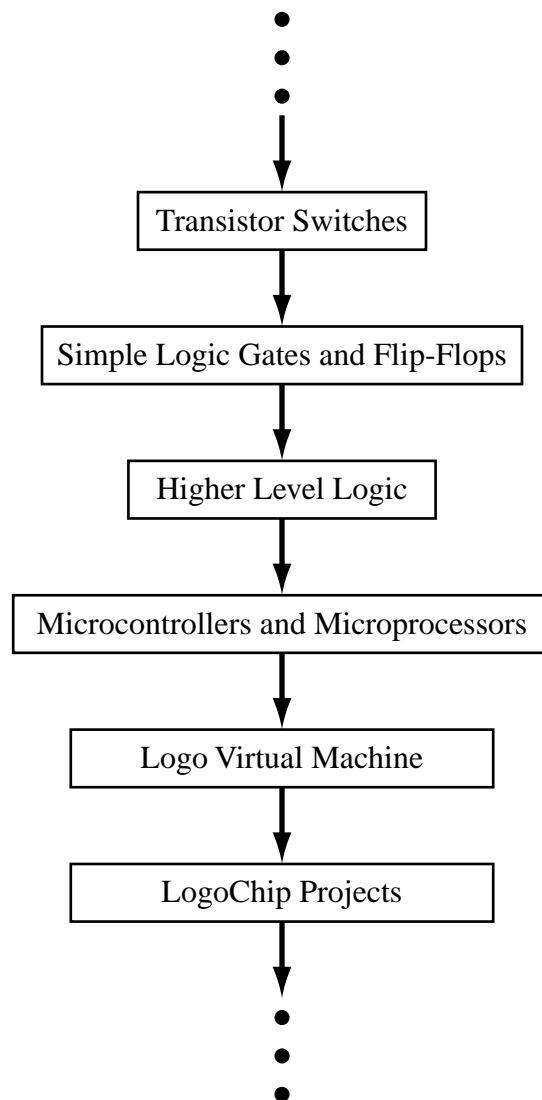


## Inside the LogoChip

<b>To Infinity and Beyond: The march towards increasing complexity and abstraction .....</b>	<b>2</b>
Assembly Language vs. Logo.....	3
<b>Lab Exercise: Digital Recording.....</b>	<b>3</b>
<b>Overview of the PIC16F876 microcontroller architecture.....</b>	<b>7</b>
<b>Data Memory Organization.....</b>	<b>8</b>
<b>The PIC16F876 Instruction Set and Assembly Language Mnemonics.....</b>	<b>10</b>
Sample instructions .....	11
<b>Hello World! (Assembly Language) .....</b>	<b>14</b>
<b>Lab Exercise: Take control of the indicator LED! .....</b>	<b>14</b>
Assembling and loading your program into the LogoChip.....	15
<b>Lab Exercise: Can you make your LED change color?.....</b>	<b>16</b>
Subroutines: Procedural abstraction in PIC assembly language .....	17
If...then...statements .....	17
If...then...else...statements .....	18
<b>Lab Exercise: Bootfash .....</b>	<b>20</b>
<b>Lab Exercise: Faster Digital Recording .....</b>	<b>21</b>
<b>Appendix: Summary of PIC 16F876 instruction set and Brian's Mnemonics .....</b>	<b>23</b>

**To Infinity and Beyond: The march towards increasing complexity and abstraction**

The idea of **abstraction** that has been a recurrent and central theme throughout our study of electronics this semester. Starting with single transistors and working all the way up through logic gates, flip-flops, memory and counters, arithmetic logic units, all the way to microcontrollers, we have repeatedly used an existing building block to construct more complicated and powerful structures. (To a somewhat lesser extent we saw the same thing in the analog portion of the course, where we first built amplifiers out of individual transistors, and later dealt with op amps.) Once we understand how our structures are built and operate we can then “abstract away,, their inner workings (that is, not worry so much about what is going inside) and use these new building blocks to make even more complicated (and powerful)

structures.

### Assembly Language vs. Logo

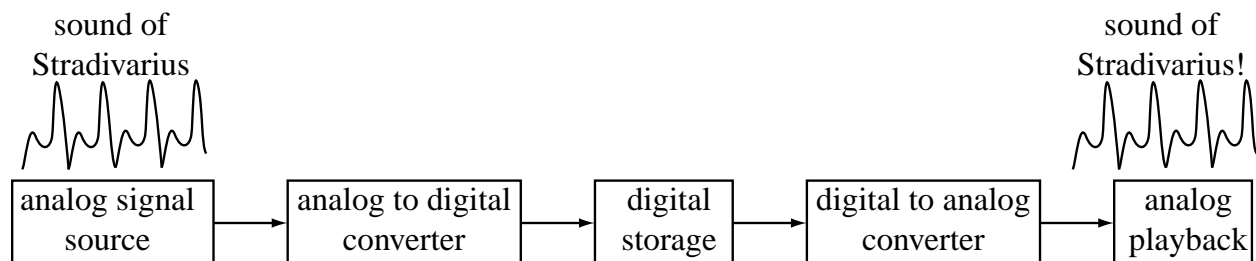
Microcontrollers such as the PIC16F876 are, as we shall see, directly capable of performing only a small set of elemental computations such as addition, subtraction, or setting and clearing bits in a register. But they can do each of these simple things very quickly (in about 200 ns in our case.) If you want a microcontroller to do things as efficiently (quickly) as possible, you generally want to program in **assembly language**, which makes direct use of these elemental capabilities. On the other hand its time consuming to write programs in assembly language that do complicated things. It's much quicker to write the program using a "higher level,, language like Logo. The downside of higher level languages is that they tend to run more slowly than a program written in assembly language. Thus there is a fundamental tradeoff between ease of use and speed of execution.

(modular furniture analogy)

	Logo	Assembly Language
Time per instruction	~70 $\mu$ sec	~200 nsec
High level primitives?	yes	no
Command Center?	yes	no

### Lab Exercise: Digital Recording

To appreciate this tradeoff, it is useful to have a specific instance in mind. Let's takes the important example of digital recording, outlined schematically below:

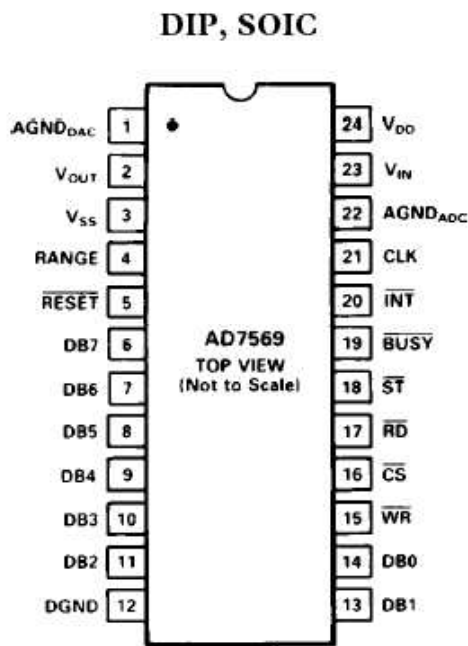


## The Holy Grail of Digital Recording

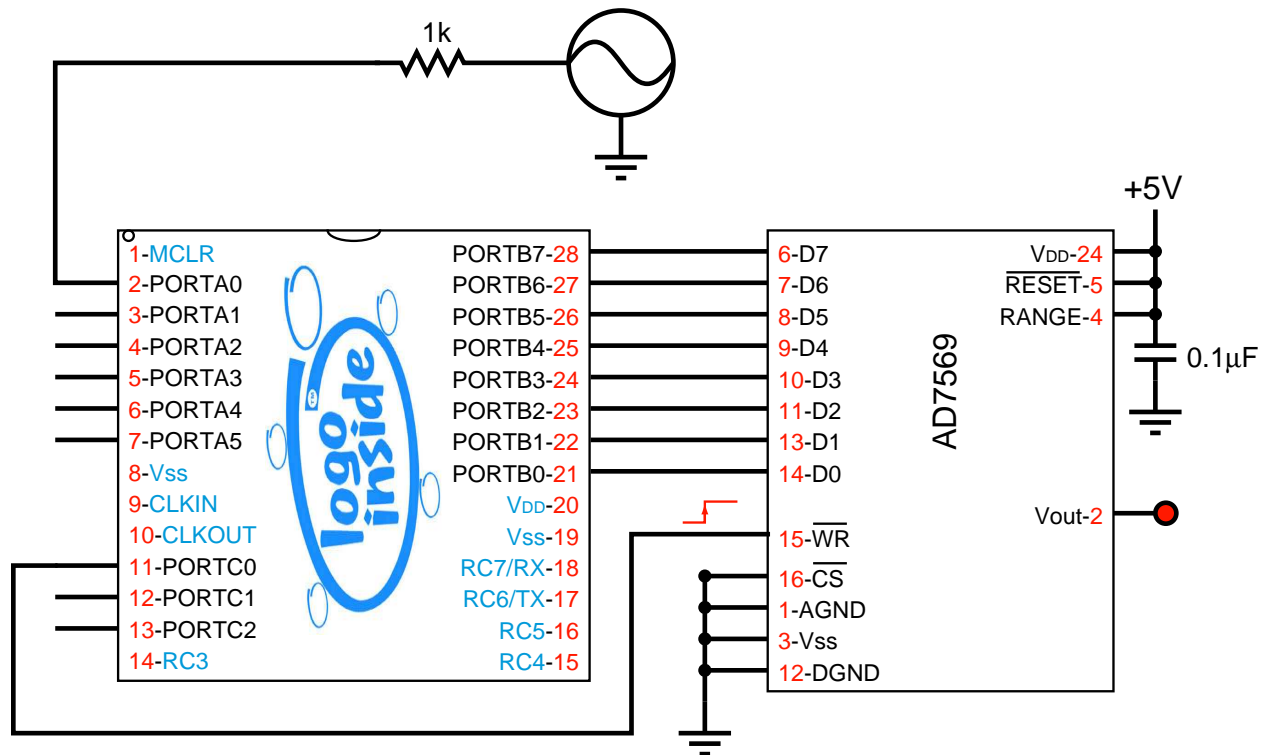
Of course the analog signal in question need not correspond to sound. It could be anything from an NMR signal to the output from a seismograph.

(See pp. 406 - 409 in *Student Manual* for more on this)

Let us start by trying to implement this scheme using a LogoChip (which, you will recall has a number of built-in 10-bit analog to digital converters) and an external 8-bit digital to analog converter, the AD7569. A pin out diagram for the AD7569 is shown below:



The wiring scheme for our digital recorder is shown in the figure below:



The analog source is a function generator. Since the analog voltages supplied to the LogoChip must lie between ground and the LogoChip's positive supply voltage (5 or 6 volts) it is important to use a function generator with an adjustable dc offset. **It is important to check to make sure that the voltages remain in this range; otherwise damage to the LogoChip is possible.** (The 1k resistor serves to limit currents and helps protect the LogoChip in the event the voltage lies outside the allowable range, but you shouldn't count on it to save you.)

The analog signal is fed into one of the LogoChip's analog inputs. The following code should result in a replica of the original signal being produced at the DAC's analog output.

```

constants
  [[portb 6][portb-ddr $86]
   [portc 7][portc-ddr $87]]

to sample-signal
write portb-ddr 0           ; make all of portb into
                             outputs

clearbit 0 portc-ddr

loop [
  write portb ((read-ad 0) / 4)
                             ; do an analog to digital
                             conversion on the signal that

```

```

        is fed into A0, then write
        the 8 most significant bits
        of this 10-bit number to
        portb
        ; strobe the DAC's WR* line
        to update the DAC's analog
        output
    ]
to strobe
    ; the WR* line is activated
    by a rising edge
    clearbit 0 portc
    setbit 0 portc
end

```

Note that this particular DAC requires a “strobe pulse,, in order to update it’s analog output. Updating will occur on the rising edge supplied to the WR\* pin. (Apparently the AD7569 contains a byte’s worth of edge-triggered flip=flops or “latches,, at its digital inputs.)

**1) Try this, using an oscilloscope to monitor the DAC’s output.** How does the “replica signal,, having through one conversion into the digital world followed by a second conversion back to the analog world, compare to the original signal.

**2) Try supplying the recorder with sine waves of various frequencies.** What’s the highest frequency for which you can obtain an accurate representation? Do you see what the LogoChip is really too slow to act even as an audio digital recorder? The situation may not be quite as bad as you first think due to **Nyquist’s Sampling Theorem**, which states all you really need to do is make at least two “samples,, per period. (It’s no accident that the 44 kHz sampling rate used in audio CDs is, roughly, twice the upper frequency cutoff of human hearing.) Can you see the effects of **aliasing** when the period of your analog signal becomes comparable to or smaller than the time between samples?

**3) Try constructing a suitable RC filter to clean up the step-like artifacts introduced by this process.**

To do better we need to sample more frequently, which means we need a faster computer. Actually the hardware in the LogoChip is up to the task, The problem lies with the Logo virtual machine, which requires an eternity (~100  $\mu$ sec!) to execute a single instruction. The simplest solution is to use a program written



Highlights of the PIC18F876 microcontroller architecture include:

- At the heart of the microcontroller is an **arithmetic logic unit (ALU)** that allows the controller to sequentially execute any one of a set of **instructions** in an order dictated by a **stored program**. (See the section entitled “How do you get from logic gates to higher level computation?,, at the end of the combinational logic notes for an outline of how to build a simple ALU out of logic gates)
- RISC (Reduced Instruction Set Computing) architecture. This means there are relatively few machine level instructions (□□□□□□□□ instructions).
- *internal* 8k x 14-bit EEPROM<sup>1</sup> program memory; each instruction occupies a single 14-bit “word,, and most execute in a single machine cycle. One “machine cycle,, takes 4 “clock cycles,,. For a 20 MHz clock this corresponds to 200 ns per instruction or 5 million instructions per second (5 MIPS).
- 8-bit wide data path; 8-bit data bus is separate from 14-bit instruction bus. (13-bit wide) address bus, instruction bus, and data bus are all internal to the chip. This means, as you already know from using the LogoChip, that there is *much* less wiring that needs to be done and *very* few external parts.
- one featured **working register** or **accumulator** is used during most instructions.
- a whopping 368 bytes of internal random access memory (RAM) for “temporary data storage,,.

## Data Memory Organization

As shown in the figure on the next page, the data memory in the PIC16F876 is organized into a series of 8-bit **registers**. These registers are of two distinct types. The first are the **special function registers** some of which should look familiar

---

<sup>1</sup> electrically erasable programmable read only memory. This kind of memory is *non-volatile*; it retains data indefinitely (for decades!) even when power is shut off. eeprom is often much slower to write to than RAM (These days it can take as much as a few milliseconds to write each byte, but it can be read just as fast as RAM. But new technologies that are now coming on line shorten the write times by orders of magnitude.)

from your earlier LogoChip experience. For example note the PORTA, PORTB, and PORTC **data registers**. The registers labeled TRISA, TRISB, and TRISC are the corresponding **data direction registers**, which, you will recall, determine whether a pin is an output or an input. (I suppose that “TRIS,, stands for “tri-state,,) The second type of register, referred to in the figure as **general purpose registers** are examples of **random access memory (RAM)** that can be used for temporary data storage.

						File Address				
Indirect addr. <sup>(*)</sup>	00h	Indirect addr. <sup>(*)</sup>	80h	Indirect addr. <sup>(*)</sup>	100h	Indirect addr. <sup>(*)</sup>	180h			
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h			
PCL	02h	PCL	82h	PCL	102h	PCL	182h			
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h			
FSR	04h	FSR	84h	FSR	104h	FSR	184h			
PORTA	05h	TRISA	85h		105h		185h			
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h			
PORTC	07h	TRISC	87h		107h		187h			
PORTD <sup>(*)</sup>	08h	TRISD <sup>(*)</sup>	88h		108h		188h			
PORTE <sup>(*)</sup>	09h	TRISE <sup>(*)</sup>	89h		109h		189h			
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah			
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh			
PIR1	0Ch	PIE1	8Ch	EEDATA	10Ch	EECON1	18Ch			
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2	18Dh			
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved <sup>(*)</sup>	18Eh			
TMR1H	0Fh		8Fh	EEADRH	10Fh	Reserved <sup>(*)</sup>	18Fh			
T1CON	10h		90h		110h		190h			
TMR2	11h	SSPCON2	91h	General Purpose Register 16 Bytes	111h	General Purpose Register 16 Bytes	191h			
T2CON	12h	PR2	92h		112h		192h			
SSPBUF	13h	SSPADDD	93h		113h		193h			
SSPCON	14h	SSPSTAT	94h		114h		194h			
CCPR1L	15h		95h		115h		195h			
CCPR1H	16h		96h		116h		196h			
CCP1CON	17h		97h		117h		197h			
RCSTA	18h	TXSTA	98h		118h		198h			
TXREG	19h	SPBRG	99h		119h		199h			
RCREG	1Ah		9Ah		11Ah		19Ah			
CCPR2L	1Bh		9Bh		11Bh		19Bh			
CCPR2H	1Ch		9Ch		11Ch		19Ch			
CCP2CON	1Dh		9Dh		11Dh		19Dh			
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh			
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh			
	20h		A0h				120h		1A0h	
General Purpose Register 96 Bytes	7Fh	General Purpose Register 80 Bytes	EFh	General Purpose Register 80 Bytes	16Fh	General Purpose Register 80 Bytes	1EFh			
								accesses 70h-7Fh	accesses 70h-7Fh	accesses 70h-7Fh
								F0h	170h	1F0h
			FFh		17Fh		1FFh			
Bank 0		Bank 1		Bank 2		Bank 3				

### The PIC16F876 Instruction Set and Assembly Language Mnemonics

Once every 4 clock cycles the microcontroller will execute a new **instruction**. The PIC 16F876 microcontroller instructions consist of 14-bit **words**. There are about

45 separate instructions that this particular microcontroller uses.

The 14 bit words, which are sometimes referred to as “machine code,, generally consist of an OPCODE part, which specifies which type of instruction it is, and one or two OPERANDs, which provides specific information to be used in the instruction.

It would be tedious and foolish to program directly in machine language, Instead we will rely on a computer-based **assembler** that will automatically translate our **assembly language mnemonics** into **machine language** and then automatically download the machine code into the PIC’s EEPROM.

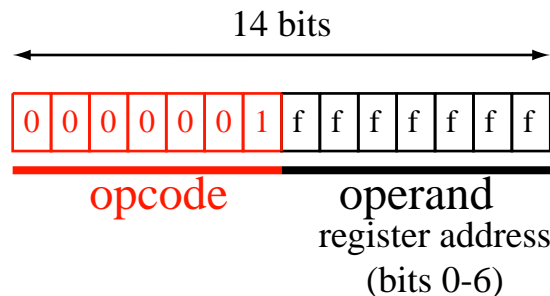
**Sample instructions**

To give you an idea of what these instructions are like, a few representative examples are shown below. A complete listing is summarized in the appendix at the end of this document entitled *Summary of PIC 16F876 Instruction Set and Brian’s Mnemonics*.<sup>2</sup>

**The “store accumulator” instruction.** The contents of the accumulator are loaded into register with an address “f,, (The value of the accumulator remains unchanged.) Since there are 384 possible register addresses in the PIC16F876 it takes a 9-bit number to specify the address. The 7 least significant bits of the address are encoded in the instruction itself while the two most significant bits are determined by the contents of bits 5 and 6 of the status register.

**status flags affected:** Z

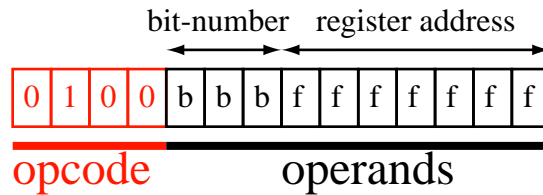
**assembly language mnemonic:** [ sta x ] where “x,, is the 7 “low order,, bits of the register address



**Clear a bit in a particular register.**  $b^{\text{th}}$  bit of register  $f$  is cleared (made equal to \_\_\_\_\_)

<sup>2</sup> The assembler was developed by Brian Silverman, at the MIT Media Lab.

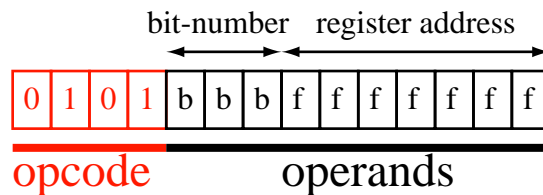
“0,,)



clear bit *b* of file *f*

**assembly language mnemonic:** [bclr b f]

**Set a bit in a particular register.** *b*<sup>th</sup> bit of register *f* is cleared (made equal to “0,,)



set bit *b* of file *f*

**assembly language mnemonic:** [bset b f]

**Decrement, skip if zero** - decrement the contents of register *f* and store the result in register *f*. If the result is zero, skip the next instruction.

**status flags affected:** none

**assembly language mnemonic:** [decsz f]

**Load accumulator with number** - The number *x* is loaded into the accumulator.

**status flags affected:** none

**assembly language mnemonic:** [ldan x]

**Exclusive or** - bitwise “xor,, the accumulator with register *f*, storing the result in the accumulator

**status flags affected:** Z, C

**assembly language mnemonic:** [xor f]

**Add a number** - add the contents of *f* to the accumulator, storing the result in the

accumulator

**status flags affected:** Z, C

**assembly language mnemonic:** [ addn f ]

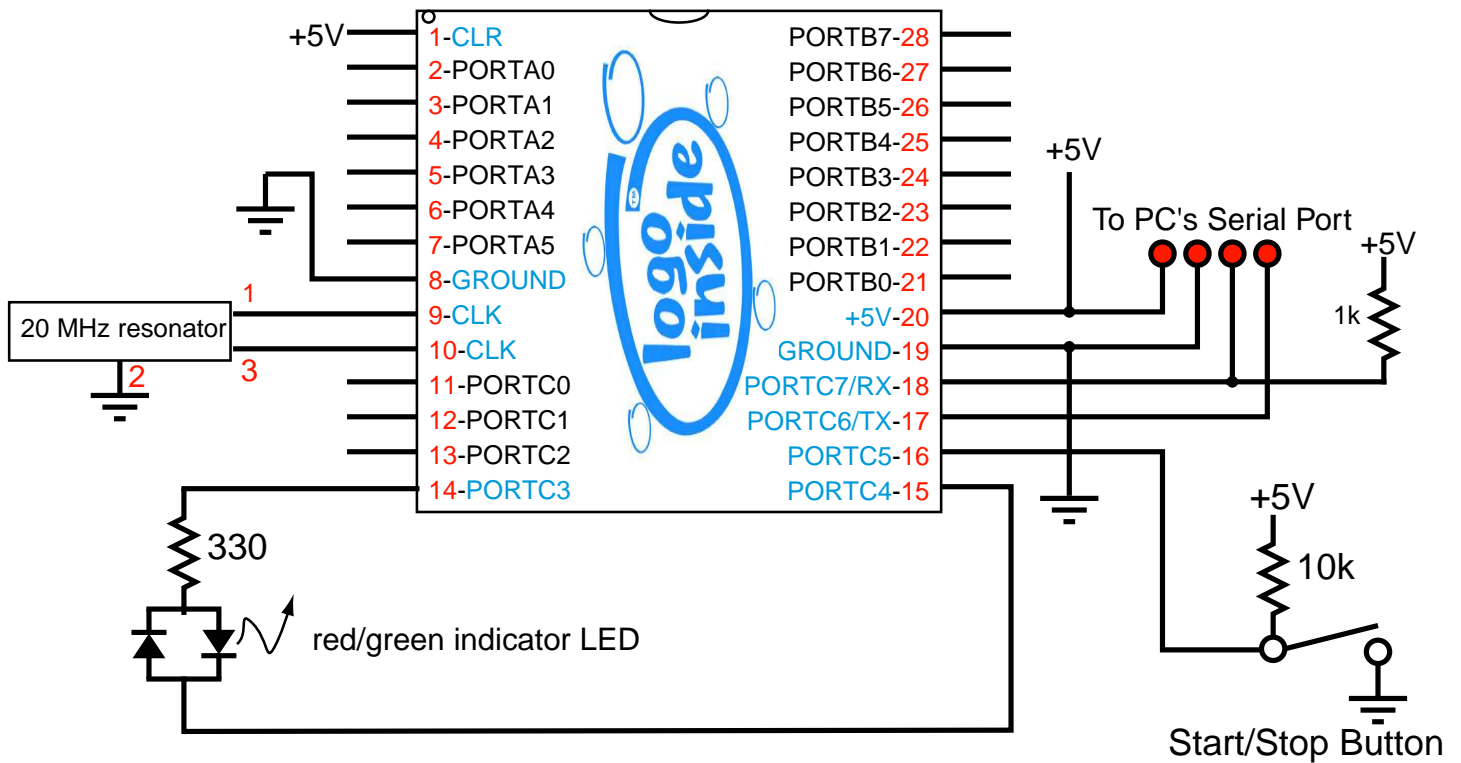
**Unconditional branch** - *Address* is a number ranging from 0 to 8191. The value of *address* is loaded into the “program counter,, so that the next instruction executed is the one located at program memory location *address*.

**status flags affected:** Z, C

**assembly language mnemonic:** [ bra *address* ]

Hello World! (Assembly Language)

LogoChip Hardware



**Lab Exercise: Take control of the indicator LED!**

The C3 and C4 pins must first be configured as an outputs and the value of the output set HIGH. In this example, as in LogoChip Logo, we use some constant definitions to make the code much more readable:

```
[const status 3]
    [const bankl 5] [const bankh 6]
[const portc 7]

[bset bankl status]           ; set bit 5 of the STATUS
                              register
[bclr bankh status]          ; clear bit 6 of the STATUS
                              register
                              ;so that register bank #1 is
                              selected
```

```

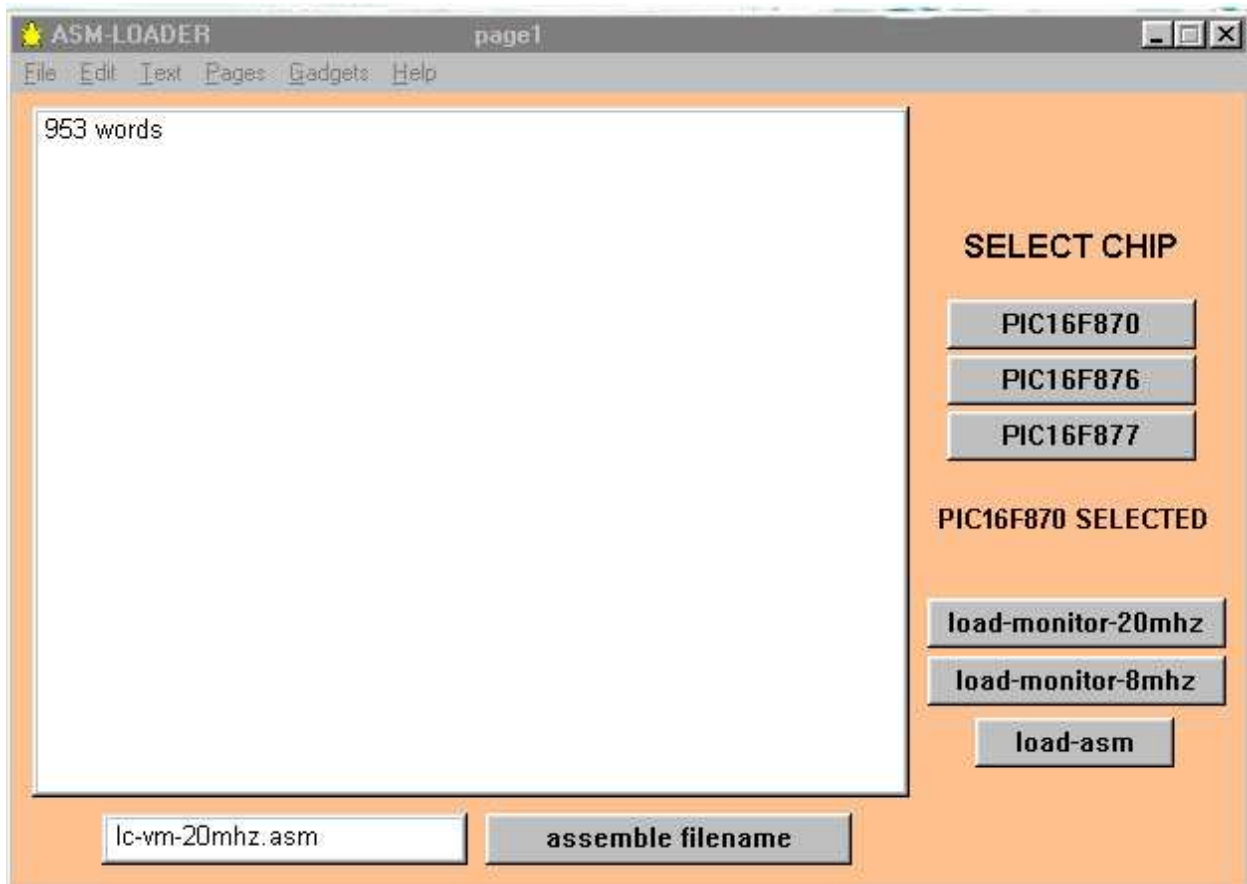
[bclr 3 portc]           ; turn C3 into an output
[bclr 4 portc]           ; turn C4 into an output
[bclr bank1 status]     ;clear bit 5 of the STATUS
                        ;register ;so that bank #0 is
                        ;selected

continue
[bset 3 portc]           ;set c3
[bclr 4 portc]           ;clear c4
                        ;to turn on the LED

[bra continue]          ; put your program in a loop
                        ;to keep it from running off
                        ;into oblivion

```

### Assembling and loading your program into the LogoChip



Here's how you can use your LogoChip to run assembly language programs.

- Save your assembly language program as a text file in the folder named VM
- Open up the ASM-LOADER.mw2 program

- Type the name of the text file that contains your assembly language program in the small text box in the lower left hand corner of the screen.
- Click on the “assemble filename,, button to convert your program into machine language. If error messages appear in the main text box, fix your program before proceeding.
- Select the PIC16F876 chip
- Turn on the power switch while holding down the LogoChip’s start/stop button. This places the LogoChip in “bootstrap mode,, so that it can accept a new assembly language program (overwriting the Logo virtual machine in the process).
- Click on the “load-asm,, button to download your program.
- Turn the power switch off, and wait a second.
- Turning the power switch back on should start your assembly language program running

**Lab Exercise: Can you make your LED change color?**

(Note: If at some point you want to change your hardware back into a LogoChip, you’ll need to reinstall the Logo virtual machine, which is simply a particular assembly language program that resides in the file named `lc-vm-20mhz.asm`. You load this assembly program just as you would any other, following the procedures described above.)

## Key Ideas in Assembly Language

### Subroutines: Procedural abstraction in PIC assembly language

A very powerful idea that is highlighted in LogoChip Logo is the notion of **procedural abstraction**, which is yet another example of abstraction that we have encountered in our travels. The way procedural abstraction is implemented in assembly language is quite clumsy compared to Logo. The PIC instruction set contains two critical instructions for implementing procedures:

[ *bsr address* ] - **Branch to subroutine**. This instruction first “pushes the return address onto the address stack,, then program execution branches to memory location *address*.”

[ *rts* ] - **Return from subroutine**. “Pops the address stack,, and places the result in the program counter, causing program execution to branch back to the line following the procedure call.”

This scheme allows procedures to be called from within procedures and enables the programmer to define procedures made out of the PIC instruction set. *Even better (and that’s putting it mildly!) once a procedure is built, it can then be used in the construction of an even “higher level” procedure, allowing the programmer to create ever more powerful building blocks.* In practice ones ability to build more complicated procedures out of simpler ones is limited by the “depth of the stack,, which is “8 levels deep,, in the case of the PIC.

### If...then...statements

To see how procedures work, let’s consider the specific case of the “**if...then**” statement. Suppose you wanted to repeatedly test a digital sensor (such as a switch or “touch sensor,,) and execute a procedure only when the sensor is pressed). In words what we seek to implement is:

If the touch sensor is pressed then do procedure1

For the sake of definiteness, let’s further suppose that the touch sensor is wired to pin RB0 of the PIC and is wired in such a way that when it is pressed RB0 reads a LOW input and when it is not pressed it reads a HIGH input. In PIC assembly language we can implement the “if...then,, construct with the following code:

```
[const portb 6]           ;portb is register #6
[const sensor 0]         ;the touch sensor is wired to
```

```

                                bit #0
                                ;of portb
[const status 3]                ;the status register is
                                register #3
[const zero 2]                  ;the zero flag is bit #2 of
                                the
                                ;status register

                                [lda portb]                ;check if touch sensor is
                                pressed
                                [andn 1]
                                [btsc zero status]
                                [bsr procedure1]

continue

procedure1
    [procedure1 instructions....
                                .....
                                ...]
                                [rts]

```

A key point is that procedure1 can be built out of other procedures that the programmer has defined.

**If...then...else...statements**

Here’s a more sophisticated challenge. Suppose you wanted to test a digital sensor (such as a switch or “touch sensor,”) and execute one of two procedures based on the result of the test. In words what we seek to implement is:

If the touch sensor is pressed then do procedure1, otherwise, do procedure2

In PIC assembly language we can implement the “if...then...else,, construct with the following code:

```

[const portb 6]                  ;portb is register #6
[const sensor 0]                 ;the touch sensor is wired to
                                bit #0
                                ;of portb
[const status 3]                ;the status register is
                                register #3
[const zero 2]                  ;the zero flag is bit #2 of
                                the

```

```

;status register

[lda portb]           ;check if touch sensor is
                      ;pressed
[andn 1]              ;use a "bit mask" to test the
                      ;value of the zeroth bit

[btsc zero status]
[bra procedure1]
[bra procedure2]

continue

procedure1
[procedure1 instructions....
.....]
[bra continue]

procedure2
[procedure2 instructions....
.....]
m [bra continue]

```

This code works, but it does not allow us to take full advantage of the power of procedural abstraction because, the way it is written, procedure1 and procedure2 cannot be readily used by other parts of the program. (Do you see why? It's because the procedures end with unconditional branches back to continue, which is one particular point in the program.) A better approach is the following:

```

[const portb 6]       ;portb is register #6
[const sensor 0]     ;the touch sensor is wired to
                      ;bit #0 of portb

[const status 3]     ;the status register is
                      ;register #3
[const zero 2]       ;the zero flag is bit #2 of
                      ;the status register
[const t0 $20]       ;temporary storage byte for
                      ;"remembering" the results of
                      ;the test

[lda portb]          ;check if touch sensor is
                      ;pressed
[andn 1]              ;use a "bit mask" to test the
                      ;value of the zeroth bit
[sta t0]             ;store result in t0

```

```

;if t0 is zero then do procedure1, if t0 is not zero then
;do procedure2
    [lda t0] [btsc zero status]
    [bsr procedure1]
    [lda t0] [btss zero status]
    [bsr procedure2]

continue
    [rest of main program....]

procedure1
    [procedure1 instructions....
    .....]
    [rts]

procedure2
    [procedure1 instructions....
    .....]
    [rts]

```

### Lab Exercise: Bootflash

Can you figure out how to make the LED blink on and off using the following “delay subroutine,, which causes a quarter second delay?

```

[const t0 $20]                ;temporary storage byte in
                                RAM

qsec [ldan 250]
    [sta t0]
    [clra]
qs20 [bsr delay-loop]
    [decsz t0][bra qs20]
    [rts]

; delay for a usec
delay-loop
    [addn -1]
    [btss z status]
    [bra delay-loop]
    [rts]

```

Can you make your LED do a bootflash?

**Lab Exercise: Faster Digital Recording**

- Try using the assembly language code below in your digital recorder.
- Use a digital scope to measure the sampling rate. What's the highest frequency signal you can observe without aliasing?
- Try using the output from your microphone as a signal source. (I found that the microphone circuit is very susceptible to “digital switching noise,, on the power supply lines generated by the microcontroller and the digital to analog converter. To minimize this problem I used a separate battery pack to power the microphone, so that the microphone circuit shares only a common ground line with the main circuit.) How well does your digital recorder do?

**Assembly language code for digital recording**

```

[const status 3]
    [const z 2][const bank 5]
[const portb 6] [const portc 7]
[const adresh $1e]           ;bank 0
[const adcon $1f]           ;bank 0
[const adcon1 $1f]          ;bank 1
    [const adon 0][const adgo 2]

    [bsr io-init]           ; initialize
loop
    [bsr get-sensor]        ;do an a/d on channel 0
    [lda adresh]
    [sta portb]             ;write result to DAC
    [bset 0 portc]
    [bclr 0 portc]         ;strobe the DAC's write line
    [bra loop]

get-sensor
    [bset adgo adcon]       ;start the conversion
sens20
    [btsc adgo adcon][bra sens20] ;wait until done
    [rts]

delay-loop                   ; delay for a number of usecs
    [addn -1]
    [btss z status]

```

```
[bra delay-loop]
[rts]

io-init
  [bset bank status]           ;switch to bank 1
  [ldan $0][sta adcon1]       ;set porta to analog (except
                               ra4), left justify adresh
  [bclr 0 portc]              ; set c0 to output
  [ldan $0][sta portb]        ; set portb to outputs
  [bclr bank status]          ;switch back to bank 0
  [bclr 0 portc]              ;set initial state of
                               outputs to LOW

  [ldan $0][sta portb]
  [ldan $81]
  [sta adcon]                 ;turn on channel 0 of
                               converter

  [ldan 20]
  [bsr delay-loop]           ; wait for 20 usec
                               acquisition time

[rts]
```

**Appendix: Summary of PIC 16F876 instruction set and Brian's Mnemonics<sup>3</sup>**

The PIC 16F876 microcontroller instructions consist of 14-bit words. There are about 45 separate instructions that the PIC uses. These are summarized below.

**Rating System:**

\* - rarely used

\*\* - sometimes used

\*\*\* - frequently used

**Adding or Subtracting (7)**

[ add f ] \*\*\*

**add** - add the contents of register  $f$  to the contents of the accumulator and store the result in the accumulator

status flags affected Z, C

[ addm f ] \*\*

**add to memory** - add the contents of the accumulator to the contents of register  $f$  and store the result in register  $f$

status flags affected Z, C

[ addn f ] \*\*\*

**add number** - add the contents of  $f$  to the accumulator, storing the result in the accumulator

status flags affected Z, C

[ sub f ] \*

**subtract** - subtract (2's complement method) the contents of the accumulator from the contents of register  $f$  and store the result in the accumulator

status flags affected Z, C (C=0 means result is negative)

[ subm f ] \*

**subtract memory** - subtract (2's complement method) the contents of the accumulator from the contents of register  $f$  and store the result in register  $f$

status flags affected Z, C (C=0 means result is negative)

[ subn f ] \*

---

<sup>3</sup> As in Brian Silverman.

**subtract number** - subtract (2's complement method) the contents of the accumulator from the number  $f$  and store the result in the accumulator  
status flags affected Z, C (C=0 means result is negative)

[dec  $f$ ] \*\*

**decrement** - decrement the contents of register  $f$  and store the result in register  $f$   
status flags affected Z

### Logical Operations (10)

[and  $f$ ] \*\*\*

**and** - bitwise “and,, accumulator with register  $f$ , storing the result in the accumulator  
status flags affected Z, C

[andm  $f$ ] \*\*

**and memory** - bitwise “and,, accumulator with register  $f$ , storing the result in the register  $f$   
status flags affected Z

[andn  $f$ ] \*\*\*

**and number** - bitwise “and,, the accumulator with the number  $f$ , storing the result in the accumulator  
status flags affected Z

[or  $f$ ] \*\*

**or** - bitwise “or,, accumulator with register  $f$ , storing the result in the accumulator  
status flags affected Z, C

[orm  $f$ ] \*

**or memory** - bitwise “or,, accumulator with register  $f$ , storing the result in the register  $f$   
status flags affected Z

[orn  $f$ ] \*\*

**or number** - bitwise “or,, the accumulator with the number  $f$ , storing the result in the accumulator  
status flags affected Z

[xor f] \*\*

**exclusive or** - bitwise “xor,, accumulator with register  $f$ , storing the result in the accumulator

status flags affected Z, C

[xorm f] \*

**exclusive or memory** - bitwise “xor,, accumulator with register  $f$ , storing the result in the register  $f$

status flags affected Z

[xorn f] \*\*

**exclusive or number** - bitwise “xor,, the accumulator with the number  $f$ , storing the result in the accumulator

status flags affected Z

[com f] \*

**complement** - complement (“invert,,) the contents of register  $f$  and store the result in register  $f$

status flags affected Z

### Moving and Loading Registers (13)

[bclr b f] \*\*\*

**bit clear** - makes the  $b^{\text{th}}$  bit ( $b$  ranges from 0 to 7) of register  $f$  equal to 0.

status flags affected: none

[bset b f] \*\*\*

**bit set** - makes the  $b^{\text{th}}$  bit ( $b$  ranges from 0 to 7) of register  $f$  equal to 1.

status flags affected: none

[ldan x] \*\*\*

**load accumulator with number** - The number  $x$  is loaded into the accumulator.

status flags affected: none

[lda f] \*\*\*

**load accumulator** - The contents of register  $f$  are loaded into the accumulator. (The value of register  $f$  remains unchanged.)

status flags affected: Z

[clr f] \*\*

**clear** - makes the contents of register  $f$  to 0.  
status flags affected: Z

[clra] \*\*

**clear accumulator** - makes the contents of the accumulator equal to 0.  
status flags affected: Z

[sta f] \*\*\*

**store accumulator** - The contents of the accumulator are loaded into register  $f$ . (The value of the accumulator remains unchanged.)  
status flags affected: Z

[lcom f] \*

**load and complement** - complement the contents of register  $f$  and store the result in the accumulator  
status flags affected Z

[ldec f] \*

**load and decrement** - decrement the contents of register  $f$  and store the result in the accumulator  
status flags affected Z

[linc f] \*

**load and increment** - increment the contents of register  $f$  and store the result in the accumulator (Read as “load and increment,,)  
status flags affected: Z

[inc f] \*\*\*

**increment** - increment the contents of register  $f$  and store the result in register  $f$  (Read as “increment,,)  
status flags affected: Z

[lrol f] \*

**load and rotate left** - The contents of register  $f$  are rotated left through the carry flag and the results are stored in the accumulator  
status flags affected: C

[rol f] \*\*

**rotate left** - The contents of register  $f$  are rotated left through the carry flag

and the results are stored in register  $f$   
 status flags affected: C

[lror  $f$ ] \*

**load and rotate right** - The contents of register  $f$  are rotated right through the carry flag and the results are stored in the accumulator  
 status flags affected: C

[ror  $f$ ] \*\*

**rotate right** - The contents of register  $f$  are rotated right through the carry flag and the results are stored in register  $f$   
 status flags affected: C

[lswap  $f$ ] \*

**load and swap** - The upper and lower nibbles of register  $f$  are swapped and the result is stored in the accumulator  
 status flags affected: none

[swap  $f$ ] \*

**swap** - The upper and lower nibbles of register  $f$  are swapped and the result is stored in register  $f$   
 status flags affected: none

### Program Flow

[ldecsz  $f$ ] \*

**load and decrement, skip if zero** - decrement the contents of register  $f$  and store the result in the accumulator. If the result is zero, skip the next instruction.  
 status flags affected: none

[decsz  $f$ ] \*\*\*

**decrement, skip if zero** - decrement the contents of register  $f$  and store the result in register  $f$ . If the result is zero, skip the next instruction.  
 status flags affected: none

[lincsz  $f$ ] \*

**load and increment, skip if zero** - increment the contents of register  $f$  and store the result in the accumulator. If the result is zero, skip the next instruction.  
 (Read as “load and increment, skip if zero”)  
 status flags affected: none

[incsz f] \*\*

**increment, skip if zero** - increment the contents of register  $f$  and store the result in register  $f$ . If the result is zero, skip the next instruction.

status flags affected: none

[tst f] \*\*

**test** - Tests to see if register  $f$  equals zero, sets the Z flag if the contents of register  $f$  is zero.

[btsc b f] \*\*\*

**bit test, skip if clear** - Tests the  $b^{\text{th}}$  bit of register  $f$ . If that bit is equal to 0, then the next program instruction is skipped.

status flags affected: none

[btss b f] \*\*\*

**bit test, skip if set**. Tests the  $b^{\text{th}}$  bit of register  $f$ . If that bit is equal to 1, then the next program instruction is skipped.

status flags affected: none

[bsr address] \*\*\*

**branch to subroutine** Branch to subroutine located at the program memory location indicated by *address* (between 0 and 8191). The value of the program counter + 1 is “pushed on to the stack,,

[rts ] \*\*\*

**return from subroutine** - The “top of the stack is popped,, and loaded into the program counter. Thus the next instruction executed after the “rts,, is the one immediately following the most recent “bsr,, instruction.

[bra address] \*\*\*

**unconditional branch** - *Address* is a number ranging from 0 to 8191. The value of *address* is loaded into the “program counter,, so that the next instruction executed is the one located at program memory location *address*.

[rti] \*

**return from interrupt**. The “top of the stack is popped,, and loaded into the program counter. Thus the next instruction executed after the “rti,, is the one immediately following the last executed instruction in the main program.

[rtv x] \*

**return from subroutine with value x** The value  $x$  is loaded into the accumulator. The “top of the stack is popped,, and loaded into the program counter. Thus the next instruction executed after the “rts,, is the one immediately following the most recent “bsr,, instruction.

### Misc. (3)

[nop] \*\*

**no operation** - does nothing

[clrwdt] \*

**clear watchdog timer** - resets the watchdog timer

[sleep] \*

**sleep** - puts the processor “to sleep,, a mode where its power consumption is dramatically reduced. The processor cannot do any computations in this mode, but it can be awakened by various interrupts\.

Appendix: The Logo Virtual Machine

