PIC16F877 Core Features

- Accumulator Based Machine
- Harvard Architecture Memory (separate program and data memory)
  - 8Kx14 Flash Based Instruction Memory
  - 368x8 Static Ram Based Data Memory (File Registers)
- 35 Instructions (fixed length encoding - 14-bit)
- 3 Addressing Modes (direct, indirect, relative)
- 8x13 Hardware Stack (8 levels - not visible from program code)
- Execution Speed
  - Overlapped Instruction Fetch and Instruction Execution
  - 1 cycle/instruction (non-branching)
  - 2 cycles/instruction (branching)
  - 1 cycle period = 4/CLK_IN (ex. 20Mhz CLK_IN -> 200ns cycle period)

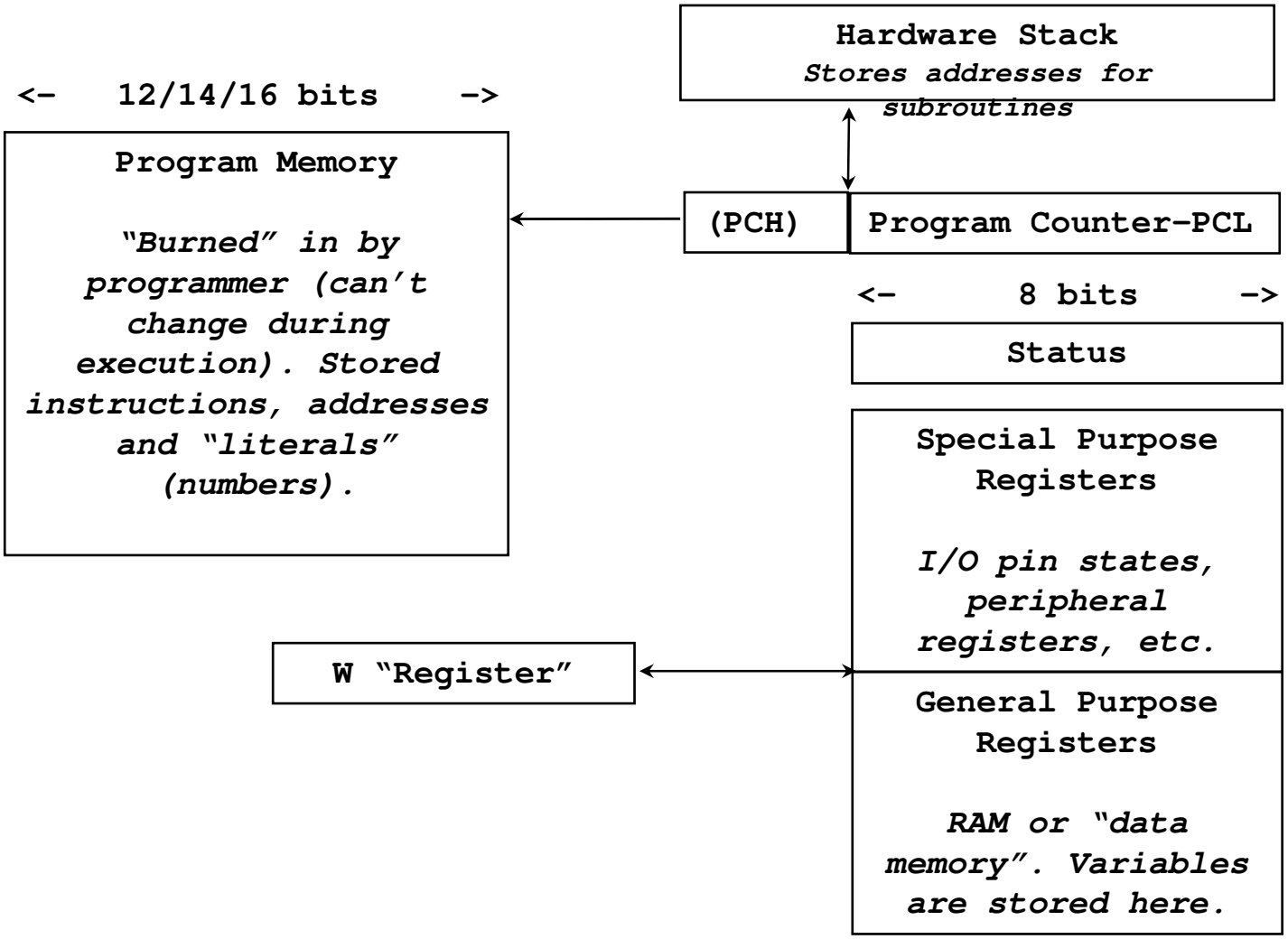
PIC16F877 Peripheral Features

- 3 Timer/counters (programmable prescalars)
  - Timer0,Timer2 8-bit
  - Timer1 16-bit
- 2 Capture/Compare/PWM modules
  - Input capture function records the Timer1 count on a pin transition
  - Output compare function transitions a pin when Timer1 matches a programmable register
  - Pulse width modulation function outputs a square wave with a programmable period and duty cycle.
- 10-bit 8 channel analog-to-digital converter
- Synchronous serial port
- USART
- 8-bit Parallel Slave Port - function allow another processor to read from a data buffer in the PIC.
- 256 bytes of EEPROM Memory
- Interrupts can be generated from each of the peripherals - single vector with a status reg.

PIC16F877 Development Tools

- **MPLAB** - Integrated Development Environment
  - Editor
  - Build Tools (Assembler and Linker)
  - Simulator
- Download for Free @ Microchip.com

Software: Programmers Model



Memory Organization

- Program Memory
- Register File Memory

Program Memory

- Used for storing compiled code
- Each location is 14 bits long
- Every instruction is coded as a 14 bit word
- Addresses H’000’ and H’004’ are treated in a special way
- PC can address up to 8K addresses

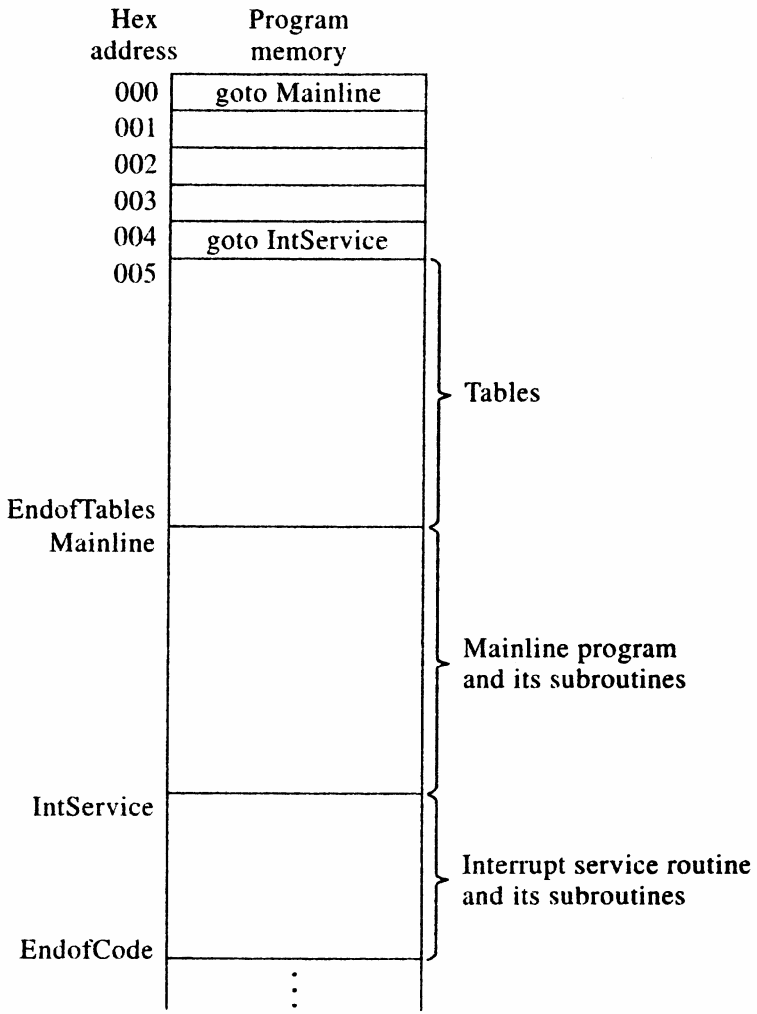
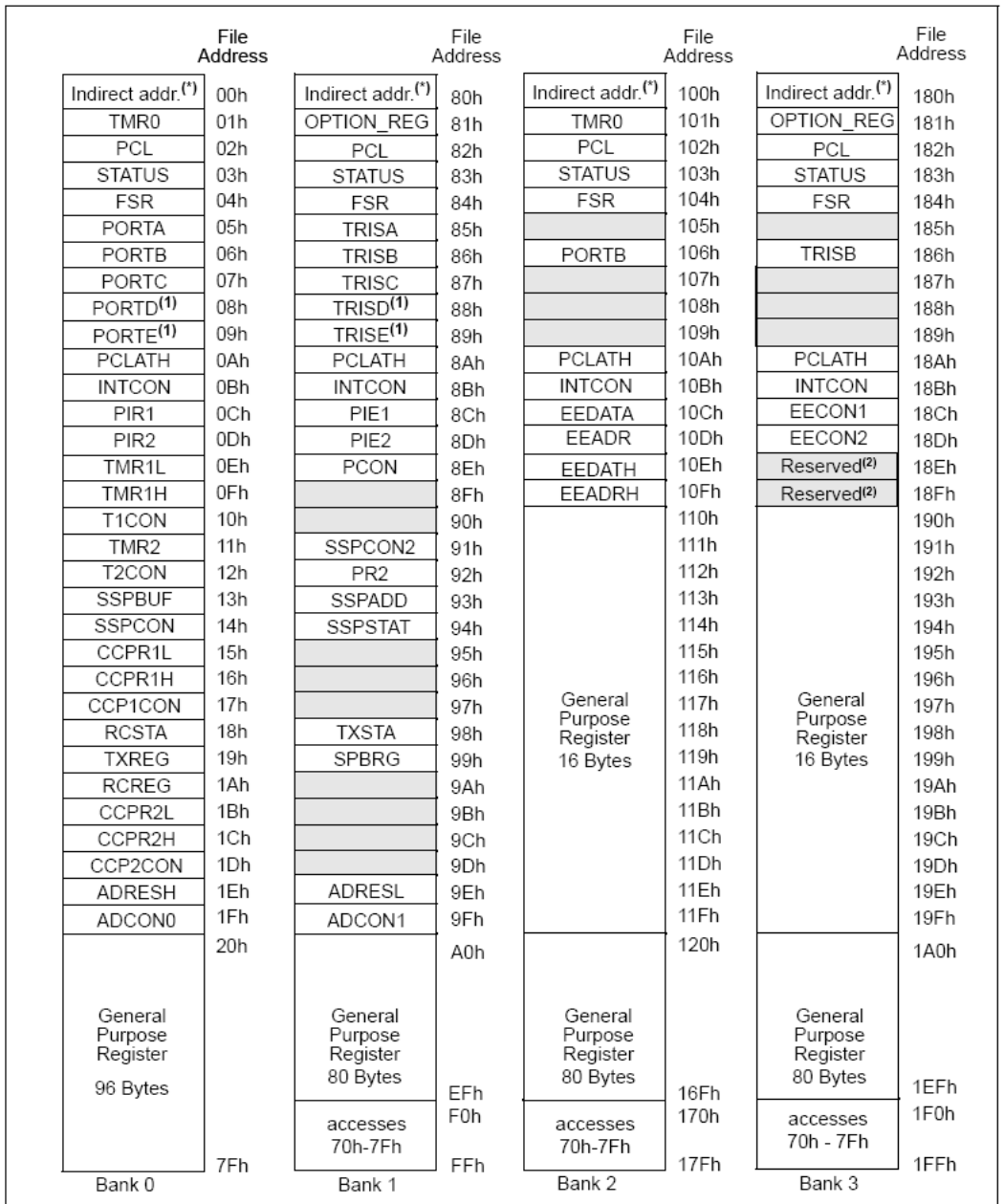


Figure 4 Program memory map

Register File Memory

- Consist of 2 Components
  - General Purpose Register (GPR) Files (RAM)
  - Special Purpose Register (SPR) files
- This portion of memory is separated into banks of 128 bytes long

FIGURE 2-3: PIC16F877/876 REGISTER FILE MAP



Unimplemented data memory locations, read as '0'.
* Not a physical register.

Note 1: These registers are not implemented on the PIC16F876.
2: These registers are reserved, maintain these registers clear.

Register Addressing Modes

- There are 3 types of addressing modes in PIC
  - Immediate Addressing
    - Movlw H'0F'
  - Direct Addressing
  - Indirect Addressing

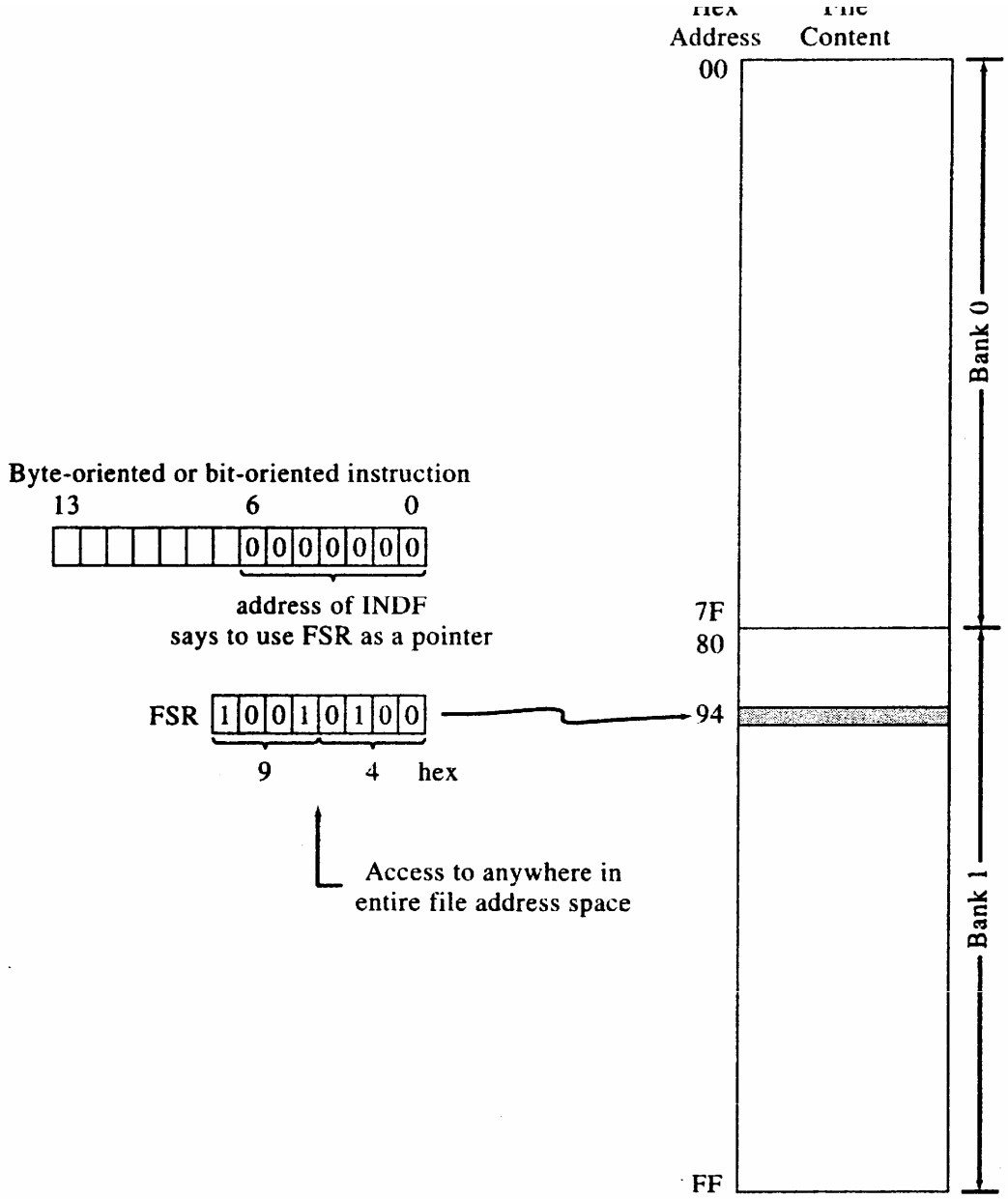
Direct Addressing

- Uses 7 bits of 14 bit instruction to identify a register file address
- 8^{th} and 9^{th} bit comes from RP0 and RP1 bits of STATUS register.

Indirect Addressing

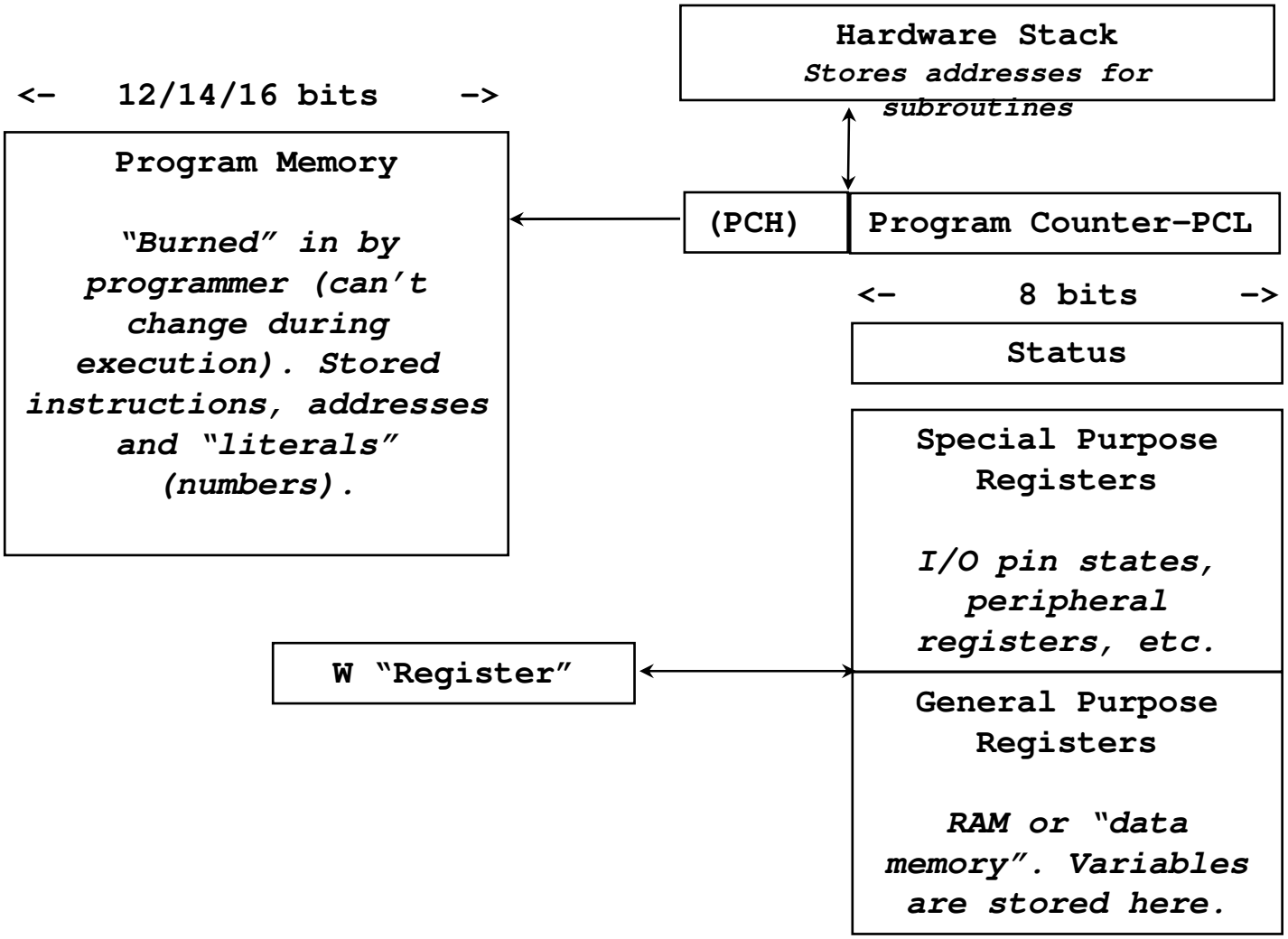
- Full 8 bit register address is written the special function register FSR
- INDF is used to get the content of the address pointed by FSR
- Exp : A sample program to clear RAM locations H’20’ – H’2F’ .

- for instance,
  - one general purpose register (GPR) at address 0Fh contains a value of 20
  - By writing a value of 0Fh in FSR register we will get a register indicator at address 0Fh,
  - and by reading from INDF register, we will get a value of 20, which means that we have read from the first register its value without accessing it directly (but via FSR and INDF).
- It appears that this type of addressing does not have any advantages over direct addressing, but certain needs do exist during programming which can be solved smoothly only through indirect addressing.
- Indirect addressing is very convenient for manipulating data arrays located in GPR registers.
  - In this case, it is necessary to initialize FSR register with a starting address of the array, and the rest of the data can be accessed by incrementing the FSR register.



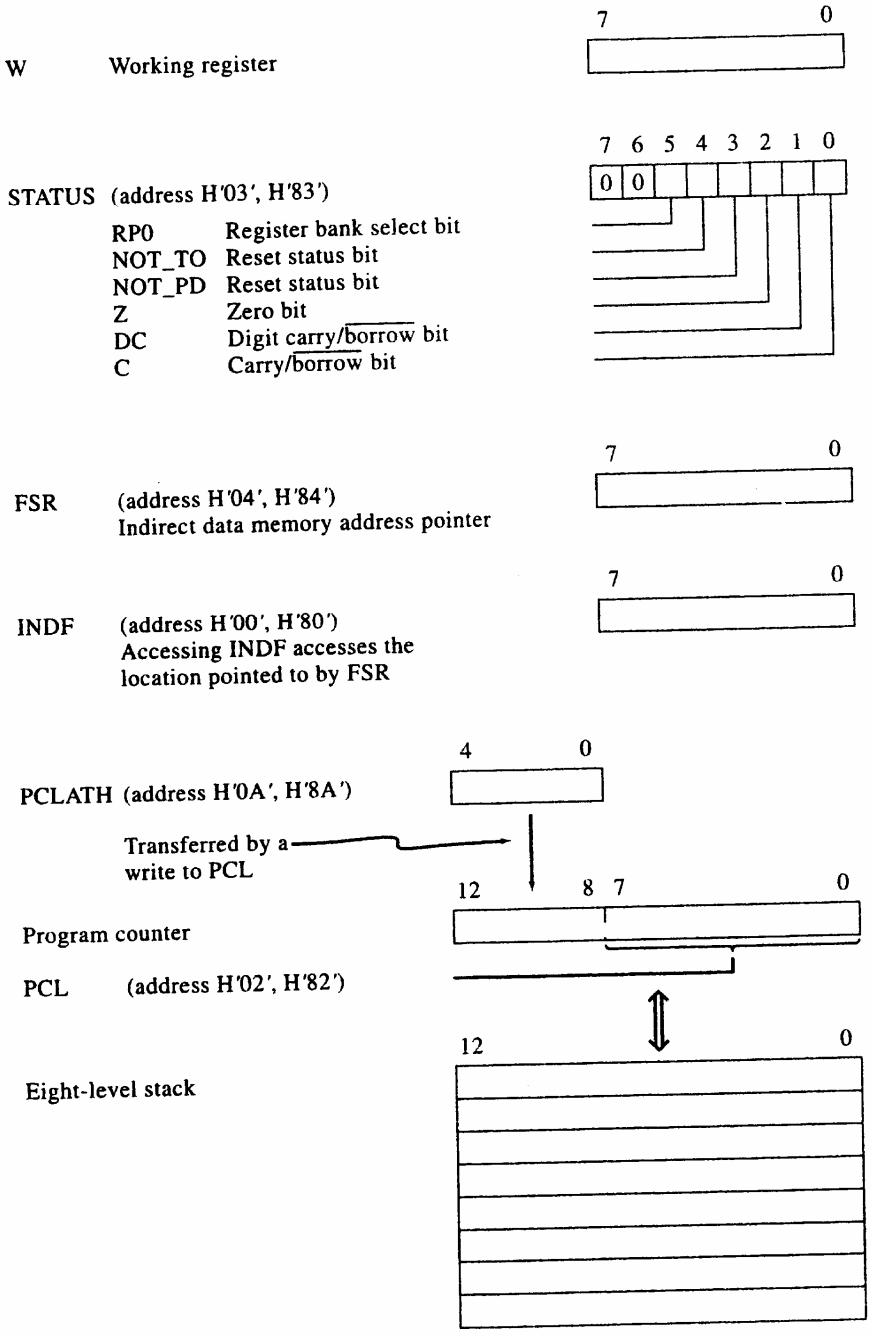
igure 7 **Indirect addressing mode.**

Software: Programmers Model



Some CPU Registers

- STATUS
- PC
- W
- PCL
- PCLATH



igure 8 CPU registers.

the accumulator

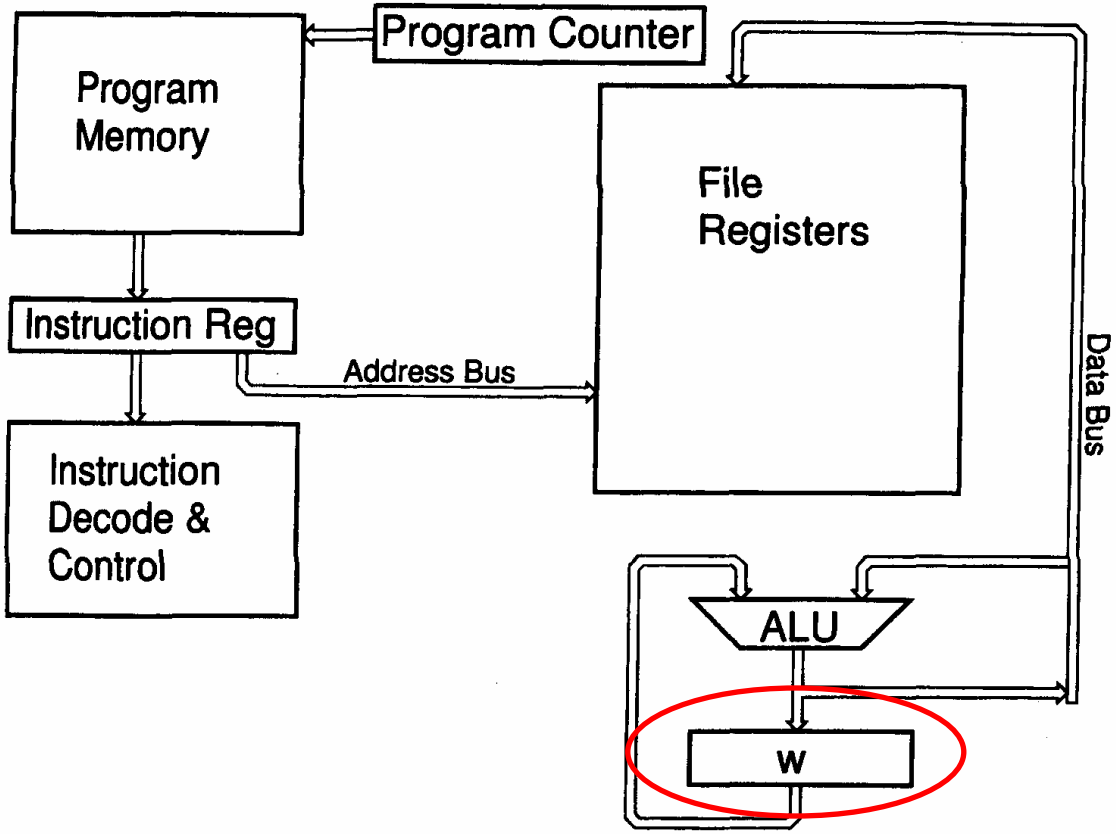
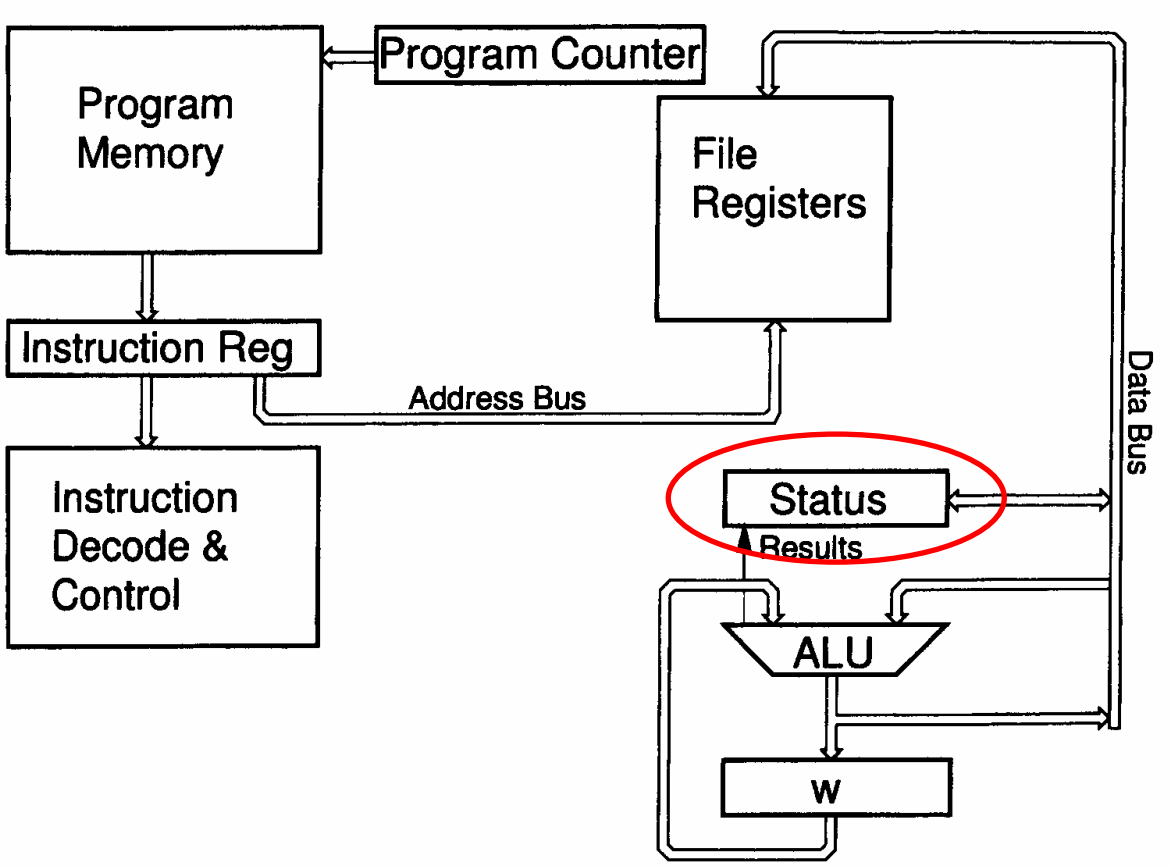


Figure 3-4 PICmicro® MCU Processor with "w" register as an "accumulator"

- to add two numbers together
  - first move the contents of one file register into the *w* register
  - then add the contents of the second file register to *w*
  - the result can be written to *w* or to the second file register

the status register



- the STATUS register stores ‘results’ of the operation
- three of the bits of the STATUS register are set based on the result of an arithmetic or bitwise operation

status register

- *three of the bits of the STATUS register are set based on the result of an arithmetic or bitwise operation*
  - *zero flag* ; this bit is set whenever the result of an operation is zero
  - *carry flag* ; this bit is set whenever the result of an operation is greater than 255 (0xFF) ; can be used to indicate that higher order bytes need to be updated
  - *digit carry flag* ; this bit is set whenever the least significant four bits of the result of an operation is greater than 15 (0x0F)

programming

- there are only 35 instructions

Instructions

- Data Movement
  - movf,movlw,movwf
- Arithmetic
  - addlw,addwf,sublw,subwf,incf,decf
- Logical
  - andlw,andwf,iorlw,iorwf,xorlw,xorwf,rrf,rlf,clrf,clrw,swapf,comf
- Bit Operators
  - bsf,bcf
- Branching
  - goto,btfss,btfsc,decfsz,incfsz
- Subroutine
  - call,return,retlw,retfie
- Misc.
  - sleep,clrwdt,nop

Instruction Set

- Every Instruction is coded in a 14 bit word
- Each instruction takes one cycle to execute
- Only 35 instructions to learn (RISC)

Instruction Set

- Uses 7 bits of 14 bit instruction to identify register file address
- For most instructions, W register is used as a source register
- The result of an operation can be stored back to the W register or back to source register

Mnemonic		Description	Operation	Flag	Cycle	Notes
		Data transfer				
MOVLW	k	Move constant to W	k → W		1	
MOVWF	f	Move W to f	W → f		1	
MOVF	f, d	Move f	f → d	Z	1	1,2
CLRW	-	Clear W	0 → W	Z	1	
CLRF	f	Clear f	0 → f	Z	1	2
SWAPF	f, d	Swap nibbles in f	f(7:4), (3:0) → f(3:0),(7:4)		1	1,2
		Arritmetic and logic				
ADDLW	k	Add constant and W	W+1 → W	C,DC,Z	1	
ADDWF	f, d	Add W and f	W+f → d	C,DC,Z	1	1,2
SUBLW	k	Subtract W from constant	W-k → W	C,DC,Z	1	
SUBWF	f, d	Subtract W from f	W-f → d	C,DC,Z	1	1,2
ANDLW	k	AND constant with W	W.AND.k → W	Z	1	
ANDWF	f, d	AND W with f	W.AND.f → d	Z	1	1,2
IORLW	k	OR constant with W	W.OR k → W	Z	1	
IORWF	f, d	OR W with f	W.OR.f → d	Z	1	1,2
XORLW	k	Exclusive OR constant with W	W.XOR.k → W	Z	1	1,2
XORWF	f, d	Exclusive OR W with f	W.XOR.f → d	Z	1	
INCF	f, d	Increment f	f+1 → f	Z	1	1,2
DECF	f, d	Decrement f	f-1 → f	Z	1	1,2
RLF	f, d	Rotate Left f trough carry		C	1	1,2
RRF	f, d	Rotate Right f trough carry		C	1	1,2
COMF	f, d	Complement f	f → d	Z	1	1,2
		Bit operations				
BCF	f, b	Bit Clear f	0 → f(b)		1	1,2
BSF	f, b	Bit Set f	1 → f(b)		1	1,2
		Directing a program flow				
BTFSC	f, b	Bit Test f, Skip if Clear	jump if f(b)=0		1 (2)	3
BTFSS	f, b	Bit Test f, Skip if Set	jump if f(b)=1		1 (2)	3
DECFSZ	f, d	Decrement f, Skip if 0	f-1 → d, jump if Z=1		1(2)	1,2,3
INCFSZ	f, d	Increment f, Skip if 0	f+1 → d, jump ifZ=0		1(2)	1,2,3
GOTO	k	Go to address	W.AND.k → W		2	
CALL	k	Call subroutine	W.AND.f → d		2	
RETURN	-	Return from Subroutine	W.OR k → W		2	
RETLW	k	Return with constant in W	W.OR.f → d		2	
RETFIE	-	Return from interrupt	W.XOR.k → W		2	
		Other instructions				
NOP	-	No Operation			1	
CLRWDT	-	Clear Watchdog Timer	0 → WDT, 1→TO,1 → PD	$\overline{TO}$,$\overline{PD}$	1	
SLEEP	-	Go into standby mode	0 → WDT, 1→TO,0→ PD	$\overline{TO}$,$\overline{PD}$	1	

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS						
ADDWF	f, d	Add W and f	1	00 0111 dfff ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00 0101 dfff ffff	Z	1,2
CLRF	f	Clear f	1	00 0001 lfff ffff	Z	2
CLRW	-	Clear W	1	00 0001 0xxx xxxx	Z	
COMF	f, d	Complement f	1	00 1001 dfff ffff	Z	1,2
DECF	f, d	Decrement f	1	00 0011 dfff ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00 1011 dfff ffff		1,2,3
INCF	f, d	Increment f	1	00 1010 dfff ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00 1111 dfff ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00 0100 dfff ffff	Z	1,2
MOVF	f, d	Move f	1	00 1000 dfff ffff	Z	1,2
MOVWF	f	Move W to f	1	00 0000 lfff ffff		
NOP	-	No Operation	1	00 0000 0xx0 0000		
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00 1110 dfff ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff	Z	1,2

For **byte-oriented** instructions, ’f’ represents a file register designator and ’d’ represents a destination designator. The file register designator specifies which file register is to be used by the instruction.

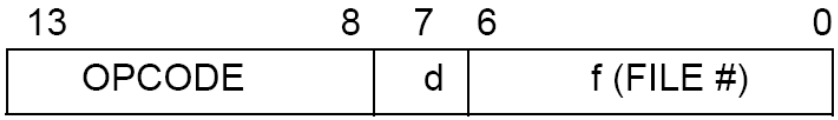
The destination designator specifies where the result of the operation is to be placed. If ’d’ is zero, the result is placed in the W register. If ’d’ is one, the result is placed in the file register specified in the instruction

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb LSb	Status Affected	Notes
BIT-ORIENTED FILE REGISTER OPERATIONS						
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff		1,2
BSF	f, b	Bit Set f	1	01 01bb bfff ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01 10bb bfff ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01 11bb bfff ffff		3
LITERAL AND CONTROL OPERATIONS						
ADDLW	k	Add literal and W	1	11 111x kkkk kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11 1001 kkkk kkkk	Z	
CALL	k	Call subroutine	2	10 0kkk kkkk kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00 0000 0110 0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10 1kkk kkkk kkkk		
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk	Z	
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk		
RETFIE	-	Return from interrupt	2	00 0000 0000 1001		
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk		
RETURN	-	Return from Subroutine	2	00 0000 0000 1000		
SLEEP	-	Go into standby mode	1	00 0000 0110 0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk	Z	

For **bit-oriented** instructions, ’b’ represents a bit field designator which selects the number of the bit affected by the operation, while ’f’ represents the address of the file in which the bit is located.

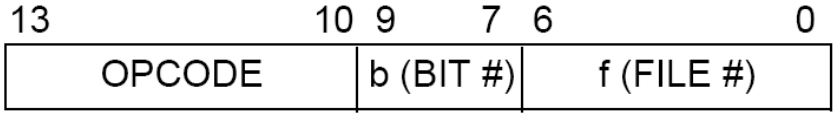
For **literal and control** operations, ’k’ represents an eight or eleven bit constant or literal value.

Byte-oriented file register operations



d = 0 for destination W
d = 1 for destination f
f = 7-bit file register address

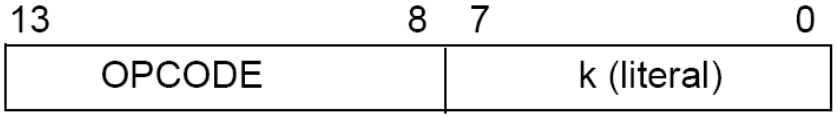
Bit-oriented file register operations



b = 3-bit bit address
f = 7-bit file register address

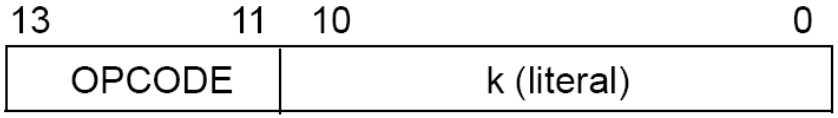
Literal and control operations

General



k = 8-bit immediate value

`CALL` and `GOTO` instructions only



k = 11-bit immediate value

addwf instruction

General form:

addwf *floc*, *d* *d*← [*floc*] + w

floc is a memory location in the file registers (data memory)

w is the working register

d is the destination, can either be the literal ‘f’ or ‘w’

[floc] means “the contents of memory location *floc*”

addwf 0x70,w w ← [0x70] + w

addwf 0x70,f [0x70] ← [0x70] + w

Move Commands:

movlw 0xF2 : stores the number 0xF2 into the W register

movwf 0x0C : stores the W register contents into file H'0C'

movf 0x0C,w : loads the contents of file H'0C' into W register

movf 0x0C,f : loads the contents of file H'0C' into file H'0C'

Bit Set/Clear Commands

```
 bcf     0x0C,0          :  clear the 0th bit of file H'0C'
bsf     0x0D,3          :  set the 3rd bit of file H'0D'
btfsc  0x42,0           :  test the 0th bit of the file H'42',  if it
                           is 0, then skip the next line of code.
btfss   0x43,1          :  test the 1st bit of the file H'43',  if it
                           is 1, then skip the next line of code.
```

- **Zero flag: Z**

  Set to 1 if the previous operation result was 0
- **Carry flag: C**

  Set to 1 if the previous result caused a carry or borrow out of the 8-bit word
- **Digit Carry flag: DC**

  Set to 1 if the previous command caused a half carry or borrow across bits 3 and 4.

PCLATH register

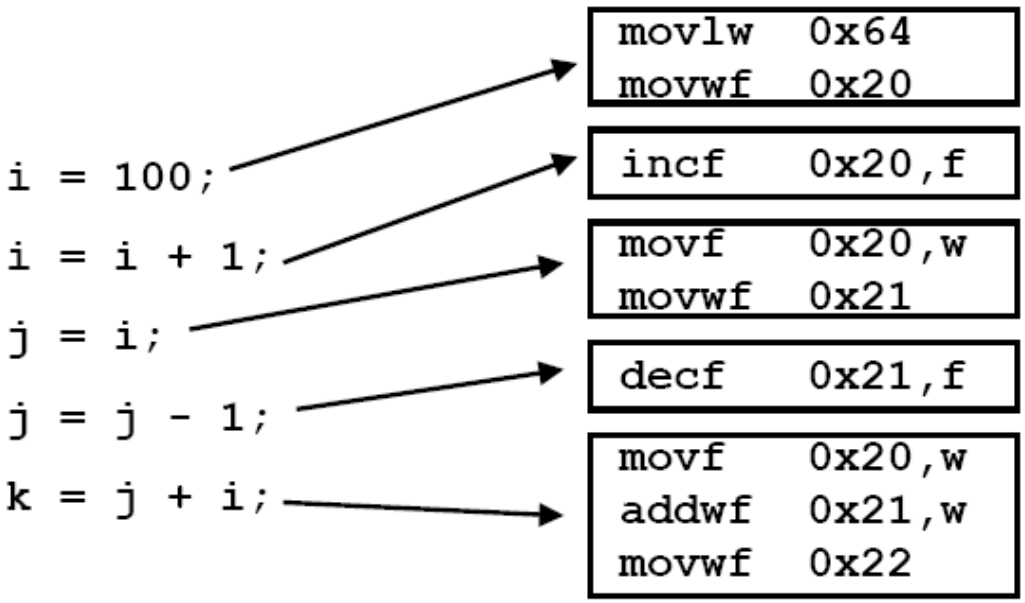
PCLATH is a special register located at 0x0A that is used by instructions that modify the PC register.

The PC register is 13 bits so programs can be a maximum of 8K (8192) instructions.

Instructions that affect the PC only change either the lower 8-bits or lower 11-bits; the remaining bits come from the PCLATH register.

If your program is less than 2K (2048) instructions, then you do not have to worry about modifying PCLATH before a *goto* because the PCLATH[4:3] bits will already be ’00’.

C to PIC Assembly



mptest.asm

```
 INCLUDE "p16f873.inc"

 ;  Register Usage
 CBLOCK 0x020   ;
  i, j,k    ; reserve space
 ENDC

myid equ  D'100'   ; define myid label

  org    0
  movlw  myid    ; w <- 100
  movwf  i       ; i <- w;

  incf   i,f     ; i <- i + 1

  movf   i,w      ; w <- i
  movwf  j       ; j <- w

  decf   j,f     ; j <- j - 1


  movf   i,w     ; w <- I
  addwf  j,w     ; w <- w + j
  movwf  k     ; k <- w

here
  goto   here      ; loop forever
  end
```

This file can be assembled by MPLAB into PIC machine code and simulated.

Labels used for memory locations 0x20 (i), 0x21(j), 0x22(k) to increase code clarity

mptst.asm (cont.)

```
INCLUDE "p16f873.inc"
```

Include file that defines various labels for a particular processor. This is an assembler directive, do not start in column 1. Only labels start in column 1.

```
;  Register Usage
 CBLOCK 0x020    ;
  i, j,k     ; reserve space
 ENDC
```

An assembler directive that reserves space for named variables starting at the specified location. Locations are reserved in sequential order, so *i* assigned 0x20, *j* to 0x21, etc. Use these variable names instead of absolute memory locations.

An ***assembler directive*** is not a PIC instruction, but an instruction to the assembler program.

mptst.asm (cont.)

```
myid equ  D'100'
```

An assembler directive that *equates* a label to a value. The D'100' specifies a decimal 100.

Could have also done:

myid equ .100
myid equ 0x64
myid equ H'64'

```
org    0
```

An assembler directive that specifies the starting location (*origin*) of the code after this statement. This places the code beginning at location 0x0000 in program memory. There must always be valid code at location 0 since the first instruction is fetched from here.

mptst.asm (cont.)

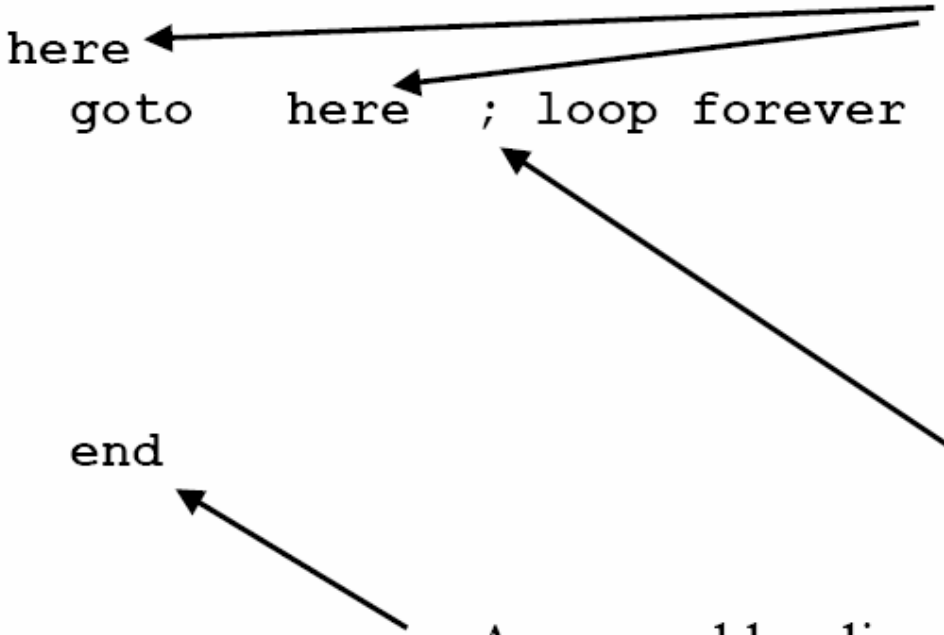
```
; i = 100;
movlw   myid      ; w <- 100
movwf   i         ; i <- w;

; i = i+1;
incf    i,f       ; i <- i + 1

;j = i
movf    i,w       ; w <- i
movwf   j         ; j <- w
```

The use of labels and comments greatly improves the clarity of the program.

It is hard to over-comment an assembly language program if you want to be able to understand it later.

Strive for at least a comment every other line; refer to lines

mptst.asm (cont.)



A label that is the target of a *goto* instruction. Labels must start in column 1, and are case sensitive (instruction mnemonics are not case sensitive.

A comment

An assembler directive specifying the end of the program. All assembly language programs must have an *end* statement.

Clock Cycles vs. Instruction Cycles

The clock signal used by a PIC to control instruction execution can be generated by an off-chip oscillator, by using an external RC network to generate the clock on-chip.

For the PIC 16F87X, the maximum clock frequency is 20 Mhz.

An **instruction cycle** is **four clock** cycles.

A PIC instruction takes 1 or 2 instruction cycles, depending on the instruction (see Table 13-2, pg. 136, PIC 16F87X data sheet).

An add instruction takes 1 instruction cycle. How much time is this if the clock frequency is 20 MHz (1 MHz = 1.0e6 = 1,000,000 Hz)?

1/frequency = period, 1/20 Mhz = 50 ns (1 ns = 1.0e-9 s)

Add instruction @ 20 Mhz takes 4 * 50 ns = 200 ns.

By comparison, a Pentium IV add instruction @ 3 Ghz takes 0.33 ns (330 ps). A Pentium IV could emulate a PIC faster than a PIC can execute! But you can’t put a Pentium IV in a toaster, or buy one from digi-key for $5.00.