PIC by example

Curso prático sobre PICmicro™

*Realizado por **Sergio Tanzilli** com a colaboração de **Tiziano Galizia** para **PicPoint***
*A tradução do Italiano para o Português é editada pôr **Marcio Nassorri***

PicByExample é um curso de programação elaborado para quem deseja iniciar pela primeira vez em projetos eletrônicos baseado nos microcontroladores **PICmicro** produzido pela **Microchip Technology Inc.**

O curso é direcionado aos principiantes que desejam ingressar pela primeira vez no fascinante mundo da microeletrônica como também para os que tem maiores conhecimentos e queiram aprender rapidamente como desenvolver aplicações para os PICmicro.

As lições podem ser lidas pagina pôr pagina como se fosse um livro, seguindo-se os argumentos propostos um por um, ou também selecionando-os diretamente segundo os próprios interesses, descritos no índice de lições.

Os argumentos tratados correspondem aos aspectos ligados aos **projetos do hardware** bem como aqueles ligados ao **desenvolvimento do software** e são amplamente explicados por exemplos práticos e indicações claras para realização das aplicações de teste.

A linguagem de programação utilizada no nosso curso e para realização das aplicações de teste é o **ASSEMBLER**.

O curso está escrito inteiramente no formato **HTML** e pode ser consultado por browsers mais difusos. Durante a leitura será possível acessar através das conexões de hipertexto (link) diretamente ao principal, aos esquemas elétricos e a especificações mais profundas com um simples click do mouse.

O exemplo prático ilustrado faz referência à placa PicTech disponível em kit, ou podendo ser feita pôr conta própriaseguindo-se as instruções contidas no curso.

O curso é completamente gratuito e pode ser acessado diretamente pela internet no endereço http://www.picpoint.com/picbyexample/index.htm . Para quem tiver dificuldade em efetuar o download da pagina, é possível receber o curso via e-mail diretamente do autor: Sergio Tanzilli (tanzilli@picpoint.com).

BOA LEITURA E BOM DIVERTIMENTO COM O PICmicro™...!!!

Seremos enormemente gratos aos comentários, indicações e conselhos diretamente ao e-mail do autor:

Sergio Tanzilli (tanzilli@picpoint.com)
Rome - Italy
Tel. +39 0335 61.52.295

Marcio A. Nassorri (sman@ttelecom.com.br)
S.J.Rio Preto – Brasil.

Índice das lições

Lição 1	Introdução	

Introdução ao PIC

Al termino desta lição saiba:

- O que é um PIC.
- Como realizar um simples circuito de prova.
- Como escrever e compilar um programa em assembler.
- Como programar um PIC.

Conteúdo da lição 1

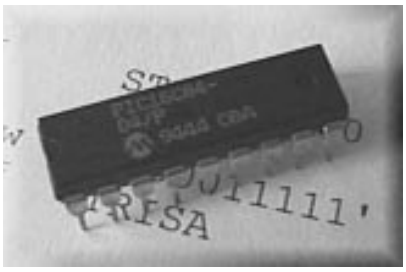
1. **O Que é um PIC**
2. **Realizando um simples lampejador com led**
3. **Escrita e compilação de um programa em assembler**
4. **Analisando um código em assembler**
5. **Compilando um código em assembler**
6. **Programando o PIC**

Lição 1 | **Passo 1/6**

O que é o PIC?

O **PIC** é um circuito integrado produzido pela Microchip Technology Inc., que pertence da categoria dos microcontroladores, ou seja, um componente integrado que em um único dispositivo contém todos os circuitos necessários para realizar um completo sistema digital programável.

Como se pode ver na figura,



o PIC (neste caso um PIC16F84) pode ser visto externamente como um circuito integrado TTL ou CMOS normal, mas internamente dispõe de todos os dispositivos típicos de um sistema microprocessado, ou seja:

- Uma **CPU** (**C**entral **P**rocessor **U**nit, ou seja, Unidade de Processamento Central) e sua finalidade é interpretar asinstruções de programa.
- Uma memória **PROM** (**P**rogrammable **R**ead **O**nly **M**emory ou Memória Programável Somente para Leitura) na qual irá memorizar de maneira permanente as instruções do programa.
- Uma memória **RAM** (**R**andom **A**ccess **M**emory ou Memoria de Accesso Aleatório) utilizada para memorizar as variáveis utilizadas pelo programa.
- Uma serie de **LINHAS de I/O** para controlar dispositivos externos ou receber pulsos de sensores, chaves, etc.
- Uma serie de dispositivos auxiliares ao funcionamento, ou seja, gerador de clock, bus, contador, etc.

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

O **PIC** está disponível em uma ampla gama de modelos para melhor adaptar-se as exigências de projetos específicos, diferenciando-se pelo número de linha de I/O 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.

Uma descrição detalhada da tipologia do PIC é disponível no site da Microchip acessável via , onde conseguimos encontrar grandes e variadas quantidades de informações técnicas, software de apoio, exemplos de aplicações e atualizações disponíveis.

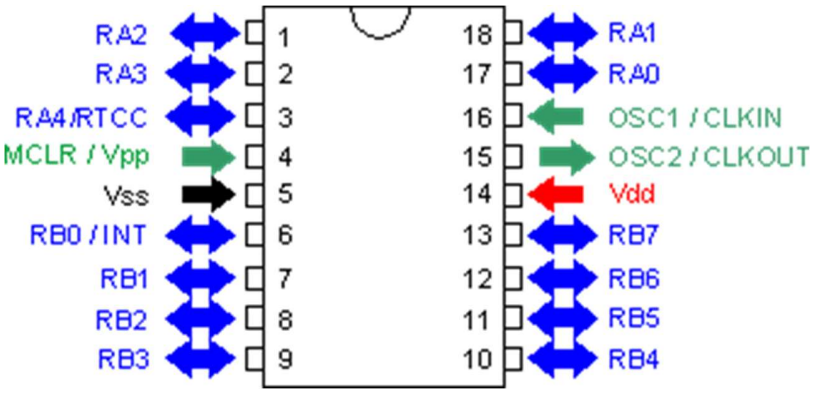
Para o nosso curso usaremos um modelo intermediário de PIC o **PIC16C84**. Este é dotado de 18 pinos sendo 13 disponíveis para o I/O, ou seja, para serem ligados ao nosso circuito e de algumas características que o tornam umcircuito que melhor atenderá as exigências do nosso curso.

Em particular o PIC16C84 dispõe de uma memória para armazenar o programa, do tipo **EEPROM** ou seja **E**lectrical **E**rasable **P**rogrammable **R**ead **O**nly **M**emory, que pode ser rescrita quantas vezes quisermos e que é ideal para o nosso experimento tornando a conexão para a programação on-board, ou seja podemos colocar o programa dentro do chip sem ter que remove-lo do circuito de prova.

Tal característica é plenamente aproveitada em nosso programador **YAPP!** descrito neste curso e fornecido a qualquer um que queira adquirir o nosso kit completo **Pic*Tech***. Em alternativa é possível utilizar um programador produzido pela Microchip ou outro programador produzido por terceiros.

Se não tiver adquirido o nosso kit mas tem conhecimento para construir o programador YAPP! poderá encontrar neste curso toda a documentação necessária para realiza-lo sozinho

E agora chegou o momento de dar uma olhada no **PIC16C84**. Vejamos na figura a reprodução da pinagem e nomenclatura de seus respectivos pinos:



Como é possível ver, o PIC16C84 é dotado de um total de **18 pinos** dispostos em duas fileiras paralela de 9 pinos cada uma (dual in line). Os pinos contrastados em **AZUL** representam as linhas de I/O disponíveis para a nossa aplicação, o pino em **VERMELHO** e o **PRETO** são os pinos de alimentação, e os em VERDE são reservados ao funcionamento do PIC (MCLR para o reset e OSC1-2 para o clock).

Clicando sobre a inicial de cada pino é possível visualizar uma breve descrição de seu respectivo funcionamento.

No próximo passo veremos como fazer a ligação desses pinos ao nosso primeiro circuito experimental para verificar imediatamente o seu funcionamento.

Lição 1	Passo 2/6	

Realizando um simples lampejador com LED

Após termos visto brevemente o que é e como é feito um PIC, faremos agora uma simples aplicação pratica.

Iremos fazer um circuito muito simples e seu propósito é fazer lampejar um diodo led. Vejamos como se escreve um programa em assembler, como se compila e como se transfere para o interior da EEPROM do PIC o programa e assim fazê-lo funcionar.

O circuito a ser realizado está representado no seguinte arquivo no formato Acrobat Reader (9Kb): example1.pdf.

Como descrito anteriormente o pino **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 logica zero. No nosso circuito é conectado diretamente ao positivo do programador YAPP!.

O pino **OSC1/CLKIN** (pino 16) e **OSC2/CLKOUT** (pino 15) são conectados internamente ao circuito para gerar a frequência de clock utilizada para temporizar todo o ciclo de funcionamento interno do chip. Desta frequência depende a maior parte das operações interna e em particular a velocidade com que o PIC processa as instruções do programa. No caso do **PIC16C84-04/P** tal frequência pode chegar a um máximo de **4Mhz** da qual se obtém uma velocidade de execução das instruções par de **1 milhão de instruções por segundo**. No nosso caso para o oscilador de clok usaremos um cristal de quartzo de 4 MHz e dois capacitores de 22pF.

O pino **RB0** (pino 6) é uma das linhas de I/O disponível no PIC. Neste caso esta linha esta conectada a um led por intermédio de um resistor de limitação de corrente.

Uma vez terminada a apresentação do circuito passemos ao próximo passo para aprender como escrever o programa que o PIC deverá executar.

Lição 1 | **Passo 3/6**

Escrita e compilação de um programa em assembler

Como em qualquer sistema microprocessado, no PIC também é necessário preparar um programa que o faça desenvolver seu trabalho.

Um programa é constituído por um conjunto de instruções em sequência, onde cada uma identificara precisamente a função básica que o PIC irá executar. Onde a instrução é representada por um código operativo (do inglês **operation code** ou abreviadamente **opcode**) podemos memorizar 14 bits em cada locação da memória EEPROM. Esta memória no PIC16F84 dispões de 1024 posições e cada uma deverá conter uma só instrução. Um exemplo de opcode em notação binaria está escrito a seguir:

00 0001 0000 0000B

é mais provável que um opcode venha representado na notação hexadecimal ou seja:

0100H

que representa exatamente o mesmo valor mas numa forma reduzida. La letra **H**, escrita no final do valor 0100, indica o tipo de notação (Hexadecimal). O mesmo valor pode ser representado em assembler com a notação 0x100 que é derivado da linguagem C ou H'0100'.

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

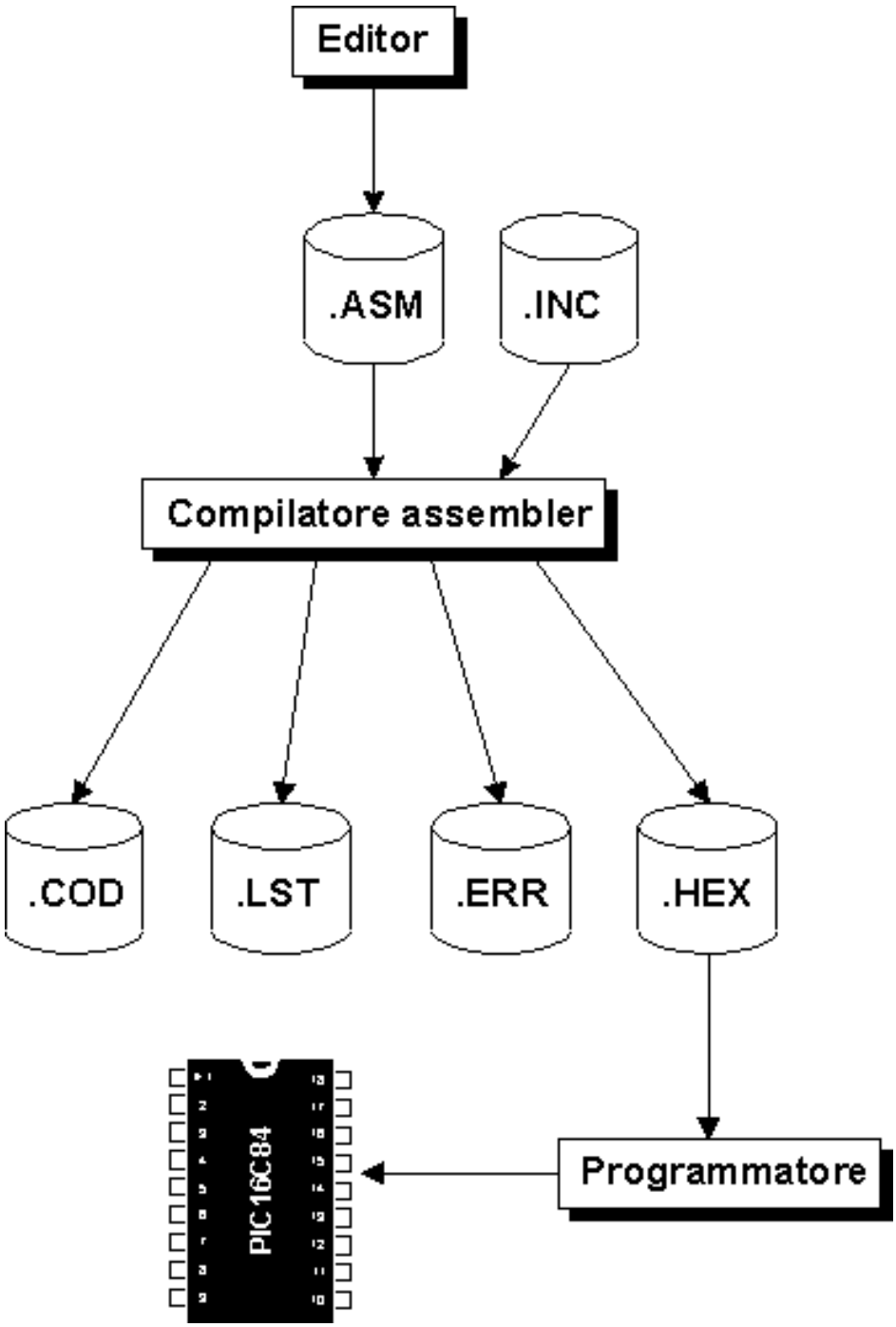
A primeira convenção é a que associa o opcode (um total de 35 para o PIC16C84) a uma sigla **mnemônica**, ou seja, umainicial que seja fácil de recordar o significado da instrução

Voltando ao nosso exemplo o opcode 0100H corresponde a instrução mnemônica **CLRW** que é a forma abreviada da **instrução CL**EA**R W** REGISTER, ou seja, zere o registro W (veremos posteriormente o que significa).

Outra convenção consiste na definição, da variável, da constante, do label(rótulo) de referência ao endereço de memória, etc. O propósito desta convenção é de facilitar a escrita de um programa para o PIC e é chamada **linguagem assembler**. Um programa escrito em linguagem assembler pode ser escrito em qualquer PC utilizando-se qualquer processador de texto que possa gerar arquivos ASCII(Word,Notpad etc).Um arquivo de texto que contenha um programa em assembler é denominado de **source ou código assembler**.

Uma vez preparado o nosso código assembler (veremos mais adiante), iremos precisar de um programa para traduzir as instruções mnemônicas e todas as outras formas convencionais com que escrevemos o nosso código em uma serie de números (o opcode) reconhecível diretamente pelo PIC. Este programa se chama **compilador assembler** ou **assemblador (montador)**.

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



Como já foi dito a primeira operação a ser efetuada é a escrita do código assembler e a sua gravação em um arquivo de texto com a extensão **.ASM**. Para fazer isto aviamos dito para recorrer a um editor ASCII, ou seja, um programa de escrita por exemplo o **NOTEPAD.EXE** do Windows 3.1© ou Windows 95© ou o **EDIT.EXE** do MS/DOS©. É possível gerar este arquivo com programas mais sofisticados como o Word© ou Wordperfect© tendo somente que tomar o cuidado de muda-los para o formato texto e não em seu formato nativo. Isto para evitar que venhamos memorizar antes o caractere de controle de formatação de testo que o compilador assembler não está preparado para reconhecer.

Na nossa primeira experiência pratica utilizaremos o arquivo de nome LED.ASM.

O próximo passo é a compilação do código, ou seja, a transformação em opcode do código mnemônico ou instruções assembler deste conteúdo.

O compilador assembler que utilizaremos é o **MPASMWIN.EXE** produto da Microchip e disponível no site internet; é possível também conseguir uma cópia nas páginas de downloads.

Como é possível ver adiante, além do nosso código com extensão **.ASM** é necessário fornecer ao compilador um segundo arquivo produto da Microchip com extensão **.INC** diferente do tipo que estamos utilizando. No nosso caso o arquivo é o P16F84.INC. Este código contém algumas definições da qual dependi o tipo de chip utilizado e que veremos mais adiante.

Durante a compilação do código, o compilador assembler gera uma serie de arquivos com nome idêntico ao código, mas com extensões diferentes:

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

O arquivo. LST, .ERR é utilizado somente para controle da compilação. Somente o arquivo. HEX será utilizado realmente para programar o PIC. Vejamos como.

O arquivo (.hex) não é um arquivo no formato binário e não reflete diretamente o conteúdo que deverá ter a EEPROM do PIC. Mas os formatos refletirão diretamente quando forem transferidos ao PIC em uma forma legível e com algumas instruções a mais.

Sem entrar em detalhes é útil saber que tal formato é diretamente reconhecido pelo programador do PIC que promovera durante a programação a conversão em binário e contem, outro opcode outras informações que serão adicionadas aos endereços na qual vamos transferir o opcode.

No próximo passo analisaremos o nosso primeiro código assembler e veremos um boa parte da convenção utilizada na linguagem assembler.

Lição 1	Passo 4/6	

Analisando um codigo assembler

Analisaremos agora linha por linha o conteúdo do nosso código LED.ASM. Se você dispõe de uma impressora é útil efetuar uma cópia do código para poder seguir melhor a nossa explicação. Em alternativa é preferível que você visualize o código em uma janela separada de maneira a poder seguir simultaneamente o código e a relativa explicação.

Partiremos da primeira linha de código:

```
PROCESSOR       16F84
```

PROCESSOR é uma diretiva do compilador assembler que indica a definição de qual microprocessador está escrito o nosso código. A diretiva não é uma instrução mnemônica que o compilador traduz no respectivo opcode, mas sim uma simples indicação enviada ao compilador para determinar o funcionamento durante a compilação. E neste caso informamos ao compilador que a instrução que acabamos de colocar no nosso código é relativa a de um PIC16C84.

```
RADIX           DEC
```

A diretiva **RADIX** serve para informar o compilador que o número sem a notação, será entendido como número decimal. Ou seja, se quisermos especificar, por exemplo o número hexadecimal 10 (16 decimal) não podemos escreversomente 10 porque ele será interpretado como 10 decimal, neste caso ecrevemos10h ou 0x10 ou H'10'.

```
INCLUDE "P16F84.INC"
```

Veja uma outra diretiva. Desta vez indicamos ao compilador a nossa intenção de incluir no código um segundo arquivo denominado P16F84.INC. O compilador se limitará a substituir a linha contendo a diretiva INCLUDE com o conteúdo do arquivo indicado e vai efetuar a compilação como se fosse ante disso parte do nosso código.

```
LED     EQU     0
```

Mais diretiva! Mas quando veremos as instruções? Calma tenha um pouco de paciência.

A diretiva **EQU** é muito importante quando si trata de definir com ela uma constante simbólica dentro do nosso código. Em particular a palavra **LED** daqui em diante no código será equivalente ao valor 0. O ponto principal da existência da diretiva EQU é se não tornar o código mais legível e podermos colocar um valor constante em um único ponto do código

É importante notar que a palavra LED não identifica uma variável, mas simplesmente um nome simbólico valido durante a compilação. Não será nunca possível inserir instruções do tipo LED = 3 dentro do código quando a determinação dinâmica de um valor e de uma variável é uma operação que recebe a intervenção da CPU do PIC e que sempre deve ser expressa com instrução e não diretiva.

A diretiva faz sentido somente durante a compilação do código depois o PIC não poderá mais seguir uma diretiva. Vejamos agora a linha seguinte:

```
ORG 0CH
```

ORG também é uma diretiva que permite definirmos o endereço na qual queremos que o compilador inicie a alocar o dado ou a instrução seguinte. E neste caso estamos definindo uma área de dados dentro do PIC, ou seja, uma área em quememorizaremos variável e contador durante a execução do nosso programa. Esta área coincide exatamente com a área de RAM do PIC definida pela Micrchip como **FILE REGISTER**.

O registrador de arquivo nada mais é do que uma locação na RAM disponível que começa a partir do endereço 0CH. Este endereço é fixo e não pode ser modificado enquanto a locação anterior for usada para outro registro especial de usointerno.

```
Count   RES 2
```

Nesta linha encontramos um label(rotulo): **Count** e uma diretiva: **RES**.

A diretiva RES indica ao compilador que queremos reservar um certo número de bytes no meio do registrador de arquivos (file register) dentro da área de dados; e neste caso 2 bytes. O label Count, onde Count é um nome para nós, é um marcador que no resto do código assumirá o valor do endereço em que está colocado. Dado este que anteriormente aviamos definido o endereço de partida em 0CH com a diretiva ORG, Count irá a 0CH. Se nesse exemplo inserirmos um label após a linha sucessiva essa irá a 0CH + 2 (dois são os bytes que aviamos reservado) ou seja 0EH. O nome do label pode ser qualquer nome com exceção da palavra reservada ao compilador as quais são as instruções mnemônicas ediretivas)

Um rótulo difere de uma constante simbólica porque seu valor é calculado durante a compilação e não atribuído estaticamente por nós.

```
ORG 00H
```

Esta segunda diretiva ORG faz referência a um endereço na área da (EEPROM) antes da área de dados. Deste ponto em diante colocaremos de fato a instrução mnemônica que o compilador devera converter no respectivo opcode do PIC.

O primeiro opcode visto pelo PIC após o reset é aquele memorizado na locação 0, e daí o valor 00H inserido na ORG.

```
bsf STATUS,RP0
```

E finalmente a primeira instrução mnemônica de parâmetro, completa. O PIC tem uma CPU interna do tipo **RISC** onde a instrução ocupa una só locação de memória, opcode e parâmetro incluso. E neste caso a instrução mnemônica **bsf** que dizer **B**IT **S**ET **F**ILE REGISTER ou seja coloque em um (condição logica alta) um dos bits contidos na locação de memória ram especificada.

O parâmetro STATUS está definido no arquivo P16F84.INC e o passa através de uma diretiva EQU. O valor colocadoneste arquivo é 03H e corresponde a um registrador de arquivo (ou seja, uma locação na ram na área de dados) reservado.

O próximo parâmetro RP0 está definido também no arquivo P16C84.INC com valor 05H e corresponde ao número do bit que se quer colocar em um. este registrador de arquivo tem 8 bits e começa pelo número 0 (bit menos significativo) e vai até o número 7 (bit mais significativo)

Esta instrução na pratica coloca em 1 o quinto bit do registrador de arquivo STATUS. Esta operação é necessária, como veremos na próxima lição, para acessar o registrador de arquivo TRISA e TRISB.

```
movlw 00011111B
```

Esta instrução significa: **MOV**E **L**ITERAL TO **W** REGISTER (passar o literal para W) ou seja mover um valor constante para o acumulador W. Como haveremos de ver mais adiante, o acumulador, é um registro particular utilizado pela CPU em todas as situações e viermos efetuar uma operação entre dois valores ou em operações de deslocamento entre posições da memória. Na pratica é um registro de apoio utilizado pela CPU para memorizar temporariamente um byte toda vez que houver necessidade.

O valor constante para memorizar no acumulador é **00011111B** ou seja um valor binário de 8 bits onde o bit mais da direita representa o bit 0 ou o bit menos significativo.

Na próxima instrução temos:

```
movwf TRISA
```

o valor 00011111 está memorizado no registro TRISA (como para o registro STATUS o TRISA também é definido através de uma diretiva EQU) a sua função é senão a de definir o funcionamento da linha de I/O do PORTA. Este bit é em particular um bit do registro TRISA e determina em leitura(entrada) sua respectiva linha do portA , se estivesse em 0 determinaria em escrita(saída).

Na tabela seguinte está descrito configuração que assumirão os pinos do PIC quando executar esta instrução:

N.bit registro TRISB	Linha porta A	N.Pino	Valor	Estado
0	RA0	17	1	Entrada
1	RA1	18	1	Entrada
2	RA2	1	1	Entrada
3	RA3	2	1	Entrada
4	RA4	3	1	Entrada
5	-	-	0	-
6	-	-	0	-
7	-	-	0	-

Como é possível se ver o bit 5, 6 e 7 não correspondem a nenhuma linha de I/O e seus valores nada influenciam.

As duas próximas instruções indicam o funcionamento do portB do PIC:

```
movlw BT11111110T
movwf TRISB
```

e neste caso a definição da linha será a seguinte:

N.bit registro TRISB	Linha porta B	N.Pino	Valor	Estado
0	RB0	6	0	Saída
1	RB1	7	1	Entrada
2	RB2	8	1	Entrada
3	RB3	9	1	Entrada
4	RB4	10	1	Entrada
5	RB5	11	1	Entrada
6	RB6	12	1	Entrada
7	RB7	13	1	Entrada

Notou como o valor 0 no bit 0 do registro TRISB determina a configuração em escrita(saída) da respetiva linha do PIC. Na nossa aplicação esta enfatizado que esta linha será usada para controlar o LED e faze-lo lampejar.

Aviamos visto que a instrução **movwf TRISB** transferia o valor contido no acumulador (inicializado anteriormente com a instrução **movlw 11111110B**) no registro TRISB. O significado de movwf é **MOV**E **W** TO **F**ILE REGISTER (passe o valor de W para o registrador de arquivo).

```
bcf STATUS,RP0
```

Esta instrução é similar a **bsf** vista anteriormente, com a diferença de colocá-lo em zero. E(bcf) significa neste caso **B**IT **C**LEAR **F**ILE REGISTER.

Do ponto de vista funcional está instrução permite o acesso ao registro interno do banco 0 ou seja da qual faz parte o portA e portB, e banco 1 da qual faz parte TRISA e TRISB. Uma descrição mais detalhada veremos mais a frente neste curso.

```
bsf PORTB,LED
```

Com esta instrução será efetuada a primeira operação na qual veremos o resultado do lado de fora do PIC. Particularmente irá acender o led conectado a linha RB0. **PORTB** é uma constante definida no P16F84.INC e faz

referência ao registrador de arquivo correspondente a linha de I/O do portB onde **LED** é o número da linha que irá a 1. Se bem recordas, no início do código a constante LED está definida em 0, quando a linha que interessa será RB0.

```
MainLoop
```

Esta linha contém um **label,** ou seja**,** uma referência simbólica a um endereço de memória. O valor do label, como dito anteriormente, vem calculado na fase de compilação com base no número de instrução, a diretiva ORG e a outra instrução alocam espaço na memória do PIC. E neste caso, se tínhamos a instrução colocada a partir da última diretiva ORG podemos colocar o valor que vira seguido a MainLoop ou seja **07H**.

Na realidade o valor que assumi o label não tem muita importância e o seu propósito é justamente o de indicar a posição precisa do opcode na memória do PIC , ou seja um modo de referenciar uma determinada locação de memória.

E neste caso o label MainLoop sera utilizado como ponto de entrada num ciclo (do inglês Loop) de acender e apagar o led, ou seja, uma parte do código que colocara o mesmo num ciclo infinito. Encontraremos mais a frente uma referência a este label.

```
call Delay
```

Esta instrução determina uma chamada (do inglês *call*) a uma sub-rotina que inicia em correspondência com o label **Delay**.

A sub rotina é parte especial de um programa que efetua uma função especifica. onde a qualquer momento esta função pode ser chamada com uma só instrução, vejamos todas as instruções necessárias para efetua-la. Onde neste caso a sub-rotina insere um retardo para o tempo de acender e apagar o led.

A instrução que compõe a sub-rotina **Delay** foi inserida como se segue no código.

```
btfsc PORTB,LED
```

O significado desta instrução é **B**IT **T**EST **F**LAG, **S**KIP IF **C**LEAR, ou seja, controla o estado de um bit dentro de um registro e pula a próxima instrução se o valor de tal bit é zero. O bit que será controlado corresponde a linha de saída na qual está conectado o led, fazendo este teste podemos determinar se o led está aceso ou apagado, e então agir sobre ele, ou seja se o led estiver aceso nós o apagaremos e se estiver apagado nós o encenderemos.

```
goto SetToZero
```

Esta instrução determina um salto incondicionado (do inglês **GO TO**, vá para) para o label SetToZero onde teremos a instrução para apagar o led. Esta instrução será pulada para instrução seguinte se o led está apagado.

```
bsf PORTB,LED
goto MainLoop
```

Estas duas instruções simplesmente **acendem** o led e retornam o programa ao início do ciclo de lampejamento.

```
SetToZero
        bcf  PORTB,LED
        goto  MainLoop
```

Estas duas instruções simplesmente **apagam** o led e retornam o programa a início do ciclo de lampejar.

A sub-rotina Delay

Como descrito anteriormente esta sub-rotina coloca um retardo de cerca de um segundo e pode ser chamada através do programa com instrução **call Delay**.

Vejamos como funciona:

```
Delay
        clrf  Count
        clrf  Count+1

DelayLoop
        decfsz  Count,1
        goto  DelayLoop
        decfsz  Count+1,1
        goto  DelayLoop
        retlw  0

        END
```

Delay e **DelayLoop** são dois label. **Delay** identifica o endereço de início da sub-rotina e será utilizado pela chamada através do corpo do programa principal. **DelayLoop** sera chamado internamente pela sub-rotina e serve como ponto de entrada para o ciclo (do inglês loop) de retardo.

Na pratica o retardo é conseguido executando-se milhares de instruções que não fazem nada!

Este tipo de retardo se chama retardo software ou retardo de programa. É o tipo de retardo mais simples de implementar e pode ser utilizado quando não se deseja que o PIC o faça.

```
clrf  Count
clrf  Count+1
```

CLEA**R** FILE REGISTER zeramento de duas posições da ram reservada anteriormente com a instrução:

```
Count   RES 2
```

Estas duas posições são adjacentes a partir do endereço referenciado pelo label Count.

```
decfsz Count,1
```

A instrução **DEC**REMENT **F**ILE REGISTER, **S**KIP IF **Z**ERO, ou seja, decremente o conteúdo do registro e pule a próxima instrução se for zero (e neste caso Count pula a próxima instrução se o valor devolvido for zero). Se o valor devolvido for diferente de zero executara a próxima instrução:

```
goto DelayLoop
```

Que manda a execução ao ciclo de retardo. Uma vez zero o contador Count ira a próxima instrução:

```
decfsz Count+1,1
goto DelayLoop
```

Que decrementara o registro seguinte até que este chegue a zero. O registro Count+1 em particular será decrementado de um até 256 decrementos de Count.

Quando então Count+1 chegar ao valor zero a instrução

```
return
```

que significa **RETURN** FROM SUBROUTINE determinara a saída da rotina de retardo e retornara a execução da instrução imediatamente após call Delay.

E por fim a diretiva **END** que indica ao compilador o final do código assembler.

No próximo passo compilaremos o código LED_1.ASM e programaremos o PIC com o código gerado pelo compilador assembler.

Lição 1	Passo 5/6	

Compilando um código assembler

Vejamos agora como é possível efetuar na pratica a compilação de um código assembler.

Primeiramente crie em seu disco rígido um diretório de trabalho que de agora em diante ficara armazenado todos os programas do curso. Escolhe-se um nome por exemplo:

C:\PICPRG

(Qualquer outro nome valido de diretório ou drive, é obviamente valido. Bastará substituir no resto do curso todo o referimento a C:\PICPRG pelo nome do drive e diretório escolhido).

Copiemos agora no nosso diretório de trabalho C:\PICPRG ou aquele que você escolheu o arquivo LED.ASM e P16F84.INC. Para fazer isto basta clicar com o mouse sobre o nome do arquivo que se quer salvar em nosso diretório de trabalho, repetindo a operação para ambos os arquivos.

Instalemos agora o software necessário para compilar o nosso programa.

A Microchip coloca disponível no site web o próprio assembler **MPASM** em dupla versão para sistemas operacionais **Microsoft Windows 3.1 / 95** e para ambiente **MS/DOS**. Em seguida faremos referência a versão MS/DOS que igualmente se pode trabalhar com o prompt MS/DOS do Microsoft Windows.

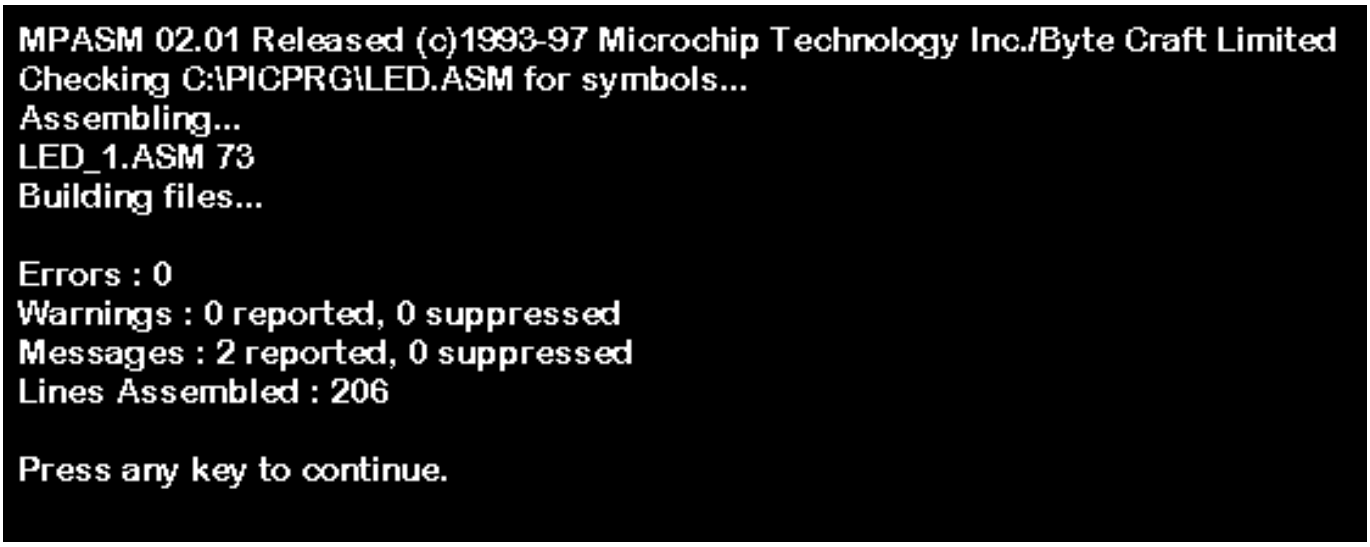
Seguiremos a instrução fornecida na página da Microchip **MPASM.EXE** que contém o seguinte para MS/DOS do assembler. Copiemos então este arquivo para o nosso diretório de trabalho **C:\PICPRG**.

Atenção ! O MPASM é um produto de propriedade da Microchip Technology inc., lembre-se então de ler atentamente as condições de uso indicadas durante a fase de instalação.

Compilaremos o nosso código LED.ASM colocando na frente do prompt do DOS a instrução:

```
MPASM LED.ASM
```

O resultado que deveremos obter no vídeo é o seguinte:



Pressionamos uma tecla como requisita o MPASM e vamos ver que arquivo nos foi gerado. Se tudo deu certo deveremos ver os seguintes novos arquivos:

LED.HEX
LED.LST
LED.ERR
LED.COD

O conteúdo dos arquivos já foi visto no passo 3 então prosseguiremos com programação do PIC utilizando um só arquivo o LED.HEX que contém o arquivo compilado no formato Intel Hex 8.

Programemos o PIC

Para programar o PIC nessa lição faremos referência ao programador YAPP contido no hardware PicTech fornecido com a versão comercial deste curso. A documentação do software para realização tanto do PicTech quanto do YAPP! está descrita em nossa pagina **hardware** de suporte ao curso.

Para a programação do chip com outro tipo de programador deve-se ver a relativa documentação.

Copiamos no nosso diretório de trabalho **C:\PICPRG** o arquivo YAPP.EXE, e começamos a execução do YAPP com o seguinte comando no prompt do DOS:

```
YAPP LED.HEX /COM2 /XT
```

com qual enviamos ao YAPP (conectado, por exemplo, na porta serial COM2) o arquivo LED.HEX contendo nosso programa compilado e programaremos o PIC para funcionar com um cristal de quartzo externo.

Para maior informação sobre a sintaxe do programa YAPP.EXE veja a documentação relativa.

Se a placa PicTech esta corretamente conectada deveremos ver aparecer o opcode hexadecimal das instruções sem nenhum erro.

Terminada a programação deveremos ver o LED 1 lampejar na placa PicTech, como escrito no programa.

Lição 2	Introdução	

Arquitetura interna do PIC

Ao termino desta lição saiba:

- Onde é memorizado o programa
- Onde são memorizados os dados
- O que é uma ALU, Um Acumulador, o Program Counter, o Stack e o Register File.

Conteúdo da lição 2

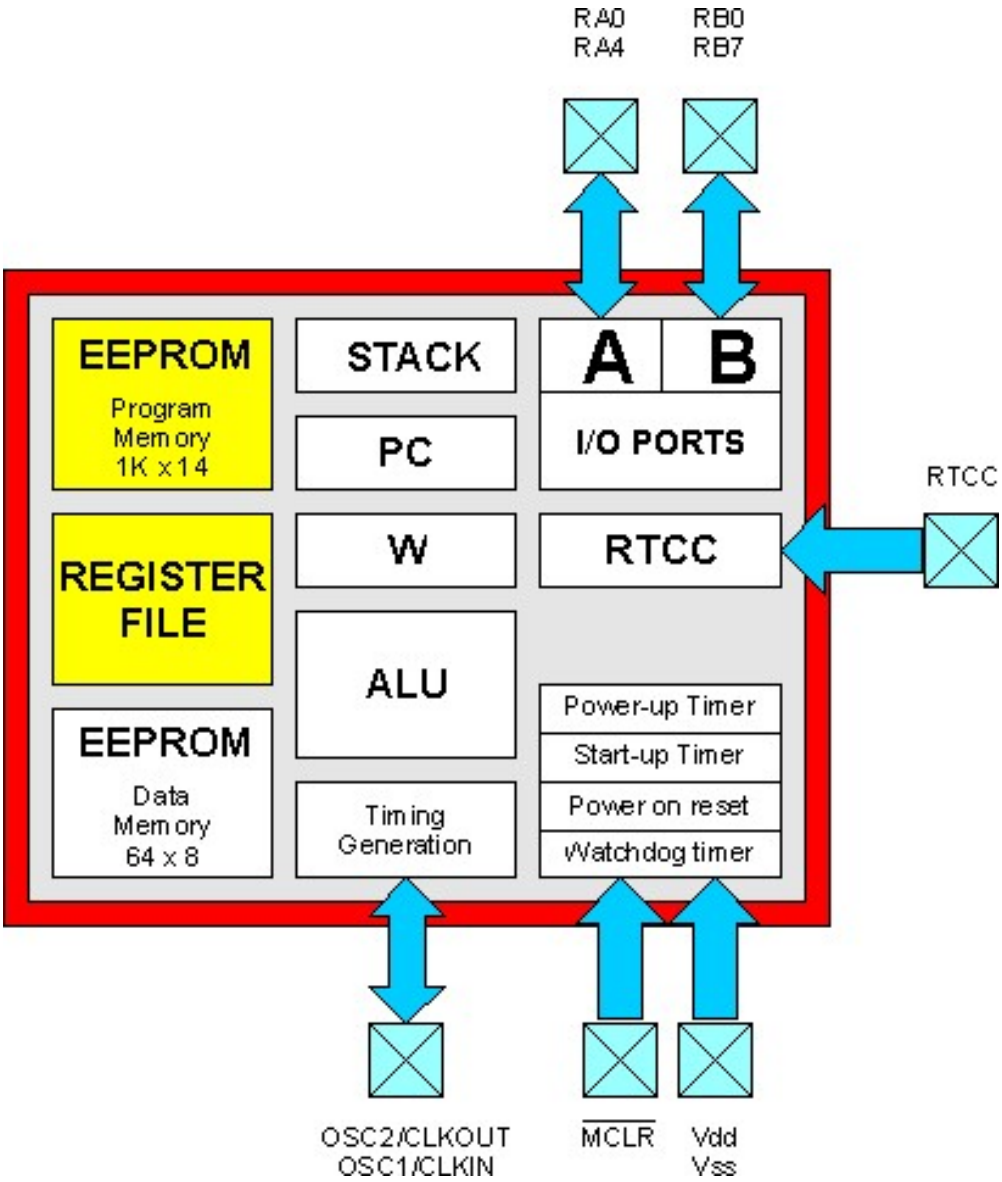
Lição 2 | **Passo 1/4**

A área de programa e o Registrador de arquivo

Depois de ter visto um pouco de pratica, passemos agora a teoria. Iniciaremos agora vendo como é feito um PIC, quais dispositivos ele contém e como interagir entre eles.

Na figura seguinte está ilustrado o esquema de blocos simplificado da arquitetura interna do **PIC16F84** que nos ajudara a entender o que será explicado. As partes evidenciadas em amarelo, é a componente que iremos analisar.

Iniciemos com a memória **EEPROM** e o **REGISTER FILE**.



A EEPROM é uma memória especial, cancelável eletricamente, utilizada no PIC para memorizar o programa a ser executado.

A sua capacidade de memorização é de **1024 posições** e que poderão conter somente um **opcode a 14 bit** ou seja uma instrução básica do PIC. Um programa mais complexo que podemos realizar não poderá ter mais do que 1024 instruções.

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

Par escrever, ler e cancelar estas posições é necessário um dispositivo externo chamado **programador**. Um exemplo de programador é o nosso **YAPP**! ou o **PICSTART-16+©** produto da Microchip ou pode ser outro qualquer disponível no comercio.

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**.

Como devemos lembrar, no código LED.ASM apresentado na primeira lição onde está inserida a primeira diretiva:

```
ORG 00H
```

para indicar o início do programa. Esta diretiva indica de fato que a execução do programa após o reset deve iniciar no endereço 0000H da área de programa.

A instrução que vem logo após a diretiva ORG 00H:

```
bsf     STATUS,RP0
```

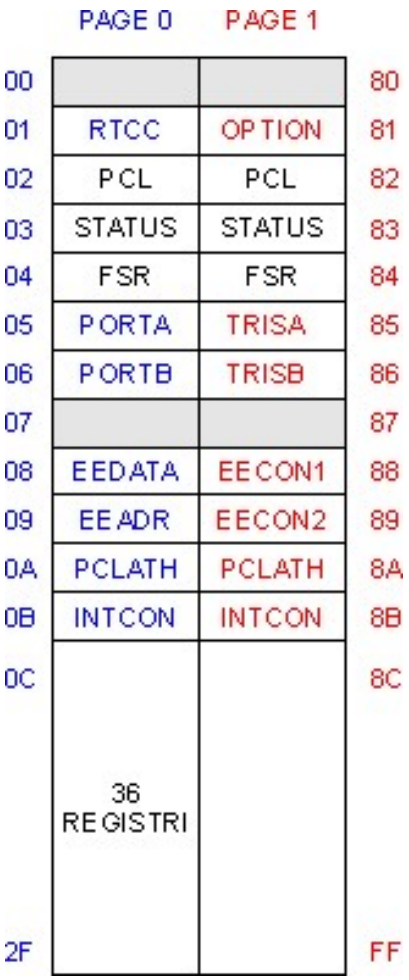
será então, a primeira instrução a ser executada.

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 podemos escrever, ler, ou modificar tranquilamente qualquer endereço do REGISTER FILE no nosso programa a qualquer momento em que for necessário.

A única limitação consiste de que alguns desses registros desenvolvem funções especiais pelo PIC e não podem ser utilizados para outra coisa a não ser para aquilo a qual eles estão reservados. Estes registros encontram-se nas posições base da área de memória RAM segundo o que está ilustrado em seguida.

A locação de memória presente no REGISTER FILE são endereçadas diretamente em um espaço de memória que vai de **00H** a **2FH** um total de 48 bytes, denominada **pagina-0**. um segundo espaço de endereçamento denominado **pagina-1** vai de **80H** a **AFH**. Para acessar esse segundo espaço é necessário recorrer a dois bits auxiliares **RP0** e **RP1** segundo a modalidade que iremos explicar mais adiante.



As primeiras **12 posições** da pagina-0 (de **00H** a **0BH**) e da pagina-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 **36 posições** na **pagina-0** pode ser endereçada de **0CH** a **2FH** podemos aqui utilizar livremente pelo nosso programa para memorizar variáveis, contadores, etc.

No nosso exemplo LED.ASMa diretiva:

```
ORG 0CH
```

indica o endereço de início da área de dados utilizada pelo nosso programa.

A diretiva que segue:

```
Count      RES 2
```

reserva um espaço de duas posições, que o programa utilizará para memorizar o contador de retardo da sub-rotina Delay.

Os registros especiais do PIC serão utilizados com muita frequência nos programas.

Por exemplo, se fizermos a cópia dos registros especiais **TRISA** e **TRISB**, para definir qual linha de I/O será entrada e qual será saída. O mesmo estado lógico da linha de I/O 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 aritméticas e logica.

É necessário conhecer, portanto, 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, no arquivo P16F84.INC (que como recordamos estava incluído no código LED.ASM com a diretiva **INCLUDE**) a Microchip inseriu uma lista de nomes que identificam univocamente qualquer registro especial e a qual está associado o endereço correspondente na área do REGISTER FILE.

Se, por exemplo, quisermos definir toda a linha do PORTB do PIC em escrita agindo sobre o **TRISB**, podemos escolher e referenciar diretamente o registro com o seu endereço:

```
movlw BT00000000T
movwf 06H
```

ou então, referenciar o mesmo registro com o seu nome simbólico:

```
movlw BT00000000T
movwf TRISB
```

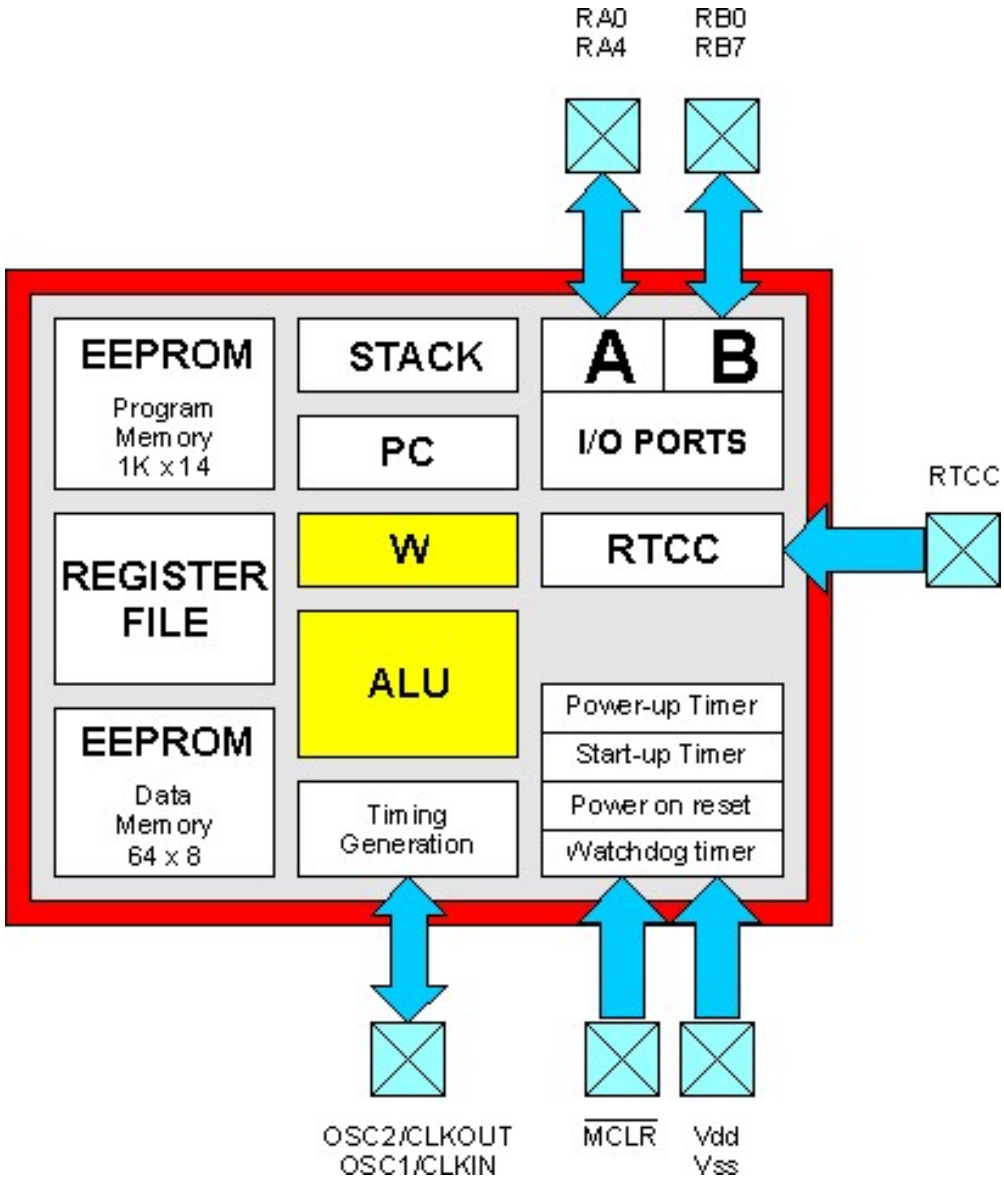
tendo que ter a certeza de ter inserido a diretiva INCLUDE "P16C84.INC" no nosso código.

Na próxima lição veremos um outro componente interno do PIC denominado de **ALU** e o **registro W** conhecido anteriormente com o nome de **acumulador**.

Lição 2	Passo 2/4	

A ALU e o registro W

Iremos agora ilustrar outros dois componentes fundamentais na arquitetura do PIC, a **ALU** e o **registro W** ou **acumulador**.



A **ALU** (siglas de **A**rithmetic and **L**ogic **U**nit, ou seja, Unidade Lógica e Aritmética) é a componente mais complexa doPIC por conter todos os circuitos destinados a desenvolver as funções de cálculo e manipulação de dados durante a execução de um programa.

A **ALU** é uma componente presente em todos os microprocessadores e dessa depende diretamente a capacidade de cálculo do micro em si.

A **ALU** do **PIC16F84** 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 é notavelmente superior em detrimento da complexidade dos circuitos internos de acessória e consequentemente do espaço ocupado.

Diretamente conheço a ALU como **registro W** denominado antes de **acumulador**. Este registro consiste de uma locação de memória destinada a conter um só valor de 8 bits.

A diferença entre o registro W e outras posições de memória consiste no fato de que, por referenciar o registro W, a ALU não pode fornecer nenhum endereço, mas podemos acessa-los diretamente.

O registro W será utilizado especificamente no programa pelo PIC.

Façamos um exemplo prático. Suponhamos querer colocar na locação de memória **0CH** do **REGISTER FILE** o valor **01H**. Procurando entre as instruções do PIC veremos rápido que não existe uma única instrução capaz de efetuar esta operação mas deveremos necessariamente recorrer ao acumulador e usar duas instruções em sequência.

Vejamos porque:

Como dissemos anteriormente, o opcode de uma instrução não pode exceder aos **14 bits** e assim teremos:

8 bits para especificar o valor que queremos colocar na locação de memória,
7 bits para especificar em qual locação de memória queremos inserir o nosso valor,
6 bits para especificar qual instrução queremos usar.

teremos um total de **8 + 7 + 6 = 21 bits**.

Devemos então recorrer a duas instruções, ou seja:

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

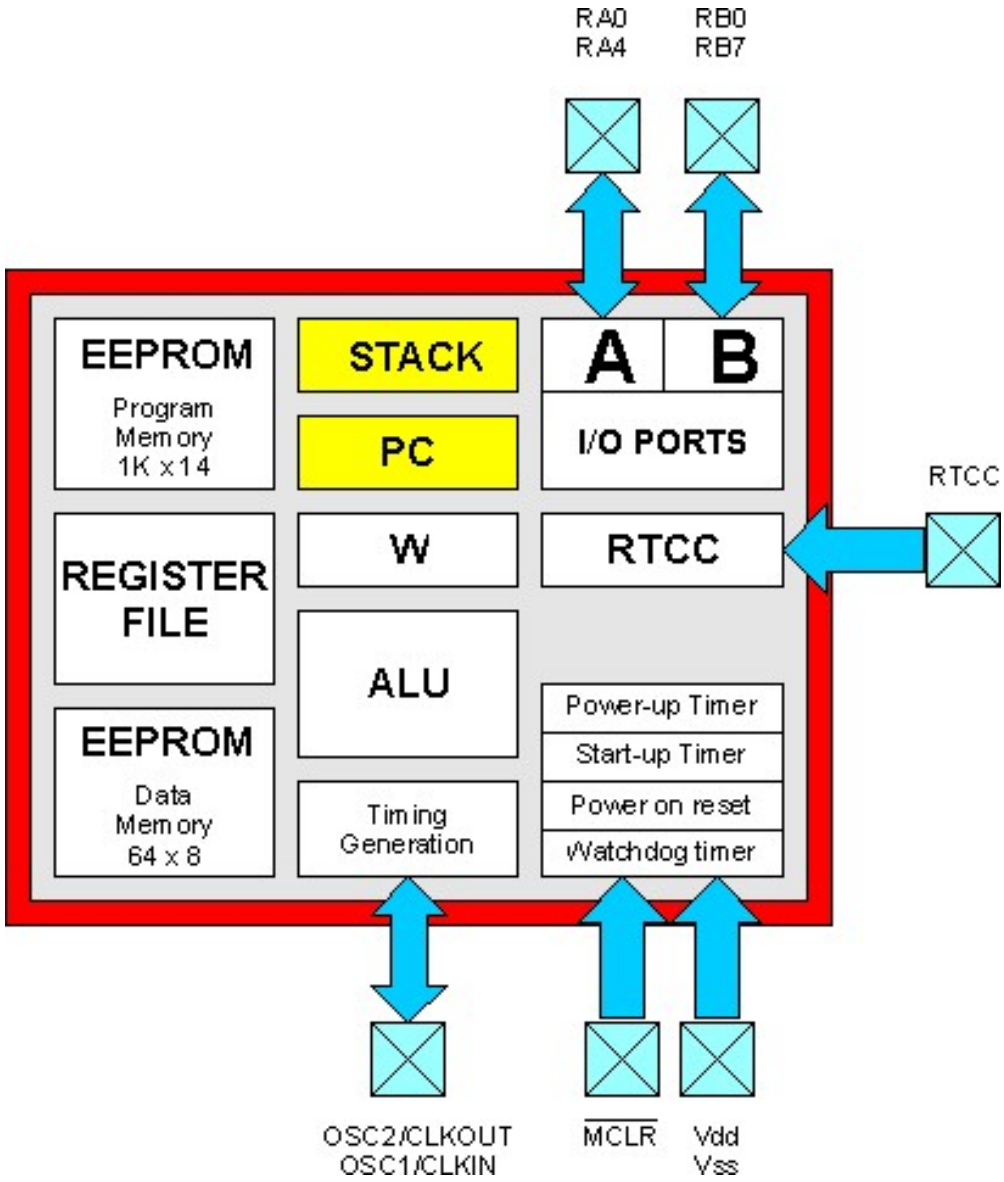
que a primeira colocará no **registro W** o valor **01H** com a instrução **MOV**e **L**iteral to **W** e depois "moveremos" para locação 0CH com a instrução **MOV**e **W** para **F**.

No próximo passo encontraremos o Program Counter e o Stack que facilitara a entender como funciona a instrução de salto do PIC.

Lição 2	Passo 3/4	

O contador de programa e o Stack

Nesta lição analisaremos o funcionamento do **Program Counter** e do **Stack** dois componentes importantes para a compreensão da instrução de salto e chamada a sub-rotina.



Como visto na lição anterior, o **PIC16F84** inicia a execução do programa a partir do **Reset Vector,** ou seja**,** da locação dememória **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.

Sabemos obviamente que não é bem assim e qualquer sistema a microprocessador ou linguagem de programação dispõe de instrução de desvio, ou seja, instruções capazes de modificar o fluxo de execução do programa.

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.

Façamos um exemplo:

```
        ORG     00H

Point1
        movlw   10
        goto    Point1
```

No reset o PIC seguira a instrução **movlw 10** memorizada na locação **0000H** que colocara no acumulador o valor decimal 10. Onde então passara à executar a próxima **goto Point1**. Esta instrução determinará um desvio incondicionado para locação de memória especificada pelo label **Point1** ou seja de novo para locação **0000H**.

O programa não fará outra coisa senão a de executar um ciclo infinito seguindo continuamente as instruções especificadas.

Durante este ciclo, para determinar qual é a próxima instrução a ser seguida, o PIC utiliza um registro especial denominado **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 início da execução no endereço 0000H, e cada instruçãoterá um incremento de um para poder passar para próxima instrução.

A instrução **goto** permite a colocação de um novo valor no **Program Counter** e consequentemente desvia-a a uma locação qualquer da área de programa do PIC.

Uma outra instrução muito interessante é o **call,** ou seja**,** a chamada a sub-rotina.

Esta instrução funciona de maneira muito similar ao **goto** com a única diferença que, a primeira, desvia para uma locação de memória especificada e continua a execução do programa, enquanto o **call** desviara o programa para uma sub-rotina especificada e executara a mesma, e retornara a execução da instrução imediatamente após a chamada **call**, ovalor imediatamente após a chamada call será armazenado em uma área particular da memória denominada **Stack**.

Vejamos melhor com um exemplo:

```
        ORG     00H

Point1
        movlw   10
        call    Point2
        goto    Point1

Point2
        movlw   11
```

```
        return
```

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, a próxima locação ao call. Passa então a executar a instrução **movlw 11**, memorizada em correspondência ao label **Point2**. E neste ponto encontra uma nova instrução o **return** que, como podemos deduzir de seu nome, permite o "RETORNO", ou seja retorne à execução da instrução imediatamente após o **call**.

Esta operação é denominada de: "**chamada a sub-rotina**", 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 **Program Counter**.

A palavra **stack** em inglês significa "**cascata, 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 **L**ast **I**n **F**irst **O**ut,em que o último elemento armazenado (last in) deve necessariamente ser o primeiro a sair (last out).

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**.

Vejamos um outro exemplo:

```
        ORG     00H

Point1
        movlw   10
        call    Point2
        goto    Point1

Point2
        movlw   11
        call    Point3
        return

Point3
        movlw   12
        return
```

No exemplo acima a rotina principal **Point1** promove a chamada do primeiro **call** para sub-rotina **Point2**, a sub-rotina **Point2** chama outra sub-rotina no caso **Point3**, este último por sua vez, encontra um **returm** 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** procurara pelo **return** correspondente ao primeiro e assim por diante. Se diz então que o **call** é “nidificate”, ou seja, um dentro do outro.

O **PIC16F84** dispõe de um **stack de 8 níveis**, ou seja, um Stack que consegue armazenar no máximo 8 chamadas

à

subrotina.

É importante assegurar-se, durante a formulação de um programa que, se tenha sempre uma instrução **returm** em correspondência a um **call** para evitar o perigo de desalinhamento do stack que em execução pode gerar erros que dificilmente encontraremos.

Na próximo passo modificaremos o nosso código LED.ASM para entender melhor o que foi dito até aqui.

Lição 2	Passo 4/4	

Realizando as "Luzes em sequência"

Faremos agora uma reelaboração do código LED.ASM apresentado na primeira lição fazendo-o realizar um lampejador de quatro led's. E o novo código modificado se chamará SEQ.ASM.

O circuito a ser realizado está representado no seguinte arquivo no formato Acrobat Reader (10Kb): example2.pdf substancialmente equivalente ao circuito apresentado na primeira lição, com a única diferença que agora a quantidade de led's conectados serão quatro antes era um.

As linha de I/O utilizadas serão **RB0** para o primeiro led, **RB1** para o segundo, **RB2** para o terceiro **RB3** para o quarto.Vamos configurar todas em escrita no início do programa trocando as instruções:

```
movlw 11111110B
movwf TRISB
```

para

```
movlw 11110000B
movwf TRISB
```

em que os quatro bits menos significativos, corresponde a linha RB0,1,2,3 foram colocados a zero para definir esta linha em escrita.

Na área de memória do REGISTER FILE (que no código inicia com a diretiva ORG 0CH) além dos dois bytes referenciados pelo label **Count**, reservaremos mais um byte com o label **Shift** que utilizaremos para determinar a sequência de funcionamento dos led's. A diretiva a ser inserida é:

```
Shift   RES     1
```

Antes de iniciar o ciclo principal (label **MainLoop**) vamos inicializar um novo registro **Shift** a **00000001B** com a seguinte instrução:

```
movlw   00000001B
movwf   Shift
```

Neste ponto, no ciclo principal do nosso programa, vamos tratar de transferir o valor memorizado no registro **Shift** para o **PortB** obtendo então o inicio de funcionamento do primeiro led, a instrução será a seguinte:

```
movf    Shift,W
movwf   PORTB
```

que então efetuara o giro para esquerda do valor contido no Shift de um bit, com a seguinte instrução:

```
bcf     STATUS,S
rlf     Shift,F
```

A primeira instrução serve para zerar o bit CARRY do REGISTRO DE STATUS que vamos analisar na próxima lição. **RLF** **R**otate **L**eft **F** through Carry (rotaciona para esquerda o bit do carry) desloca um bit para esquerda o valor memorizado no registro Shift inserindo na posição ocupada pelo bit 0 o valor do bit do Carry(que como dissemos veremos em seguida). Para fazer com que este bit seja sempre zero terá que ser executada antes da **RLF** a instrução **BCF STATUS,C** para zerar este bit.

Neste ponto o registro **Shift** será **00000010B**, onde, no próximo ciclo, uma vez transferido esse valor para o PortB se obterá o apagamento do LED1 e o acendimento do LED2 e assim por diante nos ciclos sucessivos.

Quando o bit 4 do **Shift** for a 1, então todos os quatro leds estiveram acesos pelo menos uma vez e tornara a iniciar do led 1. Na instrução seguinte veremos este tipo de controle:

```
btfsc   Shift,4
swapf   Shift,F
```

A instrução **btfsc Shift,4** controla exatamente até que o bit 4 do registro Shift vale 1. Depois executa a próxima instrução **swapf Shift,F**, e continua.

A instrução **swap** (do inglês "troca") na pratica troca o quarto bit mais significativo contido no registro **Shift**, pelo quartobit menos significativo. Do valor inicial do registro **Shift** igual a **00010000** obtido através do ciclo de repetição **MainLoop** se obtém o valor **0000001** que, na pratica, faz acender o primeiro led.

Lição 3	Introdução	

Introdução aos periféricos

Ao termino desta lição saiba:

- Como funciona e como se programa a linha de I/O

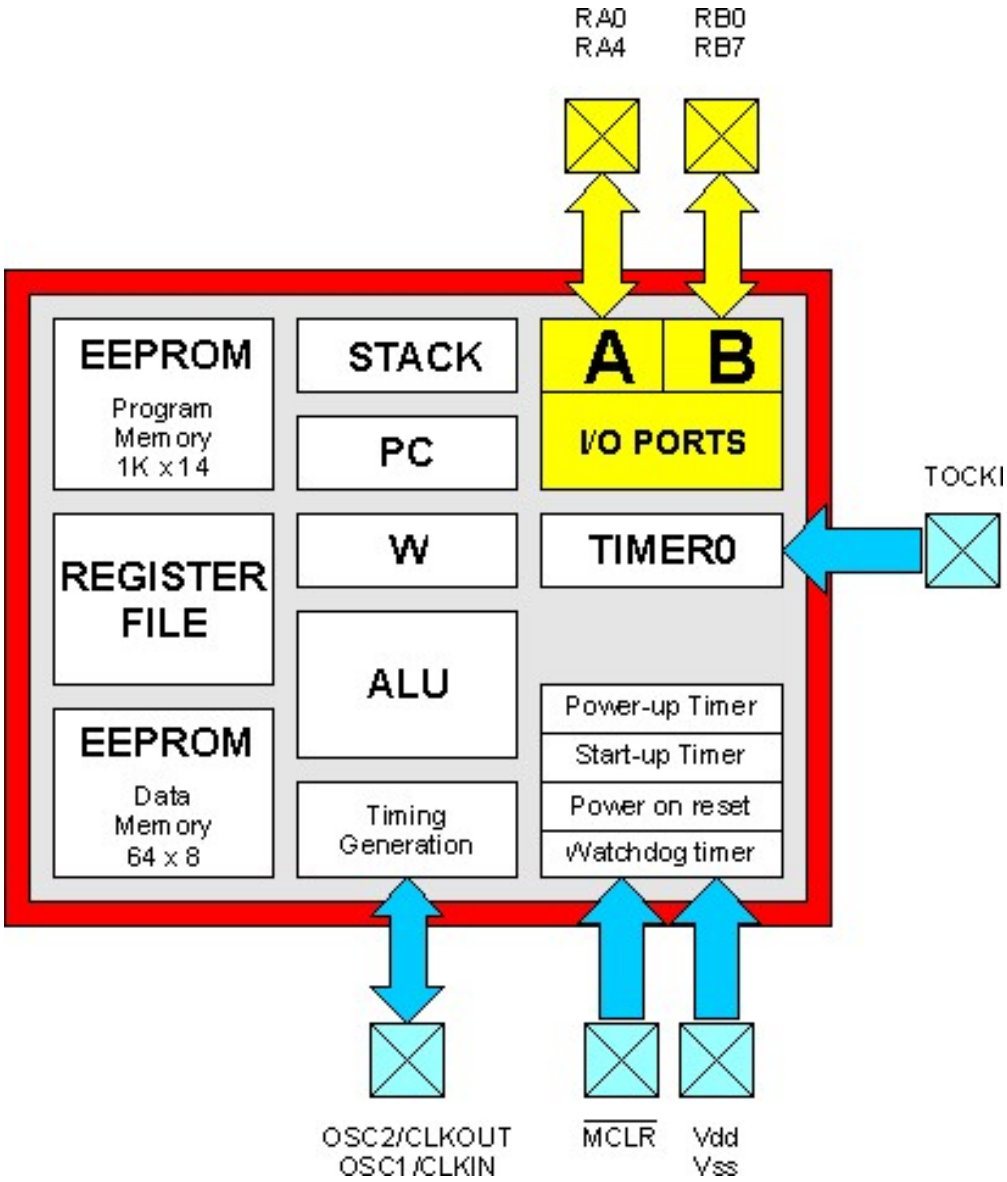
Conteúdo da Lição 3

1. A porta A e B
2. Estado de saída da linha de I/O
3. Entrada de teclas

Lição 3	Passo 1/3	

A porta A e B

O **PIC16C84** dispões de um total de **13 linhas** de I/O 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 **PIC16F84** que prevê um controle de dados de no máximo 8 bits.

Para o controle da linha de I/O 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.

Os registros **TRIS A** e **B** , determinarão o funcionamento em entrada ou em saída da mesma linha, e o registro **PORT A** e **B** determinarão o status da linha na saída ou reportarão o status da linha em entrada.

Todos os bits contidos nos registros mencionados correspondem univocamente a uma linha de I/O.

Por exemplo o **bit 0** do registro **PORTA** e do registo **TRIS A** correspondem a linha **RA0,** o **bit 1** a linha **RA1** e assim por diante.

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 irá o **bit 0** do registro **PORTA** determinara o estado logico de tal linha (0 = 0volts, 1 = 5 volts).

Se o **bit 0** do registrador **TRISA** for definido como **um**, a linha **RA0** será configurada como **linha de entrada**, portanto o estado lógico em que a linha **RA0** será colocada pelo circuito externo refletirá no estado do **bit 0** do o registro **PORTA.**

Façamos um exemplo prático, imaginemos querer conectar um led sobre a linha RB0 e uma chave sobre a linha RB4, o código a se escrever será o seguinte:

```
movlw   00010000B
tris    B
```

onde aqui será colocado a 0 o bit 0(linha RB0 em escrita(saída), e a 1 o bit 4 (linha RB4) em entrada. recorde-se de tal propósito que na notação binaria do assembler o bit mais a direita corresponde com o bit menos significativo por isso o bit 0.

Para acender o led devemos escrever o seguinte código:

```
bsf     PORTB,0
```

Para apaga-lo:

```
bcf     PORTB,0
```

Para lermos o estado da chave conectada a linha RB4, o código será:

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

No próximo passo analisaremos o estado que controlara a linha de I/O.

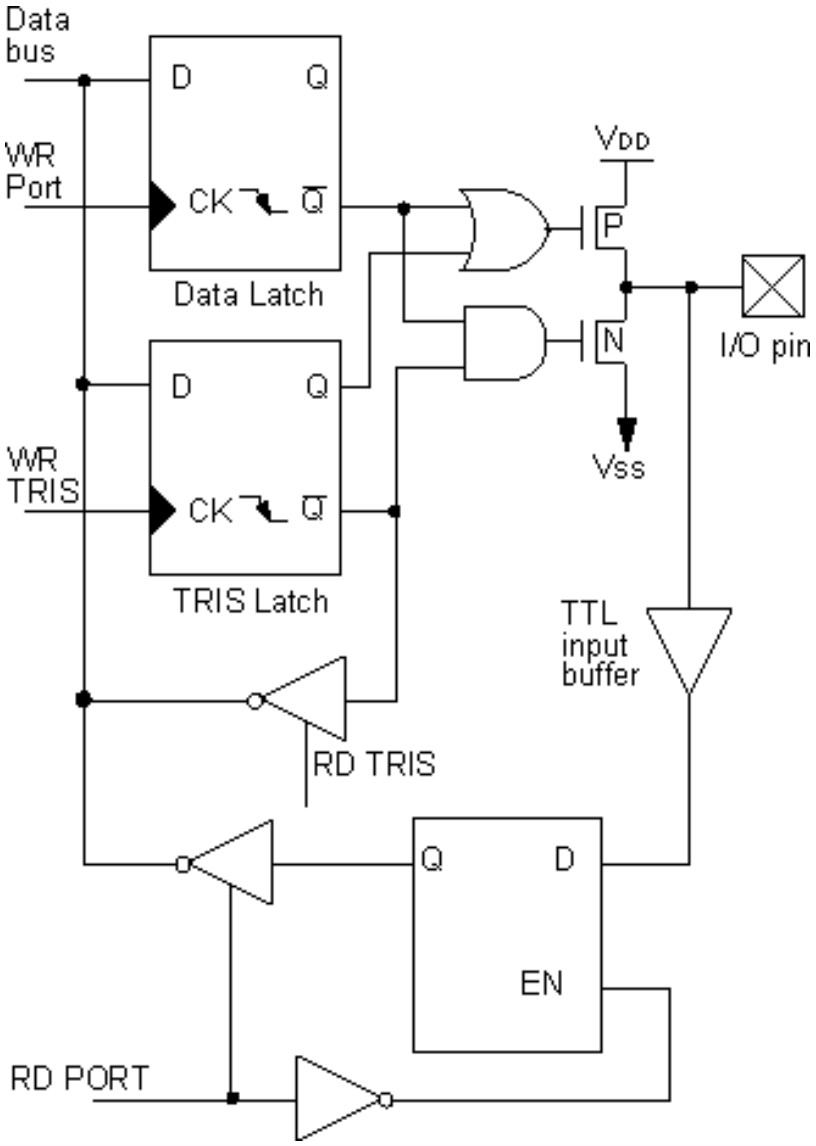
Lição 3 | **Passo 2/3**

Estado de escrita da linha de I/O

Por ser o **PIC** mais maleável as diversas exigências de utilização, a Microchip tem implementado diversas tipologias de status de escrita para a linha de I/O. Tendo então dois grupos de pinos a qual o comportamento é ligeiramente diferenciado do outro grupo. Conhecendo melhor o funcionamento dos diversos status de escrita podemos desfrutar melhor das características e otimizar melhor o nosso projeto.

Estado de escrita das linhas RA0, RA1, RA2 e RA3

Iniciaremos do grupo da linha **RA0**, **RA1**, **RA2** e **RA3** na qual representamos, na figura seguinte, o esquema do estado de escrita extraído do datasheet da Microchip:



Como dito no passo anterior, a configuração de uma linha como entrada ou saída depende do estado do bit no registro **TRIS** (**TRISA** para o **PORTA** e **TRISB** para o **PORTB**).

Pegaremos como exemplo a linha RA0 e analisaremos o funcionamento do estado de saída seja quando a mesma funciona em entrada ou quando em saída.

Funcionamento em entrada

Para configurar a linha **RA0** em entrada, devemos colocar em 1 o **bit 0** do registro **TRISA** com a instrução:

```
bsf     TRISA,0
```

Esta instrução determinara uma comutação à 1 do estado logico do **flip-flop** "**D-latch"** indicado no bloco, com o nome **TRIS latch**. O registro TRIS, está presente em todas linhas de I/O e o estado logico em que se trava depende estritamente do estado logico do relativo bit neste registro (**TRIS)** (ou melhor dizendo todos o bit's do registro **TRIS** é fisicamente implementado com um TRIS latch).

A saída **Q** do **TRIS latch** é conectada a entrada de uma porta lógica do tipo **OR**. Isto significa que, independente, do valor presente a outra entrada, a saída da porta OR estará sempre em 1 em quanto uma de suas entradas vale 1(veja na tabela verdade). E, nesta condição, o **transistor P** não conduz e mantem a linha RA0 desconectada do positivo da alimentação.

Do mesmo modo a saída negativa **Q** do **TRIS** latch e conectada a entrada de uma porta **AND** onde a saída desta estará sempre em 0 em quanto uma de suas entradas vale 0 (veja tabela verdade). E nesta condição em que o **transistor N** não conduz mantendo a linha RA0 desconectada da massa. O estado logico da linha RA0 dependerá exclusivamente do circuito externo a que o conectarmos.

Aplicando 0 ou 5 volts ao pino RA0, será possível lermos o estado presente no circuito externo a entrada do bloco representado por **TTl input buffer** e do latch de entrada.

Funcionamento em saída

Para configurar a linha de RA0 em saída, devemos colocar em 0 o **bit 0** do registro **TRISA** com a instrução:

```
bcf     TRISA,0
```

Esta determina a comutação para 0 da saída **Q** do TRIS latch (e para 1 a saída Q negativa). E neste estado o valor da saída da porta OR e AND depende exclusivamente do estado de saída do **Q negativo** do **Data Latch**. Como para o TRISLatch, em que o DataLatch depende do estado de um bit em um registro, particularmente do registro **PORTA**. A sua saída negativa será enviada para entrada das duas portas logicas **OR** e **AND** e que estão diretamente sobre a base do **transistor P** e **N**.

Se colocarmos em 0 o bit 0 do registro PORTA com a instrução:

```
bcf     PORTA,0
```

obtermos a condução do **transistor N** e, portanto, irá a 0 a linha RA0. Se ao invés colocarmos em 1 o bit 0 com ainstrução:

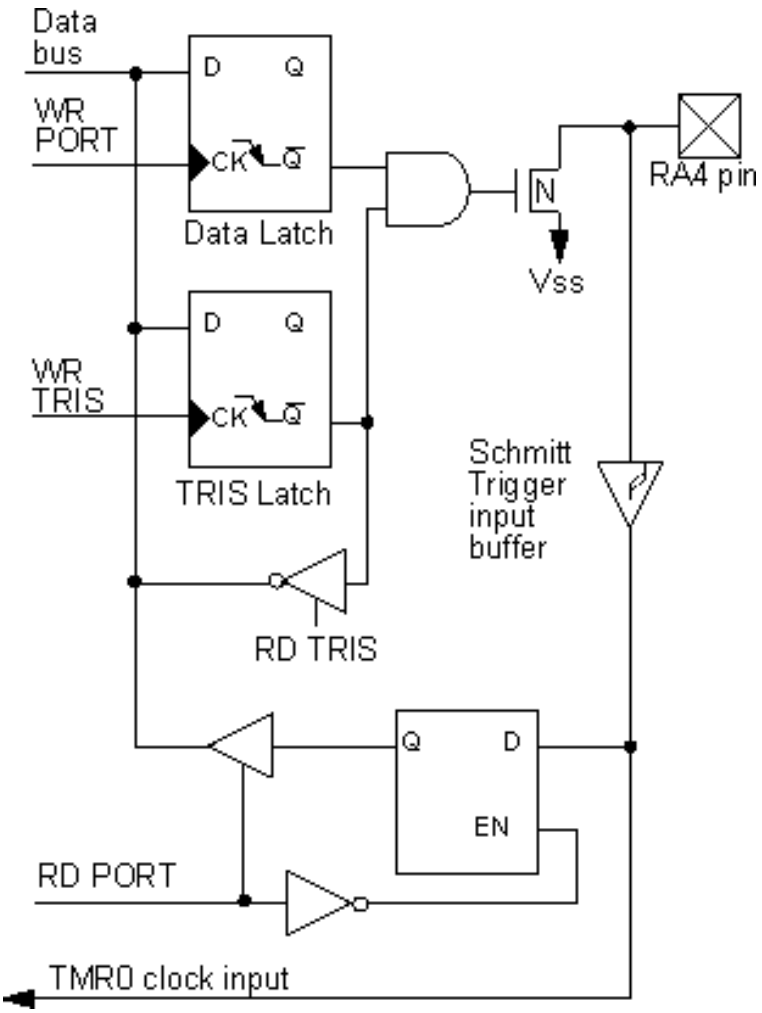
```
bsf     PORTA,0
```

obteremos a condução do **transistor P** e, portanto, ira a +5 volts a linha RA0. Nesta condição será sempre possível revero valor enviado sobre a linha através do circuito de entrada.

Estado de saída da linha RA4

Analisaremos agora o funcionamento do estado de saída da linha **RA4** que é diferente de todas as outras linhas de I/O enquanto compartilha o mesmo pino do PIC16c84 com o **TOCKI** o qual iremos analisar no próximo passo.

Na figura seguinte, está descrito o esquema de blocos do estado de saída extraído do datasheet Microchip:



A lógica de comutação é substancialmente idêntica ao grupo das linhas RA0 a RA3 com exceção da ausência da porta **OR** e do **transistor P**, ou seja, de todos os circuitos que permitem a ligação ao positivo, da linha RA4. Isto significa em termos práticos, que quando a linha RA4 está programada em saída poderá assumir um nível que dependera do circuito externo pois na realidade não está conectada ao positivo e sim desconectada. Este tipo de circuito de saída chama-se "**coletor aberto**" e é útil para aplicações em que é necessário compartilhar uma mesma ligação com mais pinos de saída, ouque se tenha a necessidade de colocar em alta impedância uma linha de saída e podendo assim reprograma-la como linha de entrada.

Se quisermos tornar seguro que a linha RA4 vá a 1 devemos conectar externamente um resistor de pull-up, ou seja, um resistor contado ao positivo da alimentação.

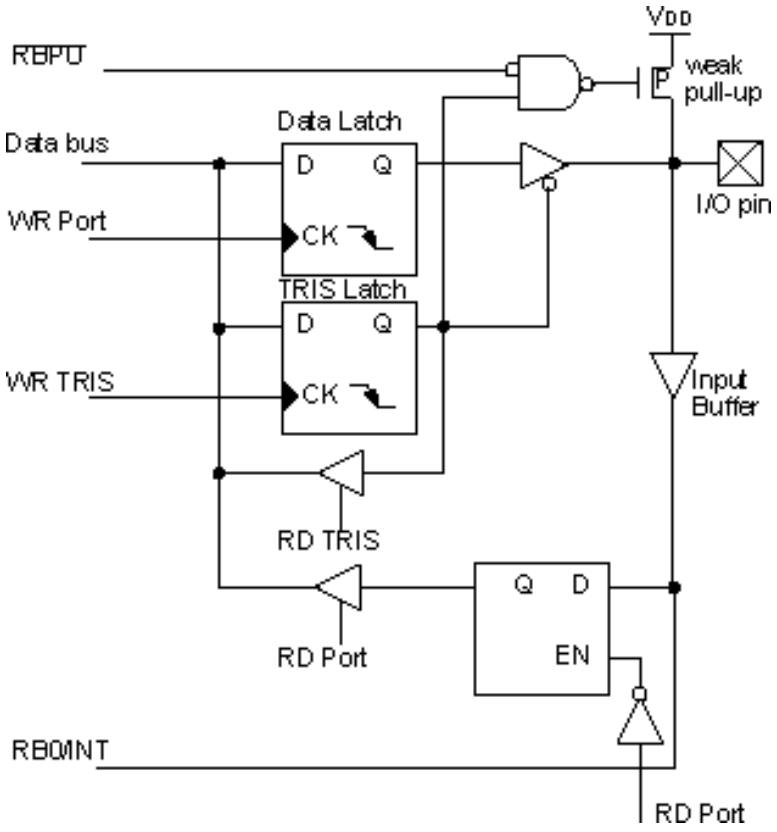
Veremos em seguida a utilização da linha indicada no esquema acima TMR0 clock input.

Estado de saída das linhas RB0, RB1, RB2 e RB3

Onde que para este grupo de linhas a lógica de comutação permanece inalterada. Esta dispõe de um circuito a mais, o **weak pull-up** ativável quando a linha for programada em entrada.

A entrada de fato, como explicado anteriormente, a linha vem completamente desligada do PIC. O estado da linha depende então exclusivamente do circuito externo. Se o circuito é do tipo de coletor aberto ou simplesmente é constituído de uma simples chave que, quando pressionada, conecta a massa a linha de I\O, é necessário inserir um resistor de pull-up vinda do positivo para tornar seguro quando a chave for solta o nível voltar a uma condição lógica 1estavel sobre a linha de entrada. O circuito de weak pull-up permite evitar o uso do resistor de pull-up e é possível de ser ativado agindo sobre o bit **RBPU** do registro **OPTION**.

Na figura seguinte está representado o esquema de blocos do estado de sida extraído do datasheet Microchip:



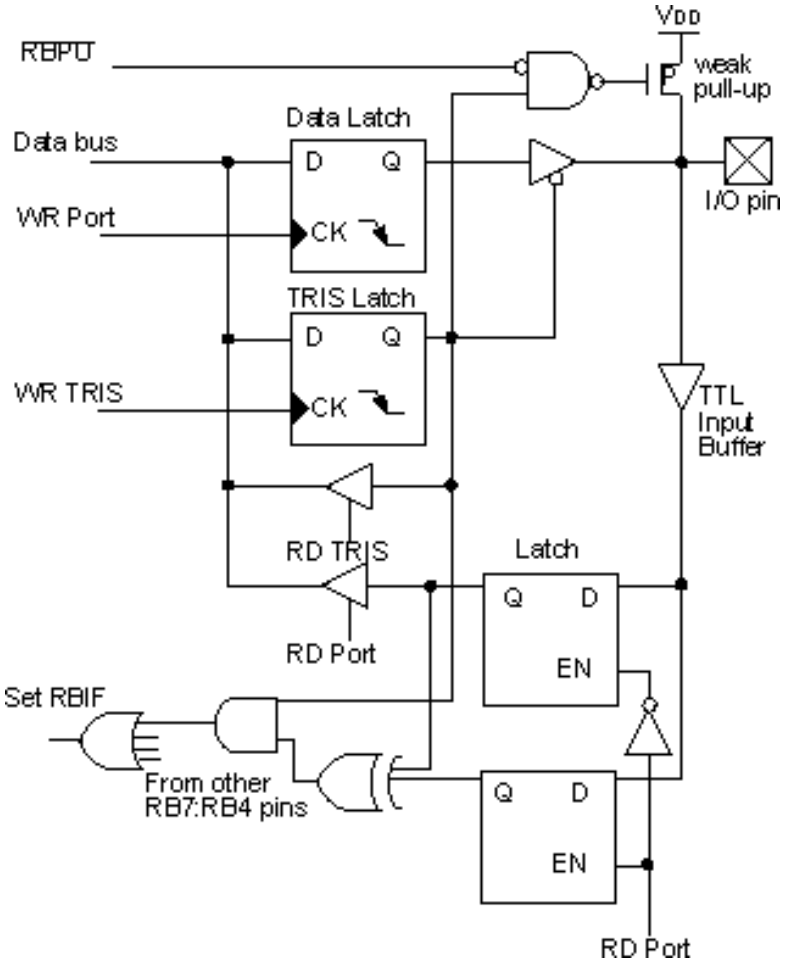
Além disso a linha **RB0** sozinha, apresenta uma característica muito particular. Esta, quando for configurada como linha de entrada, pode gerar, em correspondência a uma troca de estado lógico, uma **interrupt**, ou seja, uma interrupçãoimediata do programa em execução e uma chamada a uma sub-rotina especial denominada **interrupt handler**. Mas dissofalaremos em seguida.

Estado de saída das linhas RB4, RB5, RB6 e RB7

O circuito de comutação deste grupo de linhas é idêntico ao grupo RB0 a 3. Esta linha dispõe também de um circuito de weak pull-up. E mais, com respeito a linha RB0 - 3 tem a vantagem de poder revelar variações de estado sobre qualquer linha e gerar uma interrupção da qual falaremos na próxima lição.

Na figura seguinte esta reproduzido o esquema de blocos do estado de saída extraído do data sheet Micrchip.

Lição 3 Passo 2



Lição 3 | **Passo 3/3**

Entrada de teclas

Depois de ter realizado, no exemplo anterior, as luzes em sequência usando a linha de RB0 a RB3 como linha de saída, vejamos agora como se pode realizar uma entrada com teclas configurando a linha de RB4 a RB7 como linha de entrada.

Para isto usaremos as quatro chaves **SW1**, **SW2**, **SW3** e **SW4** disponíveis sobre a placa **PicTech**.

Cada uma destas chaves conectadas ao +5v das linhas de entrada normalmente mantidas em zero por um resistor (R1 aR4). Se conectarmos por exemplo o pino 10 do PIC16F84 a chave SW1, esta linha será mantida em 0 até não ser pressionada a tecla correspondente SW1 que promovera a mudança da linha para 1.

Para exemplo, realizaremos um programa que acendera qualquer um dos led de **D1**, **D2**, **D3** e **D4** em correspondência com o pressionamento das teclas **SW1** a **SW4**.

O circuito a ser realizado está representado no seguinte arquivo no formato Acrobat Reader (12Kb): example3.pdf.

O código do exemplo é descrito no arquivo INPUT.ASM.

Analisaremos agora o funcionamento.

A parte inicial do programa segue as mesmas funções efetuadas no exemplo anterior e é em particular as instruções:

```
    movlw 11110000B
    movwf TRISB
```

configurando a linha RB0 a RB3 em saída para conexão com os led's e as linhas RB4 a RB7 em entrada para a conexão com as quatro chaves.

```
MainLoop
    swapf PORTB,1
    goto MainLoop
```

efetua simplesmente um loop continuo a qual será executada a instrução **Swapf** que troca o conteúdo do bit **RB0-RB3** com o bit **RB4-RB7**. E deste modo o estado da linha de entrada conectada as quatro chaves serão mudadas continuamente sobre a linha conectada aos led's obtendo-se o acendimento de um led para cada chave.

Lição 4	Introdução	

O contador TMR0 e o PRESCALER

Ao termino desta lição saiba:

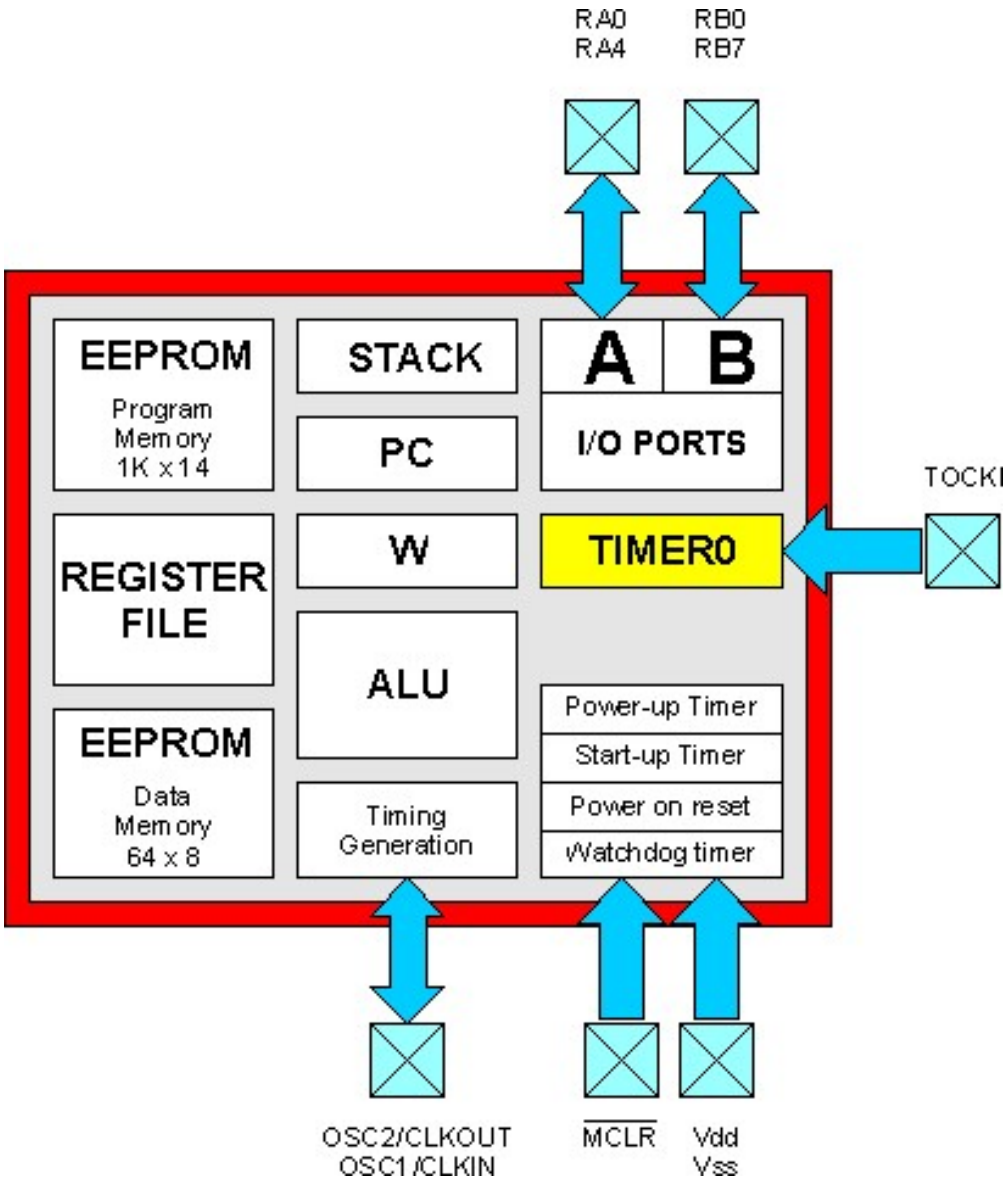
- Qual a utilidade do contador TMR0
- Para que serve e como se programa o PRESCALER

Conteúdo da lição 4

Lição 4	Passo 1/2	

O registro contador TMR0

Vejamos agora o que é e como funciona o registro **TMR0.**



O registro TMR0 é um contador, ou seja, um registro particular, na qual, seu conteúdo vê-se incrementado com cadenciaregular e programada diretamente pelo hardware do PIC. Na pratica, a diferença de outro registro, é que, o TMR0 não mante inalterado o valor que é memorizado, mas o incrementa continuamente, se por exemplo, escrevermos nele o valor 10 com a instrução:

```
movlw     10
movwf     TMR0
```

Após um tempo par de quatro ciclos de máquina, o conteúdo do registro começa a ser incrementado em +1 ou seja 11, 12, 13 e assim por diante com a cadencia constante e independente da execução do resto do programa.

Se por exemplo, após ter colocado um valor no registro TMR0, executarmos um loop infinito

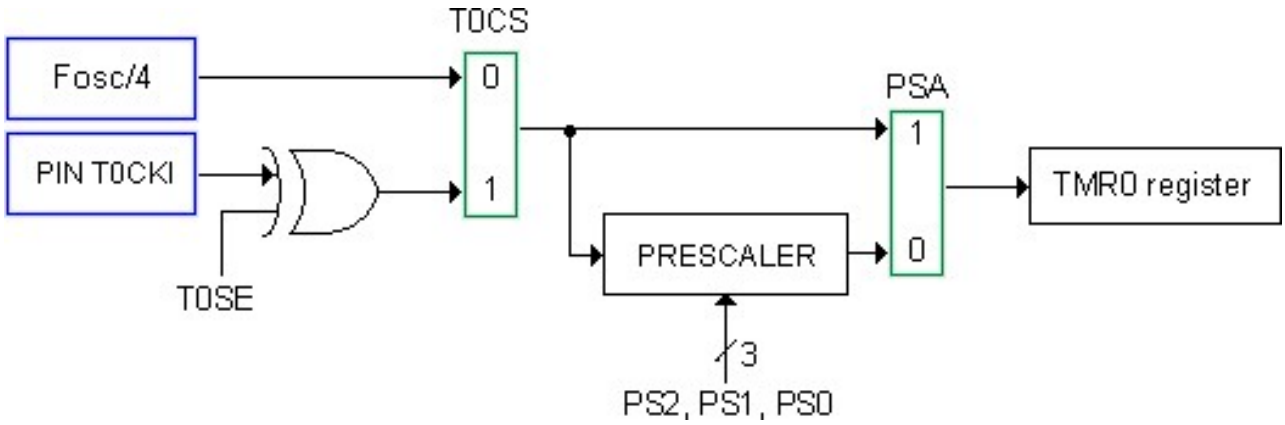
```
movlw   10
movwf   TMR0

loop
        goto loop
```

o registro TMR0 será incrementado pelo hardware interno do PIC durante a execução do loop.

Uma vez atingido o valor 255 o registro TMR0 será zerado automaticamente retornando então a contagem, mas não do valor originalmente imposto, mas do zero.

A frequência é diretamente proporcional a frequência de clock aplicada ao chip e pode ser modificada programando-se oportunamente os seus bits de configuração.

Na figura seguinte está representada a cadeia de blocos interno do PIC que determina o funcionamento do registro TMR0.



O bloco **Fosc/4** e **T0CKI** representados em **azul** representam as duas possíveis fontes de sinal para o contador **TMR0**.

Fosc/4 é um sinal gerado internamente no PIC pelo circuito de clock e é par na frequência de clock dividida por quatro.

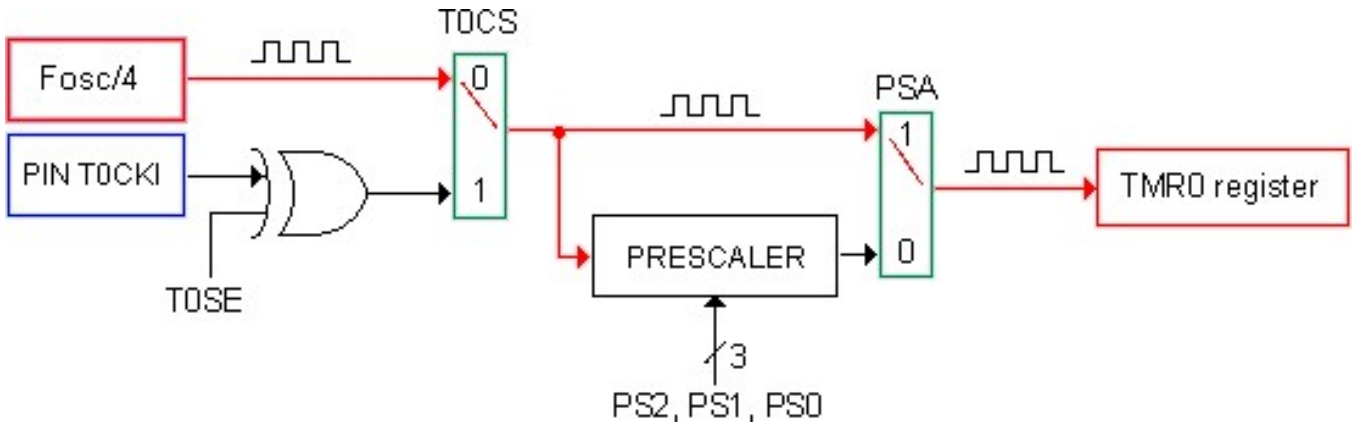
T0CKI é um sinal gerado de um eventual circuito externo e aplicado ao pino **T0CKI** correspondente ao pino 3 no PIC 16C84.

Os blocos **T0CS** e **PSA** descritos em **verde** são dois comutadores de sinal na qual estão representando um dos dois tiposde sinal de entrada com base no valor dos bits **TOCS** e **PSA** do registro **OPTION.**

O bloco **PRESCALER** é um divisor programável e que seu funcionamento será explicado na próximo passo.

Vejamos na pratica como é possível agir sobre este bloco para obter diferentes modalidades de contagem pelo registro TMR0.

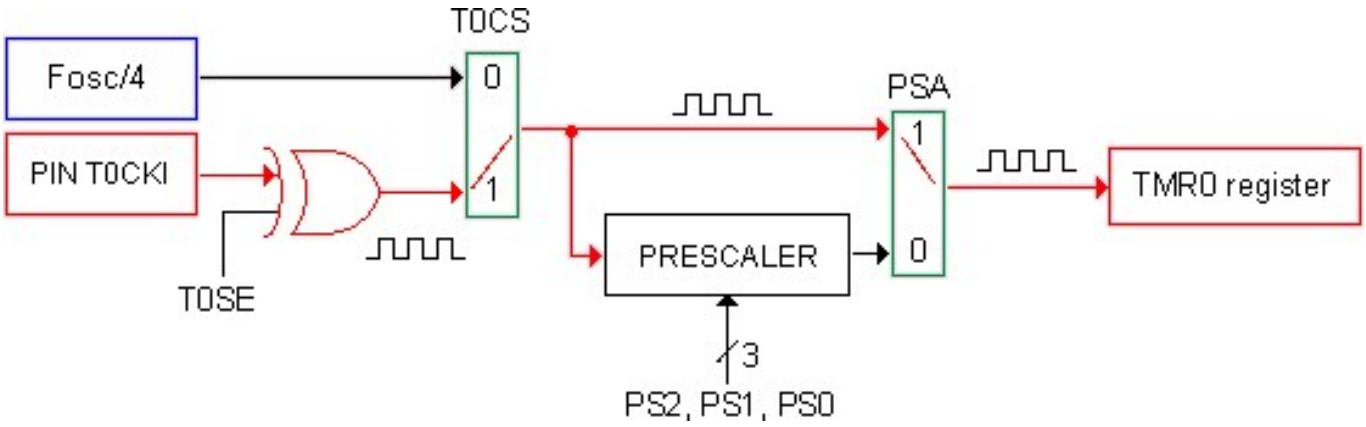
Iniciaremos programando o bit **T0CS em 0** e **PSA em 1**. A configuração de funcionamento que obteremos é a representada na figura abaixo:



A parte em **vermelho** mostra-nos o percurso que efetua o sinal antes de chegar ao contador TMR0.

Como já aviamos dito anteriormente, a frequência Fosc/4 é par e de um quarto da frequência de colck. Utilizando-se um cristal de quartzo de 4Mhz teremos uma frequência par de **1 MHz**. Tal frequência será enviada diretamente ao registro TMR0 sem haver nenhuma modificação. A cadencia de contagem que se obtém e então par e de 1 milhão de incrementos por segundo do valor presente no TMR0.

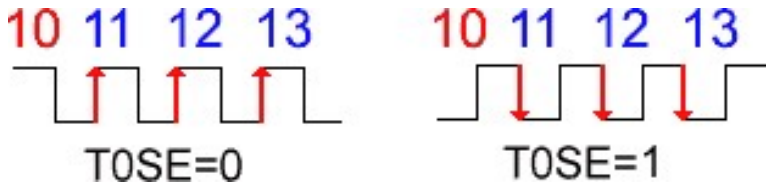
Imaginemos agora modificar o status do bit **T0CS de 0 para 1** a configuração que obteremos é seguinte:



Desta vez será o sinal aplicado ao pino TOCKI do PIC a ser enviado diretamente ao contador TMR0 determinando a frequência de contagem. Aplicando-se por exemplo a este pino uma frequência par de 100Hz obteremos uma de contagem par de cem incrementos por segundo.

A presença da porta lógica **XOR** (execlusive OR) na entrada TOCKI do PIC permite determinar o caminho do bit TOSE do registro OPTION se o contador TMR0 deve ser incrementado na decida do pulso (TOSE=1) ou na subida do pulso (TOSE=0) do sinal externo aplicado.

Na figura seguinte está representada a correspondência entre a cadencia do sinal externo e o valor que assume o contador TMR0:



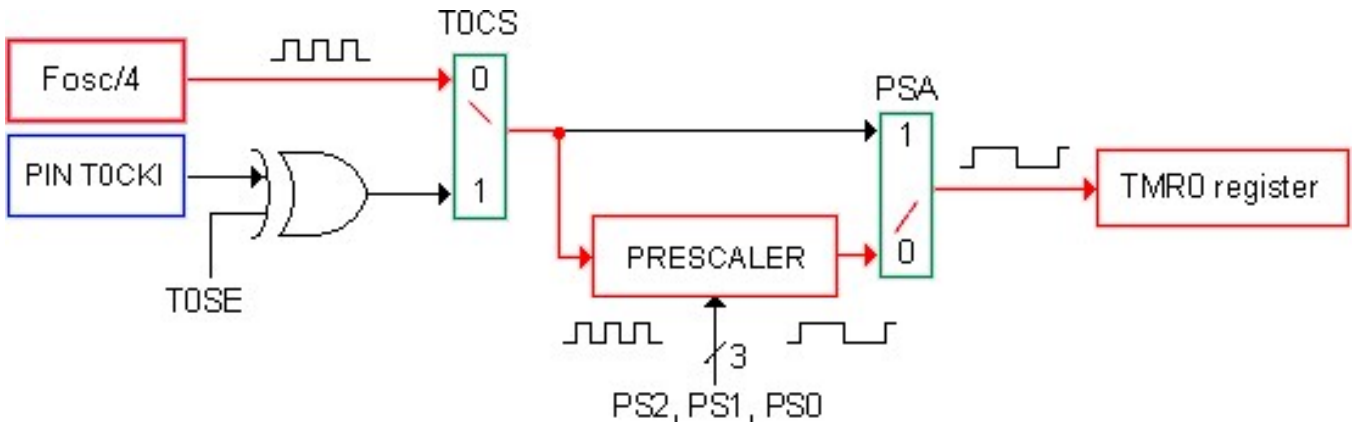
Na próximo passo veremos como é possível dividir interiormente a frequência de contagem, interna ou externa, ativando o PRESCALER.

Lição 4 | **Passo 2/2**

O PRESCALER

O último bloco a ser analisado para poder utilizar completamente o registro TMR0 é o PRESCALER.

Se configurarmos o bit **PSA** do registro **OPTION** em 0 enviamos ao registro TMR0 o sinal de saída do PRESCALER como é visível na figura abaixo:



O PRESCALER consiste na pratica de um divisor programável de 8 bits utilizado no caso pela frequência de contagem enviada ao contador TMR0 que é demasiada alta para nossos propósitos.

No exemplo descrito na lição anterior aviamos visto que utilizando um cristal de 4Mhz obtínhamos uma frequência de contagem par de 1 Mhz que para muitas aplicações poderá ser muito elevada.

Com o uso do PRESCALER podemos dividir interiormente a frequência Fosc/4 configurando desta forma os bits **PS0**, **PS1**, **PS2** do registro **OPTION** segundo a tabela abaixo:

PS2	PS1	PS0	Divisor	Frequencia de saida do prescaler (Hz)
0	0	0	2	500.000
0	0	1	4	250.000
0	1	0	8	125.000
0	1	1	16	62.500
1	0	0	32	31.250
1	0	1	64	15.625
1	1	0	128	7.813
1	1	1	256	3.906

Iremos agora efetuar um experimento sobre o que foi visto até aqui para exercitarmos o aprendizado.

Na lição 2 aviamos realizado um lampejador de quatro led's onde a sequência de lampejo era determinada por umasub-rotina software, ou seja, um retardo baseado no tempo de execução de um ciclo continuo da instrução.

Iremos agora rescrever a mesma sub-rotina para introduzir um retardo par de um segundo utilizando o registro **TMR0**.

A modificação está descrita no arquivo SEQTMR0.ASM.

Devemos antes de tudo programar o PRESCALER para obter uma frequência de contagem colocando a seguinte instrução no endereço do programa:

```
movlw     00000100B
movwf     OPTION_REG
```

Na pratica iremos programar o bit **TOCS** em **0** para selecionar como fonte de contagem o clock do PIC, o bit **PSA** em **0** para levar o PRESCALER ao registro TMR0 ao invés do Watch Dog(o qual veremos em seguida) e o bit de configuração do **PRESCALER** em **100** para obter uma frequência de divisão par de 1:32.

A frequência de contagem que obteremos sobre TMR0 que será par é:

Fosc = 1Mhz / 32 = 31.250 Hz

A sub-rotina Delay deverá utilizar oportunamente o registro TMR0 para obter um retardo par de um segundo. Vejamos como. As primeiras instruções que virão em seguida no Delay serão:

```
movlw     6
movwf     TMR0
```

e

```
movlw     125
movwf     Count
```

A duas primeira memorizam no TMR0 o valor 6 de modo que o registro TMR0 chegará a zero depois de 250 contagens (256 - 6 = 250) obtendo assim uma frequência de passagem pelo zero do TMR0 de:

31.250 / 250 = 125 Hz

A instrução seguinte memorizara em um registro de 8 bits (Count) o valor 125 de tal modo que, decrementara este registro de um a cada passagem pelo zero do TMR0, até que se obtenha uma frequência de passagem pelo zero do registro Count par de:

125/125 = 1Hz

A instrução inserida no loop DelayLoop se encarregara então de controlar se o TMR0 já chegou a zero, quando então reinicializara em 6 e decrementara o valor contido em Count. Quando Count alcançar antes desse o zero aí terá transcorrido um segundo e a sub-rotina poderá retornar ao programa que a chamou.

Lição 5	Introdução	

As interrupções

Ao termino desta lição saiba:

- O que é e como controlar as interrupções no PIC
- Como se deve escrever uma interrupt handler
- Quais os tipos de eventos controláveis do PICF84
- Come controlar mais interrupções contemporaneamente

Conteúdo da lição 5

Lição 5	Passo 1/3	

Interrupção

A interrupção é uma técnica particular do PIC que permite interceptar eventos externos ao programa em execução, interrompe momentaneamente a operação do programa em andamento, controla o evento com uma sub-rotina apropriadae retorna para a execução do programa.

Seja para fazer um parágrafo mais ou menos explicativo, podemos dizer que a interrupção é para o PIC, se não, o que para nós representaria uma chamada telefônica.

Para recebermos um telefonema não precisamos nos preocupar em ficar levantando continuamente o monofone do gancho para ver se tem alguém querendo falar com nós, mas podemos tranquilamente aguardar pelo toque da campainha quando alguém nos chama. Quando então apenas levantamos o monofone do gancho e interrompemos momentaneamente o sistema de chamada, respondemos ao telefone e , uma vez terminada a conversação, retornamos o monofone no gancho ou seja, do ponto onde aviamos interrompido.

Assim que ouvirmos o toque, podemos decidir interromper temporariamente nossos assuntos, atender o telefone e, quando a conversa terminar, retomar do ponto em que paramos.

Transportando o termo deste parágrafo ao PIC veremos que:

- O nosso telefone corresponde ao programa em execução;
- a chamada de alguém corresponde ao evento de controle;
- o monofone corresponde a requisição de interrupção;
- a nossa reposta ao telefone corresponde a sub-rotina de controle da interrupção.

É evidente que assim como é extremamente mais eficaz se ter uma Campininha conectada ao telefone é extremamente mais eficaz controlar nosso evento com uma interrupção ao invés de diretamente pelo programa.

Tipos de eventos e bit's de habilitação

O PIC16C84 está preparado para controlar interrupções ao final de quatro eventos diferentes, vejamos quais são:

1.
2. A troca de estado sobre a linha RB0 (External interrupt RB0/INT pin).
3. Ao final da contagem do registro TMR0 (TMR0 overflow interrupt).
4. A troca de estado sobre uma das linhas de RB4 a RB7 (PORTB change interrupts).
5. Ao final da escrita sobre um endereço da EEPROM (EEPROM write complete interrupt).

A interrupção de qualquer um destes eventos pode ser conseguido habilitando ou desabilitando independentemente uns dos outros, agindo sobre os seguintes bit's do registro **INTCON**:

-
- **INTE** (bit 4) se este bit estiver em 1 habilitará a interrupção de mudança de estado sobre a linha **RB0**
- **T0IE** (bit 5) se este bit estiver em 1 habilitará a interrupção de final de contagem do registro **TMR0**
- **RBIE** (bit 3) se este bit estiver em 1 habilitará a interrupção de mudança de estado sobre uma das linhas de **RB4** a **RB7**
- **EEIE** (bit 6) se este bit estiver em 1 habilitará a interrupção de final de escrita sobre um endereço da **EEPROM**

Existe um outro bit de habilitação geral de interrupção que deve ser ajustado em (1) antes destes, ou seja, o bit **GIE** (Global Interrupt Enable bit) e este é o **bit 7** do registro **INTCON**.

Vetor de Interrupção e Controle de Interrupção
(Interrupt vector e Interrupt handler)

Qualquer que seja o evento habilitado, ao se manifestar, o PIC interrompe a execução do programa em andamento, memoriza automaticamente no **STACK** o valor corrente do **PROGRAM COUNTER** e pula para a instrução presente no endereço de memória **0004H** denominada **Interrupt vector**(vetor de interrupção).

Deste ponto em diante devemos colocar a nossa sub-rotina de controle denominada **Interrupt Handler** (controle de interrupção).

Pode-se habilitar mais interrupções e, a primeira providencia da interrupt handler e a de verificar qual o evento habilitado fez gerar a interrupção e a execução da parte do programa relativo

Este controle pode ser efetuado utilizando a **Interrupt flag**.

Interrupt flag(sinalizador de interrupção)

Dado que qualquer interrupção gera uma chamada do endereço **04H**, no registro **INTCON** está presente o flag que indica qual o evento que gerou a interrupção vejamos:

-
- **INTF** (bit 1) Se vale 1 a interrupção é um estado gerado na troca de estado sobre a linha RB0.
- **T0IF** (bit 2) Se vale 1 a interrupção é um estado gerado no termino da contagem do timer TMR0.
- **RBIF** (bit 0) Se vale 1 a interrupção é um estado gerado da troca de estado de uma das linhas de RB4 a RB7.

Como se pode ver a interrupção de final de escrita na EEPROM não tem previsto nenhum flag de sinalização para que a interrup handler deva considerar que a interrupção é um estado gerado deste evento quando todos os três flags supra citados irão a 0.

Importante: Uma vez conhecido qual o flag está ativo, a interrupt handler deve zera-lo, ou então não mais gerara interrupção correspondente.

Retorno de uma interrupt handler

Quando for gerada uma interrupção o PIC desabilita automaticamente o bit GIE (global Interrupt Enable) do registro INTECON de modo a desabilitar todas as interrupções restantes. Para poder retornar ao programa principal e reinicializar

em 1este bit deve-se utilizar a instrução:

```
RETFIE
```

No próximo passo veremos um exemplo prático do uso das interrupções.

Lição 5 | **Passo 2/3**

Exemplo prático de controle de interrupção

Vejamos agora um exemplo prático de controle de interrupção. Pegaremos como base partida o código LED.ASM usado na lição 1 para realizar um lampejador a led.

Como recordamos este programa faz lampejar o LED1 simplesmente, presente sobre a placa PicTech, num ciclo continuo utilizando um retardo software introduzido da sub-rotina **Delay**

Vejamos agora como é possível fazer pressionando uma tecla acender o LED2 temporariamente com a execução doprograma principal.

O código de exemplo que iremos analisar esta disponível no INTRB.ASM

O circuito a ser realizado está representado no seguinte arquivo no formato Acrobat Reader (12Kb): example3.pdf.

Uma vez carregado o programa INTRB.ASM na placa PicTech notaremos que o **LED D1** lampejará exatamente como aviamos colocado o programa LED.ASM. Devemos agora pressionar as teclas de **SW1** a **SW4**. Veremos que o **LED D2** irá se acender imediatamente e permanecerá aceso por um tempo de 3 lampejos do LED 1.

Na pratica entre o loop principal, derivado do código LED.ASM, continua a fazer lampejar o LED D1 utilizando um retardo software introduzido da sub-rotina **Delay**, o PIC fara com que ao pressionarmos uma tecla, sinalizara imediatamente sobre o LED 2 sem influenciar de maneira evidente a frequência de lampejo.

Primeiro analisaremos o código INTRB.ASM vejamos a diferença de comportamento com o outro código que faz amesma operação sem ter que recorrer a interrupção.

Para este propósito compilaremos e colocaremos na placa Pictech o programa NOINTRB.ASM. Notaremos que a ascensão do led 2 em correspondência com o pressionamento de uma tecla é ligeiramente retardado em quanto a leitura doestado da linha RB4-7 não será efetuada pelo hardware de controle de interrupção, mas diretamente do programa principal a cada ciclo do loop. O rápido retardo é então devolvido a presença da sub-rotina **Delay** dentro do loop principal.

Analisaremos agora o código INTRB.ASM.

Partindo da diretiva **ORG 00H** que, como vimos serve para posicionar o nosso programa a partir da posição de reset, ou seja, da locação 0.

Notamos subitamente que a primeira instrução que o PIC encontra é um desvio incondicionado para o label **Start**:

```
ORG 00H
goto Start
```

seguido de uma nova diretiva:

```
ORG 04H
```

e então do código da sub-rotina de controle da interrupção:

```
bsf PORTB,LED2

movlw 3
movwf nTick

bcf INTCON,RBIF

retfie
```

Como havíamos dito na lição anterior, a interrupt handler alocar necessariamente a partir do endereço 04H, para evitarque seja executada logo após o reset devemos pula-la com uma instrução de salto incondicional.

O código da interrupt handler, neste casso, é muito simples e se limita a acender o LED D2, que inseri no registro existente **nTick** o número de lampejos até chegar a LED D2, devera apagar-se e zerar o flag **RBIF** para permitir ao circuito a geração de nova interrupção.

A instrução RETFIE permite ao PIC retornar a executar o programa do ponto que foi interrompido pela interrupção.

Mas porque é gerado uma interrupção quando pressionamos uma tecla qualquer?

Pois a primeira instrução que o PIC executa após o reset é a seguinte:

```
movlw 10001000B
movwf INTCON
```

onde, na pratica, foi colocado a um o bit **GIE** (bit 7) que é a habilitação global das interrupções e onde o bit **RBIF** (bit 3)que habilita, particularmente, a interrupção sobre troca de estado da linha RB4-7.

Na pratica, havendo contado as chaves SW1, SW2, SW3 e SW4 sobre a linha de I/O RB4, RB5, RB6 e RB7, com umpressionar de qualquer uma das teclas há uma troca de estado que gera uma interrupção

No loop principal, além da operação de acender e apagar do LED 1, será decrementado o contador **nTick** até chegar a zero. Em correspondência com este será apagado o LED 2.

Lição 5 | **Passo 3/3**

Exemplo prático de controle de mais interrupções

Vejamos agora como controlar mais interrupções contemporaneamente.

Utilizando sempre como base o código anterior INTRB.ASM poderemos controlar até mesmo uma interrupção sobre a fina contagem do registro **TMR0**. Ou seja, façamos lampejar o **LED D3** em correspondência com a contagem de **TMR0**.

O código a se utilizar é DBLINT.ASM.

Compile e descarregue o programa DBLINT.ASM na placa PicTech e vermos que além do **LED D1** que lampeja com a frequência anterior, e o **LED D3** com uma frequência mais elevada.

Pressionando-se uma tecla qualquer, obtemos a costumeira ascensão por três ciclos do **LED D2**. O efeito final que obtemosé a execução de três tarefas a uma velocidade tal que se assemelha com uma execução paralela.

Analisando agora o código DBLINT.ASM.

Ao modificar o anterior observe a interrupt handler no início do qual está efetuado um controle sobre qual evento havia gerado a interrupção. Com a instrução:

```
btfsc   INTCON,T0IF
goto    IntT0IF
btfsc   INTCON,RBIF
goto    IntRBIF
```

será controlado o flag **T0IF** e **RBIF** para ver respectivamente se o evento que provocou a interrupção provem do registro **TMR0** ou da porta **RB4-RB7**. E seguida será posta em execução a relativa sub-rotina de controle a partir do label **intT0IF** e **intRBIF.**

Antes de devolver o controle ao programa principal devemos zerar o flag **T0IF** e **RBIF** para assegurar que o próximo evento possa gerar interrupção.

Lição 6	Introdução	

O Power Down Mode (Sleep)

Ao termino desta lição saiba:

- Como colocar o PICmicro no Power Down Mode e como reaviva-lo

Conteúdo da lição 6

Lição 6	Passo 1/2	

Funcionamento do Power Down Mode

O **Power Down Mode** e **Sleep Mode** é um estado particular de funcionamento do PICmicro utilizado para reduzir o consumo de corrente no memento em que o PICmicro não é utilizado e aguarda um evento externo.

Se pegarmos como exemplo um controle remoto para TV veremos que na maior parte do tempo o PICmicro permanece aguardando que alguém pressione uma tecla. Apenas quando à pressionamos o PICmicro efetua uma breve transmissão e se coloca de novo a espera de um novo pressionar de tecla.

O tempo de utilização efetivo da CPU do PICmicro é então limitado a poucos milissegundos necessário para efetuar a transmissão ao passo que para outros não é preciso nenhuma elaboração particular.

Para evitar o consumo inútil frente a limitada energia da bateria é possível desligar boa parte do circuito de funcionamento do PICmicro e reaviva-lo somente quando um evento externo ocorrer.

Vejamos como.

A instrução SLEEP

A instrução **SLEEP** será utilizada para colocar o PICmicro em Power Down Mode e reduzir consequentemente a corrente absorvida que passara de cerca dos 2mA (a 5 volts com clock de 4Mhz) para cerca dos 2uA, ou seja, 1000 vezes menos.

Para entrar em Power Down Mode basta inserirmos esta instrução em um ponto qualquer do nosso programa:

```
SLEEP
```

Qualquer instrução presente depois de SLLEP não será executada pelo PICmicro que terminara neste ponto sua execução, desligará todos os circuitos internos, menos aqueles necessários a manter o estado da linha de I/O (estado logico alto, baixo ou de alta impedância) para informar a condição de "reaviva-lo" o qual veremos em seguida.

Para reduzir o consumo de corrente neste estado, não devemos fazer obviamente um circuito que conectado a linha de escrita do PIC consuma corrente excessiva. O melhor circuito que podemos projetar é aquele que absorverá o mínimo de corrente na condição de Power Down. Um outro truque recomendado pela Microchip é conectar ao **positivo (Vdd)** ou ao **negativo(vss)** da alimentação todas a linhas em alta impedância não utilizadas, como a linha RA4/T0CKI (pin 3).

O "WAKE UP" despertar do PICmicro

Para despertar o PICmicro do seu sono podemos utilizar diversas técnicas:

1. Reset do PICmicro colocando em 0 o pino MCLR (pino 4)
2. Timeout do timer do Watchdog (se habilitado)

3. Verificação de uma situação de interrupção (interrupção do pino RB0/INT, troca de estado sobre a porta B, termino da operação de escrita na EEPROM)

No caso 1 e 2 o PICmicro será resetado e a execução começará da posição 0. No caso 3 o PICmicro se comporta como no atendimento de uma interrupção ((veja lição 5) irá para a primeira interrup handler e então retornará para execução após a instrução SLLEP. Para que o PICmicro possa retornar de uma interrupção devemos habilitar o flag do registro **INTCON**.

O bit **TO** e **PD** do registro **STATUS** podem ser utilizados para determinar se a causa do despertar está ligada a simples ascensão do PICmicro, o reset ocorre na chegada de uma interrupção.

Lição 6	Passo 2/2	

Primeiro exemplo sobre o Power Down Mode

Vejamos primeiramente um simples exemplo de utilização do **Power Down Mode** e da modalidade de "despertar" do PICmicro. A modalidade utilizada é a interrupção sobre a frente de decida aplicada ao pino RB0/INT utilizando-se uma chave. O código utilizado é PDM1.ASM.

O circuito a ser realizado está representado no seguinte arquivo no formato Acrobat Reader (10Kb): example4.pdf

Na pratica o **LED D1** conectado na linha RB2 lampejará para indicar a execução do programa em curso. Pressionando-sea tecla SW2 o programa irá para instrução SLEEP mantendo o PICmicro em Power Down Mode. O LED D1 permanecerá aceso ou apagado dependendo do momento exceto por pressionar SW2.

Para promovermos a saída do Power Down Mode do PICmicro, bastará pressionar SW1 para gerar uma interrupção efaze-lo retornar a execução do programa.