MEC

MINISTÉRIO DA EDUCAÇÃO

SECRETARIA DE EDUCAÇÃO MÉDIA E TECNOLÓGICA

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DO ESPÍRITO SANTO

INTRODUÇÃO AO MICROCONTROLADOR PIC 16C84

Vitória – ES
2002

Sumário

O MICROCONTROLADOR PIC16C84

1 - INTRODUÇÃO 1

1.1. O Diagrama de blocos da arquitetura do PIC16C84 3
1.2. Escrita e compilação de um programa em assembler para o PIC 3
1.3. O conjunto de instruções do PIC16C84 5
1.4. Uma simples aplicação (pisca-pisca com led) 7
1.5. Analisando o código assembler do PIC 9
1.5.1. A subrotina Delay 14
1.6. Compilando o código assembler 16
1.7. Programando o PIC16C84 17

2 - ESTUDO DA ARQUITETURA DO PIC16C84 18

2.1.A área de programa e o registrador de arquivo 19
2.2. A ALU (Unidade Lógica e Aritmética) e o registro W (Acumulador) 21
2.3. O contador de programa e Stack ou pilha 22
2.4. Segunda aplicação – Implementando um sequencial de leds 25

3 - INTRODUÇÃO AOS PERIFÉRICOS DO PIC 28

3.1. As portas de entrada e saída A e B 28
3.2. Estado de escrita das linhas de I/O 29
3.2.1. Estado de escrita das linhas RA0, RA1, RA2 e RA3 29
3.2.2. Estado de saída das linhas RB0, RB1, RB2 e RB3 32
3.2.3. Estado de saída das linhas RB4, RB5, RB6 e RB7 33
3.3. Terceira aplicação – implementando um circuito com teclas 34

4 – O CONTADOR TMR0 E O PRESCALER 36

4.1. O registro contador TMR0 36
4.1.1. O PRESCALER e sua relação com o TMR0 39

5 – AS INTERRUPÇÕES NO PIC16C84 43

5.1. Tipos de eventos e bits de habilitação de interrupção 43
5.2. Vetor de interrupção e controle de interrupção 43
5.3. Interrupt flag (sinalizador de interrupção) 44
5.4. Retorno de uma interrupt handler 44
5.5. Elemplo prático de controle de interrupção 44

6 – O POWER DOWN W O SLEEP MODE 51

6.1. A Instrução SLEEP 51
6.2. O “despertar” do PIC 52
6.3. Um exemplo de uso do Power Down Mode 52

O MICROCONTROLADOR PIC 16C84

1. INTRODUÇÃO

O **PIC 16C84,** como se pode ver na figura 1, é um circuito integrado produzido pela **Microchip Technology Inc.**, pertencente a categoria dos microcontroladores de 8 bits, ou seja um componente integrado que em um único dispositivo contém todos os circuitos necessários para realizar um completo sistema digital programável.

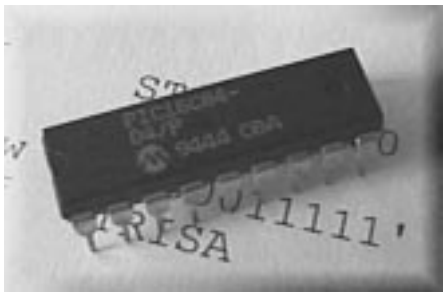

Figura 1 – O PIC 16C84

Este microcontrolador tem uma arquitetura RISC (Conjunto Reduzido de Instruções), na qual os barramentos de dados e de endereços são independentes. Isto proporcionará um reduzido número de instruções e um tempo preciso na execução de cada instrução.

Além disso, ele 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, ele possui:

- Uma **CPU** (**C**entral **P**rocessor **U**nit ou Unidade de Processamento Central) cuja finalidade é interpretar as instruções de programa;
- Uma memória **PROM** (**P**rogrammable **R**ead **O**nly **M**emory ou Memória Programável Somente para Leitura com Escrita/Gravação eletricamente) a qual irá armazenar as instruções do programa.
- Uma memória **RAM** (**R**andom **A**ccess **M**emory ou Memória de Accesso Aleatório) utilizada para memorizar as variáveis utilizadas pelo programa.
- Uma série de **LINHAS de I/O (entrada/saida)** para controlar dispositivos externos ou receber pulsos de sensores, chaves, etc.
- Uma série de dispositivos auxiliares ao seu funcionamento, ou seja, gerador de clock, controlador de bus, contador, etc.

A presença de todos estes dispositivos em um espaço extremamente pequeno, possibilita ao projetista uma ampla flexibilidade de trabalho e uma 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.

A família **PIC** tem disponível inúmeras versões para melhor adaptar-se as exigências de projetos específicos, diferenciando-se entre si pelo número de linhas de I/O, pelos recursos disponíveis e pela capacidade de processamento do dispositivo. A família inicia-se com o menor modelo identificado pela sigla **PIC12Cxx** dotado de 8 pinos, e encerra-se com modelos maiores com sigla **PIC17Cxx** dotados de 40 pinos.

Uma descrição detalhada da tipologia do PIC está disponível nos "DATA BOOKs" correspondentes ou no site da Microchip (http://www.microchip.com) acessível via internet, onde conseguimos encontrar uma grande variedade de informações técnicas, software de apoio, exemplos de aplicações e atualizações disponíveis.

Para o nosso curso usaremos o modelo intermediário de PIC o **16C84 ou 16F84**. Este é dotado de 18 pinos sendo 13 disponíveis para entrada/saída (I/O) programáveis. Este microcontrolador tem algumas características que o tornam um circuito que melhor irá atender as exigências do nosso curso.

Especificamente o PIC16C84 dispõe de uma memória para armazenar o programa, do tipo **EEPROM (E**lectrical **E**rasable **P**rogrammable **R**ead **O**nly **M**emory) que pode ser apagada e reescrita (em torno de 100 vezes), o que é ideal para os nossos experimentos. Tal característica é plenamente aproveitada em nosso **circuito programador** que será usado neste curso.

A figura 2 apresenta os pinos e a função de cada um no **PIC16C84** e nela, podemos notar a reprodução da pinagem e sua respectiva nomenclatura.

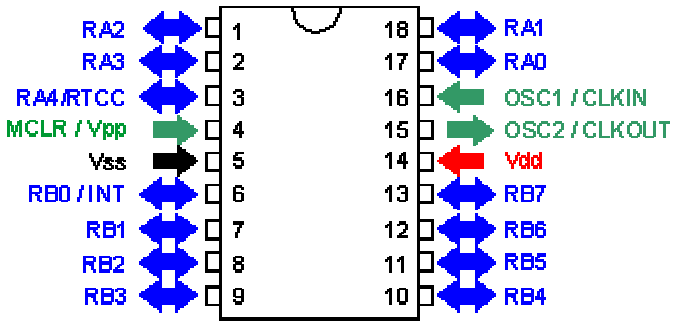

Figura 2 – A pinagem do PIC

Como é possível observar, o PIC16C84 é dotado de um total de **18 pinos** dispostos em duas fileiras paralelas de 9 pinos cada uma (**dual in line**). Os pinos com nomes iniciados em R representam as linhas de I/O disponíveis para as nossas aplicações, os com nomes iniciados em V são os pinos de alimentação, e os demais são reservados ao funcionamento do PIC (MCLR para o reset e OSC1-2 para o clock).

Como apresentado anteriormente o pino **Vdd (pino 14)** e **Vss (pino 5)** servem para fornecer alimentação de 5 Volts para o chip. Eles são ligados, respectivamente, ao pólo positivo e a referência da bateria. O pino **MCLR (pino 4)** serve para resetar o chip quando ele estiver na condição lógica zero. Ele deve ser conectado diretamente ao potencial positivo do programador.

Os pinos **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 internas 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 maximo de **4Mhz** da qual se obtém uma velocidade de execução das instruções simples de **1 milhão de instruções por segundo (1uS por instrução)**. No nosso caso para o oscilador de clock usaremos um circuito RC. Os pinos **RA0 a RA4 e RB0** (pino 6) a RB7 são as linhas de I/O disponíveis no PIC.

1.1. Diagrama de blocos da arquitetura interna do PIC

A figura 3 a seguir, apresenta um diagrama de blocos com a arquitetura interna do PIC16C84. Neste caso, podemos observar os seus registros de uso geral e especiais, tais como, o contador de programa, o registro de instrução, o registro de Status, o acumulador W e etc. Podemos observar ainda, as áreas de memória EEPROM de programas e de dados e a memória RAM, a Unidade Lógica e Aritmética (ULA), os circuitos de controle e temporização e as portas de entrada e saída, entre outros.

Nas seções posteriores deste curso, os blocos mais importantes serão tratados de forma específica e com maior profundidade. Neste caso, iremos nos deter ao seu funcionamento no que diz respeito aos programas a serem desenvolvidos.

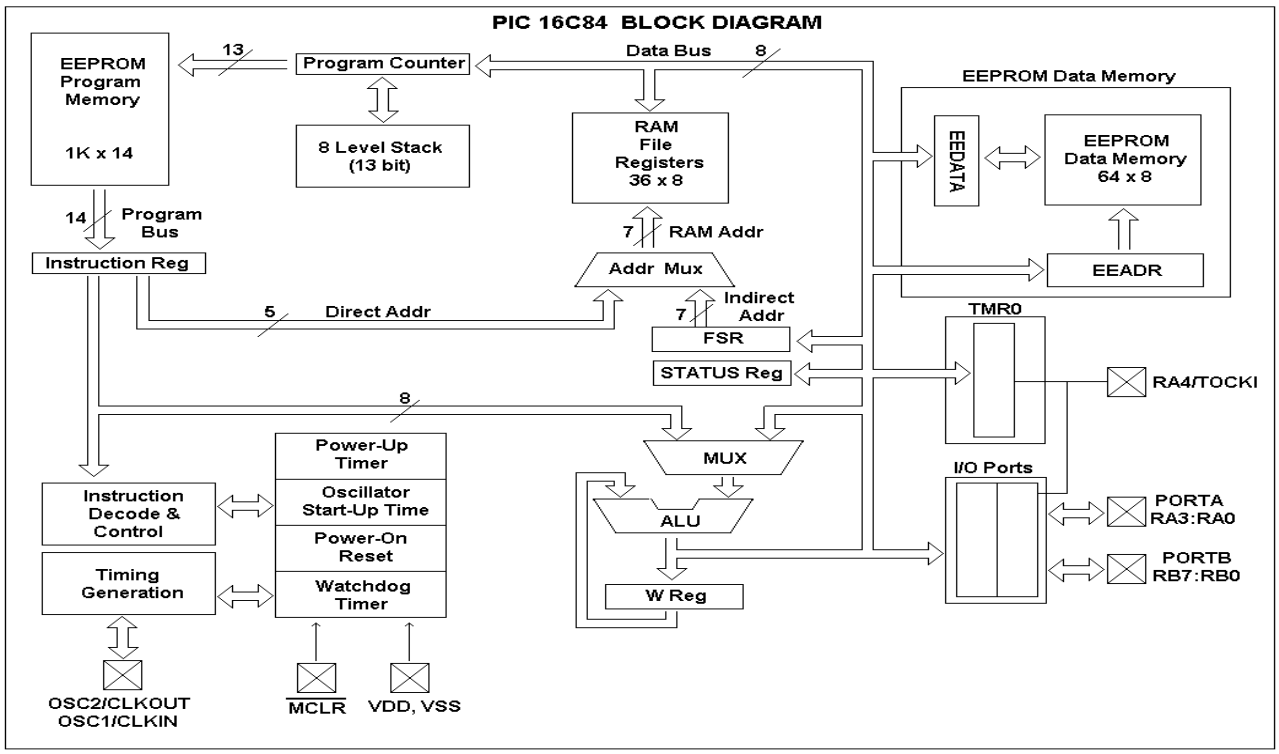

Figura 3 – Arquitetura interna do PIC16C84

1.2. Escrita e compilação de um programa em assembler para o PIC

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 identificará precisamente a função básica que o PIC ira executar. Cada instrução é representada por um código de operação (do ingles **operation code** ou abreviadamente **opcode**) e no PIC16C84, podemos memorizar 14 bits em cada locação da memória EEPROM. Esta memória no PIC16C84 dispõe de 1024 locações e cada uma deverá conter uma só instrução. Um exemplo de opcode em **notação binária** esta 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. A 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 único que o o microcntroador PIC esta preparado para entender. Para facilitar o entendimento do programador, se recorre a um instrumento e a uma convenção para tornar a instrução mais compreensível. A primeira convenção é a que associa o opcode da instrução (num total de 35 para o PIC16C84) a uma sigla **mnemônica**, ou seja uma inicial que seja fácil de recordar o significado da instrução.

Voltando ao 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 nas definições, 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 codigo assembler**.

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

Na figura 4 esta esquematizado o fluxograma das operações e arquivos que deverão ser gerados para passar um um código na linguagem assembler para um PIC a ser programado.

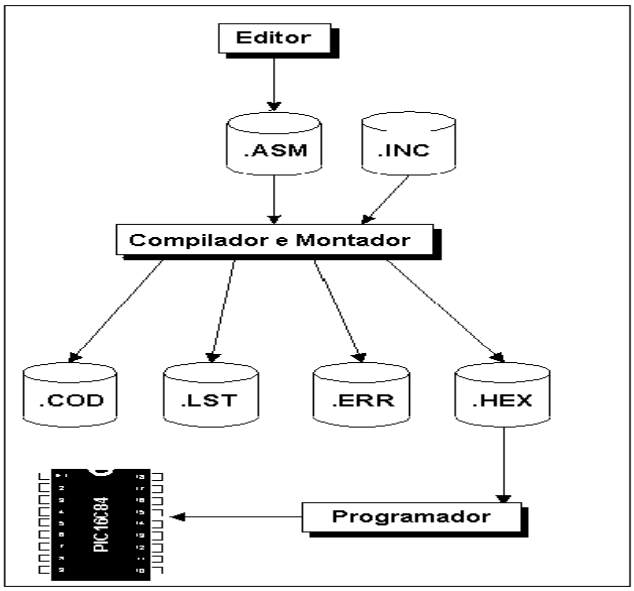

Figura 4 – Fluxo da programação do PIC

Como é possível observar, além do código com extensão **.ASM** é necessário fornecer ao compilador um segundo arquivo, produto da Microchip, com extensão **.INC** que corresponde a uma biblioteca do tipo do PIC que estamos utilizando. No nosso caso o arquivo deve ser o **P16C84.INC**. Este código contém algumas definições das quais depende o tipo de chip utilizado.

Durante a compilação do código, o compilador assembler gera uma série de arquivos com nomes idênticos ao código original, mas com extensões diferentes:

- **.HEX** é o arquivo que contém o código de operação em formato hexadecimal que será enviado ao PIC via interface de programação.
- **.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 verificação de erros e do processo de compilação que foi feito.
- **.ERR** contém uma lista de erros de compilação que mostra o número da linha do código na qual esta o erro no código assembler.

Os arquivos .LST, .ERR são utilizados somente para controle da compilação. Somente o arquivo .HEX será utilizado realmente para programar o PIC.

O arquivo .HEX não é um arquivo no formato binário e não reflete diretamante o conteúdo que deverá ter a EEPROM do PIC. Mas este formato é muito importante quando da transferência do programa para o PIC em uma forma legível.

Sem muitos detalhes, tal formato é diretamente reconhecido pelo programador do PIC que promoverá durante a programação a conversão em binário e contém outras informações que serão adicionadas aos endereços na qual vamos transferir o opcode.

1.3. O conjunto de instruções do PIC16C84

O conjunto de instruções do PIC16C84 ou 16F84 está representado na tabela a seguir.

PIC16X8Y - CONJUNTO DE INSTRUÇÕES

Mnemonico	Op.	Descrição Microchip	Descrição Equivalente	Ciclos	Cód. Operação 14-Bits				Flags	Notas
ADDLW	k	Add literal and W	W = W + k	1	11	111x	kkkk	kkkk	C,DC,Z	
ADDWF	f, d	Add W and f	d = W + f (onde d pode ser W ou f)	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDLW	k	AND literal with W	W = W AND k	1	11	1001	kkkk	kkkk	Z	
ANDWF	f, d	AND W with f	d = W AND f (onded pode ser W ou f)	1	00	0101	dfff	ffff	Z	1,2
BCF	f, b	Bit Clear f	f(b) = 0	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	f(b) = 1	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	f(b) = 0 ? Se é, salte uma instrução	1(2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	f(b) = 1 ? Se é, salte uma instrução	1(2)	01	11bb	bfff	ffff		3
CALL	k	Call subroutine	Chamada a uma subrotina no endereço k	2	10	0kkk	kkkk	kkkk		
CLRF	f	Clear f	f = 0	1	00	0001	lfff	ffff	Z	2
CLRW	-	Clear W	W = 0	1	00	0001	0xxx	xxxx	Z	
CLRWDT	-	Clear Watchdog Timer	Watchdog timer = 0	1	00	0000	0110	0100	TO,PD	
COMF	f, d	Complement f	d = not f (onde d pode ser W ou f)	1	00	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	d = f -1 (onde d pode ser W ou f)	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	d = f -1 (onde d pode ser W ou f) se d = 0 salte	1(2)	00	1011	dfff	ffff		1,2,3
GOTO	k	Go to address	Desvie para o endereço k	2	10	1kkk	kkkk	kkkk		
INCF	f, d	Increment f	d = f +1 (onde d pode ser W ou f)	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	d = f +1 (onde d pode ser W ou f) se d = 0 salte	1(2)	00	1111	dfff	ffff		1,2,3
IORLW	k	Inclusive OR literal with W	W = W OR k	1	11	1000	kkkk	kkkk	Z	
IORWF	f, d	Inclusive OR W with f	d = f OR W (onde d pode ser W ou f)	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	W = k	1	00	1000	dfff	ffff	Z	1,2
MOVLW	k	Move literal to W	d = f (onde d pode ser W ou f)	1	11	00xx	kkkk	kkkk		
MOVWF	f	Move W to f	f = W	1	00	0000	lfff	ffff		
NOP	-	No Operation	Nenhuma operação	1	00	0000	0xx0	0000		
RETFIE	-	Return from interrupt	Retorna de uma interrupt handler	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	Retorna de uma subrotina com W = k	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	Retorna de uma subrotina	2	00	0000	0000	1000		
RLF	f, d	Rotate Left f through Carry	d = f << 1 (onde d pode ser W ou f)	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	d = f >> 1 (onde d pode ser W o f)	1	00	1100	dfff	ffff	C	1,2
SLEEP	-	Go into standby mode	Coloca em standby o PIC	1	00	0000	0110	0011	TO,PD	
SUBLW	k	Subtract W from literal	W = k - W	1	11	110x	kkkk	kkkk	C,DC,Z	
SUBWF	f, d	Subtract W from f	d = f - W (onde d pode ser W ou f)	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	f = Swap dos bits 0123 com 4567 de f	1	00	1110	dfff	ffff		1,2
XORLW	k	Exclusive OR literal with W	W = W XOR k	1	11	1010	kkkk	kkkk	Z	
XORWF	f, d	Exclusive OR W with f	d = f XOR W (onde d pode ser W ou f)	1	00	0110	dfff	ffff	Z	1,2

Notes:

1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

1.4. Uma simples aplicação (pisca-pisca com LED)

Como primeira aplicação, iremos montar um circuito muito simples, como mostra a figura 5 e seu propósito é fazer piscar um diodo led.

Veremos como se escreve um programa em linguagem assembler, como se compila e como se transfere o mesmo para o interior da EEPROM do PIC e assim fazê-lo funcionar.

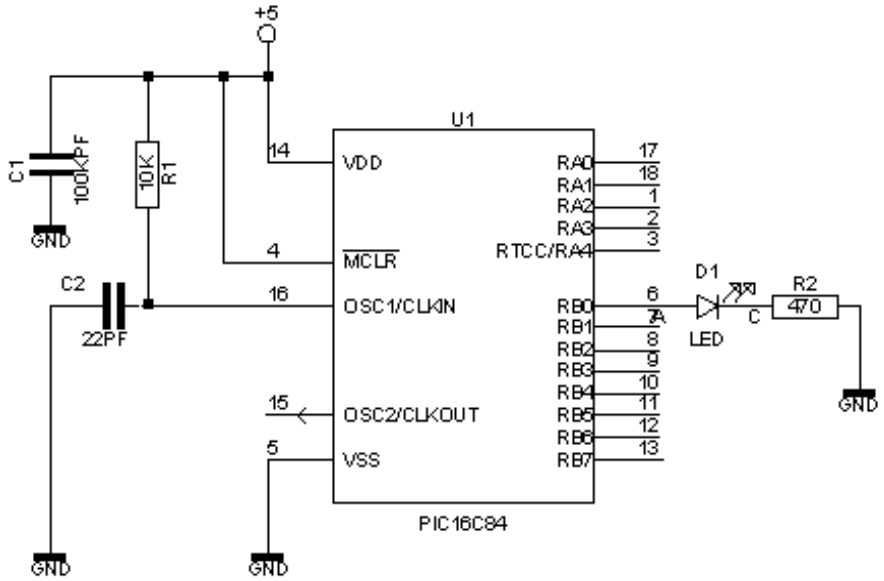

Figura 5 – Esquema do pisca-pisca com led

Serão usados os pinos 14 e 5 para alimentar o microcontrolador, o pino 16 para a entrada do sinal de clock e o pino 4 para habilitar o seu funcionamento.

O pino RB0 **(pino 6)** é uma das linhas de I/O disponível no PIC. Na nossa aplicação, após a gravação do programa no PIC, ela será conectada ao led por intermédio de uma resistor de limitação de corrente.

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.** Na nossa primeira experiência prática utilizaremos o arquivo de nome **LED.ASM**, apresentado a seguir:

```
;*************************************************
; LED.ASM
;*************************************************

        PROCESSOR       16F84
        RADIX           DEC
        INCLUDE         "P16F84.INC"

LED     EQU     0

        ORG     0CH

Count   RES     2

        ;Reset Vector
        ;Inicio do programa ao ser resetado a CPU

        ORG     00H

        bsf     STATUS,RP0
        movlw   00011111B
        movwf   TRISA
        movlw   11111110B
        movwf   TRISB
        bcf     STATUS,RP0
        bsf     PORTB,LED

MainLoop

        call    Delay
        btfsc   PORTB,LED
        goto    SetToZero
        bsf     PORTB,LED
        goto    MainLoop

SetToZero

        bcf     PORTB,LED
        goto    MainLoop


        ; Subrotinas
Delay
        clrf    Count
        clrf    Count+1

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

        return
        END
```

Após a escrita do programa, 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** que é o produto da Microchip disponível.

Como vimos anteriormente, além do código com extensão **.ASM** é necessário fornecer ao compilador um segundo arquivo, produto da Microchip, com extensão **.INC** que corresponde a biblioteca do tipo do PIC que estamos utilizando. No nosso caso o arquivo é o **P16F84.INC**. Este código contém algumas definições das quais depende o tipo de chip utilizado.

1.5. Analisando o código assembler do programa

Analisaremos agora linha por linha o conteúdo do nosso código **LED.ASM**. Partiremos da primeira linha de codigo:

```
PROCESSOR 16F84
```

PROCESSOR é uma diretiva do compilador assembler que indica a definição do microcontrolador para o qual 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 as instruções que colocaremos no nosso código são relativas a de um PIC16F84.

```
RADIX       DEC
```

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

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

Esta diretiva indica 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 ele fosse parte do nosso código.

```
LED  EQU  0
```

A diretiva **EQU** é muito importante para se definir 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 o de se tornar o código mais legível e permitir a definição de um valor constante em um único ponto do código.

OBS: A diretiva só faz sentido 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 definir o endereço na qual queremos que o compilador inicie a alocação dos dados ou a instrução seguinte. E neste caso estamos definindo uma área de dados dentro do PIC ou seja uma área em que iremos armazenar variáveis e contador durante a execução do nosso programa. Esta área coincide exatamente com a área de RAM do PIC definida pela Microchip como **FILE REGISTER (Registrador de Arquivo)**.

O registrador de arquivo nada mais é do que uma locação na memória 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 uso interno.

```
Count  RES 2
```

Nesta linha encontramos um label (rótulo) onde **Count** é uma varável e, **RES** é uma diretiva. A diretiva RES indica ao compilador que queremos reservar um certo número de bytes (neste caso 2 bytes) no meio do registrador de arquivos (File Register) dentro da área de dados.

O label Count, onde Count é um nome de uma variável para nós, é um marcador que no resto do código assumirá o valor do endereço em que esta colocado. Dado este que anteriormente havíamos definido no endereço de partida em 0CH com a diretiva ORG, então Count irá para a posição de memória 0CH.

Se nós inserirmos um label após a linha sucessiva essa irá ocupar a posição 0CH + 2 (dois bytes reservados) ou seja 0EH. O nome do label pode ser qualquer um com exceção das palavras reservadas ao compilador as quais são as instruções mnemônicas e diretivas.

```
ORG 00H
```

Esta segunda diretiva ORG 00H 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 deverá converter no respectivo opcode do PIC. Ela indica que o primeiro opcode visto pelo PIC após o **reset** é aquele memorizado na localização 0, e dai o fato do valor 00H ser inserido na diretiva ORG.

```
bsf STATUS,RP0
```

É a primeira instrução mnemônica de fato. O PIC tem uma CPU interna com arquitetura do tipo **RISC.** Neste caso, a instrução ocupa uma só locação de memória (opcode e parâmetro incluso). Aqui a instrução mnemônica **bsf** quer dizer **B**IT **S**ET **F**ILE REGISTER, ou seja,

coloque em nível alto (condição lógica alta) um dos bits contido na locação de memória RAM especificada.

O registro STATUS é um registro especial de 8 bits, começando pelo bit 0 (bit menos significativo) e indo até o 7 (bit mais significativo), que permitem, entre outras coisas, verificar o estado da CPU em relação ao resultado da ultima operação executada (carry ou transporte, zero e etc), bem como, permitir a mudança do banco de memória. Este registro está definido como um parâmetro no arquivo P16C84.INC. O valor colocado neste 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 nível alto (um). Esta instrução na prática coloca em nível 1 o quinto bit do registrador de arquivo STATUS. Esta operação é necessária, como veremos mais adiante, para acessar o registrador de arquivo TRISA e TRISB.

```
movlw 00011111B
```

Esta instrução significa: **MOV**E **L**ITERAL TO **W** REGISTER (mover o valor literal para W) ou seja mover um valor constante para o acumulador W. Como veremos mais adiante, o acumulador, é um registro particular utilizado pela CPU em todas as situações em que efetuarmos uma operação entre dois valores, ou em operações de deslocamento entre locações da memória.

Na pratica, ele é um registro de apoio utilizado pela CPU para memorizar temporariamente um byte toda vez que houver necessidade. O valor constante para memorizar no accumulador é **00011111B** ou seja um valor binário de 8 bits onde o bit mais a direita representa o bit 0 de W ou o bit menos significativo.

Na próxima instrução temos:

```
movwf TRISA
```

O valor 00011111 está memorizado no registro TRISA (O registro TRISA também é definido através de uma diretiva EQU) e a sua função é senão a de definir a direção (entrada ou saída) da linha de I/O do PORT A. Este bit é em particular um bit do registro TRISA ele, por ser igual a 1, determina como entrada sua respectiva linha no port A (se fosse 0 determinaria como saida).

Em seguida, na tabela 1, descrevemos a configuração dos pinos do PIC quando for executada 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	-

Tabela 1 – Configuração dos pinos do PIC com a instrucão

Como é possivel ver os bits 5, 6 e 7 não correspondem a nenhuma linha de I/O e seus valores em nada influenciam.

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

```
movlw B'11111110'
movwf TRISB
```

e neste caso a definição da linha será a repreentada na tabela 2 seguinte:

N.bit registro TRISB	Linha porta B	N.Pino	Valor	Estado
0	RB0	6	0	Saida
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

Tabela 2 – Configuração dos pinos após a instrução acima

Nota-se que o valor 0 no bit 0 do registro TRISB determina a configuração em escrita (saída) da respectiva linha do PIC. Na nossa aplicação esta linha será usada para controlar o LED e fazê-lo piscar.

Vimos que instrução **movwf TRISB** transferia o valor contido no acumulador (inicializado anteriormente com a instrução **movlw 11111110B**) para o 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 colocar RP0 em zero. E **bcf** significa neste caso **B**IT **C**LEAR **F**ILE REGISTER.

Do ponto de vista funcional esta instrução permite o acesso ao registro interno do banco 0 ou seja do qual faz parte o portA e portB, e o banco 1 do qual faz parte TRISA e TRISB. Uma descrição mais detalhada será vista mais adiante 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, ela irá acender o led conectado a linha RB0. **PORTB** é uma constante definida no arquivo P16C84.INC e faz referência ao registrador de arquivo correspondente a linha de I/O do portB onde **LED** é o numero da linha que irá a 1.

```
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 virá seguido a MainLoop, ou seja: **07H**.

Na realidade o valor que o label assumirá 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. Neste caso o label MainLoop será utilizado como ponto de entrada num ciclo (**do inglês Loop**) de acender e apagar o led, ou seja uma parte do código que colocará 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 subrotina que inicia vinculada ao label **Delay**. A subrotina 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. Neste caso a subrotina insere um retardo de tempo entre o acender e o apagar do led.

A instrução que compõe a subrotina **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 esta 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 acenderemos.

```
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 faz o programa pular para a instrução seguinte se o led está apagado.

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

Esta duas instruções simplesmente **acendem** o led e retornam o programa ao início do ciclo de acender/apagar o led.

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

Estas duas instruções simplesmente **apagam** o led e retornam o programa ao início do ciclo de acender/apagar o led.

1.5.1. A subrotina Delay

Como descrito anteriormente esta subrotina coloca um retardo de tempo de cerca de um segundo (usado como o tempo que o led fica aceso ou apagado) 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 é um label que identifica o endereço de início da subrotina e será utilizado pela chamada através do corpo do programa principal. **DelayLoop** é outro label que será chamado internamente pela subroutina e serve como ponto de entrada para o ciclo (do inglês loop) de retardo.

Na prática o retardo de tempo é conseguido executando-se milhares de instruções que não fazem nada!. Este tipo de retardo se chama retardo por software ou retardo de programa. Este é o tipo de retardo mais simples de implementar.

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

CLEA**R F**ILE REGISTER zera duas locações da memória RAM reservada anteriormente com a instrução **Count RES 2.**

Estas duas locações 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 será executada a próxima instrução:

```
goto DelayLoop
```

Que executa o ciclo de retardo. Uma vez igual a zero, o contador Count irá para a próxima instrução:

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

Que decrementará o registro seguinte até que este chegue a zero. O registro Count+1 em particular será decrementado de 1 (um) até 256 decrementos de Count. Quando então Count+1 chegar ao valor zero a instrução:

Return

que significa **RETURN** FROM SUBROUTINE determinará a saída da rotina de retardo e retornará à execução da instrução imediatamente após o call Delay.

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

1.6. Compilando o código assembler

Vejamos agora como é possivel efetuar na prática a compilação de um código assembler.

Primeiramente crie em seu disco rígido um diretório de trabalho que de agora em diante ficará armazenado todos os programas do curso. Escolhe-se um nome para o mesmo, por exemplo: **C:\PICPRG**

Copie para o diretório de trabalho C:\PICPRG os arquivos LED.ASM e P16C84.INC.

Para Compilar o nosso código LED.ASM existem duas opções:

- A primeira, no ambiente DOS, para compilar o programa, coloca-se na frente do prompt do DOS o comando:

MPASM LED.ASM

O resultado que deveremos obter no vídeo é o mostrado na figura 6 abaixo:

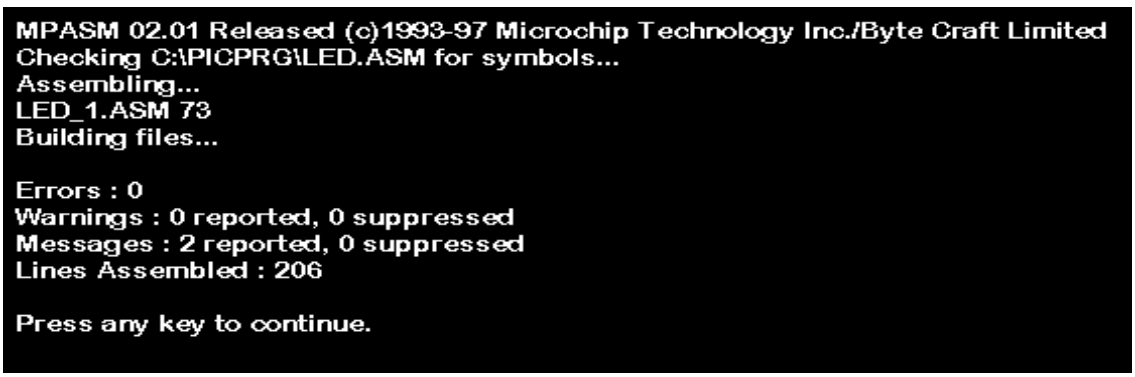

Figura 6 – Interface DOS do MPASM

- A Segunda forma, é feita através do ambiente Windows. Neste caso, inicia-se o programa MPASM for Windows e indica-se o diretório em que se encontra o arquivo a ser compilado, como sugere a figura 7 abaixo.

Figura 7 – Interface do MPASM for Windows

Nos dois casos os arquivos gerados são idênticos. Se tudo der certo deveremos ver os seguintes novos arquivos no diretório **C:\PICPRG**:

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

1.7. Programando o PIC 16C84

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

Para programar o PIC podemos utilizar o ambiente MPASM apresentado anteriormente e a interface de programacão PICSTART fornecidos pela MICROCHIP, cujo funcionamento e operação estão descritos nos respectivos manuais.

Outra opção, mais acessível e barata, é utilizar o programa opcional NOOP que, ao ser incorporado a interface de hardware (programador) própria comunicam-se com o computador "host" através da conexão com interface paralela. Neste caso, a documentação do software para realização da gravação do PIC, bem como o circuito para implementação da interface de programação, está disponível no diretório NOOP e será apresentada com o uso do sistema.

Para se usar esta opção, estando no diretório **C:\NOOP**, ao ser digitado no prompt do DOS o comando:

NOOP.EXE

o programa apresentará uma tela com a opção de escolha da porta paralela a ser utilizada (LPT1, LPT2 e etc). Em seguida, ao ser escolhida a porta paralela o programa testa a interface de hardware e o programador e apresenta as opções de escolha do tipo de PIC a ser gravado. No próximo passo, ele permite efetuar a carga (upload) do programa LED.HEX para a memória EEPROM do PIC gravando-o.

Uma vez concluído a programação devemos retirar o PIC do programador e montar o circuito da figura 5. Se tudo estiver certo, é só observar o LED piscar, como o esperado.

2. ESTUDO DA ARQUITETURA DO PIC 16C84

Mostraremos a seguir como é feito um PIC, que dispositivos ele contém e como eles se interagem. Na figura 8 está ilustrado o esquema de blocos simplificado da arquitetura interna do **PIC16C84** que nos ajudará a entender o que será explicado. Iniciaremos a nossa análise com a memória **EEPROM** e o **REGISTER FILE**.

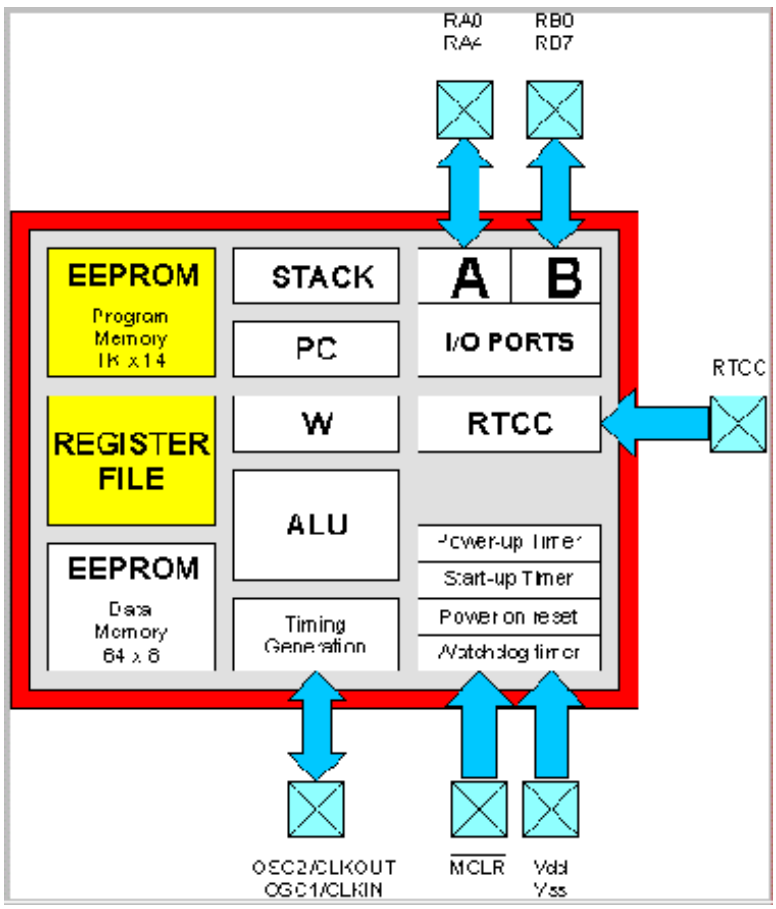

Figura 8 – Componentes internos do 16C84

2.1. A área de programa e o registrador de arquivo

A EEPROM é uma memória especial, apagável eletricamente e utilizada no PIC para memorizar o programa a ser executado. A sua capacidade de armazenamento é de **1024**

locações que pode conter, cada uma, somente um **opcode de 14 bits** , ou seja uma instrução básica do PIC.

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

Para escrever, ler ou apagar o conteúdo destas locações é necessario o dispositivo externo chamado programador.

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 no código LED.ASM apresentado anteriormente, 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 enderço 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 diretamante visível pelo resto do programa. Nesta podemos escrever, ler, ou modificar qualquer dado de um endereço do REGISTER e a qualquer momento que for necessario.

A única limitação consiste no fato 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 registro encontram-se nas locações base da área de memória RAM segundo o que esta ilustrado na figura 9.

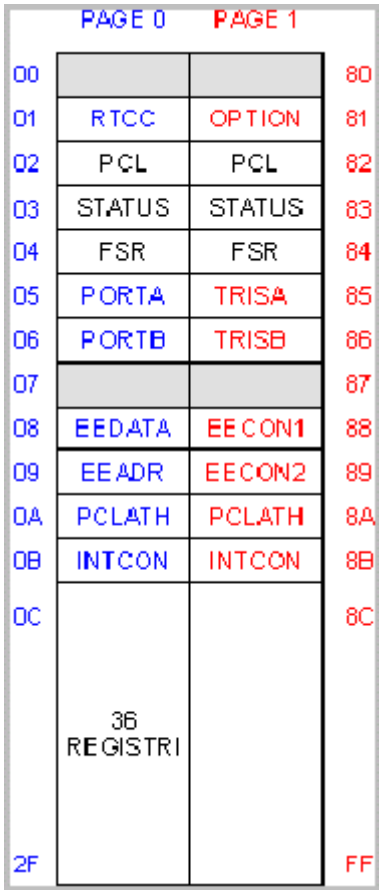

Figura 9 – Memória RAM

As locacões de memória presentes no REGISTER FILE são endereçadas diretamente em um espaço de memória que vai de **00H** a **2FH** num total de 48 bytes, denominada de **página 0**. Um segundo espaço de endereçamento, denominado de **página 1,** ocupa os endereços de **80H** até **AFH**. Para acessar esse segundo espaço é necessário recorrer a dois bits auxiliares **RP0** e **RP1** do registro STATUS, como iremos explicar mais adiante.

As primeiras **12 locações** da página 0 (de **00H** a **0BH**) e da página 1 (de **80H** a **8BH**) são aquelas reservadas às funções especiais para o funcionamento do PIC e não podem ser utilizadas para outra coisa.

As **36 locações** na **página 0** podem ser endereçadas de **0CH** a **2FH**. Neste caso, podemos utilizar pelo nosso programa para armazenamento de variáveis, contadores e etc.

No nosso exemplo LED.ASM a 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 locações, que o programa utilizará para armazenar o contador de retardo de tempo da subrotina Delay.

Os registros especiais do PIC serão utilizados com muita frequência nos programas, por exemplo, quando definimos um valor para os registros especiais **TRISA** e **TRISB** e definimos qual linha de I/O será usada como 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 indicarão o estado de funcionamento do dispositivo interno do PIC ou o resultado de operações aritméticas e lógicas. É necessário conhecer portanto exatamente qual a função desenvolvida por cada um dos registros especiais e qual efeito se obtém ao manipular seus conteúdos.

Para facilitar as operações de seus registros especiais, no arquivo P16C84.INC a Microchip inseriu uma lista de nomes que indentificam univocamente qualquer registro especial e a qual endereço correspondente está associado na área do REGISTER FILE. Sendo assim, se, por exemplo, quisermos definir toda a linha do PORTB do PIC como escrita (saída), agindo sobre o **TRISB** podemos escolher e referenciar diretamente o registro pelo seu endereço, como o exemplo:

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

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

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

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

2.2. A ALU (Unidade Lógica e Aritmética) e o registro W (Acumulador)

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

A **ALU** (**A**rithmetic and **L**ogic **U**nit, ou seja, unidade lógica e aritimética) é o componente mais complexo do PIC por conter todos os circuitos destinados a desenvolver as funções de cálculo e manipulação de dados durante a execução de um programa.

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

No nosso caso, diretamente, conhecemos a ALU como o **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 as outras locações de memória consiste no fato de que, por referenciar o registro W, a ALU não pode fornecer nenhum endereço específico, entretanto, podemos acessa-los diretamente.

O registro W será utilizado especificamente no programa pelo PIC. 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 os **14 bits** e assim teremos:

- **8 bits** para especificar o valor que queremos colocar na locação de memória;
- **7 bits** para especificar a locação de memória quer vamos inserir o nosso valor;
- **6 bits** para especificar qual instrução queremos usar.

Neste caso, teremos um total de **8 + 7 + 6 = 21 bits**. Portanto, devemos então recorrer a duas instruções, ou seja:

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

onde 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**.

2.3. O contador de programa e o Stack ou pilha

Nesta seção analizaremos o funcionamento do **Program Counter** e do **Stack,** dois componentes importantes para a compreensão da instrução de salto e chamada a subrotina.

Como vimos anteriormente, o **PIC16C84** inicia a execução do programa a partir do **Reset Vector,** ou seja da locação de memória **0000H**. Depois de ter executado esta instrução passa para a próxima instrução armazenada na locação **0001H** e assim por diante. Se não existisse uma ou mais instruções capazes de influenciarem na execução progressiva do programa, o PIC chegaria até o final na última instrução armazenada na última locação e não saberia mais como continuar.

Sabemos que não é bem assim que funciona. Qualquer sistema a microprocessador ou linguagem de programação dispõe de instruções de desvio, ou seja instruções capazes de modificarem o fluxo de execução do programa.

Uma destas instruções é o **goto** (do inglês **go to**). Quando o PIC encontra um **goto** não segue mais a instrução imediatamente após, mas sim desvia-se diretamante para a locação de memória especificada na instrução. Façamos um exemplo :

```
      ORG   00H

Ponto1
      movlw 10
```

```
      goto  Ponto1
```

No reset o PIC seguirá a instrução **movlw 10** memorizada na locação **0000H** que colocará no acumulador o valor decimal 10. Em seguida, passará a executar a próxima instrução, ou seja, **goto Ponto1**. Esta instrução determinará um desvio incondicional para a locação de memória especificada pelo label **Ponto1,** ou seja, de novo para locação **0000H**.

O programa não fará outra coisa se não a de executar um ciclo infinito seguindo continuamente as instruções especificadas. Durante este ciclo, para determinar qual é a próxima instrução a ser seguida, o PIC utiliza um registro especial denominado **Program Counter** ou contador de programa.

Este terá sempre o endereço da próxima instrução a ser executada. No reset ele estará sempre zerado, determinando o início da execução no endereço 0000H, e a cada instrução terá um incremento de um para poder passar para a próxima instrução.

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

Uma outra instrução muito interessante é a **call,** ou seja a chamada a subrotina. Esta instrução funciona de maneira muito similar ao **goto** com a única diferença de que, a primeira, desvia para uma locação de memória especificada e continua a execução do programa, enquanto o **call** desviará o programa para uma subrotina especificada que executará a mesma.

Em seguida, ao ser encontrada a instrução **return** o **Program Counter** retornará à execução da instrução imediatamente após a chamada **call**. O valor imediatamente após a chamada **call** foi armazenado em uma área particular da memória denominada **Stack ou pilha** e sua recuperação será feita ao ser encontrada a instrução **return** dentro da subrotina. Vejamos melhor um exemplo:

```
      ORG   00H

Ponto1
      movlw 10
      call  Ponto2
      goto  Ponto1

Ponto2
      movlw 11
      return
```

Neste caso o PIC, após ter executado **movlw 10** passa a executar o **call Ponto2**. Antes de desviar, o **Program Counter** 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 **Ponto2**. E neste ponto encontra uma nova instrução o **return** que, como podemos deduzir de seu nome, permite o "RETORNO", ou seja, retorna a execução da instrução imediatamente após o **call**.

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

Para poder retornar de 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 "**pilha**" e por esse fato é possivel empilhar um endereço sobre o outro para ser recuperado quando necessário. Este tipo de memorização é 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 primeniro a sair (last out).

Graças ao Stack é possivel efetuar vários **call** (oito no caso do PIC16C84) 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 a subrotina **Point2**, a subrotina **Point2** chama outra subrotina no caso **Point3**, esta última por sua vez, encontra um **return** e retorna para **Point2** que encontra o outro **return** e retorna para a execução da rotina **Point1,** que no caso é a rotina principal do programa.

Os endereços a serem memorizados no stack são dois e quando for encontrado um segundo **call** o **Program Counter** procurará pelo **return** correspondente ao primeiro e assim por diante. Se diz então que o **call** é "**aninhado**", ou seja, um dentro do outro.

O **PIC16C84** dispõe de um **stack de 8 niveis**, ou seja um Stack que consegue armazenar no máximo 8 chamadas a subrotinas.

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

2.4. Segunda aplicação - Implementando um sequencial de leds

Faremos agora uma reestruturação do código LED.ASM apresentado anteriormente fazendo-o realizar um sequencial de quatro led's. E o novo código modificado se chamará SEQ.ASM, como listado a seguir:

```
;*************************************************
; SEQ.ASM
;
;*************************************************
        PROCESSOR       16F84
        RADIX           DEC
        INCLUDE         "P16F84.INC"

        ORG     0CH

Count   RES     2
Shift   RES     1

        ;Reset Vector
        ;Ponto de inicio do programa ao reset da CPU

        ORG     00H

        bsf     STATUS,RP0
        movlw   00011111B
        movwf   TRISA
        movlw   11110000B
        movwf   TRISB
        bcf     STATUS,RP0
        movlw   00000001B
        movwf   Shift

MainLoop

        movf    Shift,W
        movwf   PORTB
        bcf     STATUS,C
        rlf     Shift,F
        btfsc   Shift,4
        swapf   Shift,F
        call    Delay
        goto    MainLoop

        ; Subrotinas

Delay
        clrf    Count
        clrf    Count+1

DelayLoop

        decfsz  Count,1
        goto    DelayLoop
```

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

O circuito a ser realizado está representado na figura 10, sendo substancialmente equivalente ao primeiro circuito apresentado, com a única diferença na quantidade de led's conectados, quatro ao invés de um.

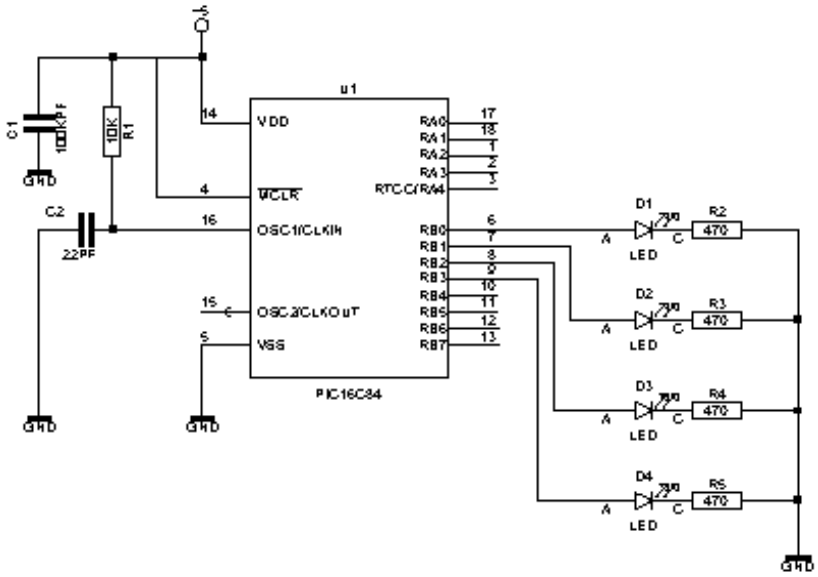

Figura 10 – Circuito sequencial com led’s

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

```
        movlw 11111110B
        movwf TRISB
para
        movlw 11110000B
        movwf TRISB
```

em que os quatro bits menos significativos, correspondentes as linhas RB0,1,2,3 foram colocados a nível zero para definir esta linha como 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 início de funcionamento do primeiro led, a instrução será a seguinte:

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

que então efetuará 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 mais adiante. **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-o 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 de 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 em ciclos sucessivos.

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

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

A instrução **btfsc Shift,4** controla o momento em que o bit 4 do registro Shift venha valer 1. Depois, executa-se a próxima instrução **swapf Shift,F**, e prossegue.

A instrução **swap** (do inglês "troca") na prática troca o quarto bit mais significativo contido no registro **Shift**, pelo seu quarto bit 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 prática faz acender o primeiro led.

3. INTRODUÇÃO AOS PERIFÉRICOS DO PIC

O **PIC16C84** dispõe 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 como entradas ou como saídas. Elas são identificadas pelas siglas RA0, RA1, RA2, RA3 e RA4. A **PORTA B** dispõe de **8 linhas** também configuráveis seja como entrada, ou como saída. Elas são identificadas pelas siglas RB0, RB1, RB2, RB3, RB4, RB5, RB6 e RB7.

A subdivisão das linhas em duas portas diferentes é devido ao tipo de arquitetura interna do **PIC16C84** que prevê um controle de dados de no máximo 8 bits.

3.1. As portas de entrada e saída A e B

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 **TRIS B**, determinarão o funcionamento em entrada ou em saída da mesma linha, e os registros **PORT A** e **PORT B** determinarão o **status** da linha em saída ou indicarão 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 registro **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** determinará o estado lógico de tal linha (0 = 0 volts, 1 = 5 volts).

Se o **bit 0** do registro **TRISA** vier a ser 1 (**um)**, a linha **RA0** será configurada como **linha de entrada**. Façamos um exemplo prático, imaginemos a conexão de um led na linha RB0 e uma chave sobre a linha RB4, o código a ser escrito será o seguinte:

```
movlw  00010000B
tris   B
```

onde será colocado a nível 0 o bit 0 (linha RB0) em escrita ou saída, e a nível 1 o bit 4 (linha RB4) em entrada. Lembrando de tal propósito que na notação binária do assembler o bit mais a direita corresponde ao 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 ler-mos o estado da chave conectada a linha RB4, o código será:

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

3.2. Estado de escrita das linhas de I/O

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

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

Iniciaremos o estudo pelo grupo das linhas **RA0**, **RA1**, **RA2** e **RA3** nas quais representamos na figura 11 o esquema do estado de escrita extraído do **data sheet** da Microchip:

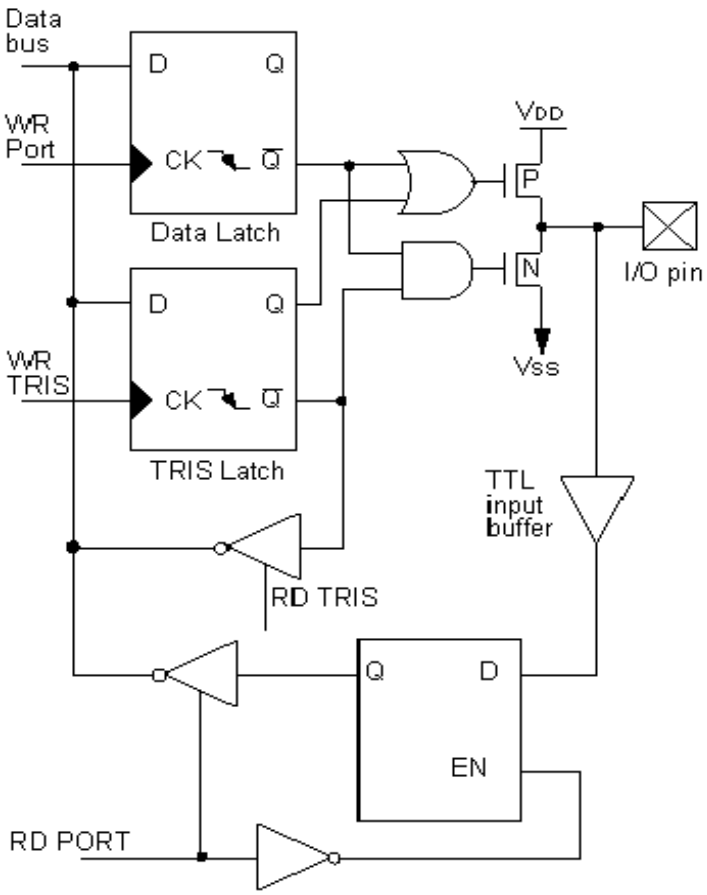

Figura 11 – Estado de escrita das linhas RA0 a RA3

Como foi dito anteriormente, 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**).

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

Funcionamento como entrada

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

```
bsf     TRISA,0
```

Esta instrução determinará uma comutação para nível 1 do estado lógico do **flip-flop** do tipo **D-latch** indicado no bloco com o nome **TRIS latch**. Para cada uma das outras linhas de I/O existe um destes flip-flop e o estado lógico em que o mesmo se trava depende estritamente do estado lógico do relativo bit no registro **TRIS** (ou melhor dizendo, todos os bit's do registro TRIS são fisicamente implementados 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 na outra entrada, a saída da porta OR estará sempre em nível 1 enquanto uma de suas entradas valer 1. E nesta condição o **transistor P** não conduz mantendo a linha RA0 desconectada do pólo positivo da alimentação.

Do mesmo modo a saída negativa **Q** do **TRIS** latch é conectada a entrada de uma porta **AND** onde a saída desta estará sempre em nível 0 enquanto uma de suas entradas valer 0. E nesta condição, o **transistor N** não conduz e mantém a linha RA0 desconectada da massa. O estado lógico 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 como saída

Para configurar a linha RA0 como saída, devemos colocar em nível 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). 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 TRIS latch, em que o Data Latch depende do estado de um bit em um registro, particularmente do registro **PORTA**. A sua saída negativa será enviada para a entrada das duas portas lógicas **OR** e **AND** e que estão diretamente sobre a base do **transistor P** e **N**. Se colocarmos em nível 0 o bit 0 do registro PORTA com a instrução:

```
bcf     PORTA,0
```

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

```
bsf     PORTA,0
```

obteremos a condução do **transistor P** e portanto irá a nível 1 (+5 Volts) a linha RA0. Nesta condição será sempre possível rever o valor enviado sobre a linha através do circuito de entrada.

Estado de saída da linha RA4

Analisaremos o funcionamento do estado de saída da linha **RA4.** Esta é diferente de todas as outras linhas de I/O uma vez que compartilha o mesmo pino do PIC16C84 com o **TOCKI,** o qual iremos analisar mais adiante.

Na figura 12, está descrito o esquema de blocos do estado de saída extraído do **data sheet** da Microchip:

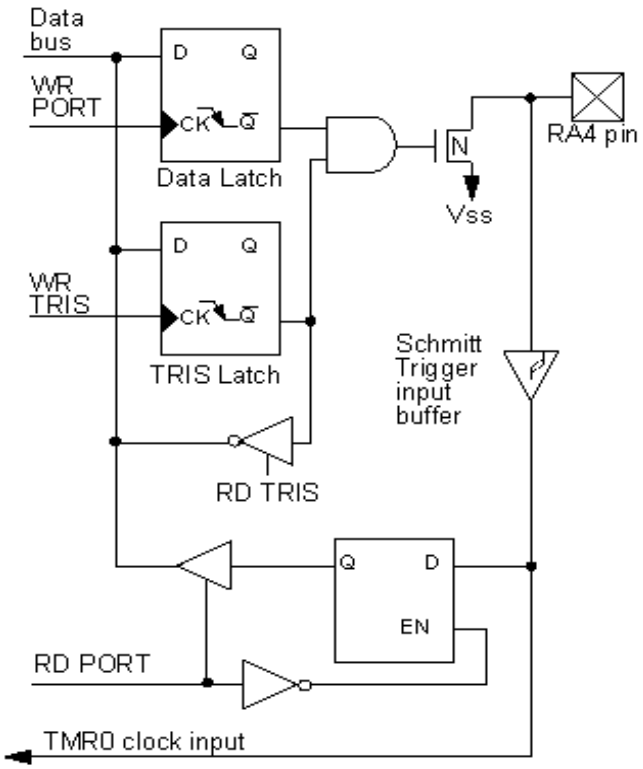

Figura 12 – Estado de saída da linha RA4

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 pólo positivo da alimentação, da linha RA4. Isto significa em termos

práticos, que quando a linha RA4 esta programada em saída, ela poderá assumir um nível que dependerá do circuito externo.

Este tipo de circuito de saída é chamado de "**coletor aberto**" e é útil para aplicações em que é necessário compartilhar uma mesma ligação com mais pinos de saída ou que se tenha a necessidade de colocar em alta impedância uma linha de saída, podendo assim reprogramá-la como linha de entrada.

Se quisermos garantir que a linha RA4 vá a nível 1 devemos conectar externamente um resistor de **pull-up**, ou seja, um resistor deverá ser conectado ao pólo positivo da alimentação. Veremos em seguida a utilização da linha indicada no esquema acima **TMR0 clock input**.

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

Neste grupo de linhas, permanece substancialmente a mesma lógica de comutação. Estas dispõem de um circuito a mais, o **weak pull-up**, que é ativado quando a linha for programada como entrada. Como entrada, como explicado anteriormente, a linha fica 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 vindo do pólo positivo para tornar seguro o manuseio quando a chave for solta, e garantir a volta do nível a uma condição lógica 1 estável 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 13, está representado o esquema de blocos do estado de saída extraído do data sheet da Microchip:

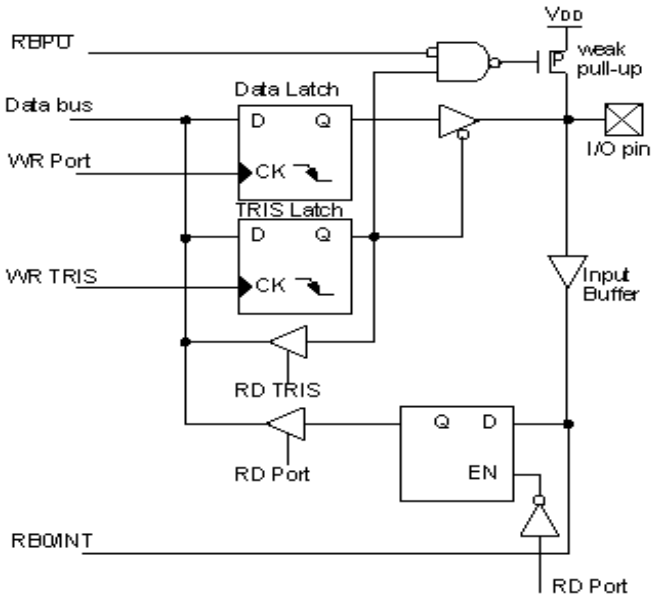

Figura 13 – Estado de saída das linhas RB0 a RB3

Além disso a linha **RB0**, apresenta uma característica muito particular. Ela pode ser configurada como linha de entrada para permitir uma troca de estado lógico em função da ocorrência de uma interrupção (**interrupt).** Ou seja, uma interrupção imediata do programa em execução e uma consequente chamada de uma subrotina especial denominada de tratamento da interrupção (**interrupt handler)**, que falaremos a respeito mais adiante.

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

O circuito de comutação deste grupo de linhas é idêntico ao grupo RB0 a RB3. Esta linha dispõe também de um circuito de **weak pull-up**. E mais, em relação as linhas RB0 a RB3, tem a vantagem de poder permitir variações de estado sobre qualquer linha e gerar uma interrupção. Na figura 14 está reproduzido o esquema de blocos do estado de saída das linhas RB4 a RB7 extraído do data sheet da Microchip.

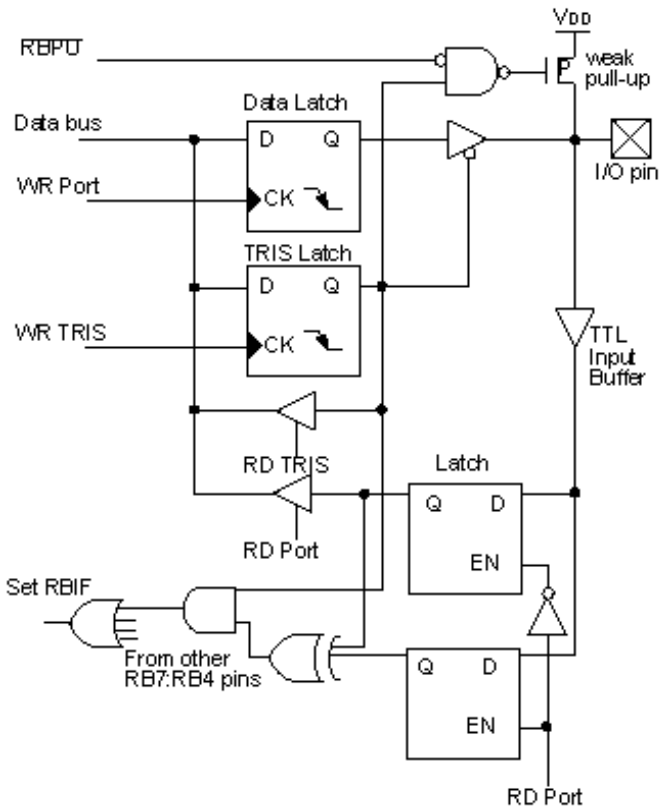

Figura 14 – Estado de saída das linhas RB4 a RB7

3.3. Terceira aplicação - Implementando um circuito com teclas

Após a implementação, no exemplo anterior, do sequencial de leds usando as linhas RB0 a RB3 como linhas de saída, vamos ver um novo exemplo de como se pode realizar uma entrada com teclas configurando as linhas de RB4 a RB7 como linhas de entrada. Para tanto, usaremos quatro chaves **SW1**, **SW2**, **SW3** e **SW4**.

Cada uma destas chaves será conectada entre o terminal de +5V e a linha de entrada. As linhas de entrada serão mantidas em 0 V por resistores (R1 a R4) ligados entre elas e a massa.

Sendo assim, se conectarmos o pino 10 do PIC16C84 à chave SW1, esta linha será mantida em nível 0 até que seja pressionada a tecla correspondente SW1. Isto provocará a mudança do estado da linha para nível 1.

Para o exemplo, faremos um programa que acenderá qualquer um dos leds **D1**, **D2**, **D3** e **D4** em correspondência com o fechamento das teclas **SW1** a **SW4**. O circuito a ser montado, após a gravação do PIC, está representado na figura 15 abaixo.

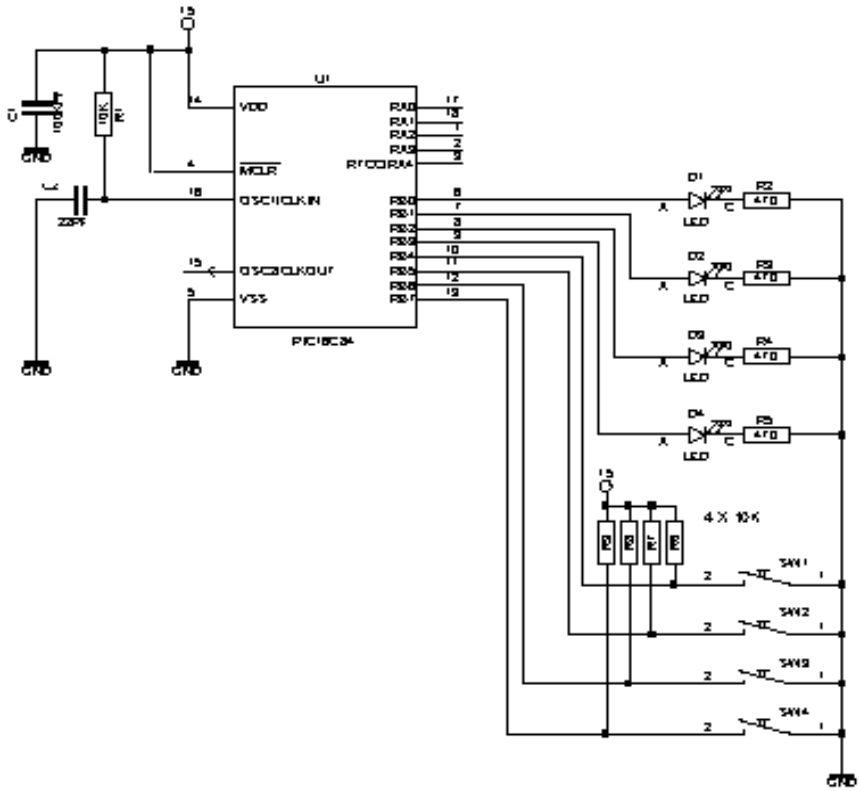

Figura 15 – Circuito de entrada com teclas

O código do exemplo é descrito no arquivo INPUT.ASM apresentado abaixo e que será analisado por nós.

```
;*************************************************
```

```
; INPUT.ASM
;************************************************
        PROCESSOR       16F84
        RADIX           DEC
        INCLUDE         "P16F84.INC"

LED1    EQU     0
LED2    EQU     1
LED3    EQU     2
LED4    EQU     3
SW1     EQU     4
SW2     EQU     5
SW3     EQU     6
SW4     EQU     7

        ORG     0CH
        ;Reset Vector
        ;Início do programa com reset da CPU
        ORG     00H

        ;Troca para o Bank 1 do register file para acessar os
        ;registros TRISA e TRISB
        bsf     STATUS,RP0

        ;Definicao da linha de I/O (0=saida, 1=entrada)
        ;Definicao da porta A
        movlw   00011111B
        movwf   TRISA

        ; Definicao da porta B
        ;As linhas de RB0 a RB3 sao programadas como saida
        ; As linhas de RB4 a RB7 sao programadas como entrada
        movlw   11110000B
        movwf   TRISB
        ;Retorna ao Bank 1
        bcf     STATUS,RP0

MainLoop
        ;Apague todos os leds
        clrf    PORTB

        ;Se for pressionado a chave SW1 acende o LED1
        btfss   PORTB,SW1
        bsf     PORTB,LED1
        ; Se for pressionado a chave SW2 acende o LED2
        btfss   PORTB,SW2
        bsf     PORTB,LED2
        ; Se for pressionado a chave SW3 acende o LED3
        btfss   PORTB,SW3
        bsf     PORTB,LED3
        ; Se for pressionado a chave SW4 acende o LED4
        btfss   PORTB,SW4
        bsf     PORTB,LED4

        goto    MainLoop
        END
```

A parte inicial do programa tem as mesmas funções do exemplo anterior e é formado pelas instruções:

```
movlw 11110000B
movwf TRISB
```

configurando as linhas RB0 a RB3 como saída para a conexão com os led's, e as linhas RB4 a RB7 como entradas para a conexão com as quatro chaves.

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

efetua simplesmente um loop contínuo. Neste, a instrução **Swapf** troca o conteúdo dos bits **RB0-RB3** com o conteúdo dos bits **RB4-RB7**. E deste modo, o estado das linhas de entrada conectadas as quatro chaves serão alterados continuamente sobre as linhas conectadas aos led's, obtendo-se assim, o acendimento de um led para cada chave.

4. O CONTADOR TMR0 E O PRESCALER

O PIC 16C84 permite acessar um contador de 8 bit's internamente chamado TIMER0, aqui abreviado como TMR0.

4.1. O registro contador TMR0

O registro TMR0 é um contador de 8 bits, ou seja. Ele é um registro particular, no qual seu conteúdo é incrementado com a entrada regular de pulsos no pino RA4/TOCKI. Esta entrada é programada diretamente pelo hardware do PIC. Na prática, a diferença ente ele e outro registro qualquer, é que o TMR0 não mantém inalterado o valor que é armazenado, mas o incrementa continuamente, se por exemplo escrevermos nele o valor 10 com a instrução:

```
movlw   10
movwf   TMR0
```

Após um tempo 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 cadência constante e independente da execução do resto do programa.

Se por exemplo, após ter colocado um valor no registro TMR0, executar-mos um loop infinito do tipo:

```
        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 agora, com valor igual a zero.

A frequência é diretamente proporcional à frequência de clock aplicada ao chip e pode ser modificada programando-se os seus bits de configuração. Na figura 16 está representada a cadeia de blocos interno do PIC que determina o funcionamento do registro TMR0.

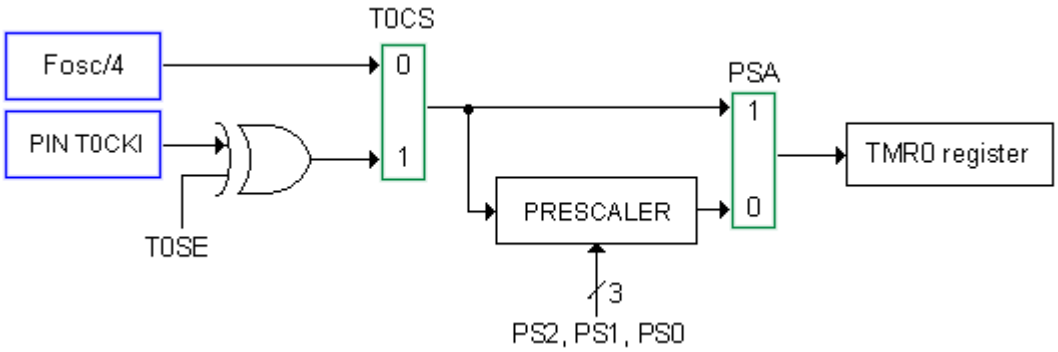

Figura 16 – Diagrama de blocos do registro TMR0

Os blocos **Fosc/4** e **T0CKI** 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 por um eventual circuito externo de pulsos e aplicado ao pino **T0CKI** correspondente ao pino 3 no PIC 16C84.

O blocos **T0CS** e **PSA** são dois comutadores de sinal na qual estão representando um dos dois tipos de sinal de entrada com base no valor dos bits **TOCS** e **PSA** do registro **OPTION.**

O bloco **PRESCALER** é um divisor programável e seu funcionamento será detalhado mais adiante.

A seguir, veremos na prática 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 que está representada na figura 17 abaixo:

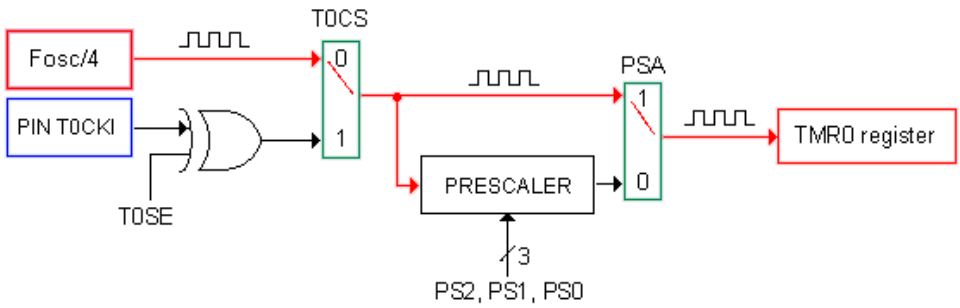

Figura 17 – Programação do bit TOCS em 0 e PSA em 1

Como foi dito anteriormente, a frequência Fosc/4 é par e igual a um quarto da frequência de clock. Se utilizarmos um cristal de quartzo de 4Mhz teremos uma frequência de

1 MHz. Tal frequência será enviada diretamente ao registro TMR0 sem nenhuma alteração. A cadência de contagem que se obtém é então de 1 milhão de incrementos por segundo do valor presente no TMR0.

Imaginemos agora que alteraremos o status do bit **T0CS de 0 para 1**, neste caso, a configuração que obteremos é a da figura 18 a seguir.

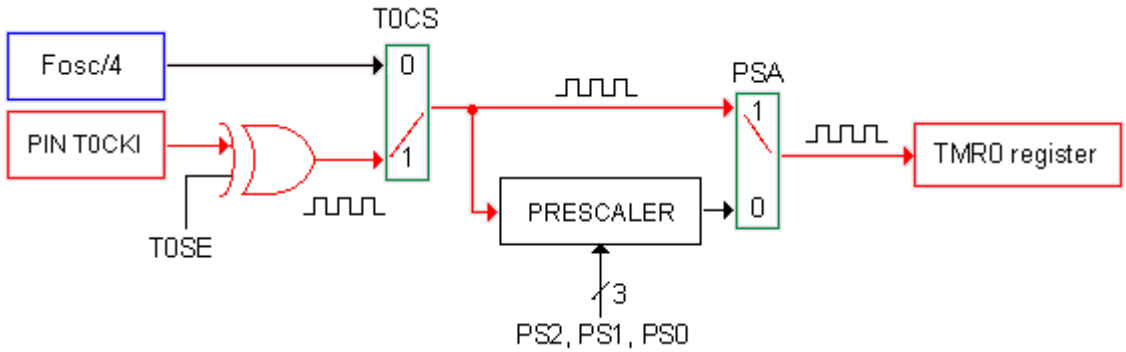

Figura 18 – Alteração do bit TOCS de 0 para 1

Neste caso, é o sinal externo aplicado ao pino TOCKI do PIC que será enviado diretamente ao contador TMR0 determinando a frequência de contagem. Aplicando-se por exemplo a este pino uma frequência de 100Hz obteremos uma frequência de contagem de 100 (cem) incrementos por segundo.

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

Na figura 19 está representada a correspondência entre a frequência do sinal externo e o valor que assumirá o contador TMR0 :

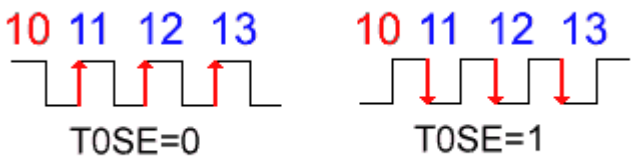

Figura 19 – Relação entre o sinal externo e o valo de TMR0

4.1.1. O PRESCALER e sua relação com o TMR0

O último bloco a ser analisado para permitir a utilização completa do 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 mostra a figura 20 abaixo.

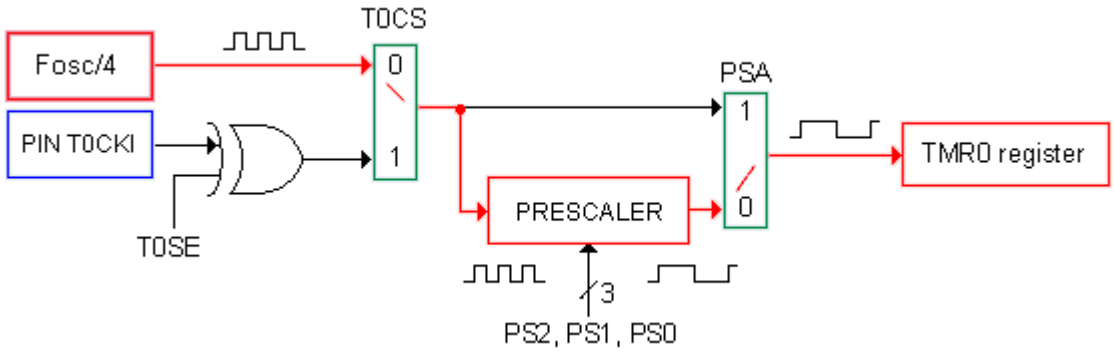

Figura 20 – Bit PSA do registro OPTION em 0 e o comportamento deTMR0

O PRESCALER consiste na prática de um divisor programável de 8 bits utilizado no caso pela frequência de contagem enviada ao contador TMR0.

Anteriormente, vimos que utilizando um cristal de 4Mhz obtínhamos uma frequência de contagem de 1 Mhz. Este valor de frequência para muitas aplicações poderá ser muito alto.

Com o uso do PRESCALER podemos dividir interiormente a frequência Fosc/4 configurando desta forma os bits **PS0**, **PS1**, **PS2** do registro **OPTION** de acordo com a tabela 3 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

Tabela 3 – O PRESCALER como divisor de frequência

Iremos implementar agora um experimento sobre o que foi visto até aqui para exercitarmos o aprendizado. Vamos rescrever a subrotina do programa do “sequencial de leds” para introduzir um atraso de um segundo utilizando o registro **TMR0**. O novo programa modificado receberá o nome de SEQTMR0.ASM, e está listado abaixo.

```
;************************************************
; SEQTMR0.ASM
;************************************************
        PROCESSOR       16F84
        RADIX           DEC
        INCLUDE         "P16F84.INC"

        ORG     0CH
Count   RES     1
Shift   RES     1

        ;Reset Vector
        ; Inicio
        ORG     00H

        ;Bank 1
        bsf     STATUS,RP0

        ;Define linhas de I/O (0=saida, 1=entrada)
        movlw   00011111B
        movwf   TRISA
        movlw   11110000B
        movwf   TRISB

        ;Aponta o PRESCALER para TMR0 e o configura com 1:32
        movlw   00000100B
        movwf   OPTION_REG

        ;Bank 0
        bcf     STATUS,RP0

        movlw   00000001B
        movwf   Shift

        ;Loop
MainLoop

        ;envia para porta B 0 registro Shift

        movf    Shift,W
        movwf   PORTB

        bcf STATUS,C.
        bcf     STATUS,C
        rlf     Shift,F

        ; 00000001 <--- Valor inicial (primeiro led aceso)
        ; 00000010 rlf
        ; 00000100 rlf
        ; 00001000 rlf
```

```
        ; 00010000 rlf

        ; 00000001 ...

        btfsc   Shift,4
        swapf   Shift,F

        ;Insere um atraso tempo
        call    Delay
        goto    MainLoop

        ;**************
        ; Subrotinas
        ;**************

        ; Insere um retardo de um segundo
        ; utilizando o registro TMR0
        ;
        ; 4Mhz / 4 / 32 = 31.250 Hz
        ; ... divide por 250 o TMR0 32.250 / 250 = 125 Hz
        ; ... e por 125 da contador Count 125 / 125 = 1Hz

Delay
        ; Inicializa TMR0 para 250 e va a zero.

        movlw   6
        movwf   TMR0
        movlw   125
        movwf   Count
        ;Loop

DelayLoop
        ;TMR0 vale 0 ?
        movf    TMR0,W
        btfss   STATUS,Z
        goto    DelayLoop
        movlw   6
        movwf   TMR0
        decfsz  Count,1
        goto    DelayLoop
        return

        END
```

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 prática 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 depois) e o bit de configuração do **PRESCALER** em **100** para obter uma frequência de divisão de 1:32. A frequência de contagem que obteremos sobre TMR0 será:

Fosc = 1Mhz / 32 = 31.250 Hz

A subrotina Delay deverá utilizar o registro TMR0 para obter um retardo de um segundo. Isto será feito como descrito mais adiante.

As primeiras instruções que virão em seguida no Delay serão:

```
        movlw   6
        movwf   TMR0
e
        movlw   125
        movwf   Count
```

As duas primeiras armazenam 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 armazenará em um registro de 8 bits (Count) o valor 125 de tal modo que, decrementará este registro de um a cada passagem pelo zero do TMR0, até que se obtenha uma frequ6encia de passagem pelo zero do registro Count de:

125/125 = 1Hz

A instrução inserida no **loop DelayLoop** se encarregará então de controlar se o TMR0 já chegou a zero, quando então o reinicializará em 6 e decrementará o valor contido em Count. Quando Count alcançar o zero ai terá transcorrido um segundo e a subrotina poderá retornar ao programa que a chamou.

5. AS INTERRUPÇÕES NO PIC 16C84

A interrupção é uma técnica particular do PIC que permite interceptar eventos externos ao programa em execução. Ele interrompe momentaneamente a operação do programa em andamento, controla o evento com uma subrotina apropriada e retorna para a execução do programa.

5.1. Tipos de eventos e bit's de habilitação da interrupção

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

- A troca de estado sobre a linha RB0 (External interrupt RB0/INT pin).
- Ao final da contagem do registro TMR0 (TMR0 overflow interrupt).
- A troca de estado sobre uma das linhas de RB4 a RB7 (PORTB change interrupts).
- 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 os seguintes bit's do registro **INTCON**:

- **INTE** (bit 4) se este bit estiver em nível 1 habilitará a interrupção de troca 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 a interrupção de troca de estado sobre uma das linhas de **RB4** a **RB7** ficará habilitada.
- **EEIE** (bit 6) se este bit estiver em 1 a interrupção de final de escrita sobre um endereço da **EEPROM** ficará habilitada.

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

5.2. 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 subrotina de controle denominada **Interrupt Handler** (controle de interrupção).

Pode-se habilitar mais de uma interrupção e, a primeira providência da interrupt handler é 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**.

5.3. Interrupt flag (sinalizador de interrupção)

Dado que qualquer interrupção gera uma chamada do endereço **04H.** No registro **INTCON** esta presente o flag que indica qual o evento gerou a interrupção em questão, vejamos:

- **INTF** (bit 1) Se for 1 a interrupção foi gerada na troca de estado sobre a linha RB0.
- **T0IF** (bit 2) Se vale 1 a interrupção foi gerada no término da contagem do TMR0.
- **RBIF** (bit 0) Se for 1 a interrupção é devido a 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 uma interrup handler deva considerar que esta interrupção é devido a um evento quando todos os três flags supracitados irão a nível 0.

Importante: Uma vez conhecido qual o flag está ativo, a interrupt handler deve zerá-lo, sob pena de não mais ser gerada uma interrupção correspondente.

5.4. Retorno de uma interrupt handler

Quando for gerada uma interrupção o PIC desabilitará 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 nível 1 este bit, deve-se utilizar a instrução :

```
RETFIE
```

5.5. Exemplo prático de controle de interrupção

Veremos agora um exemplo prático de controle de interrupção. Tomaremos como base o código LED.ASM usado como primeiro exemplo para realizar um pisca-pisca a led.

Como recordamos, este programa faz piscar o LED1 num ciclo contínuo utilizando um retardo de software introduzido da subrotina **Delay**. Vejamos então como é possível acender o LED2 pressionando uma tecla qualquer com a execução do programa principal.

O código do exemplo que iremos analisar receberá o nome de INTRB.ASM e está listado a seguir.

```
;*************************************************
; INTRB.ASM
;*************************************************
        PROCESSOR       16C84
                          RADIX           DEC
        INCLUDE         "P16C84.INC"

LED1    EQU     0
LED2    EQU     1
LED3    EQU     2
LED4    EQU     3

        ORG     0CH

Count   RES     2
nTick   RES     1     ;Registro utilizado para contar o numero de
                      ;    piscados do LED 1

        ;Reset Vector
        ;Inicio da CPU

        ORG     00H

        ;Salta o corpo principal do programa. Este jump e
       ;necessario
        ;para evitar toda a parte de codigo para a gestao de
        ;interrupcao.

        goto    Start

        ;Interrupt vector
        ;Inicio para a subrotina de tratamento da interrupcao

        ORG     04H


;*****************************************************************
        ; Interrupt handler

;*****************************************************************

        ;Acende o led 2 para sinalizar o estado de interrupt
        bsf     PORTB,LED2

        ;Inicializa o contador de piscado doLED1
        movlw   3
        movwf   nTick

        ;Zera novamente o flag RBIF
        bcf     INTCON,RBIF

        ;Retorna ao programa principal
        retfie


;*****************************************************************
        ; Programa principal
```

```
;*****************************************************************
     Start:
             ;Bank 1

             bsf     STATUS,RP0

             movlw   00011111B
             movwf   TRISA


             movlw   11110000B
             movwf   TRISB

             ;Bank 0

             bcf     STATUS,RP0

             bcf     PORTB,LED1
             bcf     PORTB,LED2
             bcf     PORTB,LED3
             bcf     PORTB,LED4

             ;Habilita interrupcoes via troca de esta de RB4,5,6,7
             movlw   10001000B
             movwf   INTCON


;*****************************************************************
             ; Loop principal

;*****************************************************************

     MainLoop

             call    Delay                   ;Retardo de software

             btfss   PORTB,LED1              ;Led aceso ?
             goto    TurnOnLed1              ;Nao, entao acende-o
             goto    TurnOffLed1             ;Sim, entao apague-o

     TurnOnLed1
             bsf     PORTB,LED1

             ;Controla se LED 2 continua aceso.
             ;Se esta aceso decrementa o contador nTick faz piscado do
             ;LED1. Quando nTick vale 0 apague LED 2

             btfss   PORTB,LED2      ;LED2 aceso ?
             goto    MainLoop        ;Nao, continua a piscar

             decf    nTick,1         ;Sim, decrementa nTick
             btfss   STATUS,Z        ;nTick = 0 ?
             goto    MainLoop        ;Nao, continua a piscar

             bcf     PORTB,LED2      ;Sim, apague LED2

             goto    MainLoop        ;Continua a piscar
```

```
TurnOffLed1

        bcf     PORTB,LED1      ;Apague LED 1
        goto    MainLoop        ;Continua a piscar


;*****************************************************************
        ; Subrotina
;*****************************************************************

        ;Subrotina de retardo de software

Delay
        clrf    Count
        clrf    Count+1

DelayLoop

        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

O circuito a ser realizado esta representado na figura 21 abaixo.

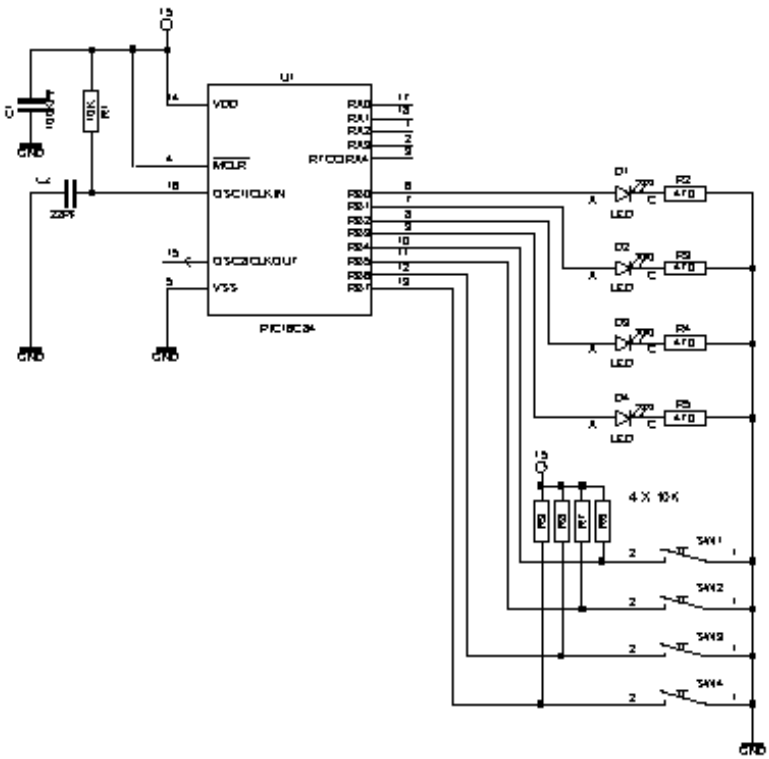

Figura 21 – O sequencial controlado por interrupção

Uma vez carregado o programa INTRB.ASM no PIC notaremos que o **LED D1** piscará exatamente como no programa LED.ASM. Devemos agora pressionar as teclas de **SW1** a **SW4**. Veremos que o **LED D2** se acende imediatamente e permanece aceso por um tempo de 3 piscadelas do LED 1.

Na prática o loop principal, derivado do código LED.ASM, continua a fazer piscar o LED D1 utilizando um retardo de software introduzido da subrotina **Delay**. O PIC fará com que ao pressionarmos uma tecla, o mesmo sinalizará imediatamente sobre o LED 2 sem influenciar de maneira evidente na frequência de piscadas.

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

Para este propósito compilaremos e colocaremos no circuito o programa NOINTRB.ASM listado abaixo. Notaremos que o acendimento do led 2 em correspondência com o pressionar de uma tecla é ligeiramente retardado enquanto a leitura do estado das linhas RB4-RA7 não serão efetuadas pelo hardware de controle de interrupção mas diretamente do programa principal a cada ciclo do loop.

```
;************************************************
; NOINTRB.ASM
;************************************************
        PROCESSOR       16C84
        RADIX           DEC
        INCLUDE         "P16C84.INC"

LED1    EQU     0
LED2    EQU     1
LED3    EQU     2
LED4    EQU     3

        ORG     0CH

Count   RES     2
nTick   RES     1

        ;Reset Vector
       ORG     00H


;*****************************************************************
*
            ; Programa principal

;*****************************************************************
*
Start:
            bsf     STATUS,RP0

            movlw   00011111B
            movwf   TRISA
            movlw   11110000B
            movwf   TRISB

            bcf     STATUS,RP0
```

```
            bcf     PORTB,LED1
            bcf     PORTB,LED2
            bcf     PORTB,LED3
            bcf     PORTB,LED4


;*****************************************************************
*
            ; Loop principal

;*****************************************************************
*

      MainLoop

      KeyCheck
            btfsc   PORTB,LED2
            goto    _KeyCheck

            comf    PORTB,W
            andlw   0F0H
            btfsc   STATUS,Z
            goto    _KeyCheck

            bsf     PORTB,LED2

            movlw   3
            movwf   nTick

      _KeyCheck

            call    Delay

            btfss   PORTB,LED1
            goto    TurnOnLed1
            goto    TurnOffLed1

            TurnOnLed1
            bsf     PORTB,LED1

            btfss   PORTB,LED2
            goto    MainLoop

            decf    nTick,1
            btfss   STATUS,Z
            goto    MainLoop

            bcf     PORTB,LED2

            goto    MainLoop

      TurnOffLed1

            bcf     PORTB,LED1
            goto    MainLoop
```

```
;*****************************************************************
;
*
              ; Subrotina

;*****************************************************************
;
*
              ;Subrotina de retardo software

      Delay
              clrf    Count
              clrf    Count+1

      DelayLoop

              decfsz  Count,1
              goto    DelayLoop

              decfsz  Count+1,1
              goto    DelayLoop

              return

              END
```

O rápido retardo é então devolvido a presença da subrotina **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 substancialmente que a primeira instrução que o PIC encontra é um desvio incondicional para o label **Start**:

```
ORG 00H
goto Start
```

seguido de uma nova diretiva:

```
ORG 04H
```

e então do código da subrotina de controle da interrupção:

```
bsf PORTB,LED2
movlw 3
movwf nTick
bcf INTCON,RBIF
retfie
```

Como havíamos dito, a interrupt Handler aloca-se necessariamente a partir do endereço 04H e para evitar que seja executada logo após o reset do PIC, devemos salta-la com uma instrução de salto incondicional. O código da interrupt handler, neste caso, é muito simples e se limita a acender o LED D2, que insere no registro existente **nTick** o número de piscadelas até chegar a LED D2, deverá apagar-se e zerar o flag **RBIF** para permitir ao circuito a geração de uma nova interrupção.

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

A interrupção é gerada ao ser pressionada uma tecla, pois a primeira instrução que o PIC executa após o reset é a seguinte:

```
movlw 10001000B
movwf INTCON
```

onde na prática foi colocado a nível 1 (um) o bit **GIE** (bit 7) que é a habilitação global das interrupções e em especial, o bit **RBIF** (bit 3) que habilita, particularmente, a interrupção sobre troca de estado das linhas RB4-RB7. Portanto havendo o fechamento das chaves SW1,SW2, SW3 e SW4 sobre a linha de I/O RB4, RB5, RB6 e RB7, com um pressionar 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 **nTich** até chegar a zero. Em correspondência com este será apagado o LED 2.

6. O POWER DOWN E O SLEEP MODE

O **Power Down Mode** e **Sleep Mode** é um estado particular de funcionamento do PIC utilizado para reduzir o consumo de corrente no momento em que o mesmo não é utilizado e aguardar a ocorrência de um evento externo.

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

O tempo de utilização efetivo da CPU do PIC é então limitado a poucos milisegundos. Sendo assim, para se evitar o consumo inútil, frente a limitada energia da bateria, é possível desligar boa parte do circuito de funcionamento do PIC e reativá-lo somente quando um evento externo ocorrer.

6.1. A instrução SLEEP

A instrução **SLEEP** será utilizada para colocar o PIC em Power Down Mode e reduzir consequentemente a corrente absorvida que passará 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 a instrução abaixo em um ponto qualquer do nosso programa:

SLEEP

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

Para reduzir o consumo de corrente neste estado, não devemos fazer 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 pólo **positivo (Vdd)** ou ao **negativo (vss)** da alimentação todas a linhas em alta impedância não utilizadas, como a linha RA4/T0CKI (pin 3).

6.2. O "despertar" do PIC

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

- Reset do PIC colocando em nível 0 o pino MCLR (pino 4)
- Timeout do timer do Watchdog (se habilitado)
- 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 primeiro e segundo casos o PIC será resetado e a execução começará da posição 0. No terceiro caso, o PIC se comporta como no atendimento de uma interrupção e irá para a primeira interrup handler e então retornará para execução após a instrução SLEEP. Para que o PIC possa retornar de uma interrupção devemos habilitar o flag do registro **INTCON**.

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

6.3. Um exemplo de uso do Power Down Mode

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

```
;*************************************************
; PDM1.ASM
;
;*************************************************
        PROCESSOR       16C84
        RADIX           DEC
```

```
            INCLUDE         "P16C84.INC"

     SWITCH1 EQU     0
     SWITCH2 EQU     1
     LED1    EQU     2

             ORG     0CH

             ;Contador a 16 bit utilizado da subrotina delay

     Count   RES     2

             ;Reset Vector
             ORG     00H

             ;Salta o corpo principal do programa.

             goto    Start


;*****************************************************************
             ;Interrupt vector

             ORG     04H


;*****************************************************************
             ; Interrupt handler

;*****************************************************************

             bcf     INTCON,INTF     ;zera flag INTF

             retfie                  ;Retorna ao programa principal


;*****************************************************************
             ; Programa principal

;*****************************************************************
     Start:

             bsf     STATUS,RP0

             movlw   00011111B
             movwf   TRISA

             bsf     TRISB,SWITCH1   ;entrada pulso
             bsf     TRISB,SWITCH2   ;entrada pulso
             bcf     TRISB,LED1      ;saida led
             bcf     TRISB,3         ; Linha nao utilizada
             bcf     TRISB,4         ; Linha nao utilizada
             bcf     TRISB,5         ; Linha nao utilizada
             bcf     TRISB,6         ; Linha nao utilizada
             bcf     TRISB,7         ;Linea non utilizzata


             ;zera bit INTEDG do registro OPTION para interrupt  pino
```

```
            ;RB0/INT

            bcf     OPTION_REG,INTEDG

            bcf     STATUS,RP0

            ;habilita interrupt
            bsf     INTCON,GIE

            ;habilita interrupt pino RB0/INT
            bsf     INTCON,INTE

            ;apaga LED1
            bcf     PORTB,LED1


;*****************************************************************
            ; Loop principal
            ;
;*****************************************************************

     MainLoop

            btfss   PORTB,SWITCH2   ;Se SWITCH2 e pressionado entra
            sleep                   ;Power Down Mode

            call    Delay           ;Retardo software

            ;Se o led acceso apaga-o e vice-versa

            btfss   PORTB,LED1      ;Led aceso ?
            goto    TurnOnLed1      ;Nao, acende
            goto    TurnOffLed1     ;Sim, apague

            ;Acende LED1

     TurnOnLed1
            bsf     PORTB,LED1
            goto    MainLoop        ;Continua a piscar


            ;Apaga LED1

     TurnOffLed1
            bcf     PORTB,LED1      ;Apaga LED 1
            goto    MainLoop        ;Continua a piscar



;*****************************************************************
            ;Subrotina de retardo software
;*****************************************************************



     Delay
```

```
        clrf    Count
        clrf    Count+1

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

        return

        END
```

O circuito a ser realizado esta representado na figura 22 abaixo.

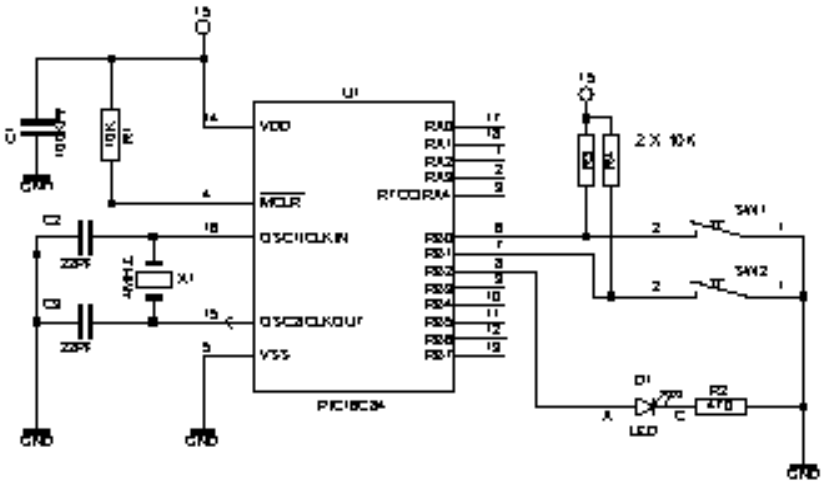

Figura 22 – Uso do POWER DOWN MODE

O **LED D1** conectado na linha RB2 irá piscar para indicar a execução do programa em curso. Pressionando-se a tecla SW2 o programa irá para instrução SLEEP mantendo o PIC 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 PIC, bastará pressionar SW1 para gerar uma interrupção e faze-lo retornar a execução do programa.