Curso de Programação PIC

Prof. Msc. Engº Getúlio Teruo Tateoki

Este Curso de Programação foi projetado para introduzir ao aluno na programação dos microcontroladores PIC16F84.

Para se entender como programar um microcontrolador faz-se necessário de contribuição de muitas fontes diferentes. Isso inclui idéias e discussões de instrutores diferentes. Cada um oferece um ponto de vista diferente com uma terminologia para descrever uma característica, especialmente sobre algo tão complexo quanto programação.

Este trabalho, baseado no artigo de Jim Brown descreve muito das características do microcontrolador PIC16F84 e ajuda a adquirir um conhecimento geral sobre este surpreendente dispositivo.

Conteúdo:

1- Introdução

Ao se deparar na programação PIC 16F84, ao primeiro momento parece ser muito complexo. A folha de dados (Datasheet) do PIC16F84 da Microchip é um excelente documento técnico e o manual do MPASM é um guia de referência completo. Porém, utilizando somente deles são insuficientes para saber e entender a sua programação. Assim este trabalho é um tutorial para programar o PIC16F84, abrangendo o conjunto de instruções e algumas diretivas do MPASM. Pelo seu conteúdo ele cobre o próprio PIC16F84 em termos de registros, pinos e assim por diante.

No final, é esperado do aluno que ele esteja bastante à vontade para dar início aos primeiros passos de sua programação deste dispositivo.

2- O Quê é preciso

Primeiro, é preciso ter em mãos alguma da documentação da Microchip. Como um mínimo, é necessária a folha de dados (Datasheet) do PIC16F84. Ela contém toda a informação real que necessário do próprio chip. Também deve se em mãos o Guia de usuário do MPASM. Ambos estes estão disponíveis no site da Microchip (http://www.microchip.com/).

Faz-se necessário também o aplicativo MPLAB (for Windows) instalado no PC, pois o mesmo possui o Ambiente de Desenvolvimento Interativo (IDE) deles, e contém um editor, o assembler e um simulador. Aqui, você pode criar seu código fonte (editor), produzir o código executável (assembler) e executar no seu PC (simulador). No simulador, pode-se observar a execução do programa enquanto mantém a vista nos registros e pode-se até mesmo simular eventos como mudanças causadas por fatores externos nos pinos de I/O. É interessante realmente adquiri-lo, pois também está disponível como o manual do MPASM. Entendendo-se os apéctos básicos e funcionais destes aplicativos irá a facilitar muito o entendimento da maioria dos tópicos que está sendo abordado neste trabalho. O Simulador (Emulador) é muito bom para observar os registradores e executar um programa em um modo passo a passo, mas isso não resolve todos os problemas quando uma eventual falha ocorrer.

O modo passo a passo não leva em conta o efeito de uma entrada em um programa e uma rotina de delay deve ser ignorada para se prevenir horas de espera para a execução do programa.

Isso é tudo que é preciso, mas há muito mais coisas das quais pode ser apreciado. Não menos, um chip PIC16F84 e um programador. O programador GTP-Gravador Testador PIC apresentado no curso é o equipamento mais barato no mercado e vem com a vantagem da não necessidade retirar o microprocessador da placa para se efetuar os testes o que tornaria os processos de desenvolvimento ou estudo mais moroso e de certa forma evitaria assim erros ou acidentes durante o intercambio entre o gravador e a placa de testes, caso fossem utilizados separadamente.

Para um trabalho mais complexo, as Notas de Aplicação da Microchip (ANs) também são de extrema ajuda. Estas notas cobrem todos os tipos de usos dos PIC's, com muitas dicas úteis para uso no mundo real. As ANs estão em formato Adobe .pdf: Usando o Adobe Acrobat você pode procurar pelas ANs por situações como 'motor' ou 'serial' e ler as anotações destinadas às suas necessidades.

Um das notas, **AN585** sobre sistemas operacionais de tempo real para o PIC, refere-

se à Programação de Tempo Real - Tópicos Negligenciados. É de extremamente interessante que se tenha uma cópia pois é em tutorial fascinante sobre todo o assunto das interrupções, controle de laço-fechado e similares.

3- O que é preciso saber

Para uma maior facilidade de entendimento e conseqüentemente um melhor aproveitamento nas atividades deste trabalho proposto, seria interessante ter algum conhecimento prévio sobre algumas terminologias básicas dos computadores, como bits, bytes & EEPROM e conceitos como binário e hexa.

4- Arquitetura do PIC16F84

Um microcontrolador como o PIC16F84 é por sua natureza, um computador completo em um chip.Ele possui: Um processador, registradores, programa e memória de dados.

Isto o faz diferente de uma CPU, (Unidade de Processador Central) que tem só o processador e registradores.

- O PIC16F84 tem um processador de 8-bits, e isso é tudo que nós precisamos saber.
- Os 90 registradores são a área de armazenamento interno do processador. Estes têm nomes como (STATUS, PORTA, etc). Alguns têm funções especiais que é examinado neste trabalho. Por exemplo, PORTA armazena os conteúdos de uma das portas de I/O, isto é a chamada porta A. Dos 90 registradores, 68 são registradores de propósito gerais, e podem ser considerados como utilitários (ao contrário daqueles especiais do processador) pois podem ser usados como 'bloco de rascunho''.

- O PIC16F84 tem uma memória de programa de 1k. Isso significa que há 1024 locais para programa. A memória vai do endereço 000h até o 3FFh. 3FFhex são 1023 em notação decimal, sendo assim há 1024 locais que incluem 000.

- O PIC16F84 tem 64 bytes de memória de dados, chamados de memória de EEPROM a qual é usada para armazenamento de, por exemplo, valores para ajuste de dados recebidos do mundo externo. O uso dessa memória EEPROM será analisado por último. Além disso, o dispositivo inclui quatro tipos de interrupção e 13 pinos de I/O.

- Interrupções são uns meios de parar a execução do programa e atender a uma operação definida pela interrupção. Por exemplo, o acionamento de um botão poderia interromper o fluxo normal do programa e poderia executar alguma operação especial. É examinadas posteriormente as técnicas de interrupção do PIC16F84.

- O uso dos pinos de I/O é a chave para o uso de um dispositivo como os '84, uma vez que qualquer processo consiste em 3 partes: a entrada para o processo, o próprio processo, e a saída. O PIC16F84 tem 13 pinos configurados como 2 porta: porta A tem 5 pinos e a porta B tem 8; e será usado mais adiante.

5- Conjunto de instruções

Há 35 instruções no conjunto de instruções do PIC16F84, consistindo em um **opcode** (código de operação) e operando(s). Basicamente, o opcode especifica o que fazer e o(s) operando(s) especificam como ou onde. Estas instruções são divididas em 3 grupos: orientadas por byte, orientadas a bit e literal & controle. Mais tarde será utilizada cada uma dessas operações.

Por enquanto, serão analisadas apenas para uma das instruções. Será focado mais no formato, em lugar da função, a qual será visto posteriormente logo mais adiante. Foi escolhida a instrução ADDWF. A sintaxe desta instrução é:

ADDWF f,d

Onde ADDWF é o opcode e "f d" são os operandos.

As letras ADDWF são chamadas de um mnemônico. Esta é uma abreviação que tanto o computador quanto os humanos podem entender. Será visto este mesmo layout em todas as instruções.

O registrador de trabalho é um registrador especial (acima de todos já mencionados), conhecido como registrador W e é onde a unidade aritmética e lógica (ALU) faz os cálculos. Qualquer instrução com W no nome age em W. Os outros registradores são conhecidos como arquivos e nós usamos a letra 'F' nos comandos para indicar que um arquivo está sendo acessado. Assim, nós podemos entender que o comando ADDWF soma W ao arquivo F.

Mas o que é arquivo F?

Para cada arquivo (diferente de W) é determinado um endereço hexadecimal. Por exemplo, o arquivo 0C é chamado "0C", arquivo 10 é chamado "10" e arquivo 2F é chamado "2F". A letra minúscula "h" sempre é incluída para indicar que arquivo tem um valor hexadecimal, como arquivo 10h, por exemplo, que é na realidade o décimo sexto arquivo na memória.

Alguns arquivos têm nomes, como por exemplo: PORTA é 05h, PORTB é 06h e TRISA é 85h. O 'f' na instrução acima é o endereço atual do arquivo. Não há nenhum arquivo chamado "F" ou "f". Isso é um símbolo para representar qualquer um dos arquivos. Assim, é codificada a instrução: "some W ao arquivo 3C" como:

ADDWF 3Ch,d

Bem, não totalmente - para que serve aquele 'd'?

É o destino do resultado, e será visto isto em muitas instruções. Dependendo da exigência, o resultado da instrução pode ser posto no registro de funcionamento (W) ou no próprio arquivo F.

Se 'd' for 0, o resultado estará em W,

Se 'd' for 1, o resultado estará em F.

A instrução pode ser seja qualquer um do seguinte:

ADDWF 3Ch,0; soma W ao arquivo 3C, resultado em W

ADDWF 3Ch,1; soma W ao arquivo 3C, resultado em 3C

6- Um programa simples para o PIC'84

Este programa de exemplo serve para vários propósitos. Além de mostrar como usar alguns instruções, apresenta também alguns conceitos do assembler e vai também mostrar algumas técnicas simples para o simulador.

O programa, Simple.asm é apresentado abaixo; Eu lhe guiarei linha por linha.

Programa 1: Simple.asm

```
;simple.asm
;para demonstrar o aspecto de um programa
;e introduzir algumas instruções e diretivas
;****************** setup ***************************
processor 16F84        ;tipo de processador
org 0010h              ;origem do programa na memória
w equ 0                ;para instruções de byte, use w & f
f equ 1                ;  ao invés de 0 & 1, isso é bem mais claro
MyReg_1 equ H'10'      ;posição Meus Registradores na memória
MyReg_2 equ H'15'      ;  em h'10' & H'15'
;****************** programa ******************************
;Nós vamos carregar o acumulador (reg w) com valores
; e efetuar algumas operações aritméticas em W utilizando
; MyReg_1&2
movlw H'08'            ;coloca o valor H'08' no registrador w
movwf MyReg_1          ;move o conteúdo de w para MyReg_1
; nota – é o mesmo que movwf 10h, já que
; MyReg_1 e 10h são a mesma coisa.
movlw 32h              ;coloca 32h no registrador W
addwf MyReg_1,f        ;soma o conteúdo de w àquele em MyReg_1
; a resposta vai para MyReg_1, devido ao f.
movlw 92h              ;coloca o valor 92h no registrador w
movwf MyReg_2          ;move o conteúdo de W para MyReg_2
; nota – é o mesmo que movwf 15h, já que
; MyReg_2 e 15h são a mesma coisa.
movlw 26h              ;coloca 26h no acumulador W
subwf MyReg_2,w        ;subtrai w de MyReg_2
; a resposta vai para w
end
```

Observando-se este exemplo, pode-se verificar o que irá ser encontrado depois de cada diretiva ou instrução. Qualquer coisa depois de um **';'** é um comentário. Todo livro de programação recomenda a criação de comentários para explicar o que está acontecendo.

Na seção chamada 'setup', nós encontramos três diretivas do assembler:

"processador 16F84" informa ao assembler para qual chip o programa foi escrito. Se isto estiver incorreto, o código será preparado pelo assembler para o processador errado
.
"org" diz ao assembler aonde começar a colocar o código na memória de programa. Você deve colocar seu código além de 004h para ficar claro o endereço para onde o processador deve ir quando uma interrupção é descoberta.

"equ" é uma equivalência ou "igual a." Simplesmente significa que os itens em qualquer lado de equ significam a mesma coisa. Por exemplo, a instrução ADDWF espera um 0 ou 1 no lugar de 'd': igualando 'w' a 0' significa que podem ser utilizados 'w' na instrução em lugar de 0'. Isto é mais fácil de se lembrar durante a codificação, e para ler depois. Analogamente, igualando MyReg_1 a 10h toda vez que ao se referir ao registrador, será mais fácil de fazê-lo através do seu nome mais significante e fácil de lembrar.

Na parte chamada 'programa', é encontrada várias instruções do PIC16F84. Podem ser conferidas as descrições completas que será objeto de estudo mais tarde.

MOVLW k faz o valor k (no exemplo 08h) ser colocado no registrador de trabalho.

MOVWF f copia o conteúdo de W no registrador f. Notar que movwf é estritamente uma nomenclatura equivocada, já que na realidade ocorre uma cópia (W não é esvaziado), e não uma movimentação (na qual W seria esvaziado). Notar também a convenção da Microchip de comparar movlw e movwf: descrevendo a operação os parênteses () significam 'o conteúdos de'. Assim k→ (W) quer dizer que o valor k se torna o conteúdo de W; (W)→ (f) meios que o conteúdo de W se tornam o conteúdo de f.
Por último, certificar de entender o uso do conceito de equ com respeito a registradores. O **' f '** em **MOVWF** f refere-se a um arquivo. Ele mostra que arquivos têm endereços hexadecimais e de se esperar ler a instrução como MOVWF 10h por exemplo. Igualando MyReg_1 a 10h, quer dizer que podem ser escritos MOVWF MyReg_1 com o mesmo resultado.

ADDWF f,d e SUBWF f,d respectivamente executam a adição e subtração aritmética nos conteúdos de W e f. Note o uso de equivalências aqui também. Podem ser referidos a MyReg_1 como antes, e também substituir os valores permitidos de d (no exemplo 0 e1) por w e f respectivamente.
Conseqüentemente podem ser escritos ADDWF MyReg_1,f no lugar de ADDWF 10h,1.

Montar e executar esse programa.

7- Usando MPLAB para depurar o programa

Este é um processo de três passos: Editar o código fonte, montá-lo e então executá-o no simulador.

Com o *exemplo.asm* montado com sucesso, e o editor sendo a única janela aberta no MPLAB, executar o programa passo a passo (o ícone de pegada). Não ajuda muito – pode ser visto cada linha do programa ser realçada à medida que é ativada, mas isso é tudo.

Assim, pode-se começar a usar algumas das outras facilidades do simulador. Sugere-se que seja aberta a seguinte janela no MPLAB. Todas servem a propósitos semelhantes - isto é, ver o que está acontecendo nos registradores - mas todas executam diferentemente. Seria útil ter todas abertas simultaneamente para comparar as implementações:

Janela > File Register

Esta janela lista o conteúdo de todos os registradores de arquivos de 00 a 4F - ie, não mostra aqueles na região 80 a 8B. À medida que se caminha pelo programa poderá ser visto o conteúdo hexadecimal dos registradores 10h e 15h mudando. Note que qualquer registrador cujo conteúdo mudou há pouco é mostrado em vermelho - e volta para azul no próximo passo (a menos que haja outra mudança).

Janela > Special Function Registers

Aqui você pode ver os SFRs (Registradores de Função Especial) por nome ao invés de local, que é o que mostra a janela anterior. O conteúdo é mostrado em decimal, hexa e binário.

Janela > New Watch Window

Esta é sua janela personalizada para os registradores, e é permitido selecionar quais registradores se quer monitorar. Todos os registradores estão disponíveis para serem escolhidos, SFRs e GPRs. Quando se escolher esta janela é mostrada uma caixa de seleção com uma seta para baixo. Clicar na seta, e uma lista de todos os registradores aparece: escolher qual irá quer (eg, Myreg_1) e então clicar OK. Será visto o conteúdo do registrador exibido na janela. Para adicionar outros registradores pode-se clicar no ícone no canto esquerdo superior da janela e então escolher Add Watch; ou simplesmente apertar a tecla insert (enquanto a janela está ativa). Você pode salvar as combinações usadas com mais freqüência como arquivos watch, e então carregá-las depois através da Janela> Load Watch Window.

Pode-se perguntar de onde veio a lista de nomes de registradores para selecionar. Os SFRs são óbvios: o MPLAB sabe a respeito deles de qualquer maneira. Os outros, como MyReg_1 e2, entram das diretivas de equ em seu código, o que é excelente. Porém, em neste exemplo, foi utilizado também w equ 0 e f equ 1 embora estes não são planejados como nomes de registro. Porém, eles ainda aparecem na lista de possibilidades que pode estar o problema de não se querer escolher. Porém não se pode esquecer que aquele W é um registrador: então agora há duas entradas de W na lista, uma desde o princípio (o registro de funcionamento) e a que foi criada a outra com equ. Usar qualquer uma delas, causa um erro de símbolo não encontrado em sua tabela de observação, o que significa que não se pode monitorar o registro de funcionamento se foi declarado um equ de W em seu código. Tentar, então comentando a linha com erro com um ‘;', re-montar e tentar novamente.

Agora se pode usar o registro de trabalho no monitoramento.

Com estas três janelas, caminhar pelo seu programa: agora você tem alguma noção do que está acontecendo.

8- O Conjunto de Instruções

A Microchip tem detalhes completos do conjunto de instruções. Elas se agrupam em 3 categorias, o que não consideradas particularmente útil. Estas categorias são operações Orientadas a Byte, a Bit e a Literal/Controle. O interessante seria em agrupar as instruções dentro daquilo que é utilizado para cada comando, tais como executar operações aritméticas, tal qual é feito neste trabalho logo a seguir. Para cada instrução, é explicada a operação. (Por exemplo, há uma explicação o que um Inclusive-OU de fato está em IORWF). Talvez o mais importante ainda é os exercícios sugeridos para usar as instruções apresentadas que são bastante simples, mas permitem usar cada instrução, e também praticar o uso do simulador. Lembrar de se usar os recursos, como a janela de monitoramento para ver o que está acontecendo a cada passo do seu programa: prestar atenção nos registradores que se decidiu usar, bem como nos registradores W e STATUS.

Formato da instrução

O agrupamento da Microchip mantém as instruções de formato semelhante juntas. As instruções aparecem assim:

byte_command f,d
onde f é a designação do arquivo do registrador e d é o destino; se d=0, o resultado vai para W. Se d=1, o resultado vai para o registrador f.

bit_command f,b
onde f é o arquivo do registrador, e b é o bit dentro dele; bits são numerados da direita para a esquerda de 0 a 7. No texto, um bit é escrito como FILE_REG <n>, por exemplo INTCON <4> significa o bit número 4 no registrador intcon. O registrador INTCON fica no local 0B.

other_command k
onde k é uma constante ou literal de 8-bit.

O registrador STATUS

O '84 têm o registrador STATUS em 03h. Ele contém o estado aritmético da unidade de aritmética e lógica. Muitas instruções do '84 afetam certas partes do STATUS. Muitas instruções afetam STATUS <2>, Z - a flag Zero que é ligada se o resultado de uma operação for zero. Certas operações afetam os bits de transporte: STATUS <0>, C - o bit Carry, é ativado se um transporte ocorrer no bit mais a esquerda no resultado; STATUS <1>, DC - o bit Digit Carry, é ativado se houver um transporte entre os dígitos hexadecimais (ie, do meio-bit à -bit 3 - ao bit à esquerda -bit 4). Dois comandos afetam STATUS <3>, o bit Power Down PD, e STATUS <4>, o bit Time Out TO.

8.1: Instruções Move

É encontrado algumas dessas instruções, que nada fazem além de colocar coisas nos registradores.

MOVF f,d (Move f)
(f) → (dest)

MOVWF f (Move W to f)
(w) → (f)

MOVLW k (Move literal to W)
k → (W)

Exercício: Escrever um programa para usar estes comandos para (por exemplo) por algo em W, mover de lá para outro registrador, depois colocar alguma outra coisa em W, e então mover a coisa original de volta para W.

Solução:

Programa 2: Moves.asm

```
;moves.asm para mostrar como MOVF, MOVWF & MOVLW funcionam
;*********************************  simulador      ***
;watch window: reg1, w, pcl
;*********************************  setup          ***
processor     16F84
reg1    equ           h'10'
;*********************************  programa       ***
start:  movlw   h'05'   ;carrega w
movwf   reg1    ;move (w) para reg1
movlw   h'82'   ;altera w
movf    reg1,0  ;restaura w
end
```

8.2: Instruções Clear

Estes dois comandos limpam um registrador.

CLRF f (Clear f)
00h → (f)

CLRW (Clear W)
00h → (W)

Exercício: Ampliar o programa anterior para limpar os registradores no final.

Solução:

Program 3: Clears.asm

```
;clears.asm para mostrar como clrf & clrw funcionam
;baseado em moves.asm
;*********************************  simulador
;watch window: reg1, w, pcl
;*********************************  setup
processor     16F84
reg1    equ           h'10'
;*********************************  programa
start:  movlw   h'05'         ;carrega w
movwf   reg1    ;move (w) para reg1
movlw   h'82'   ;altera w
```

```
movf    reg1,0  ;restaura w
clear:  clrf    reg1        ;limpa reg1
clrw        ;limpa w
end
```

8.3: Instruções Aritméticas

Executar operações aritméticas é muito importante. O '84 pode apenas somar e subtrair. A aritmética ocorre entre W e um registrador f:

ADDWF f,d (Add W & f)
(W)+(f) → (dest)

SUBWF f,d (Subtract W from f)
(f)-(W) → (dest)
or between W and a literal:

ADDLW k (Add literal & W)
(W)+k → (W)

SUBLW k (Subtract W from literal)
k-(W) → (W)

Exercício: Use estes e os comandos anteriores para carregar um registrador a partir de W e efetuar algumas adições e subtrações. Preste atenção no bit Carry e no bit Zero no registrador STATUS.

Solução:

Program 4: Arith.asm

```
;arith.asm para mostrar o uso de ADDWF, SUBWF, ADDLW, SUBLW
;************************************        simulador
;watch window: reg1,reg2,status,w,pcl
;************************************        setup
processor     16F84
reg1    equ         h'10'
reg2    equ         h'12'
;************************************        programa
loads:  movlw   d'20'   ;carrega w
movwf   reg1    ;carrega reg1
movlw   d'80'   ;carrega w mais uma vez
movwf   reg2    ;carrega reg2
arith:  addlw   d'05'   ;adiciona d'05' a w
sublw   d'100'  ;subtrai w de d'100'
addwf   reg1,1  ;soma w a reg1, dentro de reg1
subwf   reg2,1  ;subtrai w de reg2, dentro de reg2
end
```

Observação: Pode-se ver que a subtração é feita usando o método do segundo complemento. Este é o modo normal que a subtração acontece com binários. Isto pode ser explicado com um exemplo.

Primeiramente, se expressa ambos os números em binário. Deixar o número a partir do qual está se subtraindo inalterado. Formar o segundo complemento do que está sendo subtraído assim: mudar todos os 0's para 1's e todos os 1's para 0's (este é o complemento), somar 1 ao dígito à direita dígito, transportando à esquerda conforme necessário. Agora acrescentar o resultado ao outro número inalterado. Descartar o transporte à esquerda, se houver. O resultado é a resposta. Pode-se conferir...
Por exemplo, subtrair 20 de 27 . O resultado obtido deve ser 7. O procedimento deve ser como se segue:
Converter para binário: 27 → 11011 x

20 → 10100 . . . . y

Fazer o segundo complemento de y: complemento 0: 01011

somar 1: 01100 . . . . z

Somar x e z: + 11011 x

(1) 00111 = 7

O 1 entre parênteses é o transporte, que é descartado.

8.4: Funções Lógicas

Nesta fase, antes de se examinar as funções lógicas providas pelo 84, podem ser discutidas as funções lógicas em geral. Considerar um dispositivo eletrônico digital que apenas atenda em valores binários de 1 e 0 com 3 fios ligados. Supondo-se que 2 fios são entradas, e o resultado no terceiro fio dependa das entradas. As relações que podem existir entre as 2 entradas e a saída são o resultado das combinações de entrada que são: 00, 01, 10 e 11. O fio 3 podem ser 1 se, e somente se, as entradas forem ambas 1 (i.e: 11) ou se uma ou ambas as entradas forem 1 (i.e: 01, 10, 11), e outras combinações. As relações básicas são conhecidas como E e OU (AND e OR). AND significa ambas as entradas enquanto OR significa qualquer uma ou ambas. A maioria das explicações a respeito disso, resultam em uma TABELA da VERDADE, desenhada na tabela 1 para as seguintes operações lógicas: AND, OR, XOR, NAND, NOR.

Tabela 1 – Tabela Verdade das Funções AND, OR, XOR, NAND e NOR

Entradas A B	A **AND** B	A **OR** B	A **XOR** B	A **NAND** B	A **NOR** B
	ambos	ou, ambos	ou, não ambos	não ambos	nem um, não ambos
00	0	0	0	1	1
01	0	1	1	1	0
10	0	1	1	1	0
11	1	1	0	0	0

A função AND significa que o resultado só será 1 quando ambas as entradas forem 1. OR significa que qualquer uma das entradas pode ser um 1 para a saída ser 1, mas assim também será se ambas as entradas forem 1. A função XOR quer dizer "eXclusive OR", e significa dizer que qualquer entrada sendo 1 fará o resultado na saída ser 1, e, especificamente exclui a situação onde ambas as contribuições forem 1. Finalmente, NAND e NOR são as negações de AND e OR respectivamente: compare as colunas e você verá o que isso significa.

A propósito, a função OR (ao contrário da função de XOR) às vezes é conhecida como "Inclusive OR" (IOR). O PIC16F84 usa este termo.

O PIC16F84 provê várias operações lógicas que agem em dois valores de 8-bits; esses valores são comparados bit a bit. Por exemplo - sem olhar para uma instrução do '84 - considerar o ANDing dos números $(5F)_H$ equivalente a $(95)_{10}$ ou $(0101\ 1111)_2$ e $(A3)_H$ que é $(163)_{10}$ ou $(1010\ 0011)_2$; resultando em $(03)_H$ que é $(3)_{10}$ ou $(0000\ 0011)_2$.

5F: 0101 1111

A3: 1010 0011

and: 0000 0011

Claramente, só nas 2 posições mais à direita AND é satisfeito. O resultado aí é 1, e 0 em qualquer outro lugar.

As instruções fornecidas pelo 'PIC16F84 são:

A comparação ocorre entre os registradores W e f:

ANDWF f,d (AND W with f)
(W) **AND** (f) →(dest)

IORWF f,d (Inclusive OR W with f)
(W) **OR** (f)→ (dest)

XORWF f,d (Exclusive OR W with f)
(W) **XOR** (f) → (dest)

ou entre W e uma literal:
ANDLW k (AND literal with W)
(W) **AND** k → (W)

IORLW k (Inclusive OR literal with W)
(W) **OR** k → (W)

XORLW k (Exclusive OR literal with W)
(W) **XOR** k→ (W)

Pode-se notar que o '84 não suporta as funções **NAND** ou **NOR**. Ao contrário, ele provê meios de complementar (negar) um registrador; isso significa que para realizar **NAND** deve-se primeiro efetuar **AND** e então:

COMF f,d (Complementa f)
complemento de (f) → (dest)

Exercício: É sugerido aqui fazer duas coisas. Primeiro, usando a calculadora do Windows, verificar os resultados acima: isto irá assegurar que foi amplamente entendido o conceito. Então, escrever para um programa em assembler para conferir as instruções do '84, carregando os registros apropriados e executando as operações e conferindo os resultados.

8.5: Decrementando e Incrementando

Duas instruções simples podem decrementar ou incrementar o conteúdo de um registrador, assim:

DEC f,d (Decrementa f)
(f)-1 → (dest)

INC f,d (Incrementa f)
(f)+1→ (dest)

Exercício: Em um programa, talvez aproveitando um dos anteriores nos quais foi realizado alguma aritmética ou alguma lógica, verificar o funcionamento destes comandos. Ficar atento à flag Zero, que é ativada se qualquer comando fizer o registro em questão, zerar: Este fato pode ser utilizado depois para se basear nos dois outros comandos.

8.6: Ativando e limpando Bits

Usando os dois comandos a seguir, você pode ativar ou limpar qualquer bit “b” no registrador “f” por duas razões:

1- Como exemplo, o registrador em questão pode ser uma porta, controlando algum equipamento externo. Cada bit pode estar ativando um dispositivo diferente: um motor, uma luz, ou o que quer que seja. Ativando e limpando cada bit, liga e desliga cada dispositivo.

2- Como é visto no item 8, pode-se usar o fato de um bit estar ativado ou não para pular instruções no programa. A capacidade de ativar ou limpar qualquer bit, dá a habilidade para controlar esse processo.

BCF f,b (limpa Bit em f)
0 → (f<b>)

BSF f,b (ativa Bit em f)
1 → (f<b>)

Foram lidas essas duas operações como 0 (ou 1) se tornando o conteúdo do bit ‘b’ no registrador ‘f’.

Exercício: Colocar esses comandos em qualquer um dos programas anteriores e observar as mudanças a bits individuais em sua janela de monitoramento.

8.7: Controle de Programa

Por muitas razões, há a necessidade de se controlar o fluxo através do programa. Normalmente, o fluxo procede linearmente a partir do topo; e freqüentemente isto não é adequado às necessidades.

Primeiramente, pode-se precisar efetuar laços por certas instruções: poderia se ter um sistema no qual um determinado processo seja contínuo como por exemplo, o controle de um transportador que continuamente acrescenta ingredientes a um depósito. Aqui, quando um depósito é realizado é voltado ao topo e passa-se novamente pelos mesmos passos somando mais depósitos a cada passagem.

Segundo, pode ser uma parte do programa que será útil em muitas outras partes. Em lugar de se repetir este código em muitos lugares, separa-se esse pedaço do restante do código; então nós simplesmente chamamos isto tão freqüentemente quanto precisarmos. O pedaço reutilizável é chamado de SUB-ROTINA. A seqüência de dados devolve o controle ao ponto em que foi chamada quando terminar.

Observando-se os laços (looping) primeiro:

GOTO k (Vá para um endereço)
k → (PC)

Esta instrução 'vai para k', e faz isso carregando o endereço k dentro do contador do programa, PC. Para usar GOTO deve-se primeiro informar para onde quer que o programa vá, utilizando para isso um **Label**. Então se usa GOTO **Label**.

Exercício: Modificar um dos programas que já foi escrito. Pode-se colocar um rótulo (label) perto do início, e um GOTO mais adiante. À medida que se anda pelo programa verá realçada a linha em que seu programa está fazendo o loop. Olhar para o contador do programa (PCL) na janela de monitoramento e conferir isso.

Pode-se examinar agora a seqüência de dados. É preciso entender o conceito de stack (pilha). A pilha - que tem 8 níveis no PIC16F84 é o lugar onde o endereço da próxima instrução é colocado, quando uma instrução CALL é encontrada. À medida que cada instrução é executada, é trabalho do Contador do programa (PC) saber onde o microcontrolador está, a qualquer tempo. A colocação do próximo endereço na Pilha diz ao microcontrolador aonde ir, depois que uma sub-rotina tenha acabado.

É referido como popping para carregar a pilha, isto é lendo o último valor e girando a pilha para cima. A pilha só é acessível pelo topo: assim como uma pilha de pratos. O topo da pilha é abreviado por TOS (Top of stack).

Há 2 instruções associadas com qualquer sub-rotina - uma para enviar o microcontrolador à sub-rotina, e a outra para trazê-lo de volta:

CALL k (Call sub-rotina)
(PC)+1 → TOS, k → (PC)

A chamada **CALL** para uma sub-rotina empurra o PC+1 atual para o topo da pilha, então altera o PC para o endereço k. Isto resulta em um salto no fluxo do programa para a sub-rotina, mas o endereço para voltar após a execução da sub-rotina está preservado na pilha para recuperação posterior.

RETURN (Retorna da sub-rotina)
TOS → (PC)

RETURN é a última instrução da sub-rotina. A instrução de retorno lê e remove o último dado da pilha e assim o programa retorna ao lugar certo. Pensar na profundidade da pilha: significa que chamadas podem ser encadeadas, e o fluxo estará correto contanto que cada chamada tenha um retorno correspondente.

RETLW k (Retorna com literal em W)
k → (W), TOS → (PC)
Isto retorna com k → (W) somado.

RETFIE (Retorna de Interrupção)
TOS → (PC), 1 → GIE

Quando uma interrupção acontece, o PC é empurrado para a pilha da mesma forma que acontece com uma chamada de sub-rotina, então RETFIE gira a pilha. Também, a ocorrência de uma interrupção incapacita interrupções adicionais: alguém não quer as interrupções sejam interrompidas. O que acontece é que a interrupção faz com que a flag de interrupção global, GIE (INTCON <7>) seja fixada em 0. Voltar de uma interrupção significa que devem ser permitidas interrupções adicionais, conseqüentemente RETFIE fixa GIE de volta a 1.

8.8: Ignorando instruções

Existem 4 instruções que permitem que se ignore a instrução seguinte.

DECFSZ f,d (Decrementa f, ignora se 0)
(f)-1 → (dest), ignora se o resultado for = 0

INCFSZ f,d (Incrementa f, ignora se 0)
(f)+1 → (dest), ignora se o resultado for = 0

Os comandos acima são baseados nos comandos DECF e INCF vistos anteriormente. A parte SZ significa ‘skip if zero' (ignora se for zero).

Exercício: Conferir essas 2 instruções carregando um valor inicial em um registrador chamado 'cont.' Então usar qualquer instrução para mudar esse valor, efetuando laços nesse bloco de código. Conferir para ver se a instrução que segue o decfsz ou incfsz (provavelmente um GOTO, para causar laço) é ignorada ou não, como apropriado.

BTFSC f,b (Testa Bit f, ignora se limpo)
skip if (f<b>)=0

BTFSS f,b (Testa Bit f, ignora se ativado)
skip if (f<b>)=1

Ler estas instruções como ‘testa bit f e ignora se ativado’.

Exercício: Escrever um programa com um bloco de código com um loop. Colocar um BTFSS no final, seguido por um GOTO para retornar ao início. Incluir um pouco mais de código. O BTFSS precisará referir-se a uma das portas de I/O do PIC16F84 como f $(05)_H$ ou $(06)_H$ para PortA e B respectivamente), e pode se usar qualquer pino de 0 a 4 na PortaA, 0 a 7 na PortaB. Deve-se ter outro GOTO exatamente no final para efetuar um loop mais externo para o começo, assim pode-se ver o loop do programa para ver o que acontece quando um pino de entrada sofre alteração.

Há um modo fácil de se fazer com que o pino mude de estado no simulador MPLAB. Ir em Debug> Simulatos stimulus> Asynchronous stimulus; Poderá ser visto uma tabela de botões de Stim0 até Stim12. Clicar com o botão direito em qualquer um, por exemplo, Stim7, e clicar em Assign pin. Dar um duplo-clique no pino que foi escolhido para usar como sensor, talvez fosse RA3. Stim7 então muda para RA3. Agora clicar com o botão direito novamente no botão. Pode-se escolher “pulsar”, “tornar LOW”, “HIGH” ou “chavear o pino”. Aqui vai ser escolhido chavear o pino.

Agora podem se efetuar os testes. Caminhar pelo programa. Dependendo do estado inicial do pino, o qual ainda é desconhecido na ligação da alimentação, o programa fará o loop ou não. (No loop mais interno, isto é, ele sempre voltará para o início a partir do final, para propósitos de teste) A qualquer hora que quando quiser, clicar com o botão esquerdo no botão “Stim” que se escolher, que agora pode se saber que é RA3. (Se tiver a exibição das portas em uma janela de monitoramento, será visto o pino mudar de estado nos limites da próxima instrução). De qualquer modo, da próxima vez o programa chegar ao passo BTFSS, ele deveria se comportar diferentemente, já que foi chaveado o pino e assim o BTFSS deveria ter uma resposta diferente.

8.9: Rotações e Trocas

Três instruções permitem a manipulação os bits dentro de um registrador. Destas, duas deslizam os bits para a direita ou para a esquerda (através do bit de transporte), e a terceira inverte os dois meio-bits (nibbles) do registrador.

RRF f,d (Rotaciona f à direita através do bit carry).

Cada bit no registrador f é deslocado 1 para a direita. O que sai fora para a direita é movido para carry, e carry é movido para a esquerda.

RLF f,d (Rotaciona f à esquerda através do bit carry).

Cada bit no registrador f é movido 1 para a esquerda. O que sai fora pela esquerda é movido para dentro de carry, e carry circula para a direita.

SWAPF f,d (Inverte os nibbles em f)
(f<3:0) → (dest<7:4>), (f<7:4>) → (dest<3:0>)

Exercício: Observar o efeito desses comandos no conteúdo de um registrador.

8.10: Sleep e o Timer Watchdog

Estes dois comandos estão relacionados. O WDT (Watchdog Timer) é um dos caminhos para se despertar do comando sleep.

SLEEP (Sleep)
0 → WDT, 0 → WDT prescaler, 1→ TO, 0 →PD

CLRWDT (Clear watchdog timer)
0→ WDT, 0→ WDT prescaler, 1→ TO, 1→ PD

8.11: Miscelânea

NOP (Nenhuma operação) Esta instrução não faz nada. Ela é bastante útil para a criação de pequenos intervalos de atraso (1µs) para permitir que alguma coisa seja pré-ajustada antes que outra operação seja executada.

TRIS 05 Esta instrução controla a direção de cada linha da porta chamada Porta A.
É o Registro de **Controle de Direção (DCR).** Há 5 linhas na **porta A** chamadas de RA0, RA1, RA2, RA3 e RA4. Cada linha pode ser de **entrada ou de saída.** Para fazer uma linha entrada, o bit correspondente no registro TRIS é definido como "1. " Para fazer uma linha saída, o bit correspondente é definido como "0. " Assim o valor 01 fará RA0 uma entrada e todas as outras linhas serão saídas.

O valor 23 (0010 0011) fará RA0 e RA1 entradas e o bit5 não terá nenhum efeito, pois nenhuma linha corresponde a esse bit.

TRIS 06 Esta instrução controla a direção de cada linha na **porta B**.
Existem 8 linhas na porta B. RB0, RB1, RB2, RB3, RB4, RB5, RB6 e RB7. Um "0" no registrador TRIS torna a linha correspondente na porta B uma Saída. Um “1” no registrador TRIS torna a linha uma Entrada.
Isso é fácil de ser lembrado, pois “0” é similar a **0**utput (saída) e “1” é similar a **1**nput (entrada).

Não se deve usar a instrução a seguir, que foi incluída para efeitos de compatibilidade.

OPTION (Não recomendado)

9- Interrupções no '84

Interrupção é uma forma simples de atrair a atenção no computador. Uma vez uma interrupção tenha sido tratada, o controle tem que voltar para onde estava anteriormente. Não só isso, mas todo o estado do sistema deve ser recuperado. Se qualquer registrador foi mudado durante o controle da interrupção, ele deve ser restaurado.

9.1- Tipos de Interrupção e o registrador INTCON

O PIC16F84 tem 4 tipos diferentes de interrupção:
. uma interrupção externa no pino 6, também conhecido como: RB0
. uma mudança em qualquer dos pinos de 10 a 13, também chamados RB4 a RB7
. um estouro no cronômetro
. uma de gravação completa na EEPROM.

Para permitir interrupções, e para determinar a sua origem, o processador utiliza o registrador INTCON (0Bh). Cada bit serve a um propósito específico, e interrompe o trabalho de acordo com o seu valor (ativado ou não).

Primeiramente, interrupções, como um todo, podem ser habilitadas ou desabilitadas pelo bit INTCON <7>, o GIE (Habilitação de Interrupção Global). Se esse bit estiver zerado, então nenhuma interrupção pode acontecer: Esse é o valor no power-up. Também, se uma interrupção acontecer, então o GIE é zerado para prevenir a interrupção de ser suspensa; o retorno da interrupção com a instrução RETFIE re-habilita as interrupções.

Segundo: cada um dos tipos de interrupção deve ser habilitado com seu próprio bit em INTCON, antes de poder ser utilizada:

- interrupção externa no pino 6: INTCON <4>, o bit INTE

- mudança em quaisquer dos pinos 10-13: INTCON <3>, o bit RBIE

- estouro do cronômetro: INTCON <5>, o bit T0IE

- gravação completa da EEPROM: INTCON <6>, o bit EEIE

Terceiro, quando interrupções acontecem, certos bits (conhecido como flags de interrupção) são ativados de forma que podem então se determinar a origem da interrupção:

- interrupção externa no pino 6: INTCON <1>, o bit INTF

- mudança em quaisquer dos pinos 10-13: INTCON <0>, o bit RBIF

- estouro do cronômetro: INTCON <2>, o bit T0IF

Há a necessidade de se saber a origem dessa interrupção porque dependendo do seu tipo a ação que se deve entra deve ser de modo diferente. Para isso, deve-se verificar o bit apropriado que diz que tipo de interrupção é. Notar que as flags de interrupção sempre são ativadas quando há uma interrupção, independente do estado do bit de habilitação correspondente.

9.2- Lidando com uma Interrupção

No PIC16F84, todas as interrupções são enviadas para 004h. Esse ponto é o **vetor de interrupção** e o programa é orientado a dirigir-se para esse endereço. No vetor, deve-se ter a certeza irá ser suprido o código necessário para tratar com cuidado do problema. Um ponto crucial é que pode haver a necessidade de se salvar o status da máquina como ele estava antes da interrupção, antes de ser corrigido a interrupção. Claramente, as

atividades empreendidas durante a correção poderiam mudar algumas coisas tais como os registradores W e STATUS. É preciso ter-los restabelecidos após ter controlado a interrupção, para que nosso outro trabalho possa continuar. O PIC16F84 só salva o Contador do programa na pilha (PC+1, na verdade). É preciso se assegurar que estarão salva e que poderão ser recuperados qualquer outra coisa que eventualmente houver necessidade.

Há 3 partes a um programa que devem controlar interrupções simples:

- uma seção **initialize** onde, em nosso caso, nós habilitaremos as interrupções;

- uma seção **Main** onde a maior parte do é utilizada (fazendo cálculos ou o que quer que seja) e a parte de manipulador (**handler**) onde a interrupção é tratada.

Initialize: ativa GIE , INTCON <7> para habilitar interrupções, ativa INTE, INTCON <4> para habilitar interrupções no pino 6.

Handler: salva o estado da máquina; provavelmente W e STATUS; checa as flags de interrupção para determinar a origem; INTF, RBIF, T0IF derivam para o correto 'o sub-handler' para determinar o tipo de interrupção (embora nós fixamos apenas INTE). Faz o que é necessário para restaurar o estado anterior à interrupção e retorna.

Main: Efetua cálculos, por exemplo.

Exercício: Começar lentamente com as interrupções. Isto pode se tornar bastante complexo. Escrever um programa para provocar a menor interrupção possível: habilitar e fazer uma rotina de serviço de interrupção (ISR – interrupt service routine) que faça bem pouca, ou coisa alguma que não seja reabilitar as interrupções e retornar, e uma rotina principal que somente faça loops por algum código que espere por uma interrupção. Não será preciso preservar o estado da máquina, nem conferir o tipo de interrupção nesta fase. Para ver este programa funcionando corretamente no simulador, sugere-se usar a interrupção de mudança na porta B, usando a técnica de estímulo assíncrona discutida anteriormente. Usar um chaveamento do pino para provocar a mudança. Enquanto o programa estiver efetuando os loops na seção principal, chavear o pino e verificar que a interrupção acontece quando se executa o próximo passo do programa. Enquanto o programa estiver na ISR, chavear o pino novamente. Se tudo estiver funcionando, a mudança no pino não causará uma interrupção, porque GIE foi sido zerado.

Solução:

Program 5: Inter1.asm

```
;inter1.asm é um manipulador simples de interrupções-
;       ele não salva o estado da máquina
;       nem determina o tipo de interrupção.
;*****************************************simulador
;watch window: intcon, pcl
;stack: mantenha aberta também- veja o endereço de retorno ida e volta
;asynch stimulus: tenha um chaveamento em rb4
;*****************************************  setup
processor 16F84
movlw   h'0'
movwf   h'0b'   ;limpa intcon
```

```
goto    main    ;salta sobre a isr
;****************************************** início da isr
isr:    org     h'0004' ;interrupções sempre apontam para cá
nop             ;aqui nos lidamos de fato
nop             ;    com a interrupção
bcf     h'0b',0 ;limpa int antes de retornar
retfie          ;isto reativa o intcon gie
;********************************************** fim da isr

;********************************************** início de main
main    org     h'0020' ; deixa espaço bastante para a isr!
bsf     h'0b',7 ; ativa global int enable
bsf     h'0b',3 ; ativa mudança em b int enable
payrol  nop             ; loop aqui até interrução,
nop             ;   realize tarefas de payroll
goto    payrol
nop
nop
end
```

Exercício: Agora implementar esse programa para salvar o estado da máquina no momento da interrupção. Testar e ter a certeza que foi carregado W na parte principal, então alterar na ISR, e verificar se ele é restaurado corretamente.

Solução:

Program 6: Inter2.asm

```
;inter2.asm salva o estado da máquina
;       mas não determina o tipo de interrupção.
;
;*****************************************************simulador
;watch window: intcon, pcl, portb, w, w_saved
;stack: mantenha aberta também- veja o endereço de retorno ida e volta
;asynch stimulus: tenha um chaveamento em rb4
;*****************************************************  setup
processor 16F84
w_saved equ     h'10'   ;um lugar para manter w
movlw   h'0'
movwf   h'0b'   ;limpa intcon
goto    main    ;salta sobre a isr no início
;****************************************************** isr
isr     org     h'0004' ;interrupções sempre apontam para cá
movwf   w_saved         ;salva w como em main
movlw   h'65'           ;alguma coisa para mudar w
nop                     ;mais operaçõas da isr
;agora, restaurar w: não há
; "movfw"- 2 swapf's parece ser o caminho a seems to be
; seguir para faze-lo...
swapf   w_saved,1       ;primeiro, swap w_salvo nele próprio
swapf   w_saved,0       ;então, swap dentro de w
bcf     h'0b',0         ;limpa int em b antes de voltar
```

```
retfie                ;isto reativa intcon gie

;*************************************************** final da isr
;*************************************************** Início de main
main:   org     h'0020' ; deixe espaço para a isr!
bsf     h'0b',7 ; ativa global int enable
bsf     h'0b',3 ; ativa mudança em b int enable
payrol  nop             ; loop aqui até a interrupção,
nop             ;   fazendo instruções em payroll
movlw   h'0f'   ; simule um calculo carregando w
goto    payrol
nop
nop
end
```

Exercício: Finalmente, permitir mais de um tipo de interrupção - mudança em um dos pinos da porta B 4:7 bem como uma interrupção no pino RB0/INT. Neste caso será preciso determinar o tipo de interrupção, e controlar isso adequadamente.

Solução:

Program 7: Inter3.asm

```
;inter3.asm salva o estado da máquina
;           e determina o tipo de interrupção.
;
;*************************************************simulator
;watch window: intcon, pcl, portb, w_saved, w
;stack:mantenha aberta também- veja o endereço de retorno ida e volta
;asynch stimulus tenha um chaveamento em rb4 (for rbi) e rb0(int)
;************************************************  setup
processor 16F84
w_saved equ     h'10'           ;Um lugar para manter w
movlw   h'0'
movwf   h'0b'   ;limpa intcon
goto    main    ; salta sobre a isr no início
;*************************************************** isr
isr     org     h'0004'         ;interrupções sempre apontam para cá
movwf   w_saved         ;salva w como ele estava am Main
;descobre qual o tipo de interrupção
btfsc   h'0b',0         ;é uma rbi?
call    _rbi            ;sim - chame o 'sub-handler' rbi
btfsc   h'0b',1         ;é int?
call    _int            ;sim – chame o 'sub-handler' int
;termina aqui após os sub-handler
;agora, restaura w
swapf   w_saved,1       ;primeiro, swap em w_salvo dentro dele mesmo
swapf   w_saved,0       ;então, swap em w
bcf     h'0b',0         ;limpa rbif antes de voltar
bcf     h'0b',1         ;limpa intf antes de voltar
retfie                  ;isto reativa intcon gie
```

```
_rbi    movlw   h'65'         ;faz algo para mudar w
return
_int    movlw   h'24'         ;faz algo diferente com w
return
;*************************************************** Final da isr
;
;*************************************************** Início de main
main    org     h'0020' ; leave room for the isr!
bsf     h'0b',7 ; ativa global int enable
bsf     h'0b',3 ; ativa mudança em b int enable
bsf     h'0b',4 ; ativa ext int no pin6 enable
; nós tivemos 2 tipos de interrupção!
payrol  nop           ; loop aqui até interrupção,
nop           ;   fazendo operações em payroll
movlw   h'0f'   ;simule cálculo carregando w
goto    payrol
nop
nop
end
```

9.3- Timers no '84

A idéia básica é a utilização do PIC16F84 para cronometrar algo. Cronometrar, quer dizer do modo como é utilizado um relógio normal. Poderia acender a iluminação de um aquário, por exemplo, ou fechar as cortinas às 18h00 todas as noites. Mas não há nenhum relógio nos 84: nenhum relógio no sentido de um relógio normal. O que há, entretanto, é uma relação simples entre executar instruções e a velocidade em que o processador trabalha. Uma instrução é completa em 1µs quando o chip está operando à 4MHz.

Isso poderá ser conferido e ainda ser introduzido uma característica bem útil do MPLAB ao mesmo tempo: o Cronômetro. No MPLAB preparar para executar qualquer um dos programas que já foi escrito. Abrir o cronômetro através de Janela >Cronômetro. Poderá ser visualizado uma janela simples contendo 3 informações importantes: o número de passos completos em um determinado tempo, e a freqüência. Observar se a freqüência está fixada agora em 4MHz, clicar zero e executar um passo do programa uma vez. Deve ser visto que progrediu 1 ciclo em 1µs. (A menos que esse passo fosse, coincidentemente, um desvio no programa, o que é uma instrução de 2 ciclos.)

Assim, pode-se vislumbrar um relógio comum. É só se saber quantos passos foram executados para se saber quanto tem se passou. Entrar em Timer0.

Program 8: Time0.asm

```
;time0.asm para mostrar como timer0 funciona
;*********************************      setup
processor 16F84
STATUS        EQU     H'03'
OPTIO         EQU     H'81'
tmr0          EQU     H'01'
BANK_         EQU     H'05'
T0CS          EQU     H'05'
PSA           EQU     H'03'
```

```
clocked equ     h'10'
INTCON          equ     h'0B'
T0IE            equ     h'05'
GIE             EQU     H'07'
goto    top
;*********************************       isr
isr     ORG     h'0004'
incf    clocked ;clocked é o 'relógio comum'
BCF     INTCON,2
retfie
;*********************************       programa
top     org     h'0010'
clrf    tmr0            ;limpa o registrador tmr0
clrf    clocked
mode    bsf     STATUS,BANK_   ;alterna para page 1
bcf     OPTIO,T0CS      ;vai do counter para o timer
bsf     OPTIO,PSA       ;sem prescaler ainda
bsf     INTCON,GIE
bsf     INTCON,T0IE
loop    nop
nop
goto    loop
END
```

9.4- O Módulo TIMER0

O cronômetro do PIC16F84, TIMER0, trabalha numa base em que cada vez que acontece um ciclo de instrução, um certo intervalo de tempo decorreu. Quando TIMER0 está correndo, seu registrador (TMR0) é incrementado a cada ciclo; a cada 1µs se o clock for de 4MHz. Então, o valor de TMR0 representa o 'tempo' em passos de 1µs. Mas TMR0, como todos os registradores, é um registrador de 8 bits e assim só pode contar até FF: isso significa que o tempo vai até 255µS o que não é muito tempo. Projetar a sua mente até a discussão de interrupções, onde foi dito que um dos tipos de interrupção era uma interrupção de estouro de cronômetro: isso significa que o registrador TMR0 ultrapassou o limite de FF e retornou para 00. Isso significa que essa interrupção acontece a cada 256µS do tempo real. Pode-se aproveitar com esta interrupção incrementar um registrador.

Pode-se analisar passo a passo este processo observando simplesmente TIMER0 funcionando, sendo usado as técnicas de interrupção com initialize, handler e main, vistas anteriormente.

Initialize: ativa GIE, INTCON<7> para habilitar interrupções, ativa T0IE, INTCON<5> para habilitar a interrupção do timer.

limpa T0CS, OPTION<5> para habilitar o timer - veja ativar PSA, OPTIOPN<3>, para manter o prescaler fora disto.

Handler: incremento 'cronometrado em função do clock', nosso contador de overflow (estouro).

Main: tem um loop para manter o programa rodando.

Exercício: Escrever um programa para implementar o cronômetro simples acima, tendo em mente que nós não se está fazendo nada com os estouros a não ser clicá-los. Conferir o programa no simulador, observando os registradores que devem ser monitorados e também o cronômetro. Antes de começar, será preciso entender como acessar o registrador de opção (OPTION), o que não é exatamente tão simples quanto acessar o STATUS.

Acessando registradores em Bank1: Pode se perceber que alguns registradores como OPTION estão naquilo que é referido como Bank1, ao invés de Bank0. Outros como STATUS está em ambos Bank0 e Bank1. O que isto significa na prática, é que estes registradores do Bank1 normalmente não estão disponíveis, porque nossa esfera de operações é o Bank0. Foi utilizado o registrador de STATUS para alternar entre os bancos, com RP0 (STATUS <5>) fazendo o truque. Isto simplesmente significa que há a necessidade de ativar RP0 para ir para Bank1 e zerá-lo para retornar. Para aqueles registradores que estão em ambos os bancos, tem-se o acesso não importa de onde se esteja, mas lembrar que o registrador tem 2 endereços diferentes: STATUS tem seus endereços conhecidos em h'03' e h'83'.

9.5- Usando o estouro do timer

Pode-se observar que o registrador o registrador fica se atualizando sempre que o cronômetro estoura e isto acontece a cada 256µs, ou seja o timer vai estourar em 256. Existe a necessidade também em contar estes estouros em outro registrador (arquivo). Agora, se está chegando mais perto de um segundo na cronometragem, que é o objetivo que a ser perseguido. Até quanto é preciso contar no segundo arquivo para alcançar 1 segundo?

Incrementar a cada .0,0256 segundos significa que existem aproximadamente 39 incrementos em um Segundo, então isso significa que podem ser contados segundos de tempo-real incrementando um outro registrador, SECONDS por exemplo, a cada 39 passos. Para conseguir minutos e horas, simplesmente é tratado com o estouro do registrador SECONDS para minutos a cada 60 segundos e o mesmo com um registrador de minutos, a cada 60 minutos para marcar as horas.

Exercício: Criar um programa para implementar o anterior, pelo menos até a parte do registrador SECONDS.
Usando a memória de dados EEPROM

Apêndice 1 – Conjunto de Instruções

• Instruções Aritméticas	• Instruções de Movimentação de Dados	• Instruções Lógicas
ADDLW ADDWF SUBLW SUBWF INCF DECF CLRF CLRW	MOVLW MOVWF MOVF RLF RRF	ANDLW ANDWF IORLW IORWF XORLW XORWF COMF SWAPF
• Instruções de atendimento a Rotinas	**• Instruções de controle de Bit e Teste de variáveis**	**• Instruções de controle ao Processamento**
GOTO CALL RETURN RETLW RETFIE	BCF BSF BTFSS BTFSC INCFSZ DECFSZ	NOP CLRWDT SLEEP

MOVLW Escrever constante no registro W

Sintaxe:	*[rótulo]* MOVLW **k**
Descrição:	A constante de 8-bits **k** vai para o registro **W**.
Operação:	$\mathbf{k} \Rightarrow (\mathbf{W})$
Operando:	$0 \leq \mathbf{k} \leq 255$
Flag:	-
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	MOVLW 0x5A
	Depois da instrução: W= 0x5A
Exemplo 2:	MOVLW REGISTRAR
	Antes da instrução: W = 0x10 e REGISTRAR = 0x40
	Depois da instrução: W = 0x40

MOVWF Copiar W para f

Sintaxe:	*[rótulo]* MOVWF **f**
Descrição:	O conteúdo do registro **W** é copiado para o registro **f**
Operação:	$\mathbf{W} \Rightarrow (\mathbf{f})$
Operando:	$0 \leq \mathbf{f} \leq 127$
Flag:	-

Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	MOVWF OPTION_REG
	Antes da instrução: OPTION_REG = 0x20
	W = 0x40
	Depois da instrução: OPTION_REG = 0x40
	W = 0x40
Exemplo 2:	MOVWF INDF
	Antes da instrução: W = 0x17
	FSR = 0xC2
	Conteúdo do endereço 0xC2 = 0x00
	Depois da instrução: W = 0x17
	FSR = 0xC2
	Conteúdo do endereço 0xC2 = 0x17

MOVF Copiar f para d

Sintaxe:	*[rótulo]* MOVF **f, d**
Descrição:	O conteúdo do registo **f** é guardado no local determinado pelo operando **d**
	Se **d = 0**, o destino é o registro **W**
	Se **d = 1**, o destino é o próprio registro **f**
	A opção **d = 1**, é usada para testar o conteúdo do registro **f**, porque a execução desta instrução afeta a flag Z do registro STATUS.
Operação:	$\mathbf{f} \Rightarrow (\mathbf{d})$
Operando:	$0 \leq \mathbf{f} \leq 127, \mathrm{d} \in [0, 1]$
Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	MOVF FSR, 0
	Antes da instrução: FSR = 0xC2
	W = 0x00
	Depois da instrução: W = 0xC2
	Z = 0
Exemplo 2:	MOVF INDF, 0
	Antes da instrução: W = 0x17
	FSR = 0xC2
	conteúdo do endereço 0xC2 = 0x00

	Depois da instrução: W = 0x00
	FSR = 0xC2
	conteúdo do endereço 0xC2 = 0x00
	Z = 1

CLRW Escrever 0 em W

Sintaxe:	*[rótulo]* CLRW
Descrição:	O conteúdo do registo **W** passa para 0 e a flag Z do registro STATUS toma o valor 1.
Operação:	$0 \Rightarrow (\mathbf{W})$
Operando:	-
Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo:	CLRW
	Antes da instrução: W = 0x55
	Depois da instrução: W = 0x00
	Z = 1

CLRF Escrever 0 em f

Sintaxe:	*[rótulo]* CLRF **f**
Descrição:	O conteúdo do registro **'f'** passa para 0 e a flag Z do registro STATUS toma o valor 1.
Operação:	$0 \Rightarrow \mathrm{f}$
Operando:	$0 \leq \mathbf{f} \leq 127$
Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	CLRF STATUS
	Antes da instrução: STATUS = 0xC2
	Depois da instrução: STATUS = 0x00
	Z = 1
Exemplo 2:	CLRF INDF
	Antes da instrução: FSR = 0xC2
	conteúdo do endereço 0xC2 = 0x33
	Depois da instrução: FSR = 0xC2

	conteúdo do endereço 0xC2 = 0x00
	Z = 1

SWAPF Copiar o conteúdo de f para d, trocando a posição dos 4 primeiros bits com a dos 4 últimos

Sintaxe:	*[rótulo]* SWAPF **f, d**
Descrição:	Os 4 bits + significativos e os 4 bits – significativos de f, trocam de posições.
	Se **d = 0**, o resultado é guardado no registro **W**
	Se **d = 1**, o resultado é guardado no registro **f**
Operação:	$\mathbf{f}\langle 0{:}3\rangle \Rightarrow \mathbf{d}\langle 4{:}7\rangle, \mathbf{f}\langle 4{:}7\rangle \Rightarrow \mathbf{d}\langle 0{:}3\rangle,$
Operando:	$0 \le \mathbf{f} \le 127, d \in [0, 1]$
Flag:	-
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	SWAPF REG, 0
	Antes da instrução: REG = 0xF3
	Depois da instrução: REG = 0xF3
	W = 0x3F
Exemplo 2:	SWAPF REG, 1
	Antes da instrução: REG = 0xF3
	Depois da instrução: REG = 0x3F

ADDLW Adicionar W a uma constante

Sintaxe:	*[rótulo]* ADDLW **k**
Descrição:	O conteúdo do registro **W**, é adicionado à constante de 8-bits **k** e o resultado é guardado no registro **W**.
Operação:	$(\mathbf{W}) + \mathbf{k} \Rightarrow \mathbf{W}$
Operando:	$0 \le \mathbf{k} \le 255$
Flag:	**C, DC, Z**
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	ADDLW 0x15
	Antes da instrução: W= 0x10

	Depois da instrução: W= 0x25
Exemplo 2:	ADDLW REG
	Antes da instrução: W = 0x10
	REG = 0x37
	Depois da instrução: W = 0x47

ADDWF Adicionar W a f

Sintaxe:	*[rótulo]* ADDWF **f**, **d**
Descrição:	Adicionar os conteúdos dos registros **W** e **f**
	Se **d=0**, o resultado é guardado no registro **W**
	Se **d=1**, o resultado é guardado no registro **f**
Operação:	$(\mathbf{W}) + (\mathbf{f}) \Rightarrow \mathbf{d}, \mathbf{d} \in [0, 1]$
Operando:	$0 \leq \mathbf{f} \leq 127$
Flag:	**C, DC, Z**
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	ADDWF FSR, 0
	Antes da instrução: W = 0x17
	FSR = 0xC2
	Depois da instrução: W = 0xD9
	FSR = 0xC2
Exemplo 2:	ADDWF INDF,0
	Antes da instrução: W = 0x17
	FSR = 0xC2
	conteúdo do endereço 0xC2 = 0x20
	Depois da instrução: W = 0x37
	FSR = 0xC2
	Conteúdo do endereço 0xC2 = 0x20

SUBLW Subtrair W a uma constante

Sintaxe:	*[rótulo]* SUBLW **k**
Descrição:	O conteúdo do registro **W**, é subtraído à constante **k** e, o resultado, é guardado no registro **W**.
Operação:	$\mathbf{k} - (\mathbf{W}) \Rightarrow \mathbf{W}$
Operando:	$0 \leq \mathbf{k} \leq 255$
Flag:	**C, DC, Z**
Número de	1

palavras:	
Número de ciclos:	1
Exemplo 1:	SUBLW 0x03
	Antes da instrução: W= 0x01, C = x, Z = x
	Depois da instrução: W= 0x02, C = 1, Z = 0 Resultado > 0
	Antes da instrução: W= 0x03, C = x, Z = x
	Depois da instrução: W= 0x00, C = 1, Z = 1 Resultado = 0
	Antes da instrução: W= 0x04, C = x, Z = x
	Depois da instrução: W= 0xFF, C = 0, Z = 0 Resultado < 0
Exemplo 2:	SUBLW REG
	Antes da instrução: W = 0x10
	REG = 0x37
	Depois da instrução: W = 0x27
	C = 1 Resultado > 0

SUBWF Subtrair W a f

Sintaxe:	*[rótulo]* SUBWF **f**, **d**
Descrição:	O conteúdo do registro **W** é subtraído ao conteúdo do registro **f**
	Se **d=0**, o resultado é guardado no registro **W**
	Se **d=1**, o resultado é guardado no registro **f**
Operação:	$(\mathbf{f}) - (\mathbf{W}) \Rightarrow \mathbf{d}$
Operando:	$0 \leq \mathbf{f} \leq 127, \mathbf{d} \in [0, 1]$
Flag:	**C, DC, Z**
Número de palavras:	1
Número de ciclos:	1
Exemplo:	SUBWF REG, 1
	Antes da instrução: REG= 3, W= 2, C = x, Z = x
	Depois da instrução: REG= 1, W= 2, C = 1, Z = 0 Resultado > 0
	Antes da instrução: REG= 2, W= 2, C = x, Z = x
	Depois da instrução: REG=0, W= 2, C = 1, Z = 1 Resultado = 0
	Antes da instrução: REG=1, W= 2, C = x, Z = x
	Depois da instrução: REG= 0xFF, W=2, C = 0, Z = 0 Resultado < 0

ANDLW Fazer o "E" lógico de W com uma constante

Sintaxe:	*[rótulo]* ANDLW **k**

Descrição:	É executado o "E" lógico do conteúdo do registro **W**, com a constante **k**
	O resultado é guardado no registro **W**.
Operação:	$(\mathbf{W})$ .AND. $\mathbf{k} \Rightarrow \mathbf{W}$
Operando:	$0 \leq \mathbf{k} \leq 255$
Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	ANDLW 0x5F
	Antes da instrução: W= 0xA3 ; 0101 1111 (0x5F)
	; 1010 0011 (0xA3)
	Depois da instrução: W= 0x03; 0000 0011 (0x03)
Exemplo 2:	ANDLW REG
	Antes da instrução: W = 0xA3 ; 1010 0011 (0xA3)
	REG = 0x37 ; 0011 0111 (0x37)
	Depois da instrução: W = 0x23 ; 0010 0011 (0x23)

ANDWF Fazer o "E" lógico de W com f

Sintaxe:	*[rótulo]* ANDWF **f**, **d**
Descrição:	Faz o "E" lógico dos conteúdos dos registros **W** e **f**
	Se **d=0**, o resultado é guardado no registro **W**
	Se **d=1**, o resultado é guardado no registro **f**
Operação:	$(\mathbf{W})$ .AND. $(\mathbf{f}) \Rightarrow \mathbf{d}$
Operando:	$0 \leq \mathbf{f} \leq 127$, $\mathbf{d} \in [0, 1]$
Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	ANDWF FSR, 1
	Antes da instrução: W= 0x17, FSR= 0xC2; 0001 1111 (0x17)
	; 1100 0010 (0xC2)
	Depois da instrução: W= 0x17, FSR= 0x02 ; 0000 0010 (0x02)
Exemplo 2:	ANDWF FSR, 0

	Antes da instrução: W= 0x17, FSR= 0xC2; 0001 1111 (0x17)
	; 1100 0010 (0xC2)
	Depois da instrução: W= 0x02, FSR= 0xC2; 0000 0010 (0x02)

IORLW Fazer o "OU" lógico de W com uma constante

Sintaxe:	*[rótulo]* IORLW **k**
Descrição:	É executado o "OU" lógico do conteúdo do registro **W**, com a constante de 8 bits **k**, o resultado é guardado no registro **W**.
Operação:	$(\mathbf{W})\ .\mathrm{OR.}\ \mathbf{k} \Rightarrow \mathbf{W}$
Operando:	$0 \le \mathbf{k} \le 255$
Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	IORLW 0x35
	Antes da instrução: W= 0x9A
	Depois da instrução: W= 0xBF
	Z= 0
Exemplo 2:	IORLW REG
	Antes da instrução: W = 0x9A
	conteúdo de REG = 0x37
	Depois da instrução: W = 0x9F
	Z = 0

IORWF Fazer o "OU" lógico de W com f

Sintaxe:	*[rótulo]* IORWF **f**, **d**
Descrição:	Faz o "OU" lógico dos conteúdos dos registos **W** e **f**
	Se **d=0**, o resultado é guardado no registro **W**
	Se **d=1**, o resultado é guardado no registro **f**
Operação:	$(\mathbf{W})\ .\mathrm{OR.}\ (\mathbf{f}) \Rightarrow \mathbf{d}$
Operando:	$0 \le \mathbf{f} \le 127, \mathbf{d} \in [0, 1]$
Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	IORWF REG, 0
	Antes da instrução: REG= 0x13, W= 0x91
	Depois da instrução: REG= 0x13, W= 0x93

	Z= 0
Exemplo 2:	IORWF REG, 1
	Antes da instrução: REG= 0x13, W= 0x91
	Depois da instrução: REG= 0x93, W= 0x91
	Z= 0

XORLW "OU- EXCLUSIVO" de W com uma constante

Sintaxe:	*[rótulo]* XORLW **k**
Descrição:	É executada a operação "OU-Exclusivo" do conteúdo do registro **W**, com a constante **k.** O resultado é guardado no registro **W.**
Operação:	$(\mathbf{W})\ .\mathrm{XOR}.\ \mathbf{k} \Rightarrow \mathbf{W}$
Operando:	$0 \le \mathbf{k} \le 255$
Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	XORLW 0xAF
	Antes da instrução: W= 0xB5 ; 1010 1111 (0xAF)
	; 1011 0101 (0xB5)
	Depois da instrução: W= 0x1A; 0001 1010 (0x1A)
Exemplo 2:	XORLW REG
	Antes da instrução: W = 0xAF ; 1010 1111 (0xAF)
	REG = 0x37 ; 0011 0111 (0x37)
	Depois da instrução: W = 0x98 ; 1001 1000 (0x98)
	Z = 0

XORWF "OU-EXCLUSIVO" de W com f

Sintaxe:	*[rótulo]* XORWF **f**, **d**
Descrição:	Faz o "OU-EXCLUSIVO" dos conteúdos dos registros **W** e **f**
	Se **d=0**, o resultado é guardado no registro **W**
	Se **d=1**, o resultado é guardado no registro **f**
Operação:	$(\mathbf{W})\ .\mathrm{XOR}.\ (\mathbf{f}) \Rightarrow \mathbf{d}$
Operando:	$0 \le \mathbf{f} \le 127,\ \mathbf{d} \in [0, 1]$
Flag:	Z
Número de palavras:	1

Número de ciclos:	1
Exemplo 1:	XORWF REG, 1
	Antes da instrução: REG= 0xAF, W= 0xB5 ; 1010 1111 (0xAF)
	; 1011 0101 (0xB5)
	Depois da instrução: REG= 0x1A, W= 0xB5 001 1010 (0x1A)
Exemplo 2:	XORWF REG, 0
	Antes da instrução: REG= 0xAF, W= 0xB5; 1010 1111 (0xAF)
	; 1011 0101 (0xB5)
	Depois da instrução: REG= 0xAF, W= 0x1A ; 0001 1010 (0x1A)

INCF Incrementar f

Sintaxe:	*[rótulo]* INCF **f**, **d**
Descrição:	Incrementar de uma unidade, o conteúdo do registro f.
	Se **d=0**, o resultado é guardado no registro **W**
	Se **d=1**, o resultado é guardado no registro **f**
Operação:	$(\mathbf{f}) + 1 \Rightarrow \mathbf{d}$
Operando:	$0 \leq \mathbf{f} \leq 127, \mathbf{d} \in [0, 1]$
Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	INCF REG, 1
	Antes da instrução: REG = 0xFF
	Z = 0
	Depois da instrução: REG = 0x00
	Z = 1
Exemplo 2:	INCF REG, 0
	Antes da instrução: REG = 0x10
	W = x
	Z = 0
	Depois da instrução: REG = 0x10
	W = 0x11
	Z = 0

DECF Decrementar f

Sintaxe:	*[rótulo]* DECF **f**, **d**

Descrição:	Decrementar de uma unidade, o conteúdo do registro f.
	Se **d=0**, o resultado é guardado no registro **W**
	Se **d=1**, o resultado é guardado no registro **f**
Operação:	$(\mathbf{f}) - 1 \Rightarrow \mathbf{d}$
Operando:	$0 \leq \mathbf{f} \leq 127, \mathbf{d} \in [0, 1]$
Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	DECF REG, 1
	Antes da instrução: REG = 0x01
	Z = 0
	Depois da instrução: REG = 0x00
	Z = 1
Exemplo 2:	DECF REG, 0
	Antes da instrução: REG = 0x13
	W = x
	Z = 0
	Depois da instrução: REG = 0x13
	W = 0x12
	Z = 0

RLF Rodar f para a esquerda através do Carry

Sintaxe:	*[rótulo]* RLF **f**, **d**	
Descrição:	O conteúdo do registro **f** é rodado um espaço para a esquerda, através de C (flag do Carry).	
	Se **d=0**, o resultado é guardado no registro **W**	
	Se **d=1**, o resultado é guardado no registro **f**	
Operação:	$(\mathbf{f}\langle n\rangle) \Rightarrow \mathbf{d}\langle n+1\rangle, f\langle 7\rangle \Rightarrow C, C \Rightarrow \mathbf{d}\langle 0\rangle;$	
Operando:	$0 \leq \mathbf{f} \leq 127, \mathbf{d} \in [0, 1]$	
Flag:	C	
Número de palavras:	1	
Número de ciclos:	1	
Exemplo 1:	RLF REG, 0	
	Antes da instrução: REG = 1110 0110	
	C = 0	
	Depois da instrução: REG = 1110 0110	

	W = 1100 1100
	C = 1
Exemplo 2:	RLF REG, 1
	Antes da instrução: REG = 1110 0110
	C = 0
	Depois da instrução: REG = 1100 1100
	C = 1

RRF Rodar f para a direita através do Carry

Sintaxe:	*[rótulo]* RRF **f**, **d**	
Descrição:	O conteúdo do registro **f** é rodado um espaço para a direita, através de C (flag do Carry).	
	Se **d=0**, o resultado é guardado no registro **W**	
	Se **d=1**, o resultado é guardado no registro **f**	
Operação:	(**f** <n>) $\Rightarrow$ **d**<n-1>, f<0> $\Rightarrow$ C, C $\Rightarrow$ **d**<7>;	
Operando:	$0 \le \mathbf{f} \le 127$, $\mathbf{d} \in [0, 1]$	
Flag:	C	
Número de palavras:	1	
Número de ciclos:	1	
Exemplo 1:	RRF REG, 0	
	Antes da instrução: REG = 1110 0110	
	W = x	
	C = 0	
	Depois da instrução: REG = 1110 0110	
	W = 0111 0011	
	C = 0	
Exemplo 2:	RRF REG, 1	
	Antes da instrução: REG = 1110 0110	
	C = 0	
	Depois da instrução: REG = 0111 0011	
	C = 0	

COMF Complementar f

Sintaxe:	*[rótulo]* COMF **f**, **d**
Descrição:	O conteúdo do registro **f** é complementado.
	Se **d=0**, o resultado é guardado no registro **W**
	Se **d=1**, o resultado é guardado no registro **f**
Operação:	() $\Rightarrow$ d
Operando:	$0 \le \mathbf{f} \le 127$, $\mathbf{d} \in [0, 1]$

Flag:	Z
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	COMF REG, 0
	Antes da instrução: REG= 0x13 ; 0001 0011 (0x13)
	Depois da instrução: REG= 0x13 ; complementar
	W = 0xEC ; 1110 1100 (0xEC)
Exemplo 2:	COMF INDF, 1
	Antes da instrução: FSR= 0xC2
	conteúdo de FSR = (FSR) = 0xAA
	Depois da instrução: FSR= 0xC2
	conteúdo de FSR = (FSR) = 0x55

BCF Pôr a "0" o bit b de f

Sintaxe:	*[rótulo]* BCF **f**, **b**
Descrição:	Limpar (pôr a '0'), o bit **b** do registro **f**
Operação:	$0 \Rightarrow$ f<b>
Operando:	$0 \leq \mathbf{f} \leq 127, 0 \leq \mathbf{b} \leq 7$
Flag:	-
Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	BCF REG, 7
	Antes da instrução: REG = 0xC7 ; 1100 0111 (0xC7)
	Depois da instrução: REG = 0x47 ; 0100 0111 (0x47)
Exemplo 2:	BCF INDF, 3
	Antes da instrução: W = 0x17
	FSR = 0xC2
	conteúdo do endereço em FSR (FSR) = 0x2F
	Depois da instrução: W = 0x17
	FSR = 0xC2
	conteúdo do endereço em FSR (FSR) = 0x27

BSF Pôr a "1" o bit b de f

Sintaxe:	*[rótulo]* BSF **f**, **b**
Descrição:	Pôr a '1', o bit **b** do registro **f**
Operação:	$1 \Rightarrow$ f<b>
Operando:	$0 \leq \mathbf{f} \leq 127, 0 \leq \mathbf{b} \leq 7$
Flag:	-

Número de palavras:	1
Número de ciclos:	1
Exemplo 1:	BSF REG, 7
	Antes da instrução: REG = 0x07 ; 0000 0111 (0x07)
	Depois da instrução: REG = 0x17 ; 1000 0111 (0x87)
Exemplo 2:	BSF INDF, 3
	Antes da instrução: W = 0x17
	FSR = 0xC2
	conteúdo do endereço em FSR (FSR) = 0x2F
	Depois da instrução: W = 0x17
	FSR = 0xC2
	conteúdo do endereço em FSR (FSR) = 0x28

BTFSC Testar o bit b de f, saltar por cima se for = 0

Sintaxe:	*[rótulo]* BTFSC **f**, **b**
Descrição:	Se o bit **b** do registro **f** for igual a zero, ignorar instrução seguinte. Se este bit **b** for zero, então, durante a execução da instrução atual, a execução da instrução seguinte não se concretiza e é executada, em vez desta, uma instrução NOP, fazendo com que a instrução atual, demore dois ciclos de instrução a ser executada.
Operação:	Ignorar a instrução seguinte se (**f**<**b**>) = 0
Operando:	$0 \leq \mathbf{f} \leq 127, 0 \leq \mathbf{b} \leq 7$
Flag:	-
Número de palavras:	1
Número de ciclos:	1 ou 2 dependendo do valor lógico do bit **b**
Exemplo:	LAB_01 BTFSC REG, 1; Testar o bit 1 do registo REG
	LAB_02 ..... ; Ignorar esta linha se for 0
	LAB_03 ..... ; Executar esta linha depois da anterior, se for 1
	Antes da instrução, o contador de programa contém o endereço LAB_01. Depois desta instrução, se o bit 1 do registo REG for zero, o contador de programa contém o endereço LAB_03. Se o bit 1 do registo REG for 'um', o contador de programa contém o endereço LAB_02.

BTFSS Testar o bit b de f, saltar por cima se for = 1

Sintaxe:	*[rótulo]* BTFSS **f**, **b**
Descrição:	Se o bit **b** do registro **f** for igual a um, ignorar instrução seguinte. Se durante a execução desta instrução este bit **b** for um, então, a execução da instrução seguinte não se concretiza e é executada.

	em vez desta, uma instrução NOP, assim, a instrução atual demora dois ciclos de instrução a ser executada.
Operação:	Ignorar a instrução seguinte se (**f**<**b**>) = 1
Operando:	$0 \le \mathbf{f} \le 127, 0 \le \mathbf{b} \le 7$
Flag:	-
Número de palavras:	1
Número de ciclos:	1 ou 2 dependendo do valor lógico do bit **b**
Exemplo:	LAB_01 BTFSS REG, 1; Testar o bit 1 do registro REG
	LAB_02 ..... ; Ignorar esta linha se for 1
	LAB_03 ..... ; Executar esta linha depois da anterior, se for 0
	Antes da instrução, o contador de programa contém o endereço LAB_01. Depois desta instrução, se o bit 1 do registro REG for 'um', o contador de programa contém o endereço LAB_03. Se o bit 1 do registro REG for zero, o contador de programa contém o endereço LAB_02.

INCFSZ Incrementar f, saltar por cima se der = 0

Sintaxe:	*[rótulo]* INCFSZ **f**, **d**
Descrição:	**Descrição:** O conteúdo do registro **f** é incrementado de uma unidade.
	Se **d = 0**, o resultado é guardado no registro **W**.
	Se **d = 1**, o resultado é guardado no registro **f**.
	Se o resultado do incremento for = 0, a instrução seguinte é substituída por uma instrução NOP, fazendo com que a instrução atual, demore dois ciclos de instrução a ser executada.
Operação:	$(\mathbf{f}) + 1 \Rightarrow \mathbf{d}$
Operando:	$0 \le \mathbf{f} \le 127, \mathbf{d} \in [0, 1]$
Flag:	-
Número de palavras:	1
Número de ciclos:	1 ou 2 dependendo do resultado
Exemplo:	LAB_01 INCFSZ REG, 1; Incrementar o conteúdo de REG de uma unidade
	LAB_02 ..... ; Ignorar esta linha se resultado = 0
	LAB_03 ..... ; Executar esta linha depois da anterior, se der 0

	Conteúdo do contador de programa antes da instrução, PC = endereço LAB_01. Se o conteúdo do registro REG depois de a operação REG = REG + 1 ter sido executada, for REG = 0, o contador de programa aponta para o rótulo de endereço LAB_03. Caso contrário, o contador de programa contém o endereço da instrução seguinte, ou seja, LAB_02.

DECFSZ Decrementar f, saltar por cima se der = 0

Sintaxe:	*[rótulo]* DECFSZ **f**, **d**
Descrição:	O conteúdo do registro **f** é decrementado uma unidade.
	Se **d = 0**, o resultado é guardado no registro **W**.
	Se **d = 1**, o resultado é guardado no registro **f**.
	Se o resultado do decremento for = 0, a instrução seguinte é substituída por uma instrução NOP, fazendo assim com que a instrução atual, demore dois ciclos de instrução a ser executada.
Operação:	$(\mathbf{f}) - 1 \Rightarrow \mathbf{d}$
Operando:	$0 \leq \mathbf{f} \leq 127$, $\mathbf{d} \in [0, 1]$
Flag:	-
Número de palavras:	1
Número de ciclos:	1 ou 2 dependendo do resultado
Exemplo:	LAB_01 DECFSZ REG, 1; Decrementar o conteúdo de REG de uma unidade
	LAB_02 ..... ; Ignorar esta linha se resultado = 0
	LAB_03 ..... ; Executar esta linha depois da anterior, se der 0
	Conteúdo do contador de programa antes da instrução, PC = endereço LAB_01. Se o conteúdo do registro REG depois de a operação REG = REG – 1 ter sido executada, for REG = 0, o contador de programa aponta para o rótulo de endereço LAB_03. Caso contrário, o contador de programa contém o endereço da instrução seguinte, ou seja, LAB_02.

GOTO Saltar para o endereço

Sintaxe:	*[rótulo]* GOTO **k**
Descrição:	Salto incondicional para o endereço **k**.
Operação:	$\mathbf{k} \Rightarrow$ PC<10:0>, (PCLATH<4:3>) $\Rightarrow$ PC<12:11>
Operando:	$0 \leq \mathbf{k} \leq 2048$
Flag:	-
Número de palavras:	1
Número de ciclos:	2
Exemplo:	LAB_00 GOTO LAB_01; Saltar para LAB_01

	:
	LAB_01 .....
	Antes da instrução: PC = endereço LAB_00
	Depois da instrução: PC = endereço LAB_01

CALL Chamar um programa

Sintaxe:	*[rótulo]* CALL **k**
Descrição:	Esta instrução, chama um subprograma. Primeiro, o endereço de retorno (PC+1) é guardado na pilha, a seguir, o operando **k** de 11 bits, correspondente ao endereço de início do subprograma, vai para o contador de programa (PC).
Operação:	PC+1 $\Rightarrow$ Topo da pilha (TOS – Top Of Stack)
Operando:	$0 \leq \mathbf{k} \leq 2048$
Flag:	-
Número de palavras:	1
Número de ciclos:	2
Exemplo:	LAB_00 CALL LAB_02 ; Chamar a subrotina LAB_02
	LAB_01 :
	:
	LAB_02 .....
	Antes da instrução: PC = endereço LAB_00
	TOS = x
	Depois da instrução: PC = endereço LAB_02
	TOS = LAB_01

RETURN Retorno de um subprograma

Sintaxe:	*[rótulo]* RETURN
Descrição:	O conteúdo do topo da pilha é guardado no contador de programa.
Operação:	TOS $\Rightarrow$ Contador de programa PC
Operando:	-
Flag:	-
Número de palavras:	1
Número de ciclos:	2
Exemplo:	RETURN
	Antes da instrução: PC = x
	TOS = x
	Depois da instrução: PC = TOS
	TOS = TOS - 1

RETLW Retorno de um subprograma com uma constante em W

Sintaxe:	*[rótulo]* RETLW **k**
Descrição:	A constante **k** de 8 bits, é guardada no registro **W**.
Operação:	$(\mathbf{k}) \Rightarrow \mathbf{W}$; TOS $\Rightarrow$ PC
Operando:	$0 \leq k \leq 255$
Flag:	-
Número de palavras:	1
Número de ciclos:	2
Exemplo:	RETLW 0x43
	Antes da instrução: W = x
	PC = x
	TOS = x
	Depois da instrução: W = 0x43
	PC = TOS
	TOS = TOS – 1

RETFIE Retorno de uma rotina de interrupção

Sintaxe:	*[rótulo]* RETLW **k**
Descrição:	Retorno de uma subrotina de atendimento de interrupção. O conteúdo do topo de pilha (TOS), é transferido para o contador de programa (PC). Ao mesmo tempo, as interrupções são habilitadas, pois o bit GIE de habilitação global das interrupções, é posto a '1'.
Operação:	TOS $\Rightarrow$ PC ; 1$\Rightarrow$ GIE
Operando:	-
Flag:	-
Número de palavras:	1
Número de ciclos:	2
Exemplo:	RETFIE
	Antes da instrução: PC = x
	GIE = 0
	Depois da instrução: PC = TOS
	GIE = 1

NOP Nenhuma operação

Sintaxe:	*[rótulo]* NOP
Descrição:	Nenhuma operação é executada, nem qualquer flag é afectada.
Operação:	-
Operando:	-
Flag:	-

Número de palavras:	1
Número de ciclos:	1
Exemplo:	NOP

CLRWDT Iniciar o temporizador do watchdog

Sintaxe:	*[rótulo]* CLRWDT
Descrição:	O temporizador do watchdog é reposto a zero. O prescaler do temporizador de Watchdog é também reposto a 0 e, também, os bits do registro de estado e são postos a 'um'.
Operação:	$0 \Rightarrow$ WDT
	$0 \Rightarrow$ prescaler de WDT
	$1 \Rightarrow$
	$1 \Rightarrow$
Operando:	-
Flag:	
Número de palavras:	1
Número de ciclos:	1
Exemplo:	CLRWDT
	Antes da instrução: Contador de WDT = x
	Prescaler de WDT = 1:128
	Depois da instrução: Contador do WDT = 0x00
	Prescale do WDT = 0

SLEEP Modo de repouso

Sintaxe:	*[rótulo]* SLEEP
Descrição:	O processador entra no modo de baixo consumo. O oscilador pára. O bit (Power Down) do registro Status é reposto a '0'. O bit (Timer Out) é posto a '1'. O temporizador de WDT (Watchdog) e o respectivo prescaler são repostos a '0'.
Operação:	$0 \Rightarrow$ WDT
	$0 \Rightarrow$ prescaler do WDT
	$1 \Rightarrow TO$
	$0 \Rightarrow PD$
Operando:	-
Flag:	
Número de palavras:	1

Número de ciclos:	1
Exemplo 1:	SLEEP
	Antes da instrução: Contador do WDT = x
	Prescaler do WDT = x
	Depois da instrução: Contador do WDT = 0x00
	Prescaler do WDT = 0

Apêndice 2 – Biblioteca de Rotinas