

Physics 219 – Fall, 2005
LabNotes 13 – What’s Inside a LogoChip?

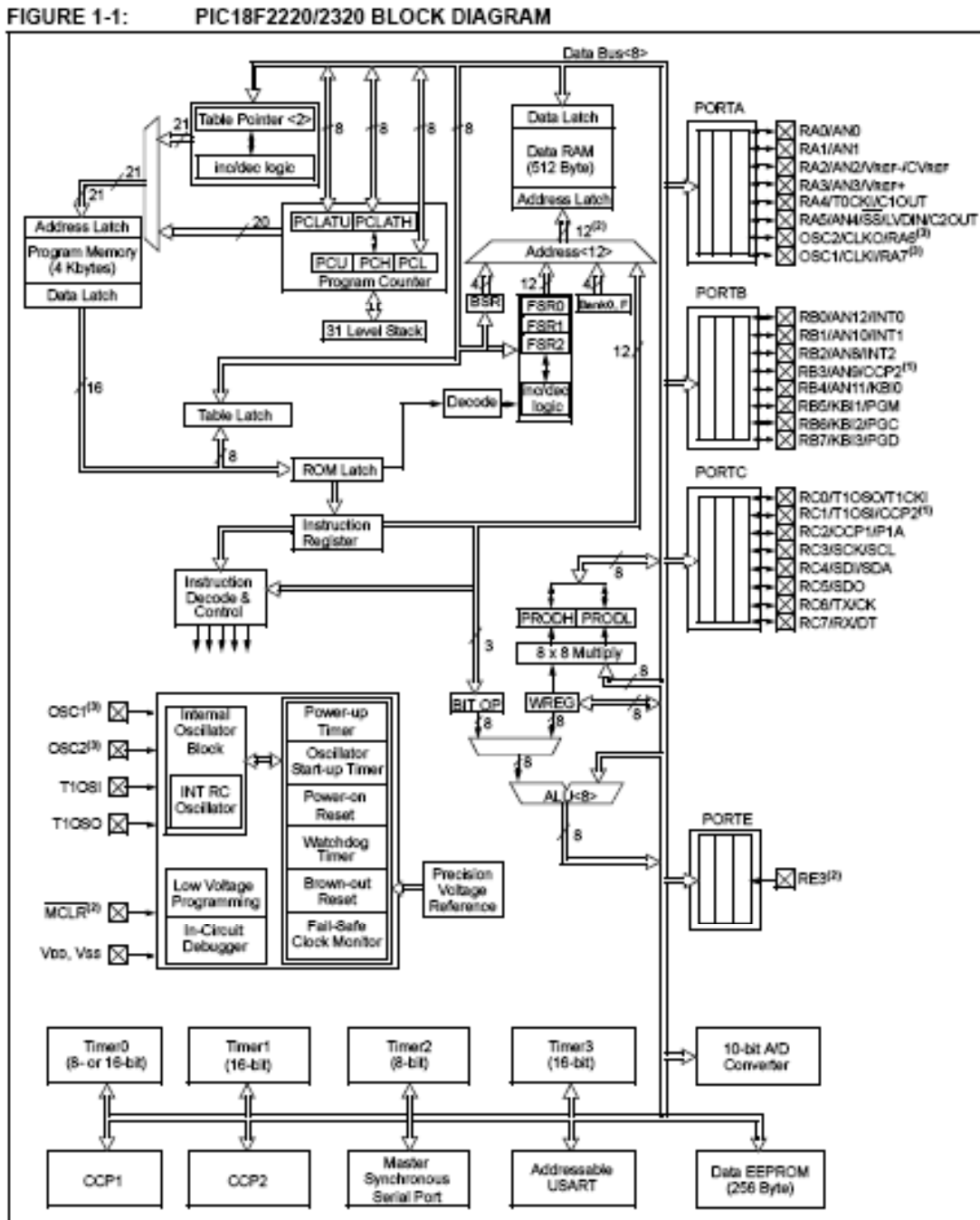
Table of Contents

Friday, December 9	2
PIC18F2320 microcontroller architecture	2
Registers.....	3
Using the special function registers	4
What’s inside a LogoChip pin?.....	8
Assembly Language vs. Logo.....	11
To Infinity and Beyond: The march towards increasing complexity and abstraction	13

Friday, December 9

PIC18F2320 microcontroller architecture

Going from logic gates and flip-flops to a microcontroller is a big leap. Consider the figure below which gives a schematic overview of the architecture of the PIC18F2320 microcontroller used to create a LogoChip. It's pretty daunting!



To really begin to understand the inner workings of a device as complicated as this is a course in itself. (CS 240, in fact.) In this lab we'll focus on just a few highlights.

Registers

In the last lab we saw a flip-flop could be used to store one bit of data. Much of the PIC18F2320 is arranged into “byte-wide” **registers**, each of which contains 8 flip flops and can therefore store 8 bits, or one byte, of data.

TABLE 5-1: SPECIAL FUNCTION REGISTER MAP FOR PIC18F2X20/4X20 DEVICES

Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDfH	INDF2 ⁽³⁾	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2 ⁽³⁾	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2 ⁽³⁾	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2 ⁽³⁾	FBCh	CCPR2H	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2 ⁽³⁾	FBBh	CCPR2L	F9Bh	OSCTUNE
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	PWM1CON ⁽²⁾	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	ECCPAS ⁽²⁾	F96h	TRISE ⁽²⁾
FF5h	TABLAT	FD5h	T0CON	FB5h	CVRCON	F95h	TRISD ⁽²⁾
FF4h	PRODH	FD4h	—	FB4h	CMCON	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	—	F90h	—
FEFh	INDF0 ⁽³⁾	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEeh	POSTINC0 ⁽³⁾	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC0 ⁽³⁾	FCDh	T1CON	FADh	TXREG	F8Dh	LATE ⁽²⁾
FECh	PREINC0 ⁽³⁾	FCCh	TMR2	FACH	TXSTA	F8Ch	LATD ⁽²⁾
FEbh	PLUSW0 ⁽³⁾	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	—	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—
FE7h	INDF1 ⁽³⁾	FC7h	SSPSTAT	FA7h	EECON2	F87h	—
FE6h	POSTINC1 ⁽³⁾	FC6h	SSPCON1	FA6h	EECON1	F86h	—
FE5h	POSTDEC1 ⁽³⁾	FC5h	SSPCON2	FA5h	—	F85h	—
FE4h	PREINC1 ⁽³⁾	FC4h	ADRESH	FA4h	—	F84h	PORTE ⁽²⁾
FE3h	PLUSW1 ⁽³⁾	FC3h	ADRESL	FA3h	—	F83h	PORTD ⁽²⁾
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	ADCON2	FA0h	PIE2	F80h	PORTA

Note 1: Unimplemented registers are read as '0'.
 Note 2: This register is not available on PIC18F2X20 devices.
 Note 3: This is not a physical register.

The above diagram shows how some of the PIC18F2320's registers are arranged. The registers each have a numerical **address**, (indicated by a 3-digit hexadecimal number). There is also a short name given to each of these registers, which tells us something about the function each one serves. In addition to the 128 "special purpose registers" shown above, the microcontroller also has about 512 unnamed "general purpose" registers (with address that run from \$0 to \$1ff, or 0 to 511) which serve as random access memory, or RAM, for the device. LogoChip Logo uses this RAM for its "stack operations" and to store global variables.

All of these registers can be set and read using the LogoChip Logo `read` and `write` primitives described below. For example:

`write register-address value` writes a one byte *value* in the register whose address is *register-address*.

Example:

```
write $f81 68           ; writes the number 68 into
                        the register whose address is
                        $f81 (the portb register)
```

(The name `portb` in Logo is just a constant that has been defined to have a value of `$f81`; try typing `print portb` in the LogoChip Logo command center and you'll see what I mean.)

These registers described above are all examples of **volatile memory**, meaning that they lose their contents every time power to the microcontroller is turned off. (This is exactly what you'd expect if you use flip-flops of the kind you saw in the last lab to store data.) The PIC18F2320 also has 8196 (8k) bytes of **non-volatile program memory**, which is used to store both the **LogoChip firmware** and **user programs**. The firmware is the program that creates a **Logo virtual machine**, turning a blank microcontroller into something that can process the "user programs" that you write in Logo.

Using the special function registers

The ability to directly write and read *all* of the microcontroller's registers is central to the LogoChip design philosophy, giving the user access to much of the microcontroller's functionality. However care must be exercised, since many of these registers are used by the Logo virtual machine so that altering these registers is likely to cause the Logo virtual machine to crash. Only a subset of the many

registers shown above should be written to by a LogoChip Logo program. A listing of some of the most commonly used registers and their addresses is shown in the table below.¹

register name	register address	register function
porta	\$f80	porta data register
porta-ddr	\$f92	porta data-direction register
portb	\$f81	portb data register
portb-ddr	\$f93	portb data-direction register
portc	\$f82	portc data register
portc-ddr	\$f94	portc data-direction register
portd *	\$f83	portd data register
portd-ddr *	\$f95	portd data-direction register
porte *	\$f84	porte data register
porte-ddr *	\$f96	porte data-direction register

There are however some additional powerful features that the microcontroller possesses which can be accessed via some of the special purpose registers. You've already seen one example when you made use of the microcontroller's ability to generate a pulse width modulation (pwm) signal. If you look back on the code you used then, you will see that it involved the use of 4 different special purpose registers:

¹ A complete register map for the PIC18F2320 microcontroller, along with an explicit listing of which registers can be written to a LogoChip Logo program without interfering with the operation of the virtual machine, is given in the *Register Map* section of the Appendix of the *LogoChip Logo Language Reference*.

```

; Pulse Width Modulation code. This program
; causes a square wave of user selected period and
; duty cycle to appear on pin C2. :val determines
; the duty cycle Values from 0 to 100 can be used.

constants [[ccplcon $fbd] [t2con $fca] [ccpr1l $fbe]
[pr2 $fcb]]

to initialize
clearbit 2 portc-ddr
write t2con 6
write pr2 100
write ccplcon 12
end

to pwm :val
write ccpr1l :val
end

```

Let me give you a second example of how to unlock another powerful hardware feature of the PIC18F2320 microcontroller. As you probably know, LogoChip Logo has a built-in timer primitive (called `timer`) that records time intervals in increments of one millisecond. That's often more than adequate. But the PIC18F2320 has a couple of hardware timers that are capable of far better time resolution. For example, the Logo code below shows how to use the microcontroller's "timer1 module" to define a Logo procedure, called `timer1`, that reads time in increments of 0.8 *microseconds* (assuming the microcontroller is operating at 40 MHz.)

```

constants [[t1con $fcd] [tmr1l $fce] [tmr1h $fcf]]

to initialize
write t1con $31
end

to resett1
write tmr1h 0
write tmr1l 0
end

```

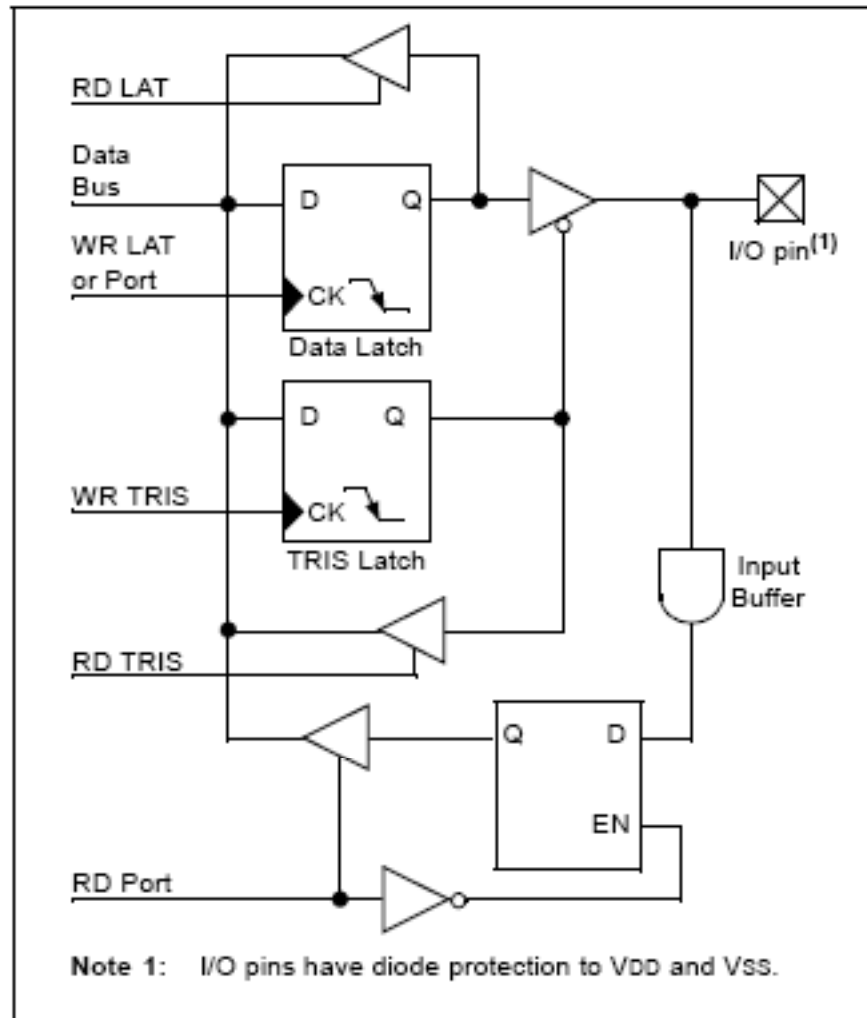
```
to timer1
output ((256 * (read tmr1h)) + read tmr1l)
end
```

In this code the `t1con` register is used to control whether the timer1 module is on and how fast it is running. (If you stare at the relevant page of the PIC18F2320 data sheet—page 123, you find that writing a value of \$31 to the `t1con` register sets a 16-bit timer to increment once every 32 clock cycles. The timer1 module can be configured to read in increments as small as $0.1 \mu s$.) The `tmr1h` and `tmr1l` registers contain the “high-byte” and the “low-byte” of the 16-bit number that corresponds to the elapsed time since the last time the `resett1` command occurred.

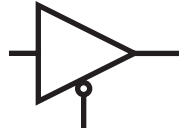
What's inside a LogoChip pin?

The schematic shown below that shows the scheme used by the PIC18F2320 microcontroller to implement its input / output pins. A lot of this schematic should be starting to look familiar and understandable to you now.

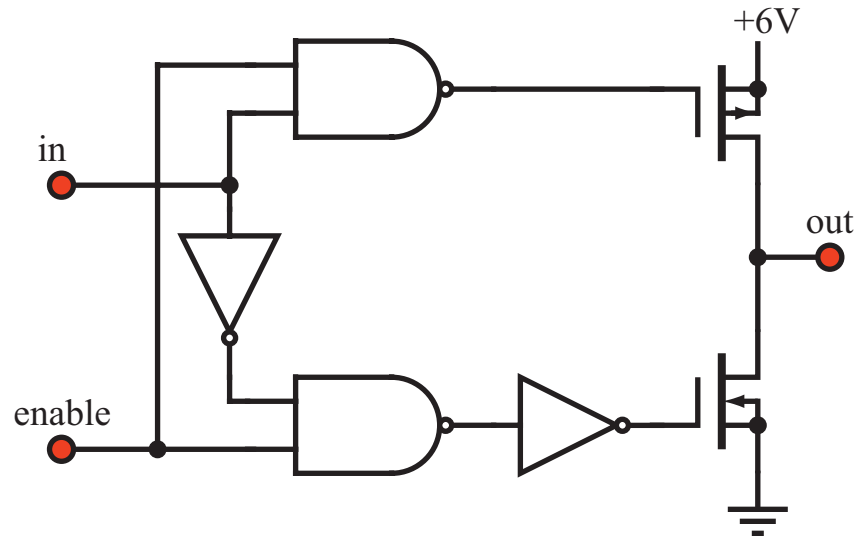
FIGURE 10-1: GENERIC I/O PORT OPERATION



Can you figure out how this logic works when the LogoChip executes a `setbit`, or a `clearbit`, or a `testbit` command directed at the corresponding bits of data register or the data direction register associated with the pin? Note that the symbol

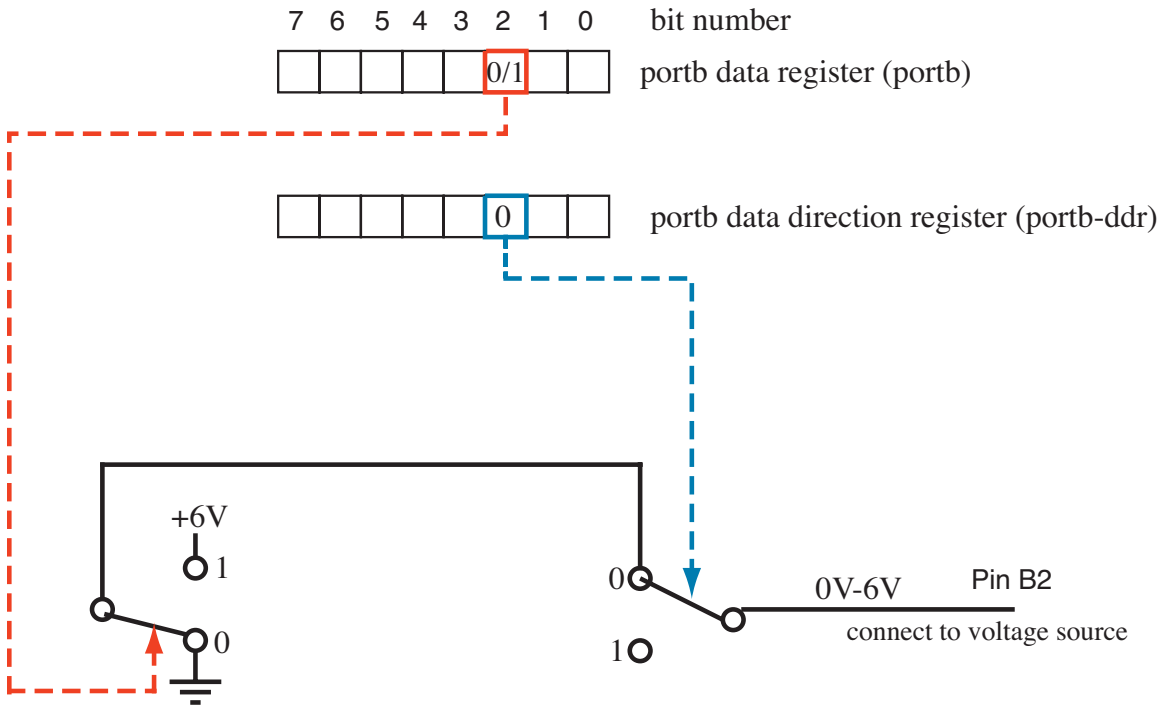


that appears in this drawing is short-hand for a CMOS three-state (or “tri-state”) “buffer” similar to the one that you saw in the combinational logic lab:



(although different in the sense that the tri-state buffer used by the microcontroller turns on when the ENABLE line goes LOW.) The data sheet calls “data-direction register “TRIS”, which stands for “tri-state”.

Looking back, can you appreciate how much fuller the description that these schematics give of what's going on inside a LogoChip pin compared to the cartoon version of events that you saw in LabNotes1?:



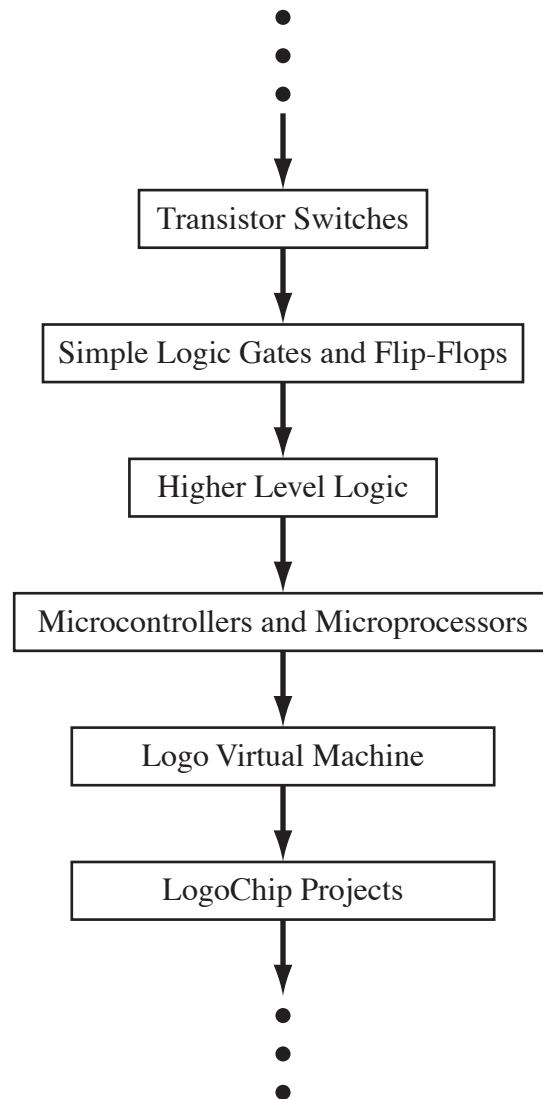
Pin B2 configured as an "output"

they can do each of these simple things very quickly (in about 100 ns in our case.) If you want a microcontroller to do things as quickly as is “machinely” possible, you generally want to program in **assembly language**, which makes direct use of these elemental capabilities. On the other hand, it is time consuming to write programs in assembly language that are capable of doing complex things. It's often *much* quicker (in terms of human time) to write the program using a “higher level” language like Logo. (The LogoChip firmware is written in assembly language. It's basic job is to insulate the LogoChip user from having to write programs in assembly language!)

The downside of higher level languages like LogoChip Logo 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.

	Logo	Assembly Language
Time per instruction	~15 μ sec	~100 nsec
High level primitives?	yes	no
Command Center?	yes	no

If you want to learn more about assembly language, you should take CS 240, Wellesley's “machine architecture” course.

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.