UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PROTÓTIPO DE EDITOR FLUXOGRAMÁTICO COM *INTERFACE* VISUAL PARA GERAÇÃO DE CÓDIGO PARA O MICROCONTROLADOR PIC16C84 DA *MICROCHIP TECHNOLOGY*

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

DRAYTON ROBERTO FONTANIVE

BLUMENAU, JUNHO/1999

1999/1-13

PROTÓTIPO DE EDITOR FLUXOGRAMÁTICO COM *INTERFACE* VISUAL PARA GERAÇÃO DE CÓDIGO PARA O MICROCONTROLADOR PIC16C84 DA *MICROCHIP TECHNOLOGY*

DRAYTON ROBERTO FONTANIVE

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Miguel Alexandre Wisintainer — Orientador

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Miguel Alexandre Wisintainer

_____

Prof. Antonio Carlos Tavares

_____

Prof. José Roque Voltolini da Silva

AGRADECIMENTOS

Nos meses que passei desenvolvendo este trabalho, tive a honra de ter como orientador o professor Miguel Alexandre Wisintainer, o qual se mostrou muito determinado na orientação, sendo professor e amigo nas horas certas.

Agradecimentos também ao professor José Roque V. da Silva que no seu papel de coordenador soube escutar e encaminhar a solução de problemas.

Por fim, mas não menos importante, agradeço o apoio recebido de minha família e de meus amigos pela compreensão, auxílio e estímulo no decorrer desse semestre.

SUMÁRIO

SUMÁRIO IV

RESUMO VI

ABSTRACT VII

LISTA DE FIGURAS VIII

LISTA DE TABELAS IX

LISTA DE QUADROS X

LISTA DE SIGLAS E ABREVIATURAS XI

1 INTRODUÇÃO 1

1.1 Motivação 2
1.2 Justificativas 3
1.3 Objetivos 3
1.4 Organização do trabalho 4

2 FLUXOGRAMAS 5

2.1 Diagrama de blocos ou fluxograma de programa 5
2.2 Princípio do Data-Mapping 6
2.3 Como codificar significados visuais 7
2.4 Componentes dos gráficos 7
2.5 Apresentação da informação no modo gráfico 8

3 MICROCONTROLADORES 11

3.1 Introdução 11
3.2 Arquitetura Harvard X Von Neumann 12
3.3 O que é o PIC? 14
3.4 Microcontrolador PIC16C84 16
3.4.1 Escrita e montagem de um programa em assemblY 17
3.4.2 Como é constituído um PIC, quais dispositivos contém e como interagir entre eles. 20
3.4.2.1 A área de programa (EEPROM) e o Registrador de arquivo (Register File) 20
3.4.2.2 A ALU e o registro W 24
3.4.2.3 O contador de programa e o Stack 26
3.4.2.4 Porta A e PORTA B 30
3.4.3 conjunto de instruções do PIC16C84 32
3.4.4 considerações sobre o MONTADOR do PIC16C84 35

4 TRABALHOS CORRELATOS 37

4.1 Onagro 37
4.1.1 AMBIENTE PROPOSTO 37
4.1.2 Opções de menu do onagro 37

4.1.3 Os Ícones de Operação 38
4.1.4 A Estrutura do Código e os Mecanismos de Tradução 39
4.1.5 Resultados 40

5 TTREE 42

5.1 Introdução 42
5.2 Propriedades e métodos mais utilizados x mais importantes 43

6 ESPECIFICAÇÃO, IMPLEMENTAÇÃO E TESTES DO PROTÓTIPO 46

6.1 Tela principal 47
6.1.1 Botões Padrões 47
6.1.2 Botões PROCESS e CONDITION 48
6.1.3 Botões GOTO e TO 49
6.1.4 Botão END 50
6.1.5 Botão DELETE 50
6.1.6 Área de definição do Zoom 50
6.1.7 Botão VARIABLES 50
6.1.8 Botão I/O 51
6.1.9 Botão ASM FILE 51
6.2 Tela Declaration of Variables 51
6.3 Tela: I/O – Input/Output 52
6.4 Tela New Process 53
6.5 Tela: Condition 55
6.6 Testes 57
6.6.1 Circuito básico para testes 57
6.6.2 Teste comparativo 58

7 CONCLUSÃO 62

ANEXO 1 - TESTES DE USUÁRIOS FINAIS 64

ANEXO 2 – PIP02 66

8 REFERÊNCIAS BIBLIOGRÁFICAS 67

RESUMO

Este trabalho implementa um editor de fluxogramas que, a partir de uma *interface* visual, gera um algoritmo voltado para o microcontrolador da família 16 da *Microchip Technology's*. Junto da descrição desse algoritmo é gerado o código *assembly* para o microcontrolador.

ABSTRACT

This work implements a flowchart editor that through of a visual interface, generates a algorithm concerning to microcontroller of family 16 of the Microchip Technology's. With a algorithm's description is create assembly code for the microcontroller.

LISTA DE FIGURAS

Figura 1 - Diagrama de blocos 11
Figura 2 - Seqüência de trabalho da CPU 12
Figura 3 - Arquitetura Harvard x Arquitetura Von Neumann 13
Figura 4 - PIC16C84 14
Figura 5 - Pinagem do PIC 16
Figura 6 - Fluxograma de operações e arquivos 19
Figura 7 - EEPROM e Register File 21
Figura 8 – Área de memória RAM 22
Figura 9 – ALU e Registro W 25
Figura 10 – Stack e PC 27
Figura 11 – Porta A e Porta B 31
Figura 12 – Onagro: Tela principal 38
Figura 13 – Onagro: Ícones 39
Figura 14 – Onagro x Avocet C 40
Figura 15 – Objeto Ttree 42
Figura 16 – Tela exemplo de métodos do objeto Ttree 45
Figura 17 – Especificação do protótipo 46
Figura 18 – Tela Principal 48
Figura 19 – Tela Declaration of Variables 52
Figura 20 – Tela Input/Output 53
Figura 21 – Tela New Process 54
Figura 22 – Tela Condition 56
Figura 23 Circuito básico para os testes 57
Figura 24 – Fluxograma teste retirado do livro [BEN96] 58
Figura 25 – Fluxograma teste criado no protótipo 60
Figura 26 – Tela MPASMWIN 61
Figura 27 – Programador utilizado com o PIP02 61

LISTA DE TABELAS

Tabela 1 – Instruções das operações de byte com registros 33
Tabela 2 – Instruções das operações de bit com registros 34
Tabela 3 – Instruções das operações com constantes e de controle 34
Tabela 4 – Métodos e propriedades do objeto Ttree 44
Tabela 5 – Código gerado pelo processo 55
Tabela 6 – Código gerado pela condição 56

LISTA DE QUADROS

Quadro 1 – Símbolos do diagrama de blocos 6
Quadro 2 – Exemplo do conjunto reduzido de instruções 13
Quadro 3 – Opcode em notação binária 17
Quadro 4 – Opcode representado em hexadecimal 18
Quadro 5 – Escrita nas portas referenciando o endereço 24
Quadro 6 – Escritas nas portas referenciando o nome simbólico 24
Quadro 7 – Utilização do acumulador (registro W) 26
Quadro 8 – Exemplo da instrução *GOTO* 26
Quadro 9 – Exemplo da instrução *CALL* 28
Quadro 10 – Exemplo da utilização de várias instruções *CALL* 29
Quadro 11 – Código para conexão de um LED a linha RB4 32
Quadro 12 – Código para leitura do estado da chave conectada a linha RB4 32
Quadro 13 – Significado das variáveis utilizadas nas tabelas 1, 2 e 3. 34
Quadro 14 – Formato do programa 35
Quadro 15 – Exemplo da utilização de métodos e propriedades do *Ttree* 44
Quadro 16 – Adicionando uma conexão com *GOTO* 49
Quadro 17 – Algoritmo recursivo 51
Quadro 18 – Comparação de códigos testados 59

LISTA DE SIGLAS E ABREVIATURAS

ALU	Arithmetic and Logic Unit
CPU	Central Processor Unit
EEPROM	Eletrical Erasable Programmable Read Only Memory
LED	Light Emitter Diode
LIFO	Last In Last Out
PC	Program Counter
PIC	Peripheral Interface Controller
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
E/S	Entrada e Saída

1 INTRODUÇÃO

Praticamente, todas as pessoas estão rodeadas de aparelhos eletrônicos que possuem dentro de si um microcontrolador, e nem mesmo temos consciência disto. Os videocassetes, telefones celulares, agendas eletrônicas, vários brinquedos, alarmes de automóvel, são apenas alguns dos exemplos mais comuns.

Os grandes projetos que antes necessitavam de circuitos digitais complexos, são agora fáceis de realizar, usando um pequeno mas potente microcontrolador PIC, fabricado pela Microchip Technology. Os circuitos lógicos que se baseiam na duração ou contagem de pulsos, normalmente ficam bastante complexos se neles forem usados integrados normais. Um só PIC pode fazer o mesmo, em menos espaço, com baixo custo e menor complexidade [PIC95].

Este trabalho diz respeito a um estudo e implementação de um protótipo de um editor que a partir de fluxogramas criados pelo usuário gera um código em *assembly* para o microcontrolador PIC16C84. Esse microcontrolador possui, dentre outras características que serão abordadas com mais detalhes no desenvolvimento do trabalho, um conjunto reduzido de instruções, o que o torna uma das melhores opções para o início do trabalho proposto aqui.

O *assembly* do microcontrolador PIC16C84 da Microchip Technology monta, a partir de instruções contidas em arquivo texto, um código hexadecimal que depois será passado para o PIC. O código mnemônico para o microcontrolador pode ser criado usando um simples editor de texto. Um exemplo muito simples de uma aplicação no microcontrolador seria o programa atribuir 0 para todas as oito linhas de entrada e saída de uma porta B do PIC16C84, e deste modo, setando todas elas como saída [BEN96]. Oito E/S são linhas que podem ser configuradas como entrada ou saída e são estudadas no item 3.4.2.4.

Para a implementação do editor será utilizado o componente *TeeTree*, que após ser instalado no ambiente de programação Delphi, este conterá uma nova paleta de componentes onde estará o objeto *Ttree* que consiste de formas (*shapes*) e conexões no estilo de orientação à árvores, enfim, permitindo “desenhar” a informação de forma hierárquica, muito parecida com a estrutura do fluxograma. Dentre outras vantagens desse componente podemos citar o seu tamanho reduzido e o baixo consumo de memória.

A partir do fluxograma será gerado um arquivo texto com mnemônicos, e posteriormente traduzi-los em uma série de números (código hexadecimal) reconhecíveis diretamente pelo PIC. Para isto será utilizado o MPASMWIN (montador),.

Para realizar os testes no PIC será utilizado um programador de baixo custo, o PIP02, e considerando-se que o PIC16C84 dispõe de uma memória do tipo EEPROM (*Electrical Erasable Programmable Read Only Memory*) para armazenar o programa, poderá ser regravada milhares de vezes.

1.1 MOTIVAÇÃO

Os microcontroladores são extremamente indicados em sistemas que necessitam de altas performances com custo e tamanho reduzido, como por exemplo as aplicações em balanças digitais, termômetros, controladores de acesso, alarmes, aparelhos médicos, controladores de videocassete, de televisão, de iluminação, dentre outros [BEN96].

O PIC16C84, o montador MPASMWIN e o baixo custo do seu programador torna o sistema de desenvolvimento viável para a criação de projetos baseados em microcontroladores. Além disso, tudo que for aprendido usando esse microcontrolador será diretamente aplicável a toda família dos microcontroladores PIC16/17, pois os *sets* de instruções são compatíveis [BEN96].

Não é fácil escrever código para o PIC16/17 à mão pois ele não é intuitivo. Muitas instruções requerem acender ou apagar um bit para especificar o destino do resultado da instrução executada [BEN96]. Isto é bom como uma experiência de aprendizado inicial, mas poderá vir a ser muito tedioso no futuro, e assim, o uso de ferramentas auxiliares é essencial, como o proposto neste trabalho.

Será utilizada uma representação gráfica na forma de fluxogramas para edição e posterior geração de código para a linguagem do microcontrolador, pois ela é mais legível que a representação textual, diminuindo a probabilidade de erros e tornando o trabalho mais agradável. Este editor proporcionará facilidades para automatizar a representação através de fluxogramas. Durante a edição, consistências serão realizadas (dinamicamente), caracterizando um montador que converterá o código simbólico em código *assembly*. A representação fluxogramática a ser definida possuirá características muito próximas da

linguagem do microcontrolador, mas ainda sendo uma linguagem simbólica.

1.2 JUSTIFICATIVAS

Para pessoas que estão começando a programar microcontroladores, o PIC16C84 torna-se a melhor opção pelo seu baixo custo de desenvolvimento, reprogramável várias vezes e quase todo seu aprendizado será reaplicável a outros componentes da família de microcontroladores PIC da Microchip Technology's [BEN96]. Segundo [SIL97], o estudo desse microcontrolador torna-se útil pelo fato do mesmo possuir todos os periféricos básicos dos demais membros da família PIC.

A utilização de gráficos para a representação da informação pode ser justificada pelos seguintes fatores:

a) aumentam a performance do trabalho;
b) tornam os documentos uma linguagem global;
c) "seduzem" leitores relutantes;
d) aumentam a credibilidade;
e) auxiliam o pensamento;
f) tornam a leitura mais eficiente;
g) podem explicar conceitos visuais;
h) são compactos;
i) fogem das limitações do texto linear;
j) são prontamente compreendidos;
k) são fáceis de relembrar;
l) permitem uma visão mais clara dos erros.

Enfim, o ponto determinante que faz com que todos os itens descritos acima tenham validade é o fato de a visão ser nosso sentido determinante.

1.3 OBJETIVOS

O objetivo principal do trabalho é especificar e implementar um protótipo de um editor gráfico que irá analisar informações que serão passadas por meio de uma *interface* visual (fluxogramas), e após, gerar o código em baixo nível (mnemônicos), o qual passará por uma

montagem final para realização dos testes.

1.4 ORGANIZAÇÃO DO TRABALHO

O presente trabalho está dividido em sete capítulos, sendo que o primeiro trata da introdução, englobando os objetivos, justificativas, motivação e estrutura do trabalho.

O segundo capítulo trata de fornecer algumas características de representação gráfica, justificando amplamente o método utilizado nesta implementação.

No terceiro capítulo são abordados os microcontroladores de uma forma geral, partindo para a família do PIC16 e aprofundando-se no PIC16C84, indispensável por ser base desse trabalho.

O quarto capítulo mostra um trabalho correlato, que contém idéias e finalidades semelhantes as propostas neste trabalho.

O quinto capítulo destina-se a fazer algumas considerações sobre o *Ttree*, componente utilizado para auxiliar na construção do editor.

O sexto capítulo concentra-se na descrição da especificação, implementação e testes do protótipo.

No sétimo capítulo apresenta-se a conclusão do trabalho.

2 FLUXOGRAMAS

Normalmente, o computador é comandado por programas, ou seja, uma série de instruções bem definidas que nos levam a um fim predeterminado.

Para fazer um programa deve-se escolher uma linguagem simbólica de programação que o computador entenda e também devem ser passadas as tarefas para que ele realize, dando-lhe condições para isto. Mas, antes de chegar a programação propriamente dita, deve-se seguir três etapas iniciais. Primeiramente, determinar qual é o problema que deverá ser resolvido, descrevendo-o minuciosamente (análise). Em seguida, estudar os meios para resolver este problema. Surge então o fluxograma de sistemas que indica os meios de entrada, isto é, como as informações fluem para dentro do sistema de processamento de dados. Determina também as formas de documentos apresentadas na saída deste sistema [BEN79].

Após estas fases concluídas, chega-se a definição lógica das etapas do programa, através do diagrama de blocos (também chamado de fluxograma de programa ou simplesmente fluxograma) [BEN79].

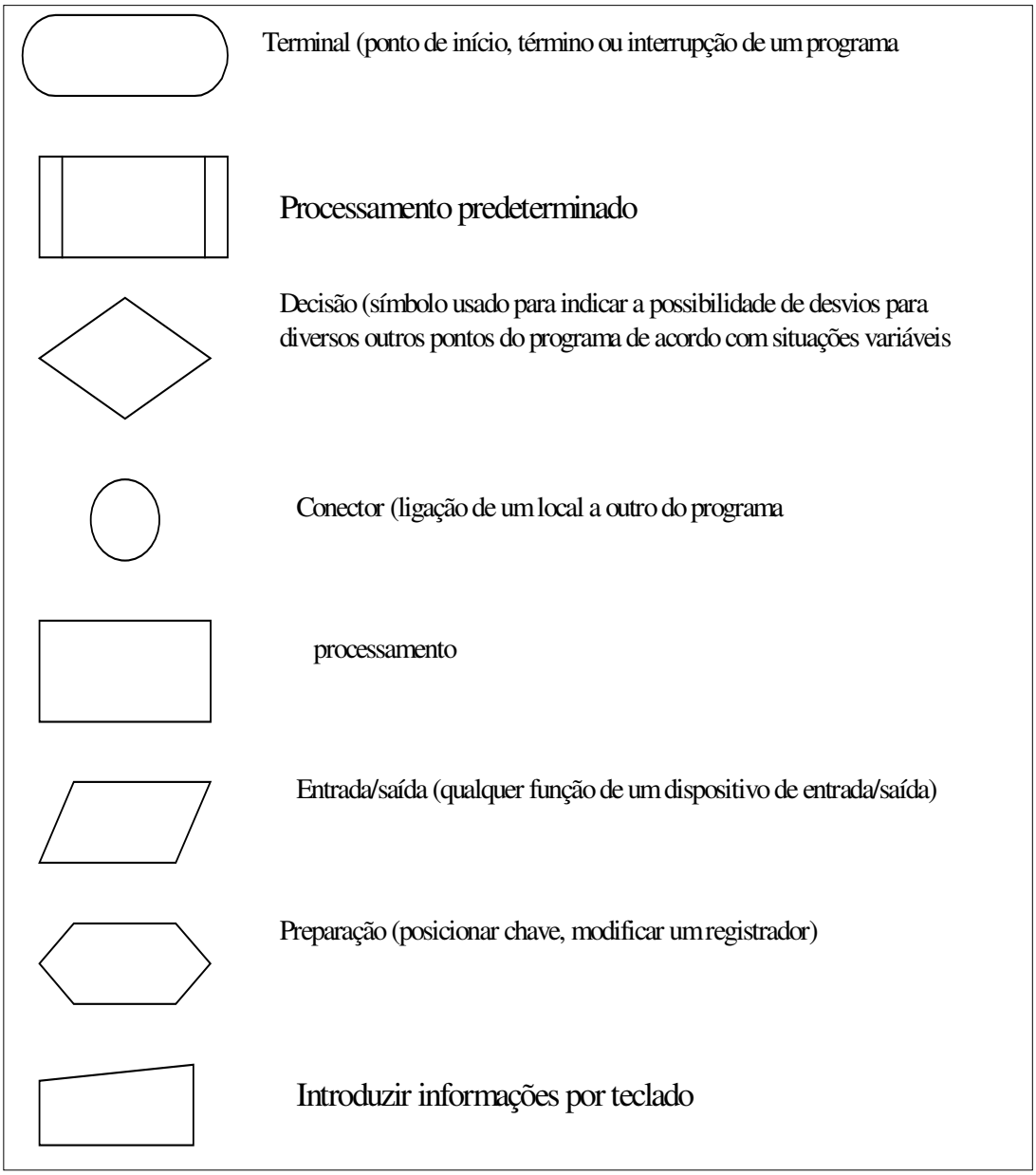
Tanto o fluxograma de sistemas como o diagrama de blocos são apresentados sob forma de conjuntos de símbolos específicos. Os símbolos do fluxograma podem ser vistos no quadro 1, o qual foi utilizado neste trabalho.

2.1 DIAGRAMA DE BLOCOS OU FLUXOGRAMA DE PROGRAMA

Ao receber instruções sobre o que se deve ser feito, os programadores tratam de resolvê-las também subdividindo em duas fases inter-relacionadas.

Em primeiro lugar, faz o diagrama de blocos logo após ter estudado o problema. Em segundo lugar codifica-se, isto é, transforma o diagrama de blocos em uma linguagem que o computador compreenda [BEN79].

O diagrama de blocos segue símbolos usados internacionalmente, como pode ser visto no quadro 1.



Quadro 1 – Símbolos do diagrama de blocos

2.2 PRINCÍPIO DO DATA-MAPPING

Segundo [HOR91], todos os gráficos técnicos são construídos sobre um princípio tão simples que todos provavelmente já sabem mas nunca pararam para examiná-lo. Quando isso ocorre, percebe-se que ele pode explicar uma enorme variedade de outros gráficos técnicos em uso e podem sugerir novas formas gráficas criativas. *Data-Mapping* é uma forma de

representar os dados e os fenômenos relativos ao mesmo por intermédio de figuras, linhas, coloridos, dentre outros, que devem ser previamente convencionados. O princípio do *Data-Mapping* é: "Use objetos gráficos para representar conceitos e características gráficas para representar dimensões correspondentes ou características dessas idéias" [HOR91].

2.3 COMO CODIFICAR SIGNIFICADOS VISUAIS

Associar símbolos a conceitos envolve um processo interativo [HOR91]:

a) escolha de um símbolo gráfico para um conceito;
b) escolha de uma característica gráfica (decide que característica gráfica irá representar a característica de um conceito);
c) escolha de uma função escalar (define um algoritmo, *procedure*, ou heurística para associar valores gráficos a valores de dados).

2.4 COMPONENTES DOS GRÁFICOS

Há muitos fatores que contribuem no significado dos gráficos: mensagem, redundância, decoração e interferências.

A mensagem é simplesmente o que se quer dizer. É o significado, a informação, o sinal que se está tentando transmitir. Redundância significa dizer a mesma coisa de mais de uma forma, tal como sinalizar uma ênfase por meio de uma forma triangular na cor amarela. Não é um desperdício, é um aumento de segurança, pois resiste a más interpretações, dentre outros problemas. A decoração atrai a atenção para o gráfico. Se a decoração não contribui no significado, pode ser útil na motivação do leitor para estudar o gráfico em detalhes, ou seja, não representa informação diretamente. A cor verde pode ser considerada uma redundância, enquanto o verde-limão, representa ainda uma decoração. Interferência é qualquer coisa que possa alterar o significado. A decoração em excesso pode se tornar uma interferência [HOR91].

Esses componentes estão altamente interrelacionados. Redundância é necessário para amenizar os efeitos das interferências, ou seja, uma interferência pequena requer uma redundância pequena. Deve-se controlar o uso de decoração, pois com o excesso dela a comunicação da informação pode deixar de ser efetiva, assim, a decoração a mais se torna

interferência [HOR91].

2.5 APRESENTAÇÃO DA INFORMAÇÃO NO MODO GRÁFICO

Segundo [HOR91], gráficos são freqüentemente difíceis de desenhar, criar, e caros de produzir, além de que sobreviveu-se com números e palavras "nuas" por milhares de anos. Então por que precisa-se de gráficos? A resposta é simples: pela facilidade que proporcionam para comunicação.

Gráficos auxiliam na performance do trabalho. Quando uma explanação é estudada vagarosamente, gráficos e cores parecem oferecer pouca vantagem. Entretanto, quando o leitor é impaciente ou quer tomar decisões rapidamente, exposições de figuras coloridas são mais efetivas. Para tarefas que envolvem muitos conceitos de números, os gráficos permitem os leitores processarem informações com precisão quase matemática e com velocidade.

Gráficos tornam documentos uma linguagem global, pois atualmente um produto pode ser desenvolvido em um país, manufaturado em outro, transportado por muitos outros, e utilizados em dúzias de outros países. A partir disso, observa-se que o produto é manipulado por pessoas de idiomas e dialetos bem diversificados, podendo causar vários erros de interpretação. Conclui-se então, que o custo potencial da tradução só é excedido pelo custo potencial da "confusão". Os gráficos podem reduzir estes custos.

Gráficos aumentam a credibilidade. Muitas pessoas acreditam que figuras refletem somente a verdade. A nossa fé na visão é tão grande que quando o que nós sentimos com as mãos discordam do que vemos, inconscientemente, as percepções do toque são alteradas para consistirem com a visão.

Gráficos ajudam a resolver problemas. Os problemas podem ser resolvidos em grande parte dependendo do caminho da representação do problema que faz sua solução transparente. Gráficos são memoráveis, manipulam facilmente símbolos que representam idéias. Também podem simplificar soluções de problemas gravando visualmente fatos, ou, caso contrário, deduzir ou buscar na memória a curto prazo. Pela informação bruta, gráficos e cores habilitam as pessoas manipular mais informações, processá-las mais eficientemente, e aplicar estratégias mais simples e eficientes de decisão e montagem.

Gráficos podem ajudar um leitor a aprender uma tarefa difícil e até mesmo entender um documento desestruturado. Podem também auxiliar os leitores a ver e compreender padrões complexos, guiando-os de forma eficiente, não tendo que ler o documento inteiro em todos os seus detalhes. Os gráficos proporcionam um mapa global do documento como também marcas para as informações críticas. O mapa deixa o leitor assimilar a organização do documento e planejar uma estratégia para achar a informação. As marcas identificam blocos de informações para os leitores a acharem mais rápido.

Gráficos podem explicar conceitos visuais. Alguns conceitos desprezam palavras. Outros podem ser expressados em palavras mas são mais eficientes em figuras. Ainda outros devem ser traduzidos antes para imagens visuais para depois serem entendidos. Imagens trabalham melhor que palavras por expressar de forma mais clara as sutilezas das dimensões, cores e relações textuais. Eles também trabalham bem por armazenar muita informação sobre todos os aspectos.

Gráficos são compactos, pois comparando-se com palavras, gráficos podem dizer mais em menos espaço. Quando gráficos servem como um meio de armazenamento para fatos gravados, podem ser bastante densos apesar de ainda serem legíveis.

Dado as altas expectativas dos leitores, os escritores têm que buscar modos para superar as limitações das páginas tradicionais e das telas de computador. Linguagens, homens e computadores, todos estão limitados por um linear, pela sintaxe palavra depois de palavra. A estrutura do idioma implica em certas suposições sobre a realidade. Orações, que são construídas de palavras isoladas, são unidirecionais. Esta estrutura do idioma pode dar a impressão de que o mundo é fragmentado ao invés de contínuo, como linear ao invés de complexo. Tal conceito restritivo também tende a filtrar outros modos de pensamento.

Gráficos corretamente projetados fazem com que seus pontos principais sejam identificados rapidamente, pois não têm que ser lidos, analisados e interpretados. Gráficos melhoram a velocidade e precisão com que informação é assimilada e processada. Através deles, comparações se tornam automáticas e os relacionamentos óbvios. São compreendidos mais rapidamente que palavras e relatam mais facilmente a realidade. Além disso, gráficos que reforçam o significado do texto aumentam a compreensão.

Gráficos são fáceis de relembrar. Testes de memorização mostraram que nós temos memória de reconhecimento para imagens gráficas quase ilimitada e que conceitos lembrados visualmente são recordados melhor que os codificados. Além disso, eles são reconhecidos quase que perfeitamente.

Um gráfico salienta os erros do contexto e facilita a dedução de conclusões corretas. Além disso, nós podemos examinar imagens visuais sugeridas por gráficos com mais confiança que formar imagens de palavras lidas. Com palavras, a imagem deve ser construída pelo leitor e nunca é tão vívido quanto a imagem explícita provida por um gráfico.

Os humanos confiam na visão sobre todos os outros sentidos. Com a audição aprende-se 11%, e em contrapartida, aprende-se 83% visualmente [HOR91].

3 MICROCONTROLADORES

3.1 INTRODUÇÃO

Basicamente, o microcontrolador é um componente que possui todos os periféricos dos microprocessadores comuns embutidos em uma só pastilha, facilitando assim o desenvolvimento de sistemas pequenos e de baixo custo, embora complexos e sofisticados [SIL97].

Costumam apresentar em uma única pastilha memórias de dados e programa, canal serial, temporizadores, *interfaces* para *displays*, memória EEPROM, e muito mais, dependendo do modelo [SIL97].

O diagrama de blocos simplificado da estrutura dos microcontroladores pode ser visto na figura 1.

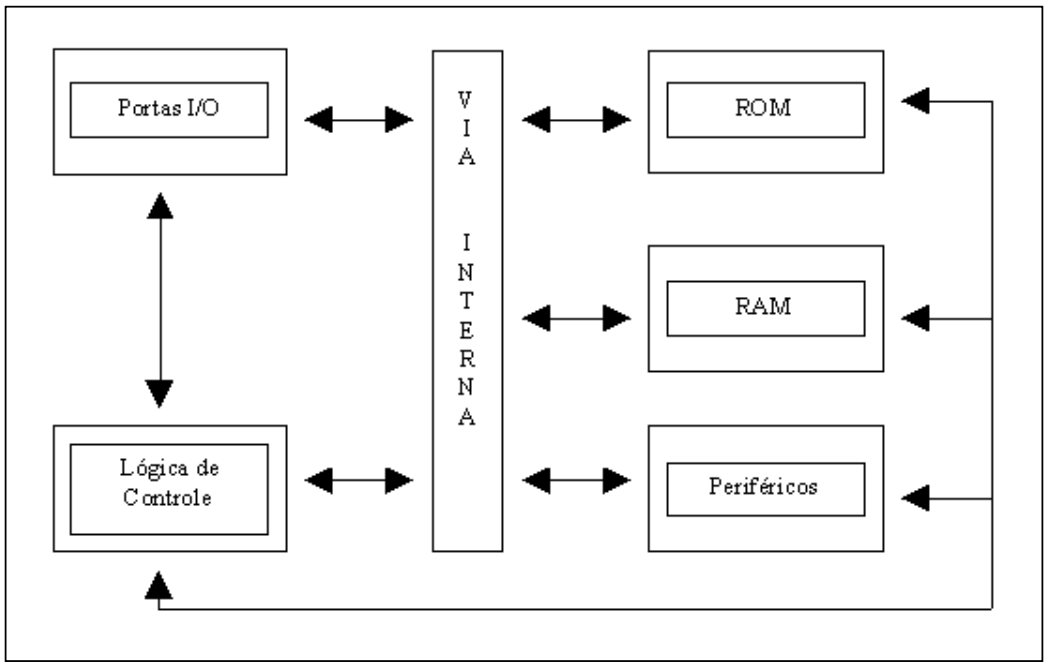


Figura 1 - Diagrama de blocos

3.2 ARQUITETURA HARVARD X VON NEUMANN

A maioria microprocessadores comuns e vários microcontroladores existentes no mercado tem sua estrutura interna de memória de dados e programa baseados na conhecida arquitetura de Von Neumann, que prevê uma única via (*bus*) de comunicação entre memórias e CPU do microcontrolador [SIL97].

Nesta estrutura todos os dados tratados pela CPU passam por uma via única, então enquanto a CPU está lendo um dado ou instrução de memória, as vias internas não podem ser usadas para outra finalidade [SIL97]. Basicamente pode-se dizer que a seqüência de trabalho está descrito na figura 2.

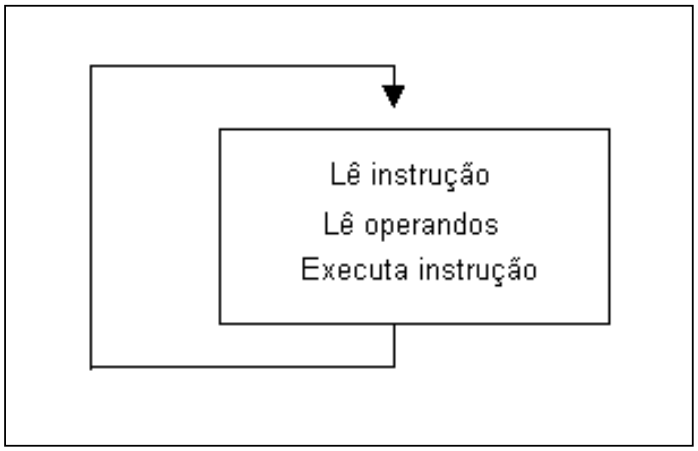


Figura 2 - Seqüência de trabalho da CPU

Os PIC's utilizam uma arquitetura diferente conhecida como *Harvard*, que prevê várias vias de comunicação entre a CPU e periféricos, permitindo a realização de várias operações simultaneamente, o que implica em aumento considerável na velocidade de execução e permite ainda que memória de dados e programas tenham tamanhos diferentes [SIL97].

Criou-se então uma terminologia chamada RISC (*Reduced Instruction Set Computer* – computador com conjunto de instruções reduzido) que faz com que existam poucas instruções (mais ou menos 35, dependendo do modelo) enquanto alguns microprocessadores tradicionais chegam a ter mais de 100 instruções [SIL97]. Este *set* reduzido de instruções facilita muito o

aprendizado. Um exemplo (somar 12 ao registro 24) pode ser visto no quadro 2.

A) Microprocessador comum: 3 bytes
Somar é a operação
12 é o operando
24 é o local
B) Microprocessador RISC: 1 palavra de 14 bits
Somar + 12 (operando) + 24 (local)
(tudo numa única palavra)

Quadro 2 – Exemplo do conjunto reduzido de instruções

Como a maioria das instruções dos microprocessadores comuns usam 2 bytes (existem instruções de 1 e 3 bytes também), vê-se que os códigos dos PIC's já tem basicamente a metade do tamanho [SIL97].

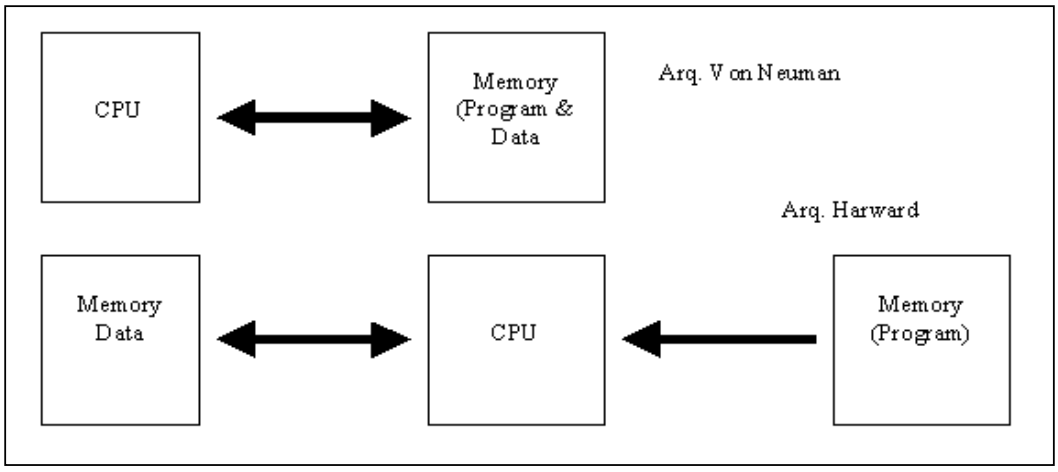


Figura 3 - Arquitetura Harvard x Arquitetura Von Neumann

Outra vantagem da arquitetura *Harvard* está no fato de que enquanto uma instrução está sendo executada e valores são lidos ou escritos na memória ou E/S pela via apropriada, outra instrução já está sendo carregada pela via da memória do programa, acarretando um aumento de velocidade [SIL97].

Apenas como comparação, um 8051 padrão, rodando a 12 MHz (sua velocidade máxima típica), executa a maioria das instruções em 1 us, enquanto para executar em 1 us o PIC precisa de apenas 4 MHz, ou seja, com 12 MHz o PIC será 3 vezes mais rápido. Nem todos os modelos de PIC chegam a 12 MHz. O PIC16C84 chega até 10 MHz [SIL97].

3.3 O QUE É O PIC?

O PIC (*Peripheral Interface Controller*) é um circuito integrado produzido pela Microchip Technology Inc., que pertence a categoria de microcontroladores, ou seja, um componente integrado que em um único dispositivo contém todos os circuitos necessários para realizar um completo sistema digital programável [GAL99].

É essencialmente um controlador de entrada e saída e é construído para ser muito rápido, pois é baseado na arquitetura *Harvard*. Além disso tem um conjunto de instruções reduzidas baseadas na família dos microcontroladores PIC16C5X [BEN96].

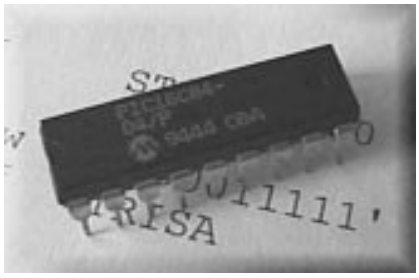


Figura 4 - PIC16C84

A figura 4 mostra um PIC (neste caso um PIC16C84), que segundo [SIL97] internamente dispõe de todos os dispositivos típicos de um sistema microprocessado, ou seja:

a) uma CPU (*Central Processor Unit* ou seja Unidade Central de Processamento) e sua finalidade é interpretar instruções de programas;

b) uma memória PROM (Programmable Read Only Memory ou Memória Programável Somente para Leitura) na qual irá memorizar de maneira permanente as instruções do programa;
c) uma memória RAM (*Random Access Memory* ou Memória de Acesso Aleatório) utilizada para memorizar as variáveis utilizadas pelo programa;
d) uma série de linhas de E/S para controlar dispositivos externos ou receber pulsos de sensores, chaves, etc;
e) uma série de dispositivos auxiliares ao funcionamento, ou seja, gerador de *clock*, *bus*, contador, etc;
f) registradores de funções especiais.

A presença de todos estes dispositivos em um espaço extremamente pequeno, permite ao projetista ampla gama de trabalho e enorme vantagem em usar um sistema microprocessado, onde em pouco tempo e com poucos componentes externos pode-se fazer o que seria oneroso fazer com circuitos tradicionais.

O PIC está disponível em uma ampla variedade de modelos para melhor adaptar-se as exigências de projetos específicos, diferenciando-se pelo número de linhas de E/S e pelo conteúdo do dispositivo. Inicia-se com modelo pequeno identificado pela sigla PIC12Cxx dotado de 8 pinos, até chegar a modelos maiores com sigla PIC17Cxx dotados de 40 pinos [GAL99].

A família PIC16C5X (família básica) oferece a melhor relação custo/benefício. São extremamente compactos com operação abaixo de 2.0 V, fazendo dela ideal para aplicações portáteis alimentados por bateria [BER96].

A família PIC16CXX (família intermediária) oferece o maior número de opções de periféricos, mantendo ainda uma boa relação custo/benefício [BER96].

A família PIC17CXX (família de alta performance) é a topo de linha (atualmente) e oferece os microcontroladores de 8 bits mais rápidos do mercado além de oferecer uma vasta gama de periféricos [BER96].

3.4 MICROCONTROLADOR PIC16C84

Este trabalho será baseado em um modelo intermediário, o PIC16C84. Este é dotado de 18 pinos, sendo 13 disponíveis para E/S, ou seja, para serem ligados ao circuito e de algumas características que o tornam um circuito que melhor atenderá as exigências do trabalho.

Em particular, o PIC16C84 dispõe de uma memória para armazenar o programa, do tipo EEPROM, que pode ser regravada milhares de vezes (aproximadamente um milhão) [GAL99]. A memória de programa tem 1Kbyte, e essa tecnologia (EEPROM) permite que esse chip seja apagado e regravado automaticamente pelos gravadores, dispensando as lâmpadas ultravioletas comuns nos desenvolvimentos com EPROM comum [SIL97].

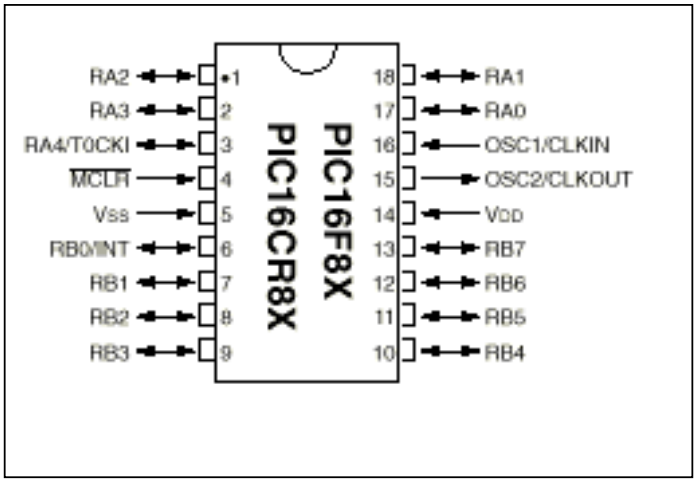


Figura 5 - Pinagem do PIC

Na figura 5 pode ser vista a reprodução da pinagem do PIC e nomenclatura de seus respectivos pinos.

Como é possível ver, o PIC16C84 é dotado de um total de 18 pinos dispostos em duas fileiras paralelas de 9 pinos cada uma (*dual in line*). Os pinos RA0..RA4 e RB0..RB7 representam as linhas de E/S disponíveis para as aplicações. Os pinos Vdd e Vss são os pinos de alimentação e os pinos MCLR\, OSC1 E OSC2 são reservados ao funcionamento do PIC (MCLR para o *reset* ; OSC1 OSC2 para o *clock*) [BEN96].

Os pinos Vdd (pino 14) e Vss (pino 5) servem para fornecer alimentação para o chip e são ligados respectivamente ao positivo e a massa. O pino MCLR (pino 4) serve para resetar o chip quando este estiver na condição lógica zero [BEN96].

Os pinos OSC1/CLKIN (pino 16) e OSC2/CLKOUT (pino 15) são conectados internamente ao circuito para gerar a freqüência de *clock* utilizada para temporizar todo o ciclo de funcionamento interno do chip [BEN96]. Desta freqüência depende a maior parte das operações internas e em particular a velocidade com que o PIC processa as instruções do programa. No caso do PIC16C84-04/P tal freqüência pode chegar a um máximo de 4Mhz da qual se obtém uma velocidade de execução das instruções perto de 1 milhão de instruções por segundo [GAL99].

3.4.1 ESCRITA E MONTAGEM DE UM PROGRAMA EM *ASSEMBLY*

Como em qualquer sistema microprocessado, no PIC também é necessário preparar um programa para que ele o execute.

Um programa é constituído por um conjunto de instruções em seqüência, onde cada uma identificará precisamente a função básica que o PIC irá executar. A instrução é representada por um código operativo (ou *opcode*, do inglês *operation code* ou abreviadamente *opcode*) podendo memorizar 14 bits em cada locação da memória EEPROM. Esta memória no PIC16C84 dispõe de 1024 locações e cada uma deverá conter uma só instrução. Um exemplo de *opcode* em notação binária está escrito no quadro 3 [GAL99].

00 0001 0000 0000B

Quadro 3 – Opcode em notação binária

É mais provável que um *opcode* venha representado na notação hexadecimal, como pode ser visto no quadro 4.

Esta última representa exatamente o mesmo valor, em um outro sistema de numeração. A letra H, escrita no final do valor 0100, indica o tipo de notação (Hexadecimal). O mesmo

valor pode ser representado em assembly com a notação 0x100 que é derivado da linguagem C ou H'0100'.

0100H

Quadro 4 – Opcode representado em hexadecimal

Este código, completamente sem sentido para os humanos, é o que o PIC está preparado para entender. Para facilitar a compreensão ao programador, recorre-se a um instrumento de convenção para tornar a instrução mais compreensível.

A primeira convenção é a que associa o *opcode* (um total de 35 para o PIC16C84) a uma sigla mnemônica, ou seja, uma inicial que seja fácil de recordar o significado da instrução. Voltando ao exemplo, o *opcode* 0100H corresponde a instrução mnemônica CLRW que é a forma abreviada da instrução CLEAR W REGISTER, ou seja, zere o registrador W. Outra convenção consiste na definição da variável, da constante, do *label* (rótulo) de referência ao endereço de memória. O propósito desta convenção é de facilitar a escrita de um programa para o PIC que é chamada linguagem *assembly*. Um programa escrito em linguagem *assembly* pode ser escrito em qualquer microcontrolador utilizando-se qualquer processador de texto que possa gerar arquivos ASCII (*Word, Notepad*, etc). Um arquivo de texto que contenha um programa em *assembly* é denominado de *source* ou código fonte [GAL99].

Uma vez preparado o código fonte, será necessário um programa para traduzir as instruções mnemônicas e todas as outras formas convencionais com que se escreve o código em uma série de números (o *opcode*) reconhecível diretamente pelo PIC. Este programa se chama montador [GAL99].

Na figura 6 está esquematizado o fluxograma de operações e arquivos que deverá ser realizado para passar um código *assembly* a um PIC a ser programado.

A primeira operação a ser efetuada é a escrita do código *assembly* e a sua gravação em um arquivo de texto com a extensão .ASM. O próximo passo é a montagem do código, ou seja, a transformação em *opcode* do código mnemônico ou instruções *assembly* deste

conteúdo.

O montador que será utilizado neste trabalho é o MPASMWIN, produto *freeware* da Microchip disponível no endereço *http://www.microchip.com/*.

Além do código com extensão .ASM, é necessário ter com o montador um segundo arquivo, com extensão .INC [GAL99]. No caso deste trabalho, o arquivo é o P16C84.INC, que contém algumas definições da qual depende o tipo de chip utilizado.

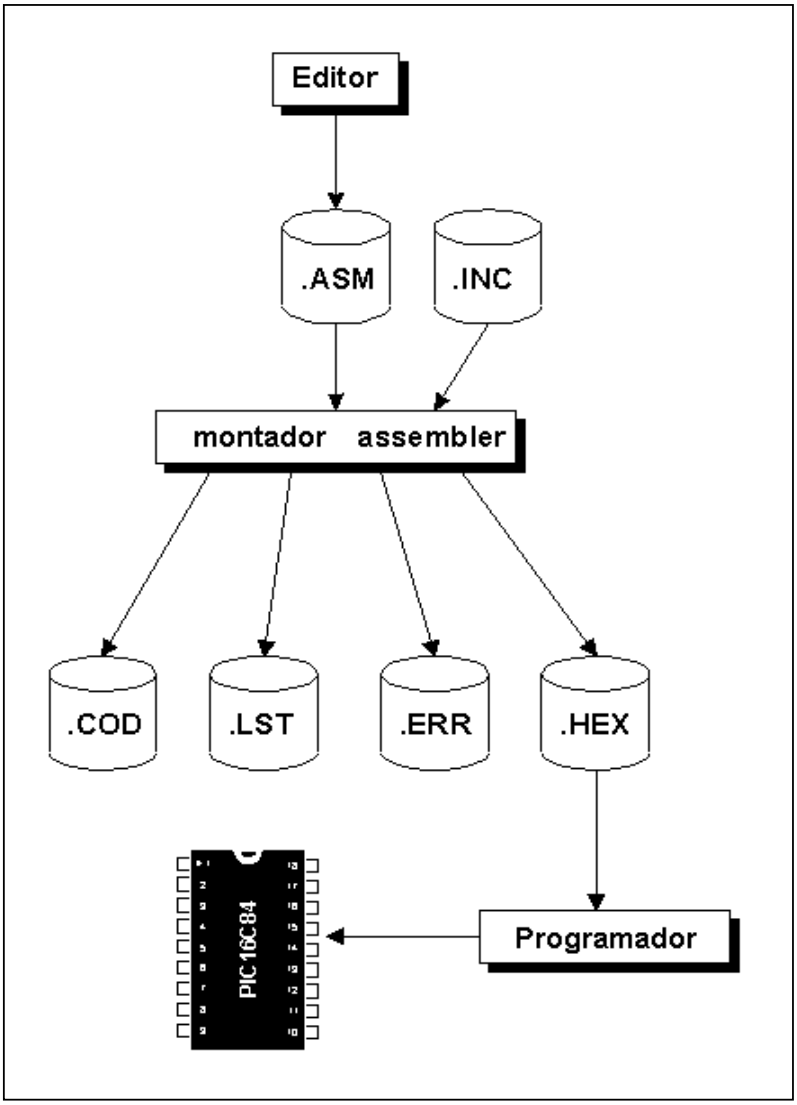


Figura 6 - Fluxograma de operações e arquivos

Durante a montagem do código, uma série de arquivos são gerados:

a) .HEX é o arquivo que contém o código de operação, o qual será gravado ao PIC via programador;
b) .LST é um arquivo de texto na qual vem reportado por inteiro o código *assembly* e a correspondente tradução em *opcode*. Não é utilizável pela programação do PIC mas é extremamente útil para verificar o processo de montagem que o compilador fez;
c) .ERR contem uma lista de erro de montagem que mostra o número da linha do código na qual está com erro no código *assembly*.

Os arquivos .LST e .ERR são utilizados somente para controle após a montagem. Somente o arquivo .HEX será utilizado realmente na programação do PIC.

O arquivo .HEX não é um arquivo no formato binário e não reflete diretamente o conteúdo que deverá ter a EEPROM do PIC. Sem entrar em detalhes é útil saber que tal formato é diretamente reconhecido pelo programador do PIC que promoverá durante a programação a conversão em binário e contém, outro *opcode* e outras informações que serão adicionadas aos endereços na qual irá transferir o *opcode* [GAL99].

3.4.2 COMO É CONSTITUÍDO UM PIC, QUAIS DISPOSITIVOS CONTÉM E COMO INTERAGIR ENTRE ELES.

Nesta parte serão descritos alguns componentes e suas interações. Não estarão descritos todos, apenas os relevantes diretamente e aqueles que mesmo indiretamente se fazem necessários para desenvolvimento desse trabalho.

3.4.2.1 A ÁREA DE PROGRAMA (EEPROM) E O REGISTRADOR DE ARQUIVO (REGISTER FILE)

Na figura 7 está ilustrado o esquema de blocos simplificado da arquitetura interna do PIC16C84.

A família PIC possui em sua arquitetura segmentos de memória separados para programas e dados. Como já visto, cada memória tem uma via separada no hardware interno, ou seja, os dois blocos podem ser acessados simultaneamente pelo programa em um mesmo

ciclo de máquina [SIL97].

A EEPROM é uma memória especial, regravável eletricamente, utilizada pelo PIC para memorizar o programa a ser executado.

A sua capacidade de memorização é de 1024 locações, as quais poderão conter somente um *opcode* a 14 bits, ou seja, uma instrução básica do PIC. Um programa por mais complexo que possa ser não poderá ter mais do que 1024 instruções [GAL99].

Os endereços reservados para EEPROM começam em 0000H e vão até 03FFH. O PIC pode somente executar instruções memorizadas nestas locações. Não se pode de maneira nenhuma ler, escrever ou cancelar dados nesses endereços [SIL97].

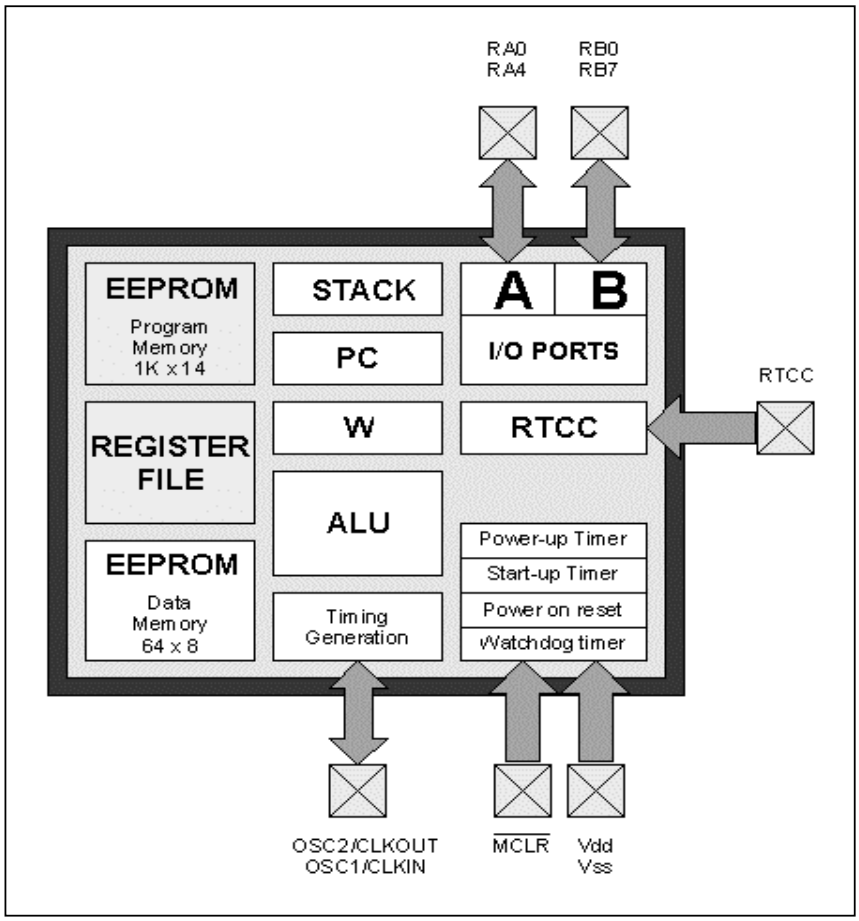


Figura 7 - EEPROM e Register File

Para escrever, ler e cancelar estas locações é necessário um dispositivo externo chamado programador. Exemplos de programadores são o YAPP! e o PICSTART-16+© produto da Microchip [GAL99]. O presente trabalho utilizou o PIP02, que pode ser visto no anexo 2.

Os membros da família 16FXXX podem acessar tanto direta quanto indiretamente qualquer posição de memória RAM ou dos registros internos, pois estão todos mapeados no mesmo bloco de memória. Qualquer operação pode ser feita com qualquer registro (de dados ou de controle) [SIL97].

A primeira locação de memória, o endereço 0000H, deve conter a primeira instrução que o PIC deverá executar após o *reset* e por isso é denominada *Reset Vector* [GAL99].

A diretiva *ORG 0000H* indica o inicio do programa. Esta diretiva indica de fato que a execução do programa após o *reset* deve iniciar no endereço 0000H da área de programa [BEN96].

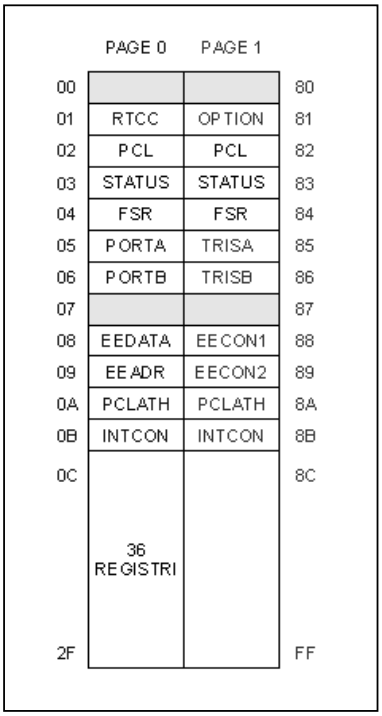


Figura 8 – Área de memória RAM

O *Register File* é uma parte da locação de memória RAM denominada Registro. Diferente da memória EEPROM destinada a conter o programa, a área de memória RAM é diretamente visível pelo resto do programa igualmente, onde pode-se escrever, ler, ou modificar qualquer endereço do *Register File* no programa a qualquer momento em que for necessário [GAL99].

A única limitação consiste de que alguns desses registros desenvolvem funções especiais pelo PIC e não podem ser utilizados para outra finalidade a não ser para aquela a qual eles estão reservados [GAL99]. Estes registros encontram-se nas locações base da área de memória RAM segundo o que está ilustrado na figura 8.

As locações de memória presentes no *Register File* são endereçadas diretamente em um espaço de memória que vai de 00H a 2FH num total de 48 bytes, denominada página 0. Um segundo espaço de endereçamento denominado página 1 vai de 80H a AFH. Para acessar esse segundo espaço é necessário recorrer a dois bits auxiliares RP0 e RP1 do registrador *STATUS* [GAL99].

As primeiras 12 locações da página 0 (de 00H a 0BH) e da página 1 (de 80H a 8BH) são aquelas reservadas as funções especiais para o funcionamento do PIC e, como já dito, não podem ser utilizadas para outra coisa. As outras 36 locações na página 0 podem ser endereçadas de 0CH a 2FH, podendo aqui ser utilizada livremente pelo programa para memorizar variáveis, contadores, etc [BEN96].

Os registros especiais do PIC serão utilizados com muita freqüência nos programas. Por exemplo, se for feita uma cópia dos registros especiais TRISA e TRISB, para definir qual linha de E/S será entrada e qual será saída, o mesmo estado lógico da linha de E/S depende do valor de dois registros, PORTA e PORTB. Alguns registros reportarão o estado de funcionamento do dispositivo interno do PIC ou o resultado de operações lógicas e aritméticas. Portanto, é necessário conhecer exatamente qual função desenvolve, cada um dos registros especiais e qual efeito se obtém ao manipular seus conteúdos. Para facilitar as operações de seus registros especiais, o P16C84.INC (que pode ser incluído no código .ASM com a diretiva INCLUDE) contém uma lista de nomes que identificam univocamente qualquer registro especial e a qual está associado o endereço correspondente na área do *Register File* [GAL99].

Segundo [SIL97], para definir toda a linha do PORTB do PIC em escrita agindo sobre o TRISB, pode-se escolher e referenciar diretamente o registro com o seu endereço conforme o quadro 5, ou então, referenciar o mesmo registro com o seu nome simbólico como está descrito no quadro 6. Para isso deve ser inserida a diretiva INCLUDE "P16C84.INC" no código.

```
movlw B'00000000'
movwf 06H
```

Quadro 5 – Escrita nas portas referenciando o endereço

```
movlw B'00000000'
movwf TRISB
```

Quadro 6 – Escritas nas portas referenciando o nome simbólico

3.4.2.2 A ALU E O REGISTRO W

A ALU (*Arithmetic and Logic Unit* ou seja unidade aritmética e lógica) é o componente mais complexo do PIC por conter todos os circuitos destinados a desenvolver as funções de cálculo e manipulação de dados durante a execução de um programa [GAL99].

A ALU é um componente presente em todos os microprocessadores e a capacidade de cálculo do micro depende diretamente dela.

A ALU do PIC16C84 está preparada para operar com 8 bits, ou seja, valor numérico não maior do que 255. Existem processadores com ALU de 16, 32, 64 bits e mais. A família Intel© 80386©, 486© e Pentium© por exemplo dispõe de uma ALU de 32 bits. A capacidade de cálculo presente nesses micros são notavelmente superior em detrimento da complexidade dos circuitos internos de acessoria e consequentemente do espaço ocupado [GAL99].

O registro W, denominado antes de acumulador, consiste de uma locação de memória

destinada a conter um só valor de 8 bits. A diferença entre o registro W e outras locações de memória consiste no fato de que, por referenciar o registro W, a ALU não pode fornecer nenhum endereço mas podemos acessá-los diretamente [GAL99].

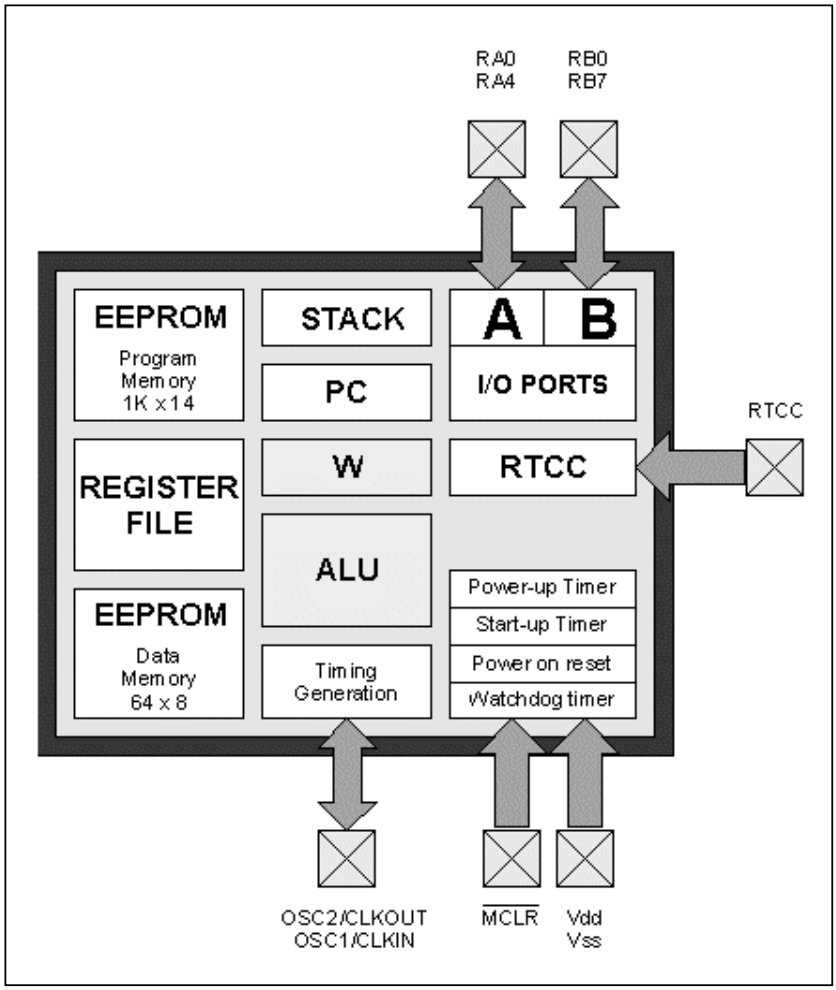


Figura 9 – ALU e Registro W

Por exemplo, para colocar na locação de memória 0CH do *Register File* o valor 01H, procurando entre as instruções do PIC, não existe uma única instrução capaz de efetuar esta operação, portanto deverá necessariamente recorrer ao acumulador e usar duas instruções em seqüência. Isto porque o *opcode* (código operacional) de uma instrução não pode exceder aos 14 bits e assim tem-se: 8 bits para especificar o valor que se deseja colocar na locação de memória, 7 bits para especificar em qual locação de memória deve-se inserir o nosso valor, 6

bits para especificar qual instrução queremos usar, tendo um total de 8 + 7 + 6 = 21 bits [GAL99]. Deverá, então, ser recorrido a duas instruções (veja quadro 7).

```
movlw    01H
movwf    0CH
```

Quadro 7 – Utilização do acumulador (registro W)

A primeira instrução do quadro 7 colocará no registro W o valor 01H com a instrução MOV Literal to W e depois "move-se" para locação 0CH com a instrução MOV W para F.

3.4.2.3 O CONTADOR DE PROGRAMA E O STACK

O PIC16C84 inicia a execução do programa a partir da locação de memória 0000H. Depois de ter executado esta instrução passa para a próxima instrução memorizada na locação 0001H e assim por diante. Se não existisse instrução capaz de influenciar a execução progressiva do programa, o PIC chegaria até o final na última instrução memorizada na última locação e não saberia mais como continuar. Porém, não é bem assim, pois qualquer sistema microprocessador dispõe de instrução de desvio, ou seja, instruções capazes de modificar o fluxo de execução do programa [GAL99].

Uma destas instruções é o *goto* (do inglês go to, vá para). Quando o PIC encontra um *goto* não segue mais a instrução imediatamente após, mas desvia-se diretamente para a locação de memória especificada na instrução. Um exemplo pode ser visto no quadro 8.

```
        ORG   00H
Point1
              movlw    10
              goto     Point1
```

Quadro 8 – Exemplo da instrução *GOTO*

No *reset* o PIC seguirá a instrução *movlw 10* memorizada na locação 0000H que colocará no acumulador o valor decimal 10, onde então passará à executar a próxima, *goto Point1*. Esta instrução determinará um desvio incondicional para locação de memória especificada pelo *label Point1*, ou seja, de novo para locação 0000H. O programa não fará outra coisa se não a de executar um ciclo infinito seguindo continuamente as instruções especificadas.

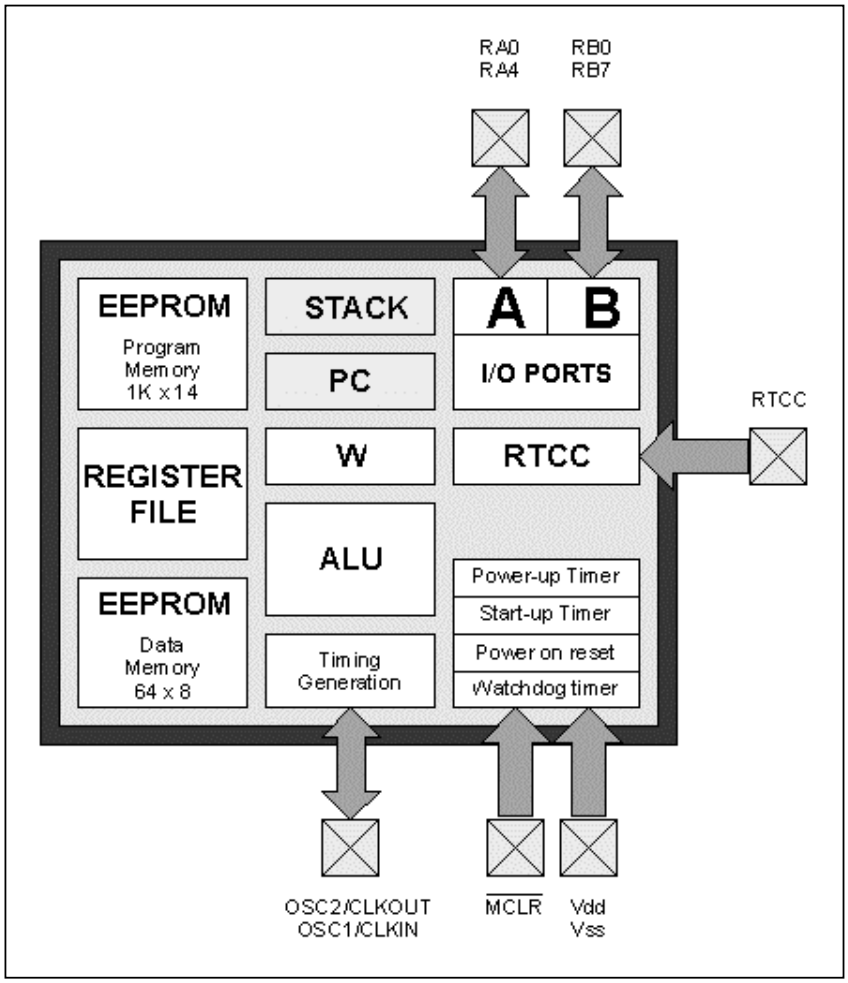


Figura 10 – Stack e PC

Durante este ciclo, para determinar qual é a próxima instrução a ser seguida, o PIC utiliza um registro especial denominado PC (*Program Counter*), ou seja, contador de programa. Este terá sempre o endereço da próxima instrução a ser executada. No *reset* este

estará sempre zerado, determinando o inicio da execução no endereço 0000H, e a cada instrução terá um incremento de um para poder passar para próxima instrução [GAL99].

A instrução *goto* permite a colocação de um novo valor no PC e consequentemente desviá-la para uma locação qualquer da área de programa do PIC.

Para continuidade do trabalho, em versões futuras, deverá ser implementado a chamado a subrotinas, que é realizado pela instrução *call.*

Esta instrução funciona de maneira muito similar ao *goto.* A única diferença é que a primeira desvia para uma locação de memória especificada e continua a execução do programa, enquanto o *call* desviará o programa para uma subrotina especificada, executará a mesma, e após executar a instrução *return,* retornará a execução da instrução imediatamente após a chamada *call*. O valor imediatamente após a chamada *call* será armazenado em uma área particular da memória denominada *Stack* (pilha) [GAL99]. Veja o exemplo do quadro 9.

```
        ORG     00H
Point1
        movlw   10
        call    Point2
        goto    Point1
Point2
        movlw   11
        return
```

Quadro 9 – Exemplo da instrução *CALL*

Neste caso o PIC, após ter executado *movlw 10* passa a executar o *call Point2*. Antes de desviar, memoriza no *Stack* o endereço 0002H, ou seja, o endereço da próxima locação ao *call*. Passa então a executar a instrução *movlw 11*, memorizada em correspondência ao *label Point2*. Neste ponto encontra uma nova instrução, o *return* que, como se pode deduzir de seu nome, permite o "RETORNO", ou seja, retorne a execução da instrução imediatamente após o *call*.

Esta operação é denominada de "chamada a subrotina", ou seja, uma interrupção momentânea do fluxo normal do programa para "chamar" a execução de uma série de instruções, para depois retornar a execução normal do programa.

Para poder retornar para onde havia interrompido, o PIC utiliza o último valor armazenado no *Stack* e o coloca de novo no PC [GAL99].

A palavra *stack* em inglês significa "pilha" e por esse fato é possível empilhar um endereço sobre o outro para ser recuperado quando necessário. Este tipo de memorização era antes denominado de LIFO (do inglês *Last In First Out*), em que o último elemento armazenado deve necessariamente ser o primeiro a sair.

Graças ao *Stack* é possível efetuar vários *call*, um dentro do outro e manter sempre o retorno ao fluxo do programa quando se encontra uma instrução *return*, como no exemplo do quadro 10.

```
        ORG     00H
Point1
        movlw10
        call Point2
        Goto Point1
Point2
        movlw11
        call Point3
        return
Point3
        movlw   12
        return
```

Quadro 10 – Exemplo da utilização de várias instruções *CALL*

No exemplo acima, a rotina principal Point1 promove a chamada do primeiro *call* para subrotina Point2, a subrotina Point2 chama outra subrotina no caso Point3, este último por sua

vez, encontra um *return* e retorna para Point2 que encontra o outro *return* e retorna para a execução da rotina Point1 que no caso é a principal.

Os endereços a serem memorizados no *stack* são dois e quando vir a encontrar um segundo *call* procurará pelo *return* correspondente ao primeiro e assim por diante. Se diz então que o *call* é "aninhado", ou seja, um dentro do outro [GAL99]. É importante assegurar-se, durante a formulação de um programa que se tenha sempre uma instrução *return* em correspondência a um *call* para evitar o perigo de desalinhamento do *stack* que em execução pode gerar erros que dificilmente será encontrado.

O PIC16C84 dispõe de um *stack* de 8 níveis, ou seja, um *Stack* que consegue armazenar no máximo 8 chamadas à subrotina. Se mais de oito *CALL*'s ou interrupções forem atendidas simultaneamente, o primeiro endereço de retorno será perdido, sobrescrito pelo nono, e assim por diante, de forma circular. Este fato deve ser controlado pelo programador, pois os membros desta família não possuem sinalizadores de *overflow* ou *underflow* do *Stack* [SIL97].

Não há nenhuma instrução que permita a manipulação da pilha (*stack*), ou seja, o programador não pode acessá-lo [BEN96]. Isto se deve ao fato de que na família PIC16CXXX o *stack* não possui um registro de ponteiro (ou *Stack* Pointer), o qual torna-se um caminho para acessá-lo, presente em microprocessadores comuns [SIL97].

3.4.2.4 PORTA A E PORTA B

O PIC16C84 necessita trocar informações com o mundo real, por isso dispõe de um total de 13 linhas de E/S organizadas em duas portas denominadas de PORTA A e PORTA B. A PORTA A dispõe de 5 linhas configuráveis tanto em entrada como em saída identificadas pelas siglas RA0, RA1, RA2, RA3 e RA4. A PORTA B dispõe de 8 linhas também configuráveis seja em entrada ou em saída identificadas pelas siglas RB0, RB1, RB2, RB3, RB4, RB5, RB6 e RB7. A subdivisão da linha em duas portas diferentes é devido ao tipo de arquitetura interna do PIC16C84 que prevê um controle de dados de no máximo 8 bits [GAL99].

Para o controle da linha de E/S do programa, o PIC dispõe de dois registros internos que controlam as portas e são chamados de TRISA e PORTA para a porta A e TRISB e

PORTB para a porta B [SIL97].

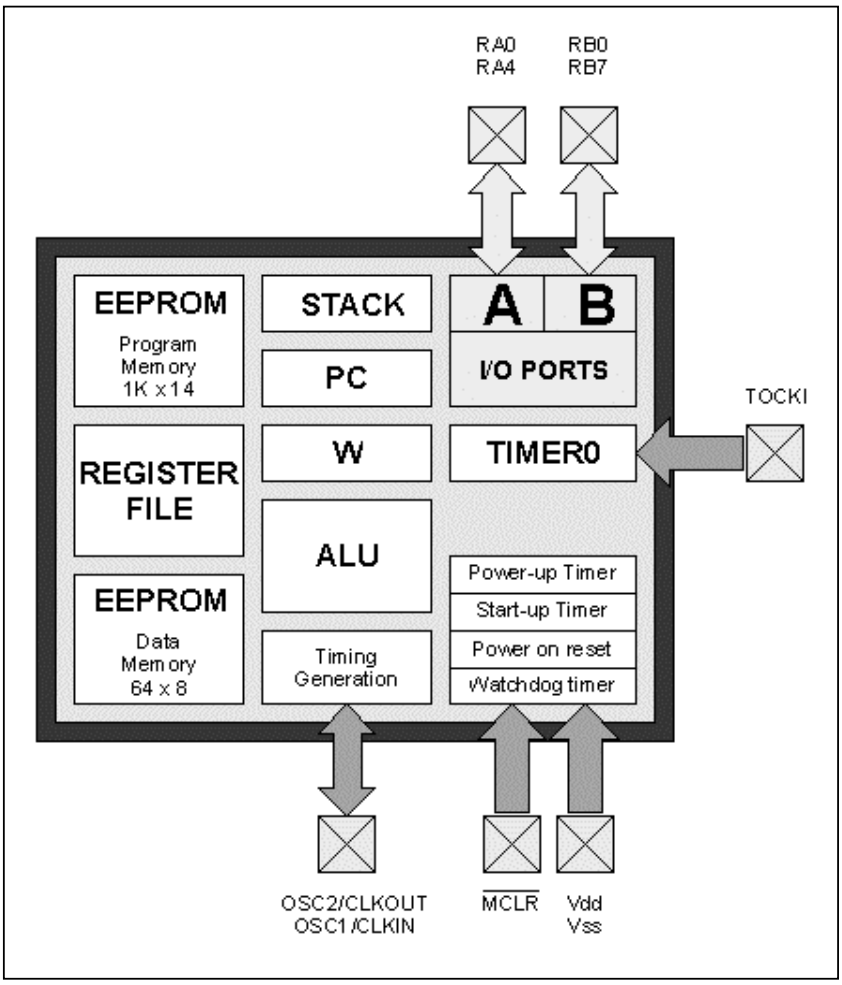


Figura 11 – Porta A e Porta B

Os registros TRISA e TRISB determinarão o funcionamento em entrada ou em saída da mesma linha, ou seja, o sentido de transferência. O registro PORTA e PORTB determinarão o *status* da linha em saída ou reportarão o *status* da linha em entrada [GAL99]. Isto permite que o projeto do hardware proporcione grande flexibilidade, usando um só pino com as duas funções.

Todos os bits contidos nos registros mencionados correspondem univocamente a uma linha de E/S, ou seja, o bit 0 do registro PORTA e do registro TRISA correspondem a linha RA0 , o bit 1 a linha RA1 e assim por diante [BEN96].

Se o bit 0 do registro TRISA for colocado em zero, a linha RA0 estará configurada como linha de saída, por isso o valor a que terá o bit 0 do registro PORTA determinará o estado lógico de tal linha (0 = 0 volts, 1 = 5 volts). Se o bit 0 do registro TRISA for colocado em um a linha RA0 estará configurada como linha de entrada [GAL99].

Como um exemplo prático, querendo conectar um *LED* sobre a linha RB0 e uma chave sobre a linha RB4, o código a se escrever será o contido no quadro 11, onde será colocado em 0 o bit 0 (linha RB0) em escrita (saída), e em 1 o bit 4 (linha RB4) em entrada. É importante lembrar que na notação binária do *assembly*, o bit mais a direita corresponde com o bit menos significativo, por isso o bit 0. Para acender o *led*, deve-se escrever o código *bsf PORTB,0,* enquanto que para apagá-lo deve-se escrever *bcf PORTB,0.*

```
movlw 00010000B
tris  B
```

Quadro 11 – Código para conexão de um LED a linha RB4

O código que possibilita a leitura do estado da chave conectada a linha RB4 está expresso no quadro 12.

```
btfss   PORTB,4
goto    SwitchAMassa
goto    SwitchAlPositivo
```

Quadro 12 – Código para leitura do estado da chave conectada a linha RB4

3.4.3 CONJUNTO DE INSTRUÇÕES DO PIC16C84

Como já foi visto, o conjunto de instruções do PIC16C84 é formado por 35 instruções, todas formadas por apenas uma palavra de 14 bits. Nela estão identificadas o código da

função a ser executada, além dos parâmetros necessários, como constantes, registros, bits [SIL97].

Na tabela 1 pode-se observar o resumo do conjunto de instruções destinada a operações de byte com registros.

Instrução	Operandos	Descrição	Bits afetados
ADDWF	f,d	Soma w e f	C,DC,Z
ANDWF	f,d	AND entre w e f	Z
CLRF	F	Zera f	Z
CLRW	-	Zera w	Z
COMF	f,d	Complementa f	Z
DECF	f,d	Decrementa f	Z
DECFSZ	f,d	Decrementa f Pula se f=0	-
INCF	f,d	Incrementa f	Z
INCFSZ	f,d	Incrementa f Pula se f=0	-
IORWF	f,d	OR entre w e f	Z
MOVF	f,d	Move f	Z
MOVWF	F	Move w para f	-
NOP	-	Nenhuma operação	-
RLF	f,d	Roda a esquerda pelo carry	C
RRF	f,d	Roda a direita pelo carry	C
SUBWF	f,d	Subtrai w de f	C,DC,Z
SWAPF	f,d	Troca nibles em f	-
XORWF	f,d	XOR entre w e f	Z

Tabela 1 – Instruções das operações de byte com registros

Na tabela 2 tem-se o resumo do conjunto de instruções destinada a operações de bit com registros.

Instruções	Operandos	Descrição	Bits afetados
BCF	f,b	Zera bit ‘b’ em f	-

BSF	f,b	Seta bit ‘b’ em f	-
BTFSC	f,b	Se bit ‘b’ em f=0, pula	-
BTFSS	f,b	Se bit ‘b’ em f=1, pula	-

Tabela 2 – Instruções das operações de bit com registros

A tabela 3 mostra o resumo do conjunto de instruções destinada a operações com constantes e de controle.

Instrução	**Operandos**	**Descrição**	**Bits afetados**
ADDLW	k	Soma w e k	C , DC , Z
ANDLW	K	AND entre w e k	Z
CALL	K	Chama sub-rotina	-
CLRWDT	-	Zera o timer do Watch Dog	TO\ , PD\
GOTO	K	Desvia para o Label ‘k’	-
IORLW	K	OR entre w e k	Z
MOVLW	K	w = k	-
RETFIE	-	Retorna da interrupção	-
RETLW	K	Retorna com w = k	-
RETURN	-	Retorna de sub-rotina	-
SLEEP	-	Entre no modo SLEEP	TO\ , PD\
SUBLW	K	Subtrai k de w	C , DC , Z
XORLW	K	XOR entre w e k	Z

Tabela 3 – Instruções das operações com constantes e de controle

O significado das variáveis utilizadas nas tabelas 1, 2 e 3 pode ser visto no quadro 13.

f : registro entre 0 e 127 (0 à 7FH); w ou W : registro W;

b : bit utilizado pela operação; k : constante ou label;

d : destino do resultado

- se d = 0, o resultado é armazenado em W;
- se d = 1, o resultado é armazenado no próprio registro indicado na operação.

Quadro 13 – Significado das variáveis utilizadas nas tabelas 1, 2 e 3.

3.4.4 CONSIDERAÇÕES SOBRE O MONTADOR DO PIC16C84

O formato do programa para esse microcontrolador segue o padrão mostrado no quadro 14.

Label:operação operando(s) ; comentários

Quadro 14 – Formato do programa

O campo *label* é facultativo e indica posições particulares do programa [SIL97]. É um nome simbólico mnemônico que está associado a um endereço. O termo mnemônico significa facilitar a memorização. Serve tanto para atribuição de variáveis como para destino de comandos *goto*. Os *labels* tem que estar na primeira posição da linha, ou seja, na coluna 1 [BEN96].

O campo operação sempre existe e indica qual operação a ser realizada. O campo operando(s) existirá se for necessário à instrução. Após o ";" tudo será ignorado pelo compilador, ou seja, tudo que vier depois dele será considerado apenas comentário [SIL97].

Segundo [BEN96,] este padrão divide uma linha em três colunas, isto para encontrar itens específicos. Por sua vez, essas colunas são utilizadas em cinco formas (seções):

a) cabeçalho;
b) seção *equate;*
c) declaração do *org (origin);*
d) corpo do programa;
e) declaração do término do programa.

As informações no topo do programa são chamados de cabeçalho (*header*). Nesta parte será definido, como no exemplo do protótipo, o tipo *default* das constantes por meio da diretiva *radix dec* (neste caso, o *default* será decimal), e definir qual o grupo de comandos será utilizado. Este grupo é relativo ao microcontrolador para qual se está construindo o programa [BEN96]. Neste trabalho será usado a linha de comando *list p=16c84.*

A diretiva *list* indicará ao montador que o grupo de comandos que estão sendo utilizados são para o microcontrolador PIC16C84.

A declaração *EQU* (*equates*), além de servir para associar um *label* a um endereço específico, é usado também para associar nomes a números.

Neste trabalho a declaração *ORG* (*origin*) terá como propósito principal definir o endereço onde o programa começa. A declaração *END* é usada para dizer ao montador que ele alcançou o fim do programa, ou seja, é a última linha [BEN96].

Quanto as constantes, para o compilador elas tem o seguinte formato:

a) Constante decimal: D'valor' ou d'valor';
b) Constante binária: B'xxxxxxxx';
c) Constante hexadecimal: 0x'valor' ou valorH.

O montador tem como padrão valores hexadecimais, logo se não for indicado o tipo da constante o compilador assumirá hexadecimal. Além disso, é importante saber que as constantes hexadecimais que iniciarem por letra (A-F) devem ser precedidas de '0' [SIL97].

4 TRABALHOS CORRELATOS

4.1 ONAGRO

Como no trabalho aqui proposto, o Onagro também é um sistema de tradução que reconhece uma descrição de algoritmos em linguagem gráfica e permite a geração de código em *assembly* para microcontroladores. Incorpora um editor gráfico para a entrada do programa fonte que é semelhante ao algoritmo descrito na linguagem fluxogramática. ONAGRO possui uma *interface* amigável, sendo implementado na linguagem visual C++ usando a metodologia de orientação a objetos. Os testes realizados mostraram que o ambiente proposto é muito intuitivo e amigável. Outro aspecto importante observado nos testes é que o código gerado provou ser compacto [UNI99].

4.1.1 AMBIENTE PROPOSTO

ONAGRO permite a entrada do dicionário de dados e fluxogramas de uma aplicação e gera automaticamente o código em *assembly* destas entradas. Os fluxogramas são introduzidos por um editor gráfico, e estão compostos por ícones de vários tipos de operações pré-definidas, ou por operações definidas pelo usuário. Os ícones serão unidos, determinando o fluxo de execução. O *dicionário de dados* é composto de uma descrição de sinais de identificadores (constantes, variáveis, portas de entrada e saída e sub-rotinas) usados no fluxograma [UNI99].

Depois da entrada do fluxograma e do dicionário de dados, o usuário pode ativar o compilador para executar a alocação de dados e a conversão das operações dos ícones para linguagem *assembly*. Também é feita a ativação de um editor de ligação *assembly* para gerar o código de máquina, como também a comunicação do ambiente programado com o sistema proposto, usando um programa de comunicação serial. A programação das interrupções dos microcontroladores, como também os dispositivos de *timer* e as *interfaces* de comunicação serial são feitas por diálogos amigáveis com os usuários, permitindo uma abstração melhor dos detalhes de operação dos recursos dos microcontroladores [UNI99].

4.1.2 OPÇÕES DE MENU DO ONAGRO

Quando o ONAGRO é iniciado, uma janela chamada tela principal é ativada. Esta tela,

mostrada na figura 12, é composta de:

a) menu para selecionar a tarefa que o usuário quer executar;
b) a barra dos Ícones de Operação é composta de botões pré-definidos, usados para representar as instruções da linguagem;
c) barra de Ferramentas, também composta de botões que permitem a seleção do modo mais rápido das tarefas principais disponíveis no menu;
d) área de desenho do programa fonte;
e) linha de *Status* e Botões para rolar a área de desenho.

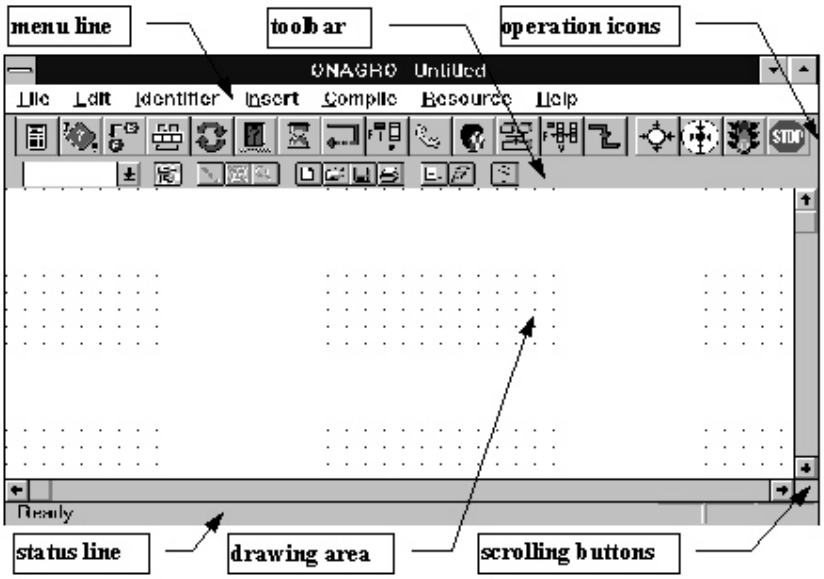


Figura 12 – Onagro: Tela principal

4.1.3 OS ÍCONES DE OPERAÇÃO

Os ícones de operação representam as operações básicas que podem ser executadas na linguagem. Eles são semelhantes a códigos de operação de uma instrução de máquina, e indicam a natureza do processo que é exigido para ser executado. Além disso, é necessário que sejam indicados os parâmetros que serão usados na operação. Estes parâmetros são normalmente identificadores de variáveis, portas de entrada e saída e identificadores de constantes. A linguagem oferece ícones já definidos para executar as operações mais comuns. Porém, ícones mais específicos são oferecidos para aplicações dedicadas. Além disso, o

usuário pode criar seus próprios ícones de operação e escolher qual tarefa deverá ser executada. Eles são organizados em classes operacionais onde cada classe é representada por uma figura diferente e indica um grupo de operação semelhante. Na maioria dos casos, o ícone que representa a classe é o mesmo que indica a operação mais usada nesta (figura 13 a). Porém, existem classes que têm um ícone específico para representá-lo, diferente das outras classes de ícones de operação. Isto é mostrado na figura 13 b [UNI99].

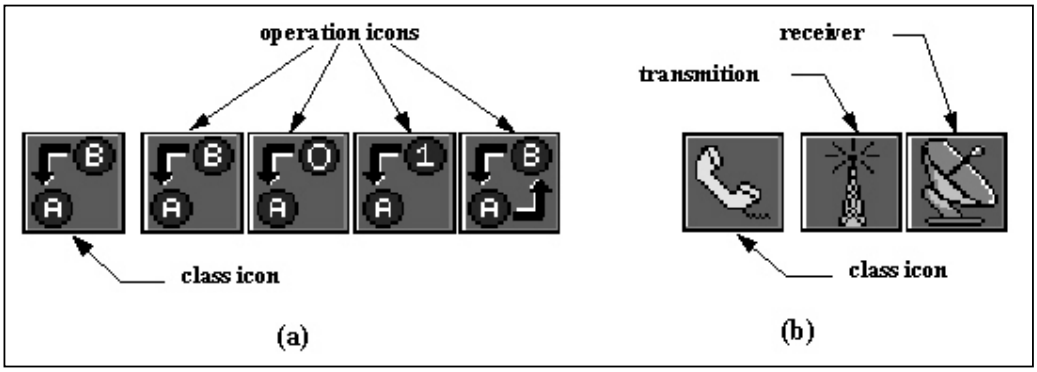


Figura 13 – Onagro: Ícones

4.1.4 A ESTRUTURA DO CÓDIGO E OS MECANISMOS DE TRADUÇÃO

ONAGRO tem o código orientado a ícones e as operações são descritas por um arranjo de símbolos padronizados (os ícones de operação). Estes arranjos têm uma sintaxe guiada por um editor gráfico, de tal modo que apenas as declarações operacionais corretas são permitidas. Não só reduz os testes de sintaxe em excesso que possa ocorrer nas linguagens tradicionais, mas também gera uma estrutura de código mais previsível ao compilador. Na geração do código, é tentado escolher um mecanismo que seja o mais genérico possível, conforme o tipo de microcontrolador. Assim, a parte principal do código gerado usa as instruções essenciais e a arquitetura básica dos registradores do microcontrolador. Obviamente, para cada microcontrolador as operações são relacionadas a suas características, pois cada um tem suas particularidades [UNI99].

4.1.5 RESULTADOS

Foram executados dezoito testes com cinco aplicações compiladas em dois tipos de modelos de memória. A intenção era comparar o tamanho de código gerado pelo ONAGRO com outro compilador comercialmente disponível. Foi escolhido utilizar o MCS-51 da família INTEL. Todos os testes feitos com o ONAGRO também foram realizados com o compilador C AVOCET que é um compilador bem conhecido desta família. Analisando o gráfico de barra (figura 14), nota-se que em todos os testes, exceto no nono e décimo, o código gerado pelo ONAGRO era menor que o gerado na linguagem C. Parte destes resultados foram obtidos porque ONAGRO tem, concernido na linguagem C, algumas restrições relacionadas para o tipo dos dados manipulados [UNI99].

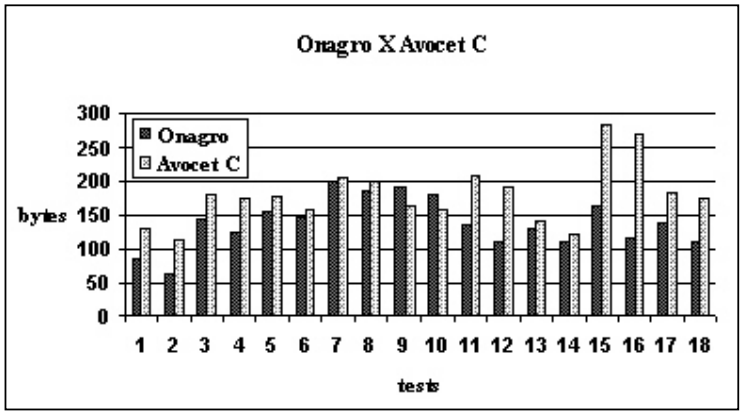


Figura 14 – Onagro x Avocet C

Outro fator importante que explica os resultados obtidos é que há ícones de operação específicos para operações dedicadas no ONAGRO, designado em um modo aperfeiçoado. Em outra linguagem (no caso de linguagem C) estas mesmas operações são compostas de instruções simples disponíveis na linguagem, e sendo usado do modo correto, pode gerar código em excesso. Afinal, o algoritmo de tradução usado pelo ONAGRO é baseado no intercâmbio de mensagens entre os objetos envolvidos nas operações. Isto permitiu, em grande parte que o código gerado era semanticamente correto, porque a maioria dos inconvenientes da generalização de uma determinada operação são eliminadas pelo compilador, pela comunicação entre os elementos que compõem tal operação. Porém, quando a aplicação usa sub-rotinas que entrem com parâmetros e retornem algum valor, o ONAGRO

gerou código pior que a linguagem C, como mostrado nos resultados obtidos nos testes 9 e 10. A análise do código gerado mostrou que os parâmetros e a manipulação dos valores retornados feitos na linguagem C parece ser aperfeiçoada mais que no ONAGRO. Outros dados que conduziu a esta conclusão é o fato de que o código gerado nos testes 11, 12, 13 e 14 estavam a favor do ONAGRO. Nestes testes foi usado também subrotinas sem parâmetros ou retorno de valores [UNI99].

5 TTREE

5.1 INTRODUÇÃO

Um objeto *TTree* consiste de formas (*shapes*) e conexões em um estilo orientado à árvore. Isto quer dizer que todo *shape* pode ter nenhum (raiz) ou vários pais, e também pode ter nenhum ou vários filhos.

O *TeeTree* é muito pequeno e não usa qualquer DLL do Windows, além do consumo de memória ser extremamente baixo [TEE98].

Os componentes do *TeeTree* podem ser usados para muitos tipos diferentes de aplicações, como gráficos organizacionais, fluxogramas (veja na figura 15), diagramação, gráficos de rede, enfim, para representar qualquer organização de dados de forma hierárquica.

Existem 3 classes chaves de componente, como pode ser visto na figura 15, as quais são:

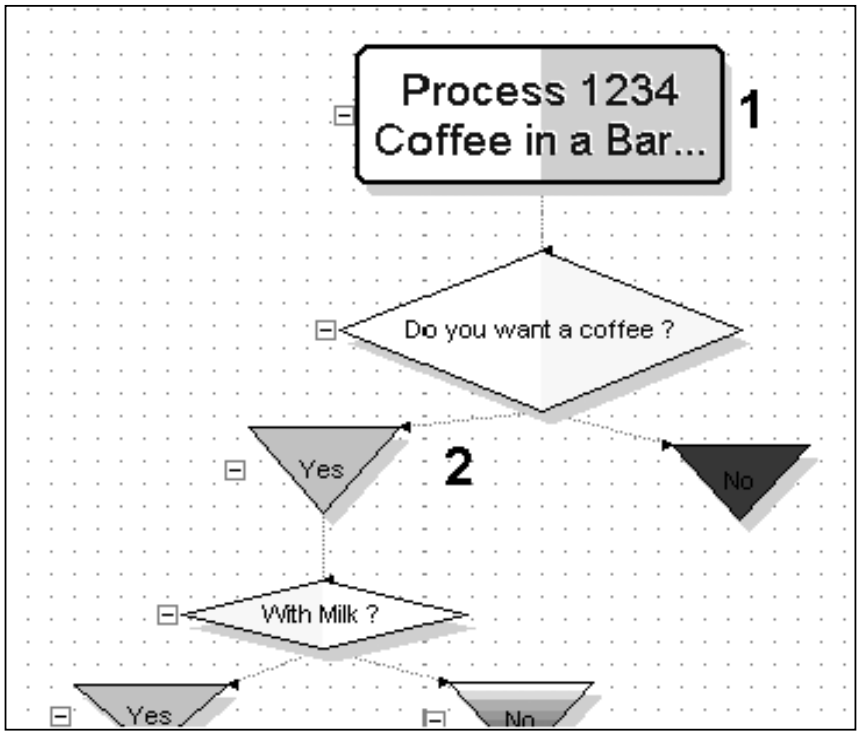


Figura 15 – Objeto Ttree

a) árvore (tree) – área pontilhada da figura 15;
b) nodos (shapes) – número 1 da figura 15;
c) conexões (connections) – número 2 da figura 15.

Tanto a árvore (objeto *Ttree*) como todos os *shapes* têm muitas propriedades para customizar, utilizando métodos e eventos para notificar cliques no *shape*, seleção, expansão/encapsulamento, etc. O tamanho da árvore (em pixels) e o número de nodos afetam proporcionalmente o desempenho da mesma.

Interiormente, há uma hierarquia de classes com antepassados abstratos [TEE98].

5.2 PROPRIEDADES E MÉTODOS MAIS UTILIZADOS X MAIS IMPORTANTES

Na tabela 4 serão relatados alguns dos métodos e propriedades mais utilizados e mais importantes para a implementação do protótipo.

Propriedade/Método Nome	**Descrição**
Propriedade Designing	quando True, permite que o shape selecionado possa ser movido e redimensionado.
Propriedade Connections	contém uma lista de todos as conexões dos nodos. Quando indicado com um índice, pode-se alterar características da conexão (linha) como por exemplo, tipo da linha e ponta da seta.
Propriedade Selected	contém uma lista de todos os nodos que estão selecionados no momento.
Método Clear (procedure)	exclui todos os nodos do objeto Ttree, ou seja, limpa a área da árvore.
Método AddRoot (function)	inclui um nodo raiz no objeto Ttree.
Propriedade AutoPosition.Left/AutoPosition.Top	quando True, as coordenadas X0/Y0 são calculadas baseando-se na posição do nodo ascendente.
Propriedade X0	coordenada do pixel do lado esquerdo do nodo.

Propriedade X1	coordenada do pixel do lado direito do nodo.
Propriedade Xcenter	coordenada do pixel central/horizontal do nodo.
Propriedade Y0	coordenada do pixel superior do nodo.
Propriedade Y1	coordenada do pixel inferior do nodo.
Propriedade Ycenter	coordenada do pixel central/vertical do nodo.
Propriedade Brush.Color	cor interna do nodo.
Propriedade Style	forma do nodo (círculo, retângulo, losângulo, etc).
Propriedade Childs	lista de nodos descendentes. Pode-se acessar os descendentes de um nodo indexando esta propriedade (Childs[i]).
Método AddChilds (function)	acrescenta um nodo descendente ao nodo selecionado, retornando o nodo criado.
Método AddConnections (function)	Acrescenta uma conexão ao nodo selecionado, destinando-se ao nodo passado na função. Retorna a conexão criada.
Método MoveRelative (procedure)	Incrementa o valor das propriedades X0 e Y0 do nodo, a partir dos parâmetros passados.

Tabela 4 – Métodos e propriedades do objeto Ttree

No quadro 15 pode ser visto um exemplo da utilização desses métodos e propriedades.

```
procedure TForm1.FormActivate(Sender: TObject);
var
  filho2:ttreenodeshape;
begin
  Tree1.AddRoot('Raiz');
  Tree1[0].expanded:=true;
  Tree1[0].AddChild('filho1');
  filho2:=Tree1[0].AddChild('filho2');
  Tree1[0].Childs[0].Style:=tsscircle;
  filho2.AddChild('');
  filho2.Brush.Color:=claqua;
end;
```

Quadro 15 – Exemplo da utilização de métodos e propriedades do *Ttree*

Após as linhas de código descritas no quadro 15, o objeto *Ttree* (representado pela variável Tree1) ficará com o aspecto da figura 16.

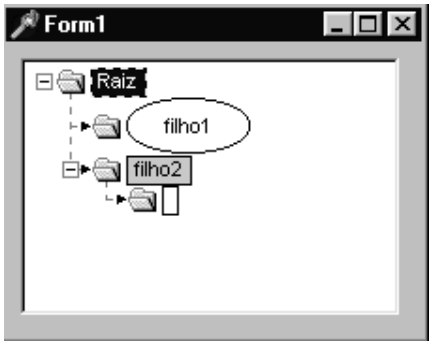


Figura 16 – Tela exemplo de métodos do objeto Ttree

6 ESPECIFICAÇÃO, IMPLEMENTAÇÃO E TESTES DO PROTÓTIPO

Na figura 17 está descrita a especificação do protótipo através de um DFD de nível 0.

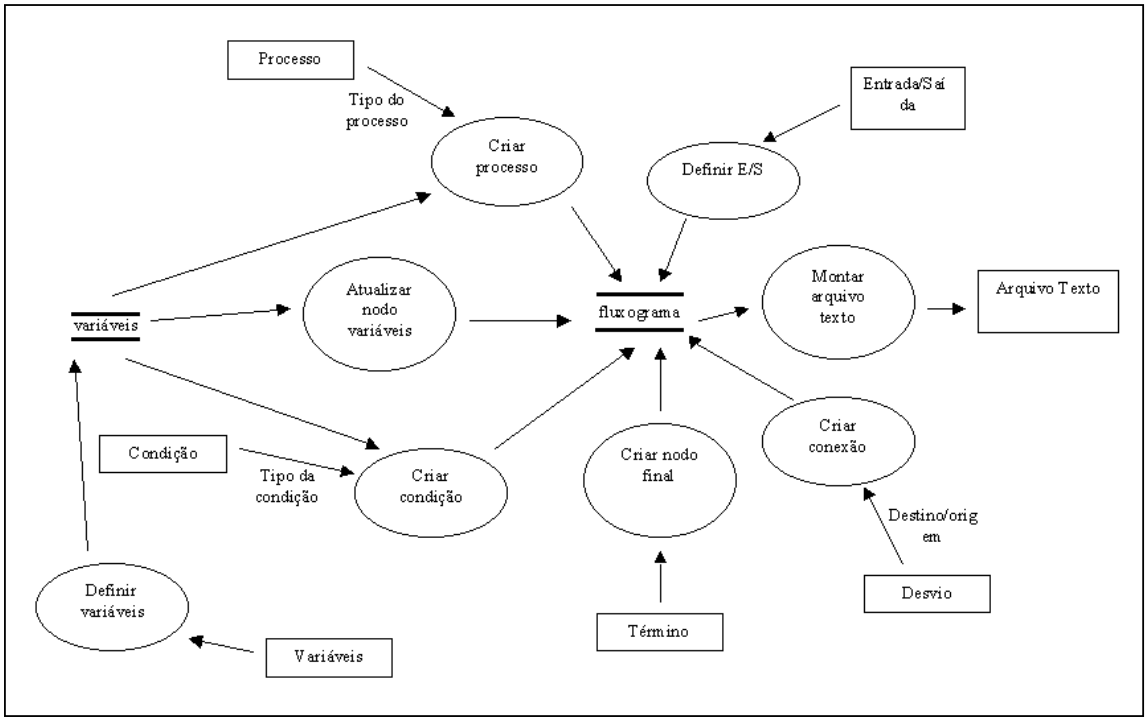


Figura 17 – Especificação do protótipo

O protótipo foi desenvolvido no ambiente de programação Delphi 3. Para a montagem do editor de fluxogramas foi utilizado um componente chamado *Teetree* (veja capítulo 5). Também foram utilizadas algumas técnicas de pesquisa em árvore, facilitadas devido a utilização do *Teetree*.

A partir do componente *Teetree* é possível montar o fluxograma, pois este contém métodos e propriedades que dinamizam a construção de qualquer tipo de árvore. Neste trabalho, logicamente, cada nodo da árvore poderá ter apenas um ascendente e um descendente, características dos fluxogramas, com exceção das condições, onde é necessário ter dois caminhos, sendo um para o "sim" e outro para o "não". Como todo fluxograma necessita sinalizar o início e o término, isto também será implementado no protótipo, com

exceção dos fluxogramas com *loop* infinito.

Para montagem do arquivo texto, será utilizado um algoritmo recursivo, percorrendo todo a árvore (fluxograma) e copiando o conteúdo da propriedade *text* (linhas de texto) do componente *Teetree* para o componente *memo* do Delphi. Este último componente armazena também linhas de texto e assim torna possível gravar os dados para posterior montagem.

Um objeto é dito recursivo se ele consiste parcialmente ou é definido sobre seus próprios termos. Recursão é uma técnica muito poderosa nas definições matemáticas. A ferramenta necessária e suficiente para expressar programas recursivos são os procedimentos e subrotinas [WIR86]. A procedimentos que especifica a montagem recursiva do arquivo texto será mostrada no item 6.1.9.

Além das facilidades que o protótipo incorpora por gerar código *assembly* a partir de uma representação visual (fluxogramas) e operações pré-definidas, também preocupou-se na simplicidade das telas, proporcionando pouca poluição visual e diminuindo o trabalho do usuário final, pois a *interface* homem-máquina se mostrou bem amigável.

Como já foi citado anteriormente, este protótipo foi disponibilizado na internet para qualquer pessoa interessada fazer os testes. Para tanto, esta versão foi traduzida para o inglês para melhor entendimento.

Tanto as características visuais (telas) quanto as internas (código) serão apresentadas neste capítulo.

6.1 TELA PRINCIPAL

Como se pode ver na figura 18, na tela principal os botões de funções estão dispostas de modo a facilitar o trabalho do operador.

6.1.1 BOTÕES PADRÕES

Os botões *New, Open, Save, Close* e *Exit* tem funções muito parecidas com a de outros aplicativos. A *procedure Salvar* além de salvar o arquivo .ASM (texto) para posterior montagem, salva também o fluxograma com todas as suas características (extensão .TEE).

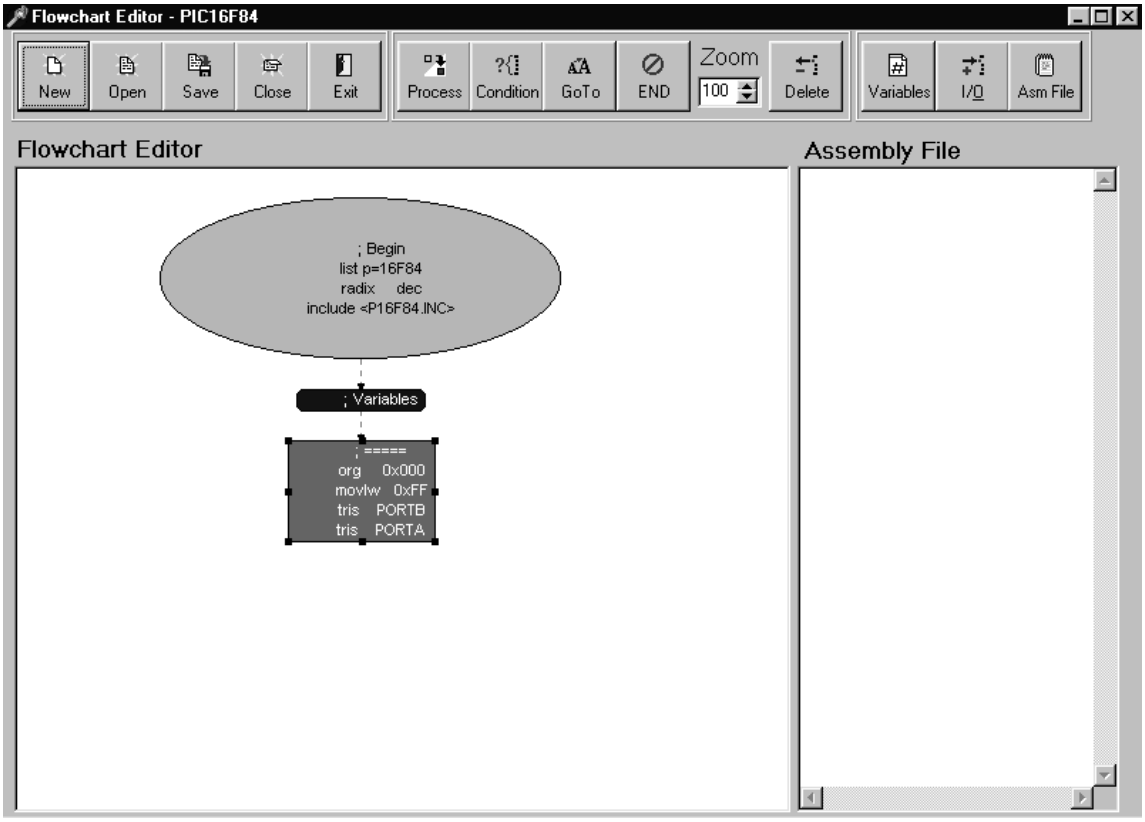


Figura 18 – Tela Principal

6.1.2 BOTÕES PROCESS E CONDITION

Com poucas diferenças, basicamente os botões *Process* e *Condition* - que são os mais utilizados - após fazerem algumas consistências, acionam uma outra tela (tela *Process* e tela *Condition*, respectivamente) onde serão escolhidas as operações e condições que o futuro nodo irá representar. As consistências são:

a) algum nodo está selecionado: é necessário que algum nodo esteja selecionado para poder incluir um descendente, ou seja, um nodo filho (método AddChild);

b) nodo selecionado é nodo raiz (elipse, figura 18), ou nodo que contém variáveis (retângulo fino, figura 18), ou nodo final, ou nodo condição: nos dois primeiros casos, não podem ser inseridos nodos pois os comandos só são aceitos após a inicialização do programa, que corresponde ao terceiro nodo da figura 18. Após o nodo final logicamente não poderão ser inseridos novos comandos. Depois da condição serão inseridos automaticamente dois nodos – um para a condição verdadeira; e outro para a condição falsa – que forçosamente serão os primeiros comandos a serem executados dependendo do resultado da condição.

Se o nodo selecionado já tem algum descendente, será chamada uma função para inserir o novo nodo, depois do selecionado e antes do descendente do mesmo, retornando o nodo inserido. Esta função cria uma árvore temporária, armazenando os descendentes do nodo selecionado, e após ter sido incluído o novo, acoplará a árvore temporária ao novo nodo.

Demais diferenças entre esses botões estão na disposição visual (cor, forma e posição) e no fato já identificado acima, de que a condição terá automaticamente dois descendentes, o nodo que segue o resultado *"sim (yes)"* e o que segue o *"não (no)"*.

6.1.3 BOTÕES GOTO E TO

O botão *GOTO* basicamente fará a consistência do nodo selecionado já ter alguma conexão, o que naturalmente o impossibilitaria de seguir outro caminho. Além disso também desabilita os demais botões e tornando visível sobre ele o botão *TO*, logo após ter mostrado uma mensagem pedindo que selecione o nodo destino da conexão.

O botão *TO* pedirá o nome do rótulo (*label*), acrescentando ao nodo origem um comando *goto* e ao nodo destino um *label*. Esta operação pode ser identificada no quadro 16.

```
origem.AddConnection(destino);    {conecta os dois nodos}
origem.Text.Add('       goto '+rotulo);   {acrescenta linha de comando no nodo origem}
destino.Text.Insert(1,rotulo);     {insere label no nodo destino}
```

Quadro 16 – Adicionando uma conexão com *GOTO*

Após estas operações realizadas, será habilitado os demais botões novamente.

6.1.4 BOTÃO END

Analisará se o nodo selecionado tem algum descendente e, em caso afirmativo, não poderá ser inserido um término neste local. Em caso contrário, acrescentará um nodo representando um fim do programa.

6.1.5 BOTÃO DELETE

Caracterizado por ter um nível de complexidade maior quanto ao código, este botão exclui somente um nodo, mesmo que tenha descendentes, o que não ocorre se for utilizado a tecla *DEL* (teclado). Se o selecionado tiver descendentes, seguirá a mesma idéia da inserção de processos, utilizando uma árvore temporária para não perder os nodos subordinados.

Sua complexidade a nível de código justifica-se pela performance. Se o nodo selecionado não tem sucessores, o exclui direto. Se selecionado é uma condição e os nodos *sim/não* não tem descendentes, também exclui direto. Caso contrário, chamará a *procedure* para não perder os demais nodos.

Além de todas as questões acima, ainda se preocupa com algumas consistências:

a) existe nodo selecionado;
b) selecionado é início, área de variáveis, área de inicialização ou nodos sim/não da condição.

6.1.6 ÁREA DE DEFINIÇÃO DO ZOOM

Esta função concentra-se no evento *onexit* do componente *spinedit* (área para entrada de números relativos ao campo *ZOOM*). A linha de comando para esta finalidade é *tree.ZoomCentered(EdZoom.value).*

6.1.7 BOTÃO VARIABLES

Ativará a tela *Declaration of Variables*, onde estarão as funções para inserir e excluir as variáveis (será descrita mais adiante).

6.1.8 BOTÃO I/O

Ativará a tela I/O, onde estarão as funções para alterar a configuração inicial da porta A e da porta B, configuração realizada pelos registradores TRISA e TRISB, que como já foi visto, determina o estado das portas em entrada ou saída.

6.1.9 BOTÃO ASM FILE

Recursivamente, percorrerá todo o fluxograma armazenando no componente *memo* (campo *Assembly File*) a propriedade *text* dos nodos. Ao final, acrescentará as linhas correspondentes ao término do programa. No quadro 17 observam-se as linhas de código.

```
Procedure AddTreeMemo( Node : TTreeNodeShape );
var
     t, i : Integer;
begin
     for i:=0 to node.Text.Count-1 do
          MmAsm.Lines.Add(Node.Text[i]);
     for t:=0 to Node.Childs.Count-1 do
          AddTreeMemo(Node.Childs[t]);
end;
```

Quadro 17 – Algoritmo recursivo

Uma consistência muito importante será feita nesta ocasião: se existir algum nodo que não tenha seqüência, ou seja, um nodo final e não for o próprio END, mostrará uma mensagem alerta comunicando que poderá ocorrer um erro de lógica, pois os fluxogramas tem que terminar com um nodo referente ao término da execução. Essa pesquisa também será feita recursivamente, indicando quantos nodos estão nessa condição.

6.2 TELA DECLARATION OF VARIABLES

Os botões *ADD* e *DELETE* simplesmente adicionam ou apagam variáveis do componente *listbox* (campo *variables*). Não será permitido adicionar variáveis se já estiverem

armazenadas 36, pois como já foi visto, a área disponível na memória do microcontrolador para este fim vai de 12 a 48 (decimal). O botão *EXIT* reconstruirá o nodo de declaração de variáveis com as mesmas armazenadas no *listbox*.

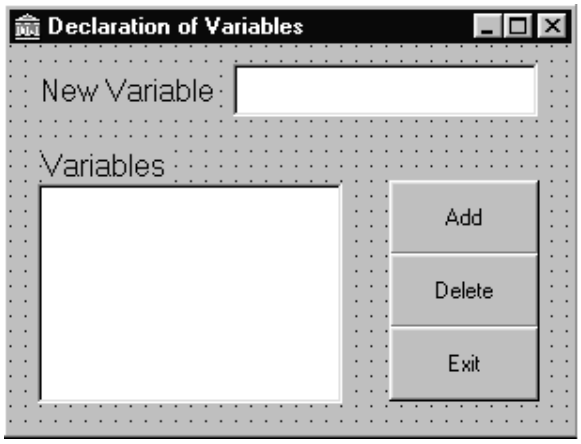


Figura 19 – Tela Declaration of Variables

6.3 TELA: I/O – INPUT/OUTPUT

Nesta tela será definido o estado inicial de todos os bits, tanto do PORT A quanto do PORT B. Inicialmente, todos estarão setados, ou seja, como *input* (entrada=1), mas poderão ser alterados a qualquer momento da criação do fluxograma.

É importante lembrar que esta tela se refere apenas ao estado dos bits no início do programa, ou seja, se for necessário alterar no código após começo do programa, deve-se utilizar um processo atribuindo novos valores à determinada porta.

O botão *OK* se encarrega de transformar os índices dos *RadioGroup* (campos relativos aos bits – *input/output*), em valores hexadecimais, colocá-los no campo *edit* respectivo (campos PORT A e PORT B), e alterar o nodo correspondente ao início do código, onde estão definidos os valores iniciais das portas.

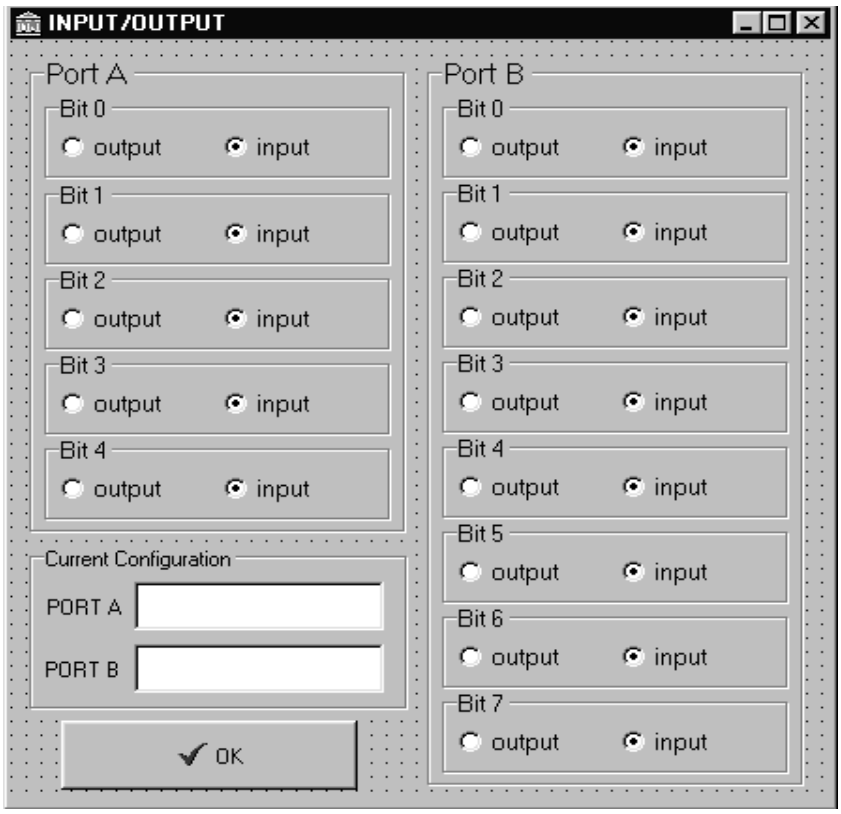


Figura 20 – Tela Input/Output

6.4 TELA NEW PROCESS

Tela com alto nível de complexidade com relação ao código, pois nela começam a serem feitas as primeiras consistências referentes ao montador. Tanto esta quanto a tela *Condition* tem as operações e condições pré-definidas, assim como os operadores e valores, tornando o erro mínimo, pois tudo que está disponível na tela pode ser usado sem problemas na montagem.

O botão "*to publish command line"* será o responsável por editar a linha de comando, bem como montar a ou as linhas de código em *assembly*. Se algum dado (campo) necessário para esta operação não estiver preenchido, o processo não será concretizado, caracterizando uma das consistências nessa tela.

Os botões *OK* e *Cancel* simplesmente passam parâmetros identificando se a operação deve ser concluída ou não.

As variáveis contidas nos espaços *variables1* e *variables2* são buscadas no *form* (tela) referente à declaração de variáveis (tela: *declaration of variables*). Por estarem incluídas pela cláusula "include <P16C84.INC>" contida no nodo referente a raiz do fluxograma e início do código *assembly*, as variáveis PORT A e PORT B também estarão presentes nessa área, podendo ser usadas (alteradas) em qualquer ponto do programa.

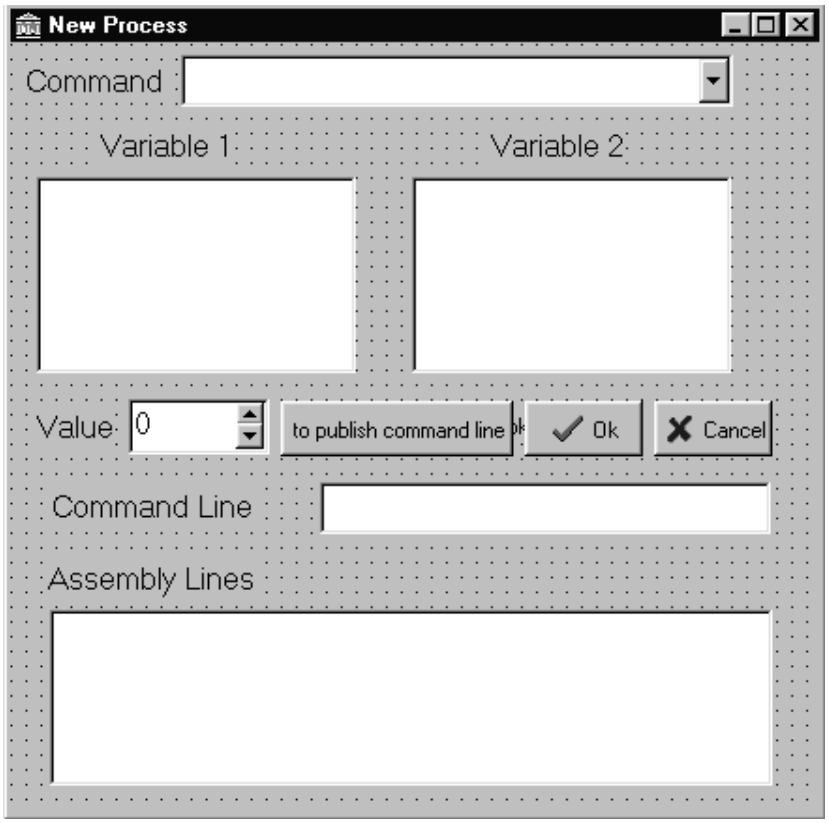


Figura 21 – Tela New Process

No *ComboBox* (campo *command*) estarão disponíveis as operações suportadas por este protótipo. Estas e suas respectivas linhas de código geradas estão representadas na tabela 5.

Comando	**Linhas *assembly***
Increment var	Incf variables1,1
Decrement var	Decf variables2,1
Move var to var	Movf variables1,w Movwf variables2

Move lit to var	Movlw value Movwf variables1
Clear bit	Bcf variables1,value
Set bit	Bsf variables1,value

Tabela 5 – Código gerado pelo processo

Dúvidas sobre a sintaxe e funcionamento dos comandos *assembly* podem ser esclarecidas no capítulo 3.

É importante lembrar que nas operações que manipulam os bits de variáveis, quando escolhido o PORT A, estarão disponíveis os bits de 0 a 4, pois nesta porta existem apenas cinco pinos configurados como linhas de entrada e saída.

6.5 TELA: CONDITION

A base do funcionamento nessa tela segue a lógica da tela de inserção de processos. As principais diferenças são:

a) o campo do operando da condição (=, >, <, <>) e;
b) operações disponíveis no protótipo.

O item selecionado no campo do operando é mais uma chave de pesquisa para montar as linhas em *assembly*. Esta afirmação e as novas operações podem ser identificadas na tabela 6.

Condição	**Linhas *Assembly***

To compare var/var	Movf variables2,0
	Subwf variables1,0
	* verifica operador

To compare var/lit	Movlw value Subwf variables1,0 * verifica operador
Bit var = set	Btfss variables1,value
Bit var = clear	Btfsc variables1,value

* verifica operador: a) se operador for sinal de igual (=) : comando → btfss status,2 b) se operador for sinal de maior (>) : comando → btfsc status,0 c) se operador for sinal de menor (<) : comando → btfss status,0 d) se operador for sinal de diferente (<>) : comando → btfsc status,2

Tabela 6 – Código gerado pela condição

Status se refere ao registro *STATUS*. Este registro possui oito bits (0-7), sendo que o bit 2 corresponde ao bit Z e o bit 0 corresponde ao bit C. Quando o bit 0 está aceso significa que w < f, caso contrário w > f. Se o bit 2 estiver aceso significa que w = f, senão w <> f.

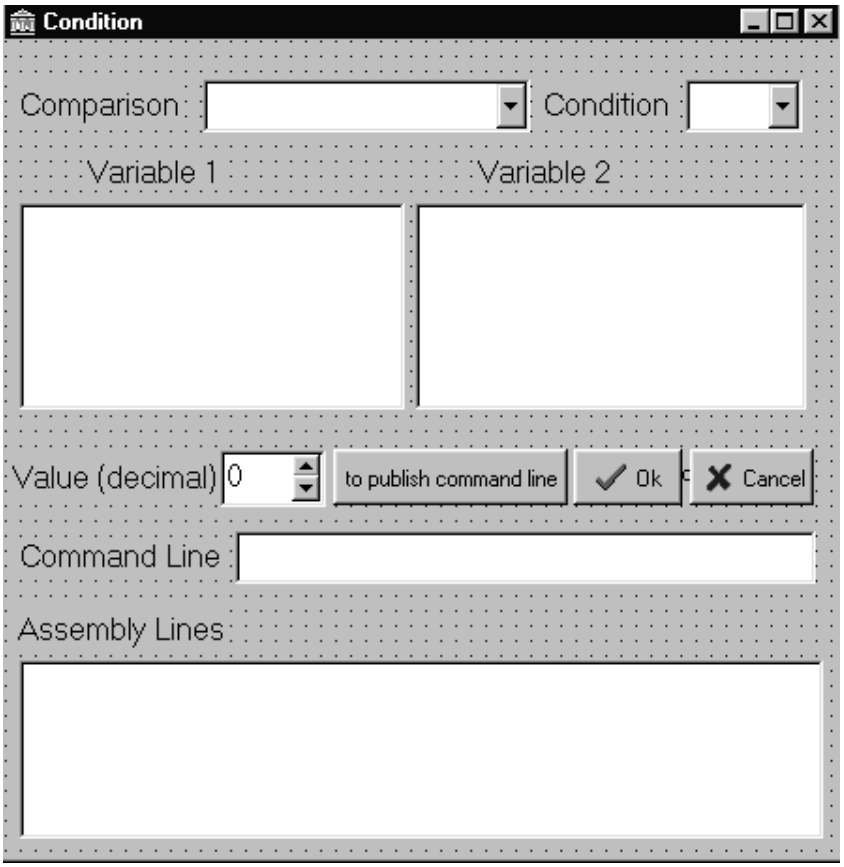


Figura 22 – Tela Condition

6.6 TESTES

6.6.1 CIRCUITO BÁSICO PARA TESTES

Para a realização dos testes foi montado um circuito básico. Este pode ser identificado na figura 23.

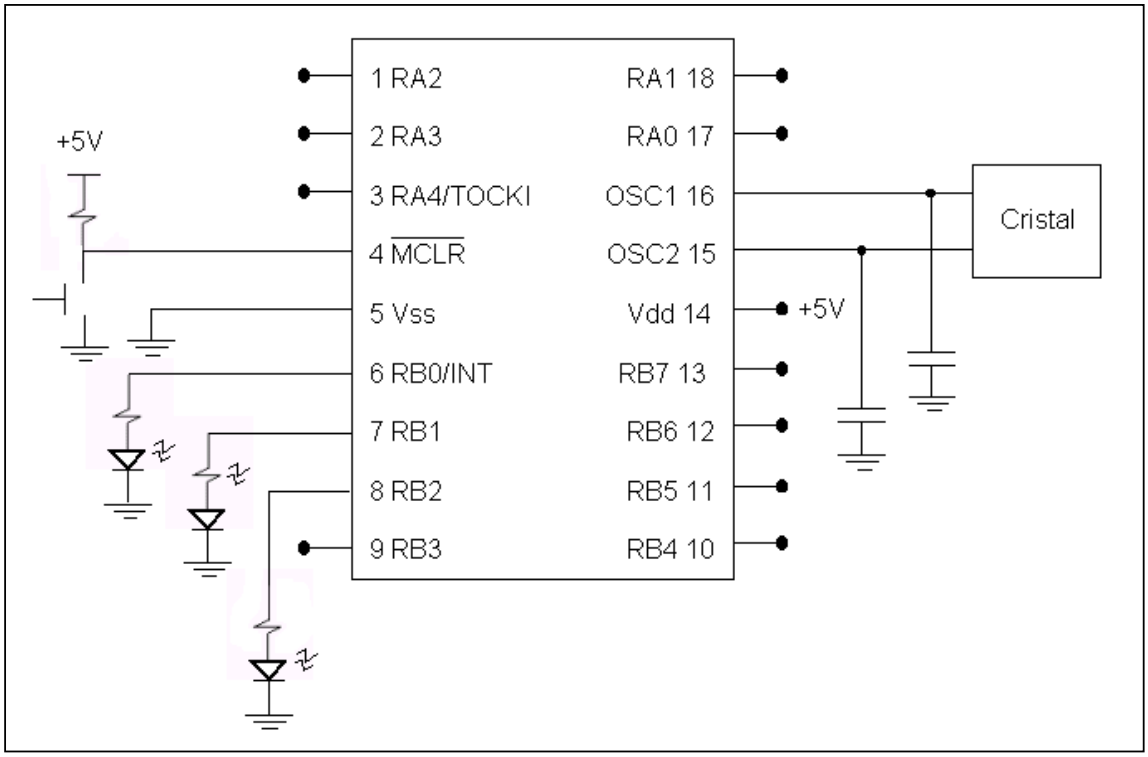


Figura 23 Circuito básico para os testes

O pino Vdd (alimentação, pino 14) recebe uma tensão de 5V. O pino Vss (terra, pino 5) é o retorno da corrente que circula dentro do circuito. Existem dois capacitores conectados em uma extremidade aos pinos OSC1 (pino16) e OSC2 (pino 15), e a outra a um cristal (*Clock* Oscillator). Esses capacitores serão os responsáveis pela oscilação do cristal. Nos pinos RB0, RB1 e RB2 estão ligados três *LED*'s, os quais no momento da execução dos testes serão utilizados para visualizar os resultados.

O pino MCLR (pino 4) estará recebendo uma corrente de 5V e, no momento que a chave for fechada, a corrente cairá a zero caracterizando o *reset* do circuito.

6.6.2 TESTE COMPARATIVO

Um dos testes realizados foi a comparação do código retirado de um exemplo do livro [BEN96] com o código gerado no protótipo. No exemplo do livro há um fluxograma (figura 24) demonstrando o funcionamento do código e, a partir desse, foi construído o fluxograma no protótipo, e posteriormente, gerado o código. Na figura 25 observa-se o fluxograma criado no protótipo.

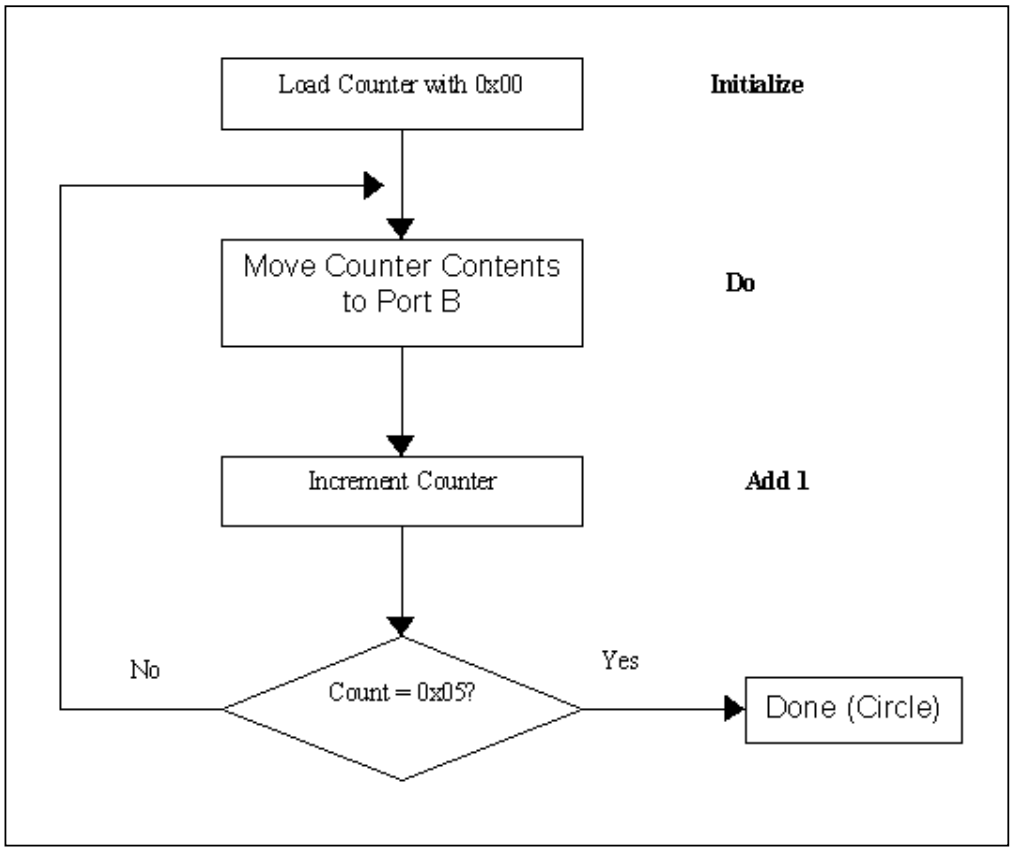


Figura 24 – Fluxograma teste retirado do livro [BEN96]

No quadro 18 é mostrado os dois códigos sem comentários para melhor comparação.

Código exemplo do livro [BEN96]			**Código gerado no protótipo**	
	list	p=16c84	List	p=16c84
	radix	hex	Radix	dec
w	equ	0	Include <P16F84.INC>	

```
f       equ     1
status  equ     0x03
portb   equ     0x06
count   equ     0x0c
        org     0x000
start   movlw   0x00
        tris    portb
        clrf    portb
        clrf    count
again   movf    count,w
        movwf   portb
        incf    count,f
        movlw   0x05
        subwf   count,w
        btfss   status,2
        goto    again
circle  goto    circle
        end
```

```
counter equ     d'12'
        org     0x000
        movlw   0x1F
        tris    PORTA
        movlw   0x00
        tris    PORTB
again
        movlw   0
        movwf   counter
        movf    counter,w
        movwf   PORTB
        incf    counter,1
        movlw   5
        subwf   counter,0
        btfss   STATUS,2
        goto    n1
        goto    y1
n1
        goto    again
y1
        goto    fim
fim     goto    fim
        end
```

Quadro 18 – Comparação de códigos testados

Observa-se com isso que o protótipo gera linhas de código corretamente em relação ao que foi descrito no fluxograma. Para leigos e iniciantes na programação do PIC, ele se torna um grande aliado no aprendizado, que, além de mostrar a lógica de uma forma visualmente agradável, pode-se comparar em momento de execução o comando escolhido e sua(s) respectiva(s) linha(s) de comando em *assembly*. Já para programadores experientes, apesar das vantagens visuais, não é muito objetivo nos seus comandos. Esse problema surgiu para amenizar outros mais graves. Um exemplo prático é o que ocorre nas condições onde sempre terão instruções *goto* para seqüência tanto nas condições verdadeiras como nas falsas.

Programadores experientes sabem que no *assembly* do PIC, uma instrução que representa uma condição (*btfss, btfsc,* etc.) saltará uma linha se a esta for falsa, o que não é sabido por muitos iniciantes. Para evitar esses enganos e pela própria lógica dos fluxogramas, esses *goto*'s "mascaram" a admissão de mais de uma instrução no caso da condição ser falsa. É importante ressaltar também que, no caso de necessitar que uma porta aceite um dado como entrada ou saída, o próprio programador deve atribuir os devidos valores para os bits através do registrador TRIS.

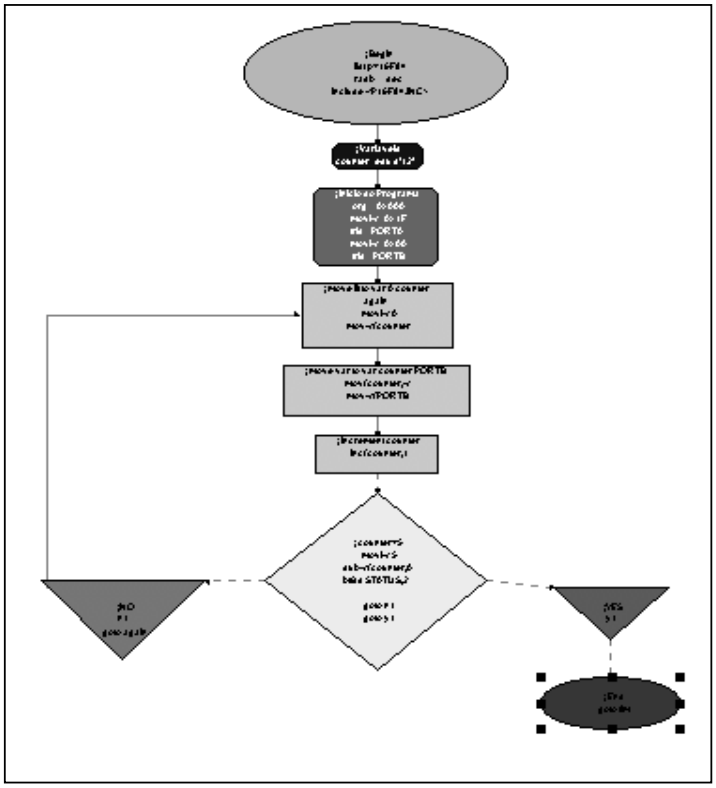


Figura 25 – Fluxograma teste criado no protótipo

Após gerado o arquivo com extensão .ASM (arquivo texto), este foi montado utilizando-se o MPASMWIN (montador do PIC, figura 26), transformando em formato hexadecimal com extensão .HEX. A partir deste momento já é possível gravar o programa na memória do microcontrolador através do PIP02, programa *freeware* utilizado neste trabalho para acionar o programador (figura 27). Depois destas fases concluídas, resetou-se o PIC para observar sua operação, mas não foi possível verificar o início do funcionamento dos *LED*'s

devido a alta velocidade de execução, mas pode-se observar o último valor escrito no PORT B. Em um segundo teste, acendia-se seqüencialmente os três *led*'s do circuito de teste. Para comprovar se o resultado estava correto foi utilizado um osciloscópio, aparelho que permitiu a visualização das oscilações da corrente. Quando a corrente está presente (+5V, no caso) acende o *LED*, e quando não está (0V), apaga.

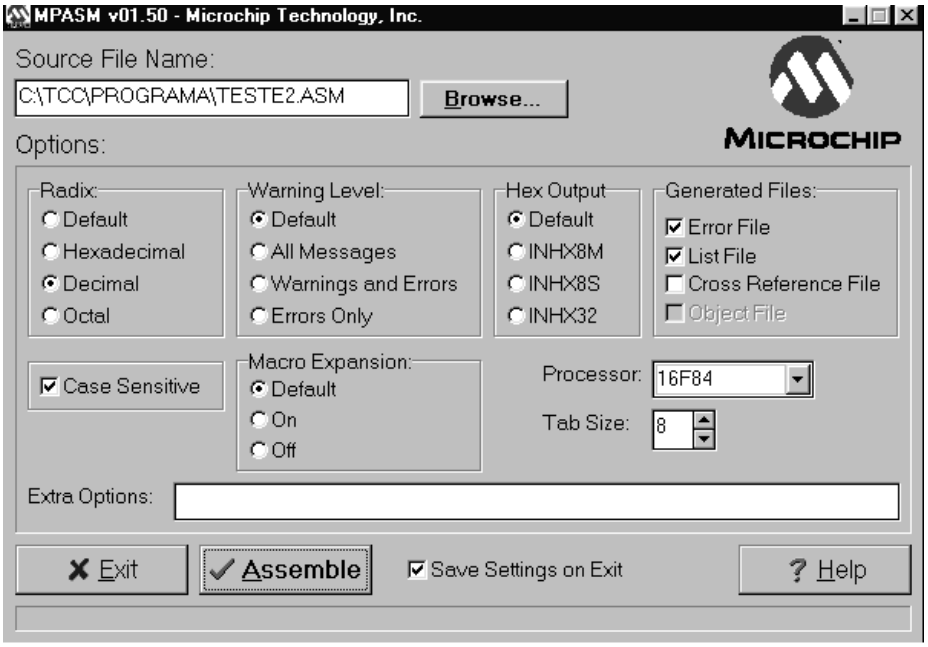


Figura 26 – Tela MPASMWIN

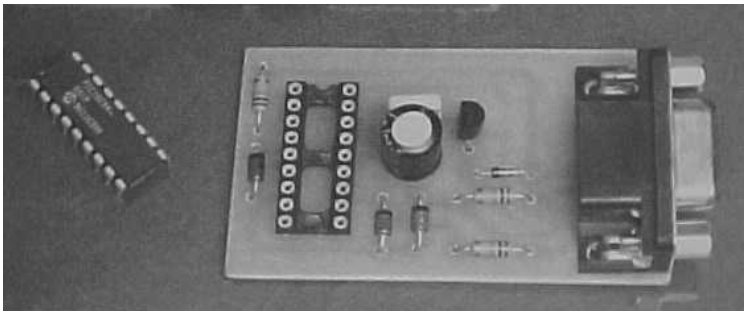


Figura 27 – Programador utilizado com o PIP02

7 CONCLUSÃO

Cada vez mais estão sendo disponibilizados componentes sofisticados a um custo baixo tanto para produção como para desenvolvimento e também sua utilização está ficando cada vez mais fácil e acessível, tornando-se uma tendência mundial.

Durante o desenvolvimento do trabalho, foi colocada na internet a idéia que estava sendo implementada, e como já era previsto, teve uma ótima aceitação e demonstração de interesse. Na primeira semana foram recebidas cerca de 70 respostas de interessados, o que reforça ainda mais a real necessidade de uma implementação nesse nível, contendo uma forma visual de representação da informação para o microcontrolador. Algumas das respostas recebidas podem ser vistas no anexo 1.

Após os testes feitos no protótipo notou-se que, a construção de um programa torna-se muito mais agradável e fácil de ser feita na forma visual, ou melhor, usando formas e cores que identificam melhor as operações.

Outras características importantes que torna uma implementação dessa proporção bem aceita ao público alvo, é o fato de ter um tamanho reduzido e baixo consumo de memória, características herdadas do *Ttree*.

A implementação mostrou-se muito amigável, principalmente para iniciantes com pouco ou nenhum conhecimento sobre o microcontrolador, bem como a sua linguagem *assembly,* pois tendo as operações pré-definidas no protótipo, sabendo-se o que é preciso para fazer a aplicação gerar o resultado desejado (lógica), basta escolher as opções também pré-definidas e assim, o próprio protótipo se encarregará de montar o código sem erros, pois no momento das escolhas já são feitas as consistências. Como ainda não é uma versão comercial, faltam ser implementadas algumas funções e melhoradas outras para se ter um aplicativo ideal.

Também foi possível observar que o conjunto de instruções reduzidas implementado nesse microcontrolador facilita o aprendizado e a construção de um protótipo para esta finalidade, podendo a partir desta implementação inicial, migrar para outros tipos de microcontroladores com certo nível de facilidade.

O objeto *Ttree* auxiliou em muito, proporcionando várias facilidades, mas ainda faltam algumas funções para se tornar um aplicativo ideal para este fim. Futuramente, para acréscimos de funções no protótipo, deverá ser estudado e entendido melhor o funcionamento do objeto *Ttree*, sendo que além de algumas funções existentes que ainda não foram aproveitadas no seu máximo, estão por serem implementadas novas características que facilitarão o melhoramento do projeto, segundo os desenvolvedores do aplicativo.

Os testes mostraram que o protótipo auxilia muito o trabalho de iniciantes, pois cria programas para o PIC de uma forma agradável e ao mesmo tempo os ensina mostrando o comentário e logo a seguir a linha de comando respectiva. Esta conclusão foi confirmada com as respostas de alguns *beta testers* que deram suas opiniões, sugestões de melhoras e de acréscimos, ou seja, idéias que contribuem para continuação do trabalho.

ANEXO 1 - TESTES DE USUÁRIOS FINAIS

Seyler Jean-Ives [JEA99]:

“Trabalho bom,

Como um iniciante no mundo do PIC, planejo usar seu trabalho para aplicações simples. Embora o usei por poucos minutos, tenho perguntas e observações:

* Há algumas palavras em português;

* Poderia ser conveniente imprimir o fluxograma;

* No gerenciamento de arquivos, deveria poder salvar sem ter que rescrever o nome ("Salvar ou Salvar Como ");

* As setas para a escolha de condição também poderiam ser movidos manualmente, porque às vezes elas não são prendidas no lugar certo;

* Como posso apagar uma label previamente colocado?

* Às vezes acontece uma " violação de Acesso quando lendo endereço 0A8 "

* Às vezes, alguns nodos desaparecem!!!

De qualquer maneira é um trabalho bom! Espero que você continue melhorando isto!”

Chaipi Wijnbergen [WIJ99]:

“Baixei o programa do fluxograma e rodei. No geral, gostei do que você fez.

Sei que alguns programadores têm dificuldades em entender qual é a sintaxe correta:

se (var1> var2)... ou se (var1 <var2). Acho que sua ferramenta é muito boa para tais pessoas.

Não sou nenhum perito nos 84, mas penso que você está perdendo alguns periféricos?! Assim, acho que você tem um começo muito agradável. Sugeriria que você tente somar os periféricos, e talvez se expande além os 84.

Também seria bom se você mudasse o texto nos títulos das janelas de português para inglês (como Novo Processo e na declaração de variáveis).

Se você tenta fechar a janela de declaração variável sem digitar um nome de variável, ocorre uma mensagem: "erro de violação de acesso"”

James Newton [NEW99]:

“Bom:

Realmente ele torna óbvio como são formados os comandos na linguagem *assembly*. Não posso imaginar uma ferramenta pedagógica melhor. É excelente. Ao começar um novo documento, como ele é mostrado, permite ao usuário entender o que o programa faz sem manuais ou instruções.

Ruim:

Eu fiquei um pouco confuso por um momento porque os novos nodos não apareciam depois de serem clicados duas vezes. Variáveis adicionais e declarações precisam de ser somadas.

Sugestões

O grid para os objetos deveriam ser ajustados de forma que as linhas entre eles permanecessem retas.

Dimensionamento e a colocação automáticos dos objetos como uma opção seria bom.

Resumo

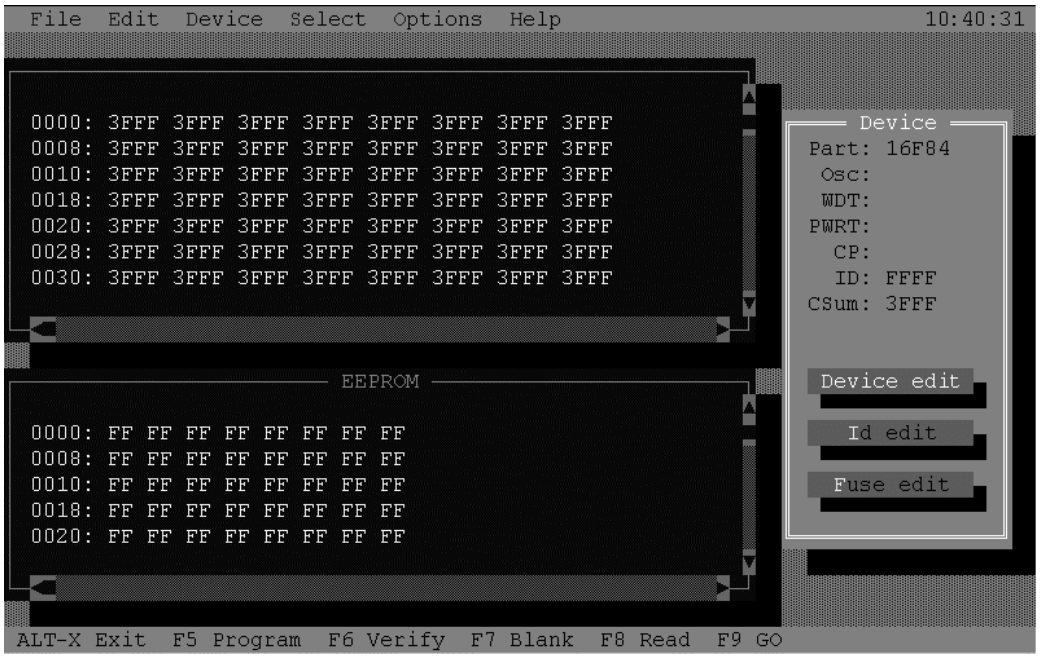
Não estou tão certo quanto útil seria a um programador profissional, mas como uma ferramenta de treinamento para pessoas novatas em microcontroladores ou em um ambiente pedagógico, por enquanto não posso imaginar nada melhor.

Pergunta:

Qual é a licença nesta aplicação? Estará disponível no futuro?”

ANEXO 2 – PIP02

Na figura abaixo pode-se observar a tel ado PIP02 e em seguida uma breve descrição dos comandos mais utilizados para realização dos testes, ou seja, a gravação no PIC.



FILE/LOAD: carregar arquivo a ser gravado no PIC;

SELECT/DEVICE: selecionar o dispositivo (tipo do PIC);

SELECT/ FUSE WORD: determinar o tipo do cristal;

DEVICE/PROGRAM FUSES: programar fusíveis;

DEVICE/PROGRAM: programação do PIC.

8 REFERÊNCIAS BIBLIOGRÁFICAS

[BEN79] BENTES, Roberto de fino. **Processamento de dados**. Curitiba : Editer, 1979.

[BEN96] BENSON, David. **Easy PIC'n**. Kelseyville : Square 1, 1996.

[BER96] BERNARDES, Luís Henrique Corrêa. **Microcontroladores PIC**. Saber Eletrônica. n. 284, p. 4-14, 1996.

[BER97] BERNARDES, Luís Henrique Corrêa. **Microcontroladores PIC**. Saber Eletrônica. n. 292, p. 14-20, 1997.

[GAL99] GALIZIA, Tiziano. **Curso PIC16C84** 1999. Endereço Eletrônico : http://www.picpoint.com/.

[HOR91] HORTON, William. **Illustrating computer documentation: the art of presenting information graphically on paper**. New York : Wisley, 1991.

[JEA99] JEAN-IVES. Seyler. **Avaliação do Protótipo**. Endereço Eletrônico : jean-ives.seyler@cnes.fr. França, 1999.

[NEW99] NEWTON, James. **Avaliação do Protótipo**. Endereço Eletrônico : eplus1@san.rr.com, 1999.

[PIC95] CURSO PIC. **Elector Eletrônica**, n. 122, p. 19-22, fev. 1995.

[SIL97] SILVA JÚNIOR, Vidal Pereira da. **Microcontroladores PIC: Teoria e Prática.** São Paulo : V. P. Silva Júnior, 1997.

[TEE98] TEETREE1.DOC. **Características, propriedades e exemplos do componente TeeTree**. David Bemeda. abr. 1998. http://www.teemach.com. Word 97.

[UNI99] UNICAMP. **Onagro** mar. 1999. Endereço eletrônico : http://www.demic.fee.unicamp.br/onagro.html.

[WIJ99] WIJNBERGEN, Chaipi. **Avaliação do Protótipo**. Endereço Eletrônico : chaipi@tohu0.weizmann.ac.il. 1999.

[WIR86] WIRTH, Niklaus. **Algorithms and Data Structures**. London : Prentice-Hall, 1986.