Prescaler* são zerados.

SUBLW k. Subtrai k de W: esta instrução subtrai do conteúdo do registrador W o valor imediato indicado por k (W = W - k).

XORLW k. OR exclusivo entre W e k: realiza a função lógica OU EXCLUSIVO entre o conteúdo de W e um valor imediato indicado por k (W = W $\oplus$ k).

3 METODOLOGIA

3.1 ESPECIFICAÇÃO DA LINGUAGEM

Para o desenvolvimento do produto final e a análise de suas características e consequências de uso, a primeira etapa é a especificação da linguagem a ser implementada, definindo-se sua gramática em todos os níveis de compilação. Em nível léxico a linguagem tem por princípio o uso de palavras em português. Em nível sintático e semântico, a mesma deve ser simples, de fácil aprendizado, e adaptada à aplicação que terá em termos do uso em *hardware* embarcado. Para isso, foi feito um estudo comparativo com outras linguagens que têm a mesma característica de aplicação.

Para facilitar a construção do compilador, a gramática já foi feita de modo que não apresentasse recursão à esquerda e fosse fatorada à esquerda. A gramática também é representada na forma de Backus-Naur (BNF) (AHO, *et al.*, 2007). A gramática do G-Portugol (ARAÚJO, 2009; SILVA, 2006) foi uma das referências utilizadas para a construção da gramática do compilador desenvolvido.

Além do G-Portugol, a primeira linguagem de programação escolhida para estudo comparativo foi a linguagem Pascal, por ter sido desenvolvida para ser a primeira linguagem de programação a ser aprendida e por ser fácil de ensinar e aprender (CARVALHO, 2006). A segunda foi a linguagem C para microcontroladores PIC, pois sua compilação é para o mesmo destino que a estabelecida por este trabalho de conclusão de curso. Com base nesta pesquisa, criou-se a gramática da linguagem, que pode ser encontrada anexa.

3.2 IMPLEMENTAÇÃO

A partir da especificação da linguagem, se inicia a implementação do compilador nos níveis de *front-end*, a saber, os de análise léxica, sintática e semântica (AHO, et al., 2007; LOUDEN, 2004; GRUNE, et al., 2002). A implementação do compilador foi feita utilizando a linguagem Java e o ambiente de desenvolvimento Netbeans.

A implementação do analisador léxico foi feita baseado nas palavras e demais expressões próprias permitidas pela linguagem. O código fonte é carregado, e a cada caractere é decidido se o mesmo faz parte de um símbolo léxico em análise, se é um novo

símbolo, ou se não condiz com a especificação formal da linguagem. A análise da entrada , por exemplo, trataria os espaços em branco como tais e levaria ao reconhecimento do número , seguido do reconhecimento do identificador , do operador binário +, e por último do identificador .

Por exemplo, na linguagem estabelecida, um identificador é definido pela expressão regular . Assim, para reconhecer um identificador, o compilador lê o primeiro caractere. Se for uma letra, continua, caso contrário o *token* não pode ser um identificador. Depois da primeira letra, o programa entra em um *loop*, do qual sai apenas quando encontrar um caractere que não seja nem letra, nem número. A palavra, no final, é comparada com as palavras-reservadas da linguagem, para se verificar se não se trata de uma.

Criou-se um tipo abstrato de dados (TAD) para definir um *token*. O TAD contém o tipo do *token*, definido por um *enum*, o conteúdo léxico, em uma variável do tipo *String* chamada "valor", e a linha onde o *token* se encontra no código-fonte. Um *enum*, abreviação para "enumeração", é um TAD que tem valores de seleção e teste atribuídos a rótulos definidos pelo programador (neste caso, do compilador). Para cada tipo de *token* considerado, definido no dicionário da linguagem, há um rótulo associado na enumeração. Assim, o analisador léxico retorna ao sistema uma lista de *tokens*, ignorando os comentários e outras características das quais a próxima fase de análise prescinde.

Para um *enum*, o código em Java é feito da seguinte maneira:

```
public enum TipoToken {
    //palavras reservadas
    PROGRAMA, VARIÁVEIS, FIMVARIÁVEIS, INTEIRO, REAL,
    CARACTERE, LÓGICO, MATRIZ, INTEIROS, REAIS,
    [...]
}
```

O TAD do *token* assume a seguinte forma:

```
public class Token {
    private String valor;
    private TipoToken tipo;
    private int linha;
    //métodos
[...]
}
```

O tipo de *parser* escolhido para implementação foi o analisador sintático descendente recursivo, por ser aquele cuja conversão a partir da gramática é mais simples, trazendo vantagens tanto na codificação, quanto na correção de possíveis erros e em atualizações, sendo o mais adequado para o desenvolvimento manual de analisadores sintáticos (LOUDEN, 2004). Neste método, cada símbolo não terminal da gramática corresponde a uma sub-rotina e, para cada símbolo terminal, ocorre a chamada de uma sub-rotina que verifica se o símbolo esperado é igual ao símbolo encontrado. Desta forma, para cada produção são feitas subsequentes chamadas de métodos, ora para avaliar se o *token* atual corresponde a determinado símbolo terminal daquela produção, ora chamando outros métodos correspondentes aos símbolos não terminais presentes na produção.

Para exemplificar, segue o algoritmo da sub-rotina que reconhecerá a produção “ ”, usando uma função *match()* que recebe um tipo de *token* como parâmetro, e verificando se o *token* atual corresponde ao que é esperado sintaticamente, gerando um erro em caso contrário.

```
função principal()
{ //função correspondente ao símbolo não terminal
“principal”
     match(“início”);    //avalia se o token atual
corresponde à palavra reservada “início”
     comandos();         //chamada da função
correspondente ao símbolo não terminal “comandos”
     match(“fim);        // avalia se o token atual
corresponde à palavra reservada “fim”
}
```

Este método requer que a gramática seja de tipo LL(1), ou seja, que apenas um *token* posterior seja utilizado para tomar decisões na análise, o que condiz com a utilizada neste trabalho. Sempre que houver mais de uma produção a partir de um símbolo não terminal, em sua sub-rotina correspondente haverá, antes da chamada das sub-rotinas subsequentes, uma verificação do tipo do próximo *token*, para assim se determinar qual caminho seguir.

Na execução da análise sintática já são feitos passos de operações subsequentes: a análise semântica e a geração de código intermediário. Como, na especificação implementada, todas as declarações de variáveis e procedimentos aparecem antes do bloco principal do programa, a tabela de símbolos é construída no início e já utilizada para a análise semântica durante uma única execução da análise sintática. O código intermediário é gerado dentro das funções correspondentes a cada não terminal, convertendo cada operação em portugol para P-Código.

Para desenvolver um compilador, é necessário ter uma compreensão abrangente das características e do funcionamento da linguagem-alvo, que no caso deste projeto é o PIC *assembly* (Microchip Technology Inc., 2001). A partir desse estudo se pode definir a tradução de cada instrução de código intermediário, e implementar a otimização de código e a geração de código final do compilador (AHO, *et al.*, 2007; LOUDEN, 2004; GRUNE, *et al.*, 2002). O código final tem como alvo o microcontrolador PIC16F628A, mas pode ser facilmente convertido para um microcontrolador PIC diferente. Devido a sua complexidade, a etapa de otimização de código ainda não foi realizada, indo-se do código intermediário diretamente para o código final. Assim, um código final específico será eficaz, porém dificilmente será eficiente como poderia.

A tradução final de algumas funcionalidades da linguagem também não foi implementada, devido ao tempo disponível e às dificuldades enfrentadas. Apesar de reconhecida pelo analisador sintático, a estrutura de dados de matrizes não foi implementada na geração de código, pois, apesar de não ser muito complexa, não foi considerada uma funcionalidade essencial por ora, sendo deixada, portanto, para possíveis trabalhos futuros. Ocorre também que nesta implementação as operações lógicas e aritméticas funcionam apenas com números inteiros. Como não há instruções do microcontrolador para operações de multiplicação e divisão, estas foram implementadas como sub-rotinas simples, que realizam somas e subtrações sucessivas, respectivamente, e

funcionam apenas para números naturais. Outra funcionalidade descartada momentaneamente foi o atraso, devido principalmente à complexidade da implementação.

As diretivas em *assembly* para configurações iniciais do PIC, no estado em que pôde ser deixado o projeto, não são alteradas pelo usuário, e as portas do microcontrolador são tratadas pelo compilador como identificadores. A chamada de procedimento é feita com a instrução CALL, que aceita apenas oito chamadas aninhadas, o que também não é tratado pelo compilador.

Como o P-Código abstrai as variáveis auxiliares com a pilha, para desenvolver o gerador de código final é necessário, a partir do código intermediário, contar quantas posições da pilha foram utilizadas em cada caso, para assim estabelecer as variáveis auxiliares. Uma instrução em P-Código de carregar um valor na pilha corresponde a atribuir em *assembly* aquele valor à variável auxiliar correspondente àquela posição da pilha. Uma operação lógica e aritmética em P-Código corresponde em *assembly* a realizar a operação entre as duas últimas variáveis auxiliares e guardar o resultado na penúltima.

Durante boa parte da execução do projeto, para garantir em cada etapa as características e o funcionamento adequados do que foi desenvolvido, testes foram realizados, tanto manualmente quanto automaticamente.

Para realizar os testes finais, o ideal seria utilizar o *hardware* eleito, associado a *kits* de desenvolvimento, inserindo o produto em algum ambiente educacional para verificar a qualidade de sua aplicação. Entretanto, os testes foram realizados apenas em um ambiente de simulação de circuitos, o Proteus, comparando-se o código final gerado com outros códigos em *assembly* com mesma funcionalidade, porém criados o mais otimizado possível.

4 RESULTADOS

Para testar as funcionalidades da linguagem implementada, o primeiro passo foi buscar exemplos de códigos escritos para PIC, em linguagem C escrita para o compilador CCS, de típica aplicação em projetos de sistemas embarcados. Esses códigos foram manualmente traduzidos para a especificação do portugol e testados no analisador desenvolvido. A partir do código escrito em portugol, o P-Código e o *assembly* correspondentes foram gerados, e este último foi comparado com uma versão adotada como ideal, ou seja, que foi programada diretamente em *assembly* visando à máxima simplicidade. Alguns trechos de código em *assembly* (a parte de configuração inicial) são iguais em todos os casos. Esses trechos serão omitidos a partir da apresentação do segundo exemplo.

Muitos exemplos de código utilizam atraso e alguns, matrizes. Como essas funcionalidades não foram implementadas, apesar dos códigos em portugol passarem pelo analisador sintático, esses exemplos não serão mostrados a seguir, pois não há uma geração de código final para estes casos.

No primeiro exemplo, o objetivo é simplesmente acender um LED. Para isso é necessária apenas uma única saída do microcontrolador, que deverá estar ligada através de um resistor ao LED que será ligado. A Figura 4 ilustra esse circuito.

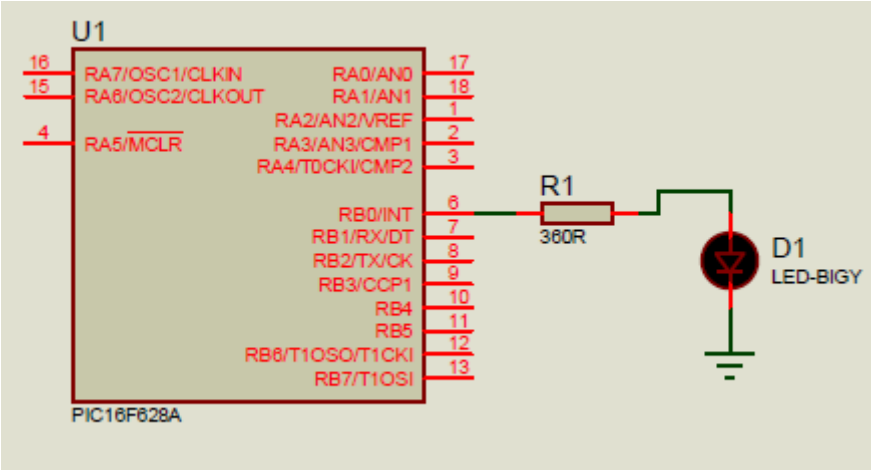

Figura 4: Primeiro circuito
Fonte: Autor Desconhecido

Código em C:

```
#include <16f628a.h>
#fuses INTRC,NOWDT,NOPROTECT,NOMCLR,NOLVP
#use delay (clock=4000000)
void main()
{
  output_high(pin_b0);
}
```

Tradução para o portugol:

```
programa AcendeLed;
variáveis
fim-variáveis
início
  ligar(pin_b0);
fim
```

Código intermediário:

```
hig pin_b0
```

Código em *assembly* gerado:

```
#INCLUDE <P16F628A.INC>
__CONFIG _BODEN_ON & _CP_OFF & _PWRTE_ON & _WDT_OFF &
_LVP_OFF & _MCLRE_ON & _XT_OSC
#DEFINE      BANK0       BCF     STATUS,RP0
#DEFINE      BANK1       BSF     STATUS,RP0
CBLOCK 0X20
VARMUL
VARDIV
ENDC
#DEFINE  pin_a0   PORTA, 0
#DEFINE  pin_a1   PORTA, 1
#DEFINE  pin_a2   PORTA, 2
#DEFINE  pin_b0   PORTB, 0
#DEFINE  pin_b1   PORTB, 1
#DEFINE  pin_b2   PORTB, 2
ORG  0X00
GOTO INICIO
ORG  0X04
RETFIE
INICIO
CLRF  PORTA
CLRF  PORTB
BANK1
MOVLW  B'11111111'
MOVWF  TRISA
MOVLW  B'00000000'
MOVWF  TRISB
MOVLW  B'10000000'
MOVWF  OPTION_REG
MOVLW  B'00000000'
MOVWF  INTCON
BANK0
MOVLW  B'00000111'
MOVWF  CMCON
```

```
PRINCIPAL
BSF pin_b0
END
```

Neste primeiro exemplo, o código ideal em *assembly*, desconsiderando-se as diretivas iniciais, é igual ao código gerado e por isso o mesmo não é reproduzido aqui.

O exemplo a seguir tem a função de ligar um LED com a ativação de uma chave. O que o diferencia do exemplo anterior é que, além da saída referente ao LED, tem também uma entrada, na qual está ligada a chave.

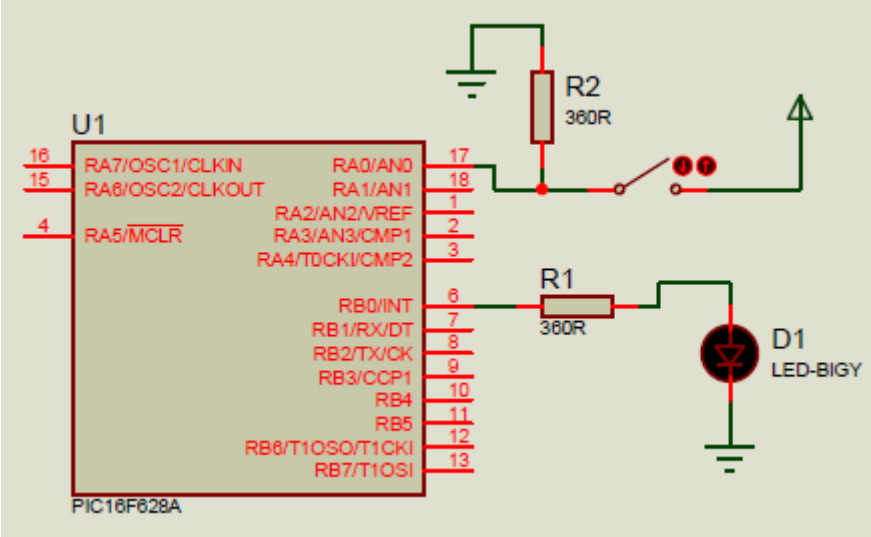

Figura 5: Segundo circuito
Fonte: Autor Desconhecido

Código em C:

```
#include <16f628a.h>
#fuses INTRC,NOWDT,NOPROTECT,NOMCLR,NOLVP
#use delay (clock=4000000)
void main()
{
  while(1)
  {
    if((input(pin_a0))==1)
      output_high(pin_b0);
    else
      output_low(pin_b0);
  }
}
```

Tradução para o portugol:

```
programa AcendeLed2;
variáveis
  entrada : inteiro;
fim-variáveis
início
  enquanto 1 faça
    ler(pin_a0, entrada);
    se entrada = 1 então
      ligar(pin_b0);
    senão
      desligar(pin_b0);
    fim-se;
  fim-enquanto;
fim
```

Código intermediário:

```
INTEIRO entrada
lab enquanto0
ldc 1
fjp enquanto1
read pin_a0
sto entrada
lod entrada
ldc 1
equ
fjp se0
hig pin_b0
ujp se1
lab se0
clr pin_b0
lab se1
ujp enquanto0
lab enquanto1
```

Código em *assembly* gerado:

```
[...]
CBLOCK 0X20
entrada
AUX1
AUX2
VARMUL
VARDIV
ENDC
[...]
enquanto0
MOVLW D'1'
MOVWF AUX1
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO enquanto1
```

```
MOVLW D'0'
BTFSC pin_a0
MOVLW D'1'
MOVWF AUX1
MOVF AUX1 ,0
MOVWF entrada
MOVF entrada ,0
MOVWF AUX1
MOVLW D'1'
MOVWF AUX2
MOVF AUX2 ,0
SUBWF AUX1 ,0
BTFSS STATUS, Z
GOTO EQU0
MOVLW D'1'
MOVWF AUX1
GOTO EQU1
EQU0
MOVLW D'0'
MOVWF AUX1
EQU1
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO se0
BSF pin_b0
GOTO se1
se0
BCF pin_b0
se1
GOTO enquanto0
enquanto1
END
```

Código ideal em *assembly*:

```
[...]
MAIN
BTFSC   BOTAO
GOTO    BOTAO_ON
GOTO    BOTAO_OFF
BOTAO_ON
BSF     LED
GOTO    MAIN
BOTAO_OFF
BCF     LED
GOTO    MAIN
END
```

A diferença já é notável, o que será comentado mais tarde.

A seguir, um exemplo de acionamento de um LED por meio de um botão. O LED acenderá quando o botão for pressionado e solto em seguida, e apagará ao se repetir o procedimento.

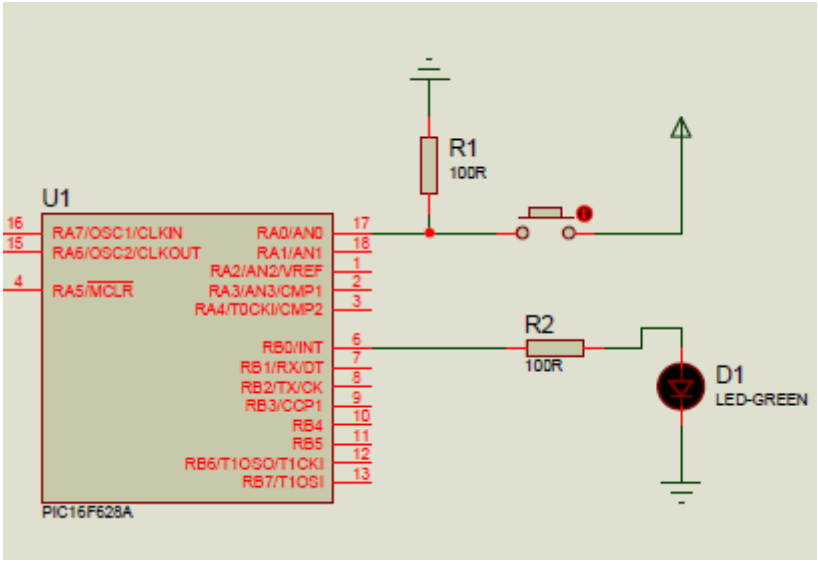

Figura 6: Terceiro circuito
Fonte: Autor Desconhecido

Código em C:

```
#include <16f628a.h>
#fuses INTRC,NOWDT,NOPROTECT,NOMCLR,NOLVP
#use delay (clock=4000000)
void main()
{
  int chave=0;
  while(true)
  {
    if(input(pin_a0)==1 && chave ==0)
    {
      while(chave==0)
      {
        if(input(pin_a0)==0)
        {
          output_high(pin_b0);
          chave=1;
        }
      }
    }
    if(input(pin_a0)== 1 && chave ==1)
    {
      while(chave ==1)
      {
        if(input(pin_a0)== 0)
        {
          output_low(pin_b0);
          chave==0;
        }
      }
    }
  }
}
```

Tradução para o portugol:

```
programa AcendeLed3;
variáveis
  chave, entrada: inteiro;
fim-variáveis
início
  chave := 0;
  enquanto 1 faça
    ler(pin_a0, entrada);
    se (entrada = 1) e (chave = 0) então
      enquanto chave = 0 faça
        ler(pin_a0, entrada);
        se entrada = 0 então
          ligar(pin_b0);
          chave := 1;
        fim-se;
      fim-enquanto;
    fim-se;
    se (entrada = 1) e (chave = 1) então
      enquanto chave = 1 faça
        ler(pin_a0, entrada);
        se entrada = 0 então
          desligar(pin_b0);
          chave := 0;
        fim-se;
      fim-enquanto;
    fim-se;
  fim-enquanto;
fim
```

Código intermediário:

```
INTEIRO chave
INTEIRO entrada
ldc 0
sto chave
lab enquanto0
ldc 1
fjp enquanto1
read pin_a0
sto entrada
lod entrada
ldc 1
equ
lod chave
ldc 0
equ
and
fjp se0
lab enquanto2
lod chave
ldc 0
equ
fjp enquanto3
read pin_a0
sto entrada
lod entrada
ldc 0
equ
fjp se2
hig pin_b0
ldc 1
sto chave
lab se2
ujp enquanto2
lab enquanto3
lab se0
lod entrada
```

```
ldc 1
equ
lod chave
ldc 1
equ
and
fjp se6
lab enquanto4
lod chave
ldc 1
equ
fjp enquanto5
read pin_a0
sto entrada
lod entrada
ldc 0
equ
fjp se8
clr pin_b0
ldc 0
sto chave
lab se8
ujp enquanto4
lab enquanto5
lab se6
ujp enquanto0
lab enquanto1
```

Código em *assembly* gerado:

```
[...]
CBLOCK 0X20
chave
entrada
AUX1
AUX2
AUX3
```

```
VARMUL
VARDIV
ENDC
[...]
MOVLW D'0'
MOVWF AUX1
MOVF AUX1 ,0
MOVWF chave
enquanto0
MOVLW D'1'
MOVWF AUX1
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO enquanto1
MOVLW D'0'
BTFSC pin_a0
MOVLW D'1'
MOVWF AUX1
MOVF AUX1 ,0
MOVWF entrada
MOVF entrada ,0
MOVWF AUX1
MOVLW D'1'
MOVWF AUX2
MOVF AUX2 ,0
SUBWF AUX1 ,0
BTFSS STATUS, Z
GOTO EQU0
MOVLW D'1'
MOVWF AUX1
GOTO EQU1
EQU0
MOVLW D'0'
MOVWF AUX1
EQU1
MOVF chave ,0
MOVWF AUX2
MOVLW D'0'
```

```
MOVWF AUX3
MOVF AUX3 ,0
SUBWF AUX2 ,0
BTFSS STATUS, Z
GOTO EQU2
MOVLW D'1'
MOVWF AUX2
GOTO EQU3
EQU2
MOVLW D'0'
MOVWF AUX2
EQU3
MOVF AUX2 ,0
ANDWF AUX1 ,1
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO se0
enquanto2
MOVF chave ,0
MOVWF AUX1
MOVLW D'0'
MOVWF AUX2
MOVF AUX2 ,0
SUBWF AUX1 ,0
BTFSS STATUS, Z
GOTO EQU4
MOVLW D'1'
MOVWF AUX1
GOTO EQU5
EQU4
MOVLW D'0'
MOVWF AUX1
EQU5
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO enquanto3
MOVLW D'0'
BTFSC pin_a0
```

```
MOVLW D'1'
MOVWF AUX1
MOVF AUX1 ,0
MOVWF entrada
MOVF entrada ,0
MOVWF AUX1
MOVLW D'0'
MOVWF AUX2
MOVF AUX2 ,0
SUBWF AUX1 ,0
BTFSS STATUS, Z
GOTO EQU6
MOVLW D'1'
MOVWF AUX1
GOTO EQU7
EQU6
MOVLW D'0'
MOVWF AUX1
EQU7
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO se2
BSF pin_b0
MOVLW D'1'
MOVWF AUX1
MOVF AUX1 ,0
MOVWF chave
se2
GOTO enquanto2
enquanto3
se0
MOVF entrada ,0
MOVWF AUX1
MOVLW D'1'
MOVWF AUX2
MOVF AUX2 ,0
SUBWF AUX1 ,0
BTFSS STATUS, Z
```

```
GOTO EQU8
MOVLW D'1'
MOVWF AUX1
GOTO EQU9
EQU8
MOVLW D'0'
MOVWF AUX1
EQU9
MOVF chave ,0
MOVWF AUX2
MOVLW D'1'
MOVWF AUX3
MOVF AUX3 ,0
SUBWF AUX2 ,0
BTFSS STATUS, Z
GOTO EQU10
MOVLW D'1'
MOVWF AUX2
GOTO EQU11
EQU10
MOVLW D'0'
MOVWF AUX2
EQU11
MOVF AUX2 ,0
ANDWF AUX1 ,1
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO se6
enquanto4
MOVF chave ,0
MOVWF AUX1
MOVLW D'1'
MOVWF AUX2
MOVF AUX2 ,0
SUBWF AUX1 ,0
BTFSS STATUS, Z
GOTO EQU12
MOVLW D'1'
```

```
MOVWF AUX1
GOTO EQU13
EQU12
MOVLW D'0'
MOVWF AUX1
EQU13
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO enquanto5
MOVLW D'0'
BTFSC pin_a0
MOVLW D'1'
MOVWF AUX1
MOVF AUX1 ,0
MOVWF entrada
MOVF entrada ,0
MOVWF AUX1
MOVLW D'0'
MOVWF AUX2
MOVF AUX2 ,0
SUBWF AUX1 ,0
BTFSS STATUS, Z
GOTO EQU14
MOVLW D'1'
MOVWF AUX1
GOTO EQU15
EQU14
MOVLW D'0'
MOVWF AUX1
EQU15
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO se8
BCF pin_b0
MOVLW D'0'
MOVWF AUX1
MOVF AUX1 ,0
MOVWF chave
```

```
se8
GOTO enquanto4
enquanto5
se6
GOTO enquanto0
enquanto1
END
```

Código ideal em *assembly*:

```
[...]
#DEFINE        BOTAO      PORTA,0
#DEFINE        LED        PORTB,0
[...]
MAIN1
BCF LED
BTFSS BOTAO
GOTO MAIN1
GOTO MAIN2
MAIN2
BTFSC BOTAO
GOTO MAIN2
GOTO MAIN3
MAIN3
BSF LED
BTFSS BOTAO
GOTO MAIN3
GOTO MAIN4
MAIN4
BTFSC BOTAO
GOTO MAIN4
GOTO MAIN1
END
```

Esses códigos foram devidamente testados e aprovados pelo analisador sintático implementado. Para facilitar a execução dos testes, utilizamos o JUnit, biblioteca Java específica para realização de testes de unidade de *software*.

Como foi observado, o código final gerado é eficaz, no sentido de que o efeito esperado é produzido. Entretanto, apesar de o P-Código facilitar a geração de código e proporcionar uma considerável flexibilidade na escolha do *hardware*, aumentando a portabilidade da linguagem, o código final não é eficiente, no sentido de que é inferior, em termos de otimização, ao código ideal. Além disso, quanto maior a complexidade do que se quer implementar, maior a diferença entre o código ideal e o código gerado, podendo-se até mesmo comprometer a eficácia do último, caso o mesmo não caiba no microcontrolador.

5 APLICAÇÃO PRÁTICA

Objetivando o uso da ferramenta em robótica educacional, foi elaborado um exemplo prático conforme à seguinte diretriz: a implementação de um algoritmo para uso em um robô seguidor de linha. O robô hipotético, ilustrado na Figura 7, tem duas rodas principais, com um motor associado a cada uma delas, e uma roda boba central. Tem também dois sensores frontais que detectam a presença da linha.

Para seguir a linha, o algoritmo utilizado foi o seguinte: quando nenhum dos sensores detecta a linha, o robô deve andar para frente, ligando os dois motores; quando o sensor da direita detecta a linha, o robô deve virar à direita, ligando apenas o motor da esquerda; quando o sensor esquerdo detecta a linha, o robô deve virar à esquerda, ligando apenas o motor da direita.

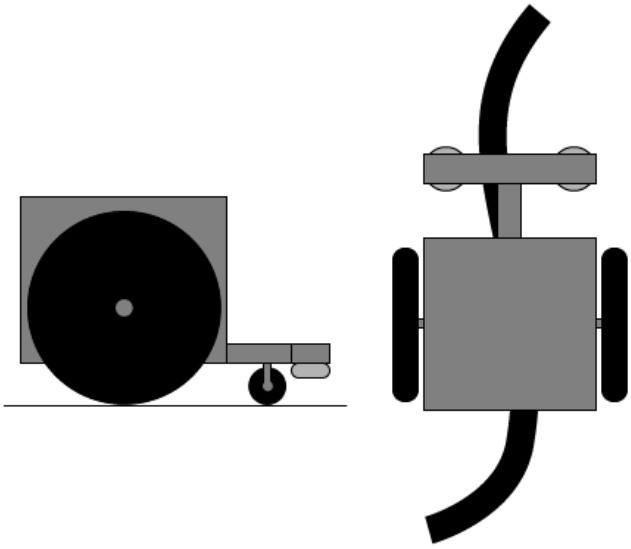

Figura 7: Ilustração do robô seguidor de linha
Fonte: Próprio autor

Utilizando a linguagem portugol especificada neste trabalho, e considerando as entradas pin_a0 e pin_a1 como as entradas dos sensores direito e esquerdo respectivamente, e as saídas pin_b0 e pin_b1 como as saídas para os motores direito e esquerdo respectivamente, o algoritmo pode ser implementado da seguinte forma:

```
programa RoboSeguidor;
variáveis
  entrada0,  entrada1: inteiro;
fim-variáveis
início
  enquanto 1 faça
    ler(pin_a0, entrada0);
    ler(pin_a1, entrada1);
    se entrada0 = 0 então
      ligar(pin_b0);
    senão
      desligar(pin_b0);
    fim-se;
    se entrada1 = 0 então
      ligar(pin_b1);
    senão
      desligar(pin_b1);
    fim-se;
  fim-enquanto;
fim
```

O código gerado:

```
#INCLUDE <P16F628A.INC>
__CONFIG _BODEN_ON & _CP_OFF & _PWRTE_ON & _WDT_OFF &
_LVP_OFF & _MCLRE_ON & _XT_OSC
#DEFINE      BANK0       BCF     STATUS,RP0
#DEFINE      BANK1       BSF     STATUS,RP0
CBLOCK 0X20
entrada0
entrada1
AUX1
AUX2
VARMUL
VARDIV
ENDC
#DEFINE  pin_a0   PORTA, 0
#DEFINE  pin_a1   PORTA, 1
#DEFINE  pin_a2   PORTA, 2
#DEFINE  pin_b0   PORTB, 0
#DEFINE  pin_b1   PORTB, 1
#DEFINE  pin_b2   PORTB, 2
ORG  0X00
GOTO INICIO
ORG  0X04
RETFIE
INICIO
CLRF  PORTA
CLRF  PORTB
BANK1
MOVLW  B'00000001'
MOVWF  TRISA
MOVLW  B'00000000'
MOVWF  TRISB
MOVLW  B'10000000'
MOVWF  OPTION_REG
MOVLW  B'00000000'
MOVWF  INTCON
BANK0
```

```
MOVLW  B'00000111'
MOVWF  CMCON
PRINCIPAL

enquanto0
MOVLW D'1'
MOVWF AUX1
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO enquanto1
MOVLW D'0'
BTFSC pin_a0
MOVLW D'1'
MOVWF AUX1
MOVF AUX1 ,0
MOVWF entrada0
MOVLW D'0'
BTFSC pin_a1
MOVLW D'1'
MOVWF AUX1
MOVF AUX1 ,0
MOVWF entrada1
MOVF entrada0 ,0
MOVWF AUX1
MOVLW D'0'
MOVWF AUX2
MOVF AUX2 ,0
SUBWF AUX1 ,0
BTFSS STATUS, Z
GOTO EQU0
MOVLW D'1'
MOVWF AUX1
GOTO EQU1
EQU0
MOVLW D'0'
MOVWF AUX1
EQU1
MOVF AUX1 ,0
```

```
BTFSC STATUS, Z
GOTO se0
BSF pin_b0
GOTO se1
se0
BCF pin_b0
se1
MOVF entrada1 ,0
MOVWF AUX1
MOVLW D'0'
MOVWF AUX2
MOVF AUX2 ,0
SUBWF AUX1 ,0
BTFSS STATUS, Z
GOTO EQU2
MOVLW D'1'
MOVWF AUX1
GOTO EQU3
EQU2
MOVLW D'0'
MOVWF AUX1
EQU3
MOVF AUX1 ,0
BTFSC STATUS, Z
GOTO se3
BSF pin_b1
GOTO se4
se3
BCF pin_b1
se4
GOTO enquanto0
enquanto1
END
```

Este código foi testado em ambiente de simulação com dispositivos análogos aos do exemplo: LEDs representando as saídas para os motores, e chaves representando as entradas do sensor, conforme à Figura 8. O comportamento foi de acordo com o esperado.

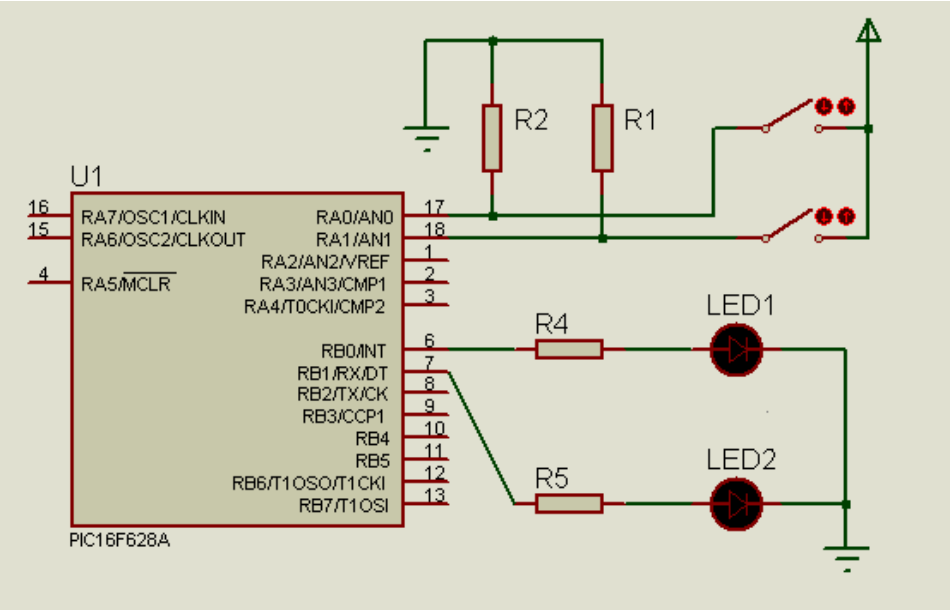

Figura 8: Circuito para teste do robô
Fonte: Próprio autor

6 CONCLUSÃO

Este trabalho teve como objetivos a especificação formal da gramática de um portugol particular, tendo como base outras linguagens com mesmo propósito, a implementação de tal linguagem, voltando-a para microcontroladores PIC e coletando resultados de testes em *hardware* embarcado, e a especificação de estudos de caso futuros com robótica educacional.

Devido à complexidade e à abrangência do projeto, nem todos os objetivos foram cumpridos. A gramática do portugol foi feita com base nas linguagens Pascal e G-Portugol, diferindo do G-Portugol principalmente no nível semântico, devido à aplicação. A linguagem final obedeceu à primeira e à segunda diretrizes levantadas por (ARAÚJO, 2009). O compilador foi implementado gerando código intermediário em P-Código e código final em PIC *assembly*, sem otimização. As funcionalidades de matrizes e a função de atraso não foram incluídas na geração de código final, assim como operações para além de números inteiros. Os códigos em portugol foram comparados com os correspondentes em linguagem C, e os códigos em *assembly* gerado pelo compilador e ideal também foram comparados. Os testes por enquanto foram realizados apenas em ambiente de simulação, não são numerosos e ainda não foi feito o estudo de adaptação do projeto a ambientes educacionais.

Entretanto, para desenvolver um compilador é preciso realizar diversas etapas que estão conceitualmente inseridas em diferentes campos da Computação. Esses campos incluem arquitetura de computadores, linguagens formais e autômatos, algoritmos e linguagens de programação, entre outras. Neste projeto, o compilador teve como alvo um microcontrolador, visando ao uso educativo em robótica educacional. Assim sendo, é possível incluir nessa lista as disciplinas de sistemas embarcados, robótica e informática na educação. Portanto, apesar de não se ter alcançado todas as funcionalidades e a totalidade dos objetivos, foi cumprido o objetivo principal deste trabalho de conclusão de curso, isto é, em apenas um trabalho abranger vários campos de estudo e prática da Computação.

Como o ideal deste projeto tem uma abrangência grande, é possível ainda realizar muitos trabalhos relacionados: completar as funcionalidades da linguagem, implementar a otimização de código, considerar e realizar outras metodologias de teste, criar tutoriais ou manuais de experiência para uso pedagógico, testar a ferramenta em ambientes educacionais, possibilitar a configuração da ferramenta para outras plataformas de

hardware, entre outros. Cada uma destas atividades pode ser realizada separadamente como um trabalho específico para completar o projeto.

7 BIBLIOGRAFIA

AHO, Alfred V., LAM, Monica S. e SETHI, Ravi. 2007. *Compiladores: Princípios, Técnicas e Ferramentas.* São Paulo : Longman do Brasil, 2007. ISBN: 9788588639249.

ARAÚJO, Adorilson B. de. 2009. *Investigação e extensão de uma ferramenta para auxílio ao ensino de algoritmos em ambientes GNU/Linux – G-Portugol.* Natal : UERN, 2009.

Autor Desconhecido. *Scribd.* [Online] [Citado em: 21 de 11 de 2012.] http://pt.scribd.com/doc/104647006/Exemplos-Em-C-Para-Pic-16f628a-Www-clocksize-blogspot-com.

AZEVEDO, Samuel, AGLAÉ, Akynara e PITTA, Renata. 2010. Minicurso: Introdução a Robótica Educacional. [Online] 2010. [Citado em: 11 de 08 de 2012.] http://www.sbpcnet.org.br/livro/62ra/minicursos/MC%20Samuel%20Azevedo.pdf.

CARVALHO, Flávia Pereira de. 2006. *Apostila de Programação I - Linguagem Pascal.* Taquara : s.n., 2006.

FONTANIVE, Drayton R. 1999. *Protótipo de Editor Fluxogramático com Interface Visual para Geração de Código para o Microcontrolador PIC16C84 da Microchip Technology.* Blumenau : Universidade Regional De Blumenau, 1999.

GOMES, M. C. 2007. *Reciclagem Cibernética e Inclusão Digital: Uma Experiência em Informática na Educação. Reescrevendo a Educação.* Chapecó : Sinproeste, 2007.

GRUNE, Dick, et al. 2002. *Projeto Moderno de Compiladores: Implementação e Aplicações.* Rio de Janeiro : Campus, 2002. ISBN: 85-352-0876-3.

LOUDEN, Kenneth C. 2004. *Compiladores: Princípios e Práticas.* São Paulo : Thomson Pioneira, 2004. ISBN: 8522104220.

MENEZES, Nilo N. C. 2003. *COMPORT – Compilador Portugol.* Manaus : Universidade do Amazonas, 2003.

Microchip Technology Inc. 2001. PIC16F84A Data Sheet: 18-pin Enhanced FLASH/EEPROM 8-bit Microcontroller. [Online] 2001. [Citado em: 12 de 08 de 2012.] http://ww1.microchip.com/downloads/en/devicedoc/35007b.pdf.

RIBEIRO, Célia, COUTINHO, Clara e COSTA, Manuel F. 2007. Robôcarochinha: Um Estudo Sobre Robótica Educativa no Ensino Básico. *V Conferência Internacional de Tecnologias de Informação e Comunicação na Educação.* 2007.

SEBESTA, Robert W. 2011. *Conceitos de Linguagens de Programação.* São Paulo : Bookman, 2011.

SILVA, Thiago. 2006. G-Portugol: Manual da versão v1.0. [Online] 2006. [Citado em: 12 de Agosto de 2012.] http://gpt.berlios.de/manual.pdf.

APÊNDICE – GRAMÁTICA DO PORTUGOL CONSIDERADO