UEL - DEEL - LONDRINA - PR

Introdução aos Microcontroladores da Família PIC

Leonimer Flávio de Melo

02/2008

Sumário

Capítulo 1

Introdução aos Microcontroladores

1.1 Introdução

As circunstâncias que se nos deparam hoje no campo dos microcontroladores têm os seus primórdios no desenvolvimento da tecnologia dos circuitos integrados. Este desenvolvimento tornou possível armazenar centenas de milhares de transistores num único chip. Isso constituiu um pré-requisito para a produção de microprocessadores e, os primeiros computadores foram construídos adicionando periféricos externos tais como memória, linhas de entrada e saída, temporizadores e outros. Um crescente aumento do nível de integração, permitiu o aparecimento de circuitos integrados contendo simultaneamente processador e periféricos. Foi assim que o primeiro chip contendo um microcomputador e que mais tarde haveria de ser designado por microcontrolador, apareceu.

1.2 História

É no ano de 1969 que uma equipa de engenheiros japoneses pertencentes à companhia BUSICOM chega aos Estados Unidos com a encomenda de alguns circuitos integrados para calculadoras a serem implementados segundo os seus projetos. A proposta foi entregue à INTEL e Marcian Hoff foi o responsável pela sua concretização. Como ele tinha tido experiência de trabalho com um computador (PC) PDP8, lembrou-se de apresentar uma solução substancialmente diferente em vez da construção sugerida. Esta solução pressupunha que a função do circuito integrado seria determinada por um programa nele armazenado. Isso significava que a configuração deveria ser mais simples, mas também era preciso muito mais memória que no caso do projeto proposto pelos engenheiros japoneses. Depois de algum tempo, embora os engenheiros japoneses tenham tentado encontrar uma solução mais fácil, a idéia de Marcian venceu e o primeiro microprocessador nasceu. Ao transformar esta idéia num produto concreto, Frederico Faggin foi de uma grande utilidade para a INTEL. Ele transferiu-se para a INTEL e, em somente 9 meses, teve sucesso na criação de um produto real a partir da sua primeira concepção. Em 1971, a INTEL adquiriu os direitos sobre a venda deste bloco integral. Primeiro eles compraram a licença à companhia BUSICOM que não tinha a mínima percepção do tesouro que possuía. Neste mesmo ano, apareceu no mercado um microprocessador designado por 4004. Este foi o primeiro microprocessador de 4 bits e tinha a velocidade de 6 000 operações por segundo. Não muito tempo depois, a companhia Americana CTC pediu à INTEL e à Texas Instruments um microprocessador de 8 bits para usar em terminais. Mesmo apesar de a CTC acabar por desistir desta idéia, tanto a Intel como a Texas Instruments continuaram a trabalhar no microprocessador e, em Abril de 1972, os primeiros microprocessadores de 8 bits apareceram no mercado com o nome de 8008. Este podia endereçar 16KB de memória, possuía 45 instruções e tinha a velocidade de 300 000 operações por segundo. Esse microprocessador foi o pioneiro de todos os microprocessadores atuais. A Intel continuou com o desenvolvimento do produto e, em Abril de 1974 pôs cá fora um processador de 8 bits com o nome de 8080 com a capacidade de endereçar 64KB de memória, com 75 instruções e com preços a começarem em $360.

Uma outra companhia Americana, a Motorola, apercebeu-se rapidamente do que estava a acontecer e, assim, pôs no mercado um novo microprocessador de 8 bits, o 6800. O construtor chefe foi Chuck Peddle e além do microprocessador propriamente dito, a Motorola foi a primeira

companhia a fabricar outros periféricos como os 6820 e 6850. Nesta altura, muitas companhias já se tinham apercebido da enorme importância dos microprocessadores e começaram a introduzir os seus próprios desenvolvimentos. Chuck Peddle deixa a Motorola para entrar para a MOS Technology e continua a trabalhar intensivamente no desenvolvimento dos microprocessadores.

Em 1975, na exposição WESCON nos Estados Unidos, ocorreu um acontecimento crítico na história dos microprocessadores. A MOS Technology anunciou que ia pôr no mercado microprocessadores 6501 e 6502 ao preço de $25 cada e que podia satisfazer de imediato todas as encomendas. Isto pareceu tão sensacional que muitos pensaram tratar-se de uma espécie de vigarice, considerando que os competidores vendiam o 8080 e o 6800 a $179 cada. Para responder a este competidor, tanto a Intel como a Motorola baixaram os seus preços por microprocessador para $69,95 logo no primeiro dia da exposição. Rapidamente a Motorola pôs uma ação em tribunal contra a MOS Technology e contra Chuck Peddle por violação dos direitos de autor por copiarem ao copiarem o 6800. A MOS Technology deixou de fabricar o 6501, mas continuou com o 6502. O 6502 é um microprocessador de 8 bits com 56 instruções e uma capacidade de endereçamento de 64KB de memória. Devido ao seu baixo custo, o 6502 torna-se muito popular e, assim, é instalado em computadores como KIM-1, Apple I, Apple II, Atari, Comodore, Acorn, Oric, Galeb, Orao, Ultra e muitos outros. Cedo aparecem vários fabricantes do 6502 (Rockwell, Sznertek, GTE, NCR, Ricoh e Comodore adquiriram a MOS Technology) que, no auge da sua prosperidade, chegou a vender microprocessadores à razão de 15 milhões por ano !

Contudo, os outros não baixaram os braços. Frederico Faggin deixa a Intel e funda a Zilog Inc. Em 1976, a Zilog anuncia o Z80. Durante a concepção deste microprocessador, Faggin toma uma decisão crítica. Sabendo que tinha sido já desenvolvida uma enorme quantidade de programas para o 8080, Faggin conclui que muitos vão permanecer fieis a este microprocessador por causa das grandes despesas que adviriam das alterações a todos estes programas. Assim, ele decide que o novo microprocessador deve ser compatível com o 8080, ou seja, deve ser capaz de executar todos os programas que já tenham sido escritos para o 8080. Além destas características, outras características adicionais foram introduzidas, de tal modo que o Z80 se tornou um microprocessador muito potente no seu tempo. Ele podia endereçar directamente 64KB de memória, tinha 176 instruções, um grande número de registros, uma opção para *refreshing* de memória RAM dinâmica, uma única alimentação, maior velocidade de funcionamento, etc. O Z80 tornou-se um grande sucesso e toda a gente se transferiu do 8080 para o Z80. Pode dizer-se que o Z80 se constituiu sem sombra de dúvida como o microprocessador de 8 bits com maior sucesso no seu tempo. Além da Zilog, outros novos fabricantes como Mostek, NEC, SHARP e SGS apareceram. O Z80 foi o coração de muitos computadores como o Spectrum, Partner, TRS703, Z-3 e Galaxy, que foram aqui usados.

Em 1976, a Intel apareceu com uma versão melhorada do microprocessador de 8 bits e designada por 8085. Contudo, o Z80 era tão superior a este que, bem depressa, a Intel perdeu a batalha. Ainda que mais alguns microprocessadores tenham aparecido no mercado (6809, 2650, SC/MP etc.), já tudo estava então decidido. Já não havia mais grandes melhorias a introduzir pelos fabricantes que fundamentassem a troca por um novo microprocessador, assim, o 6502 e o Z80, acompanhados pelo 6800, mantiveram-se como os mais representativos microprocessadores de 8 bits desse tempo.

1.3 Microcontroladores versus Microprocessadores

Um microcontrolador difere de um microprocessador em vários aspectos. Primeiro e o mais importante, é a sua funcionalidade. Para que um microprocessador possa ser usado, outros componentes devem-lhe ser adicionados, tais como memória e componentes para receber e enviar dados. Em resumo, isso significa que o microprocessador é o verdadeiro coração do computador. Por outro lado, o microcontrolador foi projetado para ter tudo num só. Nenhuns outros componentes externos são necessários nas aplicações, uma vez que todos os periféricos necessários já estão contidos nele. Assim, nós poupamos tempo e espaço na construção dos dispositivos.

1.4 Microcontroladores da família PIC

A seguir são apresentas algumas das características mais importantes a respeito da arquitetura dos microcontroladores PIC da Microchip.

1.4.1 Unidade de Memória

A memória é a parte do microcontrolador cuja função é guardar dados. A maneira mais fácil de explicar é descrevê-la como uma grande prateleira cheia de gavetas. Se supusermos que marcamos as gavetas de modo a elas não se confundirem umas com as outras, então o seu conteúdo será facilmente acessível. Basta saber a designação da gaveta e o seu conteúdo será conhecido.

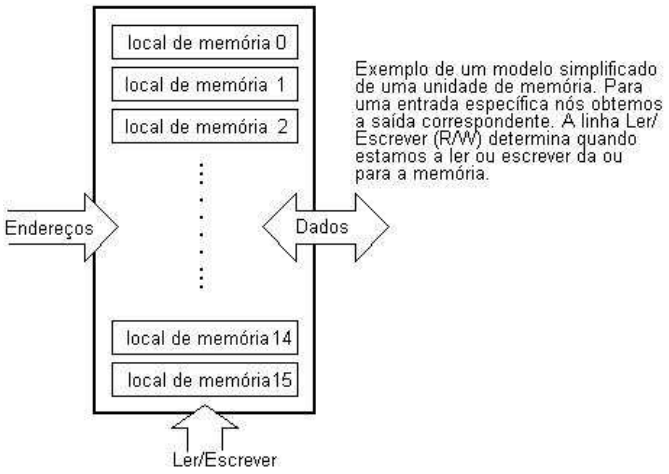


Figura 1.1: Exemplo de unidade de memória.

Os componentes de memória são exatamente a mesma coisa. Para um determinado endereço, nós obtemos o conteúdo desse endereço. Dois novos conceitos foram apresentados: endereçamento e memória. A memória é o conjunto de todos os locais de memória (gavetas) e endereçamento nada mais é que selecionar um deles. Isto significa que precisamos de selecionar o endereço desejado (gaveta) e esperar que o conteúdo desse endereço nos seja apresentado (abrir a gaveta). Além de ler de um local da memória (ler o conteúdo da gaveta), também é possível escrever num endereço da memória (introduzir um conteúdo na gaveta). Isto é feito utilizando uma linha adicional chamada linha de controle. Nós iremos designar esta linha por R/W (read/write) - ler/escrever. A linha de controle é usada do seguinte modo: se r/w=1, é executada uma operação de leitura, caso contrário é executada uma operação de escrita no endereço de memória.

A memória é o primeiro elemento, mas precisamos de mais alguns para que o nosso microcontrolador possa trabalhar.

1.4.2 Unidade Central de Processamento

Vamos agora adicionar mais 3 locais de memória a um bloco específico para que possamos ter as capacidades de multiplicar, dividir, subtrair e mover o seus conteúdos de um local de memória para outro. A parte que vamos acrescentar é chamada "*central processing unit*" (CPU) ou Unidade Central de Processamento. Os locais de memória nela contidos chamam-se registros.

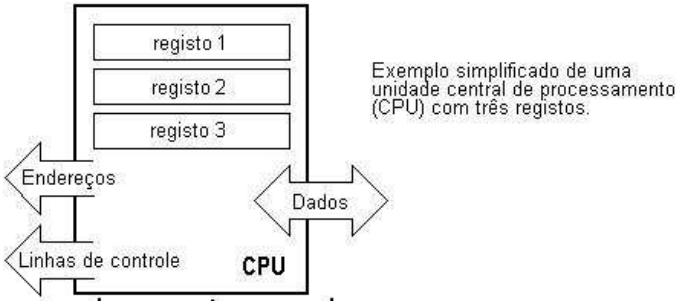


Figura 1.2: Unidade Central de Processamento.

Os registros são, portanto, locais de memória cujo papel é ajudar a executar várias operações matemáticas ou quaisquer outras operações com dados, quaisquer que sejam os locais em que estes se encontrem.

Vamos olhar para a situação actual. Nós temos duas entidades independentes (memória e CPU) que estão interligadas, deste modo, qualquer troca de dados é retardada bem como a funcionalidade do sistema é diminuída. Se, por exemplo, nós desejarmos adicionar os conteúdos de dois locais de memória e tornar a guardar o resultado na memória, nós necessitamos de uma ligação entre a memória e o CPU. Dito mais simplesmente, nós precisamos de obter um "caminho"através do qual os dados possam passar de um bloco para outro.

1.4.3 Barramentos - *Bus*

Este "caminho" designa-se por barramento (*bus*). Fisicamente ele corresponde a um grupo de 8, 16 ou mais fios.

Existem dois tipos de bus: bus de dados e de endereço. O número de linhas do primeiro depende da quantidade de memória que desejamos endereçar e o número de linhas do outro depende da largura da palavra de dados, no nosso caso é igual a oito. O primeiro bus serve para transmitir endereços do CPU para a memória e o segundo para ligar todos os blocos dentro do microcontrolador.

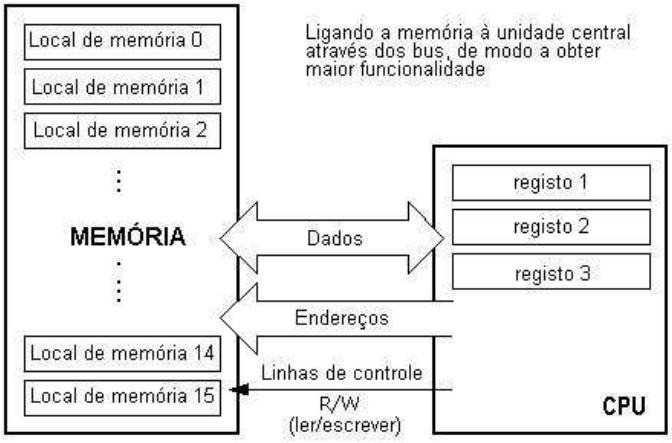


Figura 1.3: Barramentos do microcontrolador.

Neste momento, a funcionalidade já aumentou mas um novo problema apareceu: nós temos uma unidade capaz de trabalhar sozinha, mas que não possui nenhum contato com o mundo exterior, ou seja, conosco! De modo a remover esta deficiência, vamos adicionar um bloco que contém várias localizações de memória e que, de um lado, está ligado ao bus de dados e do outro às linhas de saída do microcontrolador que coincidem com pinos do circuito integrado e que, portanto, nós podemos ver com os nossos próprios olhos.

1.4.4 Unidade de entrada/saída (I/O)

Estas localizações que acabamos de adicionar, chamam-se `portas`. Existem vários tipos de portas: de entrada, de saída e de entrada/saída. Quando trabalhamos com portas primeiro de tudo é necessário escolher a porta com que queremos trabalhar e, em seguida, enviar ou receber dados para ou dessa porta.

Quando se está a trabalhar com ele, a porta funciona como um local de memória. Qualquer coisa de que se está a ler ou em que se está a escrever e que é possível identificar facilmente nos pinos do microcontrolador.

1.4.5 Comunicação série

Anteriormente, acrescentamos à unidade já existente a possibilidade de comunicar com o mundo exterior. Contudo, esta maneira de comunicar tem os seus inconvenientes. Um dos inconve-

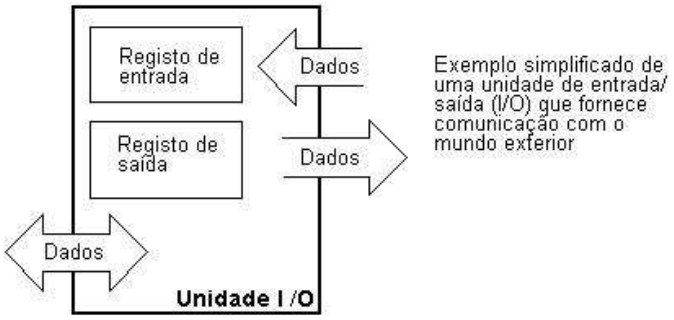


Figura 1.4: Porta de entrada/saída de dados.

nientes básicos é o número de linhas que é necessário usarmos para transferir dados. E se for necessário transferi-los a uma distância de vários quilômetros? O número de linhas vezes o número de quilômetros não atesta a economia do projeto. Isto leva-nos a ter que reduzir o número de linhas de modo a que a funcionalidade se mantenha. Suponha que estamos a trabalhar apenas com três linhas e que uma linha é usada para enviar dados, outra para os receber e a terceira é usada como linha de referência tanto do lado de entrada como do lado da saída. Para que isto trabalhe nós precisamos de definir as regras para a troca de dados. A este conjunto de regras chama-se protocolo. Este protocolo deve ser definido com antecedência de modo que não haja mal entendidos entre as partes que estão a comunicar entre si. Por exemplo, se um homem está a falar em francês e o outro em inglês, é altamente improvável que efetivamente e rapidamente, ambos se entendam. Vamos supor que temos o seguinte protocolo. A unidade lógica "1"é colocada na linha de transmissão até que a transferência se inicie. Assim que isto acontece, a linha passa para nível lógico '0' durante um certo período de tempo (que vamos designar por T), assim, do lado da recepção ficamos a saber que existem dados para receber e, o mecanismo de recepção, vai ativar-se. Regressemos agora ao lado da emissão e comecemos a pôr zeros e uns lógicos na linha de transmissão correspondentes aos bits, primeiro o menos significativo e finalmente o mais significativo. Vamos esperar que cada bit permaneça na linha durante um período de tempo igual a T, e, finalmente, depois do oitavo bit, vamos pôr novamente na linha o nível lógico "1", o que assinala a transmissão de um dado. O protocolo que acabamos de descrever é designado na literatura profissional por NRZ (Não Retorno a Zero).

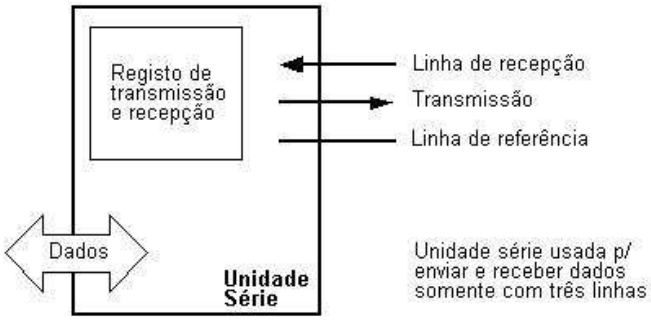


Figura 1.5: Comunicação serial.

1.4.6 Unidade de temporização

Agora que já temos a unidade de comunicação série implementada, nós podemos receber, enviar e processar dados.

Contudo, para sermos capazes de utilizar isto na indústria precisamos ainda de mais alguns blocos. Um deles é o bloco de temporização que nos interessa bastante porque pode dar-nos informações acerca da hora, duração, protocolo, etc. A unidade básica do temporizador é um contador que é na realidade um registro cujo conteúdo aumenta de uma unidade num intervalo de tempo fixo, assim, anotando o seu valor durante os instantes de tempo T1 e T2 e calculando

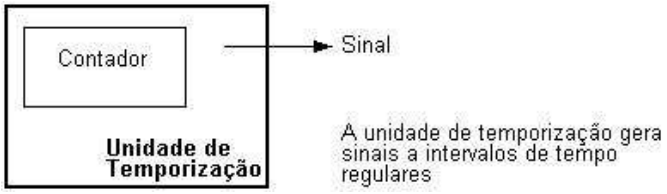


Figura 1.6: Temporizador (*timer*).

a sua diferença, nós ficamos a saber a quantidade de tempo decorrida. Esta é uma parte muito importante do microcontrolador, cujo domínio vai requerer muita da nossa atenção.

1.4.7 Watchdog timer

Uma outra coisa que nos vai interessar é a fluência da execução do programa pelo microcontrolador durante a sua utilização. Suponha que como resultado de qualquer interferência (que ocorre freqüentemente num ambiente industrial), o nosso microcontrolador para de executar o programa ou, ainda pior, desata a trabalhar incoerentemente.

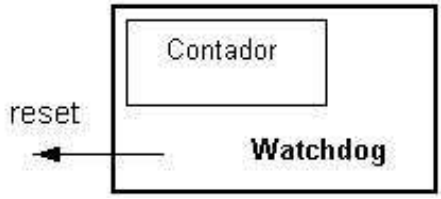


Figura 1.7: Watchdog timer.

Claro que, quando isto acontece com um computador, nós simplesmente carregamos no botão de reset e continuamos a trabalhar. Contudo, no caso do microcontrolador nós não podemos resolver o nosso problema deste modo, porque não temos botão. Para ultrapassar este obstáculo, precisamos de introduzir no nosso modelo um novo bloco chamado watchdog (cão de guarda). Este bloco é de facto outro contador que está continuamente a contar e que o nosso programa põe a zero sempre que é executado corretamente. No caso de o programa *encravar*, o zero não vai ser escrito e o contador, por si só, encarregar-se-á de fazer o reset do microcontrolador quando alcançar o seu valor máximo. Isto vai fazer com que o programa corra de novo e desta vez corretamente. Este é um elemento importante para que qualquer programa se execute confiavelmente, sem precisar da intervenção do ser humano.

1.4.8 Conversor analógico-digital (A/D)

Como os sinais dos periféricos são substancialmente diferentes daqueles que o microcontrolador pode entender (zero e um), eles devem ser convertidos num formato que possa ser compreendido pelo microcontrolador. Esta tarefa é executada por intermédio de um bloco destinado à conversão analógica-digital ou com um conversor A/D. Este bloco vai ser responsável pela conversão de uma informação de valor analógico para um número binário e pelo seu trajeto através do bloco do CPU, de modo a que este o possa processar de imediato.

Figura 1.8: Conversor analógico-digital (A/D).

1.4.9 O microcontrolador

Neste momento, a configuração do microcontrolador está já terminada, tudo o que falta é introduzi-la dentro de um aparelho eletrônico que poderá acessar aos blocos internos através dos pinos deste componente. A Figura 1.9 ilustra o aspecto interno de um microcontrolador.

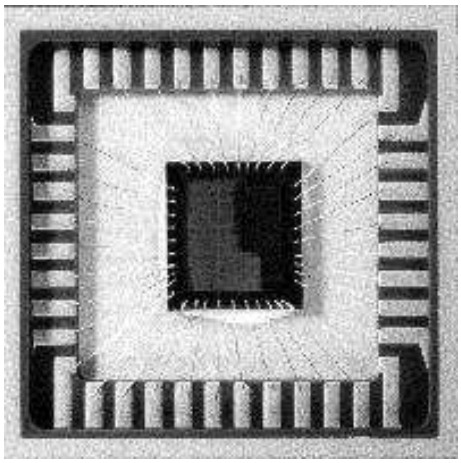


Figura 1.9: Configuração física do interior de um microcontrolador.

As linhas mais finas que partem do centro em direção à periferia do microcontrolador correspondem aos fios que interligam os blocos interiores aos pinos do invólucro do microcontrolador. A Figura 1.10 representa a parte principal de um microcontrolador.

Numa aplicação real, um microcontrolador, por si só, não é suficiente. Além dele, nós necessitamos do programa que vai ser executado e de mais alguns elementos que constituirão um interface lógico para outros elementos.

1.5 Programa

Escrever um programa é uma parte especial do trabalho com microcontroladores e é designado por *programação*. Vamos tentar escrever um pequeno programa numa linguagem que seremos nós a criar e que toda a gente será capaz de compreender.

```
INICIO
REGISTRO1=LOCAL_DE_ MEMORIA_A
REGISTRO2=LOCAL_DE_ MEMORIA_B
PORTO_A=REGISTRO1+REGISTRO2
FIM
```

O programa adiciona os conteúdos de dois locais de memória e coloca a soma destes conteúdos na porta A. A primeira linha do programa manda mover o conteúdo do local de memória "A"para um dos registros da unidade central de processamento. Como necessitamos também de outra parcela, vamos colocar o outro conteúdo noutro registro da unidade central de processamento (CPU). A instrução seguinte pede ao CPU para adicionar os conteúdos dos dois registros e enviar o resultado obtido para a porta A, de modo a que o resultado desta adição seja visível para o mundo exterior. Para um problema mais complexo, naturalmente o programa que o resolve será maior.

A tarefa de programação pode ser executada em várias linguagens tais como o Assembler, C e Basic que são as linguagens normalmente mais usadas. O Assembler pertence ao grupo das linguagens de baixo nível que implicam um trabalho de programação lento, mas que oferece os melhores resultados quando se pretende poupar espaço de memória e aumentar a velocidade de execução do programa. Como se trata da linguagem mais freqüentemente usada na programação de microcontroladores, ela será discutida num capítulo mais adiantado. Os programas na linguagem C são mais fáceis de se escrever e compreender, mas, também, são mais lentos a serem executados que os programas assembler. Basic é a mais fácil de todas para se aprender e as suas instruções

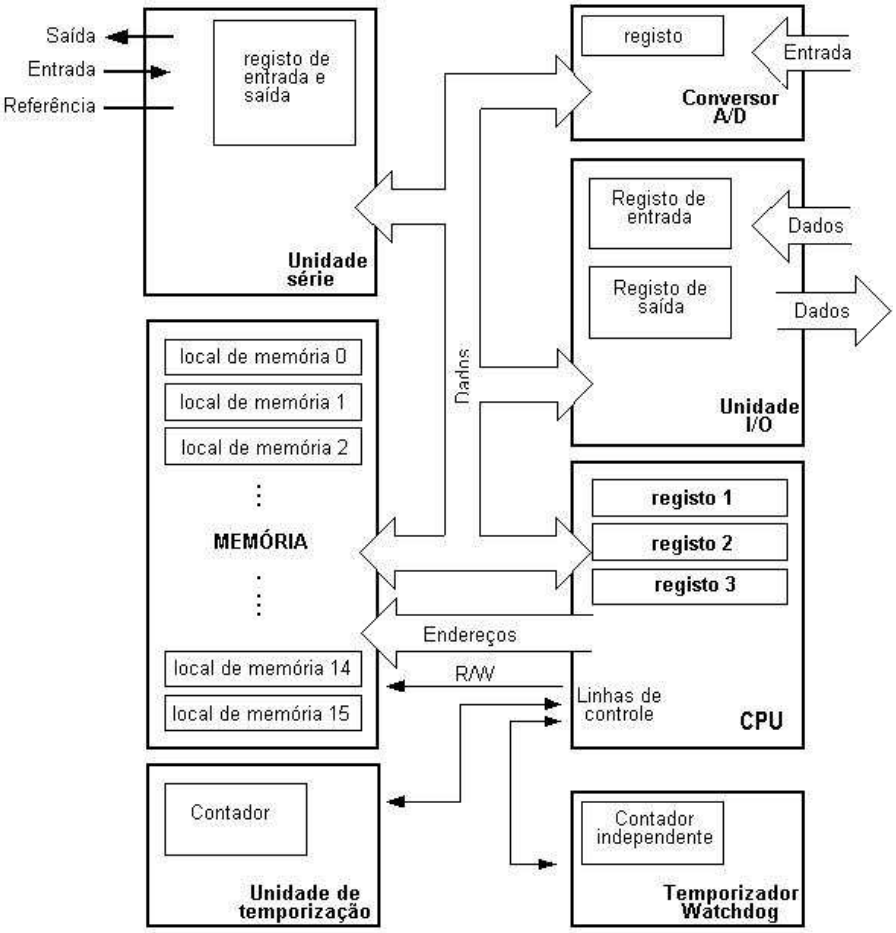


Figura 1.10: Arquitetura do microcontrolador.

são semelhantes à maneira de um ser humano se exprimir, mas tal como a linguagem C, é também de execução mais lenta que o assembler. Em qualquer caso, antes que escolha entre uma destas linguagens, precisa de examinar cuidadosamente os requisitos de velocidade de execução, de espaço de memória a ocupar e o tempo que vai demorar a fazer o programa em assembly.

Depois de o programa estar escrito, nós necessitamos de introduzir o microcontrolador num dispositivo e po-lo para trabalhar. Para que isto aconteça, nós precisamos de adicionar mais alguns componentes externos. Primeiro temos que dar vida ao microcontrolador fornecendo-lhe a tensão (a tensão elétrica é necessária para que qualquer instrumento eletrônico funcione) e o oscilador cujo papel é análogo ao do coração que bate no ser humano. A execução das instruções do programa é regulada pelas pulsações do oscilador. Logo que lhe é aplicada a tensão, o microcontrolador executa uma verificação dele próprio, vai para o princípio do programa e começa a executá-lo. O modo como o dispositivo vai trabalhar depende de muitos parâmetros, os mais importantes dos quais são a competência da pessoa que desenvolve o hardware e do programador que, com o seu programa, deve tirar o máximo do dispositivo.

Capítulo 2

O Microcontrolador PIC16F84

O PIC 16F84 pertence a uma classe de microcontroladores de 8 bits, com uma arquitetura *RISC*. A estrutura genérica é a do mapa que se segue, que nos mostra os seus blocos básicos.

2.1 Componentes básicos do PIC

Memória de programa (FLASH) - para armazenar o programa que se escreveu. Como a memória fabricada com tecnologia FLASH pode ser programa e limpa mais que uma vez. ela torna-se adequada para o desenvolvimento de dispositivos.

EEPROM - memória dos dados que necessitam de ser salvaguardados quando a alimentação é desligada. Normalmente é usada para guardar dados importantes que não se podem perder quando a alimentação, de repente, "vai abaixo". Um exemplo deste tipo de dados é a temperatura fixada para os reguladores de temperatura. Se, durante uma quebra de alimentação, se perdessem dados, nós precisaríamos de proceder a um novo ajustamento quando a alimentação fosse restabelecida. Assim, o nosso dispositivo, perderia eficácia.

RAM - memória de dados usada por um programa, durante a sua execução. Na RAM, são guardados todos os resultados intermédios ou dados temporários durante a execução do programa e que não são cruciais para o dispositivo, depois de ocorrer uma falha na alimentação.

PORTA A e PORTA B são ligações físicas entre o microcontrolador e o mundo exterior. A porta A tem cinco pinos e a porta B oito pinos.

CONTADOR/TEMPORIZADOR é um registro de 8 bits no interior do microcontrolador que trabalha independentemente do programa. No fim de cada conjunto de quatro ciclos de relógio do oscilador, ele incrementa o valor armazenado, até atingir o valor máximo (255), nesta altura recomeça a contagem a partir de zero. Como nós sabemos o tempo exato entre dois incrementos sucessivos do conteúdo do temporizador, podemos utilizar este para medir intervalos de tempo, o que o torna muito útil em vários dispositivos.

UNIDADE DE PROCESSAMENTO CENTRAL faz a conexão com todos os outros blocos do microcontrolador. Ele coordena o trabalho dos outros blocos e executa o programa do utilizador.

2.2 CISC x RISC

Já foi dito que o PIC16F84 tem uma arquitetura RISC. Este termo é encontrado, muitas vezes, na literatura sobre computadores e necessita de ser explicada aqui, mais detalhadamente. A arquitetura de Harvard é um conceito mais recente que a de von-Neumann. Ela adveio da necessidade de pôr o microcontrolador a trabalhar mais rapidamente. Na arquitetura de Harvard, a memória de dados está separada da memória de programa. Assim, é possível uma maior fluência de dados

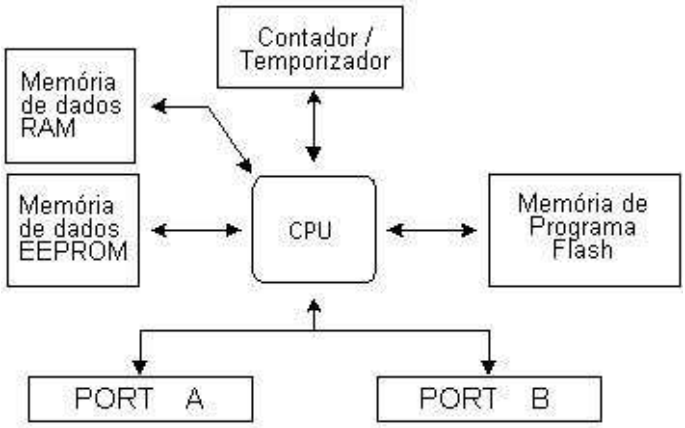


Figura 2.1: Esquema do microcontrolador PIC16F84.

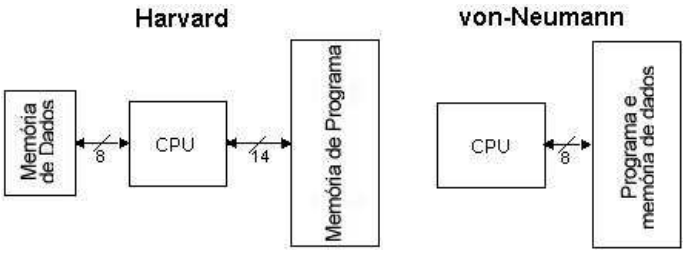


Figura 2.2: Arquiteturas Harvard versus von Neumann.

através da unidade central de processamento e, claro, uma maior velocidade de funcionamento. A separação da memória de dados da memória de programa, faz com que as instruções possam ser representadas por palavras de mais que 8 bits. O PIC16F84, usa 14 bits para cada instrução, o que permite que que todas as instruções ocupem uma só palavra de instrução. É também típico da arquitetura Harvard ter um repertório com menos instruções que a de von-Neumann's, instruções essas, geralmente executadas apenas num único ciclo de relógio.

Os microcontroladores com a arquitetura Harvard, são também designados por `microcontroladores RISC`. RISC provém de Computador com um Conjunto Reduzido de Instruções (Reduced Instruction Set Computer). Os microcontroladores com uma arquitetura von-Neumann são designados por `microcontroladores CISC`. O nome CISC deriva de Computador com um Conjunto Complexo de Instruções (Complex Instruction Set Computer).

Como o PIC16F84 é um microcontrolador RISC, disso resulta que possui um número reduzido de instruções, mais precisamente 35 (por exemplo, os microcontroladores da Intel e da Motorola têm mais de cem instruções). Todas estas instruções são executadas num único ciclo, exceto no caso de instruções de salto e de ramificação. De acordo com o que o seu fabricante refere, o PIC16F84 geralmente atinge resultados de 2 para 1 na compressão de código e 4 para 1 na velocidade, em relação aos outros microcontroladores de 8 bits da sua classe.

2.3 Aplicações

O PIC16F84, é perfeitamente adequado para muitas variedades de aplicações, como a indústria automóvel, sensores remotos, fechaduras elétricas e dispositivos de segurança. É também um dispositivo ideal para cartões inteligentes, bem como para dispositivos alimentados por baterias, por causa do seu baixo consumo.

A memória EEPROM, faz com que se torne mais fácil usar microcontroladores em dispositivos onde o armazenamento permanente de vários parâmetros, seja necessário (códigos para transmissores, velocidade de um motor, freqüências de recepção, etc.). O baixo custo, baixo consumo, facilidade de manuseamento e flexibilidade fazem com que o PIC16F84 se possa utilizar em áreas em que os microcontroladores não eram anteriormente empregues (exemplo: funções de tempo-

rização, substituição de interfaces em sistemas de grande porte, aplicações de coprocessamento, etc.).

A possibilidade deste chip de ser programável no sistema (usando somente dois pinos para a transferência de dados), dão flexibilidade do produto, mesmo depois de a sua montagem e teste estarem completos.

Esta capacidade, pode ser usada para criar linhas de produção e montagem, para armazenar dados de calibragem disponíveis apenas quando se proceder ao teste final ou, ainda, para aperfeiçoar os programas presentes em produtos acabados.

2.4 Relógio / ciclo de instrução

O relógio (clock), é quem dá o sinal de partida para o microcontrolador e é obtido a partir de um componente externo chamado `oscilador`. Se considerasse-mos que um microcontrolador era um relógio de sala, o nosso clock corresponderia ao pêndulo e emitiria um ruído correspondente ao deslocar do pêndulo. Também, a força usada para dar corda ao relógio, podia comparar-se à alimentação elétrica.

O clock do oscilador, é ligado ao microcontrolador através do pino OSC1, aqui, o circuito interno do microcontrolador divide o sinal de clock em quatro fases, Q1, Q2, Q3 e Q4 que não se sobrepõem. Estas quatro pulsações perfazem um ciclo de instrução (também chamado ciclo de máquina) e durante o qual uma instrução é executada. A execução de uma instrução, é antecedida pela extração da instrução que está na linha seguinte. O código da instrução é extraído da memória de programa em Q1 e é escrito no registro de instrução em Q4.

A decodificação e execução dessa mesma instrução, faz-se entre as fases Q1 e Q4 seguintes. Na Figura 2.3 podemos observar a relação entre o ciclo de instrução e o clock do oscilador (OSC1) assim como as fases Q1-Q4. O contador de programa (Program Counter ou PC) guarda o endereço da próxima instrução a ser executada.

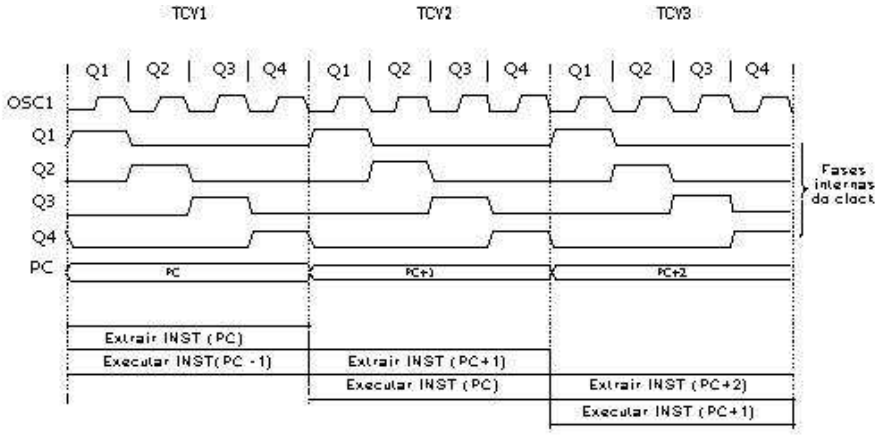


Figura 2.3: Ciclo de instrução e o clock.

2.4.1 Pipelining

Cada ciclo de instrução inclui as fases Q1, Q2, Q3 e Q4. A extração do código de uma instrução da memória de programa, é feita num ciclo de instrução, enquanto que a sua decodificação e execução, são feitos no ciclo de instrução seguinte. Contudo, devido à sobreposição – `pipelining` (o microcontrolador ao mesmo tempo que executa uma instrução extrai simultaneamente da memória o código da instrução seguinte), podemos considerar que, para efeitos práticos, cada instrução demora um ciclo de instrução a ser executada. No entanto, se a instrução provocar uma mudança no conteúdo do contador de programa (PC), ou seja, se o PC não tiver que apontar para o endereço seguinte na memória de programa, mas sim para outro (como no caso de saltos ou de chamadas de subrotinas), então deverá considerar-se que a execução desta instrução demora dois ciclos. Isto acontece, porque a instrução vai ter que ser processada de novo, mas, desta vez, a partir do endereço

correto. O ciclo de chamada começa na fase Q1, escrevendo a instrução no registro de instrução (Instruction Register - IR). A decodificação e execução continua nas fases Q2, Q3 e Q4 do clock.

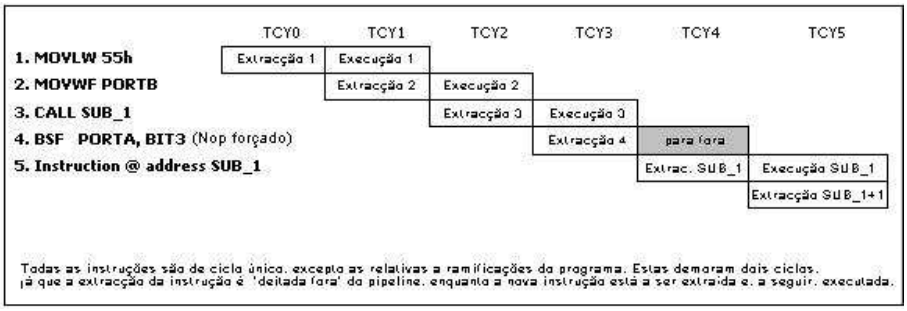


Figura 2.4: Pipeline.

2.4.2 Fluxograma das Instruções no Pipeline

TCY0 é lido da memória o código da instrução MOVLW 55h (não nos interessa a instrução que foi executada, por isso não está representada por retângulo).

TCY1 é executada a instrução MOVLW 55h e é lida da memória a instrução MOVWF PORTB.

TCY2 é executada a instrução MOVWF PORTB e lida a instrução CALL SUB1.

TCY3 é executada a chamada (call) de um subrotina CALL SUB1 e é lida a instrução BSF PORTA,BIT3. Como esta instrução não é a que nos interessa, ou seja, não é a primeira instrução da subrotina SUB1, cuja execução é o que vem a seguir, a leitura de uma instrução tem que ser feita de novo. Este é um bom exemplo de uma instrução a precisar de mais que um ciclo.

TCY4 este ciclo de instrução é totalmente usado para ler a primeira instrução da subrotina no endereço SUB1.

TCY5 é executada a primeira instrução da subrotina SUB1 e lida a instrução seguinte.

2.5 Significado dos pinos

O PIC16F84 tem um total de 18 pinos. É mais freqüentemente encontrado num tipo de encapsulamento DIP18, mas, também pode ser encontrado numa cápsula SMD de menores dimensões que a DIP. DIP é uma abreviatura para Dual In Package (Empacotamento em duas linhas). SMD é uma abreviatura para Surface Mount Devices (Dispositivos de Montagem em Superfície), o que sugere que os pinos não precisam de passar pelos orifícios da placa em que são inseridos, quando se solda este tipo de componente.

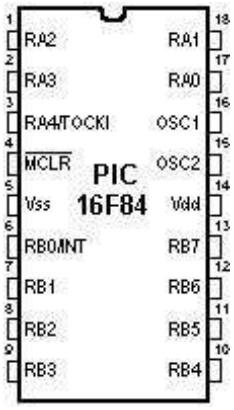


Figura 2.5: Pinagem do PIC16F84.

Os pinos no microcontrolador PIC16F84, têm o seguinte significado:

- Pino nº 1, RA2 Segundo pino da porta A. Não tem nenhuma função adicional.
- Pino nº 2, RA3 Terceiro pino da porta A. Não tem nenhuma função adicional.
- Pino nº 3, RA4 Quarto pino da porta A. O TOCK1 que funciona como entrada do temporizador, também utiliza este pino.
- Pino nº 4, MCLR Entrada de reset e entrada da tensão de programação Vpp do microcontrolador .
- Pino nº 5, Vss massa da alimentação.
- Pino nº 6, RB0, bit 0 da porta B. Tem uma função adicional que é a de entrada de interrupção.
- Pino nº 7, RB1 bit 1da porta B. Não tem nenhuma função adicional.
- Pino nº 8, RB2 bit 2 da porta B. Não tem nenhuma função adicional.
- Pino nº 9, RB3 bit 3 da porta B. Não tem nenhuma função adicional.
- Pino nº 10, RB4 bit 4 da porta B. Não tem nenhuma função adicional.
- Pino nº 11, RB5 bit 5 da porta B. Não tem nenhuma função adicional.
- Pino nº 12, RB6 bit 6 da porta B. No modo de programa é a linha de clock
- Pino nº 13, RB7 bit 7 da porta B. Linha de dados no modo de programa
- Pino nº 14, Vdd pólo positivo da tensão de alimentação.
- Pino nº 15, OSC2 para ser ligado a um oscilador.
- Pino nº 16, OSC1 para ser ligado a um oscilador.
- Pino nº 17, RA0 bit 0 da porta A. Sem função adicional.
- Pino nº 18, RA1 bit 1 da porta A. Sem função adicional.

2.6 Gerador de relógio – oscilador

O circuito do oscilador é usado para fornecer um relógio (clock), ao microcontrolador. O clock é necessário para que o microcontrolador possa executar um programa ou as instruções de um programa.

2.6.1 Tipos de osciladores

O PIC16F84 pode trabalhar com quatro configurações de oscilador. Uma vez que as configurações com um oscilador de cristal e resistência-condensador (RC) são aquelas mais freqüentemente usadas, elas são as únicas que vamos mencionar aqui.

Quando o oscilador é de cristal, a designação da configuração é de XT, se o oscilador for uma resistência em série com um condensador, tem a designação RC. Isto é importante, porque há necessidade de optar entre os diversos tipos de oscilador, quando se escolhe um microcontrolador.

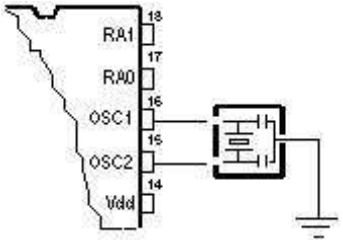


Figura 2.6: Clock de um microcontrolador com um ressonador.

2.6.2 Oscilador XT

O oscilador de cristal está contido num invólucro de metal com dois pinos onde foi escrita a freqüência a que o cristal oscila. Dois condensadores cerâmicos devem ligar cada um dos pinos do cristal à massa. Casos há em que cristal e condensadores estão contidos no mesmo encapsulamento, é também o caso do ressonador cerâmico ao lado representado. Este elemento tem três pinos com o pino central ligado à massa e os outros dois pinos ligados aos pinos OSC1 e OSC2 do microcontrolador. Quando projetamos um dispositivo, a regra é colocar o oscilador tão perto quanto possível do microcontrolador, de modo a evitar qualquer interferência nas linhas que ligam o oscilador ao microcontrolador.

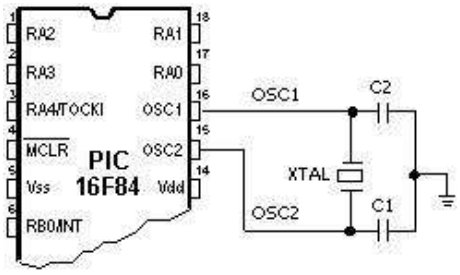


Figura 2.7: Clock de um microcontrolador a partir de um cristal de quartzo.

2.6.3 Oscilador RC

Em aplicações em que a precisão da temporização não é um factor crítico, o oscilador RC torna-se mais econômico. A freqüência de ressonância do oscilador RC depende da tensão de alimentação, da resistência R, capacitância C e da temperatura de funcionamento.

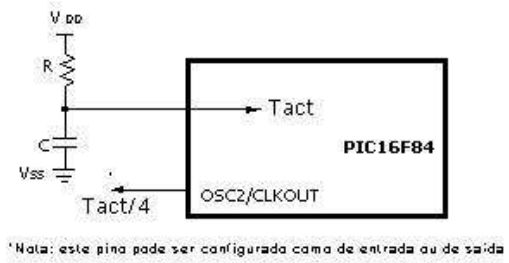


Figura 2.8: Oscilador RC.

A Figura 2.8 mostra como um oscilador RC deve ser ligado a um PIC16F84. Com um valor para a resistência R abaixo de 2,2kΩ, o oscilador pode tornar-se instável ou pode mesmo parar de oscilar. Para um valor muito grande R (1M por exemplo), o oscilador torna-se muito sensível à umidade e ao ruído. É recomendado que o valor da resistência R esteja compreendido entre 3kΩ e 100kΩ. Apesar de o oscilador poder trabalhar sem condensador externo (C = 0pF), é conveniente, ainda assim, usar um condensador acima de 20pF para evitar o ruído e aumentar a estabilidade. Qualquer que seja o oscilador que se está a utilizar, a freqüência de trabalho do microcontrolador é a do oscilador dividida por 4. A freqüência de oscilação dividida por 4 também é fornecida no pino OSC2/CLKOUT e, pode ser usada, para testar ou sincronizar outros circuitos lógicos pertencentes ao sistema.

Ao ligar a alimentação do circuito, o oscilador começa a oscilar. Primeiro com um período de oscilação e uma amplitude instáveis, mas, depois de algum tempo, tudo estabiliza.

Para evitar que esta instabilidade inicial do clock afete o funcionamento do microcontrolador, nós necessitamos de manter o microcontrolador no estado de reset enquanto o clock do oscilador não estabiliza. A Figura 2.10 mostra uma forma típica do sinal fornecido por um oscilador de cristal de quartzo ao microcontrolador quando se liga a alimentação.

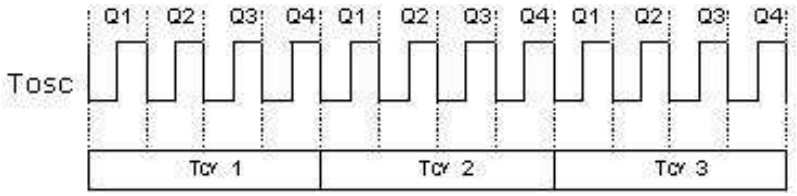


Figura 2.9: Relação entre o sinal de clock e os ciclos de instrução.

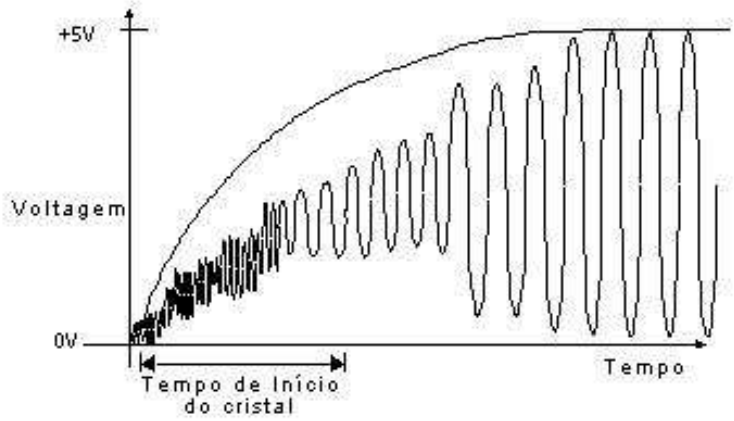


Figura 2.10: Sinal de clock do oscilador do microcontrolador depois de ser ligada a alimentação.

2.7 Reset

O reset é usado para pôr o microcontrolador num estado conhecido. Na prática isto significa que às vezes o microcontrolador pode comportar-se de um modo inadequado em determinadas condições indesejáveis. De modo a que o seu funcionamento normal seja restabelecido, é preciso fazer o reset do microcontrolador, isto significa que todos os seus registros vão conter valores iniciais pré-definidos, correspondentes a uma posição inicial. O reset não é usado somente quando o microcontrolador não se comporta da maneira que nós queremos, mas, também pode ser usado, quando ocorre uma interrupção por parte de outro dispositivo, ou quando se quer que o microcontrolador esteja pronto para executar um programa .

De modo a prevenir a ocorrência de um zero lógico acidental no pino MCLR (a linha por cima de MCLR significa o sinal de reset é ativado por nível lógico baixo), o pino MCLR tem que ser ligado através de uma resistência ao lado positivo da alimentação. Esta resistência deve ter um valor entre 5kΩ e 10kΩ. Uma resistência como esta, cuja função é conservar uma determinada linha a nível lógico alto, é chamada `resistência de pull up`.

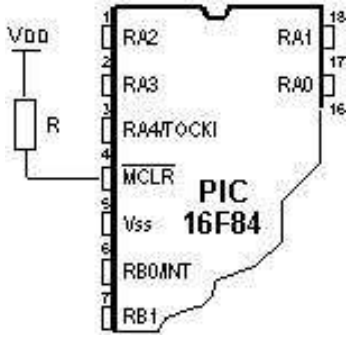


Figura 2.11: Utilização do circuito interno de reset.

O microcontrolador PIC16F84, admite várias formas de reset:

1. Reset quando se liga a alimentação, POR (Power-On Reset)
2. Reset durante o funcionamento normal, quando se põe a nível lógico baixo o pino MCLR do microcontrolador.
3. Reset durante o regime de SLEEP (dormir).
4. Reset quando o temporizador do watchdog (WDT) transborda (passa para 0 depois de atingir o valor máximo).
5. Reset quando o temporizador do watchdog (WDT) transborda estando no regime de SLEEP.

Os reset mais importantes são o primeiro e o segundo. O primeiro, ocorre sempre que é ligada a alimentação do microcontrolador e serve para trazer todos os registros para um estado inicial. O segundo que resulta da aplicação de um valor lógico baixo ao pino MCLR durante o funcionamento normal do microcontrolador e, é usado muitas vezes, durante o desenvolvimento de um programa.

Durante um reset, os locais de memória da RAM (registros) não são alterados. Ou seja, os conteúdos destes registros, são desconhecidos durante o restabelecimento da alimentação, mas mantém-se inalterados durante qualquer outro reset. Ao contrário dos registros normais, os SFR (registros com funções especiais) são reiniciados com um valor inicial pré-definido. Um dos mais importantes efeitos de um reset, é introduzir no contador de programa (PC), o valor zero (0000), o que faz com que o programa comece a ser executado a partir da primeira instrução deste.

2.7.1 Brown-out Reset

Reset quando o valor da alimentação desce abaixo do limite permitido é chamado de `Brown-out Reset`.

O impulso que provoca o reset durante o estabelecimento da alimentação (power-up), é gerado pelo próprio microcontrolador quando detecta um aumento na tensão Vdd (numa faixa entre 1,2V e 1,8V). Esse impulso perdura durante 72ms, o que, em princípio, é tempo suficiente para que o oscilador estabilize. Esse intervalo de tempo de 72ms é definido por um temporizador interno PWRT, com um oscilador RC próprio. Enquanto PWRT estiver actio, o microcontrolador mantém-se no estado de reset. Contudo, quando o dispositivo está a trabalhar, pode surgir um problema não resultante de uma queda da tensão para 0 volts, mas sim de uma queda de tensão para um valor abaixo do limite que garante o correto funcionamento do microcontrolador. Trata-se de um facto muito provável de ocorrer na prática, especialmente em ambientes industriais onde as perturbações e instabilidade da alimentação ocorrem freqüentemente. Para resolver este problema, nós precisamos de estar certos de que o microcontrolador entra no estado de reset de cada vez que a alimentação desce abaixo do limite aprovado.

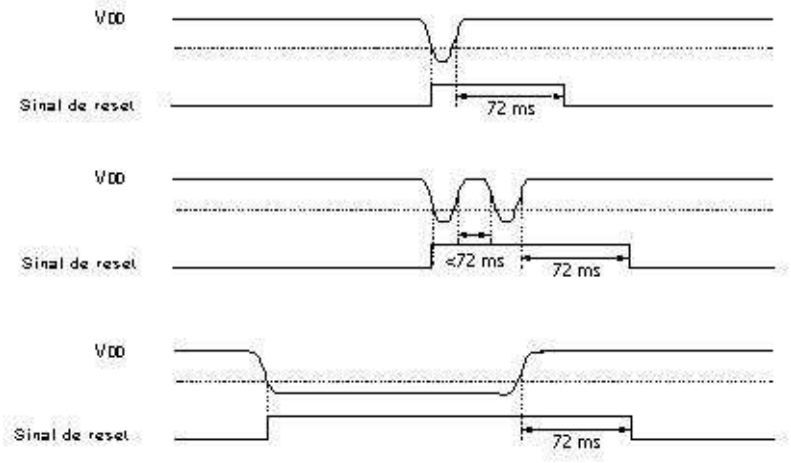


Figura 2.12: Exemplos de quedas na alimentação abaixo do limite.

Se, de acordo com as especificações elétricas, o circuito interno de reset de um microcontrolador não satisfizer as necessidades, então, deverão ser usados componentes electronics especiais, capazes

de gerarem o sinal de reset desejado. Além desta função, estes componentes, podem também cumprir o papel de vigiarem as quedas de tensão para um valor abaixo de um nível especificado. Quando isto ocorre, aparece um zero lógico no pino MCLR, que mantém o microcontrolador no estado de reset, enquanto a tensão não estiver dentro dos limites que garantem um correto funcionamento.

2.8 Unidade Central de Processamento

A unidade central de processamento (CPU) é o cérebro de um microcontrolador. Essa parte é responsável por extrair a instrução, decodificar essa instrução e, finalmente, executá-la.

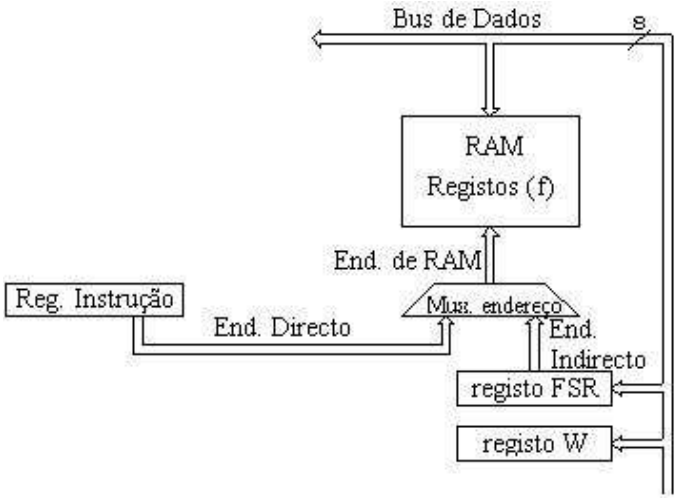


Figura 2.13: Esquema da unidade central de processamento - CPU.

A unidade central de processamento, interliga todas as partes do microcontrolador de modo a que este se comporte como um todo. Uma das sua funções mais importante é, seguramente, decodificar as instruções do programa. Quando o programador escreve um programa, as instruções assumem um claro significado como é o caso por exemplo de MOVLW 0x20. Contudo, para que um microcontrolador possa entendê-las, esta forma escrita de uma instrução tem que ser traduzida numa série de zeros e uns que é o *opcode* (operation code ou código da operação). Esta passagem de uma palavra escrita para a forma binária é executada por tradutores assembler (ou simplesmente assembler). O código da instrução extraído da memória de programa, tem que ser decodificado pela unidade central de processamento (CPU). A cada uma das instruções do repertório do microcontrolador, corresponde um conjunto de ações para a concretizar. Estas ações, podem envolver transferências de dados de um local de memória para outro, de um local de memória para os portos, e diversos cálculos, pelo que, se conclui que, o CPU, tem que estar ligado a todas as partes do microcontrolador. Os bus de de dados e o de endereço permitem-nos fazer isso.

2.8.1 Unidade Lógica Aritmética (ALU)

A unidade lógica aritmética (ALU – Arithmetic Logic Unit), é responsável pela execução de operações de adição, subtração, deslocamento (para a esquerda ou para a direita dentro de um registro) e operações lógicas. O PIC16F84 contém uma unidade lógica aritmética de 8 bits e registros de uso genérico também de 8 bits.

2.9 Registros

Por operando nós designamos o conteúdo sobre o qual uma operação incide. Nas instruções com dois operados, geralmente um operando está contido no registro de trabalho W (working register) e o outro operando ou é uma constante ou então está contido num dos outros registros. Esses

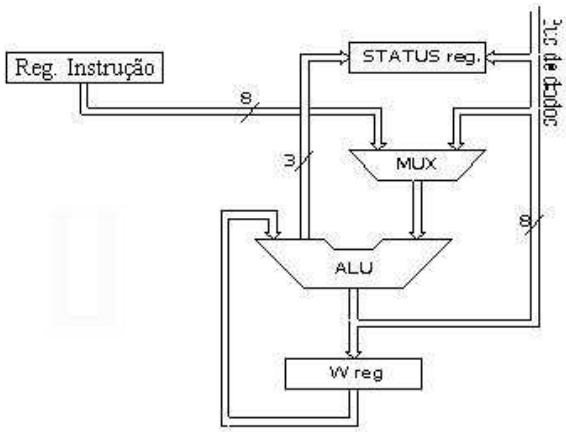


Figura 2.14: Unidade lógica-aritmética.

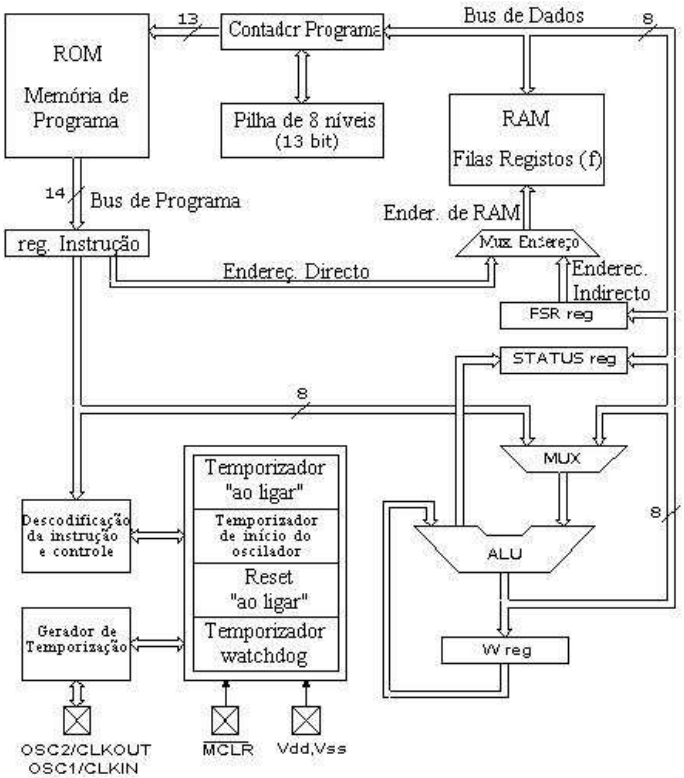


Figura 2.15: Diagrama bloco mais detalhado do microcontrolador PIC16F84.

registros podem ser *Registros de Uso Genérico* (General Purpose Registers – GPR) ou *Registros com funções especiais* (Special Function Registers – SFR). Nas instruções só com um operando, um dos operandos é o conteúdo do registro W ou o conteúdo de um dos outros registros. Quando são executadas operações lógicas ou aritméticas como é o caso da adição, a ALU controla o estado dos bits (que constam do registro de estado – STATUS). Dependendo da instrução a ser executada, a ALU, pode modificar os valores bits do Carry (C), Carry de dígito (DC) e Z (zero) no registro de estado – STATUS.

2.9.1 Registro STATUS

O registro de estado (STATUS), contém o estado da ALU (C, DC, Z), estado de RESET (TO, PD) e os bits para seleção do banco de memória (IRP, RP1, RP0). Considerando que a seleção do banco de memória é controlada através deste registro, ele tem que estar presente em todos os bancos. Os bancos de memória serão discutidos com mais detalhe no capítulo que trata da Organização da Memória. Se o registro STATUS for o registro de destino para instruções que afetem os bits Z, DC ou C, então não é possível escrever nestes três bits.

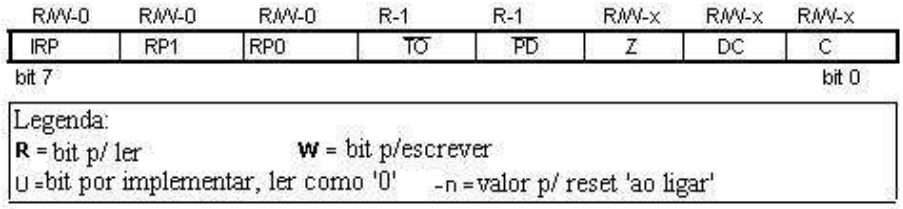


Figura 2.16: Registrador STATUS.

bit 0 C (Carry) Transporte. Este bit é afetado pelas operações de adição, subtração e deslocamento. Toma o valor **1** (set), quando um valor mais pequeno é subtraído de um valor maior e toma o valor **0** (reset) quando um valor maior é subtraído de um menor. 1= Ocorreu um transporte no bit mais significativo 0= Não ocorreu transporte no bit mais significativo O bit C é afetado pelas instruções ADDWF, ADDLW, SUBLW e SUBWF.

bit 1 DC (Digit Carry) Transporte de dígito. Este bit é afetado pelas operações de adição, subtração. Ao contrário do anterior, DC assinala um transporte do bit 3 para o bit 4 do resultado. Este bit toma o valor **1**, quando um valor mais pequeno é subtraído de um valor maior e toma o valor **0** quando um valor maior é subtraído de um menor.
1= Ocorreu um transporte no quarto bit mais significativo
0= Não ocorreu transporte nesse bit
O bit DC é afetado pelas instruções ADDWF, ADDLW, SUBLW e SUBWF.

bit 2 Z (bit Zero) Indicação de resultado igual a zero. Este bit toma o valor **1** quando o resultado da operação lógica ou aritmética executada é igual a 0.
1= resultado igual a zero
0= resultado diferente de zero

bit 3 PD (Bit de baixa de tensão – Power Down). Este bit é posto a **1** quando o microcontrolador é alimentado e começa a trabalhar, depois de um reset normal e depois da execução da instrução CLRWDT. A instrução SLEEP põe este bit a **0** ou seja, quando o microcontrolador entra no regime de baixo consumo / pouco trabalho. Este bit pode também ser posto a **1**, no caso de ocorrer um impulso no pino RB0/INT, uma variação nos quatro bits mais significativos da porta B, ou quando é completada uma operação de escrita na DATA EEPROM ou ainda pelo watchdog.
1 = depois de ter sido ligada a alimentação
0 = depois da execução de uma instrução SLEEP

bit 4 TO Time-out transbordo do Watchdog. Este bit é posto a **1**, depois de a alimentação ser ligada e depois da execução das instruções CLRWDT e SLEEP. O bit é posto a **0** quando

o watchdog consegue chegar ao fim da sua contagem (overflow = transbordar), o que indica que qualquer coisa não esteve bem.
1 = não ocorreu transbordo
0 = ocorreu transbordo

bits 5 e 6 RP1:RP0 (bits de seleção de banco de registros). Estes dois bits são a parte mais significativa do endereço utilizado para endereçamento direto. Como as instruções que endereçam directamente a memória, dispõem somente de sete bits para este efeito, é preciso mais um bit para poder endereçar todos os 256 registros do PIC16F84. No caso do PIC16F84, RP1, não é usado, mas pode ser necessário no caso de outros microcontroladores PIC, de maior capacidade.
01 = banco de registros 1
00 = banco de registros 0

bit 7 IRP (Bit de seleção de banco de registros). Este bit é utilizado no endereçamento indireto da RAM interna, como oitavo bit.
1 = bancos 2 e 3
0 = bancos 0 e 1 (endereços de 00h a FFh)

2.9.2 Registro OPTION

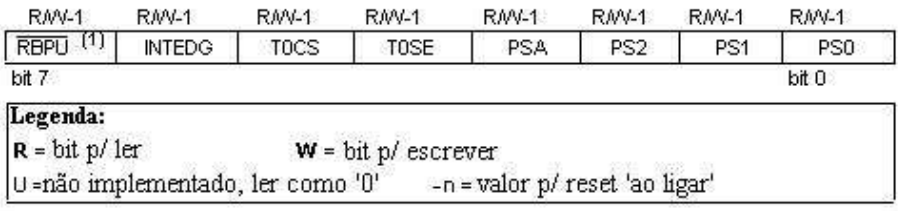


Figura 2.17: Registrador OPTION.

bits 0 a 2 PS0, PS1, PS2 (bits de seleção do divisor Prescaler). Estes três bits definem o factor de divisão do prescaler. Aquilo que é o prescaler e o modo como o valor destes três bits afectam o funcionamento do microcontrolador será estudado na seção referente a TMR0.

Bits	TMR0	WDT
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Figura 2.18: Prescaler.

bit 3 PSA (Bit de Atribuição do Prescaler). Bit que atribui o prescaler ao TMR0 ou ao watchdog.
1 = prescaler atribuído ao watchdog
0 = prescaler atribuído ao temporizador TMR0

bit 4 T0SE (bit de seleção de bordo actio em TMR0). Se for permitido aplicar impulsos em TMR0, a partir do pino RA4/TOCK1, este bit determina se os impulsos activos são os impulsos ascendentes ou os impulsos descendentes.
1 = bordo descendente
0 = bordo ascendente

bit 5 TOCS (bit de seleção de fonte de clock em TMR0). Este pino escolhe a fonte de impulsos que vai ligar ao temporizador. Esta fonte pode ser o clock do microcontrolador (freqüência

de clock a dividir por 4) ou impulsos externos no pino RA4/TOCKI.
1 = impulsos externos
0 = $\frac{1}{4}$ do clock interno

bit 6 INTEDG (bit de seleção de bordo de interrupção). Se esta interrupção estiver habilitada, é possível definir o bordo que vai ativar a interrupção no pino RB0/INT.
1 = bordo ascendente
0 = bordo descendente

bit 7 RBPU (Habilitação dos pull-up nos bits da porta B). Este bit introduz ou retira as resistências internas de pull-up da porta B.
1 = resistências de `pull-up` desligadas
0 = resistências de `pull-up` ligadas

2.10 Portas de I/O

Porta, é um grupo de pinos num microcontrolador que podem ser acessados simultaneamente, e, no qual nós podemos colocar uma combinação de zeros e uns ou ler dele o estado existente. Fisicamente, porta é um registro dentro de um microcontrolador que está ligado por fios aos pinos do microcontrolador. Os portas representam a conexão física da Unidade Central de Processamento (CPU) com o mundo exterior. O microcontrolador usa-os para observar ou comandar outros componentes ou dispositivos. Para aumentar a sua funcionalidade, os mesmos pinos podem ter duas aplicações distintas, como, por exemplo, RA4/TOCKI, que é simultaneamente o bit 4 da porta A e uma entrada externa para o contador/temporizador TMR0. A escolha de uma destas duas funções é feita através dos registros de configuração. Um exemplo disto é o TOCS, quinto bit do registro OPTION. Ao selecionar uma das funções, a outra é automaticamente inibida.

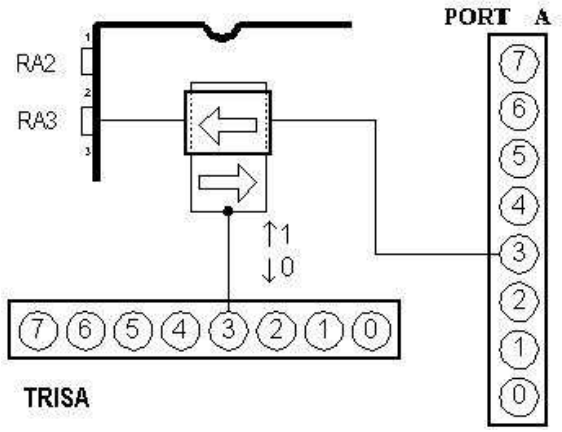


Figura 2.19: Relação entre os registros TRISA e PORTA A.

Todos os pinos das portas podem ser definidos como de entrada ou de saída, de acordo com as necessidades do dispositivo que se está a projetar. Para definir um pino como entrada ou como saída, é preciso, em primeiro lugar, escrever no registro TRIS, a combinação apropriada de zeros e uns. Se no local apropriado de um registro TRIS for escrito o valor lógico **1**, então o correspondente pino da porta é definido como entrada, se suceder o contrário, o pino é definido como saída. Todos as portas, têm um registro TRIS associado. Assim, para a porta A, existe o registro TRISA no endereço 85h e, para a porta B existe o registro TRISB, no endereço 86h.

2.10.1 Porta A

A porta A (`PORTA`) está associado a 5 pinos. O registro de direção de dados correspondente é o TRISA, no endereço 85h. Tal como no caso da porta B, pôr a **1** um bit do registro TRISA, equivale a definir o correspondente pino da porta A, como entrada e pôr a **0** um bit do mesmo registro, equivale a definir o correspondente pino da porta A, como saída.

O quinto pino da porta A tem uma função dupla. Nesse pino está também situada a entrada externa do temporizador TMR0. Cada uma destas opções é escolhida pondo a **1** ou pondo a **0** o bit TOCS (bit de seleção de fonte de clock de TMR0). Conforme o valor deste bit, assim o temporizador TMR0 incrementa o seu valor por causa de um impulso do oscilador interno ou devido a um impulso externo aplicado ao pino RA4/TOCKI.

```
bcf    STATUS,RP0   ;Banco 0
clrf   PORTA        ;Porto A = 0
bsf    STATUS,RP0   ;Banco 1
movlw  0x1F         ;Definir pinos de entrada e  de saída
movwf  TRISA        ;Escrever no registo TRISA
```

Figura 2.20: Exemplo de inicialização da PORTA A.

O exemplo da Figura 2.20 mostra como os pinos 0, 1, 2, 3 e 4 são declarados como entradas e os pinos 5, 6 e 7 como pinos de saída.

2.10.2 Porta B

A porta B tem 8 pinos associados a ele. O respectivo registro de direção de dados chama-se TRISB e tem o endereço 86h. Ao pôr a **1** um bit do registro TRISB, define-se o correspondente pino da porta como entrada e se pusermos a **0** um bit do registro TRISB, o pino correspondente vai ser uma saída. Cada pino da porta B possui uma pequena resistência de `pull-up` (resistência que define a linha como tendo o valor lógico **1**). As resistências de `pull-up` são ativadas pondo a **0** o bit RBPU, que é o bit 7 do registro OPTION. Estas resistências de pull-up são automaticamente desligadas quando os pinos da porta são configurados como saídas. Quando a alimentação do microcontrolador é ligada, as resistências de pull-up são também desativadas.

Quatro pinos da porta B, RB4 a RB7 podem causar uma interrupção, que ocorre quando qualquer deles varia do valor lógico zero para valor lógico um ou o contrário. Esta forma de interrupção só pode ocorrer se estes pinos forem configurados como entradas (se qualquer um destes 4 pinos for configurado como saída, não será gerada uma interrupção quando há variação de estado). Esta modalidade de interrupção, acompanhada da existência de resistências de pull-up internas, torna possível resolver mais facilmente problemas freqüentes que podemos encontrar na prática, como por exemplo a ligação de um teclado matricial. Se as linhas de um teclado ficarem ligadas a estes pinos, sempre que se prime uma tecla, ir-se-á provocar uma interrupção. Ao processar a interrupção, o microcontrolador terá que identificar a tecla que a produziu. Não é recomendável utilizar a porta B, ao mesmo tempo que esta interrupção está a ser processada.

```
clrf   STATUS       ;Banco 0
clrf   PORTB        ;Porto B = 0
bsf    STATUS,RP0   ;Banco 1
movlw  0x0F         ;Definir pinos de entrada e saída
movwf  TRISB        ;Escrever no registo TRISB
```

Figura 2.21: Exemplo de inicialização da PORTA B.

O exemplo da Figura 2.21 mostra como os pinos 0, 1, 2 e 3 são definidos como entradas e 4, 5, 6 e 7 como saídas.

2.11 Organização da memória

O PIC16F84 tem dois blocos de memória separados, um para dados e o outro para o programa. A memória EEPROM e os registros de uso genérico (GPR) na memória RAM constituem o bloco

para dados e a memória FLASH constitui o bloco de programa.

2.11.1 Memória de programa

A memória de programa é implementada usando tecnologia FLASH, o que torna possível programar o microcontrolador muitas vezes antes de este ser instalado num dispositivo, e, mesmo depois da sua instalação, podemos alterar o programa e parâmetros contidos. O tamanho da memória de programa é de 1024 endereços de palavras de 14 bits, destes, os endereços zero e quatro estão reservados respectivamente para o reset e para o vector de interrupção.

2.11.2 Memória de dados

A memória de dados compreende memória EEPROM e memória RAM. A memória EEPROM consiste em 64 posições para palavras de oito bits e cujos conteúdos não se perdem durante uma falha na alimentação. A memória EEPROM não faz parte diretamente do espaço de memória mas é acessada indiretamente através dos registros EEADR e EEDATA. Como a memória EEPROM serve usualmente para guardar parâmetros importantes (por exemplo, de uma dada temperatura em reguladores de temperatura), existe um procedimento estrito para escrever na EEPROM que tem que ser seguido de modo a evitar uma escrita acidental. A memória RAM para dados, ocupa um espaço no mapa de memória desde o endereço 0x0C até 0x4F, o que corresponde a 68 localizações. Os locais da memória RAM são também chamados registros GPR (*General Purpose Registers* = Registros de uso genérico). Os registros GPR podem ser acessados sem ter em atenção o banco em que nos encontramos de momento.

2.11.3 Registros SFR

Os registros que ocupam as 12 primeiras localizações nos bancos 0 e 1 são registros especiais e têm a ver com a manipulação de certos blocos do microcontrolador. Estes registros são os SFR (*Special Function Registers* ou Registros de Funções Especiais).

2.11.4 Bancos de Memória

Além da divisão em *comprimento* entre registros SFR e GPR, o mapa de memória está também dividido em *largura* (ver mapa anterior) em duas áreas chamadas *bancos*. A seleção de um dos bancos é feita por intermédio dos bits RP0 e RP1 do registro STATUS.

Exemplo: `bcf STATUS, RP0`

A instrução BCF "limpa" o bit RP0 (RP0 = 0) do registro STATUS e, assim, coloca-nos no banco 0.

```
bsf STATUS, RP0
```

A instrução BSF põe a um, o bit RP0 (RP0 = 1) do registro STATUS e, assim, coloca-nos no banco 1.

2.11.5 Macro

Normalmente, os grupos de instruções muito usados são ligados numa única unidade que pode ser facilmente invocada por diversas vezes num programa, uma unidade desse tipo chama-se genericamente `Macro` e, normalmente, essa unidade é designada por um nome especifico facilmente compreensível. Com a sua utilização, a seleção entre os dois bancos torna-se mais clara e o próprio programa fica mais legível.

```
BANK0 macro
Bcf STATUS, RP0     ;Selecionar o banco 0 da memória
Endm
```

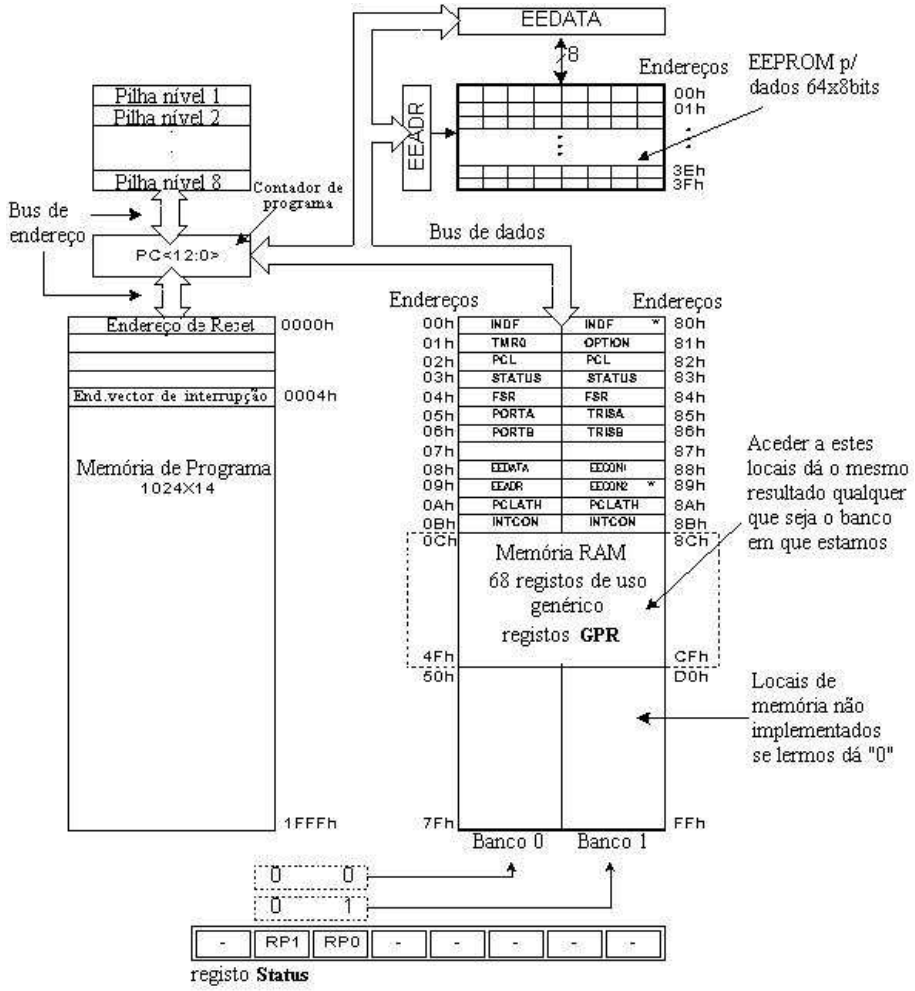


Figura 2.22: Organização da memória no microcontrolador PIC16F84.

```
BANK1 macro
Bsf STATUS, RP0     ;Selecionar o banco 1 da memória
Endm
```

Os locais de memória 0Ch–4Fh são registros de uso genérico (GPR) e são usados como memória RAM. Quando os endereços 8Ch–CFh são acessados, nós acessamos também às mesmas localizações do banco 0. Por outras palavras, quando estamos a trabalhar com os registros de uso genérico, não precisamos de nos preocupar com o banco em que nos encontramos!

2.11.6 Contador de Programa

O contador de programa (PC = *Program Counter*), é um registro de 13 bits que contém o endereço da instrução que vai ser executada. Ao incrementar ou alterar (por exemplo no caso de saltos) o conteúdo do PC, o microcontrolador consegue executar as todas as instruções do programa, uma após outra.

2.11.7 Pilha

O PIC16F84 tem uma pilha (*stack*) de 13 bits e 8 níveis de profundidade, o que corresponde a 8 locais de memória com 13 bits de largura. O seu papel básico é guardar o valor do contador de programa quando ocorre um salto do programa principal para o endereço de um subrotina a ser executado. Depois de ter executado a subrotina, para que o microcontrolador possa continuar com o programa principal a partir do ponto em que o deixou, ele tem que ir buscar à pilha esse endereço e carregá-lo no contador de programa. Quando nos movemos de um programa para um subrotina, o conteúdo do contador de programa é empurrado para o interior da pilha (um exemplo disto é a instrução CALL). Quando são executadas instruções tais como RETURN, RETLW ou RETFIE no fim de um subrotina, o contador de programa é retirado da pilha, de modo a que o programa possa continuar a partir do ponto em que a seqüência foi interrompida. Estas operações de colocar e extrair da pilha o contador de programa, são designadas por `PUSH` (colocar na pilha) e `POP` (tirar da pilha), estes dois nomes provém de instruções com estas designações, existentes nalguns microcontroladores de maior porte.

2.11.8 Programação no Sistema

Para programar a memória de programa, o microcontrolador tem que entrar num modo especial de funcionamento no qual o pino MCLR é posto a 13,5V e a tensão da alimentação Vdd deve permanecer estável entre 4,5V e 5,5V. A memória de programa pode ser programada em série, usando dois pinos *data/clock* que devem ser previamente separados do dispositivo em que o microcontrolador está inserido, de modo a que não possam ocorrer erros durante a programação.

2.11.9 Modos de endereçamento

Os locais da memória RAM podem ser acessados direta ou indiretamente.

Endereçamento Direto

O endereçamento direto é feito através de um endereço de 9 bits. Este endereço obtém-se juntando aos sete bits do endereço direto de uma instrução, mais dois bits (RP1 e RP0) do registro STATUS, como se mostra na figura que se segue. Qualquer acesso aos registros especiais (SFR), pode ser um exemplo de endereçamento direto (Fig. 2.23).

```
Bsf     STATUS, RP0   ;Banco 1
movlw   0xFF          ;w = 0xFF
movwf   TRISA         ;o endereço do registro TRISA é tirado
                      ;do código da instrução movwf TRISA
```

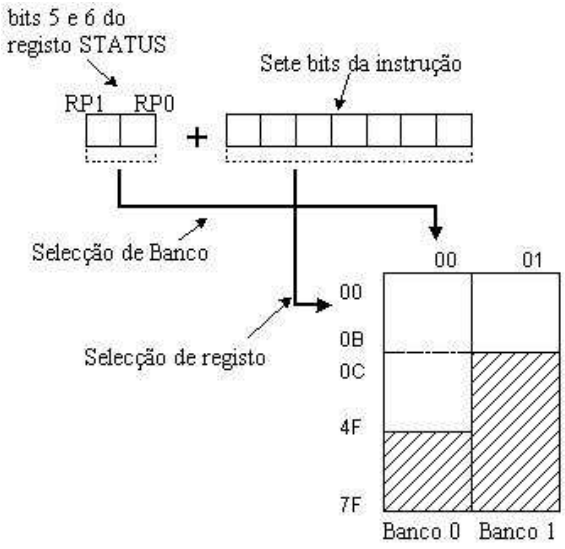


Figura 2.23: Endereçamento Direto.

Endereçamento Indireto

O endereçamento indireto, ao contrário do direto, não tira um endereço do código instrução, mas fá-lo com a ajuda do bit IRP do registro STATUS e do registro FSR. O local endereçado é acessado através do registro INDF e coincide com o endereço contido em FSR. Por outras palavras, qualquer instrução que use INDF como registro, na realidade acessa aos dados apontados pelo registro FSR. Vamos supor, por exemplo, que o registro de uso genérico de endereço 0Fh contém o valor 20. Escrevendo o valor de 0Fh no registro FSR, nós vamos obter um ponteiro para o registro 0Fh e, ao ler o registro INDF, nós iremos obter o valor 20, o que significa que lemos o conteúdo do registro 0Fh, sem o mencionar explicitamente (mas através de FSR e INDF). Pode parecer que este tipo de endereçamento não tem quaisquer vantagens sobre o endereçamento direto, mas existem problemas que só podem ser resolvidos de uma forma simples, através do endereçamento indireto (Fig. 2.24).

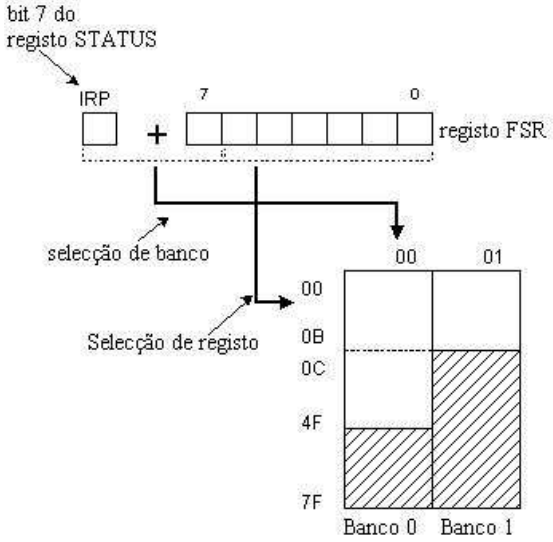


Figura 2.24: Endereçamento indireto.

Um exemplo pode ser enviar um conjunto de dados através de uma comunicação série, usando buffers e indicadores, outro exemplo é limpar os registros da memória RAM (16 endereços neste caso) como ser visto na Figura 2.25.

Quando o conteúdo do registro FSR é igual a zero, ler dados do registro INDF resulta no valor

```
        Movlw  0x0C          ;definição do endereço de início
        Movwf  FSR           ;FSR aponta p/ o endereço 0x0C
LOOP    clrf   INDF          ;INDF = 0
        incf   FSR           ;endereço = endereço inicial + 1
        btfss  FSR,4         ;todos os locais de memória limpos?
        goto   loop          ;não, para 'loop' de novo
CONTINUE
        :                    ;sim, continuar com o programa
```

Figura 2.25: Exemplo de endereçamento indireto.

0 e escrever em INDF resulta na instrução NOP (*no operation* = nenhuma operação).

2.12 Interrupções

As interrupções são um mecanismo que o microcontrolador possui e que torna possível responder a alguns acontecimentos no momento em que eles ocorrem, qualquer que seja a tarefa que o microcontrolador esteja a executar no momento. Esta é uma parte muito importante, porque fornece a ligação entre um microcontrolador e o mundo real que nos rodeia. Geralmente, cada interrupção muda a direção de execução do programa, suspendendo a sua execução, enquanto o microcontrolador corre um subrotina que é a rotina de atendimento de interrupção. Depois de este subrotina ter sido executado, o microcontrolador continua com o programa principal, a partir do local em que o tinha abandonado.

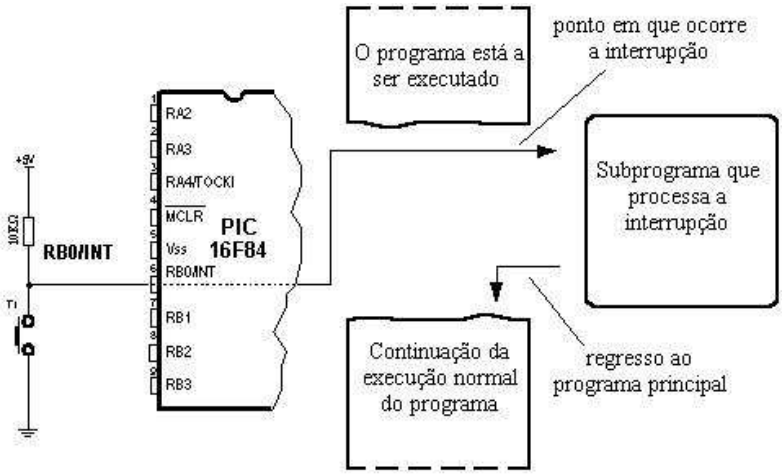


Figura 2.26: Uma das possíveis fontes de interrupção e como afeta o programa principal.

2.12.1 Registro INTCON

O registro que controla as interrupções é chamado INTCON e tem o endereço 0Bh. O papel do INTCON é permitir ou impedir as interrupções e, mesmo no caso de elas não serem permitidas, ele toma nota de pedidos específicos, alterando o nível lógico de alguns dos seus bits.

bit 0 RBIF (flag que indica variação na porta B). Bit que informa que houve mudança nos níveis lógicos nos pinos 4, 5, 6 e 7 da porta B.
1= pelo menos um destes pinos mudou de nível lógico
0= não ocorreu nenhuma variação nestes pinos

bit 1 INTF (flag de interrupção externa INT). Ocorrência de uma interrupção externa.
1= ocorreu uma interrupção externa

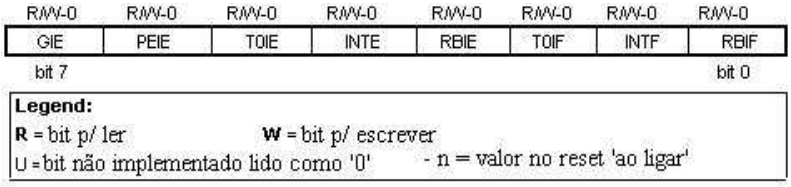


Figura 2.27: Registro INTCON.

0= não ocorreu uma interrupção externa
Se um impulso ascendente ou descendente for detectado no pino RB0/INT, o bit INTF é posto a 1 (o tipo de sensibilidade, ascendente ou descendente é definida através do bit INTEDG do registro OPTION). A subrotina de atendimento desta interrupção, deve repor este bit a 0, afim de que a próxima interrupção possa ser detectado.

bit 2 TOIF (Flag de interrupção por transbordo de TMR0). O contador TMR0, transbordou.
1= o contador mudou a contagem de FFh para 00h
0= o contador não transbordou
Para que esta interrupção seja detectado, o programa deve pôr este bit a 0.

bit 3 RBIE (bit de habilitação de interrupção por variação na porta B). Permite que a interrupção por variação dos níveis lógicos nos pinos 4, 5, 6 e 7 da porta B, ocorra.
1= habilita a interrupção por variação dos níveis lógicos
0= inibe a interrupção por variação dos níveis lógicos
A interrupção só pode ocorrer se RBIE e RBIF estiverem simultaneamente a 1 lógico.

bit 4 INTE (bit de habilitação da interrupção externa INT). bit que permite uma interrupção externa no bit RB0/INT.
1= interrupção externa habilitada
0= interrupção externa impedida
A interrupção só pode ocorrer se INTE e INTF estiverem simultaneamente a 1 lógico.

bit 5 TOIE (bit de habilitação de interrupção por transbordo de TMR0). bit que autoriza a interrupção por transbordo do contador TMR0.
1= interrupção autorizada
0= interrupção impedida
A interrupção só pode ocorrer se TOIE e TOIF estiverem simultaneamente a 1 lógico.

bit 6 EEIE (bit de habilitação de interrupção por escrita completa, na EEPROM). bit que habilita uma interrupção quando uma operação de escrita na EEPROM termina.
1= interrupção habilitada
0= interrupção inibida
Se EEIE e EEIF (que pertence ao registro EECON1) estiverem simultaneamente a 1, a interrupção pode ocorrer.

bit 7 GIE (bit de habilitação global de interrupção). bit que permite ou impede todas as interrupções.
1= todas as interrupções são permitidas
0= todas as interrupções impedidas

2.12.2 Fontes de interrupção

O PIC16F84 possui quatro fontes de interrupção:

1. Fim de escrita na EEPROM;
2. Interrupção em TMR0 causada por transbordo do temporizador;
3. Interrupção por alteração nos pinos RB4, RB5, RB6 e RB7 da porta B;
4. Interrupção externa no pino RB0/INT do microcontrolador.

De um modo geral, cada fonte de interrupção tem dois bits associados. Um habilita a interrupção e o outro assinala quando a interrupção ocorre. Existe um bit comum a todas as interrupções chamado GIE que pode ser usado para impedir ou habilitar todas as interrupções, simultaneamente. Este bit é muito útil quando se está a escrever um programa porque permite que todas as interrupções sejam impedidas durante um período de tempo, de tal maneira que a execução de uma parte crítica do programa não possa ser interrompida. Quando a instrução que faz GIE= 0 é executada (GIE= 0 impede todas as interrupções), todas os pedidos de interrupção pendentes, serão ignorados.

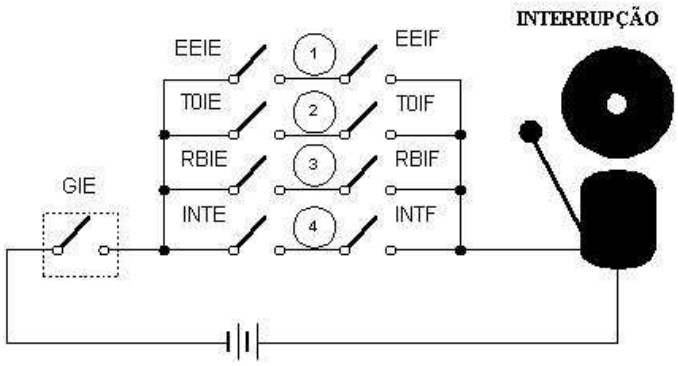


Figura 2.28: Esquema das interrupções no microcontrolador PIC16F84.

As interrupções que estão pendentes e que são ignoradas, são processadas quando o bit GIE é posto a 1 (GIE= 1, todas as interrupções permitidas). Quando a interrupção é atendida, o bit GIE é posto a 0, de tal modo que, quaisquer interrupções adicionais sejam inibidas, o endereço de retorno é guardado na pilha e, no contador de programa, é escrito 0004h – somente depois disto, é que a resposta a uma interrupção começa!

Depois de a interrupção ser processada, o bit que por ter sido posto a 1 permitiu a interrupção, deve agora ser reposto a 0, senão, a rotina de interrupção irá ser automaticamente processada novamente, mal se efetue o regresso ao programa principal.

2.12.3 Guardando os conteúdos dos registros importantes

A única coisa que é guardada na pilha durante uma interrupção é o valor de retorno do contador de programa (por valor de retorno do contador de programa entende-se o endereço da instrução que estava para ser executada, mas que não foi, por causa de ter ocorrido a interrupção). Guardar apenas o valor do contador de programa não é, muitas vezes, suficiente. Alguns registros que já foram usados no programa principal, podem também vir a ser usados na rotina de interrupção. Se nós não salvaguardamos os seus valores, quando acontece o regresso da subrotina para o programa principal os conteúdos dos registros podem ser inteiramente diferentes, o que causaria um erro no programa. Um exemplo para este caso é o conteúdo do registro de trabalho W (*work register*). Se supormos que o programa principal estava a usar o registro de trabalho W nalgumas das suas operações e se ele contiver algum valor que seja importante para a instrução seguinte, então a interrupção que ocorre antes desta instrução vai alterar o valor do registro de trabalho W, indo influenciar diretamente o programa principal.

O procedimento para a gravação de registros importantes antes de ir para a subrotina de interrupção, designa-se por `PUSH`, enquanto que o procedimento que recupera esses valores, é chamado `POP`. `PUSH` e `POP` são instruções provenientes de outros microcontroladores (da Intel), agora esses nomes são aceites para designar estes dois processos de salvaguarda e recuperação de dados. Como o PIC16F84 não possui instruções comparáveis, elas têm que ser programadas.

Devido à sua simplicidade e uso freqüente, estas partes do programa podem ser implementadas com macros. O conceito de Macro é explicado em Programação em linguagem Assembly. No exemplo que se segue, os conteúdos de W e do registro STATUS são guardados nas variáveis W_TEMP e STATUS_TEMP antes de correr a rotina de interrupção. No início da rotina PUSH, nós precisamos de verificar qual o banco que está a ser selecionado porque W_TEMP e STATUS_TEMP

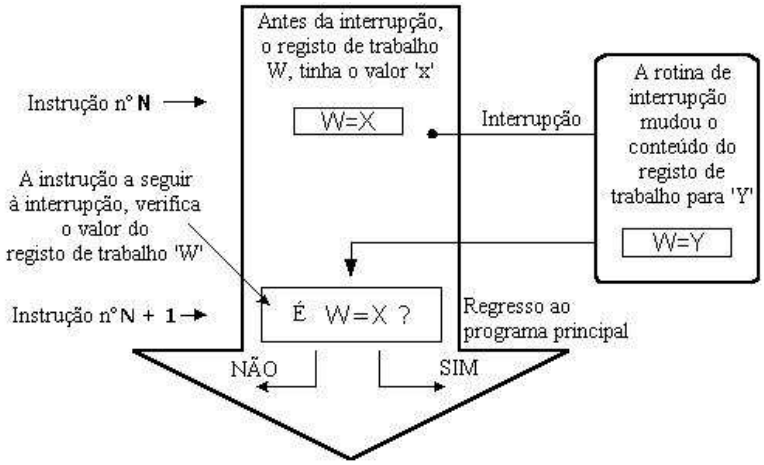


Figura 2.29: Uma das possíveis causas de erros é não salvaguardar dados antes de executar uma subrotina de interrupção.

estão situados no banco 0. Para troca de dados entre estes dois registros, é usada a instrução SWAPF em vez de MOVF, pois a primeira não afeta os bits do registro STATUS.

O exemplo é um programa assembler com os seguintes passos:

1. Verificar em que banco nos encontramos;
2. Guardar o registro W qualquer que seja o banco em que nos encontramos;
3. Guardar o registro STATUS no banco 0;
4. Executar a rotina de serviço de interrupção ISR (*Interrupt Service Routine*);
5. Recuperação do registro STATUS;
6. Restaurar o valor do registro W.

Se existirem mais variáveis ou registros que necessitem de ser salvaguardados, então, precisamos de os guardar depois de guardar o registro STATUS (passo 3) e recuperá-los depois de restaurar o registro STATUS (passo 5)(Fig. 2.30).

A mesma operação pode ser realizada usando macros, desta maneira obtemos um programa mais legível. Os macros que já estão definidos podem ser usados para escrever novos macros. Os macros BANK1 e BANK0 que são explicados no capítulo Organização da memória são usados nos macros `push` e `pop` (Fig. 2.31).

2.12.4 Interrupção externa no pino RB0/INT do microcontrolador

A interrupção externa no pino RB0/ INT é desencadeada por um impulso ascendente (se o bit INTEDG = 1 no registro OPTION 6), ou por um impulso descendente (se INTEDG = 0). Quando o sinal correto surge no pino INT, o bit INTF do registro INTCON é posto a 1. O bit INTF (INTCON 1) tem que ser reposto a 0 na rotina de interrupção, afim de que a interrupção não possa voltar a ocorrer de novo, no regresso ao programa principal. Esta é uma parte importante do programa e que o programador não pode esquecer, caso contrário o programa irá constantemente saltar para a rotina de interrupção. A interrupção pode ser inibida, pondo a 0 o bit de controle INTE (INTCON 4).

2.12.5 Interrupção por transbordar (*overflow*) o contador TMR0

O transbordar do contador TMR0 (passagem de FFh para 00h) vai pôr a 1 o bit TOIF (INTCON 2), Esta é uma interrupção muito importante, uma vez que, muitos problemas da vida real podem ser resolvidos utilizando esta interrupção. Um exemplo é o da medição de tempo. Se soubermos de quanto tempo o contador precisa para completar um ciclo de 00h a FFh, então, o número de

```
Push
        BTFSS STATUS, RP0           ; Banco 0?
        GOTO RP0CLEAR               ; Sim
        BCF STATUS, RP0             ; Não, ir p/ Banco 0
        MOVWF W_TEMP                ; Guardar registo W
        SWAPF STATUS, W             ; W <- STATUS
        MOVWF STATUS_TEMP           ; STATUS_TEMP <- W
        BSF STATUS_TEMP, 1          ; RP0(STATUS_TEMP)=1
        GOTO ISR_Code               ; Push completado
RP0CLEAR
        MOVWF W_TEMP                ; Guardar registo W
        SWAPF STATUS, W             ; W <- STATUS
        MOVWF STATUS_TEMP           ; STATUS_TEMP <- W
;
ISR_Code
        :
        ;Subprograma de Interrupção
        :
;
Pop
        SWAPF STATUS_TEMP, W        ; W <- STATUS_TEMP
        MOVWF STATUS                ; STATUS <-W
        BTFSS STATUS, RP0           ; Banco 1?
        GOTO Return_WREG            ; Não
        BCF STATUS, RP0             ; Sim, ir p/ o banco 0
        SWAPF W_TEMP, F             ; Recuperar o conteúdo de W
        SWAPF W_TEMP, W             ;
        BSF STATUS, RP0             ; Regressar ao banco 1
        RETFIE                      ; POP completo
Return_WREG
        SWAPF W_TEMP, F             ; Recuperar o conteúdo de W
        SWAPF W_TEMP, W             ;
        RETFIE                      ; POP completo
```

Figura 2.30: Guardando os conteúdos dos registros importantes.

```
push    macro
        movwf  W_Temp               ;W_Temp <- W
        swapf  W_Temp,F             ;trocar a ordem dos bits
        BANK1                       ;Macro p/ aceder ao banco 1
        swapf  OPTION_REG,W         ;W <- OPTION_REG
        movwf  Option_Temp          ;Option_Temp <- W
        BANK0                       ;Macro p/ aceder ao banco 0
        swapf  STATUS,W             ;W <- STATUS
        movwf  Stat_Temp            ;Stat_Temp <-W
        endm                        ;Fim do macro push

pop     macro
        swapf  Stat_Temp,W          ;W <- Stat_Temp
        movwf  STATUS               ;STATUS <- W
        BANK1                       ;Macro p/ aceder ao banco 1
        swapf  Option_Temp,W        ;W <- Option_Temp
        movwf  OPTION_REG           ;OPTION_REG <- W
        BANK0                       ;Macro p/ aceder ao banco 0
        swapf  W_Temp,W             ;W <- W_Temp
        endm                        ;Fim do macro pop
```

Figura 2.31: Subrotinas PUSH e POP.

interrupções multiplicado por esse intervalo de tempo, dá-nos o tempo total decorrido. Na rotina de interrupção uma variável guardada na memória RAM vai sendo incrementada, o valor dessa variável multiplicado pelo tempo que o contador precisa para um ciclo completo de contagem, vai dar o tempo gasto. Esta interrupção pode ser habilitada ou inibida, pondo a 1 ou a 0 o bit TOIE (INTCON 5).

2.12.6 Interrupção por variação nos pinos 4, 5, 6 e 7 da porta B

Uma variação em 4 bits de entrada da porta B (bits 4 a 7), põe a 1 o bit RBIF (INTCON 0). A interrupção ocorre, portanto, quando os níveis lógicos em RB7, RB6, RB5 e RB4 da porta B, mudam do valor lógico 1 para o valor lógico 0 ou vice-versa. Para que estes pinos detectem as variações, eles devem ser definidos como entradas. Se qualquer deles for definido como saída, nenhuma interrupção será gerada quando surgir uma variação do nível lógico. Se estes pinos forem definidos como entradas, o seu valor actual é comparado com o valor anterior, que foi guardado quando se fez a leitura anterior da porta B. Esta interrupção pode ser habilitada/inibida pondo a 1 ou a 0, o bit RBIE do registro INTCON.

2.12.7 Interrupção por fim de escrita na EEPROM

Esta interrupção é apenas de natureza prática. Como escrever num endereço da EEPROM leva cerca de 10ms (o que representa muito tempo quando se fala de um microcontrolador), não é recomendável que se deixe o microcontrolador um grande intervalo de tempo sem fazer nada, à espera do fim da operação da escrita. Assim, dispomos de um mecanismo de interrupção que permite ao microcontrolador continuar a executar o programa principal, enquanto, em simultâneo, procede à escrita na EEPROM. Quando esta operação de escrita se completa, uma interrupção informa o microcontrolador deste facto. O bit EEIF, através do qual esta informação é dada, pertence ao registro EECON1. A ocorrência desta interrupção pode ser impedida, pondo a 0 o bit EEIE do registro INTCON.

2.12.8 Inicialização da interrupção

Para que num microcontrolador se possa usar um mecanismo de interrupção, é preciso proceder a algumas tarefas preliminares. Estes procedimentos são designados resumidamente por *inicialização*. Na inicialização, nós estabelecemos a que interrupções deve o microcontrolador responder e as que deve ignorar. Se não pusermos a 1 o bit que permite uma certa interrupção, o programa vai ignorar a correspondente subrotina de interrupção. Por este meio, nós podemos controlar a ocorrência das interrupções, o que é muito útil.

```
clrf  INTCON              ;  todas as interrupções impedidas
movlw B'00010000'         ;  só autorizada a interrupção externa
movwf INTCON
bsf  INTCON, GIE          ;  permitida a ocorrência de interrupçõses
```

Figura 2.32: Inicialização da interrupção.

O exemplo da Figura 2.32 mostra a inicialização da interrupção externa no pino RB0 de um microcontrolador. No BIT em que vemos 1, isso significa que essa interrupção está habilitada. A ocorrência de outras interrupções não é permitida, e todas as interrupções em conjunto estão mascaradas até que o bit GIE seja posto a 1.

O exemplo da Figura 2.33 ilustra uma maneira típica de lidar com as interrupções. O PIC16F84 tem somente um endereço para a rotina de interrupção. Isto significa que, primeiro, é necessário identificar qual a origem da interrupção (se mais que uma fonte de interrupção estiver habilitada), e a seguir deve executar-se apenas a parte da subrotina que se refere à interrupção em causa.

O regresso de uma rotina de interrupção pode efetuar-se com as instruções RETURN, RETLW e RETFIE. Recomenda-se que seja usada a instrução RETFIE porque, essa instrução é a única que automaticamente põe a 1 o bit GIE, permitindo assim que novas interrupções possam ocorrer.

```
org ISR_ADDR            ;ISR_ADDR é o endereço da rotina de interrupção
btfsc INTCON, GIE       ;bit GIE desligado ?
goto ISR_ADR            ;não, voltar ao princípio
PUSH                    ;guardar os conteúdos dos registos importantes
btfsc INTCON, RBIF      ;variação nos pinos 4, 5, 6 e 7 do porto B?
goto ISR_PORTB          ;saltar para a secção correspondente
btfsc INTCON,INTF       ;ocorreu uma interrupção externa em RB0 ?
goto ISR_RB0            ;saltar p/ esse local
btfsc INTCON, TOIF      ;o temporizador TMR0 transbordou ?
goto ISR_TMR0           ;saltar p/ essa secção
BANK1                   ;Banco 1 p/ aceder a EECON1
Btfsc EECON1, EEIF      ;escrita na EEPROM completa?
goto ISR_EEPROM         ;saltar para o endereço correspondente
BANK0                   ;Banco 0

ISR_PORTB
      :                 ;parte do código processado por uma
                        ;interrupção?
      :
      goto END_ISR      ;saltar para a saída da interrupção
ISR_RB0
      :                 ;parte de código processado pela interrupção?
      :
      goto END_ISR      ;saltar para a saída da interrupção
ISR_TMR0
      :                 ;parte de código processado pela interrupção?
      :
      goto END_ISR      ;saltar para a saída da interrupção
ISR_EEPROM
      :                 ;parte de código processado pela interrupção?
      :
      goto END_ISR      ;saltar para a saída da interrupção
END_ISR                 ;
      POP               ;recuperar os conteúdos dos
                        ;registos importantes
      RETFIE            ;regressar e pôr o bit GIE a '1'
```

Figura 2.33: Maneira típica de se lidar com as interrupções.

2.13 Temporizador TMR0

Os temporizadores são normalmente as partes mais complicadas de um microcontrolador, assim, é necessário gastar mais tempo a explicá-los. Servindo-nos deles, é possível relacionar uma dimensão real que é o tempo, com uma variável que representa o estado de um temporizador dentro de um microcontrolador. Fisicamente, o temporizador é um registro cujo valor está continuamente a ser incrementado até 255, chegado a este número, ele começa outra vez de novo: 0, 1, 2, 3, 4, ..., 255, 0, 1, 2, 3,..., etc.

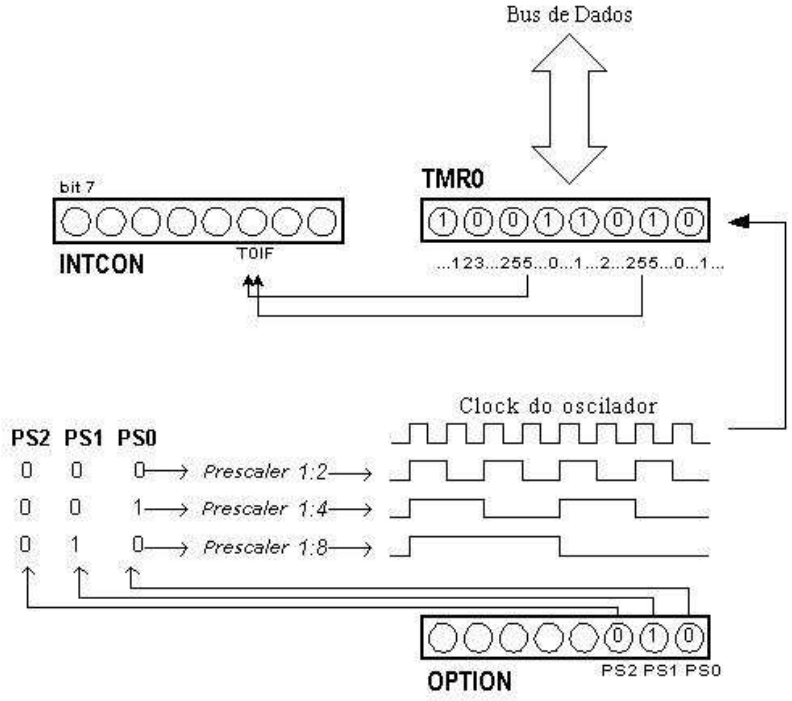


Figura 2.34: Relação entre o temporizador TMR0 e o prescaler.

O incremento do temporizador é feito em simultâneo com tudo o que o microcontrolador faz. Compete ao programador arranjar maneira de tirar partido desta característica. Uma das maneiras é incrementar uma variável sempre que o microcontrolador transborda (*overflow* - passa de 255 para 0). Se soubermos de quanto tempo um temporizador precisa para perfazer uma contagem completa (de 0 a 255), então, se multiplicarmos o valor da variável por esse tempo, nós obteremos o tempo total decorrido.

O PIC16F84, possui um temporizador de 8 bits. O número de bits determina a quantidade de valores diferentes que a contagem pode assumir, antes de voltar novamente para zero. No caso de um temporizador de 8 bits esse valor é 256. Um esquema simplificado da relação entre um temporizador e um prescaler está representado no diagrama anterior. Prescaler é a designação para a parte do microcontrolador que divide a freqüência de oscilação do clock antes que os respectivos impulsos possam incrementar o temporizador. O número pelo qual a freqüência de clock é dividida, está definido nos três primeiros bits do registro OPTION. O maior divisor possível é 256. Neste caso, significa que só após 256 impulsos de clock é que o conteúdo do temporizador é incrementado de uma unidade. Isto permite-nos medir grandes intervalos de tempo.

Quando a contagem ultrapassa 255, o temporizador volta de novo a zero e começa um novo ciclo de contagem até 255. Sempre que ocorre uma transição de 255 para 0, o bit TOIF do registro INTCON é posto a 1. Se as interrupções estiverem habilitadas, é possível tirar partido das interrupções geradas e da rotina de serviço de interrupção. Cabe ao programador voltar a pôr a 0 o bit TOIF na rotina de interrupção, para que uma nova interrupção possa ser detetada. Além do oscilador de clock do microcontrolador, o conteúdo do temporizador pode também ser incrementado através de um clock externo ligado ao pino RA4/TOCKI. A escolha entre uma destas opções é feita no bit TOCS, pertencente ao registro OPTION. Se for selecionado o clock externo,

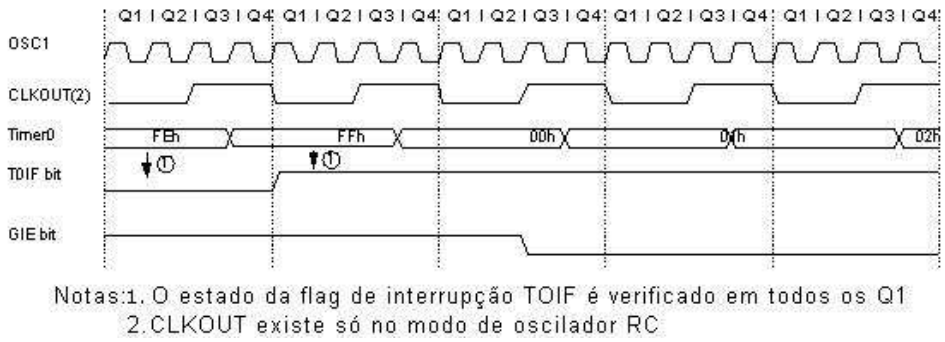


Figura 2.35: Diagrama temporal de uma interrupção causada pelo temporizador TMR0.

é possível definir o bordo ativo do sinal (ascendente ou descendente), que vai incrementar o valor do temporizador.

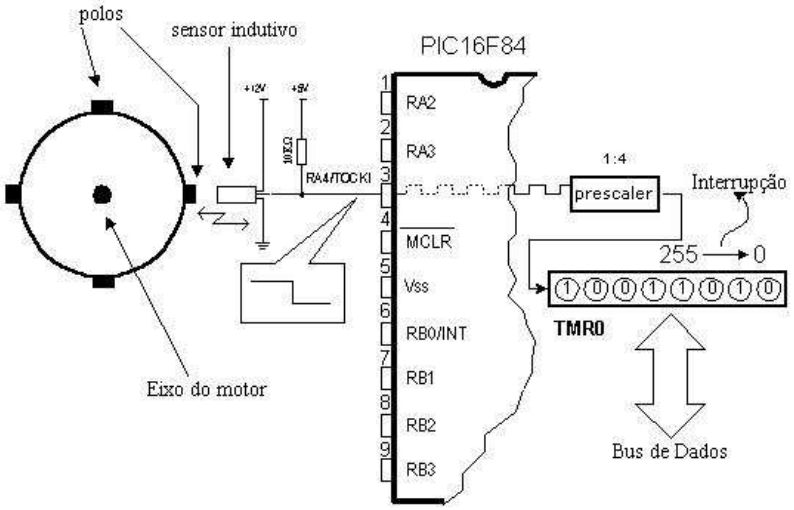


Figura 2.36: Utilização do temporizador TMR0 na determinação do número de rotações completas do eixo de um motor.

Na prática, um exemplo típico que é resolvido através de um clock externo e um temporizador, é a contagem do número de rotações completas do eixo de uma máquina, como por exemplo um enrolador de espiras para transformadores (Fig. 2.36). Vamos considerar que o rotor do motor do enrolador, contém quatro polos ou saliências. Vamos colocar o sensor indutivo à distância de 5mm do topo da saliência. O sensor indutivo irá gerar um impulso descendente sempre que a saliência se encontre alinhada com a cabeça do sensor. Cada sinal vai representar um quarto de uma rotação completa e, a soma de todas as rotações completas, ficará registado no temporizador TMR0. O programa pode ler facilmente estes dados do temporizador através do bus de dados.

O exemplo da Figura 2.37 mostra como iniciar o temporizador para contar os impulsos descendentes provenientes de uma fonte de clock externa com um prescaler 1:4.

O mesmo exemplo pode ser implementado através de uma interrupção como visto da Figura 2.38.

O prescaler tanto pode ser atribuído ao temporizador TMR0, como ao watchdog. O watchdog é um mecanismo que o microcontrolador usa para se defender contra *estouros* do programa. Como qualquer circuito elétrico, também os microcontroladores podem ter uma falha ou algum percalço no seu funcionamento. Infelizmente, o microcontrolador também pode ter problemas com o seu programa. Quando isto acontece, o microcontrolador para de trabalhar e mantém-se nesse estado até que alguém faça o reset. Por causa disto, foi introduzido o mecanismo de *watchdog* (cão de guarda). Depois de um certo período de tempo, o watchdog faz o reset do microcontrolador (o que realmente acontece, é que o microcontrolador executa o reset de si próprio). O *watchdog*

```
      clrf TMR0          ;TMR0=0
      clrf INTCON        ;Interrupções inibidas e TOIF = 0
      bsf STATUS,RP0     ;Banco 1
      movlw B'00110001'  ;prescaler 1:4; interrupção externa no bordo descendente
                         ;fonte de clock externa e resistências de pull-up do
                         ;porto B, activadas.

      movwf OPTION_REG ;OPTION_REG <- W
TO_OVFL
      btfss INTCON, TOIF       ;testando a flag de transbordo
      goto TO_OVFL             ;a interrupção não ocorreu ainda, esperar
;
; (Parte do programa que processa os dados, consoante o número de voltas)
;
goto TO_OVFL                   ;esperar que torne a transbordar
```

Figura 2.37: Programa do temporizador TMR0.

```
      org 0x00           ;endereço de reset
      goto Start         ;início do programa

      org 0x04           ;endereço de interrupção
      goto TO_OVFL       ;início da rotina de interrupção

Start clrf TMR0          ;TMR0=0
      clrf INTCON        ;Intrrupções inibidas e TOIF=0
      bsf STATUS,RP0     ;Banco 1
      movlw B'00110001'  ;prescaler 1:4; interrupção externa no bordo descendente
                         ;fonte de clock externa e resistências de pull-up do
                         ;porto B, activadas.
      movwf OPTION_REG   ;OPTION_REG <- W
      bsf INTCON,TOIE    ;interrupção ao transbordar habilitada
      bsf INTCON,GIE     ; interrupções permitidas
TO_OVFL

; (Parte do programa que processa os dados, consoante o número de voltas)
;

bcf INTCON,TOIF     ;a flag de interrupção é limpa, para que a próxima interrupção
                    ;possa ser detectada.
retfie              ;regresso da rotina de interrupção
```

Figura 2.38: Exemplo de utilização de interrupção.

trabalha na base de um princípio simples: se o seu temporizador transbordar, é feito o reset do microcontrolador e este começa a executar de novo o programa a partir do princípio. Deste modo, o reset poderá ocorrer tanto no caso de funcionamento correto como no caso de funcionamento incorreto. O próximo passo é evitar o reset no caso de funcionamento correto, isso é feito escrevendo zero no registro WDT (instrução CLRWDT) sempre que este está próximo de transbordar. Assim, o programa irá evitar um reset enquanto está a funcionar corretamente. Se ocorrer o *estouro* do programa, este zero não será escrito, haverá transbordo do temporizador WDT e irá ocorrer um reset que vai fazer com que o microcontrolador comece de novo a trabalhar corretamente.

O prescaler pode ser atribuído ao temporizador TMR0, ou ao temporizador do watchdog, isso é feito através do bit PSA no registro OPTION. Fazendo o bit PSA igual a 0, o prescaler é atribuído ao temporizador TMR0. Quando o prescaler é atribuído ao temporizador TMR0, todas as instruções de escrita no registro TMR0 (CLRF TMR0, MOVWF TMR0, BSF TMR0,...) vão limpar o prescaler. Quando o prescaler é atribuído ao temporizador do *watchdog*, somente a instrução CLRWDT irá limpar o prescaler e o temporizador do *watchdog* ao mesmo tempo. A mudança do prescaler está completamente sob o controle do programador e pode ser executada enquanto o programa está sendo executado.

Existe apenas um prescaler com o seu temporizador. Dependendo das necessidades, pode ser atribuído ao temporizador TMR0 ou ao *watchdog*, mas nunca aos dois em simultâneo.

2.14 Memória de dados EEPROM

O PIC16F84 tem 64 bytes de localizações de memória EEPROM, correspondentes aos endereços de 00h a 63h e onde podemos ler e escrever. A característica mais importante desta memória é de não perder o seu conteúdo quando a alimentação é desligada. Na prática, isso significa que o que lá foi escrito permanece no microcontrolador, mesmo quando a alimentação é desligada. Sem alimentação, estes dados permanecem no microcontrolador durante mais de 40 anos (especificações do fabricante do microcontrolador PIC16F84), além disso, esta memória suporta até 10000 operações de escrita.

Na prática, a memória EEPROM é usada para guardar dados importantes ou alguns parâmetros de processamento. Um parâmetro deste tipo, é uma dada temperatura, atribuída quando ajustamos um regulador de temperatura para um processo. Se esse valor se perder, seria necessário reintroduzi-lo sempre que houvesse uma falha na alimentação. Como isto é impraticável (e mesmo perigoso), os fabricantes de microcontroladores começaram a instalar nestes uma pequena quantidade de memória EEPROM.

A memória EEPROM é colocada num espaço de memória especial e pode ser acessada através de registros especiais. Estes registros são:

- EEDATA no endereço 08h, que contém o dado lido ou aquele que se quer escrever.
- EEADR no endereço 09h, que contém o endereço do local da EEPROM que vai ser acessado.
- EECON1 no endereço 88h, que contém os bits de controle.
- EECON2 no endereço 89h. Este registro não existe fisicamente e serve para proteger a EEPROM de uma escrita acidental.

O registro EECON1 ocupa o endereço 88h e é um registro de controle com cinco bits implementados. Os bits 5, 6 e 7 não são usados e, se forem lidos, são sempre iguais a zero. Os bits do registro EECON1, devem ser interpretados do modo que se segue.

2.14.1 Registro EECON1

bit 0 RD (bit de controle de leitura). Ao pôr este bit a 1, tem início a transferência do dado do endereço definido em EEADR para o registro EEDATA. Como o tempo não é essencial, tanto na leitura como na escrita, o dado de EEDATA pode já ser usado na instrução seguinte.
1 = inicia a leitura
0 = não inicia a leitura

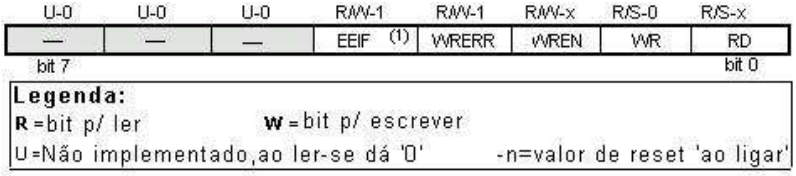


Figura 2.39: Registro EECON1.

bit 1 WR (bit de controle de escrita). Pôr este bit a 1 faz iniciar-se a escrita do dado a partir do registro EEDATA para o endereço especificado no registro EEADR.
1 = inicia a escrita
0 = não inicia a escrita

bit 2 WREN (bit de habilitação de escrita na EEPROM). Permite a escrita na EEPROM. Se este bit não estiver a um, o microcontrolador não permite a escrita na EEPROM.
1 = a escrita é permitida
0 = não se pode escrever

bit 3 WRERR (Erro de escrita na EEPROM). Erro durante a escrita na EEPROM Este bit é posto a 1 só em casos em que a escrita na EEPROM tenha sido interrompida por um sinal de reset ou por um transbordo no temporizador do watchdog (no caso de este estar actio).
1 = ocorreu um erro
0 = não houve erros

bit 4 EEIF (bit de interrupção por operação de escrita na EEPROM completa). Bit usado para informar que a escrita do dadoo na EEPROM, terminou. Quando a escrita tiver terminado, este bit é automaticamente posto a 1. O programador tem que repôr a 0 o bit EEIF no seu programa, para que possa detectar o fim de uma nova operação de escrita.
1 = escrita terminada
0 = a escrita ainda não terminou ou não começou.

2.14.2 Lendo a Memória EEPROM

Pondo a 1 o bit RD inicia-se a transferência do dado do endereço guardado no registro EEADR para o registro EEDATA. Como para ler os dados não é preciso tanto tempo como a escrevê-los, os dados extraídos do registro EEDATA podem já ser usados na instrução seguinte.

Uma porção de um programa que leia um dado da EEPROM, pode ser semelhante ao visto na Figura 2.40.

```
bcf   STATUS, RP0      ;banco 0, porque EEADR está em 09h
movlw 0x00             ;endereço do local a ler
movwf EEADR            ;endereço transferido para EEADR
bsf   STATUS, RP0      ;banco 1, porque EECON1 está em 88h
bsf   EECON1, RD       ;ler da EEPROM
bcf   STATUS, RP0      ;banco 0, porque EEDATA está em 08h
movf  EEDATA, W        ;W <-- EEDATA
```

Figura 2.40: Lendo a Memória EEPROM.

Depois da última instrução do programa, o conteúdo do endereço 0 da EEPROM pode ser encontrado no registro de trabalho `w`.

2.14.3 Escrevendo na Memória EEPROM

Para escrever dados num local da EEPROM, o programador tem primeiro que endereçar o registro EEADR e introduzir a palavra de dados no registro EEDATA. A seguir, deve colocar-se o bit WR a 1, o que faz desencadear o processo. O bit WR deverá ser posto a 0 e o bit EEIF será posto a 1 a seguir à operação de escrita, o que pode ser usado no processamento de interrupções. Os

valores 55h e AAh são as primeira e segunda chaves que tornam impossível que ocorra uma escrita acidental na EEPROM. Estes dois valores são escritos em EECON2 que serve apenas para isto, ou seja, para receber estes dois valores e assim prevenir contra uma escrita acidental na memória EEPROM. As linhas do programa marcadas como 1, 2, 3 e 4 têm que ser executadas por esta ordem em intervalos de tempo certos. Portanto, é muito importante desativar as interrupções que possam interferir com a temporização necessária para executar estas instruções. Depois da operação de escrita, as interrupções podem, finalmente, ser de novo habilitadas.

A Figura 2.41 ilustra um exemplo da porção de programa que escreve a palavra 0xEE no primeiro endereço da memória EEPROM.

```
        bcf   STATUS, RP0       ;banco 0, porque EEADR está em 09h
        movlw 0x00              ;endereço do local de memória
                                ;em que se quer escrever
        movwf EEADR             ;endereço transferido para
                                ;EEADR
        movlw 0xEE              ;escrever o valor 0xEE
        movwf EEDATA            ;dado no registo EEDATA
        bsf STATUS, RP0         ;banco 1
        bcf INTCON, GIE         ;todas as interrupções impedidas
        bsf EECON1, WREN        ;permissão de escrita
        movlw 55h
1)      movwf EECON2            ;1ª chave   55h --> EECON2
2)      movlw AAh
3)      movwf EECON2            ;2ª chave    AAh --> EECON2
4)      bsf EECON1,WR           ;iniciar a escrita
        bsf INTCON, GIE         ;interrupções habilitadas
```

Figura 2.41: Escrevendo na Memória EEPROM.

Recomenda-se que WREN esteja sempre inativo, exceto quando se está a escrever uma palavra de dados na EEPROM, deste modo, a possibilidade de uma escrita acidental é mínima. Todas as operações de escrita na EEPROM *limpam* automaticamente o local de memória, antes de escrever de novo nela!

Capítulo 3

Conjunto de Instruções

Já dissemos que um microcontrolador não é como qualquer outro circuito integrado. Quando saem da cadeia de produção, a maioria dos circuitos integrados, estão prontos para serem introduzidos nos dispositivos, o que não é o caso dos microcontroladores. Para que um microcontrolador cumpra a sua tarefa, nós temos que lhe dizer exatamente o que fazer, ou, por outras palavras, nós temos que escrever o programa que o microcontrolador vai executar. Neste capítulo iremos descrever as instruções que constituem o assembler, ou seja, a linguagem de baixo nível para os microcontroladores PIC.

3.1 Conjunto de Instruções da Família PIC16Fxxx de Microcontroladores

O conjunto completo compreende 35 instruções e mostra-se na tabela que se segue. Uma razão para este pequeno número de instruções resulta principalmente do facto de estarmos a falar de um microcontrolador RISC cujas instruções foram otimizadas tendo em vista a rapidez de funcionamento, simplicidade de arquitetura e compacidade de código. O único inconveniente, é que o programador tem que dominar a técnica *desconfortável* de fazer o programa com apenas 35 instruções!

3.2 Transferência de dados

A transferência de dados num microcontrolador, ocorre entre o registro de trabalho (`W`) e um registro `f` que representa um qualquer local de memória na RAM interna (quer se trate de um registro especial ou de um registro de uso genérico).

As primeiras três instruções (observe a tabela seguinte) referem-se à escrita de uma constante no registro W (MOVLW é uma abreviatura para MOVa Literal para W), à cópia de um dado do registro W na RAM e à cópia de um dado de um registro da RAM no registro W (ou nele próprio, caso em que apenas a flag do zero é afetada). A instrução CLRF escreve a constante 0 no registro f e CLRW escreve a constante 0 no registro W. A instrução SWAPF troca o nibble (conjunto de 4 bits) mais significativo com o nibble menos significativo de um registro, passando o primeiro a ser o menos significativo e o outro o mais significativo do registro.

3.3 Lógicas e aritméticas

De todas as operações aritméticas possíveis, os microcontroladores PIC, tal como a grande maioria dos outros microcontroladores, apenas suportam a subtração e a adição. Os bits ou flags C, DC e Z, são afetados conforme o resultado da adição ou da subtração, com uma única exceção: uma vez que a subtração é executada como uma adição com um número negativo, a flag C (Carry), comporta-se inversamente no que diz respeito à subtração. Por outras palavras, é posta a 1 se a operação é possível e posta a 0 se um número maior tiver que ser subtraído de outro mais pequeno.

A lógica dentro do PIC tem a capacidade de executar as operações AND, OR, EX-OR, complemento (COMF) e rotações (RLF e RRF). Estas últimas instruções, rodam o conteúdo do registro através desse registro e da flag C de uma casa para a esquerda (na direção do bit 7), ou para a direita (na direção do bit 0). O bit que sai do registro é escrito na flag C e o conteúdo anterior desta flag, é escrito no bit situado do lado oposto no registro.

3.4 Operações sobre bits

As instruções BCF e BSF põem a 0 ou a 1 qualquer bit de qualquer sítio da memória. Apesar de parecer uma operação simples, ela é executada do seguinte modo, o CPU primeiro lê o byte completo, altera o valor de um bit e, a seguir, escreve o byte completo no mesmo registrador.

3.5 Direção de execução de um programa

As instruções GOTO, CALL e RETURN são executadas do mesmo modo que em todos os outros microcontroladores, a diferença é que a pilha é independente da RAM interna e é limitada a oito níveis. A instrução RETLW k é idêntica à instrução RETURN, exceto que, ao regressar de um subrotina, é escrita no registro W uma constante definida pelo operando da instrução. Esta instrução, permite-nos implementar facilmente listagens (também chamadas tabelas de lookup). A maior parte das vezes, usamo-las determinando a posição do dado na nossa tabela adicionando-a ao endereço em que a tabela começa e, então, é lido o dado nesse local (que está situado normalmente na memória de programa).

A tabela pode apresentar-se como um subrotina que consiste numa série de instruções RETLW k onde as constantes k, são membros da tabela.

```
Main    movlw 2
        call Lookup
Lookup  addwf PCL, f
        retlw k
        retlw k1
        retlw k2
          :
          :
        retlw kn
```

Figura 3.1: Direção de execução de um programa.

Nós escrevemos a posição de um membro da nossa tabela no registro W e, usando a instrução CALL, nós chamamos o subrotina que contém a tabela. A primeira linha do subrotina `ADDWF PCL, f`, adiciona a posição na tabela e que está escrita em W, ao endereço do início da tabela e que está no registro PCL, assim, nós obtemos o endereço real do dado da tabela na memória de programa. Quando regressamos do subrotina, nós vamos ter no registro W o conteúdo do membro da tabela endereçado. No exemplo anterior, a constante k2 estará no registro W, após o retorno do subrotina.

RETFIE (*RETurn From Interrupt* – Interrupt Enable ou regresso da rotina de interrupção com as interrupções habilitadas) é um regresso da rotina de interrupção e difere de RETURN apenas em que, automaticamente, põe a 1 o bit GIE (habilitação global das interrupções). Quando a interrupção começa, este bit é automaticamente reposto a 0. Também quando a interrupção tem início, somente o valor do contador de programa é posto no cimo da pilha. Não é fornecida uma capacidade automática de armazenamento do registro de estado.

Os saltos condicionais estão sintetizados em duas instruções: BTFSC e BTFSS. Consoante o estado lógico do bit do registro f que está a ser testado, a instrução seguinte no programa é ou não executada.

3.6 Período de execução da instrução

Todas as instruções são executadas num único ciclo, exceto as instruções de ramificação condicional se a condição for verdadeira, ou se o conteúdo do contador de programa for alterado pela instrução.

Nestes casos, a execução requer dois ciclos de instrução e o segundo ciclo é executado como sendo um NOP (Nenhuma Operação). Quatro oscilações de clock perfazem um ciclo de instrução. Se estivermos a usar um oscilador com 4MHz de freqüência, o tempo normal de execução de uma instrução será de 1ms e, no caso de uma ramificação condicional de 2ms.

3.7 Conjunto de instruções

Significado de alguns símbolos:

f qualquer local de memória num microcontrolador
W registro de trabalho
b posição de bit no registro `f`
d registro de destino
label grupo de oito caracteres que marca o início de uma parte do programa (rótulo)
TOS topo da pilha
[ ] opcional
<> grupo de bits num registro

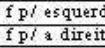
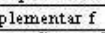
Menemónica		Descrição		Flag	CLK	Notas
		Transferência de dados				
MOVLW	k	Mova literal para W	k → W		1	
MOVWF	f	Mova W para f	W → f		1	
MOVF	f, d	Mova f	f → d	Z	1	1, 2
CLRW	-	Clear W (limpar W)	0 → W	Z	1	
CLRF	f	Clear f (limpar f)	0 → f	Z	1	2
SWAPF	f, d	Swap nibbles in f (trocar)	f(7:4),(3:0)→f(3:0),(7:4)		1	1, 2
		Lógicas e Aritméticas				
ADDLW	k	Adicionar literal a W	W+k → W	C, DC, Z	1	
ADDWF	f, d	Adicionar W a f	W+f → d	C, DC, Z	1	1, 2
SUBLW	k	Subtrair W de literal	k-W →W	C, DC, Z	1	
SUBWF	f, d	Subtrair W de f	f-W → d	C, DC, Z	1	1, 2
ANDLW	k	AND literal com W	W .AND. k → W	Z	1	
ANDWF	f, d	AND W com f	W .AND. f → d	Z	1	1, 2
IORLW	k	Inclusivo OR de literal com W	W .OR. k → W	Z	1	
IORWF	f, d	Inclusivo OR de W com f	W .OR. f → d	Z	1	1, 2
XORWF	f, d	Exclusivo OR de W com f	W . XOR. f → d	Z	1	1, 2
XORLW	k	Exclusivo OR de literal com W	W .XOR. k → W	Z	1	
INCF	f, d	Incrementar f	f+1 → f	Z	1	1, 2
DECF	f, d	Decrementar f	f-1 → f	Z	1	1, 2
RLF	f, d	Rode f p/ esquerda com o carry		C	1	1, 2
RRF	f, d	Rode f p/ a direita com o carry		C	1	1, 2
COMF	f, d	Complementar f	$\overline{f} \rightarrow d$	Z	1	1, 2
		Operações sobre bits				
BCF	f, b	Bit Clear f (bit de f a '0')	0 → f(b)		1	1, 2
BSF	f, b	Bit Set f (bit de f a '1')	1 → f(b)		1	1, 2
		Direccionamento do programa				
BTFSC	f, b	Bit Test f, Salte se Clear ('0')	salte se f(b)=0		1(2)	3
BTFSS	f, b	Bit Test f, Salte se Set ('1')	salte se f(b)=1		1(2)	3
DECFSZ	f, d	Decremente f, salte se der 0	f-1 → d, salte se der 0		1(2)	1, 2, 3
INCFSZ	f, d	Incremente f, salte se der 0	f+1 → d,salte se der 0		1(2)	1, 2, 3
GOTO	k	Go to address(Ir p/ endereço)	k → PC		2	
CALL	k	Chamar subrotina	PC → TOS, k → PC		2	
RETURN	-	Retorno de subrotina	TOS → PC		2	
RETLW	k	Retorno com literal em W	k → W, TOS → PC		2	
RETFIE	-	Retorno de interrupção	TOS → PC, 1 → GIE		2	
		Outras instruções				
NOP	-	Nenhuma operação			1	
CLRWDT	-	Temporizador do Watchdog=0	0 → WDT, 1 → TO, 1 → PD	$\overline{TO}$, $\overline{PD}$	1	
SLEEP	-	Entrar no modo 'sleep'	0 → WDT, 1 → TO, 0 → PD	$\overline{TO}$, $\overline{PD}$	1	

Figura 3.2: Mnemônicos Assembly.

Resumo das notas da Figura 3.2:

1. Se a porta de entrada/saída for o operando origem, é lido o estado dos pinos do microcontrolador.

2. Se esta instrução for executada no registro TMR0 e se d=1, o prescaler atribuído a esse temporizador é automaticamente limpo.

3. Se o PC for modificado ou se resultado do teste for verdadeiro, a instrução é executada em dois ciclos.

Capítulo 4

Programação em Linguagem Assembly

A capacidade de comunicar é da maior importância nesta área. Contudo, isso só é possível se ambas as partes usarem a mesma linguagem, ou seja, se seguirem as mesmas regras para comunicarem. Isto mesmo se aplica à comunicação entre os microcontroladores e o homem. A linguagem que o microcontrolador e o homem usam para comunicar entre si é designada por *linguagem assembly*. O próprio título não tem um significado profundo, trata-se de apenas um nome como por exemplo inglês ou francês. Mais precisamente, linguagem assembly é apenas uma solução transitória. Os programas escritos em linguagem assembly devem ser traduzidos para uma linguagem de zeros e uns de modo a que um microcontrolador a possa receber. Linguagem assembly e assembler são coisas diferentes. A primeira, representa um conjunto de regras usadas para escrever um programa para um microcontrolador e a outra, é um programa que corre num computador pessoal que traduz a linguagem assembly para uma linguagem de zeros e uns. Um programa escrito em zeros e uns diz-se que está escrito em linguagem de máquina.

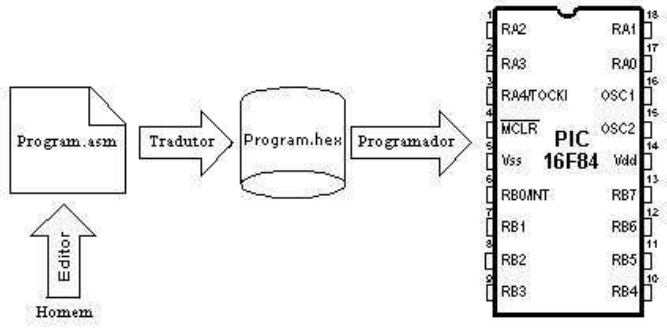


Figura 4.1: O processo de comunicação entre o homem e o microcontrolador.

Fisicamente, "Programa" representa um arquivo num disco de computador (ou na memória se estivermos a ler de um microcontrolador) e é escrito de acordo com as regras do assembly ou qualquer outra linguagem de programação de microcontroladores. O homem pode entender a linguagem assembly já que ela é constituída por símbolos alfabéticos e palavras. Ao escrever um programa, certas regras devem ser seguidas para alcançar o efeito desejado. Um Tradutor interpreta cada instrução escrita em linguagem assembly como uma série de zeros e uns com significado para a lógica interna do microcontrolador. Consideremos, por exemplo, a instrução RETURN que um microcontrolador utiliza para regressar de uma subrotina. Quando o assembler a traduz, nós obtemos uma série de uns e zeros correspondentes a 14 bits que o microcontrolador sabe como interpretar.

Exemplo: RETURN 00 0000 0000 1000

Analogamente ao exemplo anterior, cada instrução assembly é interpretada na série de zeros e uns correspondente. O resultado desta tradução da linguagem assembly, é designado por um arquivo de execução. Muitas vezes encontramos o nome de arquivo `HEX`. Este nome provém de uma representação hexadecimal desse arquivo, bem como o sufixo `hex` no título, por exemplo *correr.hex*. Uma vez produzido, o arquivo de execução é inserido no microcontrolador através de um programador.

Um programa em Linguagem Assembly é escrito por intermédio de um processador de texto (editor) e é capaz de produzir um arquivo ASCII no disco de um computador ou em ambientes próprios como o MPLAB[1].

4.1 Linguagem Assembly

Os elementos básicos da linguagem assembly são:

- Labels (rótulos)
- Instruções
- Operandos
- Diretivas
- Comentários

4.1.1 Label

Um Label (rótulo) é uma designação textual (geralmente de fácil leitura) de uma linha num programa ou de uma seção de um programa para onde um microcontrolador deve saltar ou, ainda, o início de um conjunto de linhas de um programa. Também pode ser usado para executar uma ramificação de um programa (tal como Goto....), o programa pode ainda conter uma condição que deve ser satisfeita, para que uma instrução Goto seja executada. É importante que um rótulo (label) seja iniciado com uma letra do alfabeto ou com um traço baixo "_". O comprimento de um rótulo pode ir até 32 caracteres. É também importante que o rótulo comece na primeira coluna.

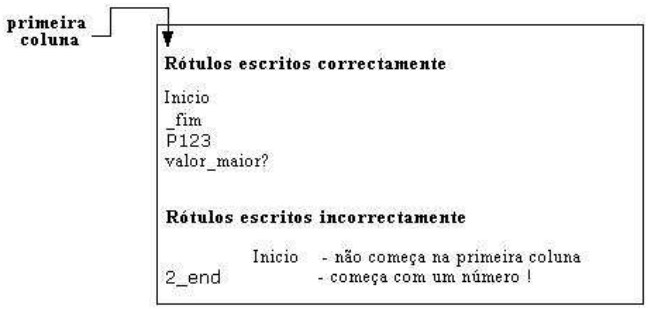


Figura 4.2: Exemplo de labels.

4.1.2 Instruções

As instruções são específicas para cada microcontrolador, assim, se quisermos utilizar a linguagem assembly temos que estudar as instruções desse microcontrolador. O modo como se escreve uma instrução é designado por sintaxe. No exemplo da Figura 4.3 é possível reconhecer erros de escrita, dado que as instruções `movlp` e `gotto` não existem no microcontrolador PIC16F84.

[1]Programa da Microchip utilizado para editar e programar microcontroladores

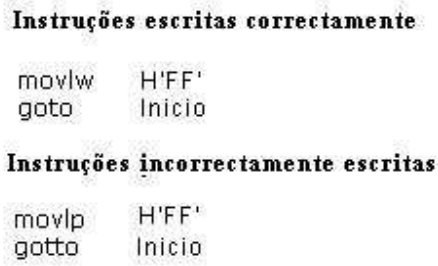


Figura 4.3: Exemplo de instruções.

4.1.3 Operandos

Operandos são os elementos da instrução necessários para que a instrução possa ser executada. Normalmente são registros, variáveis e constantes. As constantes são designadas por literais. A palavra literal significa número.

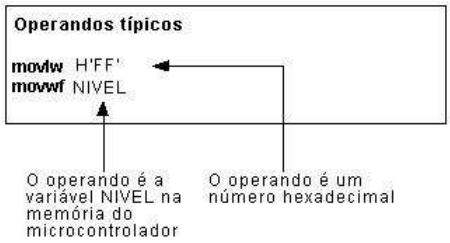


Figura 4.4: Exemplo de Operandos.

4.1.4 Comentários

Comentário é um texto que o programador escreve no programa afim de tornar este mais claro e legível. É colocado logo a seguir a uma instrução e deve começar com um ponto-e-vírgula ";".

4.1.5 Diretivas

Uma diretiva é parecida com uma instrução mas, ao contrário desta, é independente do tipo de microcontrolador e é uma característica inerente à própria linguagem assembly. As diretivas servem-se de variáveis ou registros para satisfazer determinados propósitos. Por exemplo, NIVEL, pode ser uma designação para uma variável localizada no endereço 0Dh da memória RAM. Deste modo, a variável que reside nesse endereço, pode ser acessada pela palavra NIVEL. É muito mais fácil a um programador recordar a palavra NIVEL, que lembrar-se que o endereço 0Dh contém informação sobre o nível.

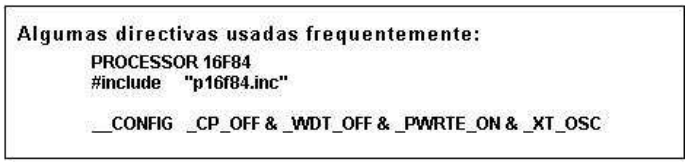


Figura 4.5: Exemplo de diretivas.

4.1.6 Exemplo de um programa em assembly

O exemplo da Figura 4.6 mostra como um programa simples pode ser escrito em linguagem assembly, respeitando regras básicas.

Quando se escreve um programa, além das regras fundamentais, existem princípios que, embora não obrigatórios é conveniente, serem seguidos. Um deles, é escrever no seu início, o nome do programa, aquilo que o programa faz, a versão deste, a data em que foi escrito, tipo de microcontrolador para o qual foi escrito e o nome do programador.

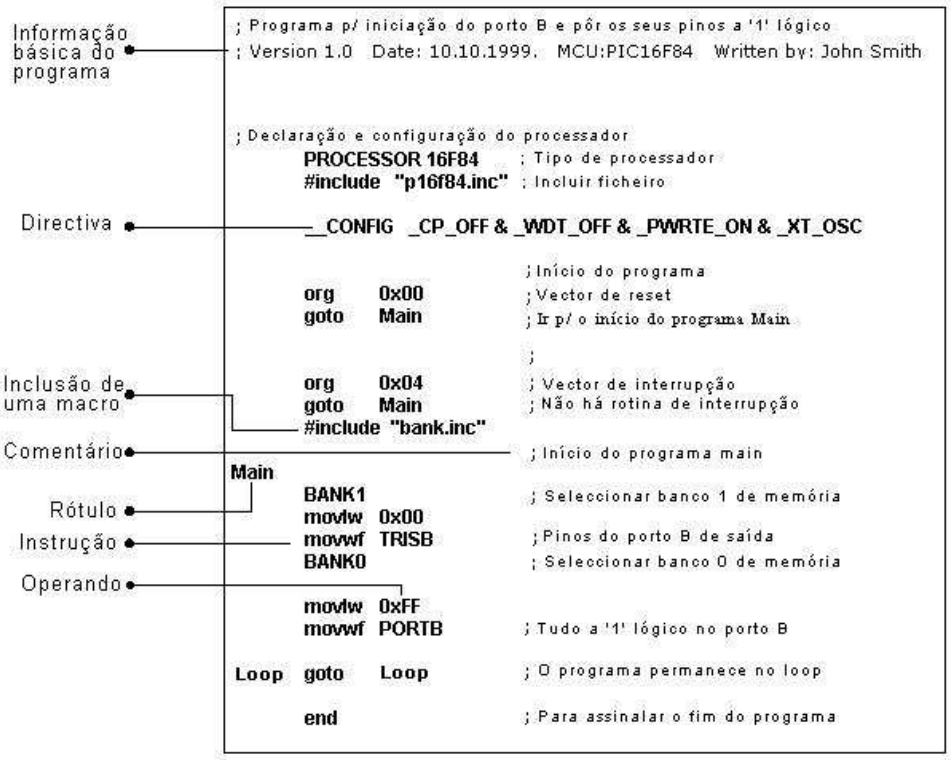


Figura 4.6: Exemplo de como se escreve um programa

Uma vez que estes dados não interessam ao tradutor de assembly, são escritos na forma de comentários. Deve ter-se em atenção que um comentário começa sempre com ponto e vírgula e pode ser colocado na linha seguinte ou logo a seguir à instrução. Depois deste comentário inicial ter sido escrito, devem incluir-se as directivas. Isto mostra-se no exemplo de cima.

Para que o seu funcionamento seja correto, é preciso definir vários parâmetros para o microcontrolador, tais como:

- tipo de oscilador
- quando o temporizador do watchdog está ligado e
- quando o circuito interno de reset está habilitado.

Tudo isto é definido na diretiva seguinte:

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC
```

Logo que todos os elementos de que precisamos tenham sido definidos, podemos começar a escrever o programa. Primeiro, é necessário definir o endereço para que o microcontrolador deve ir quando se liga a alimentação. É esta a finalidade de (org 0x00). O endereço para onde um programa salta se ocorrer uma interrupção é (org 0x04). Como este é um programa simples, é suficiente dirigir o microcontrolador para o início de um programa com uma instrução *goto Main* (Main = programa principal).

As instruções encontradas em Main, selecionam o banco 1 (BANK1) de modo a poder acessar o registro TRISB, afim de que a porta B seja definido como uma saída (movlw 0x00, movwf TRISB).

O próximo passo é selecionar o banco de memória 0 e colocar os bits da porta B no estado lógico 1 e, assim, o programa principal fica terminado. É preciso, no entanto, um outro ciclo (loop), onde

o microcontrolador possa permanecer sem que ocorram erros. Trata-se de um *loop* infinito que é executado continuamente, enquanto a alimentação não for desligada. Finalmente, é necessário colocar a palavra *end* no fim de cada programa, de modo a informar o tradutor de assembly de que o programa não contém mais instruções.

4.2 Macros

As macros são elementos muito úteis em linguagem assembly. Uma macro pode ser descrita em poucas palavras como um grupo de instruções definido pelo utilizador que é acrescentado ao programa pelo assembler, sempre que a macro for invocada. É possível escrever um programa sem usar macros. Mas, se as utilizarmos, o programa torna-se muito mais legível, especialmente se estiverem vários programadores a trabalhar no mesmo programa. As macros têm afinidades com as funções nas linguagens de alto nível.

Como as escrever:

```
<rótulo> macro
[<argumento1>,<argumento2>,.....,<argumentoN>]
.........
.........
endm
```

Pelo modo como são escritas, vemos que as macros podem aceitar argumentos, o que também é muito útil em programação. Quando o argumento é invocado no interior de uma macro, ele vai ser substituído pelo valor `argumentoN`.

```
ON_PORTB        macro ARG1
                BANK0           ;Seleccionar banco 0 de memória
                movlw ARG1      ;valor do argumento ARG1
                                ;guardado no registo de trabalho
                movwf PORTB     ;valor do argumento ARG1
                                ;guardado no Porto B
                endm            ;fim de macro
```

Figura 4.7: Exemplo de uma macro.

O exemplo da Figura 4.7 mostra uma macro cujo propósito é enviar para o porto B, o argumento ARG1, definido quando a macro foi invocada. Para a utilizarmos num programa, basta escrever uma única linha: ON_PORTB 0xFF e, assim, colocamos o valor 0xFF no porto B. Para utilizar uma macro no programa, é necessário incluir o arquivo macro no programa principal, por intermédio da instrução ♯include nome_da_macro.inc. O conteúdo da macro é automaticamente copiado para o local em que esta macro está escrita. Isto pode ver-se melhor no arquivo `lst` visto atrás, onde a macro é copiada por baixo da linha ♯include bank.inc.

Referências Bibliográficas

[1] Matic, N. *The PIC Microcontroller*, Microe, USA, 2002.

[2] Zanco, W. S. *Microcontroladores PIC 16F628A/648A. Uma abordagem prática e objetiva.* Ed. Érica. 2005.

[3] Zanco, W. S. *Microcontroladores PIC. Técnicas de software e hardware para projetos de circuitos eletrônicos. Com base no PIC 16F877A*. Ed. Érica. 2006.

[4] Souza, D. J. *Desbravando o PIC - Ampliado e Atualizado para PIC 16F628A*. Ed. Érica, 2004.

[5] Souza, D. J. e LAVINIA, N. C. *Conectando o PIC: Recursos Avançados.* Ed. Érica. 2005.

[6] *www.microchip.com.* Acessado em 02/2008.