

# Projeto de Sistemas Microcontrolados

## Aula 05 - Linguagem Assembly e ambientes de programação

# Apresentação

Nesta aula, será introduzido o estudo da linguagem de programação *Assembly* e mostrado o processo de criação de um programa em *Assembly* até sua gravação em um microcontrolador PIC, usando-se o ambiente de desenvolvimento MPLAB. No estudo da linguagem *Assembly*, serão vistos o conjunto de instruções dos PIC 16F, o formato das instruções e a sintaxe usada em sua escrita. Será mostrado também um modelo simplificado para construção de programas em *Assembly*, aplicado no desenvolvimento de um semáforo de dois tempos e apresentado o ambiente de desenvolvimento Proteus, em especial sua ferramenta ISIS, que permite simular o comportamento de um circuito elétrico baseado em microcontroladores.

## Objetivos

Ao final desta aula você será capaz de:

- Descrever o conjunto de instruções dos microcontroladores PIC e das diretivas do *assembler* (montador) que lhes permite desenvolver programas em linguagem *Assembly*.
- Criar um programa simples, baseado no funcionamento de um semáforo de dois tempos usando os ambientes de desenvolvimento Proteus e MPLAB.
- Entender o uso da ferramenta de desenvolvimento MPLAB num processo que vai da criação e edição de um projeto até sua gravação em um PIC.

# Linguagens usadas na programação de microcontroladores

Os microcontroladores eram, tradicionalmente, programados em linguagem *Assembly*. De certa forma isso era compreensível considerando-se dois aspectos: primeiro, pelo fato de cada família de microcontroladores apresentar um conjunto muito específico de instruções, que se traduz diretamente em código de máquina; segundo, pelo fato dos microcontroladores, até anos recentes (2006 mais ou menos), apresentarem uma área de memória de programa muito reduzida (lembre-se, por exemplo, que o 16F84 só possui 1 K x 14 *bits* reservado para armazenamento de programas).

À medida que a capacidade da memória de programa dos microcontroladores foi sendo aumentada (o 16F877, por exemplo, já apresenta 8 k x 14 *bits*), foi ocorrendo uma migração natural para a linguagem C, e inclusive alguns programadores já excluem a necessidade de um estudo detalhado do microcontrolador e do *Assembly* e já iniciam suas atividades diretamente em linguagem C, mais simples e mais efetiva. No entanto, é sempre oportuno iniciar o estudo de microcontroladores pelo conhecimento (embora não muito detalhado) do *Assembly* e só então migrar, o mais suavemente possível, para a linguagem C.

Ressalta-se que a própria empresa Microchip já vem desenvolvendo famílias, como as 18F e 24F com compiladores C nativos, embora a linguagem C também possa ser usada nas séries PIC menos complexas. No entanto, as aplicações devem ser relativamente simples e não podem demandar além das limitadas capacidades de memória de programa desses processadores.

Neste texto, serão apresentados exclusivamente a linguagem *Assembly* e um modelo simplificado que poderá ser adotado como padrão para desenvolvimento de seus programas. Antes, porém, torna-se necessário conhecer os comandos que compõem o conjunto de instruções dos PIC, bem como será de grande valia conhecer quais formatos são usados em suas construções.

Após este estudo, nos direcionaremos para o uso da linguagem C nas Aulas 6 e 7. Na oportunidade, também será usado um modelo simplificado, de forma a associar, tanto quanto possível, cada linha escrita em C com o seu equivalente em programação *Assembly*.

## Atividade 01

- 1. Faça uma pesquisa na internet sobre a viabilidade de uso das linguagens *Assembly* e C para microcontroladores. Veja os prós e os contras de cada uma e formule sua opinião ou elabore questionamentos que possam orientar sua opinião sobre o tema.**

## Declarações em um programa-fonte escrito em *Assembly*

As declarações de um programa-fonte escrito em *Assembly* (o que corresponde a cada linha de entrada) podem ser do tipo:

- instruções *Assembly*;
- diretivas do *assembler* (montador).

As instruções *Assembly* indicam as ordens que devem ser executadas pelo microprocessador ou microcontrolador e são transcrições (ou notações) simplificadas que correspondem aos códigos binários das instruções de máquina.

As diretivas do *assembler* são comandos especiais com o objetivo de facilitar a escrita de um programa na forma simbólica. Não são incorporadas ao programa-objeto e servem simplesmente como orientação para o montador.

# Sintaxe das instruções do *Assembly*

Para a grande maioria dos montadores, cada instrução pode ter até quatro campos delimitados de acordo com a seguinte ordem:

`[label] mnemônico [operando(s)] [;comentário]`

Destes, apenas o campo mnemônico é sempre obrigatório, o(s) operando(s) depende(m) da instrução incluída na linha, e os campos de etiqueta (*label*) e comentário são sempre opcionais.

## Sintaxe das diretivas do *assembler*

`[label] diretiva [operando(s)] [;comentário]`

Dos campos acima colocados entre colchetes, apenas o campo de comentário é sempre opcional. Os campos *label* e operando(s) dependem da diretiva empregada. Podendo um ou outro inclusive ser proibitivo.

Pode ocorrer o uso de comentários de linha, bastando para tanto iniciar a linha com um ";" (ponto e vírgula).

## Diretivas do *assembler*

Algumas diretivas do *assembler* são muito utilizadas e têm o intuito de ajudar na documentação do programa, tornando-o claro para quem o estudar.

Dois exemplos de diretivas:

### **EQU**

- Usado para atribuir uma expressão (numérica ou não) a um símbolo. Sempre que esse símbolo aparecer no programa, o montador o substituirá pela expressão que lhe foi associada.

Sintaxe:

*label* EQU expressão

Ex:

Constante1 EQU .80

...

MOVLW Constante1 ; move para o registrador W o valor 80 decimal

## DB

- Usada para a definição de variáveis de tamanho de byte.

Sintaxe:

*label* diretiva operando

Ex.:

Variavel1 DB .12

Variavel2 DB 0

...

MOVF Variavel1,W ;move para W o valor armazenado em Variavel1

MOVWF Variavel2 ;move para Variavel2 o valor 12 decimal.

Uma amostra das diretivas mais frequentemente utilizadas em um programa-fonte *Assembly* é feita na apresentação do modelo simplificado, que será visto logo mais, ainda nesta aula. Uma relação completa, com exemplos, de todas as diretivas do *assembler* pode ser encontrada no help do MPLAB IDE.

## Conjunto de instruções

Após ter visto as principais características dos microcontroladores 16F84, 16F628 e 16F877, vamos agora conhecer o seu conjunto de instruções para programação em *Assembly*. Lembre-se que quando se programa em C, o compilador C é responsável por converter as instruções da linguagem C, de mais alto nível, em certo número de comandos da linguagem *Assembly*, de baixo nível e diretamente relacionados à arquitetura do microcontrolador.

Todos os PIC da família [16F](#) (Este número aumenta consideravelmente na família 18F (são mais de 80 instruções), no entanto, as dos PIC 16F são comuns às duas famílias, de modo que um programa escrito para um 16F pode ser executado num

18F sem muitas alterações.) apresentam um conjunto de 35 instruções separadas em quatro grupos:

- instruções de operações com registros;
- instruções de operações com *bits*;
- instruções de operações com literais;
- instruções de controle.

Para essa família, cada instrução é uma palavra de 14 *bits*, dividida em um *OPCODE* (código de operação) que especifica o tipo de instrução e um ou mais argumentos que especificam o(s) operando(s).

Todas as instruções são executadas dentro de um único ciclo de instrução, a menos que um teste condicional seja verdadeiro ou o contador de programa seja mudado como resultado de uma instrução de desvio (neste caso, a execução leva dois ciclos de instrução). Cada ciclo de instrução consiste em quatro períodos de *clock*. Desse modo, para uma frequência do oscilador de 4 MHz, o tempo de execução de uma instrução normal será de 1  $\mu$ s. Em caso de ser uma instrução de desvio, para a mesma frequência de clock de 4 MHz, o tempo de execução da instrução será de 2  $\mu$ s.

## Formato das instruções

Nem todas as instruções de um microcontrolador PIC apresentam um mesmo padrão de execução. Essa diversidade de padrões de execução é caracterizada pela codificação binária que é dada para cada uma delas. Essas codificações binárias são distribuídas em quatro formatos típicos, caracterizando os quatro grupos de instruções citadas no início desta seção.

Código de operação						d	f						
13	12	11	10	9	8	7	6	5	4	3	2	1	0

**Formato 1** – Instruções de operação com registros

Nesse formato, "f" representa uma nomeação de um registro, uma posição de memória ou uma variável, "d" representa uma nomeação de destino. Se "d" é 0, o resultado é colocado no registrador de trabalho W, e se "d" é 1, o resultado é armazenado no próprio registrador, na posição de memória ou na variável referenciada na instrução.

Exemplos de instruções que se utilizam desse formato:

- ADDWF PORTA, F ; Adicione W com o valor presente em PORTA e

;armazene resultado em PORTA.

- ADDWF PORTA, W ; Adicione W com valor em PORTA e armazene

;resultado em W.

Código de operação				b			f						
13	12	11	10	9	8	7	6	5	4	3	2	1	0

**Formato 2** – Instruções de operação com *bits*

Nesse formato, "b" representa a denominação do *bit* afetado pela operação, enquanto "f" representa um endereço de registro ou variável no qual o *bit* está localizado.

Exemplo de uma instrução que se utiliza desse formato:



- BSF PORTA, RA1 ; Set *bit* 1 (RA1) do registrador PORTA

Código de operação						k (literal)							
13	12	11	10	9	8	7	6	5	4	3	2	1	0

**Formato 3** – Instruções de operação com literais de 8 *bits*

Exemplo de instruções que se utilizam desse formato:

- MOVLW .53 ; Armazene em W o literal 53 decimal.
- ADDLW 0x49 ; Adicione valor em W com o literal 49 hexadecimal.

Código de operação			k (literal)										
13	12	11	10	9	8	7	6	5	4	3	2	1	0

**Formato 4** – Instruções de operação com literais de 11 *bits*

Exemplo de uma instrução que se utiliza desse formato:

- GOTO 0x320 ; Desvie programa para posição hexadecimal 320

Um quadro-resumo do conjunto de instruções dos microcontroladores PIC 16F, separadas por grupo de instruções aritméticas, lógicas, de movimentação, de desvio, de tratamento com *bits* e de controle é apresentado no Quadro 1 e uma análise completa do conjunto de instruções PIC 16, com exemplos, pode ser acessada em:

<<http://www.cinelformacao.com/mpics/files/ud3/instr/index.htm>>.

Instrução	Descrição da operação	Nº de ciclos	Flags afetados
Aritméticas			
ADDLW k	$W = W + k$	1	C, DC, Z

Instrução	Descrição da operação	Nº de ciclos	Flags afetados
ADDWF f, d	$d = W + f$	1	C, DC, Z
SUBLW k	$W = k - W$	1	C, DC, Z
SUBWF f, d	$d = f - W$	1	C, DC, Z
INCF f, d	$d = f + 1$	1	Z
DECF f, d	$d = f - 1$	1	Z
CLRF f	$f = 0$	1	Z
CLRW	$W = 0$	1	Z
Movimentação			
MOVLW k	$W = k$	1	-
MOVWF f	$f = W$	1	-
MOVF f,d	$d = f$	1	Z
RLF f, d	Rotaciona f à esquerda através do carry	1	C
RRF f, d	Rotaciona f à direita através do carry	1	C
Lógicas			
ANDLW k	$W = W.\text{and. } k$	1	Z

Instrução	Descrição da operação	Nº de ciclos	Flags afetados
ANDWF f, d	d = W.and.f	1	Z
IORLW k	W = W.or.k	1	Z
IORWF f, d	d = W.or. f	1	Z
XORLW k	W = W.xor.k	1	Z
XORWF f, d	d = W.xor.f	1	Z
COMF f,d	d = .not.f	1	Z
SWAPF f, d	d = f<3:0> f<7:4>	1	-
Operações com <i>bit</i>			
BCF f, b	Clear <i>bit</i> b de f	1	-
BSF f, b	Set <i>bit</i> b de f	1	-
BTFSS f,b	salta próxima instrução se Bit b de f = 1	1 (2)	-
BTFSC f, b	salta próxima instrução se Bit b de f = 0	1 (2)	-
INCFSZ f, d	d = f+1 e salta próxima instrução se d=0	1 (2)	-
DECFSZ f, d	d = f-1 e salta próxima instrução se d=0	1 (2)	-

Instrução	Descrição da operação	Nº de ciclos	Flags afetados
Desvio			
GOTO k	Desvia execução do programa para k	2	-
CALL k	Chama sub-rotina no endereço k	2	-
RETURN	Retorna de sub-rotina	2	-
RETLW k	Retorna de sub-rotina fazendo W = k	2	-
RETFIE	Retorna de interrupção	2	-
Controle			
NOP	Não faça nada	1	-
CLRWDT	WDT = 0 e WDTPrescaler = 0	1	/TO=1, /PD=0
SLEEP	Clock OFF, WDT = 0 e WDTPrescaler = 0	1	/TO=1, /PD=1

**Quadro 1** – Listagem com as instruções dos microcontroladores PIC 16

## Atividade 02

1. Observe os formatos apresentados para cada grupo de instruções e responda: Por que, no formato das instruções de operação com registros, a quantidade de *bits* que representa uma nomeação de registro é de apenas 7 *bits*? Por que, no formato das instruções de

**operação com *bits*, a quantidade de *bits* que representa a escolha do *bit* é de apenas 3 *bits*?**

- 2. Busque no site da Microchip <[www.microchip.com](http://www.microchip.com)> o *datasheet* de algum microcontrolador da família 18F (o 18F4520, por exemplo) e verifique se os formatos do seu conjunto de instruções são os mesmos apresentados para a família 16F. Destaque as diferenças, caso existam.**

## Modelo simplificado para construção de programas em *Assembly*

No Quadro 2, é apresentado um *template* simplificado para construção de programas em linguagem *Assembly*. À esquerda, foram colocadas numerações de linhas apenas para melhor descrever os conteúdos colocados no modelo, mas que logicamente não fazem parte dele.

```

01 ;Modelo (template) simples para construção de programas Assembly
02 ;-----
03 ; Nome do programa (código fonte):
04 ; Autor:
05 ; Versão:                               Data:
06 ; Breve descrição do programa:
07 ;
08 ;-----
09 ; Inclusão de arquivo de definições do processador
10 #INCLUDE <p16F877.inc>
11 ;-----
12 ; Configurações de bits para gravação no PIC
13 ; _CONFIG_CP_OFF & _CPD_OFF & _LVP_OFF & _WDT_ON & _XT_OSC
14 ;-----
15 ; área reservada para definições extras a serem usadas no programa
16 #DEFINE      banco1      BSF    STATUS,RP0
17 #DEFINE      banco0      BCF    STATUS,RP0
18 #DEFINE      chave0      PORTB,0
19 #DEFINE      led0        PORTA,5
20 ;-----
21 ; Área reservada para definição de variáveis do programa
22      CBLOCK 0x20          ; início de bloco de variáveis
23          VarInt8
24          VarInt16:2
25 ;          ...          linhas para inclusão de outras variáveis
26      ENDC                ; final de bloco de variáveis
27 ;-----
28 ; Área reservada para escrita de sub-rotinas ou funções
29 ;-----
30 SubRotina1
31 ;
32 ;     Instruções da SubRotina1
33 ;
34      RETURN              ; instrução de retorno de sub-rotina
35 SubRotina2
36 ;
37 ;     Instruções da SubRotina2
38 ;
39      RETLW .10           ; instrução de retorno de sub-rotina
40 ;-----
41 ; Início do programa
42      ORG 0x00            ; endereço de reset do microprocessador
43      GOTO Principal      ; desvia para rotina principal
44 ;-----
45 ; área para tratamento de interrupções
46      ORG 0x04            ; endereço de desvio das interrupções
47 ;
48 ;     Instruções da rotina de tratamento de interrupção
49 ;
50      RETFIE              ; instrução de retorno de interrupção
51 ;-----
52 ; Rotina ou procedimento principal
53 Principal
54 ;
55 ;     Instruções do rotina ou procedimento principal
56 ;
57 fim      GOTO fim        ; comando de loop infinito, se necessário
58      END                ; finalização do programa Assembly

```

**Quadro 2** – Template simplificado para construção de programas em linguagem Assembly

No modelo apresentado, em azul encontram-se apenas as observações, em vermelho, exemplos de diretivas do *Assembly* ou de instruções do PIC, mas que não necessariamente fazem parte de todos os programas. Em preto estão as diretivas e

instruções necessárias à estruturação mínima do programa, e em verde são nomes atribuídos, mas que podem ser outros quaisquer. Por analogia ao C, o nome da rotina principal poderia ser, por exemplo, *main* ao invés de *Principal*.

Em *Assembly*, comentário deve ser precedido de ";" (ponto e vírgula). Assim, tudo que está nas **linhas 01 a 09**, por exemplo, são comentários de linhas. Um comentário pode vir após uma diretiva ou uma instrução, como pode ser visto nos comentários feitos, por exemplo, **nas linhas 22, 26 e 34**.

A diretiva **#INCLUDE**, **colocada na linha 10**, permite a inclusão de arquivos no programa. No exemplo, é feita a inclusão do arquivo *p16F877.inc*, o qual contém todas as definições feitas pela Microchip para pinos, registros e endereços de registros do microcontrolador 16F877. Com a inclusão desse arquivo (que pode ser encontrado na pasta de instalação do MPLAB, normalmente em *C:\Arquivos de programas\Microchip\MPASM Suite*), quando formos armazenar, por exemplo, um dado no endereço da porta B, não será preciso referenciar o endereço, basta utilizar o nome da porta, ou seja, *PORTB*, que o montador automaticamente insere o seu endereço, que é 0005.

Na **linha 13**, é usada a diretiva **\_CONFIG**, que permite definir os *bits* de configuração usados para operação e gravação do programa na memória do chip. Estão referenciados apenas alguns dos *bits* de configuração. No exemplo, todas as configurações como, por exemplo, o código de proteção (CP) e o *Watchdog* (WDT) estão desligados (OFF). O último *bit* de configuração mostrado **na linha 13** especifica que a fonte de *clock* é um cristal (XP\_OSC). Essa linha está comentada porque esta configuração será feita no próprio ambiente MPLAB e não por linha de código. Essa linha está colocada apenas para manter compatibilidade com a linha de comando *#fuses* usada no *template C*, que será sugerido numa das próximas aulas.

**Nas linhas 16 a 19**, são usadas diretivas **#DEFINE**. Essa diretiva é extremamente útil na escrita de programas, já que permite que sejam feitas diversas definições quer sejam para instruções completas, parte de instruções ou para operandos.

Como exemplo, **na linha 16**, é definido através dela que quando o montador encontrar a palavra *banco1*, a substitua pelo comando que seta o *bit* RP0 do registrador STATUS (*BSF STATUS,RP0*) e, **na linha 19**, com essa diretiva é dito ao montador que substitua a palavra *led0* como sendo o *bit* 5 da porta A (*PORTA,5*).

Entre as **linhas 22 e 26**, é mostrado como reservar posições de memórias para variáveis que serão utilizadas no programa usando as diretivas CBLOCK e ENDC. No exemplo, é reservado um byte de memória na posição 0x20 (20 hexadecimal) para a variável VarInt8 e duas posições para a variável VarInt16.

**Nas linhas 30 a 39**, são sugeridas duas áreas para escrita de duas sub-rotinas, denominadas no exemplo de SubRotina1 e SubRotina2. Todo retorno de chamada de uma sub-rotina feita pela instrução CALL deve ser feito pela instrução RETURN, quando não retorna valor (mostrada **na linha 34**), ou RETLW (mostrada **na linha 39**), que retorna da sub-rotina com um valor, no caso 10 decimal, para o registrador W. O endereço de retorno é buscado na memória pilha. No caso, o último endereço armazenado.

**Nas linhas 42 e 46**, se observa o uso da diretiva ORG. Esta diretiva define a partir de qual posição de memória a instrução seguinte e posteriores deverão ser armazenadas na memória de programa.

Como o endereço de tratamento de interrupção já é o endereço 04, na **linha 43**, é feito um desvio para a rotina principal, ficando a área do endereço 04 até a instrução RETFIE reservada para inserção da rotina de tratamento de interrupções.

A instrução RETFIE encerra a rotina de tratamento de interrupção e força o retorno do programa para o ponto de execução em que estava o programa no momento de sua ocorrência. Esse endereço de retorno é também buscado na memória pilha.

A área de memória reservada para que seja escrita a rotina principal do programa encontra-se entre a **linha 54** (abaixo do nome da rotina, no caso Principal) e a **linha 56**. **Na linha 57**, exemplifica-se como criar um loop infinito.

Complementando a análise do *template*, **na linha 58**, está a diretiva de encerramento de programa para o montador: a diretiva END.



## Atividade 03

**1. Procure construir seu próprio *template*. Busque na internet outras sugestões.**

**Uma possibilidade de busca seria: PIC16F, *template*, *Assembly*.**

## Modelo simplificado para construção de programas em *Assembly* - pt.2

Nas próximas seções, através de um exemplo prático, serão apresentados dois ambientes de programação de microcontroladores: o MPLAB e o Proteus. Nesses ambientes você terá a oportunidade de acompanhar a escrita, simulação e execução de um programa escrito em *Assembly* utilizando o *template* mostrado no Quadro 2, mas, se preferir, use o seu *template*, definido na Atividade 3.

Quando se escreve programas em *Assembly* para PIC da família 16F, deve-se ter um cuidado especial com as instruções GOTO e CALL (desvio de programa e chamada de sub-rotina, respectivamente). Fiz essa observação na aula anterior, lembra-se? Se não, seria bom fazer uma pequena revisão.

Se a posição chamada não estiver dentro da mesma página, antes de se executar o GOTO ou o CALL, deve-se posicionar através dos *bits* 3 e 4 do registrador PCLATH essa nova página.

Esse posicionamento é feito de acordo com o seguinte mapa de *bits*:

PCLATH<4:3> = 00 para a página 0; PCLATH<4:3> = 01 para a página 1;  
PCLATH<4:3> = 10 para as páginas 2 e; PCLATH<4:3> = 11 para a página 3.

Para o programador não se obrigar a conhecer posicionamentos de memória pois existe uma diretiva, a PAGESEL, que pode ser usada com essa finalidade, como no exemplo mostrado a seguir:

PAGESEL EndereçoDeDesvio ; seleciona página onde está o

; EndereçoDeDesvio

GOTO EndereçoDeDesvio ; desvia para página selecionada

De forma equivalente, a seleção de um banco de dados na memória de dados em um PIC 16F é feita através dos *bits* RP1 e RP0 do registrador STATUS de acordo com o seguinte mapeamento: STATUS<RP1:RP0> = 00 para o banco 0; STATUS<RP1:RP0> = 01 para o banco 1; STATUS<RP1:RP0> = 10 para o banco 2 e; STATUS<RP1:RP0> = 11 para o banco 3.

Ao invés de alterar diretamente os *bits* RP1 e RP0, o programador pode usar a diretiva BANKSEL, como mostra a seguinte sequência de instruções:

MOVLW 10h

BANKSEL Variavel\_1 ; seleciona banco onde se encontra a

; Variavel\_1

MOVWF Variavel\_1 ; atua no banco selecionado

Uma vantagem de se usar linguagem C é que esse tipo de preocupação, descrito nos últimos parágrafos, não se aplica, uma vez que o próprio compilador C se encarrega de fazer os devidos posicionamentos.

## Atividade 04

- 1. Quais as finalidades das diretivas PAGESEL E BANKSEL? Para quais áreas de memórias se aplicam?**

# Ambientes de desenvolvimento Proteus e MPLAB

Para apresentar os ambientes de desenvolvimento Proteus e MPLAB, será usado um exemplo prático, que é o projeto de um sistema de controle de um semáforo de dois tempos. Um semáforo de dois tempos, mostrado na Figura 1, consiste em dois semáforos interligados de forma que quando um está verde, o outro está vermelho e vice-versa.

**Figura 01** - Semáforo de dois tempos



Para simular as luzes do semáforo, serão utilizados LED. A mudança de cor de cada semáforo será feita a cada 1 s. A inteligência do sistema, ou seja, a sequência de luzes (representadas pelos LED) que será acesa e/ou apagada em cada semáforo, bem como a frequência com que isso ocorre, será de responsabilidade de um PIC16F877A.

Para acionar os LED serão utilizados os pinos do PORTB do PIC16F877A segundo o mapeamento de *bits* mostrado no Quadro 3. S1 e S2 são as denominações usadas para os dois semáforos.

	S1	S2
Verde	RB6	RB2
Amarelo	RB5	RB1
Vermelho	RB4	RB0

**Quadro 3** – Mapeamento de *bits* usado para os dois semáforos

Para controlar o semáforo, será estabelecida uma máquina de estado com quatro estados possíveis, designados de estados 0, 1, 2 e 3. No estado 0 (considerado o estado inicial), o semáforo S1 inicia verde, enquanto o semáforo S2 inicia vermelho. Após 1 s, o estado muda para o estado 1, em que S1 fica amarelo e S2 permanece vermelho. Todas as mudanças de estados e os correspondentes valores de *bits* do PORTB estão representados no Quadro 4.

Estado	Semáforo S1	Semáforo S2	PORTB	
			Valor hexadecimal	Valor binário
0	Verde	Vermelho	0x41	0b01000001
1	Amarelo	Vermelho	0x21	0b00100001
2	Vermelho	Verde	0x14	0b00010100
3	Vermelho	Amarelo	0x12	0b00010010

**Quadro 4** – Correspondência entre os estados do semáforo e os *bits* de atuação do PORTB

## Atividade 05

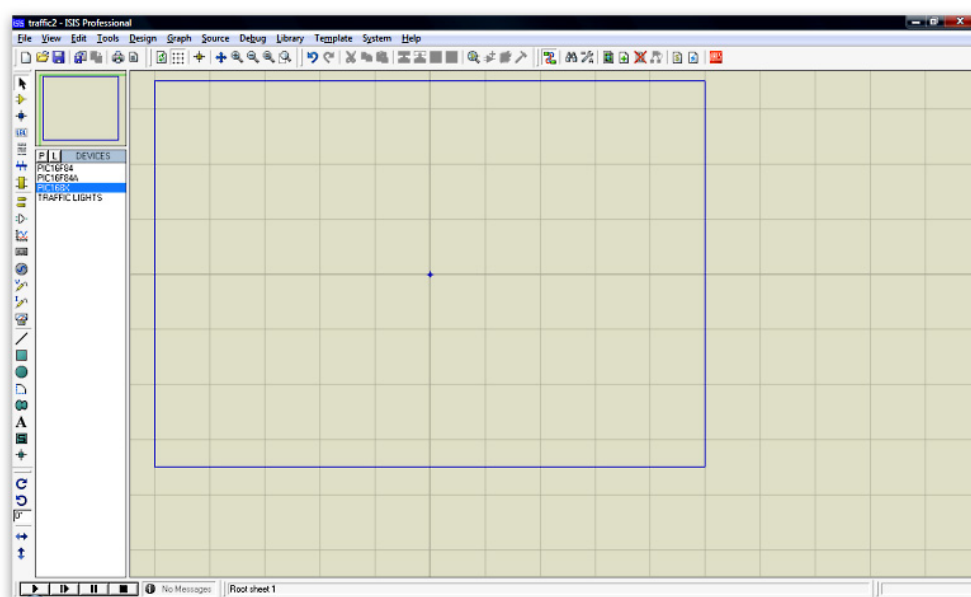
1. Ao invés de utilizar seis pinos do PORTB, seria possível utilizar três do PORTA e três do PORTC? Caso seja possível, quais alterações ocorreriam no Quadro 4?

## Proteus

O Proteus é um software que permite criar esquemas de circuito. É composto basicamente por dois módulos: o ISIS e o ARES. No ISIS, é possível escolher quais componentes o circuito terá (resistor, capacitor, conector de fonte de alimentação, LED etc.) e determinar as ligações elétricas entre eles. O ARES é responsável por definir o posicionamento dos componentes no layout final do circuito e também realizar o roteamento das trilhas do circuito. Nesta aula, apenas o ISIS será abordado.

Na Figura 2, é mostrada a tela de trabalho do ISIS, na qual se pode ver um retângulo azul, área onde devem ser inseridos os componentes dos nossos circuitos.

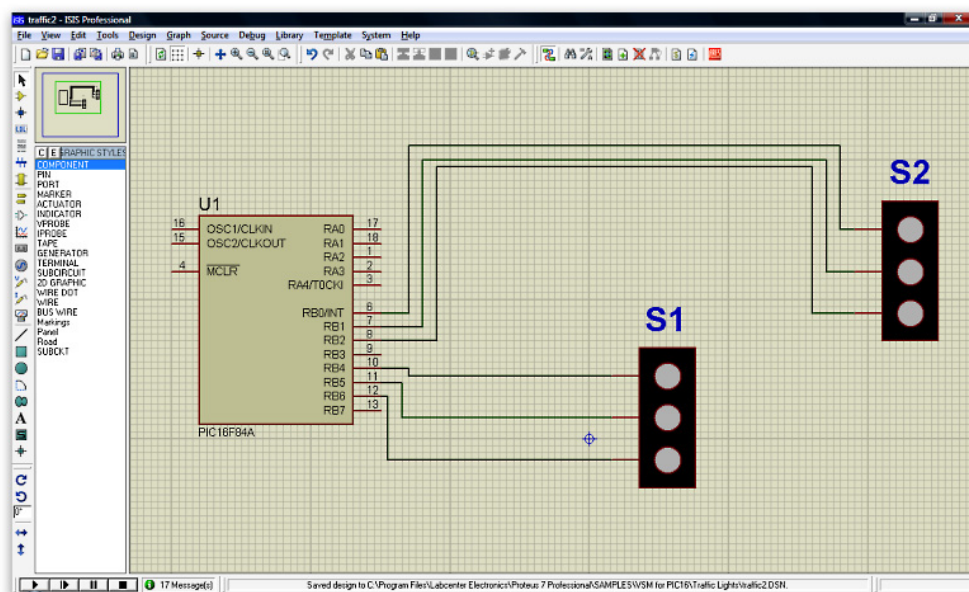
**Figura 02** - Tela de trabalho do ISIS, um dos elementos gráficos do Proteus.



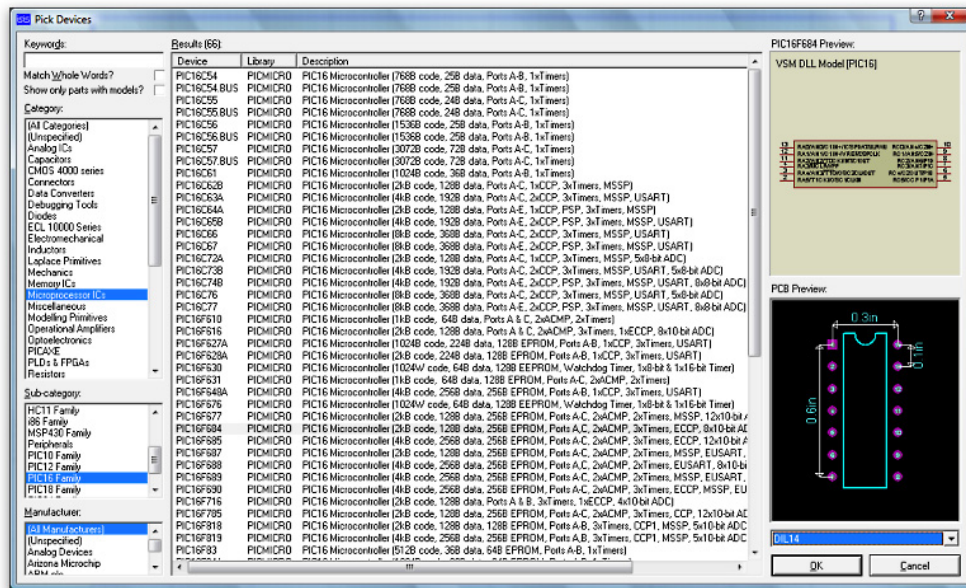
A Figura 3 mostra nosso primeiro esquema elétrico ou desenho: o circuito que irá controlar através de um PIC 16F877A a temporização do semáforo de dois tempos apresentado anteriormente. No ISIS, os circuitos desenhados são tratados por **designs**. Para abrir um design é só navegar até o menu **File** e, em seguida, **Open Design**.

Observe na Figura 2 que ao lado esquerdo do retângulo azul há um ícone **P** e um ícone **L**. Para adicionar um componente, deve-se clicar no ícone **P**. A Figura 4 mostra a tela que surge quando essa ação é efetuada. É possível fazer uma busca pelo nome do componente (no campo **keywords**) ou escolher uma categoria (no campo **category**). Nesse caso, dependendo do componente, deve-se escolher em seguida a subcategoria (no campo **subcategory**) e, por último, o dispositivo (no campo **results**).

**Figura 03** - Circuito com PIC para temporizar um semáforo de dois tempos.



**Figura 04 - Tela de escolha de componentes.**



Uma vez adicionados todos os componentes ao circuito, o próximo passo será realizar as conexões elétricas entre os componentes. Para criar uma conexão elétrica, basta clicar uma vez com o botão esquerdo do mouse no pino de origem da conexão, levar o mouse até o pino de destino da conexão elétrica e dar outro clique com o botão esquerdo.

Feitas todas as conexões, o próximo passo será escrever o código do microcontrolador. Para isso, vamos precisar conhecer o ambiente de desenvolvimento MPLAB.

## Atividade 06

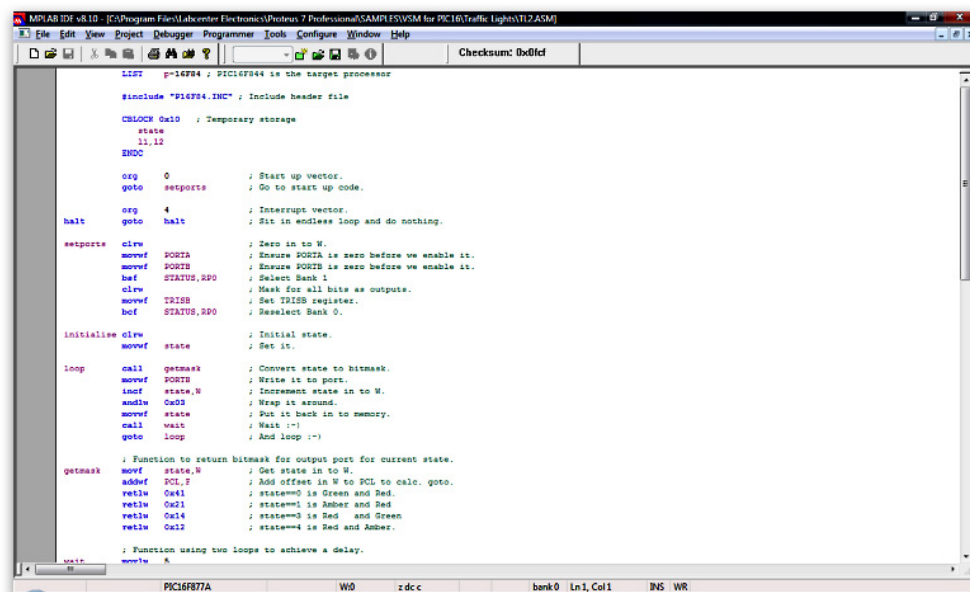
1. Pesquise na internet se o Proteus possui suporte para as seguintes famílias de microcontroladores: AVR, da Atmel, e MSP430, da Texas Instruments.

# MPLAB

O MPLAB é um software de desenvolvimento projetado e disponibilizado pela própria Microchip (fabricante dos microcontroladores PIC). Ele permite escrever, compilar, depurar código em tempo real, simular a execução do programa e, por fim, gravar o código executável gerado em um microcontrolador. Nesta aula, o MPLAB será utilizado apenas para as tarefas de edição e montagem do código.

Na Figura 5, é mostrada, no ambiente de desenvolvimento MPLAB, a tela de edição com o código de controle dos semáforos. Para abrir um código em *Assembly*, deve-se navegar até o menu **File** e, em seguida, clicar em **Open**.

**Figura 05** - Ambiente de desenvolvimento MPLAB.



```
LIST p=16F877 ; PIC16F877 is the target processor
#include "P16F877.INC" ; Include header file

CBLOCK 0x00 ; Temporary storage
state
0x12
ENDC

org 0 ; Start up vector.
goto setports ; Go to start up code.

org 4 ; Interrupts vector.
goto halt ; Sit in endless loop and do nothing.

setports cldw ; Zero in to W.
movwf PORTA ; Ensure PORTA is zero before we enable it.
movwf PORTB ; Ensure PORTB is zero before we enable it.
btf STATUS,RP0 ; Select Bank 1
cldw ; Mask for all bits as outputs.
movwf TRISS ; Set TRISS register.
btf STATUS,RP0 ; Deselect Bank 0.

initialise cldw ; Initial state.
movwf state ; Set it.

loop call getmask ; Convert state to bitmask.
movwf PORTB ; Write it to port.
incf state,W ; Increment state in to W.
andlw 0x03 ; Wrap it around.
movwf state ; Put it back in to memory.
call wait ; Wait 10ms
goto loop ; And loop :-)
```

```
getmask movf state,W ; Get state in to W.
addwf PCL,F ; Add offset in W to PCL to calc. goto.
retlw 0x01 ; state==0 is Green and Red.
retlw 0x02 ; state==1 is Amber and Red.
retlw 0x04 ; state==2 is Red and Green.
retlw 0x08 ; state==3 is Red and Amber.

; Function using two loops to achieve a delay.
wait movlw 0x00
loopwait movlw 0x00
loopwait movlw 0x00
```

Na Figura 6, pode-se observar parte do programa escrito em linguagem *Assembly*. A primeira linha contém a diretiva **LIST p=16F877**, a qual especifica que o modelo de PIC a ser utilizado será o 16F877. É necessário incluir o arquivo de cabeçalho (ou definições) do modelo de PIC a ser utilizado. Isso é feito através da diretiva **#include**.



Uma vez escolhido o modelo e incluído o arquivo de definições, o próximo passo é a criação (ou definição) das variáveis que serão utilizadas no programa. Para isso, foi utilizada a diretiva **CBLOCK**. Neste exemplo da Figura 6, são utilizadas três variáveis: **state**, **I1** e **I2**. Após a definição da última variável (nesse caso, **I2**) o bloco de variáveis é encerrado com a diretiva **ENDC**.

A próxima linha contém a diretiva **org 0**, definindo que o código que vem a seguir será armazenado a partir da posição 0 da memória de programa.

Em seguida, tem-se a diretiva **goto setports** que desvia a execução do programa para a função **setports**, responsável pela inicialização do PORTB como saída.

Após o **goto**, tem-se a diretiva **org 4**, e mais uma vez o armazenamento do código é reposicionado. Dessa vez para o endereço 4 da memória de programa, reservada para rotinas de tratamento de interrupções.

Em seguida, tem-se a linha de comando **halt goto halt**, a qual leva o programa a executar um *loop* infinito.

No código escrito para esse programa está também definida a função **initialise**, que define o estado inicial do semáforo.

**Figura 06** - Parte inicial do programa escrito em Assembly para controle dos semáforos.

```
LIST      p=16F877 ; PIC16F877 é o modelo utilizado

#include "P16F877.INC" ; Inclui arquivo de cabeçalho

CBLOCK 0x10 ; Definição de variáveis
    state,I1,I2
ENDC

org      0
goto    setports

org      4
halt     goto halt ; Looping infinito

setports clrw
movwf   PORTA
movwf   PORTB
bsf     STATUS,RP0
clr     TRISB
movwf   STATUS,RP0 ; Define todos os bits do PORTB como saída

initialise clrw ; Definição do estado inicial
movwf    state
```

A Figura 7 mostra uma segunda parte do programa. A função **loop** é responsável por atribuir o estado do semáforo ao PORTB do PIC. Para isso, pode-se observar que a primeira linha da função **loop** contém uma chamada (através da diretiva **call**) à função **getmask**, responsável pelo cálculo do próximo estado que o semáforo irá assumir. Uma vez definido o próximo estado, a função **loop** se encarrega de atribuir as condições desse novo estado ao PORTB do PIC.

**Figura 07** - Segunda parte do programa em Assembly.

```
loop      ; Função que atribui o estado do semáforo à saída do PORTB do PIC
          call    getmask
          movwf   PORTB
          incf    state,W
          andlw   0x03
          movwf   state
          call    wait
          goto    loop

getmask   ; Função que calcula o próximo estado
          movf    state,W
          addwf   PCL,F
          retlw   0x41
          retlw   0x21
          retlw   0x14
          retlw   0x12
```

A Figura 8 mostra duas funções: **wait** e **wait2**. Essas duas funções trabalham em conjunto e servem para gerar um atraso (ou **delay**). Neste exemplo, o atraso é usado para que seja possível visualizar a mudança de estado do semáforo.

**Figura 08** - Terceira parte do programa em *Assembly*.

```
wait      ; Função que utiliza dois loops para delay
          movlw   5
          movwf   11

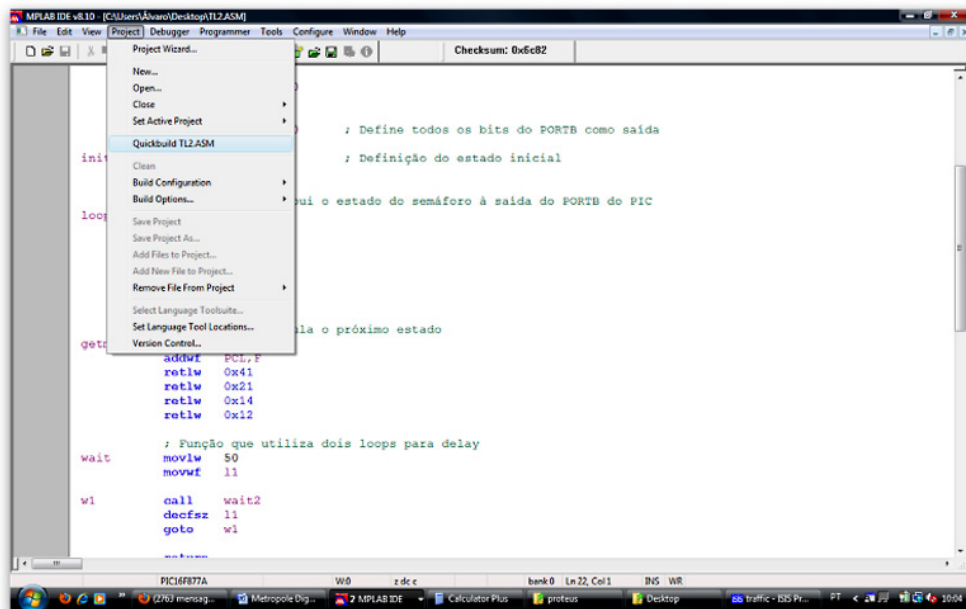
w1         call    wait2
          decfsz  11
          goto    w1

          return

wait2     clrf     12
w2        decfsz  12
          goto    w2
          return
          END
```

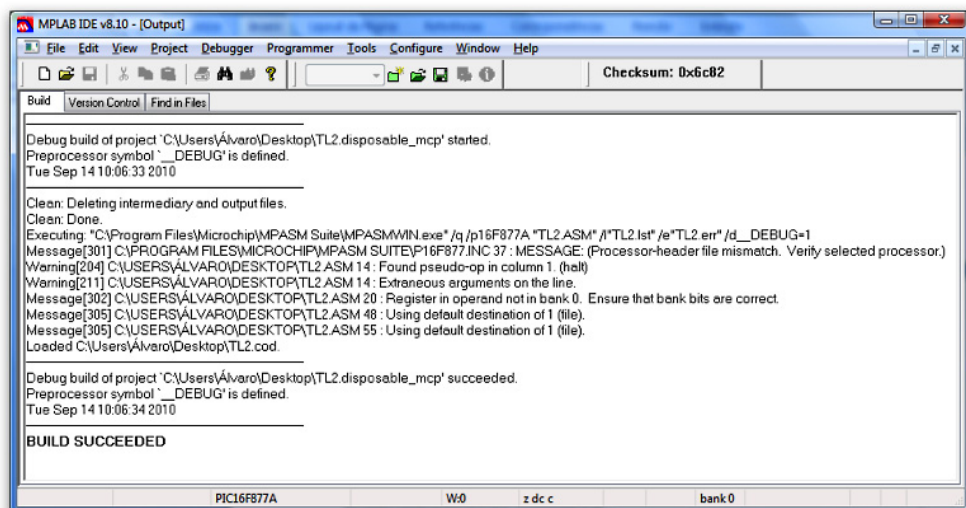
Uma vez escrito o código, o próximo passo é sua compilação ou montagem. Para montar o código no MPLAB, basta navegar até o menu **Project** e clicar na opção **Quickbuild**. Esse processo é ilustrado na Figura 9.

**Figura 09** - Processo de montagem de um programa em *Assembly* no MPLAB.



Se a montagem for bem-sucedida, ou seja, se o código não apresentar nenhum erro, aparecerá a mensagem **BUILD SUCCEEDED**, conforme mostrado na Figura 10.

**Figura 10** - Tela mostrada no MPLAB após um processo de montagem.



Observe ainda na Figura 10 que acima da mensagem **BUILD SUCCEEDED** aparece todos os avisos (**warnings**) gerados durante a montagem do código. O montador mostra ainda a linha de código de cada um dos avisos e erros. Isso é útil quando o código apresenta erros, pois o programador já sabe imediatamente em qual linha esse erro ocorreu.

## Atividade 07

1. **Pesquise na internet se é possível utilizar o MPLAB para escrever e compilar o código escrito em linguagem C. Pesquise também se é possível simular a execução do código utilizando o MPLAB.**

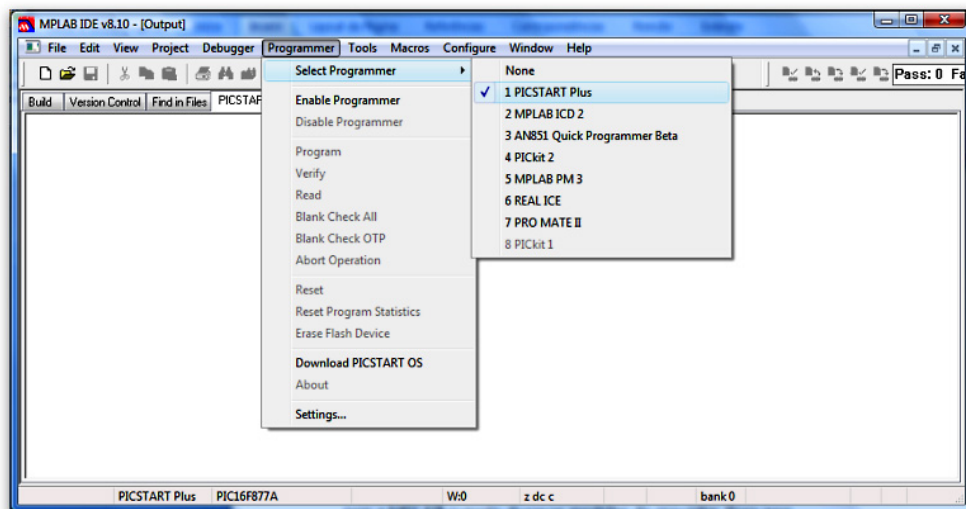
## Usando o Proteus para simulação de programas

Com o código gerado, deve-se retornar ao Proteus, associar o código .hex gerado pelo MPLAB para daí poder simular a execução do programa. Essa tarefa deverá ser feita com o auxílio do monitor ou tutor durante a aula prática.

## Gravando o código gerado em um microcontrolador PIC

Uma vez compilado o código no MPLAB e testado sua execução correta no Proteus, o próximo passo seria gravar o código em um microcontrolador. Para isso, é necessário utilizar um circuito de gravação. No MPLAB existem várias opções. Escolher qual gravador usar e como proceder também é uma tarefa que deverá ser feita com o auxílio do monitor ou tutor durante a aula prática. Só adiantando, para selecionar o gravador e iniciar o processo de gravação, deve-se navegar até o menu **Programmer** e escolher a opção **Select Programmer**. Essa nova tela é mostrada na Figura 11.

**Figura 11** - Menu para escolha do programador.



Uma vez feito isso, se deve navegar novamente até o menu **Programmer** e escolher a opção **Enable Programmer**. Por último, navega-se até o menu **Programmer** e escolhe-se a opção **Program**.

# Resumo

Nesta aula, foi apresentada a linguagem *Assembly*, algumas de suas diretivas, o formato das instruções e o conjunto de instruções dos microcontroladores da família 16F. Foram apresentados um ambiente de desenvolvimento de programas e outro de simulação para microcontroladores: o MPLAB e o Proteus. Foi proposta também uma atividade prática: o desenvolvimento de um circuito de controle de um semáforo de dois tempos.

## Autoavaliação

- 1. Quais são as linguagens mais apropriadas para escrever programas para microcontroladores? Como essa definição ocorreu ou está ocorrendo?**
- 2. Redefina o modelo básico fornecido para construção de programas em linguagem *Assembly* para um modelo que possa ser utilizado em seus projetos.**
- 3. Quais os grupos de instruções dos microprocessadores da família PIC 16F? Descreva pelo menos uma de cada grupo.**
- 4. Quais as principais diretivas usadas no modelo fornecido para programação *Assembly*. Para que é usada a diretiva ORG? E a diretiva #INCLUDE?**
- 5. Qual dos ambientes de desenvolvimento apresentados é mais adequado para simulações durante a execução de programas? Por quê?**

## Referências

MPASM/MPLINK User's Guide. **Microchip Technology Inc.** 2009.

MPLAB IDE User's Guide. **Microchip Techonology Inc.** 2009.

PEREIRA, Fábio. **Microcontroladores PIC**: Programação em C. 4. ed. São Paulo: Érica, 2005.

SOUZA, David José de; LAVINIA, Nicolas Cesar. **Conectando o PIC**: explorando recursos avançados. São Paulo: Érica, 2003.

ZANCO, Wagner da Silva. **Microcontroladores PIC**: técnicas de software e hardware para projetos de circuitos eletrônicos. São Paulo: Érica, 2006.