Microcontroller (µC) vs. Microprocessor (µP)

- µC intended as a single chip solution, µP requires external support chips (memory, interface)
- µC has on-chip non-volatile memory for program storage, µP does not.
- µC has more interface functions on-chip (serial interfaces, Analog-to-Digital conversion, timers, etc.) than µP
- µC does not have virtual memory support (I.e, could not run Linux), while µP does.
- General purpose µPs are typically higher performance (clock speed, data width, instruction set, cache) than µCs
- Division between µPs and µCs becoming increasingly blurred

PIC 16F87x µC

Features	Comments
Instruction width	14 bits
On-chip program memory (non-volatile, electrically erasable)	Varies, up to 8K x 14 words
On-chip Random Access Memory (RAM)	Varies, up to 368 x 8
Clock speed	DC to 20 Mhz
Architecture	Accumulator, 35 instructions
On-chip modules	Async serial IO, I2C, SPI, A/D, 16-bit timer, two 8-bit timers

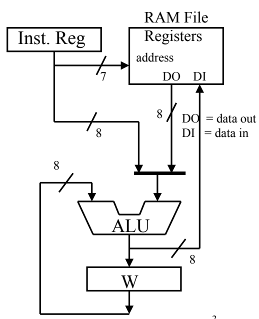
Accumulator-Based Instruction Set

Two operand instructions have the Working Register (*w* reg) as one operand, and memory or data in the current instruction as the second operand.

The destination can be either be *w* reg or file registers.

A register used in the manner of the *w* register is generally called an ***accumulator***.

The ***instruction register*** contains the machine code of the instruction currently being executed.



The *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

addwf Examples

Data Memory

Location	contents
0x58	0x2C
0x59	0xBA
0x5A	0x34
0x5B	0xD3

Assume data memory contents on right

w register contains 0x1D

Execute: addwf 0x59, w w ← [0x59] + w

w = [0x59] + w = 0xBA + 0x1D = 0xD7

After execution w = 0xD7, memory unchanged.

Execute: addwf 0x59, f [0x59] ← [0x59] + w

[0x59] = [0x59] + w = 0xBA + 0x1D = 0xD7

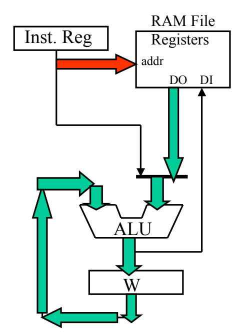
After execution [0x59] = 0xD7, *w* is unchanged.

addwf *floc*, w

Active data paths

Address/control

w is a source and destination.

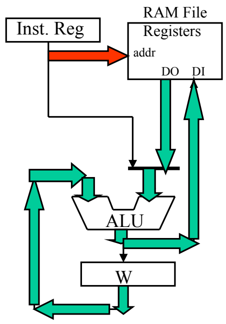


addwf *floc*, f

Active paths

Address/control

floc is a source and destination.



addwf instruction encoding

See page 136 in PICF87X datasheet

addwf *floc*, d

B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	1	0	1	1	1	d	f	f	f	f	f	f	f

'fffffff' lower 7-bits of *floc* address

'd': 0 = *w* reg, 1 = *f*

Machine code	Instruction
0x07D9	addwf 0x59, f
0x0759	addwf 0x59, w

Move literal to *w* (*movlw*)

The previous example assumed that *w* contained a value of 0x1D. How did this get into *w* in the first place ?

movlw *k*

w ← *k*

B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
1	1	1	0	0	0	k	k	k	k	k	k	k	k

"kkkkkkkk" 8-bit constant, loaded into w register

Machine code	Instruction
0x301D	movlw 0x1D

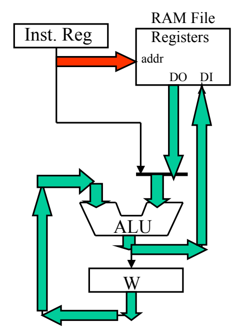
Note that the ***instruction word*** contains the 8-bit constant, not data memory.

movlw *k*

Active paths

Address/control

The ***instruction register*** contains the machine code of the instruction currently being executed. The *k* value is from the lower 8 bits of the instruction register.



Move *w* to f (*movwf*)

A common operation is to store *w* to a location in the file registers

movwf *floc*

[floc] ← *w*

B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	0	0	1	f	f	f	f	f	f	f

'fffffff' lower 7-bits of *floc* address

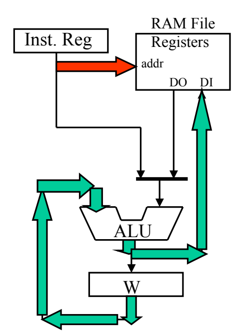
Machine code	Instruction
0x00D9	movfw 0x59

movwf *floc*

Active paths

Address/control

The *w* value passes through the ALU unchanged on its way to the RAM register file.



Increment (*incf*)

See page 136 in PICF87X datasheet

incf *floc*, d

Increment destination by 1

B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	0	1	0	1	0	d	f	f	f	f	f	f	f

'fffffff' lower 7-bits of *floc* address

'd': 0 = *w* reg, 1 = *f*

Machine code	Instruction	
0x0AD9	incf 0x59, f	;[0x59] ← [0x59] +1
0x0A59	incf 0x59, w	; w ← [0x59] + 1

Decrement (*decf*)

See page 136 in PICF87X datasheet

decf *floc*, d

Decrement destination by 1

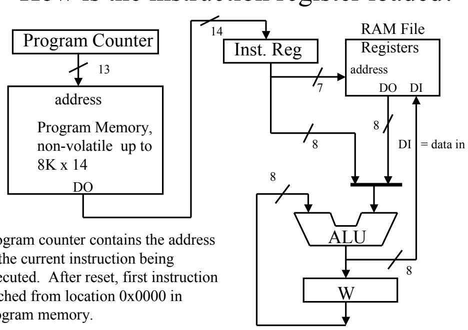
B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	1	1	d	f	f	f	f	f	f	f

'fffffff' lower 7-bits of *floc* address

'd': 0 = *w* reg, 1 = *f*

Machine code	Instruction	
0x03D9	decf 0x59, f	;[0x59] ← [0x59] -1
0x0359	decf 0x59, w	; w ← [0x59] - 1

How is the instruction register loaded?



Program counter contains the address of the current instruction being executed. After reset, first instruction fetched from location 0x0000 in program memory.

Goto location (*goto*)

The program counter specifies the location of the current location. How is this changed?

goto *k*

PC[10:0] ← *k*

B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
1	0	1	k	k	k	k	k	k	k	k	k	k	k

"kkkkkkkkkkk" lower 11-bits of a location , loaded into lower 11-bits of the program counter register (PC[10:0]); PC[12:11] ← PCLATH[4:3].

Machine code	Instruction
0x2809	goto 0x9

The next instruction is fetched from the target address.

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'.

A Simple Program

In this class, will present programs in *C* form, then translate (*compile*) to PIC assembly language.

A 'char' variable is 8-bits (1 byte)

```
C Program equivalent
  #define myid 100
  unsigned char i,j,k;


    i = myid;   /* myvalue = 100 */
    i = i + 1;   /* i++, i = 101 */
    j = i;       /* j is 101 */
    j = j - 1;   /* j--, j is 100 */
    k = j + i;    /* k = 201 */
```

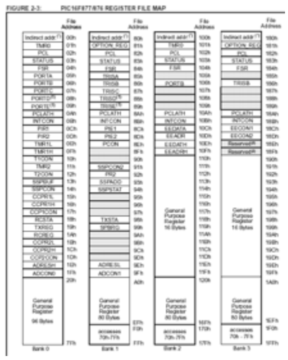
Where are variables stored?

The file registers are split into 4 banks, each bank has 128 locations (0x80).

Some locations are reserved for **special registers** (like the *w* register).

Easiest place to store values is bank0 (0x00-0x7F); free locations start at 0x20.

Assign *i* to **0x20**, *j* to **0x21**, and *k* to **0x22**. Other choices could be made.



C to PIC Assembly

`i = 100;`	`movlw 0x64` / `movwf 0x20`
`i = i + 1;`	`incf 0x20,f`
`j = i;`	`movf 0x20,w` / `movwf 0x21`
`j = j - 1;`	`decf 0x21,f`
`k = j + i;`	`movf 0x20,w` / `addwf 0x21,w` / `movwf 0x22`

Comments: The assembly language program operation is not very clear. Also, multiple assembly language statements are needed for one *C* language statement. Assembly language is more *primitive* (operations less powerful) than *C*.

PIC Assembly to PIC Machine Code

- Could perform this step manually by determining the instruction format for each instruction from the data sheet.
- Much easier to let a program called an ***assembler*** do this step automatically
- MPLAB Integrated Design Environment (IDE) is used to assemble PIC programs and simulate them
  - Simulate means to execute the program without actually loading it into a PIC microcontroller

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.)

```
here
  goto   here   ; loop forever

  end
```

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.

General MPLAB Comments

- See Experiment #2 for detailed instructions on installing MPLAB on your PC and assembling/simulating programs.
- The assembly language file must have the *.asm* extension and must be a TEXT file
  - Microsoft *.doc* files are NOT text files
  - MPLAB has a built-in text editor. If you use an external text editor, use one that displays line numbers (e.g. don't use notepad – does not display line numbers)
- You should use your portable PC for experiments 1-5 in this class, all of the required software is freely available.

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.

PIC18xx2

- Microchip has an extensive line of PIC microcontrollers, of which the PIC18xx2 is the most recent.
- During the semester, will contrast features of the PIC16F87x with those of the PIC18xx2.
- Do not assume that because something is done one way in the PIC16F87x, that it is the most efficient method for accomplishing that action.
- The datasheet for the PIC18xx2 is found on the class web site.

PIC16F87x vs. PIC18Fxx2

One word that can be used to describe the PIC16F87x Instruction Set Architecture (ISA) is '**small**'.   The small number of instruction types, small data size width, small instruction word size allows an implementation using a small number gates, resulting in a microcontroller that is very inexpensive to manufacture. This results in an **unconventional** instruction set, that is inefficient at many common operations. But doing things slowly is often good enough, if it is cheap enough.

The PIC18xx2 has a more **conventional** instruction set. Direct comparisons between the PIC18xx2 instructions and microcontroller instruction sets from Intel, Motorola, etc can be made. The PIC18xx2 ISA is **better** than the PIC16F87x, and these improvements will be discussed during the semester.

PIC16F87x vs PIC18Fxx2

Features	16F87x	18Fxx2
Instruction width	14 bits	16 bits, 4 instructions take 32 bits.
Program memory	Up to 8K x 14	Up to 16K x 16 words
Data Memory	Up to 368 x 8	Up to 1536 x 8
Clock speed	Max 20 Mhz	Max 40 Mhz
Architecture	Accumulator, 35 instructions	Accumulator, 75 instructions

Features in PIC18 not present in PIC16: 8x8 hardware multiplier, stack push/pop instructions, branch instructions, signed, better support for signed comparisons (V, N flags). Peripherals are essentially the same for both processors. Both processors take 4 clock cycles for 1 instruction cycle.

What do you need to know?

- Understand the operation of *movelw*, *addwf*, *incf*, *decf*, *goto* instructions
- Be able convert PIC assembly mnemonics to machine code and vice-versa
- Be able to compile/simulate a PIC assembly language program in MPLAB
- Understand the relationship between instruction cycles and machine cycles
- Trace active datapaths in architectural diagram during instruction execution.
- Be prepared to discuss PIC18, PIC16 major differences