Lesson 1
The PIC Architecture

Overview

Introduction	This lesson gives an overview of the PIC microcontroller from a software perspective. Hardware implementation and issues are covered in other lessons.
In this section	Following is a list of topics in this section:

Architectural Overview

Introduction

Let me apologize in advance for this section. You may find this particular section especially dull. There is a bunch of background information to get to you, and here it is. In later sections we will actually *do* something, and you will likely find that a lot more fun.

This section is about a number of basic concepts. If there are some details that don't quite gel for you, don't get overly concerned. As we begin to poke at the PIC and do things with it, some of what we are talking about will become more clear.

The PIC is a self-contained, Harvard architecture microcontroller. It can be highly effective at replacing relatively complex discrete logic circuits. The PIC is not a microprocessor in the sense of a Pentium or 80386. The PIC is targeted at embedded applications where there is a fixed program which is relatively simple.

The PIC is a RISC (**R**educed **I**nstruction **S**et **C**omputer) processor. It has only 35 instructions. While this means that it may take more instructions to accomplish a simple task than a Pentium, which has hundreds of instructions, it also means that there are only 35 instructions to learn. This makes the PIC simple to apply.

Harvard vs. von Neumann Architecture

The traditional microprocessor is what is called a von Neumann architecture. The significant characteristic is that there is a single memory. Programs and data are stored in the same memory, which makes it possible for a program to load another program into memory. This is a key characteristic of what we generally think of as a computer. In most practical computers, some portion of the address space is implemented in ROM so that the computer has some sense when it is first powered up, but from the point of view of a program, it's all the same.

In contrast, a Harvard architecture machine has multiple memories. In the case of the PIC there are three; the program memory, the file register, and the EEPROM. The EEPROM is almost like an I/O device, although it is implemented on-chip. We won't talk about the EEPROM much until later.

One of the issues with von Neumann machines is that the instructions and data generally need to be different sizes. A lot of data is character data, which is typically 8 bits long. To be effective, a processor needs to be able to fetch a character from memory, so the memory wants to be organized in 8 bit pieces. However, if the instructions were only 8 bits, the processor would be awfully limited. The designer would be faced with, for example, allocating 2 bits to the instruction and 6 bits to the address of the operand. This would allow a machine with 4 instructions which could address 64 bytes of memory – not a pleasant prospect. The way around it is to make the instructions multiple bytes. This slows things down a lot, and adds a lot of complexity to the chip, but it's the price of the flexibility the von Neumann architecture affords.

Harvard machines don't have that limitation. Since the program is stored in its own memory, the program memory doesn't have to be friendly to storing data. There is no reason to make multiple memories the same width, and in most Harvard machines they aren't. In the case of the PIC, there are three series of PICs with 12, 14, and 16 bit wide program memories. We will largely concentrate on the 14 bit parts.

Continued on next page

Architectural Overview, Continued

Major Addressable Parts of the PIC	The PIC, as we mentioned, includes a program memory. In the parts we will talk about, this is a 14 bit wide memory. In the 16F84A, which we will use for our experiments, there are 1024 program memory locations. The program memory is either CMOS PROM or FLASH. The 'F' parts include FLASH memory for program storage. The PIC also includes a "file register", which is where data is stored. The file register is volatile, that is, its contents are random on power up. The file register is 8 bits wide. In the 16F84A, there are 68 memory locations in the file register. Some parts also include EEPROM. This is a non-volatile memory where a program can keep parameter data. In the 16F84A, this memory is 64 bytes long. Each of the PIC parts has a complement of other registers which control various devices. These may include simple I/O pins, counters, timers and A/D converters. The number and type of these devices varies with the part. In the case of the 16F84A there are 13 bidirectional I/O pins, one of which can be used as a counter or timer.

The File Register

Introduction	The file register is where temporary data is stored. Each file register location is 8 bits wide. The file register is volatile, which means that its contents are random when power is applied.
Addressing	Every memory cell, whether file register, program memory, or EEPROM, has an address. Typically, these addresses are expressed in hexadecimal. Hexadecimal numbers have digits that range from 0 through F (A comes after 9), so there are 16 possible values for each digit. The addresses of the file register run from H'0c' to H'4f'. (In the assembler, hexadecimal numbers are represented as strings with the letter H in front of them). Notice that we could use decimal numbers, or even binary or octal numbers, but traditionally the memory addresses in the PIC are referenced as hexadecimal numbers. Locations H'00' through H'0b' can be used to address the special function registers. Some instructions use these registers specifically, but any instruction which affects the file register memory can also be used on the special function registers by referencing an address less than H'0c'. Typically, we provide a name for file register locations to make our program easier to read. We cannot assign initial values to locations in the file register since their contents are indeterminate at power up.

Special Function Registers

Introduction	Much of the work of the PIC gets accomplished in a few special purpose registers. These registers can be addressed as the first few locations in the file register address space, but they each have a special capability that most of the file register locations don't share.
The Working Register	The most commonly used register is the working register, or **W** register. Although many functions can be performed on any file register location, operations that take two operands almost always involve the W register. Unlike the other special purpose registers, the W register is not mapped into the file register space.
The Status Register	Very often we would like to be able to test some consequence of an operation. For example, following an addition, we might like to know whether a carry occurred. If we decremented the contents of a memory cell, it may be useful to know whether the decrement caused the contents to reach zero. Results of these kinds of operations are recorded in the status register. We will talk about the status register in some detail later on.
PORTA and PORTB	These 2 locations look like ordinary file register locations, but they are connected to the outside world. Each bit in PORTA and PORTB is connected to a pin on the chip. Depending on whether we have set the pin to be an input or an output, the contents of the location will either set the pin high or low, or reflect the voltage on the pin. Depending on the particular PIC model, all of the bits may or may not be implemented, and they may be shared with other functions. In the case of the 16F84, all of the PORTB pins are implemented, but only 5 of the PORTA pins are implemented, and one of those is shared with the counter/timer.
TRISA and TRISB	These registers determine whether the corresponding PORTA and PORTB pins will be inputs or outputs. For example, if I set bit one of TRISB to a 0, then bit one of PORTB will be an output. If I now store a 1 into bit one of PORTB, the corresponding pin on the chip (pin 7) will go high. Similarly, if I set bit two of TRISB to be a 1, then bit two of PORTB will follow the state of pin 8 on the chip.
TMR0	TMR0 is a special thing. We can put it to one of two uses. In one case, it's a timer. When we set it up to be a timer it keeps counting as our program executes. We can even set it up to tap us on the shoulder after a specific amount of time has passed. The other use is as a counter. We can set TMR0 up to count how many cycles pin 3 has been through. Better yet, it has a built in prescaler that allows us to count frequency up to 60 MHz.

Program Memory

Introduction	The program memory is, well, where the program is stored. There really isn't a lot to say about it. In the 16F84, the program memory is 14 bits wide by 1024 locations long. The program memory is nonvolatile memory, which means we can program it and it will retain its value even when power is removed. There are two series of PIC parts, those with an F in the part number and those with a C. The F parts, like the 16F84, have FLASH program memory. The C parts have CMOS ROM. The C parts are less expensive, but they can be programmed only once. The F parts can be programmed thousands of times.
Programming the Program memory	Unlike your PC, the PIC itself cannot change its program memory. This requires special circuitry. In the case of the 16F84, a supply of over 12 volts is required to enable programming, in addition to the normal 5 volt logic supply. Some other PIC parts can be programmed with 5 volts only, but currently, all of the PICs can be programmed with 12 volt programmers. However, only a few pins are required to program a PIC. Because of this, it is practical to integrate some of the program circuitry into the application circuit. This is called "in-circuit" programming. There may be some cases where it isn't realistic to provide the isolation needed to program in-circuit, but in most cases it only adds a handful of parts, and it is a great convenience when you are experimenting.

EEPROM

Introduction	The final type of memory in the PIC is Electrically Erasable PROM. This is data memory that the PIC itself can alter, and which retains its value even when the power is removed. EEPROM takes multiple steps to access, so we won't deal with it until later in the course. It is useful, though, to remember things like code speed or the last frequency set.

OK, what does this mean?

Introduction	So far, we've talked about a lot of dull stuff that may not mean very much. Let's take a minute to pull some of this together.
Power Up	When power is first applied to a computer, any computer, there is a lot of unknown stuff. In the case of the PIC, the contents of the data memory and registers are random. Fortunately, the program memory remembers what we put in it. One thing that is constant among all computers, though, on power up, the state of the program counter is known. In the case of the PIC, the program counter contains a zero on power up. There are lots of things that have to happen on power up with any computer. Fortunately, on the PIC, most of these are handled very smoothly and we can ignore them for most purposes.
Clock Ticks	The PIC has a clock, whose frequency can be set with a crystal, a resonator, or an RC circuit. Some PICs even have a clock on board, so you don't need to add any frequency controlling circuitry. As the clock relentlessly ticks along, the PIC uses it to do its thing. On the PIC, every instruction takes 4 cycles of the clock. That means that with a 20 MHz crystal, the PIC is going to execute 5 million instructions a second. Five million times a second the PIC will: • Examine the program counter • Fetch the contents of program memory pointed to by the program counter • Interpret the instruction • Do what it says • Increment the program counter That's basically all any computer does. It just relentlessly picks one instruction after another to execute. Instructions may tell it to load the working register with something, do some math, or store the result somewhere. Those I/O pins allow that "somewhere" to affect the outside world, that is, the circuit we want it to control.

Assembler

Introduction	The PIC, or any computer for that matter, simply stores combinations of bits. It gets tough looking at long strings of ones and zeroes, so we need some sort of help. An assembler is a program that takes representations of these numbers that are more readable to a human, and translates them into a form that the computer can understand. You may also have heard of a compiler. The main difference between an assembler and a compiler is that everything you do in the assembler translates one to one into something in the computer. One thing you do in a compiler may translate into hundreds of things in the computer. The advantage of a compiler is that, if you are doing something the compiler knows how to do, it is a lot simpler than an assembler. The advantage of an assembler is that you control precisely what the computer will do. When we are using the computer for an embedded controller, we are often very concerned about timing, and this is a huge advantage.
Number Representation	As we said, to the computer, any computer, the world is a series of ones and zeroes. To us humans, though, it's a real pain looking at these strings. In the assembler, we can, if we choose, represent a ten as B'00001010', but that's kind of tough to look at. In the assembler, we can represent a ten as a decimal number by putting D in front of it. So the value D'10' means exactly the same thing as B'00001010'. More often, we use hexadecimal representation for numbers, because decimal numbers don't map nicely into the bits. Hexadecimal numbers are like decimal numbers, except there are 16 values for each digit instead of 10. They run from 0 to F. Thus, 6 decimal would be represented as H'06' in hexadecimal. But 10 decimal would be represented as H'0a' in hex. There is quite a good discussion of number representations on the web at: http://vwop.port5.com/beginner/bhextut.html if this is a little unclear.

Continued on next page

Assembler, Continued

Machine Instructions

Within the PIC, or any computer for that matter, any number that is pointed to by the program counter is interpreted as an instruction. Generally, these instructions include an operator and an operand. Also, in general, different operators use different numbers of bits, thus leaving a different number of bits for the operand.

We could manually translate the operation we want into a number and store it in the machine's program memory, but this is a big pain. The assembler deals with all this stuff for us.

Let's assume that we want to place the number 4 into the working register. We could refer to the programming card and recognize that we need to put:

```
11000000000100
```

Into the program memory to cause this to happen. It's a lot easier, though, to say:

```
MOVLW H'04'
```

The `MOVLW` tells the assembler that we want to move a literal constant into the W register. The `H'04'` tells it that the literal constant we want to move is a 4. The assembler figures out that the left 6 bits being `110000` tells the chip to load the right 8 bits into the W register.

Symbols

OK, so the assembler helps us with the instructions, and it allows us to represent numbers in a variety of ways. Perhaps the most important feature, though, is that it allows us to associate symbols with values.

Imagine, for example, that we were building a keyer. We decide to store the current code speed in location H'3b' in the file register. Now, if we wanted to load the current code speed into the working register, we could tell the assembler:

```
MOVF H'3b',0
```

The zero tells the assembler that the target of the move is the W register. However, we could define the symbol CodeSpd to mean 03bh. Now we can write:

```
MOVF CodeSpd,W
```

We not only don't need to be constantly remembering where we put things, but later on, when we come back to look at this program, it will make a lot more sense.

Wrap Up

Summary	In this lesson we have gotten exposed to some of the basic concepts of microcontrollers in general, and the Microchip PIC in particular. At this point, these concepts are pretty theoretical, but they will become clearer as we begin to use them in our programs. For some of you, this has been pretty old news. For others, some of it may be confusing, and even a bit strange. There is no need to feel like you need to memorize everything in this lesson. It is sufficient that you have seen it, so that it can gel as we practice.
Coming Up	In the next lesson we will download and install Microchip's Integrated Development Environment or IDE. This program will become our primary tool, and will be used heavily in the following lessons.