UNIVERSIDADE FEDERAL DA BAHIA

ESCOLA POLITÉCNICA

LUCAS MENEZES PEREIRA

MATHEUS OLIVER DE CARVALHO CERQUEIRA

RELATÓRIO DE DESCRIÇÃO DA ARQUITETURA PIC16 COM EXEMPLO DE FUNCIONAMENTO

Salvador

2018

1. APRESENTAÇÃO

A linguagem *Assembly* é muito utilizada, tanto para *Software*, quanto para *Hardware*. É a responsável por interfacear os comandos expressos pelo programador para linguagem de máquina, e, consequentemente, configuração dos circuitos internos do dispositivo. Nesse contexto, cada fabricante utiliza essa linguagem em um dispositivo específico, para determinado fim, muitas vezes criando a própria arquitetura (conjunto de atributos que, organizados de tal forma possibilitam à sintaxe da linguagem interagir sobre os circuitos), como um protecionismo de propriedade industrial.

Dentre os dispositivos eletrônicos mais usados, estão os Microprocessadores e os Microcontroladores. Os primeiros contêm apenas uma *CPU* (do inglês, Unidade Central de Processamento) interna, servindo como computadores de propósito geral. Enquanto isso, os últimos, além de CPU, contêm Memórias, barramentos e periféricos (dispositivos de entrada/saída). Uma Unidade Central de Processamento, por sua vez, é composta de uma ALU (do inglês, Unidade Lógico-Aritmética), registradores, conexões e controlador de dados. Os microcontroladores, portanto, são mais utilizados em tarefas específicas, geralmente relacionadas ao gerenciamento de entrada e saída.

O microcontrolador *PIC* (do inglês, Controlador de Interface Programável), fabricado atualmente pela *Microchip Technology Inc.*, é um dos mais conhecidos, tendo diversas aplicações, com uma comunidade de usuários e desenvolvedores ativa. Junto com o 8051 e a Placa Arduino, é um dos Microcontroladores mais ensinados e porta de entrada ao desenvolvimento de projetos em dispositivos embarcados.

2. OBJETIVOS

2.1 Geral

Apresentar e analisar o Microcontrolador PIC16 e suas possíveis aplicações em projetos.

2.2 Específicos

- Introduzir o PIC16;
- Detalhar a Arquitetura;
- Apresentar um projeto prático que utilize a plataforma.

3. SOBRE O MICROCONTROLADOR

3.1 Introdução

Existem vários modelos do circuito integrado, categorizados em Famílias, que podem ser PIC10, PIC12, PIC16, PIC18 e muitos outros,

com diferenças principalmente em relação à CPU (tamanho de palavra, ou conjunto de informações que podem ser resgatadas por ciclo de máquina, mais explicado posteriormente), memória e conjunto de mnemônicos (um mnemônico é uma instrução que, para a programação do CI, representa uma configuração de circuito interna, que realiza uma única operação). Todos são atendidos pela IDE em comum, MPLAB, onde os programas, no interesse do relatório, *Assembly*, são escritos, compilados, depurados, e gravados no chio através de um módulo conectado via comunicação USB. Para exemplificar, o modelo abordado será o PIC16F84A.

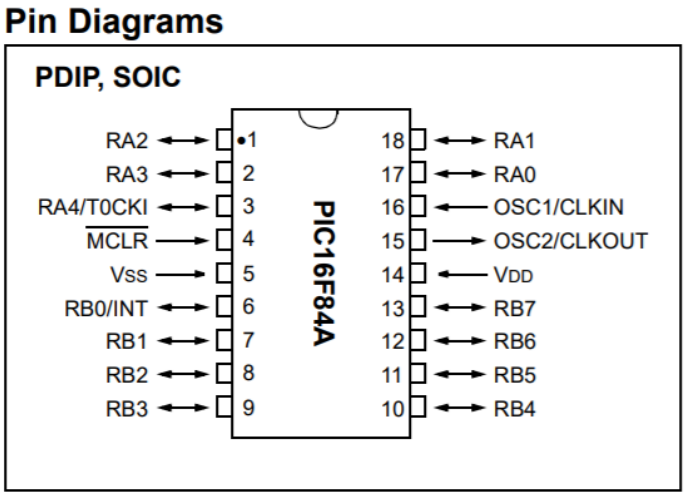


Figura 1 – Diagrama de Pinos do PIC16F84A.
Fonte: Microchip. PIC16F84A Data Sheet. 2001.

O modelo escolhido possui 18 pinos, sendo eles:

- 2 para frequência de operação (CLOCK), um de entrada (16) e um de saída (15);
- 2 de alimentação de circuito, sendo um de maior potencial (14) e um de menor potencial (5), associados a um positivo e um negativo de uma fonte ou bateria, por exemplo;
- 1 reset (4), que limpa o estado do programa e as entradas ao receber valor lógico “0” (tensão de 0V);
- 5 Pinos para o PORTA (lê-se PORT A), sendo pinos bidirecionais de Entrada/Saída (17,18,1,2,3);
- 8 Pinos para o PORTB, sendo pinos bidirecionais de Entrada/Saída (6,7,8,9,10,11,12,13), que também podem ser usados para programação serial ou para ativar interrupções.

Programas escritos para PIC são passados através de um gravador específico, que consiste em um circuito impresso, com uma entrada usb e um conector para pôr e remover o microcontrolador.

3.2 Arquitetura

O PIC utiliza arquitetura Harvard, caracterizada pelo acesso à memória de dados separado da memória de programa existindo um barramento para cada componente (dados, programa e entrada/saída), à unidade de controle. Como ele possui uma ULA, todos os comandos internos são baseados em operações matemáticas de soma, subtração, negação, multiplicação e outras.

As memórias de Programa e de Dados, são o que pode se chamar de memórias dedicadas. Abstendo um pouco sobre os tipos de memória (RAM – Memória de Acesso aleatório, ROM – Memória de Apenas Leitura, EEPROM – Memória Programável de Apenas Leitura Eletricamente apagável, e outras), ao se escrever um programa para PIC, ele identifica alguns registradores específicos para o programa e os acessa.

A memória é dividida em dois bancos de registradores. O de Registradores de Propósitos Gerais (GPR), manipulado pelo programador, e o dos Registradores de Funções Especiais, configurados na fabricação. O que determina a escolha de um registrador é o endereço representado na instrução de máquina. Assim, o TRISB (importante para definir se um pino é de entrada ou saída, por exemplo), encontra-se no endereço 86h (os endereços são dispostos de 00h, ou 00000000 até FFh, ou 11111111). Esse endereço é usado na memória de dados ao ser passado como operador de uma função.

Abaixo, alguns Registradores Importantes:

- STATUS: Registrador 8 bits, os quais podem ser citados:
  - Bit 7 - 6: livres;
  - Bit 5: Seletor de Banco (1 para banco 1, 80h-FFh; 0 para banco 0, 00h-7Fh);
  - Bit 2: Informa se o resultado de uma operação aritmética é zero (1 para sim, zero para não);
  - Bit 1 - 0: Carry de uma operação.
- PORTA: Armazena os bits dos pinos de Entrada/Saída A; definido pelo Bit 5 do STATUS (se 0, PORTA, se 1, TRISA). O mesmo ocorre com PORTB;
- TRISA: Define a direção dos Pinos mencionados em PORTA (se Entrada ou Saída);
- INTCON: Tanto para escrita ou leitura, responsável pelo tratamento de interrupções.

Ao executar um programa, cuja ordem aliás é dada de cima para baixo, começando no endereço 0x00h, frequentemente ocorrem interrupções, em sua grande maioria, definidas pelo usuário. São muito úteis para atrasar um programa (delay proposital, para sincronizar com outros dispositivos que sejam controlados pelo PIC, por exemplo), entrar em um loop (cuja mudança de estado seja assíncrona, como um contador), e muitos outros desvios.

A interrupção ocorre quando a chamada no programa é feita, de forma que o endereço da última instrução executada é armazenado, o

ponteiro na pilha de execução (outra memória disponível no PIC) é movido para o endereço da instrução que inicia o laço (O endereço usado, por costume dos programadores e por orientação da fabricante, é 0x04h). Ao terminar a condição que mantém o programa no laço, o ponteiro aponta para a posição imediatamente posterior à armazenada antes da interrupção.

3.3 Dilema CISC x RISC

Sabendo que qualquer CPU tem um conjunto de instruções (no caso do PIC16 escolhido, será apresentado mais adiante), que é reconhecido e ao qual responde, sendo a base para todos os programas a serem gravados na memória dedicada. Assim, historicamente houve um debate sobre como se daria a evolução tecnológica nos microcontroladores e afins, com o objetivo de respostas cada vez mais rápidas: Aumentar o conjunto de instruções básicas, ou CISC – (do inglês, Computador de Conjunto de de Instruções Complexas), ou prezar por instruções mais simples em um conjunto menor, RISC (do inglês, Computador de Conjunto de Expressões Reduzidas).

Em uma visão superficial, o CISC parece melhor, pois torna a tarefa do programador mais fácil, considerando que as instruções já estão prontas. Entretanto, como, nessa camada de programação, cada instrução representa uma configuração de circuito interno, quanto maior a complexidade da instrução, maior o circuito necessário, e, portanto, maior a dificuldade de implementação. Assim, boa parte dos Microprocessadores, incluindo o PIC, favorece a construção RISC (embora atualmente, os CI's apresentem uma mistura entre as duas opções).

3.4 Instruções no PIC16

Como mencionado, o PIC é um microcontrolador predominantemente RISC, então a dificuldade de se implementar um programa (piscar um LED, por exemplo), é compensada pela pequena quantidade de instruções (35, exatamente).

Para que o processador de qualquer arquitetura computacional realize uma tarefa, é preciso que ele receba um opcode, ou seja, um código de operação. Esse código inclui referências sobre endereços de memória do(s) operando(s) e operador(es) da instrução, número de ciclos (repetições) da operação, etc. O formato dos opcodes varia entre as diferentes arquiteturas de microcontroladores e microprocessadores, sendo que eles são convertidos diretamente em código de máquina.

Entretanto, programar uma máquina, como as ditas acima, através de opcodes é trabalhoso demais para um ser humano, pois não é nada intuitivo (a sintaxe do comando não permite enxergar claramente qual a tarefa realizada pelo mesmo). Para tornar a programação Assembly um tanto mais fácil, são usados os mnemônicos, sendo instruções, as citadas

anteriormente, que já permitem ao programador perceber a tarefa realizada pela semântica do comando (assemelha-se à língua inglesa).

Vejamos o seguinte exemplo, que faz o valor 61 em hexadecimal (97 em decimal) ser movido para o registrador AL (da arquitetura x86, Intel).

*Opcode: IA-32 (B0 61)

* Mnemônico: MOV AL, 61h

Graças ao Assembler (montador) é possível que uma programação feita unicamente com mnemônicos seja convertida inteiramente para opcodes e, consequentemente, para o código de máquina a ser executado diretamente pelo processador. O montador, portanto, é o principal personagem na interação homem-máquina quando se programa em baixo nível (Assembly).

Cada instrução no PIC16F84A é uma “palavra de 14bits” é dividida num opcode que especifica o tipo da instrução, além de um ou mais argumentos que especificam os operandos e operação. Para cada tipo de instrução é mostrado a seguir a quantidade de bits reservada para cada parte.

3.4.1 Instruções Orientadas por Byte

13					8	7	6						0
OPCODE						d	f						

d (bit do registrador destino do resultado da operação): pode receber 0 para que o destino seja registrador W (acumulador) ou 1 para que o destino seja registrador f.

f: letra para simbolizar qualquer registrador de propósito geral (SRAM) dentre os endereços 0x0C a 0x4F, que pode ser usado na operação.

3.4.2. Instruções Orientadas por Bit

13			10	9		7	6						0
OPCODE				b			f						

b = endereço de 3 bits

f = registrador de propósito geral (como dito em x.2.1.1)

3.4.3 Instruções Orientadas por Literal (Valor numérico)

13					8		6						0
OPCODE					k (literal)								

k: literal constante de 8 bits (0 até 255) usado na operação.

3.4.4 Instruções CALL e GOTO

13		11	10										0
OPCODE			k (literal)										

k: literal constante de 11 bits (0 até 2047)

3.5 Conjunto de Instruções do PIC16F84A

INSTRUÇÕES ORIENTADAS POR BYTE	**INSTRUÇÕES ORIENTADAS POR BIT**	**INSTRUÇÕES ORIENTADAS POR LITERAIS**
A. Aritméticas		
A1) ADDWF		A6) ADDLW
A2) DECF		A7) SUBLW
A3) DECFSZ		
A4) INCF		
A5) INCFSZ		
A6) SUBWF		
B. Lógicas		
B1) ANDWF		B5) ANDLW
B2) COMF		B6) IORLW
B3) IORWF		B7) XORLW
B4) XORWF		
C. Movimentação		
C1) MOVF		C3) MOVLW
C2) MOVFW		
D. Outras		
D1) CLRF	D7) BCF	D11) CALL
D2) CLRW	D8) BSF	D12) CLRWDT
D3) NOP	D9) BTFSC	D13) GOTO
D4) RLF	D10) BTFSS	D14) RETFIE
D5) RRF		D15) RETLW
D6) SWAPF		D16) RETURN
		D17) SLEEP

Tabela 1 – Conjunto Organizado das instruções do PIC16F84A.

A tabela acima mostra todos os 35 mnemônicos do PIC16F84A, organizados de acordo com a função que desempenham e com os tipos de dados que manipulam.

A1) ADDWF: Soma W e f

- Sintaxe: ADDWF f, d.
- Operandos: $0 \leq f \leq 127$, sendo $d \in [0,1]$.
- Operação: (W) + (f) → (destino).
- Bits setados do registrador STATUS ao executar: C, DC, Z.
- Descrição: Soma o conteúdo de W ao registrador ‘f’ especificado. Se 'd' for 0,

o resultado é armazenado em W. Caso 'd' seja 1, a soma é armazenada de volta no registrador designado por f.

A2) DECF Decrementa f
- Sintaxe: DECF f, d
- Operandos: 0 ≤ f ≤ 127 e d ∈ [0,1]
- Operação: (f) - 1 → (destino)
- Bits setados do registrador STATUS ao executar: Z
- Descrição: Decrementa o valor de f (subtrai 1 de seu conteúdo) e destina o resultado de forma coerente ao valor de d, como nos exemplos anteriores.

A3) DECFSZ decrementa f e ignora se resultado é 0
- Sintaxe: DECFSZ f, d
- Operandos: 0 ≤ f ≤ 127 e d ∈ [0,1]
- Operação: (f) - 1 → (destino); se o resultado da operação é 0, executa No Operation.
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Decrementa o conteúdo do registrador ‘f’ e joga o resultado convenientemente à escolha do valor de ‘d’ (se d=0, coloca resultado em W; se d=1, coloca no próprio ‘f’). Se o decremento de ‘f’ resultar valor maior que 0, a próxima instrução é executada; entretanto, se o resultado for 0, um NOP é executado.

A4) INCF: Incrementa f
- Sintaxe: INCF f, d
- Operandos: 0 ≤ f ≤ 127 e d ∈ [0,1]
- Operação: (f) + 1 → (destino)
- Bits setados do registrador STATUS ao executar: Z
- Descrição: O conteúdo do registro 'f' é incrementado (somado a 1) e a resposta é colocada em W (se d=0) ou ‘f’ (d=1).

A5) INCFSZ Incremento f e ignora se resultado é 0
- Sintaxe: INCFSZ f, d
- Operandos: 0 ≤ f ≤ 127 e d ∈ [0,1]
- Operação: (f) + 1 → (destino), salta se resultado = 0
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Aqui o conteúdo de 'f' é incrementado e o resultado é testado, se 1, a próxima instrução é executada; se 0, uma pausa (*No Operation*) é executada, tornando-se uma instrução de dois ciclos. O resultado do incremento de ‘f’ é colocado no registrador W se ‘d’ for 0, entretanto, se 'd' for 1, o resultado é colocado de volta em ‘f’.

A6) SUBWF: subtrai W de f
- Sintaxe: SUBWF f, d
- Operandos: 0 ≤ f ≤ 127 e d ∈ [0,1]
- Operação: (f) - (W) → (destino)
- Bits setados do registrador STATUS ao executar: C, DC, Z
- Descrição: Subtrair (método do complemento 2) W do conteúdo de 'f'. Se 'd'

for 0, a diferença é armazenada em W; entretanto, se 'd' for 1, será armazenada de volta em ‘f’.

A7) ADDLW: Soma literal ao registrador acumulador (W)
- Sintaxe: ADDLW k
- Operando: $0 \leq k \leq 255$
- Operação: $(W) + k \rightarrow (W)$
- Bits setados do registrador STATUS ao executar: C (bit 0), DC (bit 1), Z (bit 2)
- Descrição: Ao conteúdo do registrador W é somado o valor da constante k e o resultado retorna para W.

A8) SUBLW: Subtrai W do Literal
- Sintaxe: SUBLW k
- Operandos: $0 \leq k \leq 255$
- Operação: $k - (W) \rightarrow (W)$
- Bits setados do registrador STATUS ao executar: C, DC, Z
- Descrição: Do valor de ‘k’ é subtraído o conteúdo de W (usando o método de complemento de dois). O resultado é colocado em W (acumulador para operações matemáticas).

B1) ANDWF: E lógico de W com f
- Sintaxe: ANDWF f, d
- Operandos: $0 \leq f \leq 127$, sendo que $d \in [0,1]$
- Operação: (W) AND $(f) \rightarrow$ (destino)
- Bits setados do registrador STATUS ao executar: Z
- Descrição: Análogo ao 3), mas a operação é realizada entre conteúdo de W e o de f. Se 'd' for 0, a operação é armazenada em W, senão, o resultado é armazenado de volta em ‘f’.

B2) COMF Complemento f
- Sintaxe: COMF f, d
- Operandos: $0 \leq f \leq 127$ e $d \in [0,1]$
- Operação: $(f) \rightarrow$ (destino)
- Bits setados do registrador STATUS ao executar: Z
- Descrição: O conteúdo do registrador 'f' é complementado, ou seja, os bits são invertidos. O destino do resultado é escolhido pelo digito d: se 0, o resultado é armazenado em W, senão, no próprio registrador 'f', de origem.

B3) IORWF: OU lógico (inclusivo) de W com f
- Sintaxe: IORWF f, d
- Operandos: $0 \leq f \leq 127$ e $d \in [0,1]$
- Operação: (W) OR $(f) \rightarrow$ (destino)
- Bits setados do registrador STATUS ao executar: Z
- Descrição: Realiza o “ou inclusivo” bit a bit de W com o registrador ‘f’. O resultado da operação é colocado no próprio f (caso d=1) ou em W (caso d=0).

B4) XORWF OU lógico (exclusivo) de W com f
- Sintaxe: XORWF f, d
- Operandos: 0 ≤ f ≤ 127 e d ∈ [0,1]
- Operação: (W) XOR (f) → (destino)
- Bits setados do registrador STATUS ao executar: Z
- Descrição: É tomado o conteúdo de W, feito o “ou exclusivo” bit a bit com o conteúdo de 'f'. O resultado pode ser colocado no registrador W (se d=0) ou em ‘f’ (se d=1).

B5) ANDLW: E lógico k com W
- Sintaxe: ANDLW k
- Operandos: 0 ≤ k ≤ 255
- Operação: (W) AND (k) → (W)
- Bits setados do registrador STATUS ao executar: Z
- Descrição: É feita a operação do “E lógico” (AND) do conteúdo de W com o valor binário de 'k'. Faz-se a operação bit a bit entre os dois valores e armazena-se em W.

B6) IORLW: OU lógico (inclusivo) de literal k com W
- Sintaxe: IORLW k
- Operandos: 0 ≤ k ≤ 255
- Operação: (W) OR k → (W)
- Bits setados do registrador STATUS ao executar: Z
- Descrição: É feita operação lógica bit a bit do “ou inclusivo” de 'k' (oito bits) o conteúdo de W. O resultado é colocado em W. A tabela para o “ou inclusivo” encontra-se abaixo:

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

Figura 2 – Operação do OU lógico inclusivo

B7) XORLW: OU lógico (exclusivo) de literal k com W
- Sintaxe: XORLW k
- Operandos: 0 ≤ k ≤ 255
- Operação: (W) XOR k → (W)
- Bits setados do registrador STATUS ao executar: Z
- Descrição: É tomado o valor binário de ‘k’, feito o “ou exclusivo” bit a bit com o conteúdo de W. O resultado é colocado no registrador W.

A	B	Saída
0	0	0
0	1	1
1	0	1
1	1	0

Figura 3 – Operação do OU lógico exclusivo

C1) MOVF: Move conteúdo de registrador f (origem) para destino
- Sintaxe: MOVF f, d
- Operandos: $0 \leq f \leq 127$ e $d \in [0,1]$
- Operação: (f) → (destino)
- Bits setados do registrador STATUS ao executar: Z
- Descrição: O conteúdo do registrador ‘f’ é movido para o registrador acumulador W (se d=0) ou o próprio ‘f’ (se d=1). O caso de mover de ‘f’ para ele mesmo é útil para testar se a operação é realizada com sucesso, pois a sinalização do seu funcionamento é a alteração do bit Z do registrador STATUS mudar de 0 para 1.

C2) MOVWF: Move conteúdo de W para f
- Sintaxe: MOVWF f
- Operandos: $0 \leq f \leq 127$
- Operação: (W) → (f)
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Mover dados do registro W para registre-se 'f'.

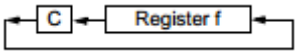
C3) MOVLW: Move valor binário de literal k para W
- Sintaxe: MOVLW k
- Operandos: $0 \leq k \leq 255$
- Operação: k → (W)
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: O valor literal 'k' (oito bits) é carregado no registrador W.

D1) CLRF: Limpa f
- Sintaxe: CLRF f
- Operandos: $0 \leq f \leq 127$
- Operação: 00h → (f), 1 → Z
- Bits setados do registrador STATUS ao executar: Z
- Descrição: O conteúdo do registrador 'f' é limpo (zerado) e no bit Z (bit 2 do registrador STATUS) é colocado 1.

D2) CLRW: Limpa W
- Sintaxe: CLRW
- Operandos: Nenhum
- Operação: 00h → (W), 1 → Z
- Bits setados do registrador STATUS ao executar: Z
- Descrição: O conteúdo do registrador é desmarcado. Bit Z do registrador STATUS também é “setado” em 1.

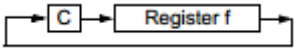
D3) NOP: No Operation
- Sintaxe: NOP
- Operandos: Nenhum
- Operação: sem operação
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Interrupção da execução de qualquer instrução para que o sistema fique “ocioso”. A instrução NOP dura 1 ciclo (4 períodos do oscilador).

D4) RLF: Rotate Left f to Carry
- Sintaxe: RLF f, d
- Operandos: 0 ≤ f ≤ 127 e d ∈ [0,1]
- Operação:



- Bits setados do registrador STATUS ao executar: C
- Descrição: O conteúdo (binário) do registrador 'f' é deslocado um bit para a esquerda (o ‘L’ do mnemônico vem de “Left”) considerando também a flag de Carry. Se 'd' for 0, o resultado do deslocamento é colocado em W. Se 'd' for 1, é armazenado de volta no registrador 'f'. O antigo bit mais significativo de ‘f’ torna-se o dígito da nova flag de Carry com o deslocamento.

D5) RRF: Rotate Right f to Carry
- Sintaxe: [rótulo] RRF f, d
- Operandos: 0 ≤ f ≤ 127 e d ∈ [0,1]
- Operação:



- Bits setados do registrador STATUS ao executar: C
- Descrição: Esquema de funcionamento similar ao RLF, mas aqui o deslocamento é de um bit à direita (o segundo ‘R’ do mnemônico vem de “Right”). O resultado pode ser jogado em W (se d=0) ou posto de volta em ‘f’ (se d=1). O digíto da antiga flag de carry torna-se o bit mais significativo de ‘f’ com o deslocamento.

D6) SWAPF: Troca Nibbles em f
- Sintaxe: SWAPF f, d
- Operandos: 0 ≤ f ≤ 127 e d ∈ [0,1]
- Operação: (f <3: 0>) → (destino <7: 4>), (f <7: 4>) → (destino <3: 0>)
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Os nibbles (conjunto de 4 bits) superiores e inferiores do registrador simbolizado por ‘f’ (8 bits) são trocados de posição. Assim, os 4 bits mais significativos de f tornar-se-ão os menos significativos e vice-versa. O ‘d’ (destino) pode assumir 0 (resultado da troca vai para W) ou 1 (vai de volta para ‘f’).

D7) BCF: Bit Clear f
- Sintaxe: BCF f, b
- Operandos: 0 ≤ f ≤ 127 e 0 ≤ b ≤ 7
- Operação: 0 → (f <b>)
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Bit 'b' no registrador 'f' é limpo (colocado como 0). Os valores de b vão de 0 a 7 pois a operação trata cada bit individualmente. Como exemplo, pode-se fazer o 2º mais significativo (bit 6) do registrador em questão ser 0, através dessa operação.

D8) BSF: Bit Set f
- Sintaxe: BSF f, b
- Operandos: 0 ≤ f ≤ 127 e 0 ≤ b ≤ 7
- Operação: 1 → (f <b>)
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Bit 'b' no registrador é “setado” (colocado como 1). Os valores de b vão de 0 a 7 por motivo igual ao mnemônico 5).

D9) BTFSC: Bit Test, Ignorar se zero
- Sintaxe: BTFSC f, b
- Operandos: 0 ≤ f ≤ 127 e 0 ≤ b ≤ 7
- Operação: salte se (f <b>) = 0
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Se o bit 'b' no registro 'f' for '1', a próxima instrução é executada. Se bit 'b' no registro 'f' for '0', a próxima instrução é descartada e um NOP é executado

D10) BTFSS: Bit Test f, Ignorar se um
- Sintaxe: BTFSS f, b
- Operandos: 0 ≤ f ≤ 127 e 0 ≤ b <7
- Operação: ignorar se (f <b>) = 1
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Se o bit 'b' no registrador 'f' for '0', a próxima instrução é executada. Se bit 'b' for '1', então a próxima instrução é descartada e um NOP (No Operation) é executada. Dessa forma, a execução toma dois ciclos do processador: uma pro teste do bit e outra pra execução da instrução (caso o bit seja 0) ou para o No Operation (caso o bit testado seja 1).

D11) CALL: Chama execução de label
- Sintaxe: CALL k
- Operandos: 0 ≤ k ≤ 2047
- Operação: (PC) + 1 → TOS, k → PC <10: 0>, (PCLATH <4: 3>) → PC <12:11>
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Chama uma subrotina (label). Primeiro endereço de retorno (PC + 1) é empurrado para a pilha. O endereço da constate k (de onze bits) é carregado no contador de programa (Program Counter) em <10: 0>. Os bits superiores do PC são carregados do PCLATH. CALL is uma instrução de dois ciclos.

D12) CLRWDT: Limpa o Watchdog Timer
- Sintaxe: CLRWDT
- Operandos: Nenhum
- Operação: 00h → WDT, 0 → Prescaler WDT, 1 → TO, 1 → PD
- Bits setados do registrador STATUS ao executar: TO, PD
- Descrição: A instrução restaura o Watchdog Timer e Prescaler do WDT.

D13) GOTO: desvio incondicional
- Sintaxe: GOTO k
- Operandos: 0 ≤ k ≤ 2047
- Operação: k → PC <10: 0>, PCLATH <4: 3> → PC <12:11>
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: A instrução é uma ordem de desvio para um label em qualquer parte do código. O "salto" é feito sem testar qualquer condição, portanto chama-se instrução de desvio incondicional. O valor de onze bits de k é carregado no Program Counter <10: 0>, os bits superiores do PC são carregados de PCLATH <4: 3>.

D14) RETFIE: Retorno de Interrupção
- Sintaxe: RETFIE
- Operandos: Nenhum
- Operação: TOS → PC, 1 → GIE
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Caso alguma instrução do programa necessite de "atenção especial" do sistema (pedido de interrupção), a interrupção para execução daquele recurso é concedida e, após a final da interrupção, retoma-se o andamento do programa. Para esse retorno, a instrução RETFIE é necessária. Ela deve ser escrita logo após a definição do endereço 0x04 à interrupção através da diretiva ORG.

D15) RETLW: Retorno de literal k em W
- Sintaxe: RETLW k
- Operandos: 0 ≤ k ≤ 255
- Operação: k → (W); TOS → PC
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: O conteúdo do reg. acumulador W é carregado com o valor do literal 'k' de oito bits. O PC (contador de programa) é carregado do topo da pilha (o endereço de retorno). A instrução toma dois ciclos de operação.

D16) RETURN: Retorno de sub-rotina
- Sintaxe: RETURN
- Operandos: Nenhum
- Operação: TOS → PC
- Bits setados do registrador STATUS ao executar: Nenhum
- Descrição: Retorno da sub-rotina/label. O topo da pilha (TOS) é retirado da mesma e seu endereço de retorno para o programa principal é carregado no contador de programa (PC). A instrução toma dois ciclos de operação.

D17) SLEEP (pausa ou descanso do sistema)
- Sintaxe: SLEEP
- Operandos: Nenhum
- Operação: 00h → WDT, 0 → Prescaler WDT, 1 → TO, 0 → PD
- Bits setados do registrador STATUS ao executar: TO, PD
- Descrição: O bit de status de desligamento (PD) é limpo (jogado 0). O bit de status ocioso do sistema (TO) é setado (jogado 1). Os bits Watchdog Timer e o pre-scaler são apagados. O processador é colocado no modo de descanso e o oscilador de clocks do sistema fica parado.

4. EXEMPLO PRÁTICO NO SIMULADOR

Dentre o conjunto de simuladores para PIC disponíveis no mercado, um dos mais difundidos é o MPLABX IDE®, da MicroChip®. Nele, várias arquiteturas são suportadas, inclusive o PIC16F84A, escolhido pela equipe. O software é completo, tendo a capacidade de exibir o conteúdo de todos os registradores de propósito específicos, de registradores de memória (propósito geral) e flags. É satisfatória a interação do usuário com o dispositivo simulado, pois pode-se escolher limpar ou setar os bits de vários registradores da maneira desejada.

Como exemplo prático, será relatado toda a experiência da simulação de um programa no PIC16F84A, mostrando todas as etapas necessárias desde a criação do projeto até o resultado final. Vale lembrar que, para avançar nas etapas de criação do projeto, é necessário pressionar o botão “Next”.

4.1. Abertura do software e criação do projeto

A tela inicial do programa conta com vários menus em sua parte superior. Para simular qualquer aplicação, é necessário criar um novo projeto, o que se pode fazer em File > New Project, que abrirá a janela exibida na Fig. 4. Caso se queira abrir um projeto já existente, o processo é similar, bastando ir em File > Open Project. Dentre as Categorias e Projetos, escolher “Microchip Embedded” e “Standalone Project” respectivamente.

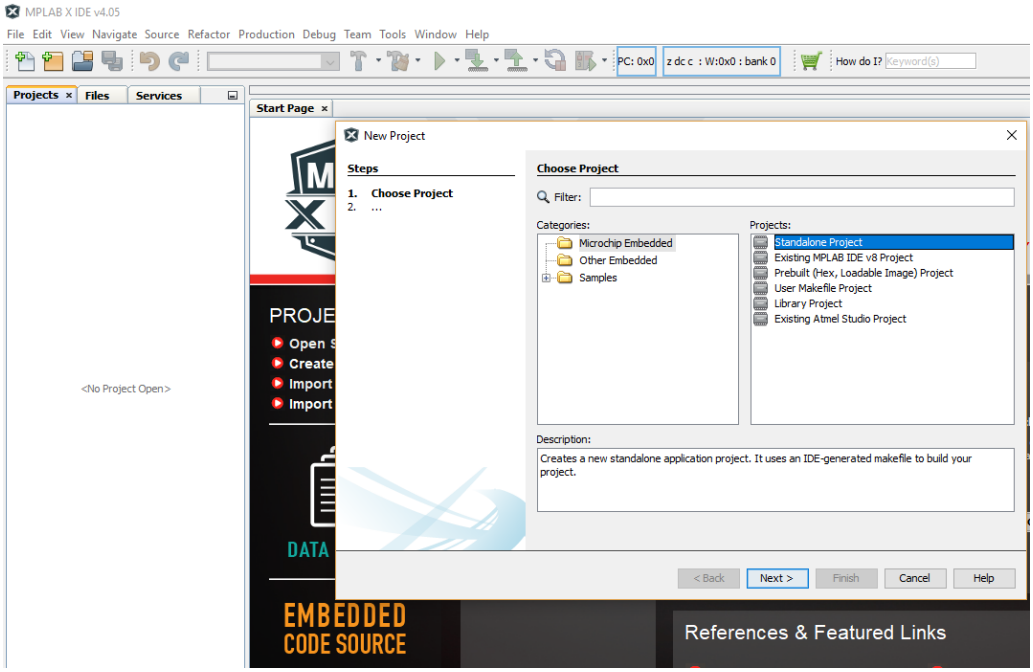


Fig. 4

Na criação de um novo projeto, é necessário especificar a arquitetura que será utilizada, para isso, o MPLABX requer que o usuário especifique a família e o próprio dispositivo. No nosso caso, a família foi escolhida como *Baseline 8-bit MCUs (PIC 10/12/16)* e dispositivo como *PIC16F84A* (Fig. 5).

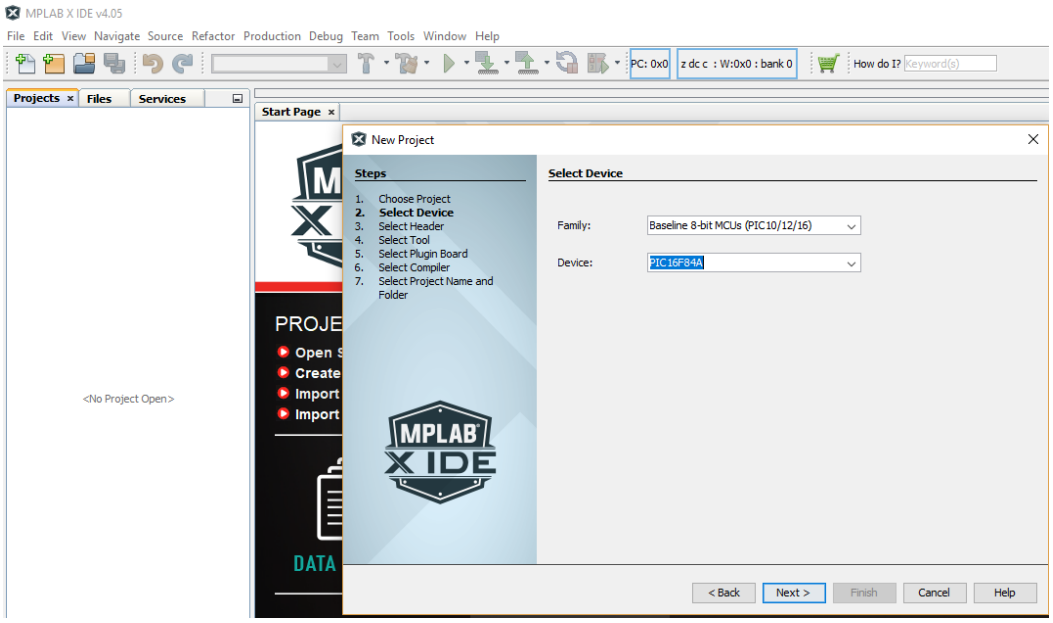


Fig. 5: Escolha da família e dispositivo

Após a escolha do chip do chip, o software exibe uma lista com as várias opções de ferramentas (Tools) disponíveis para uso. Aqui, pode-se escolher trabalhar com kits para desenvolvedores de PIC, outras ferramentas da Microchip, além do simulador, objeto da nossa escolha (Fig. 6)

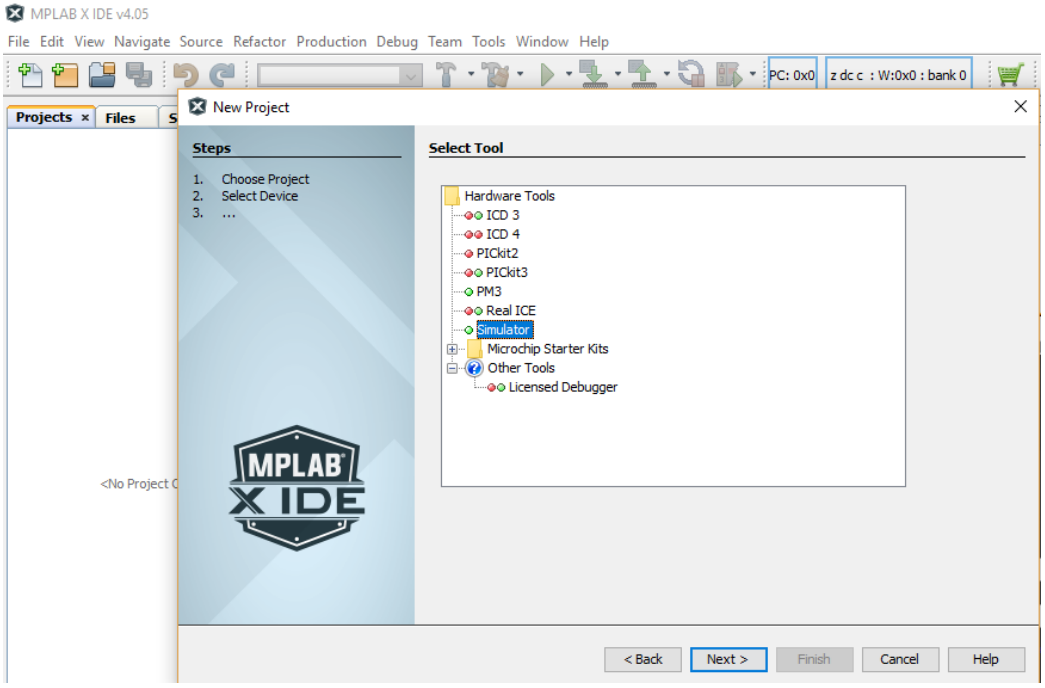


Fig. 6: Escolha de ferramentas auxiliares para execução

Próxima etapa é a escolha do compilador/montador. Para a linguagem assembly, é necessário um montador e várias são as opções disponíveis no mercado, apesar do MPLABX recomendar o *mpasm*, que já vem integrado à IDE da Microchip. Uma vez que o software suporta programação em outras linguagens (como C) para microcontroladores, pode-se integrar compiladores C à IDE, a exemplo do XC8, que consta na figura abaixo, apesar de não ser utilizada neste propósito.

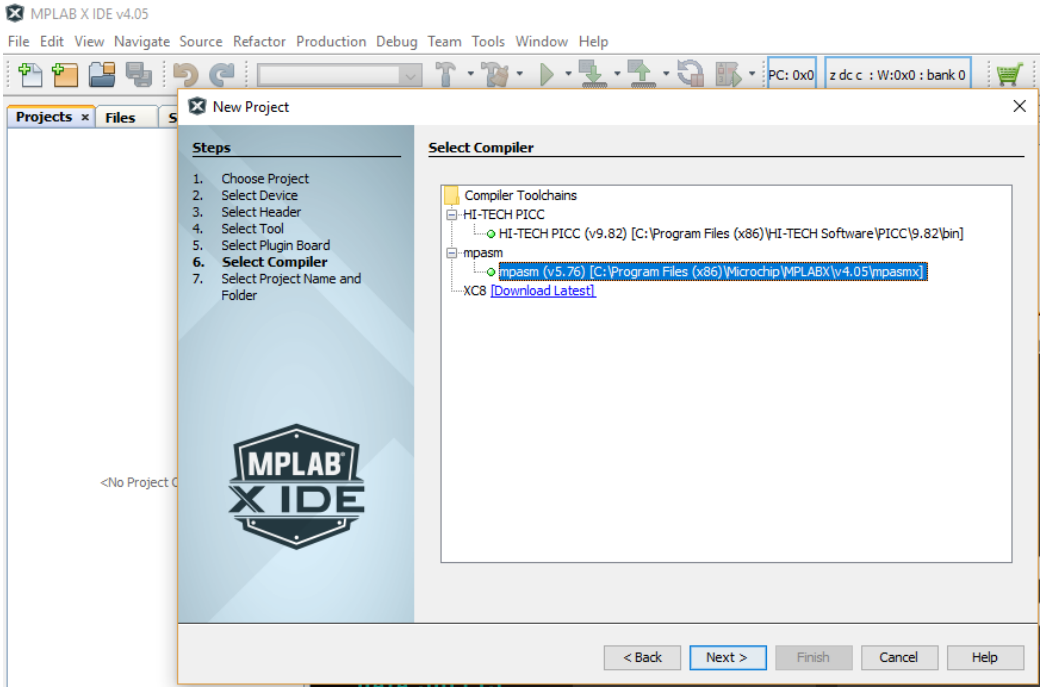


Fig. 7: Opção do montador

Já na etapa final de criação do projeto, é necessário escolher o nome para o mesmo, além do local onde sua pasta ficará armazenada. A escolha de definir o projeto em questão como o principal (main project) e o tipo de codificação (Encoding) são de escolha do programador. Aqui, estes dois não foram alterados pela equipe e ficaram com seus valores padrão (Fig. 8). Agora, o botão a ser pressionado é “Finish”, que conduzirá à parte de codificação do programa.

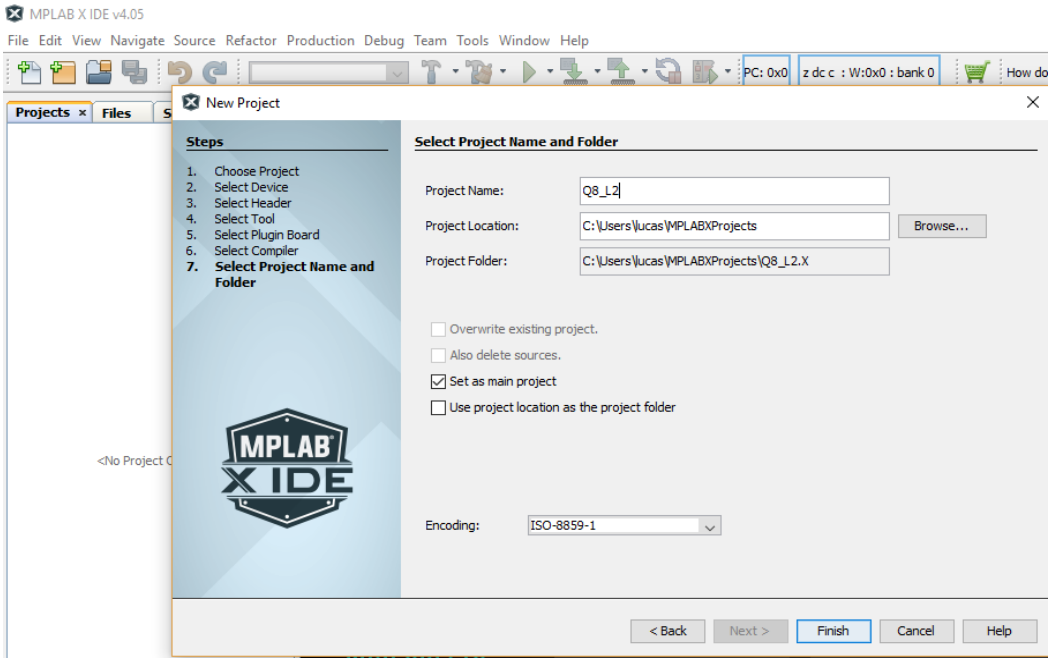


Fig. 8: Nome do projeto, local e codificação

Terminada a criação, o projeto recém-criado ficará disposto num esquema de árvore hierarquizada, donde o código-fonte a ser inserido pelo programador deverá constar no arquivo .asm dentro de *NomeDoProjeto>SourceFiles>NomedoArquivo.asm.* O exemplo a ser descrito no relatório é de um exercício da disciplina ENG50 - Sistemas Microprocessados, portanto o nome do projeto e arquivo ficaram respectivamente como “Q8_L2” e “q8_lista2.asm” (Fig. 9).

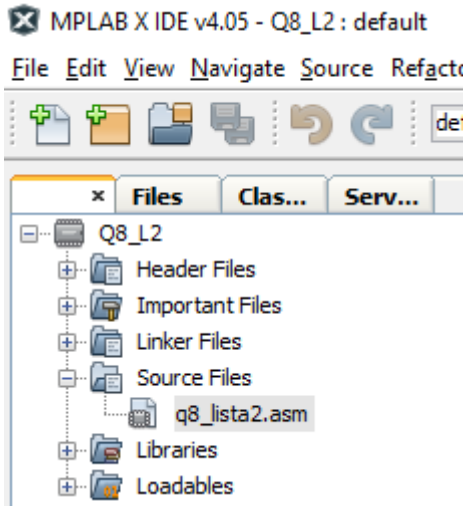


Fig. 9: Estrutura de navegação de projetos no MPLABX

Antes de executar a simulação, é importante abrir janelas auxiliares à depuração. São elas:

- SFRs: acessível por Window > PIC Memory Views > SFRs
- File Registers: acessível por Window > PIC Memory Views > File Registers
- Stimulus: acessível por Window > Simulator > Stimulus

4.2 Detalhes da aplicação e seu funcionamento

A aplicação a ser simulada recebe 4 números pela entrada, calcula a média aritmética deles e envia o resultado por alguma saída. É importante lembrar que a média é aproximada, retornando um natural truncado. A seguir, estão as indicações do papel de cada registrador escolhido no nosso programa:

- Entrada: registrador PORTA
- Saída: registrador PORTB
- Acumulador dos termos das somas: registrador W

4.2.1 Código Fonte

O código completo da aplicação, apesar de ter sido desenvolvido em um único arquivo .asm, foi dividido em blocos neste documento por ser melhor para relacionar cada trecho à explicação e processo de depuração correspondente. Foram coletadas imagens da depuração, que facilitará o entendimento do leitor.

4.2.1.1 Bloco 1: Listagem do dispositivo, inclusão de cabeçalhos obrigatórios e definição de endereços de início e interrupção

```
list p=pic16f84a
    #include <p16f84a.inc>
    __config _XT_OSC & _PWRTE_ON & _CP_OFF & _WDT_OFF
    org 0x000
    goto start
    org 0x004
    retfie
```

Além da importação obrigatória dos cabeçalhos e da associação dos endereços padrão de início do programa (0x000) e de interrupção (0x004), há um salto incondicional para o trecho de código definido após a label “start”. Aqui também são definidos a ativação e desativação de recursos do PIC16F84A:

- Cristal Oscilador (XT_OSC): gerador de clocks do sistema.

- Power-Write (PWRTE_ON)
- Clock Pulse (CP_OFF)
- WatchDog Timer (WDT_OFF)

4.2.1.2 Bloco 2: Definição dos bits de PORTA como entrada e dos bits de PORTB como saída

```
start bsf STATUS,RP0
bsf TRISA,0
bsf TRISA,1
bsf TRISA,2
bsf TRISA,3
bsf TRISA,4
bcf TRISB,0
bcf TRISB,1
bcf TRISB,2
bcf TRISB,3
bcf TRISB,4
bcf TRISB,5
bcf TRISB,6
bcf TRISB,7
bcf STATUS,RP0
```

Cada bit (0,1,2,3,4) de TRISA setado (definido como 1, pelo mnemônico *bsf*) fará com que o bit correspondente de PORTA seja definido como bit para “entrada” no sistema. O contrário ocorre com PORTB, onde todos seus bits (0-7) são definidos como saída, pois os correspondentes em TRISB foram limpos pelo mnemônico *bcf*. As linhas que envolvem o registrador “STATUS” fazem respectivamente, entra no banco de registradores onde pode acessar e modificar TRISA e TRISB e sai dele, em seguida.

Vale lembrar que a definição de entrada e saída é totalmente arbitrária e depende unicamente da escolha do programador, servindo então como modo de se orientar em que local inserir o “estímulo do mundo exterior” e onde aguardar os resultados do processamento da aplicação.

4.2.1.3 Bloco 3: Definição da rotina “soma”, cálculo da média, movimentação para saída e fim do programa

```
soma
movf PORTA,0
addwf PORTA,0
addwf PORTA,0
addwf PORTA,0
andlw b'11111100'
movwf 0x0C
rrf 0x0C,1
rrf 0x0C,0
movwf PORTB
end
```

O procedimento “soma” inicia movendo o valor binário do 1º termo da soma para o registrador W (mnemônico *movf* com destino *0*). Em seguida, soma esse valor de W a mais três números recebidos em seguida pelo PORTA (três linhas seguidas), jogando o resultado novamente em W.

O mnemônico *andlw* serve para “mascaramento” do valor em W. Apesar desse processo não ser obrigatório, ele “arredonda” o valor para algum cuja divisão por 4 seja exata. Como exemplo, 14/4=3,5, mas 14 and 3, isto é, 00001110 and 11111100, resulta 00001100 = 12, sendo a divisão de 12 por 4 igual a 3, valor exato. Esse arredondamento é uma sofisticação do programa e não fazê-lo não incorreria em erros no resultado da média pois a arquitetura não opera em ponto flutuante.

Como há um grande número de registradores de memória à disposição, o valor de W dos 4 termos somados é armazenado no registrador qualquer, como o de endereço hexadecimal 0x0C.

A operação de divisão por 2 vem da rotação de todos os bits uma vez para a direita. Assim, ao rotacionarmos duas vezes consecutivas, estaremos dividindo o número por 4. Da primeira vez, o destino do resultado ainda é o registrador de memória 0x0C, mas, da segunda, é o acumulador W. Ao fim da operação, move-se a média para PORTB e lá o resultado é impresso.

Algumas imagens a seguir mostram os valores nos registradores para o exemplo feito com os termos 14, 14, 0, 0, que resultam na média 7.

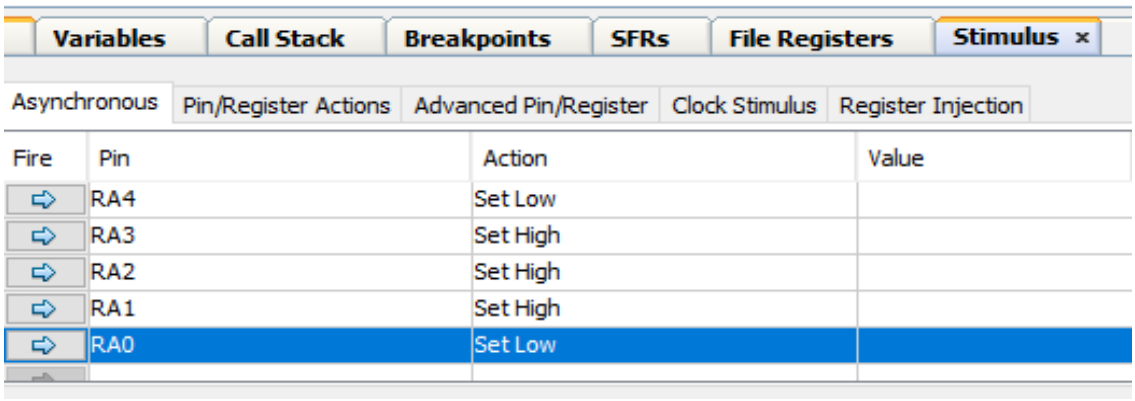


Figura 10: Forma de inserir o número 14 no PORTA (bits 4 e 0 limpos e 3,2,1 setados: 01110)

```
movf PORTA,0 ;MOVE PRIMEIRO NUMERO PARA REG W
; AQUI DEVE ENTRAR NÚMERO 2 A SER SOMADO
addwf PORTA,0
; AQUI DEVE ENTRAR NÚMERO 3 A SER SOMADO
addwf PORTA,0
; AQUI DEVE ENTRAR NÚMERO 4 A SER SOMADO
addwf PORTA,0
; SOMADOS ATÉ ENTÃO OS 4 NÚMEROS AGORA FAZ-SE A DIVISÃO POR 4

movwf 0x0C; moveu o conteúdo de W (número somado) para o registr
andlw b'11111100'
```

Variables | Call Stack | Breakpoints | SFRs | File Registers | Stimulus

Address	Name	Hex	Decimal	Binary	Char
	WREG	0x0E	14	00001110	'.'
	TMR0	0x00	0	00000000	'.'
	PCL	0x15	21	00010101	'.'
	STATUS	0x1B	27	00011011	'.'

Figura 11: Após a execução de **movf PORTA,0**, o acumulador W (WREG) recebe 14 (em vermelho, pois é alteração mais recente).

```
40  movwf 0x0C; moveu o conteúdo de W (número somado) para o registr
    andlw b'11111100'
42  rrf 0x0C,1 ; para rotacionar uma vez para a direta (dividir por 2) o número em W
43  rrf 0x0C,0; para rotacionar uma vez para a direta (dividir por 2) o número em W
44  movwf PORTB; move de volta o conteúdo do reg W para PORTB, para ser exibido.
45  end
```

Output | Variables | Call Stack | Breakpoints | SFRs | File Registers | Stimulus

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
00	00	00	19	18	00	00	00	--	00	00	00	00	1C	00	00	00	.....- .....
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Figura 12: Após a soma de 14,14, 0, 0, e da movimentação de W=28 para o registrador 0x0C (recebe 1C em hexa, que é o mesmo que 28 em decimal)

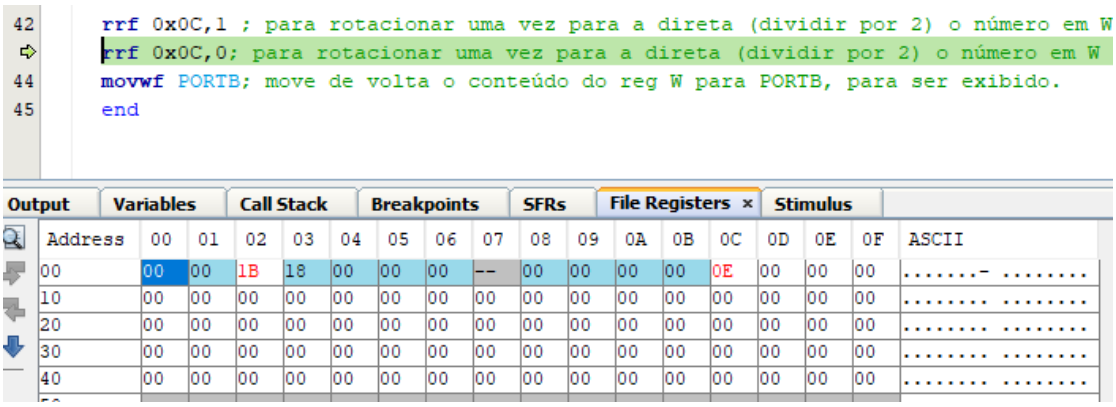


Figura 13: 0x0C recebe o resultado do primeiro deslocamento de bit (1ª divisão por 2) que resultou 14 (0E em hexa)

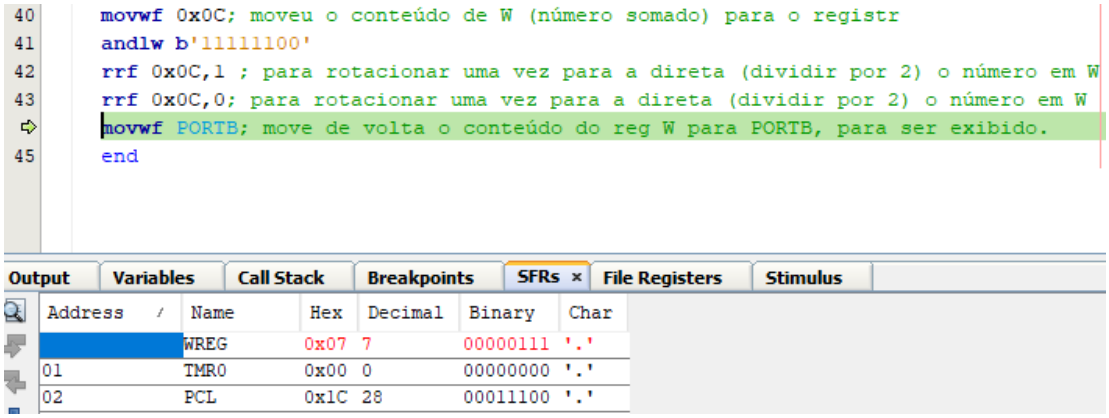


Figura 14: W recebe o resultado do segundo deslocamento de bit (2ª divisão por 2) que resultou em 7.

5. CONCLUSÃO

O processo de analisar a estrutura interna do microcontrolador PIC16, escolhido devido à utilização da linguagem Assembly, e verificar seu funcionamento em um projeto prático, facilitou a compreensão de como essa linguagem de programação está relacionada ao desempenho do circuito, bem como sua importância. O estudo também reforçou conhecimentos de disciplinas anteriores, como Sistemas Microprocessados.

6. REFERÊNCIAS

Engineers Garage.
Disponível em: <https://www.engineersgarage.com/tutorials/difference-between-microprocessor-and-microcontroller>. Acesso em 27 de Janeiro de 2018.

Microchip. PIC16F84A Data Sheet. 2001.

Wikipedia. Disponível em:
<https://pt.wikipedia.org/wiki/Microcontrolador_PIC>. Acesso em 27 de Janeiro de 2018.